

**Published by Randy Fadler July 2025**

**Executive Summary**

This booklet guides you through integrating Python applications with SQL Server using SQLAlchemy. It starts by setting up your development environment, including Python virtual environments and required drivers. You’ll learn both the Core and ORM layers of SQLAlchemy, from defining tables and executing raw SQL to modeling complex relationships and performing migrations with Alembic.

Each chapter provides concrete, laptop-ready examples tailored for SQL Server, complete with connection strings, code snippets, and step-by-step instructions. By the end, you’ll be equipped to build robust, maintainable data-driven Python applications, optimize performance, and adopt best practices for testing and deployment.

**Table of Contents**

1. Chapter 1: Setting Up Your Environment
2. Chapter 2: SQLAlchemy Core Basics
3. Chapter 3: The ORM Layer and Declarative Mapping
4. Chapter 4: Querying and Filtering Data
5. Chapter 5: Inserting, Updating, and Deleting Records
6. Chapter 6: Working with Relationships
7. Chapter 7: Migrations with Alembic
8. Chapter 8: Transaction Management and Concurrency
9. Chapter 9: Performance Tuning and Bulk Operations
10. Chapter 10: Asynchronous Patterns with SQLAlchemy
11. Chapter 11: Testing Strategies and Fixtures
12. Chapter 12: Real-World Case Studies
13. Appendix A: Connection Strings and Drivers
14. Appendix B: Common Error Messages and Troubleshooting

Contents

[Chapter 1: Setting Up Your Environment (Updated for Python 3.12.10, Windows Authentication, pyodbc) 7](#_Toc202959305)

[1.1 Verify Python Installation 7](#_Toc202959306)

[Chapter 2: SQLAlchemy Core Basics 9](#_Toc202959307)

[2.1 Prerequisites 9](#_Toc202959308)

[2.2 Defining Table Schemas 9](#_Toc202959309)

[2.3 Creating Tables in the Database 10](#_Toc202959310)

[2.4 Inserting Records 10](#_Toc202959311)

[2.5 Querying Records 11](#_Toc202959312)

[2.6 Updating and Deleting 11](#_Toc202959313)

[Chapter 3: The ORM Layer and Declarative Mapping 13](#_Toc202959314)

[3.1 Why Use the ORM? 13](#_Toc202959315)

[3.2 Setting Up the Declarative Base 13](#_Toc202959316)

[3.3 Defining Model Classes 14](#_Toc202959317)

[3.4 Creating Tables from Models 15](#_Toc202959318)

[3.5 Session Management and Object Lifecycle 16](#_Toc202959319)

[3.6 CRUD with the ORM 16](#_Toc202959320)

[3.7 Transaction Handling and Error Scenarios 18](#_Toc202959321)

[3.8 Hybrid Attributes and Computed Columns 19](#_Toc202959322)

[3.9 Reflecting Existing Schemas 19](#_Toc202959323)

[Chapter 4: Querying and Filtering Data 21](#_Toc202959324)

[4.1 Core vs ORM Querying 21](#_Toc202959325)

[4.2 Filtering with WHERE Clauses 21](#_Toc202959326)

[4.3 Ordering, Limiting, and Pagination 22](#_Toc202959327)

[4.4 Joining Tables 23](#_Toc202959328)

[4.5 Aggregations and GROUP BY 24](#_Toc202959329)

[4.6 Aliases, Labels, and Subqueries 25](#_Toc202959330)

[4.7 Executing Raw SQL 26](#_Toc202959331)

[4.8 Best Practices 26](#_Toc202959332)

[Chapter 5: Inserting, Updating, and Deleting Records 27](#_Toc202959333)

[5.1 Inserting Single Records 27](#_Toc202959334)

[5.2 Bulk Inserts 28](#_Toc202959335)

[5.3 Updating Single Records 30](#_Toc202959336)

[5.4 Bulk Updates 31](#_Toc202959337)

[5.5 Deleting Single Records 32](#_Toc202959338)

[5.6 Bulk Deletes 32](#_Toc202959339)

[5.7 Upserts (MERGE) in SQL Server 33](#_Toc202959340)

[5.8 Using OUTPUT to Return Affected Rows 34](#_Toc202959341)

[Chapter 6: Working with Relationships 36](#_Toc202959342)

[6.1 Overview of Relationship Types 36](#_Toc202959343)

[6.2 One-to-Many / Many-to-One Example 36](#_Toc202959344)

[6.2.2 Creating Tables 37](#_Toc202959345)

[6.3 Many-to-Many Relationships 38](#_Toc202959346)

[6.5 Relationship Loading Strategies 40](#_Toc202959347)

[6.6 Cascading and Orphans 41](#_Toc202959348)

[Chapter 7: Migrations with Alembic 42](#_Toc202959349)

[7.1 Installing and Configuring Alembic 42](#_Toc202959350)

[7.2 Configuring alembic.ini and env.py for SQL Server 42](#_Toc202959351)

[7.3 Generating New Migrations 44](#_Toc202959352)

[7.4 Applying and Rolling Back Migrations 44](#_Toc202959353)

[7.5 Revision Management and Branching 45](#_Toc202959354)

[7.6 Handling Complex Schema Changes 45](#_Toc202959355)

[7.7 Best Practices and Tips 46](#_Toc202959356)

[Chapter 8: Transaction Management and Concurrency 47](#_Toc202959357)

[8.1 Basics of Transactions in SQLAlchemy 47](#_Toc202959358)

[8.2 Isolation Levels 48](#_Toc202959359)

[8.3 Savepoints and Nested Transactions 48](#_Toc202959360)

[8.5 Optimistic Concurrency Control (Version Counters) 49](#_Toc202959361)

[8.6 Two-Phase (Distributed) Transactions 50](#_Toc202959362)

[8.7 Best Practices 51](#_Toc202959363)

[Chapter 9: Performance Tuning and Bulk Operations 52](#_Toc202959364)

[9.1 Understanding SQL Server Indexing 52](#_Toc202959365)

[9.2 Measuring Query Performance 53](#_Toc202959366)

[9.2.2 Execution Plan 53](#_Toc202959367)

[9.3 Connection Pool and Engine Configuration 53](#_Toc202959368)

[9.5 Bulk Updates and Deletes in Batches 55](#_Toc202959369)

[9.6 Streaming Large Result Sets 55](#_Toc202959370)

[9.7 Compiled Queries and Statement Caching 56](#_Toc202959371)

[9.8 Using Table-Valued Parameters via Stored Procedures 56](#_Toc202959372)

[9.9 Best Practices Recap 57](#_Toc202959373)

[Chapter 9: Performance Tuning and Bulk Operations 58](#_Toc202959374)

[9.1 Understanding SQL Server Indexing 58](#_Toc202959375)

[9.2 Measuring Query Performance 59](#_Toc202959376)

[9.3 Connection Pool and Engine Configuration 59](#_Toc202959377)

[9.4 Fast Bulk Inserts with pyodbc.fast\_executemany 60](#_Toc202959378)

[9.5 Bulk Updates and Deletes in Batches 61](#_Toc202959379)

[9.7 Compiled Queries and Statement Caching 62](#_Toc202959380)

[9.8 Using Table-Valued Parameters via Stored Procedures 62](#_Toc202959381)

[9.9 Best Practices Recap 63](#_Toc202959382)

[Chapter 11: Testing Strategies and Fixtures 64](#_Toc202959383)

[11.1 Types of Tests 64](#_Toc202959384)

[11.2 Isolated Test Databases 64](#_Toc202959385)

[11.3 Using pytest Fixtures for Engine and Session 64](#_Toc202959386)

[11.4 Example: Testing CRUD Operations 65](#_Toc202959387)

[11.5 Advanced Fixtures and Factories 66](#_Toc202959388)

[11.6 Continuous Integration Considerations 66](#_Toc202959389)

[Chapter 12: Real-World Case Studies 68](#_Toc202959390)

[12.1 Case Study: ETL Data Warehouse Load 68](#_Toc202959391)

[12.2 Case Study: Web Dashboard with Dynamic Filters 70](#_Toc202959392)

[12.3 Case Study: Audit Logging All SQL Executions 71](#_Toc202959393)

[12.4 Case Study: Real-Time Chat App with FastAPI & Socket.IO 72](#_Toc202959394)

[12.5 Case Study: Tkinter Desktop App for Data Entry 73](#_Toc202959395)

[Appendix A: Connection Strings and Drivers 75](#_Toc202959396)

[A.1 ODBC Driver 17 for SQL Server 75](#_Toc202959397)

[A.2 pymssql Driver (Alternative) 76](#_Toc202959398)

[A.3 Driver Comparison 76](#_Toc202959399)

[A.4 Choosing the Right Pattern 77](#_Toc202959400)

[Appendix B: Common Error Messages and Troubleshooting 78](#_Toc202959401)

[B.1 Connection and Authentication Errors 78](#_Toc202959402)

[B.2 SQLAlchemy Core and ORM Errors 79](#_Toc202959403)

[B.3 Query and Transaction Errors 80](#_Toc202959404)

[B.4 Alembic Migration Errors 80](#_Toc202959405)

[B.5 General Troubleshooting Tips 81](#_Toc202959406)

# Chapter 1: Setting Up Your Environment (Updated for Python 3.12.10, Windows Authentication, pyodbc)

## 1.1 Verify Python Installation

1. **Open PowerShell or CMD, run:**

**bash**

python –version

1. **You should see:**
2. **Python 3.12.10**
3. **If not installed, download from https://python.org and install with “Add to PATH” checked.**

**1.2 Create and Activate a Virtual Environment**

**powershell**

cd C:\Projects\sqlalchemy\_booklet

python -m venv venv

.\venv\Scripts\activate

**Your prompt should now begin with (venv).**

**1.3 Install Dependencies**

**With (venv) active, install SQLAlchemy and pyodbc:**

**powershell**

pip install sqlalchemy pyodbc

**Optionally pin versions for reproducibility:**

**powershell**

pip install sqlalchemy==2.0.18 pyodbc==4.0.40

**1.4 Configure SQL Server Connectivity**

**Since you’re using Windows Authentication and pyodbc, your connection string will look like:**

mssql+pyodbc://@<SERVER\_NAME>/<DATABASE\_NAME>

?driver=ODBC+Driver+17+for+SQL+Server

&trusted\_connection=yes

* **Replace <SERVER\_NAME> with your instance (e.g., localhost\SQLEXPRESS).**
* **Replace <DATABASE\_NAME> with the database you’ll use (e.g., TestDB).**

**1.5 Test the Connection**

**Create a file connect\_test.py in your project folder:**

**python**

from sqlalchemy import create\_engine, text

engine = create\_engine(

    "mssql+pyodbc://@localhost/master"

    "?driver=ODBC+Driver+17+for+SQL+Server"

    "&trusted\_connection=yes",

    echo=False,

)

with engine.connect() as conn:

    rows = conn.execute(text("SELECT name FROM sys.databases;"))

    print([row.name for row in rows])

**Run it:**

**powershell**

python connect\_test.py

**You should see the SQL Server version printed, confirming successful connectivity.**

**With Chapter 1 complete, you’re ready to start using SQLAlchemy Core for table definitions and data operations.**

# Chapter 2: SQLAlchemy Core Basics

In this chapter, you’ll learn how to define table schemas, create tables in SQL Server, and perform basic CRUD (Create, Read, Update, Delete) operations using SQLAlchemy’s Core API.

