

# Master vs Replica Database

## 1 What is a Master Database? (Purpose)

The **master database** is the **main database** where **all write operations** happen.

Its core responsibility is:

- Handling **INSERT**
- Handling **UPDATE**
- Handling **DELETE**

It is the **source of truth** for the data.

In most systems, **only the master database is allowed to modify data**.

---

## 2 What is a Replica Database? (Purpose)

A **replica database** is a **read-only copy** of the master database.

Its main purpose is:

- Handling **read operations (SELECT)**
- Reducing load on the master
- Improving read performance

The replica keeps syncing data from the master using **replication mechanisms**.

---

## 3 Why Do We Need Master–Replica Setup?

In real software systems:

- Writes are **less frequent**
- Reads are **very frequent**

If everything hits one database:

- Performance degrades
- Database becomes a bottleneck

So we separate responsibilities:

- **Master** → writes
- **Replica** → reads

## What Happens Behind the Scenes? (Important)

When data is written:

1. Master updates its data
2. Changes are sent to replica
3. Replica applies changes

⚠ Replication is usually **not instant** (called **replication lag**).

That means:

- A read immediately after a write **might not reflect new data**

### Connection Pool:

## What you got right

Yes:

- A **connection pool** is a set of database connections
- These connections are **created and managed once**
- When a request needs DB access → it **borrow**s a connection
- When done → the connection is **returned to the pool**
- Connections are **reused**, not destroyed

So this mental model is correct 🙌

Request → get connection → run query → return connection

---

## ❌ Small Correction (Very Important)

“No other connections can be built except of pools”

This part is **not strictly true**.

### Correct version:

- **Connection pool limits how many connections SHOULD exist**
- But technically:
  - You *can* create connections outside the pool

- You *should not* in a well-designed backend system
- 

## Correct & Professional Statement (Use This)

A connection pool manages a limited number of database connections and reuses them for incoming requests. New connections are created only by the pool, and requests must wait if all connections are in use.

---

## What Actually Happens in Practice

### Pool Rules:

- Pool has a **maximum size** (e.g. 10 connections)
- Pool creates connections **lazily or at startup**
- Requests must **borrow** from the pool

### If all connections are busy:

- Request **waits**
- OR times out
- OR fails (depending on config)

No unlimited connection creation.

---

## Why This Rule Exists (System Design Reason)

Databases have:

- Limited connection capacity
- Memory and CPU cost per connection

So pooling:

- Protects the database
- Controls concurrency
- Prevents crashes

# SQLAlchemy Engine (Built on Connection Pool Concept)

## 1 What is SQLAlchemy Engine? (Main Purpose)

The **SQLAlchemy Engine** is the **central object** that:

- Manages **database connections**
- Uses a **connection pool internally**
- Executes SQL queries
- Acts as the **bridge between Python code and the database**

👉 You never talk to the database directly

👉 You talk to the Engine

---

## 2 How Engine Relates to Connection Pool (Key Idea)

When you create an engine:

```
engine = create_engine("postgresql://user:pass@localhost/db")
```

What actually happens:

- SQLAlchemy creates a **connection pool**
- Connections are created **lazily**
- Engine becomes the **manager of that pool**

So:

**Engine = Pool + Dialect + DBAPI**

---

## 3 What the Engine Does Internally

Behind the scenes:

- Keeps a **pool of open connections**
- Gives a connection when needed
- Takes it back when done
- Prevents unlimited connections

You **never see the pool**, but it's always there.

What happens internally:

1. engine.connect() → asks pool for a connection
2. Query runs
3. with block ends → connection goes back to pool

No manual open/close.

## Engine with Explicit Pool Configuration

```
engine = create_engine(  
    "postgresql://user:pass@localhost/db",  
    pool_size=5,  
    max_overflow=10,  
    pool_timeout=30  
)
```

Meaning:

- pool\_size=5 → always keep 5 connections
- max\_overflow=10 → allow up to 10 extra temporarily
- pool\_timeout=30 → wait 30 seconds if pool is full

This is **exactly your pool concept**, now formalized.

## Session Factory (SQLAlchemy)

### What is a Session Factory? (Main Purpose)

A **session factory** is an object that **creates Session instances** when needed.

It does **not** talk to the database itself.

Its only job is:

 **“Whenever the application needs to interact with the database, create a new Session configured with the Engine.”**

Think of it as a **Session generator**, not a Session.

---

## 2 Why Session Factory Exists (Very Important)

In a backend application:

- Multiple requests happen at the same time
- Each request must have its **own session**
- Sessions **must not be shared** across requests

So instead of manually creating sessions:

- We define **one factory**
  - Factory creates **independent sessions**
- 

## 3 How Session Factory is Created (Python)

```
from sqlalchemy.orm import sessionmaker  
  
SessionLocal = sessionmaker(bind=engine)
```

Here:

- `SessionLocal` is the **session factory**
  - `engine` provides access to the connection pool
- 

## 4 How Session Factory is Used

```
session = SessionLocal()  
  
users = session.query(User).all()  
  
session.close()
```

Flow:

1. Factory creates a new session
  2. Session borrows a connection from Engine
  3. Queries execute
  4. Session closes → connection returned to pool
- 

## 5 How Session Factory Fits in the Big Picture

Request  
↓  
Session (created by factory)  
↓  
Engine  
↓  
Connection Pool  
↓  
Database

The **factory** sits above the **Engine**, not below it.

---

## 6 Session Factory vs Session (Critical Difference)

Session Factory	Session
Creates sessions	Executes queries
One per application	One per request
Stateless	Stateful
Long-lived	Short-lived

## Clean Explanation (Professional Version)

The SQLAlchemy Engine is created once at application startup.  
When the engine is created, it also initializes a connection pool.  
This pool persists for the lifetime of the application.

