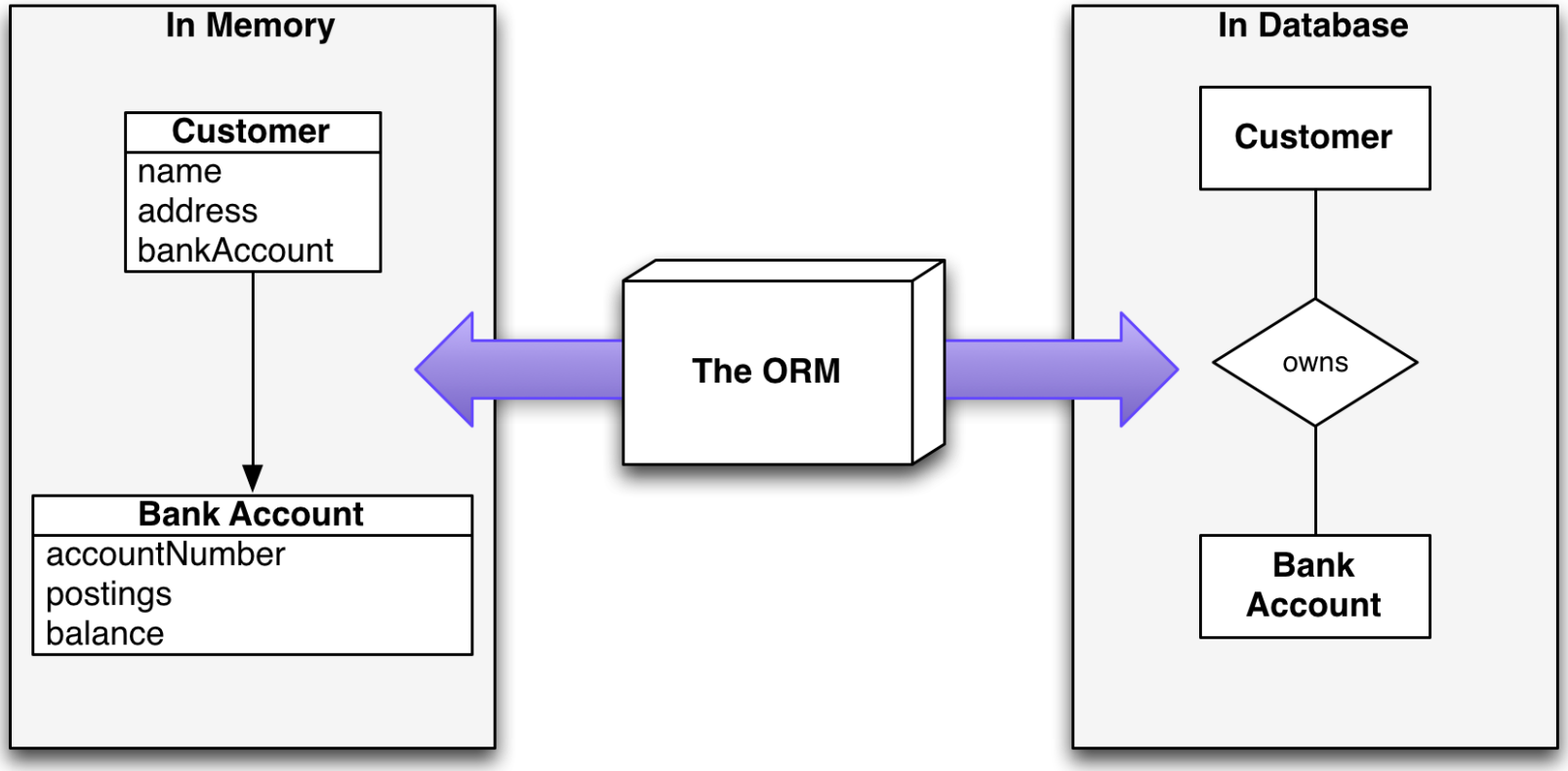
2.1. ORM 소개



■ What is ORM?

○ ORM stands Object Relational Mapping

○ Programming technique for converting data between incompatible type systems using object-oriented programming languages

■ Why use ORM?

○ Mismatch between the object model and the relational database

- ➢ RDBSs represent data in tabular format
- ➢ Object-Oriented languages represent data as an interconnected graph of objects

○ ORM frees the programmer from dealing with simple repetitive database queries

○ Automatically mapping the database to business objects

○ Programmers focus more on business problems and less with data storage

○ The mapping process can aid in data verification and security before reaching the database

○ ORM can provide a caching layer above the database

■ Disadvantages

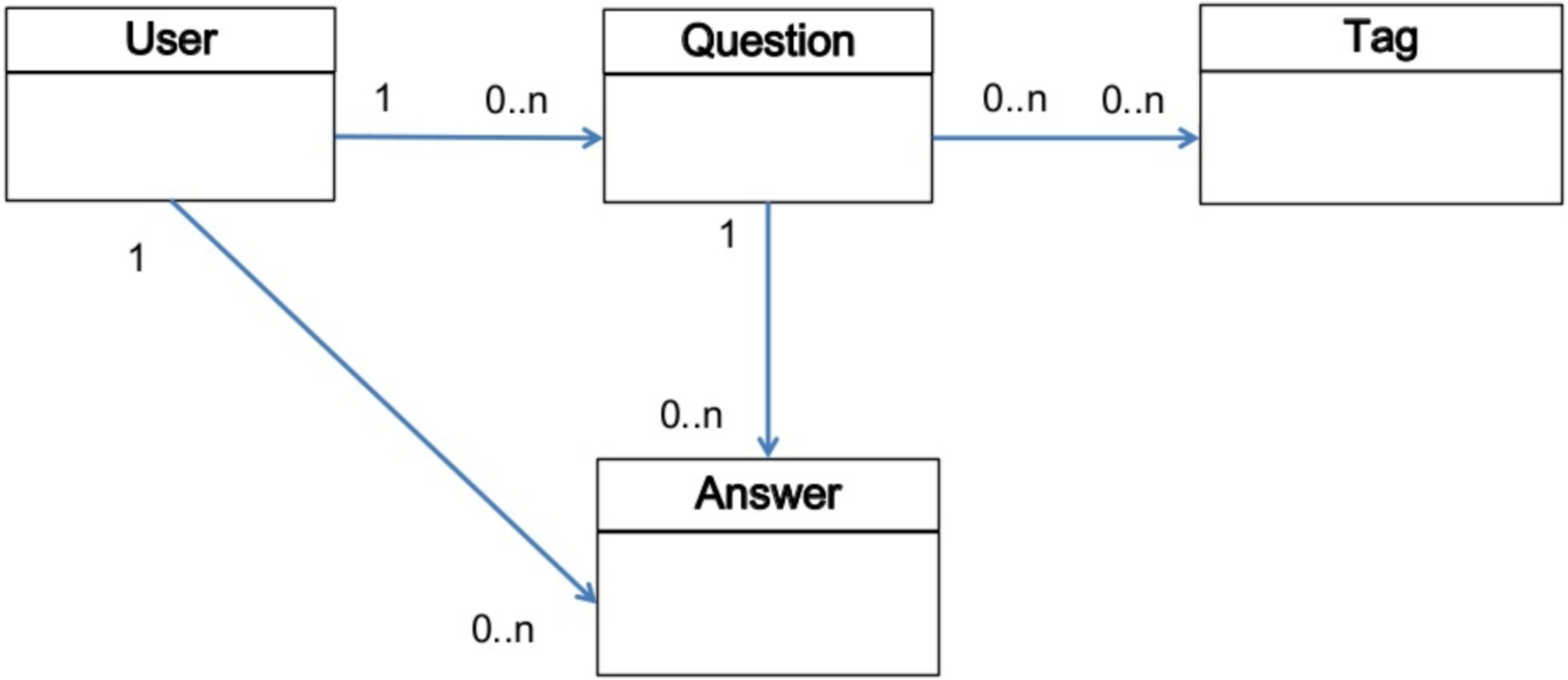
○ Potentially increasing processing overhead

