Advanced SQLAlchemy Features and Methods

---

1. Eager Loading (Optimizing Queries with `joinedload`)

By default, SQLAlchemy uses **lazy loading**, meaning related objects are not loaded until explicitly accessed. However, in certain cases, you want to optimize by loading everything in one query. This is where **eager loading** comes in with the `joinedload` method.

Example: Eager Load `Student` and `Courses` (One-to-Many)

```
python
Copy code
from sqlalchemy.orm import joinedload

# Eager load the courses when querying students
students_with_courses =
session.query(Student).options(joinedload(Student.courses)).all()

for student in students_with_courses:
    print(f"Student: {student.name}")
    for course in student.courses:
        print(f"  Course: {course.course_name}")
```

- **Eager Loading** prevents multiple round-trips to the database by loading all related data at once.

---

2. Bulk Insert (Efficiently Insert Multiple Records)

When inserting a large number of records, using `session.add_all()` or `session.add()` for each record can be inefficient. SQLAlchemy provides **bulk insert** methods for improved performance.

Example: Bulk Insert Multiple Students

```
python
Copy code
students = [
    {"name": "George", "age": 20, "grade": "A"},
    {"name": "Helen", "age": 22, "grade": "B"},
    {"name": "Isaac", "age": 21, "grade": "C"},
]

# Bulk insert using SQLAlchemy's built-in method
session.bulk_insert_mappings(Student, students)
session.commit()
```

- **Bulk Insert** is useful for inserting a large number of records efficiently.

3. Advanced Querying (Filtering, Joins, and Aggregates)

SQLAlchemy supports complex queries similar to SQL, such as filtering, joins, and aggregate functions.

Example 1: Filtering with Multiple Conditions

```
python
Copy code
# Query students who are older than 21 and have a grade of 'A'
students = session.query(Student).filter(Student.age > 21, Student.grade ==
'A').all()

for student in students:
    print(student)
```

Example 2: Joining Tables

```
python
Copy code
# Join Students and Courses
result = session.query(Student.name,
Course.course_name).join(Student.courses).all()

for student_name, course_name in result:
    print(f"{student_name} is enrolled in {course_name}")
```

Example 3: Aggregating with COUNT and GROUP BY

```
python
Copy code
# Count how many students are enrolled in each course
course_counts = session.query(Course.course_name,
func.count(Student.id)).join(Student.courses).group_by(Course.course_name).
all()

for course_name, count in course_counts:
    print(f"{course_name}: {count} students")
```

- **Advanced querying** allows you to build complex queries similar to raw SQL but with the ORM's safety and convenience.

---

4. Transaction Management (Commit, Rollback, and Savepoint)

Transactions are crucial for maintaining data integrity. SQLAlchemy supports transactions via `session.commit()` and `session.rollback()`. For more complex cases, you can also create **savepoints**.

Example: Basic Transaction with Commit and Rollback

```
python
Copy code
try:
    new_student = Student(name='Jason', age=24, grade='A')
    session.add(new_student)
    session.commit()  # Commit the transaction
except:
    session.rollback()  # Roll back if an error occurs
    print("Transaction rolled back.")
```

Example: Using Savepoints

```
python
Copy code
from sqlalchemy import exc

# Start a new transaction
session.begin()

# Add the first student and create a savepoint
session.add(Student(name='Kevin', age=23, grade='B'))
session.flush()  # Flush the transaction to the database
savepoint = session.begin_nested()  # Create a savepoint

try:
    # Add another student that could cause an error
    session.add(Student(name='Invalid Student', age=-1, grade='C'))
    session.flush()  # Force the error here
except exc.SQLAlchemyError:
    print("Rolling back to savepoint.")
    savepoint.rollback()  # Rollback to the savepoint (does not affect
Kevin's addition)

session.commit()  # Commit the transaction after resolving the issue
```

---

5. Composite Primary Keys (Multi-Column Primary Keys)

In some tables, you may need to define a composite primary key (using more than one column to uniquely identify a row).

Example: Composite Primary Key for Enrollment

```
python
Copy code
class Enrollment(Base):
    __tablename__ = 'enrollments'

    student_id = Column(Integer, ForeignKey('students.id'),
primary_key=True)
    course_id = Column(Integer, ForeignKey('courses.id'), primary_key=True)
    enrollment_date = Column(String)

    student = relationship("Student", back_populates="enrollments")
    course = relationship("Course", back_populates="enrollments")
```

- Here, the **`student_id`** and **`course_id`** together form a composite primary key for the `Enrollment` table.

6. Inheritance (Single Table and Multi-Table)

SQLAlchemy supports **inheritance**, which allows different classes to be mapped to a single table or multiple tables.

Example: Single Table Inheritance

```
python
Copy code
class Person(Base):
    __tablename__ = 'persons'

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

    __mapper_args__ = {
        'polymorphic_identity': 'person',
        'polymorphic_on': type
    }

class Teacher(Person):
    __mapper_args__ = {
        'polymorphic_identity': 'teacher'
    }

class Student(Person):
    __mapper_args__ = {
        'polymorphic_identity': 'student'
    }
```

- **Single Table Inheritance** stores all data in one table and uses a `type` column to distinguish between different classes.

---

7. Custom SQL (Executing Raw SQL Queries)

Sometimes, you may need to run raw SQL queries directly.

Example: Executing Raw SQL

```
python
Copy code
# Executing raw SQL queries
result = session.execute("SELECT * FROM students WHERE age > :age", {'age':
20})

for row in result:
    print(row)
```

- This is useful for complex queries that are difficult to express with the ORM.

8. Declarative Mixins

SQLAlchemy supports **mixins** to share common logic or fields across models.

Example: Timestamp Mixin

```
python
Copy code
from datetime import datetime

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

class Student(TimestampMixin, Base):
    __tablename__ = 'students'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)
    grade = Column(String)
```

- **Mixins** allow you to add common fields like `created_at` and `updated_at` to multiple models.

---

9. Database Events and Triggers

You can define **event listeners** or **triggers** in SQLAlchemy to automatically run code when certain events occur, like inserts or updates.

Example: Listening for Insert Events

```
python
Copy code
from sqlalchemy import event

def before_insert_listener(mapper, connection, target):
    print(f"About to insert {target.name}")

# Attaching the event listener to the Student class
event.listen(Student, 'before_insert', before_insert_listener)
```

- This event will be triggered before a `Student` is inserted.