## 2.1 Prerequisites

Make sure you have:

* **Python 3.12.10** virtual environment activated
* **sqlalchemy** and **pyodbc** installed
* A database named TestDB (or any database you prefer) on your local SQL Server instance, with your Windows login mapped and granted at least **db\_datareader/db\_datawriter** rights

Connection setup (reuse this in every script):

python

from sqlalchemy import create\_engine

odbc\_url = (

"mssql+pyodbc:///"

"?odbc\_connect="

"Driver=ODBC+Driver+17+for+SQL+Server;"

"Server=localhost;"

"Database=TestDB;"

"Trusted\_Connection=yes;"

)

engine = create\_engine(f"mssql+pyodbc:///?odbc\_connect={odbc\_url}", echo=True)

## 2.2 Defining Table Schemas

Use MetaData and Table to declare your tables in Python:

python

from sqlalchemy import MetaData, Table, Column, Integer, String, DateTime, text

from datetime import datetime

metadata = MetaData()

users = Table(

"users",

metadata,

Column("id", Integer, primary\_key=True, autoincrement=True),

Column("username", String(50), nullable=False, unique=True),

Column("fullname", String(120), nullable=False),

Column("created\_at", DateTime, nullable=False, server\_default=text("GETDATE()")),

)

* MetaData holds a collection of table objects
* Each Column specifies name, type, and constraints

## 2.3 Creating Tables in the Database

Invoke metadata.create\_all() against your engine to generate DDL:

python

def create\_tables():

metadata.create\_all(engine)

if \_\_name\_\_ == "\_\_main\_\_":

create\_tables()

print("Tables created successfully.")

Run:

bash

python create\_tables.py

Check in SSMS under **Databases → TestDB → Tables** to confirm a new **dbo.users** table.

## 2.4 Inserting Records

Use the Core insert() construct:

python

from sqlalchemy import insert

def insert\_users():

stmt = insert(users).values(

username="randy",

fullname="Randy Fadler",

created\_at=datetime.utcnow()

)

with engine.begin() as conn:

result = conn.execute(stmt)

print(f"Inserted user id: {result.inserted\_primary\_key[0]}")

if \_\_name\_\_ == "\_\_main\_\_":

insert\_users()

* engine.begin() provides a transactional context
* result.inserted\_primary\_key returns the new row’s PK

## 2.5 Querying Records

Fetch data with select():

python

from sqlalchemy import select

def fetch\_users():

stmt = select(users.c.id, users.c.username, users.c.created\_at)

with engine.connect() as conn:

for row in conn.execute(stmt):

print(row.id, row.username, row.created\_at)

if \_\_name\_\_ == "\_\_main\_\_":

fetch\_users()

* users.c exposes column expressions
* Iterating conn.execute(stmt) yields Row objects

## 2.6 Updating and Deleting

Perform updates and deletes via update() and delete():

python

from sqlalchemy import update, delete

def update\_user(user\_id, new\_fullname):

stmt = (

update(users)

.where(users.c.id == user\_id)

.values(fullname=new\_fullname)

)

with engine.begin() as conn:

conn.execute(stmt)

print(f"Updated user {user\_id}")

def delete\_user(user\_id):

stmt = delete(users).where(users.c.id == user\_id)

with engine.begin() as conn:

conn.execute(stmt)

print(f"Deleted user {user\_id}")

if \_\_name\_\_ == "\_\_main\_\_":

update\_user(1, "R. Fadler")

delete\_user(1)

* Both update() and delete() return Result objects with rowcount

You now have a solid foundation in SQLAlchemy Core: schema definitions, table creation, and basic CRUD. Next up is using the ORM layer to map Python classes to your tables.

# Chapter 3: The ORM Layer and Declarative Mapping

In this chapter, we’ll step through how to map Python classes to SQL Server tables, configure session management, and perform CRUD operations using SQLAlchemy’s ORM. You’ll see detailed explanations of each concept and code tailored for SQL Server.

## 3.1 Why Use the ORM?

Object-Relational Mapping (ORM) translates database tables into Python classes, letting you:

* Treat rows as native Python objects instead of raw SQL results.
* Encapsulate schema definitions and business logic together.
* Leverage session-level caching to avoid redundant queries.
* Write more maintainable, testable code with higher-level abstractions.

By the end of this chapter, you’ll appreciate how the ORM streamlines data access and enforces consistency between your code and database.

## 3.2 Setting Up the Declarative Base

First, create a module db.py to centralize engine and base setup:

python

# db.py

from sqlalchemy import create\_engine

from sqlalchemy.orm import declarative\_base, sessionmaker

from urllib.parse import quote\_plus

# Build ODBC connection string for SQL Server TestDB

odbc\_str = quote\_plus(

"Driver=ODBC Driver 17 for SQL Server;"

"Server=localhost;"

"Database=TestDB;"

"Trusted\_Connection=yes;"

)

# Create the Engine with echo=True to log SQL statements

engine = create\_engine(

f"mssql+pyodbc:///?odbc\_connect={odbc\_str}",

echo=True,

)

# Base class for all ORM models

Base = declarative\_base()

# Session factory: each call to SessionLocal() produces a new Session

SessionLocal = sessionmaker(

bind=engine,

autoflush=False,

autocommit=False,

)

Important points:

* engine manages the connection pool and SQL dialect.
* Base is the superclass for all mapped classes.
* SessionLocal yields sessions that track object changes and coordinate transactions.

## 3.3 Defining Model Classes

Next, declare your Python classes in models.py to mirror table schemas:

python

# models.py

from sqlalchemy import Column, Integer, String, DateTime, text

from db import Base

class User(Base):

\_\_tablename\_\_ = "users" # Table name in TestDB

id = Column(

Integer,

primary\_key=True,

index=True,

autoincrement=True,

)

username = Column(

String(50),

nullable=False,

unique=True,

index=True,

)

fullname = Column(

String(120),

nullable=False,

)

created\_at = Column(

DateTime,

nullable=False,

server\_default=text("GETDATE()"),

)

def \_\_repr\_\_(self):

return (

f"<User(id={self.id}, username={self.username!r}, "

f"fullname={self.fullname!r})>"

)

Explanation:

* \_\_tablename\_\_ ties this class to the **dbo.users** table.
* Column arguments define data types and constraints.
* index=True speeds up lookups on that column.
* server\_default=text("GETDATE()") ensures SQL Server sets created\_at automatically.

## 3.4 Creating Tables from Models

Generate tables in your SQL Server database by importing all models and calling create\_all():

python

# create\_models.py

from db import engine, Base

import models # ensure the User class is registered

def init\_db():

# This issues CREATE TABLE statements for all Base subclasses

Base.metadata.create\_all(bind=engine)

print("Tables created successfully in TestDB.")

if \_\_name\_\_ == "\_\_main\_\_":

init\_db()

Steps:

1. Run python create\_models.py.
2. In SQL Server Management Studio, under **Databases → TestDB → Tables**, verify **dbo.users** exists with columns as defined.

## 3.5 Session Management and Object Lifecycle

A **Session** tracks changes to objects and handles transactions. Use a context manager for safety:

python

# session\_scope.py

from contextlib import contextmanager

from db import SessionLocal

@contextmanager

def session\_scope():

session = SessionLocal()

try:

yield session # allow caller to use session

session.commit() # commit if no exceptions

except:

session.rollback() # undo on error

raise

finally:

session.close() # always release connection

Key behaviors:

* **Autoflush** writes pending changes before queries if needed.
* **Commit** flushes and persists changes.
* **Rollback** reverts uncommitted changes on exceptions.
* **Close** returns the connection to the pool.

## 3.6 CRUD with the ORM

Using session\_scope(), you can perform Create, Read, Update, and Delete operations.

**3.6.1 Create**

python

# create\_user.py

from session\_scope import session\_scope

from models import User

def create\_user(username, fullname):

with session\_scope() as session:

new\_user = User(username=username, fullname=fullname)

session.add(new\_user) # mark for insertion

session.flush() # send INSERT, assign new\_user.id

print("Created:", new\_user)

if \_\_name\_\_ == "\_\_main\_\_":

create\_user("randy", "Randy Fadler")

Notes:

* session.add() schedules an INSERT.
* session.flush() sends pending INSERTs so new\_user.id is available before commit.

**3.6.2 Read**

python

# get\_users.py

from session\_scope import session\_scope

from models import User

def get\_all\_users():

with session\_scope() as session:

users = session.query(User).all()

for user in users:

print(user)

if \_\_name\_\_ == "\_\_main\_\_":

get\_all\_users()

**3.6.3 Update**

python

# update\_user.py

from session\_scope import session\_scope

from models import User

def update\_fullname(user\_id, new\_name):

with session\_scope() as session:

user = session.get(User, user\_id) # retrieve by primary key

if user:

user.fullname = new\_name # mark change

print("Updated:", user)

else:

print("User not found")

if \_\_name\_\_ == "\_\_main\_\_":

update\_fullname(1, "R. Fadler")

Session auto-detects attribute changes and issues an UPDATE on commit.

**3.6.4 Delete**

python

# delete\_user.py

from session\_scope import session\_scope

from models import User

def delete\_user(user\_id):

with session\_scope() as session:

user = session.get(User, user\_id)

if user:

session.delete(user) # schedule DELETE

print("Deleted user:", user\_id)

else:

print("User not found")

if \_\_name\_\_ == "\_\_main\_\_":

delete\_user(1)

## 3.7 Transaction Handling and Error Scenarios

The session\_scope() context manager ensures:

* **Atomicity**: commit only when all operations succeed.
* **Isolation**: until commit, changes aren’t visible to other sessions.
* **Error rollback**: any exception triggers a rollback, preserving data integrity.

Example of handling unique-constraint violations:

python

# error\_handling.py

from sqlalchemy.exc import IntegrityError

from session\_scope import session\_scope

from models import User

def safe\_create(username, fullname):

try:

with session\_scope() as session:

session.add(User(username=username, fullname=fullname))

except IntegrityError:

print(f"Username {username!r} already exists.")

## 3.8 Hybrid Attributes and Computed Columns

Use @hybrid\_property to define Python methods accessible in queries:

python

# models.py (extend User)

from sqlalchemy.ext.hybrid import hybrid\_property

from sqlalchemy import func

class User(Base):

# existing columns...

@hybrid\_property

def name\_length(self):

return len(self.fullname)

@name\_length.expression

def name\_length(cls):

# Generates LEN(fullname) in SQL

return func.len(cls.fullname)

Now you can do:

python

# hybrid\_example.py

with session\_scope() as session:

users = session.query(User).filter(User.name\_length > 10).all()

for u in users:

print(u.username, u.name\_length)

## 3.9 Reflecting Existing Schemas

If your database already has tables, you can reflect them:

python

# reflected.py

from sqlalchemy import MetaData, Table

from sqlalchemy.orm import mapper

from db import engine, Base

metadata = MetaData()

# Pull columns for the existing dbo.users table

existing\_users = Table(

"users", metadata, autoload\_with=engine, schema="dbo"

)

class ReflectedUser(Base):

\_\_table\_\_ = existing\_users # map class to the reflected Table

def \_\_repr\_\_(self):

return f"<ReflectedUser(id={self.id}, username={self.username})>"

Reflection pros and cons:

* **Pros**: No need to write column definitions manually.
* **Cons**: Slower startup, less compile-time checking of schema mismatches.