A session factory is then bound to this engine.

For every request:

- A new Session is created by the session factory
- The Session uses the Engine
- The Engine provides a connection from the existing pool
- After the request finishes, the Session is closed
- The connection is returned to the pool, not destroyed

So:

**Engine & Pool = long-lived**  
**Session = short-lived (per request)**

---

## Visual Flow (Board-Friendly)

```
App Startup:
  Engine created
    ↓
  Connection Pool created
```

```
Runtime:
Request 1 → Session → Engine → Pool → DB → return connection
Request 2 → Session → Engine → Pool → DB → return connection
Request 3 → Session → Engine → Pool → DB → return connection
```

---

## Key Sentence (Simple & Accurate)

**The engine creates and owns the connection pool once, and all sessions reuse that same pool through the engine.**

---

## Even Simpler Version (Beginner Language)



The engine is created once.

The connection pool is created once.

Every request only creates a new session, not new connections.

Sessions reuse the same pool through the engine.

---

## Python Structure (Concept Mapping)

```
# created once
engine = create_engine(DB_URL)

# created once
SessionLocal = sessionmaker(bind=engine)

# per request
session = SessionLocal()
```

Only **session** is per request.

Engine and pool are **global/shared**.

---

## Interview-Grade One-Liner

**The engine initializes the connection pool once at startup, and all sessions created per request reuse that pool via the engine.**

---

## If Someone Tries to Trap You 🐼

**Q:** “Does every session create a new connection pool?”

**Answer:**

No. The pool is created by the engine once. Sessions only borrow connections from the existing pool.

# Multi-Tenancy

## 1 What is Multi-Tenancy? (Main Purpose)

**Multi-tenancy** is an architecture where **one application instance serves multiple independent customers (tenants)**, while **keeping each tenant's data isolated**.

A *tenant* can be:

- A company
- An organization
- A team
- A client account

**Main purpose:**

👉 **Serve multiple tenants using shared infrastructure while isolating data.**

---

## 2 Why Multi-Tenancy Exists

Without multi-tenancy:

- You would deploy **one application per customer**
- Separate databases, servers, configs
- High cost, hard maintenance

With multi-tenancy:

- One backend
  - One codebase
  - Controlled data separation
- 

## 3 Core Rule of Multi-Tenancy (Must Remember)

**Tenants share the application but not each other's data.**

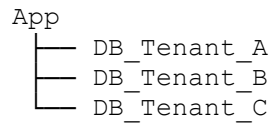
Isolation is **logical**, not accidental.

---

## 4 Common Multi-Tenancy Models (Software-Only)

## ◆ 1. Database-per-Tenant

Each tenant has its **own database**.

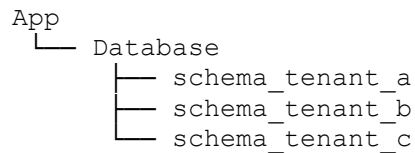


**Use when:**

- Strong isolation required
  - Enterprise clients
  - Custom scaling per tenant
- 

## ◆ 2. Schema-per-Tenant

One database, multiple schemas.



**Use when:**

- Medium isolation
  - Controlled environment
  - PostgreSQL systems
- 

## ◆ 3. Table-level (Shared Tables)

All tenants share tables, data separated by `tenant_id`.

```
users
-----
id | name | tenant_id
```

**Use when:**

- Large number of tenants
- SaaS products

- Cost efficiency matters
- 

## 5 Python Example (Table-Level Multi-Tenancy)

```
def get_users(session, tenant_id):  
    return session.query(User).filter(  
        User.tenant_id == tenant_id  
    ).all()
```

Here:

- Same table
  - Same database
  - Data separated by `tenant_id`
- 

## 6 How Multi-Tenancy Works with SQLAlchemy (Important)

At request start:

1. Identify tenant (from token, subdomain, header)
2. Configure session/query to use tenant context
3. Ensure **every query is tenant-scoped**

Example:

```
tenant_id = request.tenant_id  
session.query(Order).filter(Order.tenant_id == tenant_id)
```

---

## 7 Multi-Tenancy with Multiple Databases (Advanced)

```
engine = create_engine(DB_URL_TENANT_A)  
SessionLocal = sessionmaker(bind=engine)
```

Tenant decides:

- Which engine
- Which database

Engine & pool still follow the same rules you learned earlier.

---

## 8 Multi-Tenancy + Connection Pool (Key Insight)

- Engine created per database
  - Each engine has its **own pool**
  - Sessions reuse pools
  - Tenant routing happens **before session creation**
- 

## 9 Use Cases in Software Engineering

- SaaS platforms
- CRM systems
- HR management systems
- Analytics dashboards
- ERP systems
- Multi-company admin panels

Basically:

👉 **Any product serving multiple organizations**

---

## 10 Advantages

- ✓ Cost efficient
  - ✓ Single deployment
  - ✓ Easy updates
  - ✓ Scalable architecture
- 

## 1 1 Challenges (Must Mention)

- ✗ Data isolation bugs
- ✗ Security risks if tenant filtering fails
- ✗ Complex queries
- ✗ Migration complexity

## 1 2 Interview-Ready Definition

**Multi-tenancy is an architectural approach where a single application serves multiple tenants while keeping their data logically isolated.**

## What this code is (one line)

This is an async database connection manager that sets up master–replica PostgreSQL engines, connection pools, and separate session factories for read and write operations.

---

## High-Level What It Does

1. Creates **one async SQLAlchemy engine for master**
2. Optionally creates **one async engine for replica**
3. Each engine **creates its own connection pool once**
4. Creates **two session factories**:
  - Write sessions → master
  - Read sessions → replica (fallback to master)
5. Provides a **clean API** to get the correct session per request
6. Handles **startup, shutdown, and health checks**