■ 예제

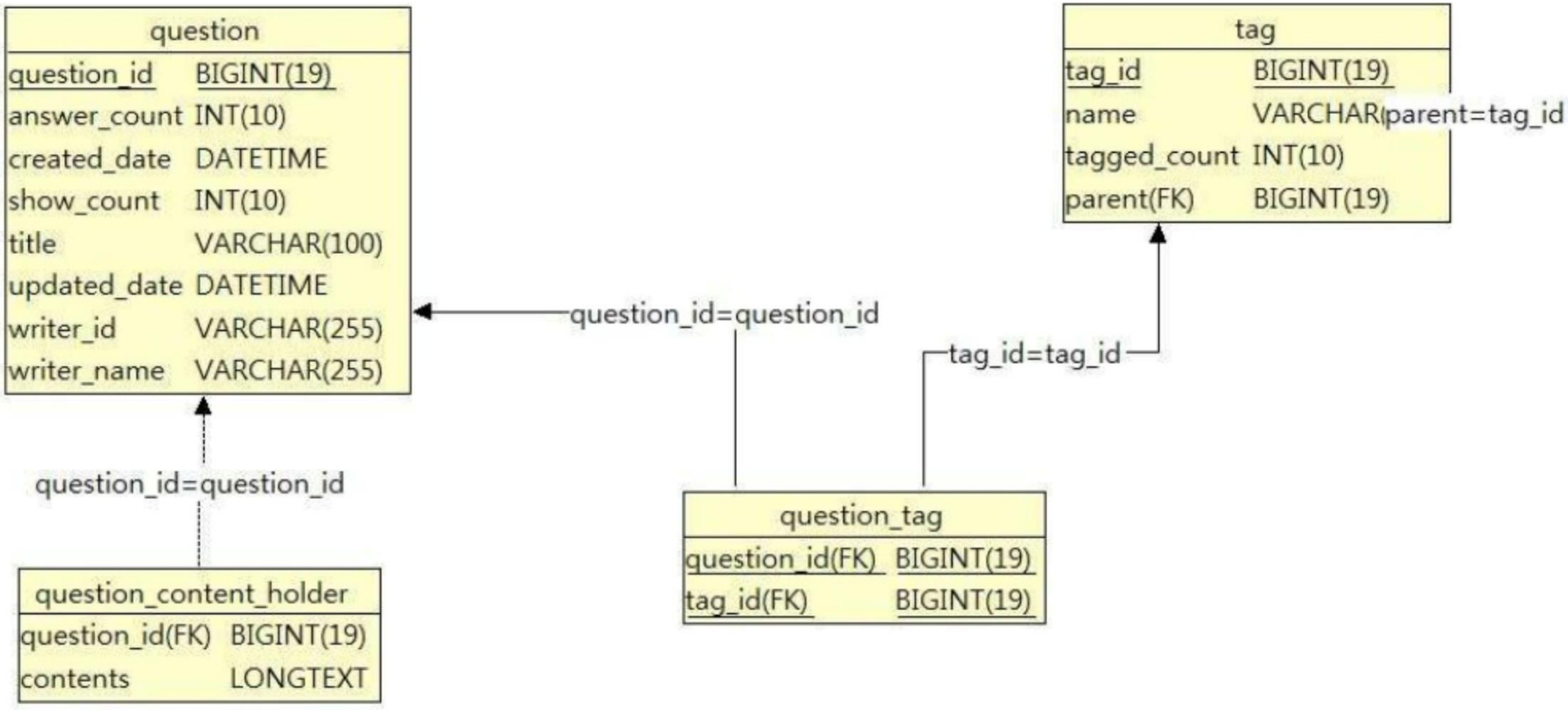
○ 요구사항

- 사용자는 질문 할 수 있어야 한다
- 질문에 대한 답변을 할 수 있어야 한다
- 질문할 때 태그를 추가할 수 있어야 한다
- 태그는 태그 풀에 존재하는 태그만 추가할 수 있다
- 태그가 추가될 경우 해당 태그 수는 +1 증가, 삭제될 경우 해당 태그 수는 -1 감소되어야 한다

○ 개체-관계 모델(ER Model)



○ 데이터베이스 설계


question
question_id BIGINT(19)
answer_count INT(10)
created_date DATETIME
show_count INT(10)
title VARCHAR(100)
updated_date DATETIME
writer_id VARCHAR(255)
writer_name VARCHAR(255)
tag
tag_id BIGINT(19)
name VARCHAR
tagged_count INT(10)
parent(FK) BIGINT(19)
parent=tag_id
question_id=question_id
tag_id=tag_id
question_id=question_id
question_tag
question_id(FK) BIGINT(19)
tag_id(FK) BIGINT(19)
question_content_holder
question_id(FK) BIGINT(19)
contents LONGTEXT

○ 질문 추가 시

- 질문 등록
- 태그 등록
  - 태그 풀에서 해당 태그 ID 가져오기
  - 태그 풀에서 해당 태그 수 증감하기

```
INSERT INTO question VALUES(?, ?, ?, ?, ?); → question_id = 1

SELECT tag_id, name FROM tag WHERE name="python"; → tag_id = 1
SELECT tag_id, name FROM tag WHERE name="alchemy"; → tag_id = 2

INSERT INTO question_tag VALUES(1, 1);
INSERT INTO question_tag VALUES(1, 2);

UPDATE tag SET tagged_count = tagged_count + 1 where name="python";
UPDATE tag SET tagged_count = tagged_count + 1 where name="alchemy";
```

○ 질문 수정 시

- 질문 수정
- 태그 수정
  - 태그 풀에서 해당 태그 ID 가져오기
  - 태그 풀에서 해당 태그 수 증감하기

```
UPDATE question SET title=?, contents=? WHERE question_id = 1;

SELECT tag_id, name FROM tag WHERE name="python"; → tag_id = 1
SELECT tag_id, name FROM tag WHERE name="alchemy"; → tag_id = 2
SELECT tag_id, name FROM tag WHERE name="orm"; → tag_id = 3

INSERT INTO question_tag VALUES(1, 3);
DELETE FROM question_tag where question_id=1 and tag_id=2;

UPDATE tag SET tagged_count = tagged_count - 1 where name="alchemy"
UPDATE tag SET tagged_count = tagged_count + 1 where name="orm"
```