# Chapter 4: Querying and Filtering Data

In this chapter, you’ll master building complex queries in SQLAlchemy for SQL Server. You’ll learn to filter rows, sort and paginate results, join tables, perform aggregations, work with aliases and subqueries, and combine SQL expressions—all with clear, runnable examples.

## 4.1 Core vs ORM Querying

You can query your database using either the Core API (imperative constructs) or the ORM (session-based, object-oriented). Both compile to efficient T-SQL for SQL Server. We’ll show both styles.

## 4.2 Filtering with WHERE Clauses

**4.2.1 Core Style**

python

from sqlalchemy import select, and\_, or\_, text

from db import engine, Base, metadata

from models import User

# SELECT \* FROM users WHERE username = 'randy';

stmt = select(User).where(User.username == "randy")

with engine.connect() as conn:

for row in conn.execute(stmt):

print(row)

Common comparison operators:

* ==, !=, <, >, <=, >=
* User.fullname.like("%Fadler%") for wildcard searches
* User.id.in\_([1, 2, 3]) to filter on collections
* User.created\_at.between('2025-01-01', '2025-06-30')

Combine conditions:

python

stmt = select(User).where(

and\_(

User.username != "admin",

or\_(

User.fullname.like("R%"),

User.created\_at >= text("DATEADD(month, -1, GETDATE())")

)

)

)

**4.2.2 ORM Style**

python

from db import SessionLocal

from models import User

session = SessionLocal()

users = (

session.query(User)

.filter(User.username == "randy")

.filter(User.fullname.like("%Fadler%"))

.all()

)

for u in users:

print(u)

session.close()

## 4.3 Ordering, Limiting, and Pagination

**4.3.1 ORDER BY**

python

stmt = select(User).order\_by(User.created\_at.desc(), User.username)

**4.3.2 TOP and FETCH**

SQL Server uses TOP; SQLAlchemy translates limit():

python

# TOP 5

stmt = select(User).order\_by(User.created\_at.desc()).limit(5)

For offset and fetch next (cursor pagination):

python

stmt = stmt.offset(10).limit(5)

**4.3.3 ORM Pagination**

python

session = SessionLocal()

page = session.query(User).order\_by(User.id).offset(0).limit(10).all()

## 4.4 Joining Tables

First, define a second table—**orders**—in models.py:

python

# models.py (append)

from sqlalchemy import ForeignKey, Float

class Order(Base):

\_\_tablename\_\_ = "orders"

id = Column(Integer, primary\_key=True, autoincrement=True)

user\_id = Column(Integer, ForeignKey("users.id"), nullable=False, index=True)

total = Column(Float, nullable=False)

created\_at = Column(DateTime, nullable=False, server\_default=text("GETDATE()"))

def \_\_repr\_\_(self):

return f"<Order(id={self.id}, user\_id={self.user\_id}, total={self.total})>"

Run create\_models.py again to create **dbo.orders**.

**4.4.1 Inner Join (Core)**

python

from sqlalchemy import select

from models import User, Order

stmt = (

select(User.username, Order.id, Order.total)

.join(Order, User.id == Order.user\_id)

.where(Order.total > 100)

)

with engine.connect() as conn:

for username, order\_id, total in conn.execute(stmt):

print(username, order\_id, total)

**4.4.2 Left Outer Join**

python

from sqlalchemy import outerjoin

j = outerjoin(User, Order, User.id == Order.user\_id)

stmt = select(User.username, Order.id).select\_from(j)

**4.4.3 ORM Joins**

python

session = SessionLocal()

results = (

session.query(User.username, Order.total)

.join(Order, User.id == Order.user\_id)

.filter(Order.total < 50)

.all()

)

session.close()

## 4.5 Aggregations and GROUP BY

**4.5.1 Counting and Summing (Core)**

python

from sqlalchemy import func

stmt = (

select(User.username, func.count(Order.id).label("order\_count"), func.sum(Order.total).label("total\_spent"))

.join(Order, User.id == Order.user\_id)

.group\_by(User.username)

.having(func.sum(Order.total) > 500)

)

with engine.connect() as conn:

for row in conn.execute(stmt):

print(row.username, row.order\_count, row.total\_spent)

**4.5.2 ORM Aggregations**

python

session = SessionLocal()

results = (

session.query(

User.username,

func.count(Order.id).label("order\_count"),

func.sum(Order.total).label("total\_spent")

)

.join(Order)

.group\_by(User.username)

.having(func.sum(Order.total) > 500)

.all()

)

session.close()

## 4.6 Aliases, Labels, and Subqueries

**4.6.1 Aliasing Tables**

python

from sqlalchemy import alias

u\_alias = alias(User, name="u")

stmt = select(u\_alias.c.username, func.count(Order.id)).join\_from(u\_alias, Order).group\_by(u\_alias.c.username)

**4.6.2 Subqueries**

python

subq = (

select(Order.user\_id, func.max(Order.total).label("max\_order"))

.group\_by(Order.user\_id)

).subquery()

stmt = (

select(User.username, subq.c.max\_order)

.join(subq, User.id == subq.c.user\_id)

)

with engine.connect() as conn:

print(conn.execute(stmt).all())

## 4.7 Executing Raw SQL

If you need a specific T-SQL feature, you can run raw SQL:

python

from sqlalchemy import text

with engine.connect() as conn:

result = conn.execute(text("SELECT TOP 10 \* FROM orders ORDER BY created\_at DESC"))

print(result.fetchall())

## 4.8 Best Practices

* **Use parameter binding** (:param) to avoid SQL injection.
* **Avoid** text() when ORM or Core constructs can express your intent.
* **Leverage SQL Server functions** via func, e.g., func.getdate().
* **Monitor generated SQL** by keeping echo=True during development.
* **Index columns** used in JOINs and WHERE conditions for performance.

# Chapter 5: Inserting, Updating, and Deleting Records

In this chapter, you’ll learn how to add, modify, and remove data in SQL Server tables using both the Core and ORM APIs. You’ll see patterns for single-row operations, bulk operations, and upserts (MERGE), with detailed explanations and runnable code.

## 5.1 Inserting Single Records

**5.1.1 Core API**

1. Import and define your insert statement.
2. Execute within a transaction.

python

# insert\_core.py

from sqlalchemy import insert

from db import engine, metadata

from models import User

from datetime import datetime

def insert\_user\_core(username: str, fullname: str):

stmt = insert(User).values(

username=username,

fullname=fullname,

created\_at=datetime.utcnow()

)

with engine.begin() as conn:

result = conn.execute(stmt)

new\_id = result.inserted\_primary\_key[0]

print(f"Inserted User ID (Core): {new\_id}")

if \_\_name\_\_ == "\_\_main\_\_":

insert\_user\_core("alice", "Alice Anderson")

* engine.begin() starts a transaction, auto-committing on success.
* result.inserted\_primary\_key returns the new primary key.

**5.1.2 ORM API**

1. Create an ORM Session.
2. Instantiate your model and add it to the session.

python

# insert\_orm.py

from session\_scope import session\_scope

from models import User

def insert\_user\_orm(username: str, fullname: str):

with session\_scope() as session:

new\_user = User(username=username, fullname=fullname)

session.add(new\_user)

session.flush() # sends INSERT so new\_user.id is populated

print(f"Inserted User ID (ORM): {new\_user.id}")

if \_\_name\_\_ == "\_\_main\_\_":

insert\_user\_orm("bob", "Bob Brown")

* session.add() schedules an INSERT.
* session.flush() forces the INSERT so you can read new\_user.id before commit.

## 5.2 Bulk Inserts

**5.2.1 Core Bulk Insert**

Use execute\_many by passing a list of dicts:

python

# bulk\_insert\_core.py

from sqlalchemy import insert

from db import engine

from models import User

from datetime import datetime

def bulk\_insert\_users\_core(users\_data: list[dict]):

stmt = insert(User)

with engine.begin() as conn:

result = conn.execute(

stmt,

[

{\*\*user, "created\_at": datetime.utcnow()}

for user in users\_data

]

)

print(f"Inserted {result.rowcount} users (Core)")

if \_\_name\_\_ == "\_\_main\_\_":

users\_list = [

{"username": "carol", "fullname": "Carol Clark"},

{"username": "dave", "fullname": "Dave Davis"},

]

bulk\_insert\_users\_core(users\_list)

* Passing a list to conn.execute() triggers a fast, multi-row INSERT.

**5.2.2 ORM Bulk Insert**

python

# bulk\_insert\_orm.py

from session\_scope import session\_scope

from models import User

from datetime import datetime

def bulk\_insert\_users\_orm(users\_data: list[dict]):

with session\_scope() as session:

objects = [

User(username=u["username"], fullname=u["fullname"])

for u in users\_data

]

session.bulk\_save\_objects(objects)

# session.bulk\_insert\_mappings(User, users\_data) is an alternative

print(f"Inserted {len(objects)} users (ORM)")

if \_\_name\_\_ == "\_\_main\_\_":

users\_list = [

{"username": "eve", "fullname": "Eve Evans"},

{"username": "frank", "fullname": "Frank Foster"},

]

bulk\_insert\_users\_orm(users\_list)

* bulk\_save\_objects() skips ORM unit-of-work bookkeeping for speed.
* bulk\_insert\_mappings() can insert dicts directly, bypassing object instantiation.

## 5.3 Updating Single Records

**5.3.1 Core API**

python

# update\_core.py

from sqlalchemy import update

from db import engine

from models import User

def update\_user\_fullname\_core(user\_id: int, new\_fullname: str):

stmt = (

update(User)

.where(User.id == user\_id)

.values(fullname=new\_fullname)

)

with engine.begin() as conn:

result = conn.execute(stmt)

print(f"Updated {result.rowcount} row(s) (Core)")

if \_\_name\_\_ == "\_\_main\_\_":

update\_user\_fullname\_core(1, "Alice A.")

* update().where() builds a parameterized UPDATE.
* result.rowcount tells how many rows were changed.

**5.3.2 ORM API**

python

# update\_orm.py

from session\_scope import session\_scope

from models import User

def update\_user\_fullname\_orm(user\_id: int, new\_fullname: str):

with session\_scope() as session:

user = session.get(User, user\_id)

if user:

user.fullname = new\_fullname

print(f"Updated User (ORM): {user}")

else:

print("No such user")

if \_\_name\_\_ == "\_\_main\_\_":

update\_user\_fullname\_orm(2, "Bob B.")

* Loading, modifying, and committing uses the identity map to issue the UPDATE.

## 5.4 Bulk Updates

**5.4.1 Core Bulk Update**

python

# bulk\_update\_core.py

from sqlalchemy import update

from db import engine

from models import Order

def bulk\_discount\_orders\_core(threshold: float, discount: float):

stmt = (

update(Order)

.where(Order.total > threshold)

.values(total=Order.total \* (1 - discount))

)

with engine.begin() as conn:

result = conn.execute(stmt)

print(f"Discounted {result.rowcount} orders (Core)")

if \_\_name\_\_ == "\_\_main\_\_":

bulk\_discount\_orders\_core(100.0, 0.10) # 10% off orders > $100

* SQL Server computes total \* (1 - discount) in place.

