🌿 Complete Guide to Learning SQLAlchemy 🌿

📚 Introduction

In this guide, based on the **official SQLAlchemy documentation** and with the help of **various artificial intelligences**, I try to examine the applications and important details of this library.
The goal is to review concepts not only in the context of **Python**, but also in relation to **different languages and frameworks** so that learning has a more practical and comprehensive aspect.

🗂️ Table of Contents

- Relationships
  - One-to-Many
  - Many-to-One
  - One-to-One
  - Many-to-Many
  - Back references
- Lazy Loading vs Eager Loading
  - Lazy loading patterns
  - Eager loading (joinedload, selectinload)
  - N+1 problem
  - Loading strategies
- Advanced Querying
  - Complex joins
  - Subqueries
  - Union operations
  - Window functions
  - Raw SQL integration

Advanced Topics

- Session Management Patterns
  - Session per request
  - Contextual sessions
  - Thread-local sessions
  - Session events
- Advanced Relationships
  - Self-referential relationships
  - Polymorphic relationships
  - Hybrid properties
- Customization and Extensions
  - Custom types
  - Validators
  - Events and listeners
  - Mixins
  - Custom loading techniques

Performance and Optimization

- Performance Tuning
  - Query optimization
  - Index strategies
  - Connection pooling
  - Bulk operations
- Caching
  - Query result caching
  - Second-level cache
  - Dogpile.cache integration

Advanced Features

- Migrations with Alembic
  - Migration basics
  - Auto-generating migrations
  - Manual migrations
  - Branching and merging
- Testing
  - Testing patterns
  - Fixtures
  - Mock strategies
  - Database testing

Real-world Applications

- Design Patterns
  - Repository pattern
  - Unit of Work
  - Data Mapper
  - Active Record considerations
- Integration
  - Web framework integration (Flask, FastAPI)
  - Async SQLAlchemy
  - Multi-database setups
  - Sharding strategies

Basics and Fundamentals

Introduction to SQLAlchemy

What is SQLAlchemy?

SQLAlchemy is one of the most powerful and popular **Python** libraries for working with databases.

This library provides two main approaches:

- **SQL Toolkit** ← For writing and executing SQL queries with more features and lower level.
- **ORM (Object Relational Mapper)** ← For working with databases through **Python objects** instead of writing raw SQL directly.

Advantages and Applications

1. High Abstraction

```
# Instead of writing raw SQL:
cursor.execute("SELECT * FROM users WHERE age > 18")

# You can write:
users = session.query(User).filter(User.age > 18).all()
```

2. Support for Multiple Databases

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Microsoft SQL Server
- and many others

3. Type Safety and IntelliSense

- **Type Safety**: When you're writing code, the computer helps ensure data types are correct and prevents common mistakes
- **IntelliSense**: This is an IDE feature that automatically shows suggestions, auto-completion, and brief documentation while you're coding

4. Migration Management

Database change management when you modify models.

Installation and Setup

Basic Installation

To install SQLAlchemy in basic form:

```
pip install sqlalchemy
```

To install with **async** support:

```
pip install "sqlalchemy[asyncio]"
```

Installing Database Drivers

- **PostgreSQL**:

```
pip install psycopg2-binary
```

- **MySQL**:

```
pip install pymysql
```

- **Oracle**:

```
pip install cx_Oracle
```

Installation for Different Frameworks

- **Flask**:

```
pip install flask flask-sqlalchemy flask-migrate
```

- **FastAPI**:

```
pip install fastapi sqlalchemy alembic uvicorn
```

- **Django**: Django has a built-in ORM by default, but you can also use SQLAlchemy if needed:

```
pip install django sqlalchemy
```

Installation Test

To ensure correct installation, run this code:

```
import sqlalchemy
print(sqlalchemy.__version__)  # Should display the installed version
```

Core vs ORM

execute is a method that runs an SQL command (like select, insert, update, delete) on the database.

SQLAlchemy provides two main approaches:

SQLAlchemy Core (Low Level)

Core is for those who want more control over SQL.

```python
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String, sel

# Table definition
metadata = MetaData()
users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('email', String(100))
)

# Connection and table creation
engine = create_engine("sqlite:///example.db")
metadata.create_all(engine)

# Query
with engine.connect() as conn:
    # Insert
    conn.execute(users.insert().values(name='Ahmad', email='ahmad@example.com'))

    # Select
    result = conn.execute(select(users).where(users.c.name == 'Ahmad'))
    for row in result:
        print(f"ID: {row.id}, Name: {row.name}")
```

Core Advantages:

- Higher speed
- Complete control over SQL
- Suitable for complex queries
- Lower memory usage

SQLAlchemy ORM (High Level)

ORM is for developers who want to work with Python objects.

```python
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

Base = declarative_base()

# Model definition
class User(Base):
```

```
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    email = Column(String(100))

    def __repr__(self):
        return f"<User(name='{self.name}', email='{self.email}')>"

# Setup
engine = create_engine("sqlite:///example.db")
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
session = Session()

# Usage
# Insert
new_user = User(name='Ahmad', email='ahmad@example.com')
session.add(new_user)
session.commit()

# Select
users = session.query(User).filter(User.name == 'Ahmad').all()
for user in users:
    print(user)
```

ORM Advantages:

- More readable code
- Type safety
- Easy relationship handling
- Less SQL writing

Database Connection

Engine and Connection

What is **Engine**? **Engine** in SQLAlchemy acts like a database connection manager.

- Created once and used throughout the application.
- Responsible for managing connections and executing SQL commands.

Creating Engine

```
from sqlalchemy import create_engine

# SQLite
engine = create_engine("sqlite:///myapp.db")

# PostgreSQL
engine = create_engine("postgresql://user:password@localhost/dbname")
```

Using Connection

Method 1: With Context Manager (Best Practice)

```
with engine.connect() as conn:
    result = conn.execute("SELECT 1")
    print(result.fetchone())
```

✅ Advantage: After the block ends, the connection is automatically closed.

Method 2: Manual (You must close it yourself)

```
conn = engine.connect()
result = conn.execute("SELECT 1")
print(result.fetchone())
conn.close()  # Make sure to close the connection
```

Example in Flask

```
from flask import Flask
from sqlalchemy import create_engine

app = Flask(__name__)
engine = create_engine("sqlite:///app.db")

@app.route("/test")
```

```python
def test_db():
    with engine.connect() as conn:
        result = conn.execute("SELECT COUNT(*) FROM users")
        return f"Number of users: {result.scalar()}"
```

Connection Strings

Connection String is your database address:

Different types:

```python
# SQLite
"sqlite:///path/to/database.db"
"sqlite:///C:/path/to/database.db"  # Windows
"sqlite:///:memory:"  # In memory

# PostgreSQL
"postgresql://username:password@localhost:5432/dbname"
"postgresql+psycopg2://user:pass@localhost/dbname"

# MySQL
"mysql://username:password@localhost:3306/dbname"
"mysql+pymysql://user:pass@localhost/dbname"

# SQL Server
"mssql+pyodbc://user:pass@server/dbname?driver=ODBC+Driver+17+for+SQL+Server"
```

Connection Pooling

Pool means connection pool - keeps a number of connections ready.

```python
# Pool configuration
engine = create_engine(
    "postgresql://user:pass@localhost/db",
    pool_size=10,        # 10 connections
    max_overflow=20,     # max 20 additional
    pool_timeout=30,     # 30 seconds wait
    pool_recycle=3600    # reset connections every hour
)

# Check Pool status
print(f"Pool size: {engine.pool.size()}")
print(f"Active connections: {engine.pool.checked_in()}")
```

Database Events

In SQLAlchemy, Events are mechanisms that allow developers to listen to various database operation events and stages and react to them. This capability provides monitoring, logging, and applying custom changes at the connection, query, or transaction level.

```
# Before connection
@event.listens_for(engine, "connect")
def set_sqlite_pragma(dbapi_connection, connection_record):
    if "sqlite" in str(engine.url):
        cursor = dbapi_connection.cursor()
        cursor.execute("PRAGMA foreign_keys=ON")
        cursor.close()

# Before query execution
@event.listens_for(engine, "before_cursor_execute")
def receive_before_cursor_execute(conn, cursor, statement, parameters, context, exe
    print(f"Executing: {statement}")
```

SQLAlchemy Core

Table Definition and MetaData

- **MetaData** is a central object that stores and manages all information related to tables, columns, and database relationships.
- This means every table you define is usually connected to a **MetaData** so that later you can create, delete, or manage them all together.