*테이블 간의 관계보다 **데이터베이스 대한 처리**에 집중*

○ 객체-관계로 접근

```
public class Question {
    private long qid;
    private string title;
    [ … ]

    public processTags(string tag) {
    }
}
```

```
public class Tag {
    private long tid;
    private string tag;
    [ … ]

    public add(string tag) {
    }
}
```

INSERT INTO question VALUES(?, ?, ?, ?, ?); → question(1)
SELECT tag_id, name FROM tag WHERE name="python"; → tag.getByName("python")

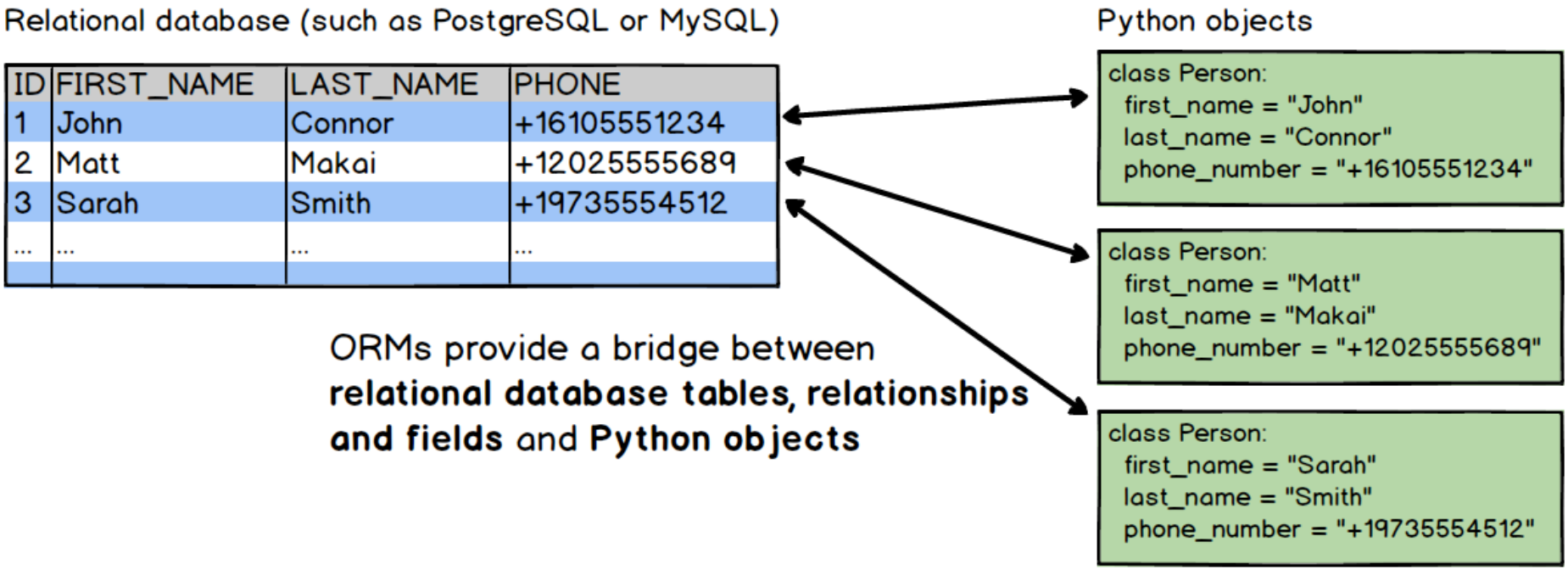
■ ORM in Python

○ ORM allows a developer to write Python code instead of SQL to create, read, update and delete

- ➢ Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures

○ ORMs make it theoretically possible to switch an application between various relational databases

- ➢ In practice however, it's best to use the same database for local development as is used in production



■ ORMs

○ Python ORM libraries are not required for accessing relational databases

➢ In fact, the low-level access is typically provided by another library called a *database connector*

web framework	None	Flask	Flask	Django
ORM	SQLAlchemy	SQLAlchemy	SQLAlchemy	Django ORM
database connector	(built into Python stdlib)	MySQL-python	psycopg	psycopg
relational database	SQLite	MySQL	PostgreSQL	PostgreSQL

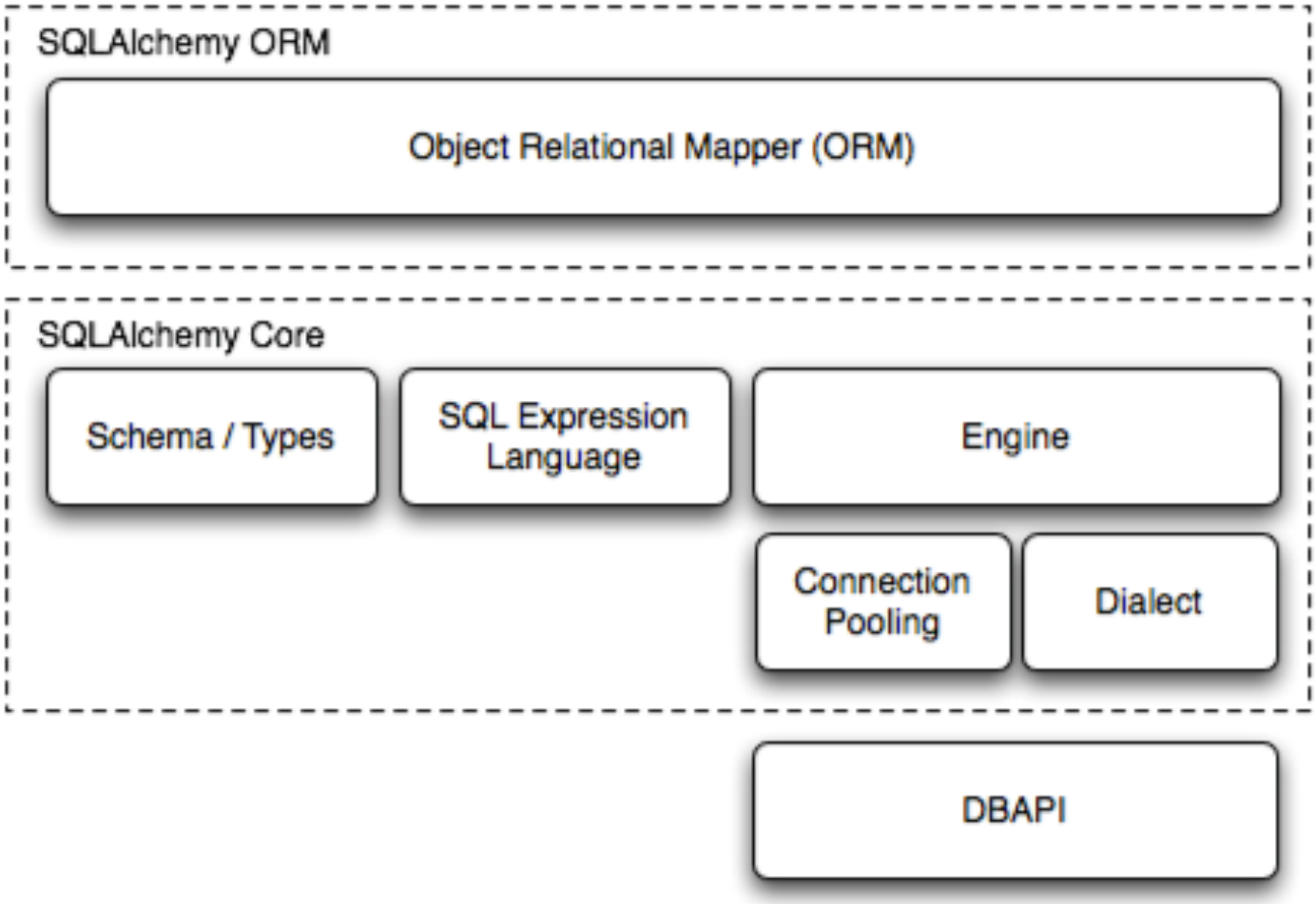
2.2. SQLAlchemy

■ What is SQLAlchemy?