**5.4.2 ORM Bulk Update**

python

# bulk\_update\_orm.py

from session\_scope import session\_scope

from models import Order

def bulk\_discount\_orders\_orm(threshold: float, discount: float):

with session\_scope() as session:

orders = session.query(Order).filter(Order.total > threshold).all()

for order in orders:

order.total \*= (1 - discount)

print(f"Discounted {len(orders)} orders (ORM)")

if \_\_name\_\_ == "\_\_main\_\_":

bulk\_discount\_orders\_orm(50.0, 0.05) # 5% off orders > $50

* ORM approach loads objects, updates in Python, then flushes.

## 5.5 Deleting Single Records

**5.5.1 Core API**

python

# delete\_core.py

from sqlalchemy import delete

from db import engine

from models import User

def delete\_user\_core(user\_id: int):

stmt = delete(User).where(User.id == user\_id)

with engine.begin() as conn:

result = conn.execute(stmt)

print(f"Deleted {result.rowcount} user(s) (Core)")

if \_\_name\_\_ == "\_\_main\_\_":

delete\_user\_core(3)

**5.5.2 ORM API**

python

# delete\_orm.py

from session\_scope import session\_scope

from models import User

def delete\_user\_orm(user\_id: int):

with session\_scope() as session:

user = session.get(User, user\_id)

if user:

session.delete(user)

print(f"Deleted User (ORM): {user\_id}")

else:

print("User not found")

if \_\_name\_\_ == "\_\_main\_\_":

delete\_user\_orm(4)

## 5.6 Bulk Deletes

**5.6.1 Core Bulk Delete**

python

# bulk\_delete\_core.py

from sqlalchemy import delete

from db import engine

from models import Order

from datetime import datetime, timedelta

def delete\_old\_orders\_core(days: int):

cutoff = datetime.utcnow() - timedelta(days=days)

stmt = delete(Order).where(Order.created\_at < cutoff)

with engine.begin() as conn:

result = conn.execute(stmt)

print(f"Deleted {result.rowcount} orders older than {days} days (Core)")

if \_\_name\_\_ == "\_\_main\_\_":

delete\_old\_orders\_core(365)

**5.6.2 ORM Bulk Delete**

python

# bulk\_delete\_orm.py

from session\_scope import session\_scope

from models import Order

from datetime import datetime, timedelta

def delete\_old\_orders\_orm(days: int):

cutoff = datetime.utcnow() - timedelta(days=days)

with session\_scope() as session:

orders = (

session.query(Order)

.filter(Order.created\_at < cutoff)

.all()

)

for order in orders:

session.delete(order)

print(f"Deleted {len(orders)} orders (ORM)")

if \_\_name\_\_ == "\_\_main\_\_":

delete\_old\_orders\_orm(180)

## 5.7 Upserts (MERGE) in SQL Server

SQL Server supports MERGE for insert-or-update logic. Use Core’s text() or a custom DDL:

python

# upsert\_merge.py

from sqlalchemy import text

from db import engine

def upsert\_user(username: str, fullname: str):

merge\_sql = text("""

MERGE INTO dbo.users AS target

USING (VALUES (:username, :fullname))

AS src (username, fullname)

ON target.username = src.username

WHEN MATCHED THEN

UPDATE SET fullname = src.fullname

WHEN NOT MATCHED THEN

INSERT (username, fullname) VALUES (src.username, src.fullname);

""")

with engine.begin() as conn:

conn.execute(merge\_sql, {"username": username, "fullname": fullname})

print(f"Upserted user: {username}")

if \_\_name\_\_ == "\_\_main\_\_":

upsert\_user("alice", "Alice A.")

* The MERGE statement atomically updates or inserts.
* Parameters :username and :fullname are safely bound.

## 5.8 Using OUTPUT to Return Affected Rows

SQL Server’s OUTPUT clause lets you capture inserted/updated rows. Combine with Core:

python

# insert\_output.py

from sqlalchemy import insert, literal\_column

from db import engine

from models import User

from datetime import datetime

def insert\_user\_with\_output(username: str, fullname: str):

stmt = (

insert(User)

.values(username=username, fullname=fullname, created\_at=datetime.utcnow())

.returning(User.id, User.username)

)

with engine.begin() as conn:

for row in conn.execute(stmt):

print(f"OUTPUT: id={row.id}, username={row.username}")

if \_\_name\_\_ == "\_\_main\_\_":

insert\_user\_with\_output("greg", "Greg Green")

* .returning() instructs SQLAlchemy to append OUTPUT to the INSERT.
* Rows yielded include the requested columns.

# Chapter 6: Working with Relationships

In this chapter, you’ll learn how to model and navigate relationships between tables in SQL Server using SQLAlchemy’s ORM. We’ll cover one-to-many, many-to-one, many-to-many, self-referential mappings, and how to control loading and cascading behaviors.

## 6.1 Overview of Relationship Types

Relationships let you treat related rows as attributes on your Python objects. The main patterns are:

* One-to-Many / Many-to-One: e.g., a User has many Order objects.
* Many-to-Many: e.g., Post objects tagged with many Tag objects via an association table.
* Self-Referential: e.g., an Employee who reports to another Employee.

Each pattern involves:

* Foreign keys in the database schema
* relationship() declarations on your model classes
* Optional back\_populates or backref for bidirectional access

## 6.2 One-to-Many / Many-to-One Example

**6.2.1 Defining the Models**

In models.py, we already have User and Order. Extend them with relationship():

python

# models.py

from sqlalchemy import Column, ForeignKey, Integer, Float, DateTime, text

from sqlalchemy.orm import relationship

from db import Base

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True, autoincrement=True)

username = Column(String(50), nullable=False, unique=True, index=True)

# … other columns …

# one-to-many: a user has multiple orders

orders = relationship(

"Order",

back\_populates="user",

cascade="all, delete-orphan",

lazy="selectin"

)

class Order(Base):

\_\_tablename\_\_ = "orders"

id = Column(Integer, primary\_key=True, autoincrement=True)

user\_id = Column(Integer, ForeignKey("users.id"), nullable=False, index=True)

total = Column(Float, nullable=False)

created\_at = Column(DateTime, server\_default=text("GETDATE()"))

# many-to-one: each order references one user

user = relationship("User", back\_populates="orders", lazy="joined")

* ForeignKey("users.id") enforces referential integrity in SQL Server.
* back\_populates ties the two sides together.
* cascade="all, delete-orphan" ensures orders are deleted if their parent user is removed.
* lazy controls how related rows load (more in §6.6).

## 6.2.2 Creating Tables

Run:

bash

python create\_models.py

In SSMS, confirm **dbo.users** and **dbo.orders** exist with the user\_id FK constraint.

**6.2.3 Adding and Querying Related Objects**

python

# relationship\_example.py

from session\_scope import session\_scope

from models import User, Order

def create\_user\_with\_orders():

with session\_scope() as session:

u = User(username="emma", fullname="Emma Example")

u.orders = [

Order(total=120.5),

Order(total=75.0)

]

session.add(u)

print(f"Created {u} with orders: {u.orders}")

def fetch\_user\_orders(user\_id: int):

with session\_scope() as session:

u = session.get(User, user\_id)

print(f"User {u.username} has {len(u.orders)} orders:")

for o in u.orders:

print(f" Order {o.id} – ${o.total}")

if \_\_name\_\_ == "\_\_main\_\_":

create\_user\_with\_orders()

fetch\_user\_orders(1)

* Assigning to u.orders auto-sets order.user\_id.
* Accessing u.orders or o.user issues the necessary SELECTs per lazy policy.

## 6.3 Many-to-Many Relationships

**6.3.1 Association Table**

Define an association table in models.py:

python

from sqlalchemy import Table

post\_tags = Table(

"post\_tags",

Base.metadata,

Column("post\_id", ForeignKey("posts.id"), primary\_key=True),

Column("tag\_id", ForeignKey("tags.id"), primary\_key=True),

schema="dbo"

)

**6.3.2 Post and Tag Models**

python

class Post(Base):

\_\_tablename\_\_ = "posts"

id = Column(Integer, primary\_key=True, autoincrement=True)

title = Column(String(200), nullable=False)

content = Column(String, nullable=False)

tags = relationship(

"Tag",

secondary=post\_tags,

back\_populates="posts",

lazy="selectin"

)

class Tag(Base):

\_\_tablename\_\_ = "tags"

id = Column(Integer, primary\_key=True, autoincrement=True)

name = Column(String(50), nullable=False, unique=True)

posts = relationship(

"Post",

secondary=post\_tags,

back\_populates="tags",

lazy="selectin"

)

**6.3.3 Creating and Linking**

python

# many\_to\_many\_example.py

from session\_scope import session\_scope

from models import Post, Tag

def create\_post\_with\_tags():

with session\_scope() as session:

p = Post(title="SQLAlchemy 101", content="...")

t1 = Tag(name="python")

t2 = Tag(name="sqlalchemy")

p.tags = [t1, t2]

session.add(p)

print(f"Created Post '{p.title}' with tags {[t.name for t in p.tags]}")

def fetch\_posts\_by\_tag(tag\_name: str):

with session\_scope() as session:

tag = session.query(Tag).filter\_by(name=tag\_name).one()

for p in tag.posts:

print(f"Post: {p.title}")

if \_\_name\_\_ == "\_\_main\_\_":

create\_post\_with\_tags()

fetch\_posts\_by\_tag("python")

* Using secondary=post\_tags binds the two models through the association table.
* SQLAlchemy auto-inserts into **dbo.post\_tags**.

**6.4 Self-Referential Relationships**

Model an employee-manager hierarchy:

python

class Employee(Base):

\_\_tablename\_\_ = "employees"

id = Column(Integer, primary\_key=True)

name = Column(String(100), nullable=False)

manager\_id = Column(Integer, ForeignKey("employees.id"))

reports = relationship(

"Employee",

back\_populates="manager",

cascade="all, delete-orphan"

)

manager = relationship(

"Employee",

back\_populates="reports",

remote\_side=[id]

)

* remote\_side=[id] tells SQLAlchemy which column represents the remote parent.
* You can assemble a tree of Employee objects.

## 6.5 Relationship Loading Strategies

Control when and how SQLAlchemy retrieves related rows:

| **Strategy** | **Description** | **SQL Issued** |
| --- | --- | --- |
| lazy="select" | Load on first attribute access | One extra SELECT ... WHERE fk = ? per parent |
| lazy="joined" | Use SQL Server JOIN to load in same query | Single SELECT ... JOIN |
| lazy="selectin" | Two queries, one for parents, one for children | Efficient IN-based select |
| lazy="subquery" | Use subquery load | SELECT ... WHERE id IN (subquery) |

Example of joined loading:

python

u = session.query(User).options(joinedload(User.orders)).get(1)

## 6.6 Cascading and Orphans

The cascade argument on relationship() controls what happens when you add, delete, or detach objects:

* all, delete-orphan: delete child rows when parent is deleted
* save-update: default, propagates session.add()
* merge, expunge, refresh-expire: other available cascade rules

# Chapter 7: Migrations with Alembic

In this chapter, you’ll learn how to manage evolving SQL Server schemas alongside your Python code using Alembic. We’ll cover installation, configuration, generating and applying revisions, autogeneration of migrations, handling complex schema changes, and best practices to keep your database in sync with your models.