Table objects

Table in SQLAlchemy is like a blueprint of your table.

```
from sqlalchemy import MetaData, Table, Column, Integer, String, DateTime
from datetime import datetime

metadata = MetaData()

# Define users table
users_table = Table('users', metadata,
```

```
    Column('id', Integer, primary_key=True),
    Column('name', String(50), nullable=False),
    Column('email', String(100), unique=True),
    Column('created_at', DateTime, default=datetime.utcnow)
)

# Create table
engine = create_engine("sqlite:///example.db")
metadata.create_all(engine)
```

Column types

Main Column types:

```
from sqlalchemy import Integer, String, Text, Boolean, DateTime, Float, Numeric

users = Table('users', metadata,
    # Numbers
    Column('id', Integer),
    Column('age', Integer),
    Column('salary', Float),
    Column('balance', Numeric(10, 2)),  # 10 digits, 2 decimal places

    # Text
    Column('name', String(50)),         # max 50 characters
    Column('bio', Text),                # long text

    # Date and time
    Column('created_at', DateTime),
    Column('birth_date', DateTime),

    # True/False
    Column('is_active', Boolean, default=True)
)
```

In FastAPI/Flask:

```
# Flask-SQLAlchemy
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), unique=True)
    is_active = db.Column(db.Boolean, default=True)
```

```
# FastAPI
from sqlalchemy import Column, Integer, String, Boolean
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    is_active = Column(Boolean, default=True)
```

Constraints

Types of Constraints:

```
from sqlalchemy import ForeignKey, CheckConstraint, UniqueConstraint

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50), nullable=False),
    Column('email', String(100), unique=True),
    Column('age', Integer),

    # Check constraint
    CheckConstraint('age >= 18', name='age_check'),

    # Unique constraint
    UniqueConstraint('email', 'name', name='unique_email_name')
)

# orders table with Foreign Key
orders = Table('orders', metadata,
    Column('id', Integer, primary_key=True),
    Column('user_id', Integer, ForeignKey('users.id')),
    Column('total', Float)
)
```

Indexes

Index for speeding up searches:

```
from sqlalchemy import Index

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('email', String(100)),
    Column('city', String(50)),
```

```
    Column('age', Integer),

    # Simple Index
    Index('idx_email', 'email'),

    # Composite Index
    Index('idx_city_age', 'city', 'age'),

    # Unique Index
    Index('idx_email_unique', 'email', unique=True)
)
```

Schema operations

Creation and deletion:

```
# Create all tables
metadata.create_all(engine)

# Create only one table
users_table.create(engine)

# Drop all tables
metadata.drop_all(engine)

# Drop one table
users_table.drop(engine)
```

SQL Expression Language

Let's assume we have a table called users_table with columns: id, name, email, age, is_active and orders_table with columns: id, user_id, total

Select statements

SELECT statements in SQLAlchemy are used to select and retrieve data from tables.

Select all rows

```
from sqlalchemy import select
```

```
stmt = select(users_table) # select all rows
with engine.connect() as conn:
    result = conn.execute(stmt)
    for row in result:
        print(f"Name: {row.name}, Email: {row.email}")
```

Select specific columns

```
stmt = select(users_table.c.name, users_table.c.email)
# c means column
```

With simple condition

```
stmt = select(users_table).where(users_table.c.age > 25)
```

Multiple conditions with AND

```
stmt = select(users_table).where(
    (users_table.c.age > 18) & (users_table.c.is_active == True)
)
```

LIKE condition

```
stmt = select(users_table).where(users_table.c.name.like('Ahmad%'))
```

Insert, Update, Delete

Insert – Adding data

```
# Insert one
stmt = users_table.insert().values(name='Ali', email='ali@test.com')
with engine.connect() as conn:
    result = conn.execute(stmt)
    print(f"New ID: {result.inserted_primary_key}")

# Insert multiple
stmt = users_table.insert()
with engine.connect() as conn:
    conn.execute(stmt, [
```

```
        {'name': 'Ahmad', 'email': 'ahmad@test.com'},
        {'name': 'Fateme', 'email': 'fateme@test.com'}
    ])
```

Update – Updating

```
stmt = users_table.update().where(users_table.c.id == 1).values(name='Ali Ahmadi')

with engine.connect() as conn:
    result = conn.execute(stmt)
    print(f"{result.rowcount} records updated")
```

Delete – Deletion

```
stmt = users_table.delete().where(users_table.c.age < 18)

with engine.connect() as conn:
    result = conn.execute(stmt)
    print(f"{result.rowcount} records deleted")
```

Joins

Joins are used to merge data from two or more tables based on a common condition.

1️⃣ Inner Join

Returns data only when rows from both tables match the join condition.

Practical example: Getting only users who have placed at least one order.

```
# Inner Join
stmt = select(users_table.c.name, orders_table.c.total).join(
    orders_table, users_table.c.id == orders_table.c.user_id
)
```

Simple concept:

You want to get user information and their order total.

Inner Join means only users who have at least one order are displayed.

The condition `users_table.c.id == orders_table.c.user_id` specifies which user connects to which order.

💡 **Result:** Users without orders are not displayed.

2️⃣ Outer Join (Left Join)

Returns all rows from the main table (left), even if there's no match in the second table.

Practical example: Getting all users, even those who haven't placed any orders yet; in this case, the second table (orders) columns have NULL values.

```
# Left (Outer) Join
stmt = select(users_table.c.name, orders_table.c.total).select_from(
    users_table.outerjoin(orders_table)
)
```

Simple concept:

Again, you want to get user information and their order total.

Left Outer Join means all users are displayed, even if they have no orders.

Order columns for users who have no orders will have NULL values.

💡 **Result:** No user is removed, even without orders.

🔹 **The .c concept**

.c means table columns.

When you write users_table.c.name it means the name column from the users_table.

Functions and Operators

Functions – SQL functions

Practical examples:

```
from sqlalchemy import select, func

# Count users
stmt = select(func.count(users_table.c.id))

# Maximum, minimum and average age of users
stmt = select(
    func.max(users_table.c.age),
    func.min(users_table.c.age),
    func.avg(users_table.c.age)
)
```

```
# Group by city and count users
stmt = select(
    users_table.c.city,
    func.count(users_table.c.id)
).group_by(users_table.c.city)
```

💡 **Note:**

- `func` is equivalent to SQL functions
- You can use aggregation functions like COUNT, MAX, MIN, AVG, SUM
- aggregation functions are used to combine or analyze data from tables.