○ SQLAlchemy is a well-regarded database toolkit and ORM implementation written in Python

○ SQLAlchemy provides a generalized interface for creating and executing database-agnostic code without needing to write SQL statements.

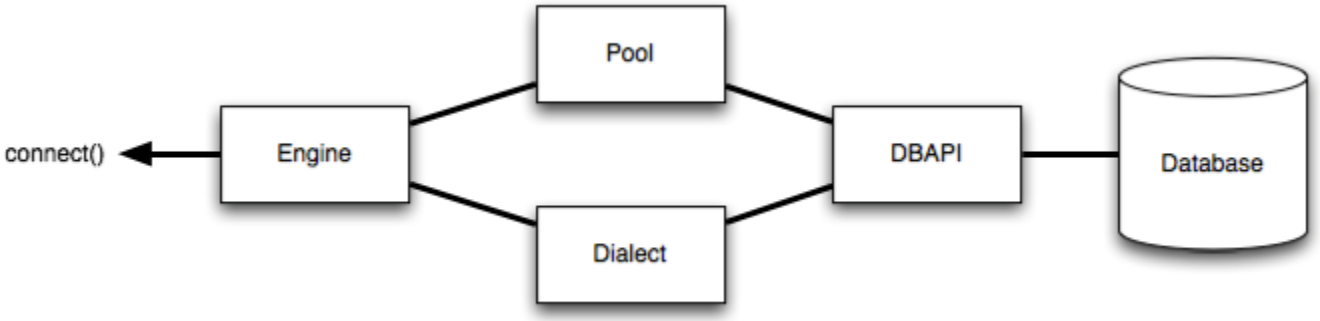
■ SQLAlchemy architecture



■ SQLAlchemy CORE

○ Engine

- ➢ starting point for any SQLAlchemy application
- ➢ a registry which provides connectivity to a particular database server



○ Dialect

- ➢ communicate with various types of DBAPI implementations and databases
- ➢ interprets generic SQL and database commands in terms of specific DBAPI and database backend
  - – Firebird, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, Sybase

○ Connection Pool

- ➢ holds a collection of database connections in memory for fast re-use

■ Connecting

○ create_engine

```
sqlalchemy.create_engine(*args, **kwargs)
```

```
dialect+driver://username:password@host:port/database
```

```
# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')
```

○ 예제

```
import sqlalchemy

sqlalchemy.__version__
```

```
from sqlalchemy import create_engine

engine = create_engine("sqlite://", echo=True)
#engine = create_engine("sqlite:///:memory:", echo=True)
#engine = create_engine("sqlite:///test.db", echo=True)

print(engine)
```

- Lazy connecting
- The echo flag is a shortcut to setting up SQLAlchemy logging

■ Create

○ Table

```
class sqlalchemy.schema.Table(*args, **kw)
```

- Table object constructs a unique instance of itself based on its name and optional schema name within the given MetaData object
- name, metadata, columns, constraint, …

○ Column

```
class sqlalchemy.schema.Column(*args, **kwargs)
```

- Represents a column in a database table
- name, type, constraint, authoincreament, default, nullable, …

○ MetaData

```
class sqlalchemy.schema.MetaData
```

- A collection of Table objects and their associated schema constructs
- Holds a collection of Table objects as well as an optional binding to an Engine or Connection. If bound, the Table objects in the collection and their columns may participate in implicit SQL execution
- Table objects themselves are stored in the MetaData.tables dictionary

○ 예제

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()
users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('fullname', String),
)

addresses = Table('addresses', metadata,
    Column('id', Integer, primary_key=True),
    Column('user_id', None, ForeignKey('users.id')),
    Column('email_address', String, nullable=False)
)

metadata.create_all(engine)
```

■ Insert

○ insert

```
insert(values=None, inline=False, **kwargs)
```

- Represents an INSERT construct
- generate an insert() construct against this TableClause

○ compile

```
compile(bind=None, dialect=None, **kw)
```

- Compile this SQL expression
- Return value is a Compiled object
- Compiled object also can return a dictionary of bind parameter names and values using the params accessor

○ 예제

```
insert = users.insert()
print(insert)

insert = users.insert().values(name='kim', fullname='Anonymous, Kim')
print(insert)

insert.compile().params
```

■ Executing

○ Connection

```
class sqlalchemy.engine.Connection
```

- Provides high-level functionality for a wrapped DB-API connection
- Provides execution support for string-based SQL statements as well as ClauseElement, Compiled and DefaultGenerator objects

○ execute

```
execute(object, *multiparams, **params)
```

- Executes a SQL statement construct and returns a ResultProxy

○ ResultProxy

```
class sqlalchemy.engine.ResultProxy(context)
```

- Wraps a DB-API cursor object to provide easier access to row columns

○ 예제

```
conn = engine.connect()
conn

insert.bind = engine
str(insert)

result = conn.execute(insert)

result.inserted_primary_key
```

➢ execute의 params 사용

```
insert = users.insert()

result = conn.execute(insert, name="lee", fullname="Unknown, Lee")

result.inserted_primary_key
```

➢ DBAPI의 executemany() 사용

```
conn.execute(addresses.insert(), [
    {"user_id":1, "email_address":"anonymous.kim@test.com"},
    {"user_id":2, "email_address":"unknown.lee@test.com"}
])
```

■ Select

○ select

`sqlalchemy.sql.expression.select`

- Construct a new Select
- columns, whereclause, from_obj, group_by, order_by, ...

○ 예제

```
from sqlalchemy.sql import select

query = select([users])
result = conn.execute(query)

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

```
result = conn.execute(select([users.c.name, users.c.fullname]))

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

■ ResultProxy

○ fetchone

```
fetchone()
```

➢ Fetch one row, just like DB-API cursor.fetchone()

○ fetchall

```
fetchall()
```

➢ Fetch all rows, just like DB-API cursor.fetchall()

○ 예제

```
result = conn.execute(query)

row = result.fetchone()
print("id -", row["id"], ", name -", row["name"], ", fullname -", row["fullname"])

row = result.fetchone()
print("id -", row[0], ", name -", row[1], ", fullname -", row[2])

result = conn.execute(query)
rows = result.fetchall()

for row in rows:
    print("id -", row[0], ", name -", row[1], ", fullname -", row[2])

result.close()
```

■ Conjunctions

○ 예제

```
from sqlalchemy import and_, or_, not_

print(users.c.id == addresses.c.user_id)

print(users.c.id == 1)

print((users.c.id == 1).compile().params)

print(or_(users.c.id == addresses.c.user_id, users.c.id == 1))

print(and_(users.c.id == addresses.c.user_id, users.c.id == 1))

print(and_(
        or_(
            users.c.id == addresses.c.user_id,
            users.c.id == 1
        ),
        addresses.c.email_address.like("a%")
     )
)

print((
        (users.c.id == addresses.c.user_id) |
        (users.c.id == 1)
      ) & (addresses.c.email_address.like("a%")))
```

■ Selecting

○ 예제