## 7.1 Installing and Configuring Alembic

1. Activate your virtual environment:

powershell

.\venv\Scripts\activate

1. Install Alembic:

powershell

pip install alembic

1. Initialize an Alembic directory in your project root:

bash

alembic init alembic

This creates:

* + an alembic folder containing env.py, script.py.mako, and a versions subfolder
  + an alembic.ini file for configuration

## 7.2 Configuring alembic.ini and env.py for SQL Server

**7.2.1 Updating alembic.ini**

Locate the [alembic] and [sqlalchemy] sections. Replace the SQLAlchemy URL placeholder with your Windows-authenticated ODBC connection:

ini

[alembic]

script\_location = alembic

[sqlalchemy]

# Use URL-style or odbc\_connect; here’s the odbc\_connect version:

url = mssql+pyodbc:///?odbc\_connect=Driver%3DODBC+Driver+17+for+SQL+Server%3BServer%3Dlocalhost%3BDatabase%3DTestDB%3BTrusted\_Connection%3Dyes%3B

Use urllib.parse.quote\_plus in Python to build the odbc\_connect string and then percent-encode it for alembic.ini.

**7.2.2 Editing env.py**

In alembic/env.py, modify the import and run context to target your Base.metadata:

python

from logging.config import fileConfig

from sqlalchemy import engine\_from\_config, pool

from alembic import context

# this imports your db setup and model metadata

import os, sys

sys.path.insert(0, os.path.dirname(os.path.dirname(\_\_file\_\_)))

from db import Base

from models import \* # ensure all models are imported

config = context.config

fileConfig(config.config\_file\_name)

target\_metadata = Base.metadata

def run\_migrations\_offline():

url = config.get\_main\_option("sqlalchemy.url")

context.configure(

url=url,

target\_metadata=target\_metadata,

literal\_binds=True,

dialect\_opts={"paramstyle": "pyformat"},

)

with context.begin\_transaction():

context.run\_migrations()

def run\_migrations\_online():

connectable = engine\_from\_config(

config.get\_section(config.config\_ini\_section),

prefix="sqlalchemy.",

poolclass=pool.NullPool,

)

with connectable.connect() as connection:

context.configure(

connection=connection,

target\_metadata=target\_metadata,

compare\_type=True, # detects column type changes

compare\_server\_default=True, # detects default changes

)

with context.begin\_transaction():

context.run\_migrations()

if context.is\_offline\_mode():

run\_migrations\_offline()

else:

run\_migrations\_online()

compare\_type=True and compare\_server\_default=True enable richer autogeneration.

## 7.3 Generating New Migrations

After you change or add a model:

1. **Autogenerate** a new revision:

bash

alembic revision --autogenerate -m "add orders table"

1. A new file appears in alembic/versions/ with upgrade() and downgrade() stubs populated.
2. **Review** the generated code to ensure it matches your intent and tweak if necessary.

## 7.4 Applying and Rolling Back Migrations

* To apply all pending migrations:

bash

alembic upgrade head

* To roll back to the previous revision:

bash

alembic downgrade -1

* To target a specific revision ID <rev\_id>:

bash

alembic upgrade <rev\_id>

Monitor the migration history in the alembic\_version table in SQL Server.

## 7.5 Revision Management and Branching

* **Naming revisions** with clear messages helps track changes (e.g., "add-user-email").
* **Branching**: if two feature branches generate migrations concurrently, you may need to merge them by creating a “merge” revision:

bash

alembic revision --message "merge heads" --head <head1> --splice --head <head2>

* **Stamping**: mark the database at a given revision without running migrations:

bash

alembic stamp head

## 7.6 Handling Complex Schema Changes

**7.6.1 Renaming Columns or Tables**

Alembic’s autogenerate can’t detect renames. Use op.alter\_column:

python

def upgrade():

op.alter\_column("users", "fullname", new\_column\_name="full\_name")

def downgrade():

op.alter\_column("users", "full\_name", new\_column\_name="fullname")

**7.6.2 Splitting or Merging Tables**

Manually write DDL in the migration:

python

from alembic import op

import sqlalchemy as sa

def upgrade():

# create new table

op.create\_table(

"profiles",

sa.Column("id", sa.Integer, primary\_key=True),

sa.Column("user\_id", sa.Integer, sa.ForeignKey("users.id")),

sa.Column("bio", sa.String(500)),

)

# migrate data

op.execute("""

INSERT INTO profiles (user\_id, bio)

SELECT id, ''

FROM users

""")

## 7.7 Best Practices and Tips

* **Version Control**: commit migration scripts alongside model changes.
* **Review Generated Code**: always inspect autogen scripts before applying.
* **Idempotency**: ensure upgrade() and downgrade() can run without side effects.
* **Testing**: apply migrations on a throwaway SQL Server database (e.g., a local Docker container) to validate.
* **Environments**: maintain separate branches or configs for development, staging, and production—stamp or upgrade accordingly.

# Chapter 8: Transaction Management and Concurrency

Understanding transactions and concurrency is critical for data integrity and performance in multi-user SQL Server environments. This chapter explores SQLAlchemy’s transaction APIs, isolation levels, savepoints, nested transactions, and patterns for optimistic and pessimistic concurrency control.

## 8.1 Basics of Transactions in SQLAlchemy

A **transaction** groups multiple operations so they either all succeed or all fail.

* **Begin/Commit/Rollback**

python

from sqlalchemy import create\_engine, text

engine = create\_engine("mssql+pyodbc:///?odbc\_connect=...")

with engine.begin() as conn:

conn.execute(text("UPDATE accounts SET balance = balance - 100 WHERE id=1"))

conn.execute(text("UPDATE accounts SET balance = balance + 100 WHERE id=2"))

* + engine.begin() opens a transaction.
  + Exiting the with block **commits** if no exception, **rolls back** on error.
* **Session Transactions (ORM)**

python

from db import SessionLocal

session = SessionLocal()

try:

user = session.get(User, 1)

user.fullname = "New Name"

session.commit()

except:

session.rollback()

raise

finally:

session.close()

* + commit() flushes and ends the transaction.
  + rollback() discards uncommitted changes.

## 8.2 Isolation Levels

Isolation levels determine how concurrent transactions interact.

* **READ COMMITTED (default SQL Server)** Prevents dirty reads; allows non-repeatable reads and phantom reads.
* **REPEATABLE READ** Prevents dirty and non-repeatable reads; phantom rows still possible.
* **SERIALIZABLE** Full isolation; prevents phantom reads by locking ranges.
* **SNAPSHOT** (SQL Server only) Uses row versioning to give consistent view without locking.

**8.2.1 Setting Isolation Level**

Use execution options on the engine or per-connection:

python

# Per-connection

with engine.connect().execution\_options(isolation\_level="SERIALIZABLE") as conn:

conn.execute(text("SELECT \* FROM users"))

python

# Engine-wide default

engine = create\_engine(

"...",

isolation\_level="SNAPSHOT"

)

## 8.3 Savepoints and Nested Transactions

Savepoints let you roll back part of a transaction without aborting it entirely.

python

from sqlalchemy import text

with engine.begin() as conn:

conn.execute(text("INSERT INTO logs (msg) VALUES ('start')"))

trans = conn.begin\_nested() # creates a savepoint

try:

conn.execute(text("INSERT INTO logs (msg) VALUES ('inner')"))

raise ValueError("Something went wrong")

trans.commit()

except Exception:

trans.rollback() # only undoes the inner block

conn.execute(text("INSERT INTO logs (msg) VALUES ('end')"))

* begin\_nested() issues a SQL Server SAVE TRANSACTION.
* On rollback(), only to the savepoint, outer transaction continues.

**8.4 Pessimistic Concurrency Control (Row Locking)**

Use SELECT FOR UPDATE style locks to prevent other transactions from modifying selected rows:

python

from sqlalchemy import select

from sqlalchemy.orm import Session

with Session(engine) as session:

stmt = select(User).where(User.id==1).with\_for\_update()

user = session.execute(stmt).scalar\_one()

user.fullname = "Locked Update"

session.commit()

* SQLAlchemy issues SELECT ... WITH (UPDLOCK, ROWLOCK) by default on SQL Server.
* You can customize locking hints:

python

stmt = stmt.with\_for\_update(read=False, of=User, nowait=True)

* + nowait=True adds NOWAIT so SQL Server raises immediately if lock cannot be acquired.

## 8.5 Optimistic Concurrency Control (Version Counters)

Optimistic locking detects conflicts only at commit time, using a version counter column:

1. **Add a version column** to your model:

python

from sqlalchemy import Column, Integer

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True)

fullname= Column(String(120))

version = Column(Integer, nullable=False, default=1)

\_\_mapper\_args\_\_ = {

"version\_id\_col": version

}

1. **SQLAlchemy** auto-includes version in the WHERE clause of updates:

sql

UPDATE users

SET fullname = ?, version = version + 1

WHERE id = ? AND version = ?

If no rows are affected, a StaleDataError is raised, indicating someone else modified the row.

## 8.6 Two-Phase (Distributed) Transactions

When you need atomic commits across multiple databases or resources, use two-phase commit:

python

from sqlalchemy import create\_engine

eng1 = create\_engine("mssql+pyodbc:///DB1?...")

eng2 = create\_engine("mssql+pyodbc:///DB2?...")

from sqlalchemy import Session

session1 = Session(eng1)

session2 = Session(eng2)

try:

session1.begin\_twophase("txn1")

session2.begin\_twophase("txn2")

session1.add(User(username="x", fullname="X"))

session2.execute(text("UPDATE accounts SET balance = balance - 50 WHERE id=1"))

session1.prepare()

session2.prepare()

session1.commit\_prepared()

session2.commit\_prepared()

except:

session1.rollback()

session2.rollback()

raise

finally:

session1.close()

session2.close()

* Each session phases through **prepare** and **commit\_prepared**.
* Ensures either all commit or all roll back across resources.

## 8.7 Best Practices

* **Keep transactions short**: hold locks briefly to reduce contention.
* **Select appropriate isolation levels**: use SNAPSHOT to avoid blocking readers.
* **Use optimistic locking** for high-concurrency low-conflict scenarios.
* **Avoid manual SQL** in transactions when ORM handles change detection.
* **Monitor deadlocks** in SQL Server and implement retry logic on deadlock errors (error code 1205).

# Chapter 9: Performance Tuning and Bulk Operations

In this chapter, you’ll optimize SQL Server access and handle large-scale data operations with SQLAlchemy. We cover index strategies, measuring query cost, engine tuning, bulk inserts/updates/deletes, streaming results, compiled queries, and using SQL Server-specific features like fast\_executemany and Table-Valued Parameters.

## 9.1 Understanding SQL Server Indexing

Indexes speed lookups and joins but slow writes.

* **Clustered vs Nonclustered**
  + Clustered: table rows stored in key order (one per table).
  + Nonclustered: separate structure with row pointers.
* **Creating an index via Alembic**

python

from alembic import op

import sqlalchemy as sa

def upgrade():

op.create\_index(

"ix\_orders\_user\_total",

"orders",

["user\_id", "total"],

unique=False

)

* **ORM model index**

python

class Order(Base):

\_\_tablename\_\_ = "orders"

id = Column(Integer, primary\_key=True)

user\_id = Column(Integer, ForeignKey("users.id"), index=True)

total = Column(Float, index=True)

# …

* **When to index**
  + Columns in WHERE, JOIN, ORDER BY, GROUP BY.
  + Avoid indexing highly volatile columns or tables with heavy writes without clear benefit.