Operators

Comparison:

```
users_table.c.age > 18
users_table.c.name == 'Ali'
users_table.c.email.like('%gmail%')
users_table.c.id.in_([1, 2, 3])
```

Logical operators:

```
# AND
(users_table.c.age > 18) & (users_table.c.is_active == True)

# OR
(users_table.c.city == 'Tehran') | (users_table.c.city == 'Isfahan')
```

💡 **Note:**

- `|` = OR
- `&` = AND
- `like()` is used for string pattern matching
- `in_()` checks membership in a list

Subqueries

Simple Subquery

```
# Users who have at least one order
subq = select(orders_table.c.user_id).distinct()
```

```
stmt = select(users_table).where(users_table.c.id.in_(subq))
```

Subquery with alias and scalar

```
subq = select(func.avg(users_table.c.age)).scalar_subquery()
stmt = select(users_table).where(users_table.c.age > subq)
```

EXISTS

```
from sqlalchemy import exists

stmt = select(users_table).where(
    exists().where(orders_table.c.user_id == users_table.c.id)
)
```

💡 **Note:**

- **Subquery** = inner query
- **scalar_subquery()** = returning a single value
- **exists()** = checking existence of at least one row

Executing Statements

Connection execution

Execution methods

```
from sqlalchemy import create_engine, text, select

engine = create_engine("sqlite:///example.db")

# Method 1: With Connection
with engine.connect() as conn:
    result = conn.execute(text("SELECT * FROM users"))
    print(result.fetchall())

# Method 2: Direct from Engine
result = engine.execute("SELECT COUNT(*) FROM users")
print(result.scalar())
```

```
# Method 3: With Statement Object (more modern and secure)
stmt = select(users_table).where(users_table.c.age > 25)
with engine.connect() as conn:
    result = conn.execute(stmt)
    for row in result:
        print(f"Name: {row.name}")
```

Parameterized Queries

```
with engine.connect() as conn:
    # One parameter
    result = conn.execute(
        text("SELECT * FROM users WHERE age > :min_age"),
        {"min_age": 18}
    )

    # Multiple parameters
    result = conn.execute(
        text("SELECT * FROM users WHERE age BETWEEN :min_age AND :max_age"),
        {"min_age": 18, "max_age": 65}
    )
```

💡 Advantage: Better security (prevents SQL Injection)

Result objects

Getting data

```
stmt = select(users_table)
with engine.connect() as conn:
    result = conn.execute(stmt)

    # One row
    first_row = result.fetchone()
    print(f"First user: {first_row.name}")

    # Multiple rows
    some_rows = result.fetchmany(5)

    # All rows
    all_rows = result.fetchall()
```

Transactions

📌 Transaction = a set of operations that either all execute or none do.

Simple (automatic)

```
with engine.connect() as conn:
    conn.execute(users_table.insert().values(name="Ahmad"))
    conn.execute(users_table.insert().values(name="Ali"))
    # At the end of with block → automatic commit
```

Explanation:

Here when with ends, SQLAlchemy had automatically opened a transaction and commits it.

If an error occurs during query execution, the transaction is automatically rolled back.

This means all queries executed inside the with block are either all saved or none.

✅ So this state is atomic.

Manual

```
conn = engine.connect()
trans = conn.begin()
try:
    conn.execute(users_table.insert().values(name="Ahmad"))
    conn.execute(users_table.insert().values(name="Ali"))
    trans.commit()  # Save changes
except:
    trans.rollback()  # Rollback
finally:
    conn.close()
```

Explanation:

Here you manually open a transaction (conn.begin()).

After executing queries, you must manually call commit() or rollback().

This method is needed when you want complete control over the transaction (for example, check a specific condition in the middle and decide whether to commit or rollback).

✅ This state is also atomic, because if it doesn't commit, all changes are rolled back.

🔹 Summary:

Connection ← for executing queries

Result ← for getting output and working with rows

Transaction ← for ensuring data integrity

SQLAlchemy ORM

Declarative Base

🔹 What is Declarative Base?

An object-oriented way to define tables.

Instead of manually creating Tables, you write a Python class and SQLAlchemy automatically maps it to a database table.

In SQLAlchemy we have two ways to define tables:

Manual method (Table-based): Using Table and columns to create tables.

Object-oriented method (Declarative Base): Using Python classes to define tables.

Model definition

Old method (SQLAlchemy < 2.0)

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

# Base
Base = declarative_base()

# Model definition
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    email = Column(String(100))
```

➡️ This code creates a table named users in the database. Each object from the User class is equivalent to a record (row) in the table.

New method (SQLAlchemy 2.0)

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column

class Base(DeclarativeBase):
    pass

class User(Base):
    __tablename__ = "users"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(50))
    email: Mapped[str] = mapped_column(String(100))
```

✔️ The difference is that in version 2.0, types (Mapped[int]) are defined more clearly and compatible with modern Python.

Table mapping

Table Mapping = connecting a Python class to a database table

- 🔹 Simple case (same names)

When the table and column names are the same as the class and attribute names, it's very simple:

```
class User(Base):
    __tablename__ = 'users'   # Database table
    id = Column(Integer, primary_key=True)  # id column
    name = Column(String(50))               # name column
```

Here the User class is connected to the users table.

The id attribute in Python connects exactly to the id column in the database.

The name attribute in Python connects exactly to the name column in the database.

📌 This means the names are the same ← automatic and direct mapping.

- 🔹 Custom case (different names)

Sometimes in the database the column or table names are something else, but you don't want to use those hard names in Python. Here custom Mapping helps:

```python
class User(Base):
    __tablename__ = 'user_accounts'   # Database table name is user_accounts

    user_id = Column("id", Integer, primary_key=True)     # id column → user_id att
    full_name = Column("name", String(100))               # name column → full_name
```

The actual database table name is user_accounts.

In the database we have a column named id, but in Python we named it user_id.

In the database we have a column named name, but in Python we use full_name.

📌 This way Python code becomes more readable, but still connects to the database.

Primary keys

Simple

```python
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)  # Auto increment
```

With UUID

```python
import uuid
from sqlalchemy.dialects.postgresql import UUID

class User(Base):
    __tablename__ = "users"
    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
```

Composite Key

```python
class OrderItem(Base):
    __tablename__ = "order_items"
    order_id = Column(Integer, primary_key=True)
    product_id = Column(Integer, primary_key=True)
    quantity = Column(Integer)
```

Column options

```
from sqlalchemy import Boolean, DateTime
from datetime import datetime

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String(50), nullable=False)   # Required
    email = Column(String(100), unique=True)    # Must be unique
    is_active = Column(Boolean, default=True)   # Default value
    created_at = Column(DateTime, default=datetime.utcnow)  # Creation time
    updated_at = Column(DateTime, onupdate=datetime.utcnow) # Update time
    city = Column(String(50), index=True)       # Index
    status = Column(String(20), server_default="active") # Server-side default
```

Sessions

🟢 What is Session?

A communication bridge between Python code and database.

This means all ORM operations (add, update, delete, read) are done through Session.

Session holds changes until you call commit() to save them in the database.

📌 So Session is like a temporary notebook where you can finally decide to save or discard changes.

Don't confuse this Session in SQLAlchemy with authentication Session; this will be explained later.

Session lifecycle

🔹 Session Lifecycle

Creating Session Factory

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
```

This means a "factory" for creating Sessions.

Creating Session

```
session = Session()
```

Using Session

```
user = User(name="Ahmad")
session.add(user)
```

Commit (save to database)

```
session.commit()
```

Closing Session

```
session.close()
```

Creating and configuring sessions

```
Session = sessionmaker(
    bind=engine,
    autoflush=False,        # Don't flush changes before query automatically
    autocommit=False,       # Don't commit automatically
    expire_on_commit=False  # Don't expire data after commit
)
```

📌 Usually frameworks set these configurations so you have more control.

Session states

```
session = Session()

# 1. Transient (disconnected)
user = User(name="Ahmad")

# 2. Pending (waiting)
session.add(user)
print(user in session.new)  # True
```

```
# 3. Persistent (stable)
session.commit()
print(user in session)  # True
print(user.id)  # Now has ID

# 4. Detached (separated)
session.expunge(user)  # Remove from session
print(user in session)  # False

# 5. Deleted (deleted)
session.delete(user)
print(user in session.deleted)  # True
```

Committing and rollback

Commit: Saving changes to database.

Rollback: Reverting all changes (if an error occurred).

Example:

```
session = Session()
try:
    user1 = User(name="Ahmad")
    user2 = User(name="Ali")
    session.add_all([user1, user2])
    session.commit()
except:
    session.rollback()
finally:
    session.close()
```

🔹 **Using Context Manager (recommended method)**

```
with Session() as session:
    user = User(name="Ahmad")
    session.add(user)
    session.commit()
```

📌 This way Session closes itself at the end.

Final notes: 1️⃣ Session in SQLAlchemy / ORM

Purpose: Managing database connection and transactions

Responsible for saving, editing, deleting, and reading data.

Controls whether changes are committed to database or rolled back.

Maintains object states: Transient, Pending, Persistent, Detached, Deleted

✅ Result: All ORM operations and queries are done through this session.

2️⃣ Session in frameworks (Flask, FastAPI, Django)

Purpose: Managing logged-in user and authentication

Usually when user logs in, information like user_id or token is stored in session.

This Session is not connected to database, but is a Context for storing information between requests.

In Flask/FastAPI it's usually implemented with cookie or server-side session store.

In Django, session is managed and information is stored in database or cache.

Session Type	**SQLAlchemy ORM**	**Framework (Flask/FastAPI/Django)**
Purpose	Managing database connection	Managing authentication / user info
Contains	Object states, transactions	user_id, token, login state
Lifetime	Short-term, usually per request	Longer, between requests
Key operations	add, commit, rollback, query	set/get/remove, check login
Storage	Python memory, database	cookie, database, cache

- Key note:

SQLAlchemy Session and framework Session are completely independent.

You can use both simultaneously: **SQLAlchemy Session** for working with data, and framework Session for user management.

Basic Queries

When working with Session, you're essentially querying the database. Session gives you capabilities to read, filter, sort and limit data.

Query objects

Query Objects

- When you want to get data, first you create a Query Object:

```
session = Session()

# Create a Query for User table
query = session.query(User)

print(type(query))
# Output: <class 'sqlalchemy.orm.query.Query'>
```

This Query Object is like a ready template. Then you can execute on it:

```
users = query.all()   # All Users
first_user = query.first()  # First User
user_count = query.count()  # Total count
one_user = query.one()      # Must be exactly one
```

Filtering

Want to get only a part of the data? Use filter or filter_by.

```
# All people over 18 years old
adult_users = session.query(User).filter(User.age >= 18).all()

# All active users
active_users = session.query(User).filter_by(is_active=True).all()
```

🔹 You can also use multiple conditions:

```
query = session.query(User).filter(
    User.age >= 18,
    User.is_active == True,
    User.city == 'Tehran'
)
```

📌 Operators:

```
session.query(User).filter(User.name.like('Ahmad%'))   # Starts with Ahmad
session.query(User).filter(User.email.contains('@gmail'))  # Contains @gmail
session.query(User).filter(User.id.in_([1, 2, 3]))    # In list
```

```
session.query(User).filter(User.age.between(18, 65))  # Between 18 and 65
session.query(User).filter(User.phone.is_(None))      # NULL
```

🔹 Combined conditions:

```
from sqlalchemy import and_, or_, not_

# AND
session.query(User).filter(and_(User.age >= 18, User.is_active == True))

# OR
session.query(User).filter(or_(User.city == 'Tehran', User.city == 'Isfahan'))

# NOT
session.query(User).filter(not_(User.is_deleted))
```

Ordering

Ordering (Sorting)

```
# Ascending
users = session.query(User).order_by(User.name).all()

# Descending
users = session.query(User).order_by(User.created_at.desc()).all()

# Multiple columns
users = session.query(User).order_by(
    User.city,
    User.name.desc()
).all()
```

📌 More advanced:

```
from sqlalchemy import func

# By name length
session.query(User).order_by(func.length(User.name))

# Random
session.query(User).order_by(func.random()).limit(5)
```

```
# NULLs last
session.query(User).order_by(User.phone.nullslast())
```

Limiting

Often you don't want all the data. For this we have Pagination:

```
# Only 10 records
users = session.query(User).limit(10).all()

# From record 20 to 30
users = session.query(User).offset(20).limit(10).all()

# Latest 5 people
latest_users = session.query(User)\
    .order_by(User.created_at.desc())\
    .limit(5)\
    .all()
```

Relationships and Advanced ORM

Relationships

When we have multiple tables, we need to define their relationships:

- One user can have multiple orders (One-to-Many)
- One order belongs to only one user (Many-to-One)
- Each user has one profile (One-to-One)
- Users can have different roles and vice versa (Many-to-Many)
- ForeignKey ← In database tells which table this record connects to. A column that points to a record in another table
- relationship ← In Python allows you to easily access related records. This is specific to ORM (like SQLAlchemy or Django ORM). Instead of constantly JOINing, you can directly use objects.

One-to-Many

One user can have multiple orders.

```python
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    orders = relationship("Order", back_populates="user")  # List of orders

class Order(Base):
    __tablename__ = "orders"
    id = Column(Integer, primary_key=True)
    total = Column(Float)
    user_id = Column(Integer, ForeignKey("users.id"))  # Which user does this order

    user = relationship("User", back_populates="orders")
```

Usage:

```python
user = session.query(User).first()
print(user.orders)       # List of orders
```

In the Order table we have a ForeignKey (user_id) ← this is the main connection in the database.

But ORM (SQLAlchemy) wants to create a bidirectional relationship:

From User side: access to list of orders ← user.orders

From Order side: access to order owner ← order.user

If you only put relationship on one side, it works from that side, but usually bidirectional is more convenient (so you can use both user.orders and order.user).

👉 So: Two relationships are for coding convenience, not for determining relationship type. The relationship type is actually determined by the ForeignKey.

Many-to-One

Same example above, just looking from the order side:

```python
order = session.query(Order).first()
print(f"This order is for {order.user.name}")
```

One-to-One

Each user has one profile.

```
class Profile(Base):
    __tablename__ = "profiles"
    id = Column(Integer, primary_key=True)
    bio = Column(String(200))
    user_id = Column(Integer, ForeignKey("users.id"), unique=True)

    user = relationship("User", back_populates="profile")

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    profile = relationship("Profile", back_populates="user", uselist=False)
```

Foreign key (FK) means "this column connects to another table".

When you also add unique=True it means each user_id can only appear once in the table.

✅ Result: This way the relationship becomes One-to-One (each user can only have one profile).

Without unique=True it becomes One-to-Many (each user can have multiple profiles).

By default relationship thinks it might return multiple objects (e.g., user.orders → a list).

But when the relationship is One-to-One, you want a single object returned, not a list.

This is where `uselist=False` is used.

Many-to-Many

Users can have multiple roles (admin, regular user).

```
user_roles = Table(
    "user_roles", Base.metadata,
    Column("user_id", Integer, ForeignKey("users.id")),
    Column("role_id", Integer, ForeignKey("roles.id"))
)

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

```
    roles = relationship("Role", secondary=user_roles, back_populates="users")

class Role(Base):
    __tablename__ = "roles"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    users = relationship("User", secondary=user_roles, back_populates="roles")
```

What is this secondary=user_roles in Many-to-Many? In Many-to-Many relationships we must have an intermediate table Here the user_roles table acts like a bridge between users and roles.

Back references

- back_populates

You must define both sides of the relationship yourself.

You manually specify the attribute name.

Example One-to-Many (user ↔ orders):

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    orders = relationship("Order", back_populates="user")  # User side

class Order(Base):
    __tablename__ = "orders"
    id = Column(Integer, primary_key=True)
    total = Column(Float)
    user_id = Column(Integer, ForeignKey("users.id"))

    user = relationship("User", back_populates="orders")  # Order side
```

📌 This means:

From user side: user.orders ← list of orders

From order side: order.user ← user related to that order

Here you have two relationship lines and they must point to each other exactly.

- backref

Instead of defining twice, you define once, and sqlalchemy creates the opposite side itself.

It's a shortcut.

Same example above, but shorter:

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

class Order(Base):
    __tablename__ = "orders"
    id = Column(Integer, primary_key=True)
    total = Column(Float)
    user_id = Column(Integer, ForeignKey("users.id"))

    user = relationship("User", backref="orders")
```

📌 Now you have both:

```
order.user
user.orders
```

But you only wrote relationship once.