```
result = conn.execute(select([users]).where(users.c.id==1))
for row in result:
    print(row)

result = conn.execute(select([users, addresses]).where(users.c.id==addresses.c.user_id))
for row in result:
    print(row)

result = conn.execute(select([users.c.id, users.c.fullname, addresses.c.email_address])
                      .where(users.c.id==addresses.c.user_id))
for row in result:
    print(row)

result = conn.execute(select([users.c.id, users.c.fullname, addresses.c.email_address])
                      .where(users.c.id==addresses.c.user_id)
                      .where(addresses.c.email_address.like("un%")))
for row in result:
    print(row)
```

■ Join

○ join

```
join(right, onclause=None, isouter=False, full=False)
```

➢ Return a Join from this FromClause to another FromClause

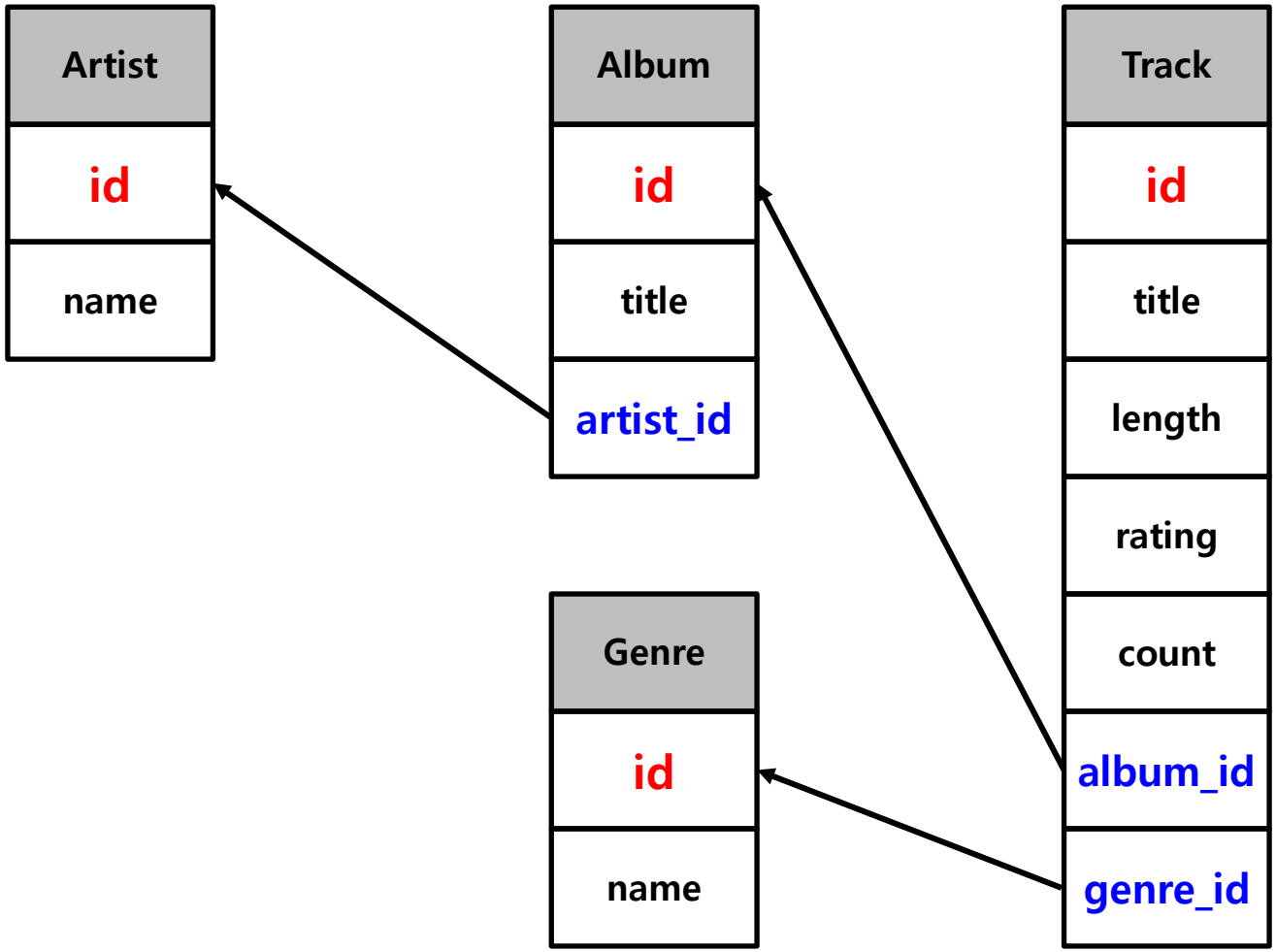
○ 예제

```
from sqlalchemy import join

print(users.join(addresses))

print(users.join(addresses, users.c.id == addresses.c.user_id))
```

➢ ON condition of the join, as it's called, was automatically generated based on the ForeignKey object

```
query = select([users.c.id, users.c.fullname, addresses.c.email_address]) \
        .select_from(users.join(addresses))

result = conn.execute(query).fetchall()
for row in result:
    print(row)
```

■ 예제


Artist
id
name
Album
id
title
artist_id
Track
id
title
length
rating
count
album_id
genre_id
Genre
id
name

○ Create

```
artist = Table("Artist", metadata,
               Column("id", Integer, primary_key=True),
               Column("name", String, nullable=False),
               extend_existing=True)

album = Table("Album", metadata,
              Column("id", Integer, primary_key=True),
              Column("title", String, nullable=False),
              Column("artist_id", Integer, ForeignKey("Artist.id")),
              extend_existing=True)

genre = Table("Genre", metadata,
              Column("id", Integer, primary_key=True),
              Column("name", String, nullable=False),
              extend_existing=True)

track = Table("Track", metadata,
              Column("id", Integer, primary_key=True),
              Column("title", String, nullable=False),
              Column("length", Integer, nullable=False),
              Column("rating", Integer, nullable=False),
              Column("count", Integer, nullable=False),
              Column("album_id", Integer, ForeignKey("Album.id")),
              Column("genre_id", Integer, ForeignKey("Genre.id")),
              extend_existing=True)

metadata.create_all(engine)
```

○ SHOW TABLES

```
tables = metadata.tables
for table in tables:
    print(table)

for table in engine.table_names():
    print(table)
```

○ Insert

```
conn.execute(artist.insert(), [
    {"name":"Led Zepplin"},
    {"name":"AC/DC"}
])

conn.execute(album.insert(), [
    {"title":"IV", "artist_id":1},
    {"title":"Who Made Who", "artist_id":2}
])

conn.execute(genre.insert(), [
    {"name":"Rock"},
    {"name":"Metal"}
])

conn.execute(track.insert(), [
    {"title":"Black Dog", "rating":5, "length":297, "count":0, "album_id":1, "genre_id":1},
    {"title":"Stairway", "rating":5, "length":482, "count":0, "album_id":1, "genre_id":1},
    {"title":"About to rock", "rating":5, "length":313, "count":0, "album_id":2, "genre_id":2},
    {"title":"Who Made Who", "rating":5, "length":297, "count":0, "album_id":2, "genre_id":2}
])
```

○ Select