## 9.2 Measuring Query Performance

**9.2.1 SET STATISTICS IO / TIME**

Use raw SQL to see logical reads and CPU/time:

python

from sqlalchemy import text

with engine.connect() as conn:

conn.execute(text("SET STATISTICS IO ON; SET STATISTICS TIME ON;"))

result = conn.execute(text("SELECT \* FROM orders WHERE total > 1000;")).fetchall()

# Check your console for statistics messages

## 9.2.2 Execution Plan

Retrieve the estimated plan:

python

with engine.connect() as conn:

plan = conn.execute(text("SET SHOWPLAN\_XML ON; SELECT \* FROM orders;")).scalar\_one()

with open("plan.xml", "wb") as f:

f.write(plan.encode("utf-8"))

Open plan.xml in SSMS to visualize.

## 9.3 Connection Pool and Engine Configuration

Tweak the SQLAlchemy engine for concurrency:

python

from sqlalchemy import create\_engine

from urllib.parse import quote\_plus

odbc\_str = quote\_plus(

"Driver=ODBC Driver 17 for SQL Server;"

"Server=localhost;"

"Database=TestDB;"

"Trusted\_Connection=yes;"

)

engine = create\_engine(

f"mssql+pyodbc:///?odbc\_connect={odbc\_str}",

echo=False,

pool\_size=10, # number of persistent connections

max\_overflow=20, # extra connections beyond pool\_size

pool\_timeout=30, # seconds to wait for a connection

pool\_pre\_ping=True, # test connections before use

)

* **pool\_pre\_ping** avoids “stale” connections.
* Adjust **pool\_size**/​**max\_overflow** to match your application load.

**9.4 Fast Bulk Inserts with pyodbc.fast\_executemany**

By default, pyodbc inserts row-by-row. Enabling fast\_executemany uses array API for high-speed loads:

python

import pyodbc

from sqlalchemy import event

from db import engine

# Hook into connection creation

@event.listens\_for(engine, "before\_cursor\_execute")

def receive\_before\_cursor\_execute(conn, cursor, statement, parameters, context, executemany):

if executemany:

cursor.fast\_executemany = True

def bulk\_insert\_fast(users\_data):

from models import User

stmt = insert(User)

with engine.begin() as conn:

conn.execute(stmt, users\_data) # executemany=True triggers fast\_executemany

# Example usage

users = [

{"username": f"user{i}", "fullname": f"User {i}", "created\_at": datetime.utcnow()}

for i in range(10000)

]

bulk\_insert\_fast(users)

* Place the listener in your app startup.
* Gains of 10–100× speed improvement on large batches.

## 9.5 Bulk Updates and Deletes in Batches

Large UPDATEs/DELETEs can lock tables. Break into chunks:

python

from sqlalchemy import select, update, delete

from models import Order

BATCH\_SIZE = 1000

def bulk\_delete\_old\_orders(days):

cutoff = datetime.utcnow() - timedelta(days=days)

while True:

stmt = (

delete(Order)

.where(Order.created\_at < cutoff)

.limit(BATCH\_SIZE)

.returning(Order.id)

)

with engine.begin() as conn:

deleted = conn.execute(stmt).rowcount

if deleted < BATCH\_SIZE:

break

* Using .limit() ensures each transaction touches only a subset.
* For updates, apply the same pattern with update().where(...).limit(...).

## 9.6 Streaming Large Result Sets

Pulling millions of rows at once can exhaust memory. Use yield\_per and stream\_results:

python

from sqlalchemy import select

from sqlalchemy.orm import Session

def stream\_orders(batch\_size=500):

session = Session(engine)

stmt = select(Order).order\_by(Order.id)

# stream\_results keeps server cursor open

result = session.execute(stmt.execution\_options(stream\_results=True)).scalars()

for order in result.yield\_per(batch\_size):

process(order)

session.close()

* stream\_results=True uses server-side cursors.
* yield\_per(n) fetches n rows at a time.

## 9.7 Compiled Queries and Statement Caching

Prepare frequently used statements once to reduce overhead:

python

from sqlalchemy import select

from sqlalchemy.sql import compiles

from models import Order, User

stmt = select(User, Order).join(Order)

# Cache the compiled SQL once

compiled = stmt.compile(engine)

def get\_user\_orders():

with engine.connect() as conn:

rs = conn.execute(compiled)

return rs.fetchall()

SQLAlchemy 2.0’s ORM also caches compiled queries by default when using Session.execute() with a static select().

## 9.8 Using Table-Valued Parameters via Stored Procedures

When loading complex data sets, SQL Server Table-Valued Parameters (TVPs) are ideal:

1. **Create a TVP and Proc** in SQL Server:

sql

CREATE TYPE dbo.UserType AS TABLE (

username NVARCHAR(50),

fullname NVARCHAR(120)

);

GO

CREATE PROCEDURE dbo.usp\_InsertUsers

@users dbo.UserType READONLY

AS

BEGIN

INSERT INTO dbo.users (username, fullname)

SELECT username, fullname FROM @users;

END

1. **Call from Python**:

python

import pyodbc

from db import engine

import pandas as pd

def bulk\_insert\_via\_tvp(df: pd.DataFrame):

# raw pyodbc to leverage TVP

with engine.raw\_connection() as conn:

cursor = conn.cursor()

# Create TVP as list of tuples

tvp = [tuple(row) for row in df[["username", "fullname"]].itertuples(index=False)]

# prepare TVP parameter

cursor.setinputsizes([("dbo.UserType", tvp)])

cursor.execute("{CALL dbo.usp\_InsertUsers(?)}", tvp)

conn.commit()

* TVPs push set-based operations into SQL Server, minimizing round-trips.

## 9.9 Best Practices Recap

* Profile queries early with **STATISTICS IO/TIME** and execution plans.
* Index wisely: balance read performance with write overhead.
* Tune your connection pool to match concurrency.
* Use **fast\_executemany** for bulk inserts.
* Batch large updates/deletes to avoid long-held locks.
* Stream large result sets using yield\_per and stream\_results.
* Leverage compiled queries and caching to reduce overhead.
* Consider TVPs or SSIS for complex bulk data loading.

# Chapter 9: Performance Tuning and Bulk Operations

In this chapter, you’ll optimize SQL Server access and handle large-scale data operations with SQLAlchemy. We cover index strategies, measuring query cost, engine tuning, bulk inserts/updates/deletes, streaming results, compiled queries, and using SQL Server-specific features like fast\_executemany and Table-Valued Parameters.

## 9.1 Understanding SQL Server Indexing

Indexes speed lookups and joins but slow writes.

* **Clustered vs Nonclustered**
  + Clustered: table rows stored in key order (one per table).
  + Nonclustered: separate structure with row pointers.
* **Creating an index via Alembic**

python

from alembic import op

import sqlalchemy as sa

def upgrade():

op.create\_index(

"ix\_orders\_user\_total",

"orders",

["user\_id", "total"],

unique=False

)

* **ORM model index**

python

class Order(Base):

\_\_tablename\_\_ = "orders"

id = Column(Integer, primary\_key=True)

user\_id = Column(Integer, ForeignKey("users.id"), index=True)

total = Column(Float, index=True)

# …

* **When to index**
  + Columns in WHERE, JOIN, ORDER BY, GROUP BY.
  + Avoid indexing highly volatile columns or tables with heavy writes without clear benefit.

## 9.2 Measuring Query Performance

**9.2.1 SET STATISTICS IO / TIME**

Use raw SQL to see logical reads and CPU/time:

python

from sqlalchemy import text

with engine.connect() as conn:

conn.execute(text("SET STATISTICS IO ON; SET STATISTICS TIME ON;"))

result = conn.execute(text("SELECT \* FROM orders WHERE total > 1000;")).fetchall()

# Check your console for statistics messages

**9.2.2 Execution Plan**

Retrieve the estimated plan:

python

with engine.connect() as conn:

plan = conn.execute(text("SET SHOWPLAN\_XML ON; SELECT \* FROM orders;")).scalar\_one()

with open("plan.xml", "wb") as f:

f.write(plan.encode("utf-8"))

Open plan.xml in SSMS to visualize.

## 9.3 Connection Pool and Engine Configuration

Tweak the SQLAlchemy engine for concurrency:

python

from sqlalchemy import create\_engine

from urllib.parse import quote\_plus

odbc\_str = quote\_plus(

"Driver=ODBC Driver 17 for SQL Server;"

"Server=localhost;"

"Database=TestDB;"

"Trusted\_Connection=yes;"

)

engine = create\_engine(

f"mssql+pyodbc:///?odbc\_connect={odbc\_str}",

echo=False,

pool\_size=10, # number of persistent connections

max\_overflow=20, # extra connections beyond pool\_size

pool\_timeout=30, # seconds to wait for a connection

pool\_pre\_ping=True, # test connections before use

)

* **pool\_pre\_ping** avoids “stale” connections.
* Adjust **pool\_size**/​**max\_overflow** to match your application load.

## 9.4 Fast Bulk Inserts with pyodbc.fast\_executemany

By default, pyodbc inserts row-by-row. Enabling fast\_executemany uses array API for high-speed loads:

python

import pyodbc

from sqlalchemy import event

from db import engine

# Hook into connection creation

@event.listens\_for(engine, "before\_cursor\_execute")

def receive\_before\_cursor\_execute(conn, cursor, statement, parameters, context, executemany):

if executemany:

cursor.fast\_executemany = True

def bulk\_insert\_fast(users\_data):

from models import User

stmt = insert(User)

with engine.begin() as conn:

conn.execute(stmt, users\_data) # executemany=True triggers fast\_executemany

# Example usage

users = [

{"username": f"user{i}", "fullname": f"User {i}", "created\_at": datetime.utcnow()}

for i in range(10000)

]

bulk\_insert\_fast(users)

* Place the listener in your app startup.
* Gains of 10–100× speed improvement on large batches.

## 9.5 Bulk Updates and Deletes in Batches

Large UPDATEs/DELETEs can lock tables. Break into chunks:

python

from sqlalchemy import select, update, delete

from models import Order

BATCH\_SIZE = 1000

def bulk\_delete\_old\_orders(days):

cutoff = datetime.utcnow() - timedelta(days=days)

while True:

stmt = (

delete(Order)

.where(Order.created\_at < cutoff)

.limit(BATCH\_SIZE)

.returning(Order.id)

)

with engine.begin() as conn:

deleted = conn.execute(stmt).rowcount

if deleted < BATCH\_SIZE:

break

* Using .limit() ensures each transaction touches only a subset.
* For updates, apply the same pattern with update().where(...).limit(...).

**9.6 Streaming Large Result Sets**

Pulling millions of rows at once can exhaust memory. Use yield\_per and stream\_results:

python

from sqlalchemy import select

from sqlalchemy.orm import Session

def stream\_orders(batch\_size=500):

session = Session(engine)

stmt = select(Order).order\_by(Order.id)

# stream\_results keeps server cursor open

result = session.execute(stmt.execution\_options(stream\_results=True)).scalars()

for order in result.yield\_per(batch\_size):

process(order)

session.close()

* stream\_results=True uses server-side cursors.
* yield\_per(n) fetches n rows at a time.