Lazy Loading vs Eager Loading

Lazy loading patterns

Data is loaded when needed:

```
user = session.query(User).first()  # Only User table is queried
print(user.name)

# When you access here, orders are loaded now
print(len(user.orders))
# => Now a new query is executed
```

Eager loading (joinedload, selectinload)

All data is loaded at once:

🔹 Difference between selectinload and joinedload

- Both are for Eager Loading (meaning they bring data from the beginning), but their working method is different:

📍 joinedload

Uses JOIN in the main query itself.

Everything comes with one query.

Faster when there's little data.

Problem: If the child table (like orders) has many records, a lot of duplicate data is pulled.

📌 Example:

```
users = session.query(User).options(joinedload(User.orders)).all()
# SQL: SELECT users.*, orders.* FROM users LEFT JOIN orders ...
```

📍 selectinload

First brings all Users, then makes a separate query for all orders.

Result: Two queries (but optimized).

Better when there are many child records.

📌 Example:

```
users = session.query(User).options(selectinload(User.orders)).all()
# SQL 1: SELECT * FROM users
# SQL 2: SELECT * FROM orders WHERE user_id IN (list of all ids)
```

✅ So:

Little data ← joinedload is good.

Lots of data ← selectinload is more optimal.

N+1 problem

When to get a dataset, 1 initial query + N additional queries for each record are executed. 📌 **Example (N+1):**

```
users = session.query(User).all()  # 1 query
for user in users:
    print(len(user.orders))        # N additional queries (separate for each User)
```

📌 **Example (Eager Loading):**

```
users = session.query(User).options(joinedload(User.orders)).all()
for user in users:
    print(len(user.orders))  # No additional queries
```

Loading strategies

🔹 Loading Strategies (relationship loading methods)

When you define relationship, you can determine how data is loaded:

1. Lazy (default: "select")

Query is only executed when needed. 📌 Example:

```
orders = relationship("Order", lazy="select")
# Only when you call user.orders, orders query is executed
```

2. Always Eager ("joined")

Brings along with main entity (User) with JOIN. 📌 Example:

```
orders = relationship("Order", lazy="joined")
# When you get Users, orders come simultaneously
```

3. Only when needed ("dynamic")

Instead of a list, returns a Query object.

You can filter and limit on that same relationship.

📌 Example:

```
orders = relationship("Order", lazy="dynamic")
```

```
user = session.query(User).first()
big_orders = user.orders.filter(Order.price > 100).all()
```

4. Manual Load ("noload")

Never loads automatically.

If you want, you must manually query yourself.

📌 Example:

```
orders = relationship("Order", lazy="noload")
user = session.query(User).first()
print(user.orders)  # Returns empty
```

Advanced Querying

Sometimes simple queries aren't enough and we need to use more advanced tools: Joins, Subquery, Union, Window Functions, Raw SQL.

Complex joins

📍 Inner Join (direct connection)

Only brings data that exists in both tables.

```
result = session.query(User, Order)\
    .join(Order)\
    .filter(Order.total > 100)\
    .all()
# All users whose orders are more than 100
```

📍 **Outer Join (external connection)**

Even if there's no data in the second table, it brings the first table record.

```
result = session.query(User)\
    .outerjoin(Order)\
    .filter(Order.id.is_(None))\
    .all()
# All users who have no orders
```

Subqueries

A query that is used inside another query.

📍 Regular Subquery

```
subq = session.query(Order.user_id)\
    .filter(Order.total > 1000)\
    .subquery()

rich_users = session.query(User)\
    .filter(User.id.in_(subq))\
    .all()
# Users who have orders more than 1000
```

📍 Scalar Subquery

Returns a single value (like average or max).

```
avg_total = session.query(func.avg(Order.total)).scalar_subquery()

above_avg_orders = session.query(Order)\
    .filter(Order.total > avg_total)\
    .all()
# Orders that are above the overall average
```

Explanation of scalar_subquery:

```
session.query(func.avg(Order.total)).scalar()
```

👈 This actually returns a Python number (e.g., 125.3). This means the query executes right now and the average value comes from database. (You can't use it in subsequent queries anymore.)

```
session.query(func.avg(Order.total)).scalar_subquery()
```

👈 This creates a subquery that hasn't been executed in the database yet. You can use this subquery as part of a larger query condition. scalar_subquery() itself returns a subquery object (SQL expression object)

Union operations

For combining output of multiple different queries.

- Union Combines the result of two queries without duplicates.
- **Union (without duplicates)**

```
young_users = session.query(User).filter(User.age < 25)
old_users = session.query(User).filter(User.age > 65)

all_users = young_users.union(old_users).all()
# All users, either very young or very old
```

- **Union All** Combines the result of two queries with duplicates.
- **Union All (with duplicates)**

```
all_users = young_users.union_all(old_users).all()
```

Window functions

For analyzing data across rows (like ranking or numbering).

- **Row Number** Gives each row a unique number, purely for ordering.

```
result = session.query(
    User.name,
    func.row_number().over(order_by=User.created_at).label('row_num')
).all()
# Row number for each user based on registration time
```

- **Rank** Gives rank, but if two records are equal they get the same rank.

Simple example:

- User 1 ← order 500 ← rank 1
- User 1 ← order 500 ← rank 1
- User 1 ← order 300 ← rank 3

```
result = session.query(
    Order.user_id,
```

```
    Order.total,
    func.rank().over(
        partition_by=Order.user_id,
        order_by=Order.total.desc()
    ).label('rank')
).all()
# Rank of each user's order based on amount
```

Raw SQL integration

When query is very complex or you want to use database-specific functions.

- text() for writing raw queries.
- pattern is a placeholder that we safely inject its value.
- fetchall() ← all rows.
- fetchone() ← only one row.
- fetchmany(5) ← specific number of records.
- from_statement() when you want to connect a raw query to ORM (make output ORM)

📍 Direct SQL execution

```
result = session.execute(
    text("SELECT * FROM users WHERE name LIKE :pattern"),
    {"pattern": "Ahmad%"}
).fetchall()
# All users whose names start with Ahmad
```

📍 Combining with ORM

```
users = session.query(User)\
    .from_statement(
        text("SELECT * FROM users WHERE complex_condition()")
    ).all()
```

Advanced Topics

Session Management Patterns

In SQLAlchemy, Session is responsible for managing database connection and executing queries. We have different methods for managing Session that are used in frameworks

Session per request

For each HTTP request a new Session is created and closed after the request ends. This method is common and safe because Sessions are not shared between requests.

FastAPI Example

```
def get_db():
    db = SessionLocal()   # Create a new Session
    try:
        yield db          # Use in endpoint
    finally:
        db.close()        # Close Session after request ends

@app.get("/users")
def get_users(db: Session = Depends(get_db)):
    return db.query(User).all()
```

Contextual sessions

Sometimes multiple functions or threads need to use one Session. scoped_session does this and each Thread has a dedicated Session.

```
from sqlalchemy.orm import scoped_session, sessionmaker

Session = scoped_session(sessionmaker(bind=engine))

def some_function():
    user = Session.query(User).first()  # Same Session for current Thread
    return user

# Clear Session after work is done
Session.remove()
```

- sessionmaker is a Factory for creating Sessions.
- engine is responsible for database connection (like TCP connection or SQLite file).
- When you set bind=engine, it means this Session knows which database to work with.