```
artistResult = conn.execute(artist.select())
for row in artistResult:
    print(row)

albumResult = conn.execute(album.select())
for row in albumResult:
    print(row)

genreResult = conn.execute(genre.select())
for row in genreResult:
    print(row)

trackResult = conn.execute(track.select())
for row in trackResult:
    print(row)
```

○ Where

```
trackResult = conn.execute(select([track])
                           .where(
                               and_(track.c.album_id == 1, track.c.genre_id == 1)
                           )
                          )
for row in trackResult:
    print(row)
```

○ Update

```
from sqlalchemy import update

conn.execute(track.update().values(genre_id=2).where(track.c.id==2))
conn.execute(track.update().values(genre_id=1).where(track.c.id==3))
```

○ Where

```
trackResult = conn.execute(select([track])
                           .where(
                               and_(track.c.album_id == 1,
                                    or_(track.c.genre_id == 1,
                                        track.c.genre_id == 2,)
                                   )
                           )
                          )
for row in trackResult:
    print(row)
```

○ Join

```
print(track.join(album))

result = conn.execute(track
                      .select()
                      .select_from(track.join(album)))

for row in result.fetchall():
    print(row)

result = conn.execute(track
                      .select()
                      .select_from(track.join(album))
                      .where(album.c.id==1))

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

○ Multiple Join

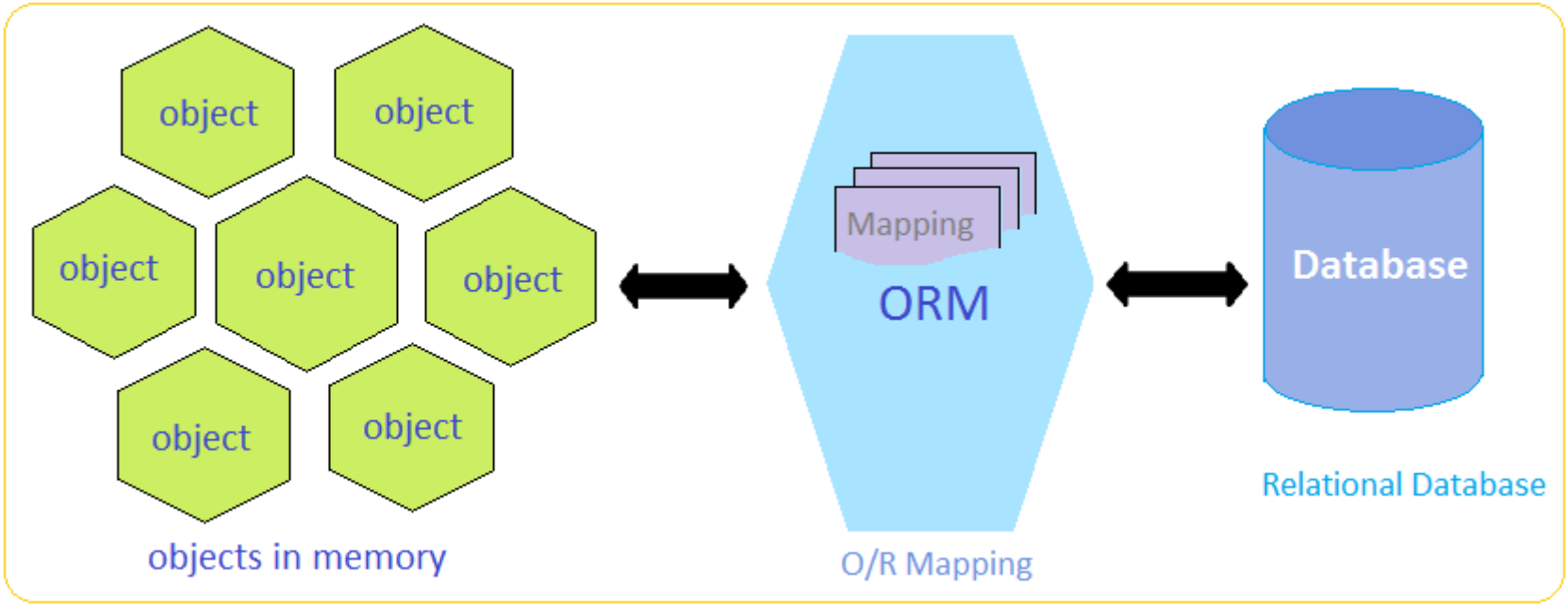
```
print(track.join(album))
print(track.join(album).join(genre))
print(track.join(album).join(artist))
print(track.join(album).join(genre).join(artist))

result = conn.execute(select([track.c.title, album.c.title, genre.c.name, artist.c.name])
                      .select_from(track.join(album).join(genre).join(artist)))

for row in result.fetchall():
    print(row)


result = conn.execute(track
                      .select()
                      .select_from(track.join(album).join(genre).join(artist))
                      .where(
                          and_(
                              genre.c.id==1,
                              artist.c.id==1,
                          )
                      )
                     )

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

○ Open/Close

```
from sqlalchemy import create_engine, MetaData

engine = create_engine("sqlite:///alchemy_core.db", echo=True)
conn = engine.connect()

metadata = MetaData(bind=engine, reflect=True)
metadata.reflect(bind=engine)

for row in metadata.tables:
    print(row)
```

```
tables = metadata.tables
for table in tables:
    print(table)

#album

track = metadata.tables["Track"]
track

for row in conn.execute(track.select()).fetchall():
    print(row)

conn.close()
metadata.clear()
```

■ SQLAlchemy ORM



Data Mapping to Classes

■ Declare

○ declarative_base

```
sqlalchemy.ext.declarative.declarative_base
```

➢ Construct a base class for declarative class definitions

○ 예제

```
from sqlalchemy import create_engine

engine = create_engine("sqlite:///:memory:", echo=True)
```

```
from sqlalchemy.ext.declarative import declarative_base

base = declarative_base()
```

➢ declarative_base() callable returns a new base class from which all mapped classes should inherit

■ Create

○ declarative_base 상속 class 선언

```
class User(base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column("passwd", String)

    def __repr__(self):
        return "<T'User(name='%s', fullname='%s', password='%s')>" \
                % (self.name, self.fullname, self.password)
```

- a new Table and mapper() will have been generated
- table and mapper are accessible via __table__ and __mapper__ attributes

○ Schema

```
User.__table__
```

○ Create Table

```
base.metadata.create_all(engine)
```

○ Create Instance

```
kim = User(name="kim", fullname="anonymous, Kim", password="kimbap heaven")

print(kim)
print(kim.id)
```

■ Session

○ Session

```
class sqlalchemy.orm.session.Session
```

- Session establishes all conversations with the database and represents all the objects
- Manages persistence operations for ORM-mapped objects

○ sessionmaker

