SQL Databases and Python

Databases

Pierre Senellart

PSL UNIVERSITÉ PARIS

22 April 2026

Outline

SQL is not enough

- SQL is a powerful language for data manipulation
- But real applications need more:
  - User interfaces (web pages, forms, reports)
  - Complex business logic (loops, conditionals, functions)
  - Interaction with external services (APIs, files, email)
  - Session management and authentication
- Solution: embed database access in a general-purpose language
- The language drives the computation; SQL queries the data

The object–relational mismatch

Relational model (SQL)

- Data in tables (sets of tuples)
- Set-oriented operations
- Declarative: *what* to retrieve
- Types: `INT`, `VARCHAR`, ...

Programming languages

- Data in objects / variables
- Record-at-a-time processing
- Imperative: *how* to compute
- Types: `int`, `str`, `list`, ...

The object–relational mismatch

The structural incompatibility between the relational model and object-oriented / procedural programming languages. Bridging this gap is the central challenge of database application development.

Outline

Approaches to database connectivity

Three main approaches have evolved historically:

Embedded SQL SQL written *directly* in the host language source; a preprocessor transforms it into library calls before compilation. Examples: ESQL/C, Oracle Pro*C. Rarely used today (legacy systems).

Call-level interface (CLI) The application calls a *library* at runtime; SQL is a string passed to a function. No preprocessor needed. The standard approach today.

Object-Relational Mapper (ORM) A library maps tables to classes and rows to objects; SQL is generated behind the scenes. Examples: SQLAlchemy (Python), Hibernate (Java), ActiveRecord (Ruby).

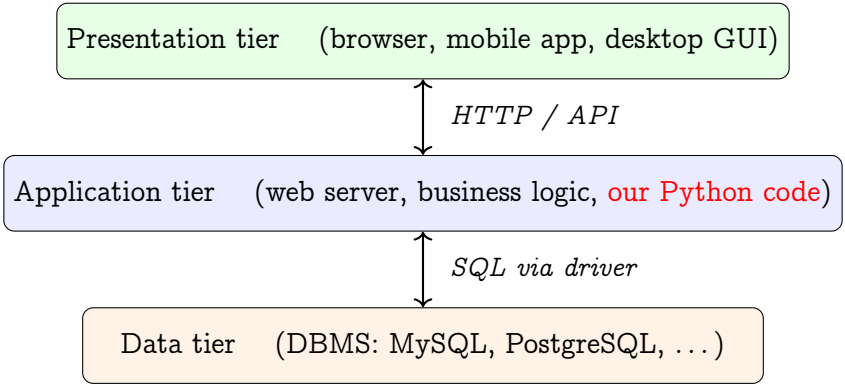
Call-level interfaces by language

Language	Standard / Library	Notes
C / C++	ODBC	Windows-native standard, any DBMS
C / C++	MySQL C API	Native MySQL / MariaDB library
Java	JDBC	Java standard, one driver per DBMS
Python	DB-API 2.0	PEP 249 standard, one driver per DBMS

- All share the same conceptual model: connect → execute → fetch
- A standard ensures that switching databases requires minimal code changes

Three-tier architecture

Most database-backed applications follow a three-tier structure:



- Each tier runs on separate processes, often separate machines
- The application tier is where DB-API / SQLAlchemy code lives
- Many application-tier instances share a connection pool to the DBMS – opening one connection per request would not scale

Python DB-API 2.0 (PEP 249)

- Python's standard for database connectivity, defined in PEP 249 (2001)
- Specifies a common interface: any compliant driver behaves the same way
- Two key objects: Connection and Cursor
- Popular drivers for MySQL (used as `import` *name*):
  - `mysql.connector`: official Oracle driver, pure Python
  - `pymysql`: pure Python, lightweight
  - `MySQLdb`: wraps `libmysqlclient`, fastest
- We use `mysql.connector` throughout this course

One interface, many databases

Switching from MySQL to PostgreSQL (using `psycopg2`) requires changing only the `connect()` call. The rest of the code – cursors, execute, fetch, commit – is identical.