## 9.7 Compiled Queries and Statement Caching

Prepare frequently used statements once to reduce overhead:

python

from sqlalchemy import select

from sqlalchemy.sql import compiles

from models import Order, User

stmt = select(User, Order).join(Order)

# Cache the compiled SQL once

compiled = stmt.compile(engine)

def get\_user\_orders():

with engine.connect() as conn:

rs = conn.execute(compiled)

return rs.fetchall()

SQLAlchemy 2.0’s ORM also caches compiled queries by default when using Session.execute() with a static select().

## 9.8 Using Table-Valued Parameters via Stored Procedures

When loading complex data sets, SQL Server Table-Valued Parameters (TVPs) are ideal:

1. **Create a TVP and Proc** in SQL Server:

sql

CREATE TYPE dbo.UserType AS TABLE (

username NVARCHAR(50),

fullname NVARCHAR(120)

);

GO

CREATE PROCEDURE dbo.usp\_InsertUsers

@users dbo.UserType READONLY

AS

BEGIN

INSERT INTO dbo.users (username, fullname)

SELECT username, fullname FROM @users;

END

1. **Call from Python**:

python

import pyodbc

from db import engine

import pandas as pd

def bulk\_insert\_via\_tvp(df: pd.DataFrame):

# raw pyodbc to leverage TVP

with engine.raw\_connection() as conn:

cursor = conn.cursor()

# Create TVP as list of tuples

tvp = [tuple(row) for row in df[["username", "fullname"]].itertuples(index=False)]

# prepare TVP parameter

cursor.setinputsizes([("dbo.UserType", tvp)])

cursor.execute("{CALL dbo.usp\_InsertUsers(?)}", tvp)

conn.commit()

* TVPs push set-based operations into SQL Server, minimizing round-trips.

## 9.9 Best Practices Recap

* Profile queries early with **STATISTICS IO/TIME** and execution plans.
* Index wisely: balance read performance with write overhead.
* Tune your connection pool to match concurrency.
* Use **fast\_executemany** for bulk inserts.
* Batch large updates/deletes to avoid long-held locks.
* Stream large result sets using yield\_per and stream\_results.
* Leverage compiled queries and caching to reduce overhead.
* Consider TVPs or SSIS for complex bulk data loading.

# Chapter 11: Testing Strategies and Fixtures

In this chapter, you’ll learn how to write reliable, isolated tests for your SQLAlchemy–powered SQL Server data layer. We’ll cover types of tests, fixture patterns using pytest, transactional rollbacks, and example test cases.

## 11.1 Types of Tests

* Unit Tests Test individual functions or methods in isolation. For data layers, pure unit tests often require heavy mocking of sessions and queries, which can be brittle and offer limited value.
* Functional/Integration Tests Exercise real database interactions—creating tables, inserting and querying rows—to catch schema mismatches and SQL errors early. These tests balance realism with speed and reliability.

## 11.2 Isolated Test Databases

To prevent tests from interfering:

* **Dedicated Test Database** Create a separate SQL Server database (e.g., TestDB) for your CI pipeline or local test runs. Use credentials with permission only on that database.
* **In-Memory or Disposable Instances** While SQL Server lacks a true in-memory engine, you can spin up throwaway instances (e.g., Docker) or use ephemeral databases dropped after tests run.
* **Test Fixtures Package** Tools like **db-testtools** provide fixtures and resources to bring up isolated databases and sessions, ensuring tests don’t interfere with each other.

## 11.3 Using pytest Fixtures for Engine and Session

Leverage pytest to manage setup and teardown. Create a conftest.py in your tests folder:

python

# conftest.py

import pytest

from urllib.parse import quote\_plus

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

from db import Base

# Build ODBC URL for TestDB

odbc\_str = quote\_plus(

"Driver=ODBC Driver 17 for SQL Server;"

"Server=localhost;"

"Database=TestDB;"

"Trusted\_Connection=yes;"

)

@pytest.fixture(scope="session")

def engine():

url = f"mssql+pyodbc:///?odbc\_connect={odbc\_str}"

e = create\_engine(url, echo=False)

Base.metadata.create\_all(e) # create tables once

yield e

Base.metadata.drop\_all(e) # clean up after all tests

@pytest.fixture(scope="function")

def session(engine):

# begin a SAVEPOINT-backed transaction

connection = engine.connect()

transaction = connection.begin()

SessionLocal = sessionmaker(bind=connection)

sess = SessionLocal()

yield sess

sess.close()

transaction.rollback() # undo changes

connection.close()

* scope="session" builds the schema once per test run.
* scope="function" ensures each test runs in its own transaction, rolled back at the end.

## 11.4 Example: Testing CRUD Operations

**11.4.1 Test: Create and Read User**

python

# tests/test\_user\_model.py

from models import User

def test\_create\_user(session):

user = User(username="tester", fullname="Test User")

session.add(user)

session.commit()

assert user.id is not None

fetched = session.get(User, user.id)

assert fetched.username == "tester"

**11.4.2 Test: Unique Constraint Violation**

python

import pytest

from models import User

from sqlalchemy.exc import IntegrityError

def test\_username\_unique(session):

u1 = User(username="dup", fullname="Dup One")

u2 = User(username="dup", fullname="Dup Two")

session.add\_all([u1, u2])

with pytest.raises(IntegrityError):

session.commit()

Because each test runs in its own rollback-backed transaction, data from one test never persists to the next.

## 11.5 Advanced Fixtures and Factories

* **Factory Boy** or **Model Bakery** Use factories to generate valid model instances with sensible defaults, reducing boilerplate in tests.
* **Parameterized Tests** Combine pytest’s @pytest.mark.parametrize with fixtures to test multiple scenarios succinctly.
* **Session Scoping** For tests that need nested transactions (e.g., testing savepoints), you can nest session.begin\_nested() within your function-scoped session fixture.

## 11.6 Continuous Integration Considerations

* **Environment Setup** Ensure your CI environment has SQL Server available (via service or container) and that TestDB exists with proper permissions.
* **Test Isolation** Use unique database names per job or clear the schema between builds to avoid cross-job contamination.
* **Speed vs Fidelity** Balance the number of integration tests against suite run time. Mock pure business logic but preserve realistic DB tests for critical paths.

# Chapter 12: Real-World Case Studies

This chapter presents practical applications of Python + SQLAlchemy with SQL Server across diverse domains. Each case study highlights an actual challenge, a step-by-step solution, and concrete code you can adapt for your own projects.

## 12.1 Case Study: ETL Data Warehouse Load

**Scenario:** You need to extract daily sales data from a staging database, transform it, and bulk-load it into a star-schema data warehouse. You also want to log each run’s status.

**12.1.1 Strategy**

* Use SQLAlchemy Core for high-speed, set-based operations.
* Enable fast\_executemany on pyodbc for multi-row inserts.
* Wrap each phase in a transaction; record success/failure in a log table.

**12.1.2 Key Tables**

| **Table** | **Purpose** |
| --- | --- |
| staging.sales\_tmp | Incoming raw sales data |
| warehouse.fact\_sales | Star-schema fact table |
| etl.log\_runs | Records each ETL run’s timing & status |

**12.1.3 Code Example**

python

# etl\_load.py

from datetime import datetime

from sqlalchemy import (

create\_engine, MetaData, Table, select, insert, text

)

from sqlalchemy import event

from urllib.parse import quote\_plus

# Build engine with fast\_executemany

odbc = quote\_plus(

"Driver=ODBC Driver 17 for SQL Server;"

"Server=localhost;Database=TestDB;"

"Trusted\_Connection=yes;"

)

engine = create\_engine(f"mssql+pyodbc:///?odbc\_connect={odbc}", echo=False)

@event.listens\_for(engine, "before\_cursor\_execute")

def \_enable\_fast(cursor, stmt, params, context, executemany):

if executemany:

cursor.fast\_executemany = True

# Reflect tables

meta = MetaData(bind=engine)

stg = Table("sales\_tmp", meta, autoload\_with=engine, schema="staging")

fact = Table("fact\_sales", meta, autoload\_with=engine, schema="warehouse")

runs = Table("log\_runs", meta, autoload\_with=engine, schema="etl")

def run\_etl():

run\_id = None

with engine.begin() as conn:

# 1. Start log

result = conn.execute(

insert(runs).values(start\_time=datetime.utcnow(), status="RUNNING")

)

run\_id = result.inserted\_primary\_key[0]

# 2. Extract + transform

rows = conn.execute(select(stg)).mappings().all()

transformed = [

{

"product\_id": r["product\_id"],

"quantity": r["qty"] \* 1, # simple transform

"sale\_date": r["sale\_dt"].date()

}

for r in rows

]

# 3. Bulk-load into fact\_sales

conn.execute(insert(fact), transformed)

# 4. Mark success

conn.execute(

runs.update().where(runs.c.id == run\_id)

.values(end\_time=datetime.utcnow(), status="SUCCESS")

)

if \_\_name\_\_ == "\_\_main\_\_":

run\_etl()

print("ETL run completed.")

## 12.2 Case Study: Web Dashboard with Dynamic Filters

**Scenario:** Build a FastAPI dashboard that lets users filter orders by date range, user, and minimum total. You need safe parameter binding, pagination, and model-to-dict serialization.

**12.2.1 API Endpoint**

python

# app.py

from fastapi import FastAPI, Query

from sqlalchemy import select, and\_

from sqlalchemy.orm import Session

from db import engine, Base, SessionLocal

from models import User, Order

app = FastAPI()

@app.get("/orders/")

def list\_orders(

user\_id: int | None = None,

date\_from: str | None = None,

total\_min: float = 0.0,

page: int = 1,

page\_size: int = 20

):

with SessionLocal() as session:

stmt = select(Order).join(User)

conditions = []

if user\_id:

conditions.append(Order.user\_id == user\_id)

if date\_from:

conditions.append(Order.created\_at >= date\_from)

if total\_min:

conditions.append(Order.total >= total\_min)

if conditions:

stmt = stmt.where(and\_(\*conditions))

stmt = stmt.order\_by(Order.created\_at.desc())

stmt = stmt.offset((page-1)\*page\_size).limit(page\_size)

orders = session.execute(stmt).scalars().all()

return [o.\_\_dict\_\_ for o in orders]

* **Parameter binding** prevents SQL injection.
* **Pagination** uses offset()/limit().
* Converting ORM objects to dicts for JSON output.

## 12.3 Case Study: Audit Logging All SQL Executions

**Scenario:** You must capture every SQL statement run by your application into an audit table for compliance.

**12.3.1 Audit Table**

sql

CREATE TABLE audit.sql\_history (

id INT IDENTITY PRIMARY KEY,

timestamp DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME(),

statement NVARCHAR(MAX),

parameters NVARCHAR(MAX)

);

**12.3.2 Event Listener**

python

# audit.py

import json

from sqlalchemy import event

from db import engine

@event.listens\_for(engine, "before\_execute")

def log\_query(conn, clauseelement, multiparams, params, execution\_options):

stmt\_text = str(clauseelement)

audit = {

"statement": stmt\_text,

"parameters": json.dumps(multiparams or params)

}

conn.execute(

"INSERT INTO audit.sql\_history (statement, parameters) VALUES (:statement, :parameters)",

audit

)

* Fires **before\_execute** on every SQLAlchemy call.
* Logs raw SQL plus bound parameters.