Example:

```
engine = create_engine("sqlite:///app.db")
Session = sessionmaker(bind=engine)
session = Session()  # This Session is connected to "app.db"
```

Thread-local sessions

Each Thread has its own separate Session. Similar to scoped_session but managed manually.

```
import threading

session_registry = threading.local()

def get_session():
    if not hasattr(session_registry, 'session'):
        session_registry.session = SessionLocal()
    return session_registry.session
```

- session_registry is a storage space specific to each Thread (threading.local()).
- When we write session_registry.session = SessionLocal(), we're creating and storing a dedicated Session for the current Thread.

Session events

We can perform desired operations before or after commit/rollback database changes, like logging or validation.

```
from sqlalchemy import event

@event.listens_for(Session, 'before_commit')
def before_commit(session):
    print("Before Commit")

@event.listens_for(Session, 'after_commit')
```

```
def after_commit(session):
    print("After Commit")
```

before_commit ← before final saving of changes

after_commit ← after final saving of changes

Advanced Relationships

SQLAlchemy has advanced capabilities for modeling relationships between tables. In this section we cover important topics

Self-referential relationships

A table can reference itself

```
class Employee(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_id = Column(Integer, ForeignKey('employees.id'))

    manager = relationship("Employee", remote_side=[id], back_populates="subordinat
    subordinates = relationship("Employee", back_populates="manager")
```

remote_side in SQLAlchemy:

- When a table references itself (Self-referential), SQLAlchemy must know which column is the "other side of the relationship".

Polymorphic relationships

- Child classes can be stored in a shared structure and SQLAlchemy understands what type each row is
- When you have multiple entity types that share some columns, you can use Polymorphic.

```
class Person(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(20))  # Specifies what type it is
    __mapper_args__ = {'polymorphic_on': type, 'polymorphic_identity': 'person'}

class Employee(Person):
    salary = Column(Float)
```

```
    __mapper_args__ = {'polymorphic_identity': 'employee'}

class Customer(Person):
    credit_limit = Column(Float)
    __mapper_args__ = {'polymorphic_identity': 'customer'}
```

1. Role of **mapper_args**

This is a special SQLAlchemy ORM dictionary that tells the class how to behave with tables and inheritance.

Especially when using Polymorphic Inheritance (table inheritance), here we specify how SQLAlchemy should recognize record types.

2. polymorphic_on

polymorphic_on: type

Specifies which table column holds the record type.

In our example, the type column specifies whether this record is a Person, Employee, or Customer.

This means when each record is stored in the table, this column's value determines which class SQLAlchemy assigns to it.

3. polymorphic_identity

polymorphic_identity is a value stored in the table's type column and specifies which class the record belongs to.

This value can be any string, but usually readable and related to the class is chosen

Hybrid properties

You can define a method in a class that is both accessible in Python and usable in SQL queries.

```
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy import func

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    first_name = Column(String(30))
    last_name = Column(String(30))

    @hybrid_property # for python
    def full_name(self):
        return self.first_name + ' ' + self.last_name
```

```
    @full_name.expression # for SQL
    def full_name(cls):
        return func.concat(cls.first_name, ' ', cls.last_name)

# Usage
user = session.query(User).first()
print(user.full_name)  # "Ahmad Ali"
```

When you want to use full_name in filter() or order_by(), SQLAlchemy automatically converts it to SQL:

```
users = session.query(User).filter(User.full_name == 'Ahmad Ali').all()
# Here @full_name.expression is used
```

Customization and Extensions

Custom types

For when we want to store non-standard data types in database, like JSON:

```
from sqlalchemy.types import TypeDecorator, String
import json

class JSONType(TypeDecorator):
    impl = String

    def process_bind_param(self, value, dialect):
        return json.dumps(value) if value else value

    def process_result_value(self, value, dialect):
        return json.loads(value) if value else value

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    preferences = Column(JSONType())

# Usage
user = User(preferences={'theme': 'dark', 'lang': 'en'})
```

TypeDecorator in SQLAlchemy means:

- I want to create a custom database type that is implemented behind the scenes on a standard type.
- Assume database only understands String, Integer, etc.
- But I want to have a column that can store dictionary/JSON.

impl:

- impl means the main database type that this custom type is built on.
- Here we say: this JSONType is actually the same String
- So a VARCHAR/TEXT is stored in the database, but we see a dictionary/JSON in Python.

process_bind_param:

- When you want to send data to database (insert/update), this function runs.

process_result_value:

- When you get data from database (select), this function runs.

Validators

With @validates you can perform validation before saving data:

```
from sqlalchemy.orm import validates

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    email = Column(String(100))
    age = Column(Integer)

    @validates('email')
    def validate_email(self, key, email):
        if '@' not in email:
            raise ValueError("Invalid email")
        return email

    @validates('age')
    def validate_age(self, key, age):
        if age < 0 or age > 150:
            raise ValueError("Invalid age")
        return age
```

✅ Wrong data is prevented before Commit.

@validates decorator:

- Before data is stored in this field, first pass it through this function
- key: name of field being validated (e.g., 'email').
- If data is correct, it's returned and stored.
- If wrong, throws Exception.

Validating multiple fields in one function

If you want, you can put one function for multiple fields:

```
@validates('email', 'age')
def validate_fields(self, key, value):
    if key == 'email':
        if '@' not in value:
            raise ValueError("Invalid email")
    elif key == 'age':
        if value < 0 or value > 120:
            raise ValueError("Invalid age")
    return value
```

Events and listeners

We can react to Insert, Update, Delete:

```
from sqlalchemy import event
from datetime import datetime

@event.listens_for(User, 'before_insert')
def set_created_at(mapper, connection, target):
    target.created_at = datetime.utcnow()

@event.listens_for(User, 'after_update')
def log_update(mapper, connection, target):
    print(f"User {target.id} updated")
```

Mixins

We can write common columns and properties in a class and add them to models:

```
class TimestampMixin:
    created_at = Column(DateTime, default=datetime.utcnow)
    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow
```

```
class User(Base, TimestampMixin):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

Custom loading techniques

For controlling what data is loaded during Query

Common problem: N+1 Query Problem

When you get a list of users, then separately query Orders or Profile for each user.

This becomes dozens of separate queries (very slow).

Solution: joinedload / selectinload or custom Loader.

```
from sqlalchemy.orm import joinedload

# custom loader
class UserLoader:
    @staticmethod
    def full_load():
        return [joinedload(User.orders), joinedload(User.profile)]

users = session.query(User).options(*UserLoader.full_load()).all()
```

Note: You can both write a loader and also write manually like this

Complete FastAPI Example

```
from fastapi import FastAPI, Depends
from sqlalchemy.orm import Session, joinedload

app = FastAPI()

@app.get("/users/{user_id}/full")
def get_user_full(user_id: int, db: Session = Depends(get_db)):
    user = db.query(User)\
        .options(joinedload(User.orders), joinedload(User.profile))\
        .filter(User.id == user_id)\
        .first()
    return user
```

Performance and Optimization

Performance Tuning

Query optimization

Common mistake: Get all data, then filter in Python.

Correct method: Use SQL itself to filter and limit data.

