[SQLite and SQLAlchemy] [cheatsheet]

1. Connecting to the Database

- Import SQLAlchemy: `from sqlalchemy import create_engine`
- Create an SQLite database engine: `engine = create_engine('sqlite:///database.db')`
- Create a connection: `connection = engine.connect()`
- Create a session: `from sqlalchemy.orm import sessionmaker; Session = sessionmaker(bind=engine); session = Session()`

2. Creating Tables

- Define a table using SQLAlchemy declarative base: `from sqlalchemy.ext.declarative import declarative_base; Base = declarative_base()`
- Define a table class: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); name = Column(String(50)); age = Column(Integer)`
- Create tables: `Base.metadata.create_all(engine)`

3. Inserting Data

- Insert a single record: `user = User(name='John', age=25); session.add(user); session.commit()`
- Insert multiple records: `users = [User(name='Alice', age=30), User(name='Bob', age=35)]; session.add_all(users); session.commit()`
- Insert records using a dictionary: `user_data = {'name': 'Charlie', 'age': 40}; user = User(**user_data); session.add(user); session.commit()`

4. Querying Data

- Query all records: `users = session.query(User).all()`
- Query records with a filter: `users = session.query(User).filter(User.age > 30).all()`
- Query records with multiple filters: `users = session.query(User).filter(User.age > 30, User.name.like('%ali%')).all()`
- Query records with an IN clause: `users = session.query(User).filter(User.name.in_(['Alice', 'Bob'])).all()`

By: Waleed Mousa

- Query records with an OR condition: `from sqlalchemy import or_; users = session.query(User).filter(or_(User.age < 30, User.name == 'Alice')).all()`
- Query records with an AND condition: `from sqlalchemy import and_; users = session.query(User).filter(and_(User.age > 30, User.name.like('%ali%'))).all()`
- Query records with an ORDER BY clause: `users = session.query(User).order_by(User.age).all()`
- Query records with a LIMIT clause: `users = session.query(User).limit(5).all()`
- Query records with an OFFSET clause: `users = session.query(User).offset(5).limit(5).all()`
- Query distinct records: `distinct_names = session.query(User.name).distinct().all()`
- Query records with aggregate functions: `from sqlalchemy import func; max_age = session.query(func.max(User.age)).scalar()`
- Query records with a GROUP BY clause: `from sqlalchemy import func; age_groups = session.query(User.age, func.count(User.id)).group_by(User.age).all()`
- Query records with a HAVING clause: `from sqlalchemy import func; age_groups = session.query(User.age, func.count(User.id)).group_by(User.age).having(func.count(User.id) > 1).all()`
- Query records with a subquery: `subquery = session.query(User.age).filter(User.name == 'Alice').subquery(); users = session.query(User).filter(User.age.in_(subquery)).all()`
- Query records with a join: `from sqlalchemy import join; users_orders = session.query(User, Order).join(Order, User.id == Order.user_id).all()`
- Query records with a left join: `from sqlalchemy import outerjoin; users_orders = session.query(User, Order).outerjoin(Order, User.id == Order.user_id).all()`
- Query records with a self join: `managers = session.query(Employee).join(Employee, Employee.manager_id == Employee.id).all()`
- Query records with a UNION: `from sqlalchemy import union; combined_results = session.query(User.name).union(session.query(Employee.name)).all()`
- Query records with an INTERSECT: `from sqlalchemy import intersect; common_results = session.query(User.name).intersect(session.query(Employee.name)).all()`

By: Waleed Mousa

- Query records with an EXCEPT: `from sqlalchemy import except_; unique_results = session.query(User.name).except_(session.query(Employee.name)).all()`

5. Updating Data

- Update a single record: `user = session.query(User).filter(User.id == 1).first(); user.name = 'Updated Name'; session.commit()`
- Update multiple records: `session.query(User).filter(User.age < 30).update({User.age: 30}); session.commit()`
- Update records with a correlated update: `from sqlalchemy import update; correlated_update = update(User).where(User.id == Address.user_id).values(address='New Address'); session.execute(correlated_update)`

6. Deleting Data

- Delete a single record: `user = session.query(User).filter(User.id == 1).first(); session.delete(user); session.commit()`
- Delete multiple records: `session.query(User).filter(User.age < 30).delete(); session.commit()`
- Delete all records: `session.query(User).delete(); session.commit()`
- Delete records with a correlated delete: `from sqlalchemy import delete; correlated_delete = delete(User).where(User.id == Address.user_id); session.execute(correlated_delete)`

7. Transactions

- Begin a transaction: `transaction = session.begin()`
- Commit a transaction: `transaction.commit()`
- Rollback a transaction: `transaction.rollback()`
- Use a context manager for transactions: `with session.begin(): # Perform database operations`