Outline

Running example: a movie database (1/2)

SQL

```sql
CREATE TABLE director (
  director_id INT PRIMARY KEY AUTO_INCREMENT,
  name        VARCHAR(100) NOT NULL,
  nationality VARCHAR(50)
);

CREATE TABLE movie (
  movie_id    INT PRIMARY KEY AUTO_INCREMENT,
  title       VARCHAR(200) NOT NULL,
  year        INT,
  genre       VARCHAR(50),
  director_id INT,
  FOREIGN KEY (director_id) REFERENCES director(director_id)
);
```

Running example: a movie database (2/2)

SQL

```sql
CREATE TABLE actor (
  actor_id   INT PRIMARY KEY AUTO_INCREMENT,
  name       VARCHAR(100) NOT NULL,
  birth_year INT
);

CREATE TABLE starring (
  movie_id INT,
  actor_id INT,
  role     VARCHAR(100),
  PRIMARY KEY (movie_id, actor_id),
  FOREIGN KEY (movie_id) REFERENCES movie(movie_id),
  FOREIGN KEY (actor_id) REFERENCES actor(actor_id)
);
```

Establishing a connection

Python

```python
import mysql.connector

conn = mysql.connector.connect(
    host="localhost",
    user="student",
    password="s3cr3t",
    database="movies"
)
```

- Returns a `Connection` object representing an open session
- Raises `mysql.connector.Error` on failure
- Always close when done: `conn.close()`
- In production, credentials are read from a config file or environment variable – never hard-coded in source code

The Cursor object

A Cursor is the object through which queries are executed and results are retrieved. One connection can have multiple cursors.

Method / Attribute	Description
`cursor.execute(sql, params)`	Execute a SQL statement
`cursor.executemany(sql, seq)`	Execute for each set of params
`cursor.fetchone()`	Next row as tuple, or `None`
`cursor.fetchall()`	All remaining rows as list of tuples
`cursor.fetchmany(n)`	Next n rows
`cursor.description`	Column metadata (name, type, ...)
`cursor.rowcount`	Rows affected by last DML statement
`cursor.lastrowid`	Auto-generated key of last INSERT
`cursor.close()`	Release cursor resources

Executing a SELECT query

Python

```python
cursor = conn.cursor()
cursor.execute("""
    SELECT title, year
    FROM   movie
    WHERE  genre = 'Drama'
    ORDER  BY year
""")
for title, year in cursor:    # iterate row by row
    print(f"{year}  {title}")
cursor.close()
```

- Iterating over the cursor is equivalent to repeated `fetchone()` calls – rows are not all loaded at once
- Rows are returned as tuples by default; use `conn.cursor(dictionary=True)` for `dict` rows

Column metadata

Python

```python
cursor.execute("SELECT * FROM director LIMIT 3")
# cursor.description: one 7-item sequence per output column
# (name, type_code, display_size, internal_size,
#  precision, scale, null_ok)
col_names = [desc[0] for desc in cursor.description]
# ['director_id', 'name', 'nationality']
rows = cursor.fetchall()
for row in rows:
    record = dict(zip(col_names, row))
    print(record)
# {'director_id': 1, 'name': 'Wong Kar-wai', 'nationality': 'HK'}
```

Useful when the query is built dynamically and column names are not known in advance.

Outline

Never concatenate user input into SQL

Python

```
genre = input("Enter genre: ")
# NEVER DO THIS:
cursor.execute(
    "SELECT title FROM movie WHERE genre = '" + genre + "'"
)
```

If the user types "`' OR '1'='1`", the query becomes:

SQL

```
SELECT title FROM movie WHERE genre = '' OR '1'='1'
```

...which returns *every* movie. If they type "`'; DROP TABLE movie; --`":

SQL

```
SELECT title FROM movie WHERE genre = ''; DROP TABLE movie; --'
```

SQL Injection

SQL injection

An attack in which user-supplied data is interpreted as SQL code, allowing an attacker to read, modify, or destroy database content – without any special access privileges.