```
class sqlalchemy.orm.session.sessionmaker
```

- sessionmaker factory generates new Session objects when called
- sessionmaker class is normally used to create a top level Session configuration

○ 예제

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

■ Insert

○ add

```
add(instance, _warn=True)
```

- Place an object in the Session, persisted to the database on the next flush operation.
- Repeated calls to add() will be ignored

○ addall

```
add_all(instances)
```

- Add the given collection of instances to this Session

○ 예제

```
session.add(kim)

session.add_all([
    User(name="lee", fullname="unknown, Lee", password="123456789a"),
    User(name="park", fullname="nobody, Park", password="Parking in Park")
])
```

- Pending. no SQL has yet been issued and the object is not yet represented by a row in the database

■ Update

○ dirty

```
dirty
```

- The set of all persistent instances considered dirty
- Instances are considered dirty when they were modified but not deleted

○ is_modified

```
is_modified(instance, include_collections=True, passive=True)
```

- Return True if the given instance has locally modified attributes

○ 예제

```
kim.password = "password"

session.dirty

session.is_modified(kim)
```

■ Commit

○ commit

```
commit()
```

- ➢ Flush pending changes and commit the current transaction
- ➢ If no transaction is in progress, this method raises an InvalidRequestError

■ Select

○ query

```
class sqlalchemy.orm.query.Query(entities, session=None)
```

- ➢ ORM-level SQL construction object
- ➢ Query is the source of all SELECT statements generated by the ORM

○ 예제

```
for row in session.query(User):
    print(type(row))
    print(row.id, row.name, row.fullname, row.password)
```

○ filter

filter(**criterion*)

- Apply the given filtering criterion to a copy of this Query, using SQL expressions
- Allow you to use regular Python operators with the class-level attributes on your mapped class

○ filter_by

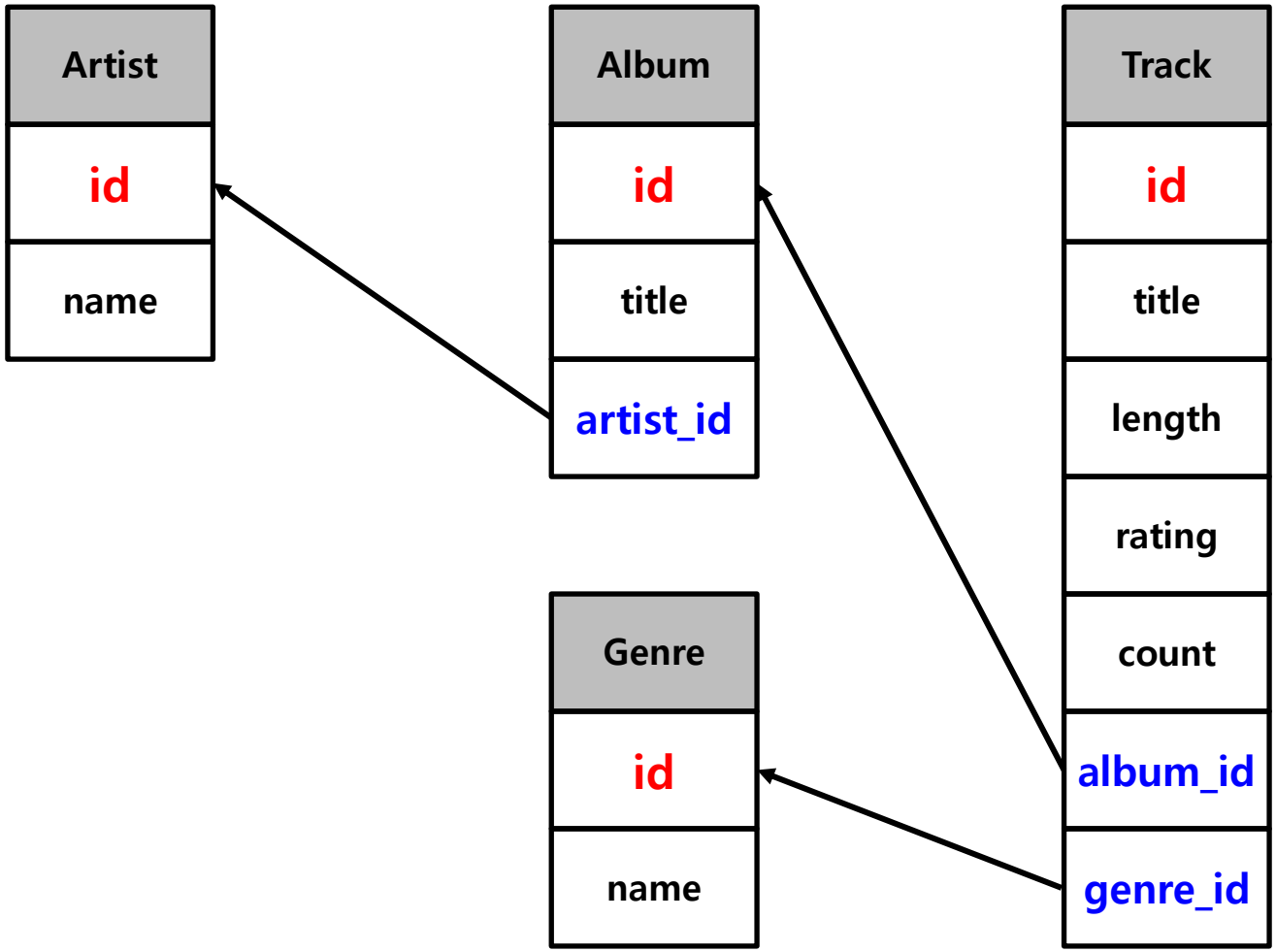
filter_by(***kwargs*)

- apply the given filtering criterion to a copy of this Query, using keyword expressions

○ 예제

```
for row in session.query(User.id, User.fullname).filter(User.name == "lee"):
    print(type(row))
    print(row.id, row.fullname)

for row in session.query(User.id, User.fullname).filter_by(name = "lee"):
    print(type(row))
    print(row.id, row.fullname)
```

■ 예제


Artist
id
name
Album
id
title
artist_id
Track
id
title
length
rating
count
album_id
genre_id
Genre
id
name

○ Create

```
class Artist(Base):
    __tablename__ = "Artist"

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

class Album(Base):
    __tablename__ = "Album"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    artist_id = Column(Integer, ForeignKey("Artist.id"))

class Genre(Base):
    __tablename__ = "Genre"

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

class Track(Base):
    __tablename__ = "Track"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    length = Column(Integer)
    rating = Column(Integer)
    count = Column(Integer)
    album_id = Column(Integer, ForeignKey("Album.id"))
    genre_id = Column(Integer, ForeignKey("Genre.id"))
```

○ Insert