```
# Bad: All users, then filter in Python
all_users = session.query(User).all()
active_users = [u for u in all_users if u.is_active]

# Good: Filter directly in database
active_users = session.query(User).filter(User.is_active == True).all()
```

Index strategies

Index works like a book's table of contents ← instead of flipping through the entire table, quickly finds the desired record.

For columns that are frequently searched/filtered, add index.

- **Example:**

```
class User(Base):
    __tablename__ = 'users'
    email = Column(String(100), index=True)  # Index on email
```

- **Composite index (multiple columns together):**

```
Index('idx_status_date', Order.status, Order.created_at)
```

Connection pooling

Connecting to database each time is expensive.

Connection Pool causes connections to be cached and reused.

- **Example:**

```
engine = create_engine(
    "postgresql://user:pass@localhost/db",
    pool_size=20,        # 20 ready connections
    max_overflow=10,     # 10 additional connections at peak
    pool_timeout=30,     # Wait 30 seconds
    pool_recycle=3600,   # Refresh every 1 hour
    pool_pre_ping=True   # Test before use
)
```

Bulk operations

Instead of insert/update/delete one by one → do all at once.

- **Insert:**

```
# Bad
for i in range(1000):
    session.add(User(name=f"User{i}"))
session.commit()

# Good
users_data = [{"name": f"User{i}"} for i in range(1000)]
session.bulk_insert_mappings(User, users_data)
session.commit()
```

- **Update:**

```
# Good
session.query(User)\
    .filter(User.last_login < old_date)\
    .update({"is_active": False})
session.commit()
```

- **Delete:**

```
session.query(LogEntry)\
    .filter(LogEntry.created_at < old_date)\
    .delete()
session.commit()
```

Caching

Cache means temporarily storing data in memory so that next time we need the same data, we don't refer to database and response is faster.

Types of cache and applications

1. Simple in-process cache

- Storage location: Memory of the same Python process.
- Limitation: Only for the same process and not shared across multiple servers or processes.
- Conceptual example: A simple dictionary that holds query results.
- Common usage: functools.lru_cache for storing function results.

2. Distributed cache

- Storage location: External system like Redis or Memcached.
- All processes and servers can use it.
- Can define TTL (expiration time) for data.
- Requires data serialization (JSON or pickle).

3. Second-level cache (SQLAlchemy)

- SQLAlchemy itself has cache at model and session level.
- Concept: A record from database that was read first time is kept in second-level cache so next Session can use it without referring to database.
- Conceptual example: User with id=1 was read from database first time, second time returns from SQLAlchemy internal cache.

4. Dogpile.cache (Professional Cache)

- A powerful tool for cache management.
- Configurable cache regions.
- TTL (expiration time) and automatic invalidation.
- Support for Redis, Memcached, and internal memory.
- Concept: Wraps functions or queries and safely puts results in cache.
- Cache invalidation after data changes is essential to avoid returning stale data.

Query result caching

Simple cache with functools.lru_cache

Idea: Keep function results in memory so next time without going to database, return the same result.

```python
from functools import lru_cache

# Function that reads user from database
@lru_cache(maxsize=100)  # Keep maximum 100 results in memory
def get_user_by_email(email):
    print("Querying DB for", email)
    return session.query(User).filter(User.email == email).first()

# Usage
user1 = get_user_by_email("test@example.com")  # This time read from database
user2 = get_user_by_email("test@example.com")  # This time read from cache
```

Second-level cache

Goal: Data shared between multiple Sessions or Threads is kept in a central cache.

Unlike simple Session cache (which is only active in the same Session), this cache is usable between different Sessions.

```python
from sqlalchemy_utils import CacheManager

# Setup Cache Manager
cache_manager = CacheManager()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    # Cache for this model
    cache = cache_manager.cache

# Usage
user = session.query(User).filter(User.id == 1).first()  # From DB
user = session.query(User).filter(User.id == 1).first()  # From Cache
```

Dogpile.cache integration

A specialized cache library with advanced features like automatic invalidation and TTL.

Installation and setup:

```
pip install dogpile.cache
```

```
from dogpile.cache import make_region

# Setup Region
region = make_region().configure(
    'dogpile.cache.memory',  # Internal memory; can also use Redis
    expiration_time=600      # 10 minutes
)

@region.cache_on_arguments()
def get_user_profile(user_id):
    return session.query(User).filter(User.id == user_id).first()

# Usage
user = get_user_profile(1)  # First time from DB
user = get_user_profile(1)  # Second time from cache

# Clear cache after data change
def update_user(user_id, **kwargs):
    session.query(User).filter(User.id == user_id).update(kwargs)
    session.commit()

    region.delete(f"get_user_profile|{user_id}")  # Clear cache
```

✅ Note:

cache_on_arguments() automatically creates cache key.

After updating data, make sure to clear cache so fresh data returns.

Advanced Features

Migrations with Alembic

When your models (models.py) change (e.g., you add a column or table), database doesn't change automatically.

Migration is a tool that records changes step by step and applies them to database, like versioning for database.

Migration basics

Installation and setup:

```
pip install alembic

# Start project
alembic init alembic
```

An alembic/ folder is created.

Inside it there's env.py where you should give it Base.metadata (so it knows what models are).

Inside alembic.ini you put database address:

```
# alembic.ini configuration
sqlalchemy.url = postgresql://user:pass@localhost/dbname
```

Auto-generating migrations

When model changed (e.g., User added):

```
alembic revision --autogenerate -m "Add user table"
alembic upgrade head
```

First command creates a migration file in alembic/versions/.

Second command runs that migration on database.

To go back:

```
alembic downgrade -1   # Go back one migration
```

This means upgrade applies, downgrade reverses it.

Simple example of generated Migration

```
def upgrade():
    op.create_table(
        'users',
        sa.Column('id', sa.Integer, primary_key=True),
        sa.Column('name', sa.String(50)),
        sa.Column('email', sa.String(100))
    )

def downgrade():
    op.drop_table('users')
```

Manual migrations

Manual Migration means when automatic (autogenerate) is not enough — you must manually change Schema and/or data.

Manual Migration means you create an empty revision (alembic revision -m "...") and inside upgrade() and downgrade() you write the necessary op.* and SQL codes yourself.

- **General template of a migration file**

```
# versions/xxxx_custom_migration.py
from alembic import op
import sqlalchemy as sa

revision = 'xxxx'
down_revision = 'prev_rev'
branch_labels = None
depends_on = None

def upgrade():
    # Commands that should run when upgrading
    pass

def downgrade():
    # Reverse of upgrade for rollback
    pass
```

Note: Always write a downgrade() even if simple so you can go back if needed.

This section is somewhat more specialized and is avoided from being mentioned here

Branching and merging

When multiple people work on project, each might create their own migration → migrations become branched.

Creating branch:

```
alembic revision -m "Feature A" --branch-label=feature_a
```

Merging multiple migrations:

```
alembic merge -m "merge features" head1 head2
```

Viewing history:

```
alembic history --verbose
```

Viewing current database status:

```
alembic current
```

Migration is like git commit for database.

Testing

Testing patterns

What is @pytest.fixture?

In pytest, Fixture is a solution for preparing data, resources, or test environment.

Instead of recreating things like Session or data in each test, Fixture does this once and injects it into tests.

What does scope="session" mean?

Fixture can have different lifetimes:

- function (default): Each test receives a new instance.
- class: Each test class has one instance.
- module: Each module has one instance.
- session: All tests in one run share one instance.

Test setup

We use SQLite in-memory for fast testing:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from myapp.models import Base
import pytest

@pytest.fixture(scope="session")
def test_engine():
    engine = create_engine("sqlite:///:memory:")  # Temporary database in memory
    Base.metadata.create_all(engine)
    return engine

@pytest.fixture
def session(test_engine):
    Session = sessionmaker(bind=test_engine)
    session = Session()
    yield session
    session.rollback()   # After each test all changes are rolled back
    session.close()
```

Each test has a fresh Session, so changes don't interfere between tests.

scope session means creating engine only once for all tests.

Simple model testing

```
def test_create_user(session):
    user = User(name="Ahmad", email="ahmad@test.com")
    session.add(user)
    session.commit()

    assert user.id is not None   # User was created
    assert user.name == "Ahmad"
```

Testing relationships

```python
def test_user_relationships(session):
    user = User(name="Ali")
    order = Order(total=100.0)
    user.orders.append(order)

    session.add(user)
    session.commit()

    assert len(user.orders) == 1
    assert user.orders[0].total == 100.0
```

Fixtures

We can create ready samples and use them in multiple tests:

```python
@pytest.fixture
def sample_user(session):
    user = User(name="Test User", email="test@example.com")
    session.add(user)
    session.commit()
    return user
```

Mock strategies

Suppose a function connects to external API or other service, we don't want each test to call this real service.