## 12.4 Case Study: Real-Time Chat App with FastAPI & Socket.IO

**Scenario:** You’re building a chat service where messages are persisted in SQL Server, and users receive them in real time.

**12.4.1 Models**

python

# models.py (append)

class Message(Base):

\_\_tablename\_\_ = "messages"

id = Column(Integer, primary\_key=True)

user\_id = Column(Integer, ForeignKey("users.id"), nullable=False)

content = Column(String(500), nullable=False)

sent\_at = Column(DateTime, server\_default=text("SYSUTCDATETIME()"))

user = relationship("User")

**12.4.2 FastAPI with Socket.IO**

python

# chat\_app.py

from fastapi import FastAPI

from fastapi\_socketio import SocketManager

from sqlalchemy.orm import Session

from db import engine, SessionLocal

from models import Message

app = FastAPI()

sio = SocketManager(app=app)

@sio.on("send\_message")

async def handle\_message(sid, data):

user\_id = data["user\_id"]

content = data["content"]

# Persist

with SessionLocal() as session:

msg = Message(user\_id=user\_id, content=content)

session.add(msg)

session.commit()

session.refresh(msg)

# Broadcast

await sio.emit("new\_message", msg.\_\_dict\_\_)

* Messages saved synchronously, then emitted asynchronously.
* Clients subscribe to "new\_message" for live updates.

## 12.5 Case Study: Tkinter Desktop App for Data Entry

**Scenario:** A lightweight GUI for non-technical staff to CRUD customer records in SQL Server.

**12.5.1 GUI Layout**

* Entry widgets for customer fields (name, email, join date).
* Buttons: **Create**, **Update**, **Delete**, **Refresh List**.
* A Listbox showing current records.

**12.5.2 Core Integration**

python

# gui\_app.py

import tkinter as tk

from models import User

from session\_scope import session\_scope

class CustomerApp:

def \_\_init\_\_(self, root):

self.root = root

root.title("Customer Manager")

# ... build widgets ...

self.refresh\_list()

def refresh\_list(self):

with session\_scope() as session:

self.customers = session.query(User).all()

self.listbox.delete(0, tk.END)

for u in self.customers:

self.listbox.insert(tk.END, f"{u.id}: {u.fullname}")

def create\_customer(self):

name = self.name\_var.get()

with session\_scope() as session:

session.add(User(username=name.lower(), fullname=name))

self.refresh\_list()

# ... similarly for update and delete ...

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

app = CustomerApp(root)

root.mainloop()

* Each CRUD button calls a session-scoped function.
* GUI remains responsive as each DB call is brief.

# Appendix A: Connection Strings and Drivers

This appendix summarizes the most common approaches to connecting SQLAlchemy to SQL Server on your local machine. You’ll find ready-to-use connection strings, driver options, and tips for both Windows and SQL authentication scenarios.

## A.1 ODBC Driver 17 for SQL Server

* **Installation**
  + Download and install the Microsoft ODBC Driver 17 for SQL Server from Microsoft’s site.
  + Confirm installation via “ODBC Data Sources (64-bit)” in Windows.
* **Driver Name**
* ODBC Driver 17 for SQL Server
* **Connection String Patterns**

python

# URL style, Windows Authentication, default instance

conn\_str = (

r"mssql+pyodbc://"

r"@localhost/master"

r"?driver=ODBC+Driver+17+for+SQL+Server"

r"&trusted\_connection=yes"

)

python

# URL style, SQL Authentication, named instance on custom port

conn\_str = (

r"mssql+pyodbc://"

r"sa:YourPassword@MYBOX\SQL2022,1435/YourDB"

r"?driver=ODBC+Driver+17+for+SQL+Server"

)

python

# ODBC-connect style for complex DSNs or percent-encoding

from urllib.parse import quote\_plus

params = quote\_plus(

"Driver=ODBC Driver 17 for SQL Server;"

"Server=localhost,1433;"

"Database=YourDB;"

"UID=sa;PWD=YourPassword;"

)

engine = create\_engine(f"mssql+pyodbc:///?odbc\_connect={params}")

## A.2 pymssql Driver (Alternative)

* **Installation**

bash

pip install pymssql

* **Pros**
  + Simplified connection string syntax.
  + No need to install separate ODBC driver.
* **Cons**
  + Less actively maintained; may lag on T-SQL features.
  + Bulk-insert performance lower than pyodbc’s fast\_executemany.
* **Example Connection**

python

engine = create\_engine(

"mssql+pymssql://sa:YourPassword@localhost:1433/YourDB",

echo=True

)

## A.3 Driver Comparison

| **Driver** | **Auth Methods** | **Installation Overhead** | **Bulk Speed** | **Notes** |
| --- | --- | --- | --- | --- |
| pyodbc | Windows, SQL | ODBC Driver install required | Very fast with fast\_executemany | Best all-around support for SQL Server |
| pymssql | SQL only | pip install pymssql | Moderate | Easy setup, but development has slowed |
| aioodbc\* | Windows, SQL | ODBC Driver + pip install aioodbc | N/A (async wrapper) | Enables AsyncEngine patterns (experimental) |

\\* Requires additional configuration and may not support all SQL Server features.

## A.4 Choosing the Right Pattern

* **URL-style** is concise for simple use cases; watch out for proper URL-escaping of backslashes and reserved characters.
* **ODBC-connect-style** via quote\_plus avoids manual escaping and is preferred when you have many parameters (e.g., encryption, timeout).
* **Environment Variables** Store credentials and connection details in environment variables to avoid hard-coding secrets in your code.

With these patterns and tables, you can select and configure the optimal driver and connection string for your SQL Server integrations.

# Appendix B: Common Error Messages and Troubleshooting

This appendix lists frequent errors you may encounter when using Python, SQLAlchemy, and SQL Server, explains their causes, and offers concrete steps to resolve them.

## B.1 Connection and Authentication Errors

| **Error Message** | **Cause** | **Troubleshooting Steps** |
| --- | --- | --- |
| Login failed for user 'HOST\username'. (18456)<br>Cannot open database "X" requested by the login. (4060) | Your Windows account isn’t mapped to the target database<br>Or the database name is wrong | 1. In SSMS, verify the exact database name under **Databases**.<br>2. In **Security → Logins**, map HOST\username to that database and grant **db\_datareader**/**db\_datawriter** roles.<br>3. Retry with the correct database name in the connection string. |
| pyodbc.InterfaceError: ('IM002', '[IM002] [Microsoft][ODBC Driver Manager] Data source name not found...') | ODBC driver not installed or misspelled driver name | 1. Install “ODBC Driver 17 for SQL Server” from Microsoft.<br>2. In your connection string, use exactly ODBC Driver 17 for SQL Server (case-sensitive). |
| AttributeError: ‘str’ object has no attribute 'dialect' | Misformatted connection URL passed to create\_engine | 1. Ensure you pass a proper SQLAlchemy URL or odbc\_connect param, not a raw string.<br>2. For complex strings, use from urllib.parse import quote\_plus and the ?odbc\_connect= style. |
| PowerShell PSSecurityException UnauthorizedAccess | PowerShell execution policy blocking script execution | 1. Run PowerShell as Administrator.<br>2. Execute Set-ExecutionPolicy RemoteSigned -Scope CurrentUser.<br>3. Always wrap script paths in quotes when invoking with &. |

## B.2 SQLAlchemy Core and ORM Errors

| **Error Message** | **Cause** | **Troubleshooting Steps** |
| --- | --- | --- |
| sqlalchemy.exc.IntegrityError: (mssql, ...) UNIQUE KEY... violation | Attempted to insert or update a value that violates a unique constraint | 1. Wrap your session.commit() in a try/except IntegrityError block.<br>2. Inspect the offending value and adjust your logic to avoid duplicates or handle conflicts (e.g., upsert). |
| sqlalchemy.exc.InvalidRequestError: Object '<User at 0x…>' is already attached to session '1' | Mixing objects from different sessions | 1. Do not share ORM objects across sessions.<br>2. Use session.merge(obj) to reattach an object to the current session. |
| sqlalchemy.exc.InvalidRequestError: Table 'X' is not bound to an Engine or Connection | Called insert() or metadata.create\_all() before binding | 1. Ensure you created your engine before importing models.<br>2. In db.py, create the engine first, then declare Base = declarative\_base(), then import models. |
| AttributeError: 'Row' object has no attribute 'some\_column' | Accessing columns incorrectly from a Core select() result | 1. Use row['column\_name'] or unpack with for id, name in result:.<br>2. Or switch to ORM querying to get attribute access (session.query(Model)...). |

## B.3 Query and Transaction Errors

| **Error Message** | **Cause** | **Troubleshooting Steps** |
| --- | --- | --- |
| sqlalchemy.exc.TimeoutError: QueuePool limit of size X overflow Y reached, connection timed out | All connections in pool are in use | 1. Increase pool\_size/max\_overflow in create\_engine.<br>2. Use pool\_pre\_ping=True to recycle stale connections.<br>3. Close sessions promptly with session.close(). |
| sqlalchemy.exc.OperationalError: (mssql (1205)) Deadlock victim | Two transactions deadlocked each other | 1. Catch error code 1205 and retry the transaction.<br>2. Keep transactions short.<br>3. Use SNAPSHOT isolation or row-level locks only when needed. |
| Invalid transaction state; no transaction is active | Calling session.rollback() or `commit()`` outside of an active transaction | 1. Use the session\_scope() context manager to guarantee proper transaction boundaries.<br>2. Do not manually mix engine.begin() with Session operations on the same connection. |

## B.4 Alembic Migration Errors

| **Error Message** | **Cause** | **Troubleshooting Steps** |
| --- | --- | --- |
| ModuleNotFoundError: No module named 'models' | env.py can’t import your model definitions | 1. In alembic/env.py, adjust sys.path.insert(0, ...) so the project root is on the path.<br>2. Verify your package/module layout matches the import statements. |
| Revision X failed because: Target database is not up to date | Autogenerate detected out-of-sync references | 1. Run alembic stamp head to match your code state if you’ve manually applied changes.<br>2. Inspect the migration script for conflicting operations and resolve manually. |
| Autogenerate does not detect COLUMN TYPE CHANGE | Alembic’s default compare\_type is off | 1. In env.py, set context.configure(compare\_type=True).<br>2. Use op.alter\_column() in your revision to explicitly change types when needed. |

## B.5 General Troubleshooting Tips

* **Verify in SSMS First**: Before coding, connect with SQL Server Management Studio, run SELECT name FROM sys.databases; and check your table schemas.
* **Echo SQL**: Use create\_engine(..., echo=True) during development to inspect generated T-SQL.
* **Check Driver Versions**: Mismatched ODBC driver and SQL Server versions can cause cryptic errors; upgrade to the latest Microsoft driver.
* **Network and Firewall**: For remote servers, ensure TCP/IP is enabled in SQL Server Configuration Manager and that firewalls allow port 1433 (or your custom port).
* **Logging**: Configure Python’s logging to capture SQLAlchemy debug messages for deeper insights:

python

import logging

logging.basicConfig()

logging.getLogger("sqlalchemy.engine").setLevel(logging.INFO)

* **Isolation**: When tests or local scripts fail unpredictably, drop and recreate the database or use a fresh test database to rule out data corruption.

With these error explanations and solutions at your fingertips, you can swiftly diagnose and fix common issues as you build your Python + SQLAlchemy applications on SQL Server.