8. Advanced Querying

- Query records with a CASE statement: `from sqlalchemy import case; users = session.query(User, case([(User.age < 30, 'Young'), (User.age >= 30, 'Old')], else_='Unknown').label('age_group')).all()`
- Query records with a CAST: `from sqlalchemy import cast; users = session.query(User, cast(User.age, String)).all()`

By: Waleed Mousa

- Query records with a COALESCE: `from sqlalchemy import func; users = session.query(User, func.coalesce(User.name, 'Unknown')).all()`
- Query records with a NULLIF: `from sqlalchemy import func; users = session.query(User, func.nullif(User.name, 'John')).all()`
- Query records with a GREATEST: `from sqlalchemy import func; users = session.query(User, func.greatest(User.age, 30)).all()`
- Query records with a LEAST: `from sqlalchemy import func; users = session.query(User, func.least(User.age, 30)).all()`
- Query records with a BETWEEN: `users = session.query(User).filter(User.age.between(25, 35)).all()`
- Query records with a LIKE: `users = session.query(User).filter(User.name.like('%ali%')).all()`
- Query records with an ILIKE (case-insensitive): `users = session.query(User).filter(User.name.ilike('%ali%')).all()`
- Query records with a REGEXP: `users = session.query(User).filter(User.name.op('regexp')('^J')).all()`
- Query records with an EXISTS: `from sqlalchemy import exists; users = session.query(User).filter(exists().where(User.age > 30)).all()`
- Query records with a NOT EXISTS: `from sqlalchemy import exists; users = session.query(User).filter(~exists().where(User.age > 30)).all()`
- Query records with a subquery in WHERE clause: `subquery = session.query(User.id).filter(User.name == 'Alice').subquery(); users = session.query(User).filter(User.id.in_(subquery)).all()`
- Query records with a subquery in FROM clause: `subquery = session.query(User.name, func.count(User.id).label('count')).group_by(User.name).subquery(); user_counts = session.query(subquery.c.name, subquery.c.count).all()`

9. Relationships

- Define a one-to-many relationship: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); orders = relationship('Order', back_populates='user')`
- Define a many-to-one relationship: `class Order(Base): __tablename__ = 'orders'; id = Column(Integer, primary_key=True); user_id = Column(Integer, ForeignKey('users.id')); user = relationship('User', back_populates='orders')`
- Define a one-to-one relationship: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); profile = relationship('Profile', uselist=False, back_populates='user')`

- Define a many-to-many relationship: `association_table = Table('association', Base.metadata, Column('user_id', Integer, ForeignKey('users.id')), Column('group_id', Integer, ForeignKey('groups.id'))); class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); groups = relationship('Group', secondary=association_table, back_populates='users')`
- Eager loading relationships: `users = session.query(User).options(joinedload(User.orders)).all()`
- Lazy loading relationships: `users = session.query(User).options(lazyload(User.orders)).all()`
- Joining relationships: `users_orders = session.query(User, Order).join(User.orders).all()`
- Filtering by related entities: `users = session.query(User).join(User.orders).filter(Order.status == 'pending').all()`

10. Advanced Relationship Querying

- Querying with a relationship EXISTS: `users = session.query(User).filter(User.orders.any()).all()`
- Querying with a relationship NOT EXISTS: `users = session.query(User).filter(~User.orders.any()).all()`
- Querying with a relationship HAS: `users = session.query(User).filter(User.orders.any(Order.status == 'pending')).all()`
- Querying with a relationship HAS NOT: `users = session.query(User).filter(User.orders.any(Order.status != 'pending')).all()`
- Querying with a relationship COUNT: `users = session.query(User, func.count(Order.id).label('order_count')).join(User.orders).group_by(User.id).all()`
- Querying with a relationship SUM: `users = session.query(User, func.sum(Order.total).label('total_spent')).join(User.orders).group_by(User.id).all()`
- Querying with a relationship AVG: `users = session.query(User, func.avg(Order.total).label('average_spent')).join(User.orders).group_by(User.id).all()`
- Querying with a relationship MIN: `users = session.query(User, func.min(Order.total).label('min_spent')).join(User.orders).group_by(User.id).all()`

By: Waleed Mousa

- Querying with a relationship MAX: `users = session.query(User, func.max(Order.total).label('max_spent')).join(User.orders).group_by(User.id).all()`

11. Advanced Relationship Updating and Deleting

- Updating related entities: `user = session.query(User).filter(User.id == 1).first(); user.orders[0].status = 'shipped'; session.commit()`
- Deleting related entities: `user = session.query(User).filter(User.id == 1).first(); session.delete(user.orders[0]); session.commit()`
- Cascading deletes: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); orders = relationship('Order', back_populates='user', cascade='all, delete-orphan')`
- Cascading updates: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); orders = relationship('Order', back_populates='user', cascade='save-update')`