Mock means replacing real with simulated version that returns desired result.

Database testing

Real database testing

We can use real PostgreSQL or MySQL:

```python
@pytest.fixture(scope="session")
def postgres_engine():
    engine = create_engine("postgresql://test_user:test_pass@localhost/test_db")
    Base.metadata.create_all(engine)
    yield engine
    Base.metadata.drop_all(engine)

@pytest.fixture
```

```
def postgres_session(postgres_engine):
    Session = sessionmaker(bind=postgres_engine)
    session = Session()
    yield session
    session.rollback()
    session.close()
```

Advantage: Testing is done on real database, real SQL bugs are found.

Real-world Applications

Design Patterns

Repository pattern

What is it? Creates a layer between application code and database to make data access organized and standardized. Instead of directly writing session.query(User), you use a Repository.

Advantages:

Separating database logic from business layer

Making tests easier

Changing database without changing entire code

Simple example:

```
# repository.py
class UserRepository:
    def __init__(self, session):
        self.session = session

    def get_by_id(self, user_id):
        return self.session.query(User).filter(User.id == user_id).first()

    def add(self, user):
        self.session.add(user)
        self.session.commit()

# Usage
repo = UserRepository(session)
```

```
user = repo.get_by_id(1)
repo.add(User(name="Ali"))
```

✅ Result: Entire application depends on Repository, not directly on Session or SQLAlchemy.

Unit of Work

What is it? A pattern for managing transactions and Sessions. Ensures all changes are committed or rolled back together.

Simple example:

```
class UnitOfWork:
    def __init__(self, session_factory):
        self.session_factory = session_factory

    def __enter__(self):
        self.session = self.session_factory()
        return self.session

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type:
            self.session.rollback()
        else:
            self.session.commit()
        self.session.close()

# Usage
with UnitOfWork(Session) as session:
    user = User(name="Ahmad")
    session.add(user)  # If there's an error, rollback happens
```

Important note: **enter** and **exit** methods are automatically executed when you use with.

When you write with UnitOfWork(Session) as session:, Python automatically calls **enter** and gives you the result (session).

- When the with block ends (even if an error occurred), Python automatically calls **exit**.
- Inside **exit** we have specified:
- If there was an error rollback()
- If there was no error commit()
- Then Session is closed with close()

Data Mapper

What is it? SQLAlchemy itself uses this pattern. Main idea: Python classes are mapped to database tables and classes operate independently from database.

- Classes only hold data and business logic.
- Session / Mapper is responsible for storing and loading data.

Simple example:

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

# Mapper separate from User class
user = User(name="Sara")
session.add(user)
session.commit()
```

Mapper is the same Base and SQLAlchemy ORM that maps User class to users table behind the scenes.

You haven't written any SQL at all.

Mapper does this: converts User to an INSERT INTO users ... and sends to database.

✅ This means Mapper is what establishes connection between Python class and database table.

✅ Result: Your classes are independent and database only works with Mapper and Session.

Active Record considerations

What is it? A pattern where class contains both data and database logic. For example, class itself performs Save, Update and Delete.

This pattern is commonly seen in Django ORM or Rails.

SQLAlchemy can become like Active Record but Data Mapper is more recommended because it has more flexibility.

Simple Active Record-style example:

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
```

```
    name = Column(String(50))

    def save(self, session):
        session.add(self)
        session.commit()

# Usage
user = User(name="Neda")
user.save(session)  # Class itself saved the data
```

Summary:

1️⃣ Session

Main job: Managing all interactions with database.

Its duties:

- Getting data (query)
- Adding or deleting records (add, delete)
- commit and rollback transactions
- Maintaining Object states (Transient, Pending, Persistent, Detached)

You can think: Session is like a workbook that records and manages all changes to database.

2️⃣ Mapper (or ORM)

Main job: Mapping Python classes to database tables.

Its duties:

- Specifying which Python class belongs to which table (Declarative Base)
- Converting Objects to SQL and vice versa
- Managing relationships between classes (One-to-Many, Many-to-Many, Polymorphic)

You can think: Mapper is the bridge between Python classes and database tables.

Integration

Web framework integration (Flask, FastAPI)

Here the goal is to use SQLAlchemy alongside a web framework so Requests can easily work with database.

FastAPI example:

```python
from fastapi import FastAPI, Depends
from sqlalchemy.orm import Session
from database import SessionLocal, User

app = FastAPI()

# Session per Request
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.get("/users/{user_id}")
def get_user(user_id: int, db: Session = Depends(get_db)):
    return db.query(User).filter(User.id == user_id).first()
```

Async SQLAlchemy

Async version is used for asynchronous database work in frameworks like FastAPI or Starlette.

```python
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker
from models import User

engine = create_async_engine("postgresql+asyncpg://user:pass@localhost/db")
AsyncSessionLocal = sessionmaker(engine, class_=AsyncSession, expire_on_commit=Fals

async def get_user(user_id: int):
    async with AsyncSessionLocal() as session:
        result = await session.get(User, user_id)
        return result
```

✅ Note: Async is very suitable for applications with high I/O and prevents blocking the event loop.

Multi-database setups

Sometimes projects have multiple databases (e.g., separate read and write or different databases for modules). SQLAlchemy allows defining multiple engines and Sessions.

```python
# Two Engines for two different databases
engine_main = create_engine("postgresql://user:pass@main_db")
engine_logs = create_engine("postgresql://user:pass@logs_db")

SessionMain = sessionmaker(bind=engine_main)
SessionLogs = sessionmaker(bind=engine_logs)

# Usage
with SessionMain() as main_session, SessionLogs() as log_session:
    user = main_session.query(User).first()
    log_session.add(LogEntry(message="Fetched user"))
    log_session.commit()
```

Sharding strategies

Sharding means dividing data across multiple databases for scalability. SQLAlchemy manages Sharding itself.

```python
from sqlalchemy.ext.horizontal_shard import ShardedSession

# Assume we have two databases
shards = {
    'shard_1': create_engine('postgresql://user:pass@db1'),
    'shard_2': create_engine('postgresql://user:pass@db2')
}

def shard_chooser(mapper, instance, clause=None):
    # Decision making for where record should go
    return 'shard_1' if instance.id % 2 == 0 else 'shard_2'

session = ShardedSession(shards=shards, shard_chooser=shard_chooser)

user = User(id=5, name="Ali")
session.add(user)  # Automatically goes to appropriate shard
session.commit()
```

✅ Note: Sharding is mostly used for large and scalable databases.

🎉 End of Comprehensive SQLAlchemy Reference

Congratulations! 🎊

You have successfully studied one of the **most complete and comprehensive English SQLAlchemy references**. This reference includes:

- ✅ **Basic concepts** from zero to hundred
- ✅ **Advanced techniques** for real projects
- ✅ **Best practices** in industry
- ✅ **Professional design patterns**
- ✅ **Performance optimization** and caching
- ✅ **Testing and Migration** with Alembic
- ✅ **Integration with modern web frameworks**

Next suggestions 🚀

Now that you have learned SQLAlchemy well:

1. **Build a practical project** - The best way to learn is practice
2. **Combine with FastAPI** - For building modern APIs
3. **Try Async SQLAlchemy** - For high-traffic applications
4. **Learn advanced PostgreSQL** - For optimal use of features

Special thanks 🙏

We thank you for your **patience and companionship** in this learning journey. We hope this reference has been useful for you and will help you in your future projects.

Stay in touch 📫

If you have questions, suggestions, or need more guidance, we'd be happy to help you.

Be successful and victorious! 💪

🌟 **Made with ❤️ by Amirhosein Babaee for the English-speaking developer community** 🌟