```
artist1 = Artist(name="Led Zepplin")
artist2 = Artist(name="AC/DC")

session.add_all([artist1, artist2])
session.commit()

album = [Album(title="IV", artist_id=artist1.id),
         Album(title="Who Made Who", artist_id=artist2.id)]

session.add_all(album)
session.commit()

session.add_all([Genre(name="Rock"), Genre(name="Metal")])
session.commit()

album1 = session.query(Album).filter(Album.artist_id==artist1.id).one()
album2 = session.query(Album).filter(Album.artist_id==artist2.id).one()

genre1 = session.query(Genre).filter(Genre.name=="Rock").filter(Genre.id==1).one()
genre2 = session.query(Genre).filter(Genre.name=="Metal").filter(Genre.id==2).one()

track = [Track(title="Black Dog", rating=5, length=297, count=0, album_id=album1.id, genre_id=genre1.id),
         Track(title="Stairway", rating=5, length=482, count=0, album_id=album1.id, genre_id=genre2.id),
         Track(title="About to rock", rating=5, length=313, count=0, album_id=album2.id, genre_id=genre1.id),
         Track(title="Who Made Who", rating=5, length=297, count=0, album_id=album2.id, genre_id=genre2.id)]
session.add_all(track)
session.commit()
```

○ Join

```
result = session.query(Track.title, Album.title, Genre.name, Artist.name) \
         .select_from(Track) \
         .join(Album)\
         .join(Genre)\
         .join(Artist).all()

for row in result:
    print(row)

songList = [dict(Track=row[0], Album=row[1], Genre=row[2], Artist=row[3]) for row in result]

songList
```

■ Relationship

○ relationship

```
sqlalchemy.orm.relationship
```

- ➢ Provide a relationship between two mapped classes
- ➢ This corresponds to a parent-child or associative table relationship

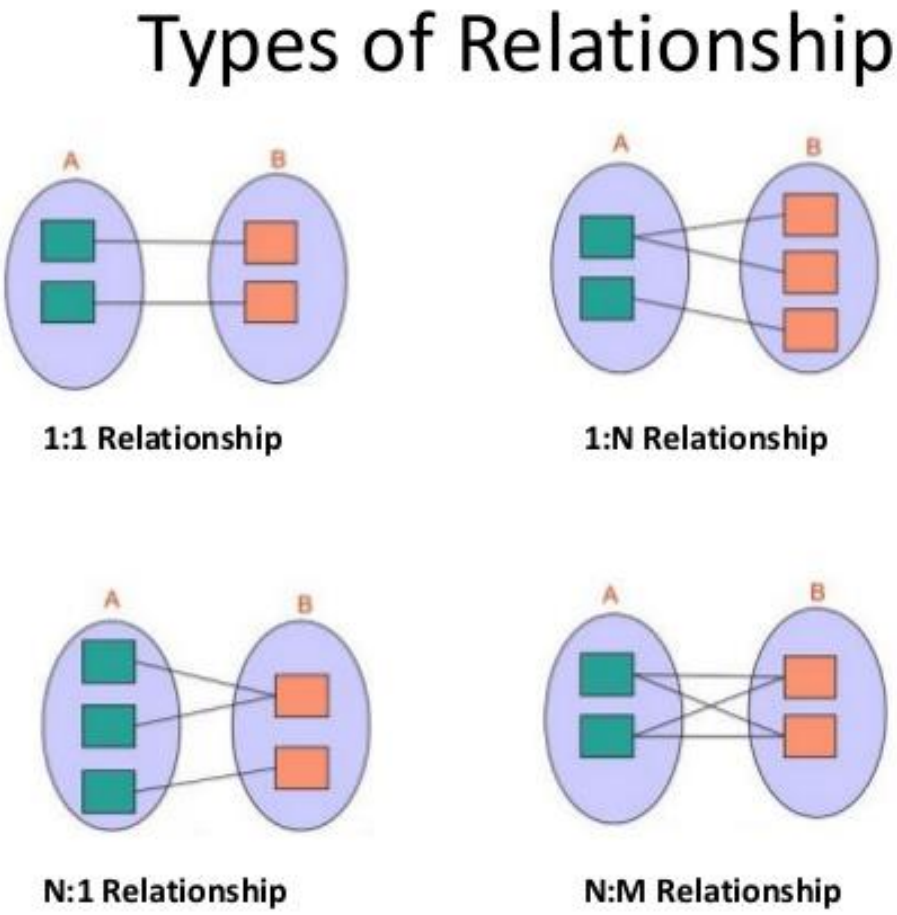
○ back_populates / backref

- ➢ indicates the string name of a property to be placed on the related mapper's class that will handle this relationship in the other direction

○ uselist

- ➢ a boolean that indicates if this property should be loaded as a list or a scalar

■ Relationship Model



■ 예제

```
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column("passwd", String)

    #addresses = relationship("Address", back_populates="user")
    #addresses = relationship("Address", back_populates="user", uselist=False)

    def __repr__(self):
        return "<User(name='%s', fullname='%s', password='%s')>" % (self.name, self.fullname, self.password)
```

```
class Address(Base):
    __tablename__ = "addresses"

    id = Column(Integer, primary_key=True)
    email_address = Column(String, nullable=False)
    user_id = Column(Integer, ForeignKey("users.id"))

    #user = relationship("User", back_populates="addresses", uselist=False)
    #user = relationship("User", back_populates="addresses")
    #user = relationship("User")

    def __repr__(self):
        return "<Address(email_address='%s')>" % self.email_address
```

■ 예제

```
class Artist(Base):
    __tablename__ = "Artist"

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

    albumList = relationship("Album", back_populates="artist")

class Album(Base):
    __tablename__ = "Album"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    artist_id = Column(Integer, ForeignKey("Artist.id"))

    artist = relationship("Artist", back_populates="albumList", uselist=False)
    trackList = relationship("Track", back_populates="album")

class Genre(Base):
    __tablename__ = "Genre"

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

    trackList = relationship("Track", back_populates="genre")

class Track(Base):
    __tablename__ = "Track"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    length = Column(Integer)
    rating = Column(Integer)
    count = Column(Integer)
    album_id = Column(Integer, ForeignKey("Album.id"))
    genre_id = Column(Integer, ForeignKey("Genre.id"))

    album = relationship("Album", back_populates="trackList", uselist=False)
    genre = relationship("Genre", back_populates="trackList", uselist=False)
```

Insert

```
track1 = Track(title="Black Dog", rating=5, length=297, count=0)
track2 = Track(title="Stairway", rating=5, length=482, count=0)
track3 = Track(title="About to rock", rating=5, length=313, count=0)
track4 = Track(title="Who Made Who", rating=5, length=297, count=0)
```

```
track1.album = track2.album = Album(title="IV")
track3.album = track4.album = Album(title="Who Made Who")
```

```
track1.genre = track3.genre = Genre(name="Rock")
track2.genre = track4.genre = Genre(name="Metal")
```

```
track1.album.artist = track2.album.artist = Artist(name="Led Zepplin")
track3.album.artist = track4.album.artist = Artist(name="AC/DC")
```

Select

```
print("Title: %s, Album: %s, Genre: %s, Artist: %s" %
      (track1.title, track1.album.title, track1.genre.name, track1.album.artist.name))
print("Title: %s, Album: %s, Genre: %s, Artist: %s" %
      (track3.title, track3.album.title, track3.genre.name, track3.album.artist.name))
```

```
print("TrackID: %d, AlbumID: %d, GenreID: %d, Artist: %d" %
      (track1.id, track1.album.id, track1.genre.id, track1.album.artist.id))

print("TrackID: %d, AlbumID: %d, GenreID: %d, Artist: %d" %
      (track3.id, track3.album.id, track3.genre.id, track3.album.artist.id))
```