12. Indexes and Constraints

- Create a unique constraint: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); email = Column(String(50), unique=True)`
- Create a check constraint: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); age = Column(Integer, CheckConstraint('age >= 18'))`
- Create an index: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); name = Column(String(50), index=True)`
- Create a composite index: `Index('idx_user_email_age', User.email, User.age)`
- Create a unique index: `Index('idx_user_email', User.email, unique=True)`
- Create a partial index: `Index('idx_user_email', User.email, postgresql_where=(User.age > 18))`
- Create a functional index: `Index('idx_user_lower_email', func.lower(User.email))`

13. Advanced Schema Operations

- Add a new column to a table: `session.execute(AddColumn('users', Column('new_column', String(50))))`
- Rename a column: `session.execute(RenameColumn('users', 'old_column', 'new_column'))`

By: Waleed Mousa

- Drop a column from a table: `session.execute(DropColumn('users', 'column_to_drop'))`
- Add a foreign key constraint: `session.execute(AddConstraint(ForeignKeyConstraint(['user_id'], ['users.id'])))`
- Drop a constraint: `session.execute(DropConstraint(ForeignKeyConstraint(['user_id'], ['users.id'])))`
- Create a new table: `session.execute(CreateTable('new_table', Column('id', Integer, primary_key=True), Column('name', String(50))))`
- Rename a table: `session.execute(RenameTable('old_table', 'new_table'))`
- Drop a table: `session.execute(DropTable('table_to_drop'))`

14. Alembic Migrations

- Install Alembic: `pip install alembic`
- Initialize Alembic: `alembic init alembic`
- Create a migration script: `alembic revision --autogenerate -m "Add a new column"`
- Apply migrations: `alembic upgrade head`
- Downgrade migrations: `alembic downgrade base`
- Show current revision: `alembic current`
- Show migration history: `alembic history`

15. Advanced SQLAlchemy Features

- Use a hybrid property: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); first_name = Column(String(50)); last_name = Column(String(50)); @hybrid_property def full_name(self): return f'{self.first_name} {self.last_name}'`
- Use a hybrid method: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); age = Column(Integer); @hybrid_method def is_adult(self): return self.age >= 18`
- Use a composite foreign key: `class OrderItem(Base): __tablename__ = 'order_items'; id = Column(Integer, primary_key=True); order_id = Column(Integer); user_id = Column(Integer); ForeignKeyConstraint(['order_id', 'user_id'], ['orders.order_id', 'orders.user_id'])`
- Use a synonymous column: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); name = Column(String(50)); fullname = synonym('name')`

By: Waleed Mousa

- Use a deferred column: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); name = deferred(Column(String(50)))`
- Use a column_property: `class User(Base): __tablename__ = 'users'; id = Column(Integer, primary_key=True); first_name = Column(String(50)); last_name = Column(String(50)); full_name = column_property(first_name + ' ' + last_name)`

16. Performance Optimization

- Use connection pooling: `from sqlalchemy.pool import QueuePool; engine = create_engine('sqlite:///database.db', poolclass=QueuePool)`
- Use result set caching: `from sqlalchemy.orm import Query; query = session.query(User).options(Query.with_expression_cache())`
- Use eager loading: `users = session.query(User).options(joinedload(User.orders)).all()`
- Use lazy loading: `users = session.query(User).options(lazyload(User.orders)).all()`
- Use subquery loading: `users = session.query(User).options(subqueryload(User.orders)).all()`
- Use selectin loading: `users = session.query(User).options(selectinload(User.orders)).all()`
- Use batch processing: `session.query(User).yield_per(100)`
- Use bulk inserts: `session.bulk_insert_mappings(User, [{'name': 'John'}, {'name': 'Alice'}])`
- Use bulk updates: `session.bulk_update_mappings(User, [{'id': 1, 'name': 'Updated John'}, {'id': 2, 'name': 'Updated Alice'}])`
- Use parameterized queries: `users = session.query(User).filter(User.name == bindparam('name')).params(name='John').all()`

17. Testing and Debugging

- Use in-memory SQLite database for testing: `engine = create_engine('sqlite:///:memory:')`
- Use a separate database for testing: `engine = create_engine('sqlite:///test_database.db')`
- Use SQLAlchemy's echo mode for logging: `engine = create_engine('sqlite:///database.db', echo=True)`
- Use Python's logging module for SQLAlchemy logging: `import logging; logging.basicConfig(); logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)`

By: Waleed Mousa

- Use pytest for testing: `def test_user_creation(): user = User(name='John'); assert user.name == 'John'`
- Use SQLAlchemy's `text()` function for raw SQL queries: `result = session.execute(text('SELECT * FROM users')).fetchall()`