- One of the most critical web application security risks
- Responsible for many high-profile data breaches
- Completely preventable: always use parameterized queries

"Little Bobby Tables" – `https://xkcd.com/327/`

Parameterized queries

Python

```python
genre = input("Enter genre: ")
cursor.execute(
    "SELECT title FROM movie WHERE genre = %s",
    (genre,)    # second argument: tuple of parameter values
)
# Multiple parameters:
cursor.execute(
    "SELECT title FROM movie WHERE genre = %s AND year > %s",
    ("Drama", 2000)
)
```

- %s is the placeholder for MySQL's connector (other drivers use ? or :name)
- The driver escapes and quotes the value safely before sending to the DBMS
- Parameters are always passed as a separate argument – never interpolated with Python's % operator or f-strings

Named parameters and batch execution

Python

```python
# Named parameters (dict-style):
cursor.execute(
    "SELECT title FROM movie "
    "WHERE genre = %(genre)s AND year > %(min_year)s",
    {"genre": "Drama", "min_year": 2000}
)
# executemany: efficient batch insertion
new_actors = [
    ("Meryl Streep", 1949), ("Cate Blanchett", 1969),
    ("Joaquin Phoenix", 1974),
]
cursor.executemany(
    "INSERT INTO actor (name, birth_year) VALUES (%s, %s)",
    new_actors
)
```


executemany is more efficient than calling execute() in a loop.

Outline

INSERT, UPDATE, DELETE

Python

```
# Insert; AUTO_INCREMENT assigns director_id
cursor.execute(
    "INSERT INTO director (name, nationality) VALUES (%s, %s)",
    ("Bong Joon-ho", "South Korean")
)
new_id = cursor.lastrowid    # the auto-generated id
# Update; rowcount tells how many rows were affected
cursor.execute(
    "UPDATE movie SET genre = %s WHERE movie_id = %s",
    ("Thriller", 42)
)
print(cursor.rowcount, "row(s) updated")
# DELETE is analogous
```

Error handling

Python

```python
import mysql.connector
try:
    cursor.execute(
        "INSERT INTO starring (movie_id, actor_id, role) "
        "VALUES (%s, %s, %s)",
        (999, 1, "Lead")    # movie_id 999 does not exist
    )
    conn.commit()
except mysql.connector.Error as e:
    print(f"Database error {e.errno}: {e.msg}")
    conn.rollback()
```

- `mysql.connector.Error` is the base exception class
- `e.errno`: MySQL error number (e.g., 1452 = foreign key constraint violation)
- Always `rollback()` on error to leave the database in a consistent state

NULL and None

SQL `NULL` and Python `None` map to each other in both directions:

Python

```
# NULL in a result set comes back as None
cursor.execute("SELECT title, genre FROM movie WHERE movie_id = 1")
title, genre = cursor.fetchone()
if genre is None:           # or: genre == None
    print("genre unknown")
# Pass None to insert NULL
cursor.execute(
    "INSERT INTO movie (title, year, genre, director_id) "
    "VALUES (%s, %s, %s, %s)",
    ("Untitled Project", 2026, None, None)
)
```

- Three-valued logic: "`NULL = NULL`" is `NULL` in SQL, but "`None == None`" is `True` in Python – the semantics differ

Outline

The problem: multiple operations

Real applications often need several database operations as one logical unit.

Example: a user transfers 100 € from account A to account B

1. `UPDATE account SET balance = balance - 100 WHERE id = 'A'`
2. `UPDATE account SET balance = balance + 100 WHERE id = 'B'`

What could go wrong?

- The application crashes between step 1 and step 2
  $\Rightarrow$ 100 € vanishes: debited from A, never credited to B
- Another connection reads between step 1 and step 2
  $\Rightarrow$ it sees the total balance reduced by \$100

Transactions

Transaction

A transaction is a sequence of database operations that the DBMS executes as a single, indivisible unit.

From the application's perspective, a transaction provides two key guarantees:

Atomicity Either *all* operations succeed and their effects are made permanent (commit), or *none* of them take effect (rollback). Partial updates are impossible.

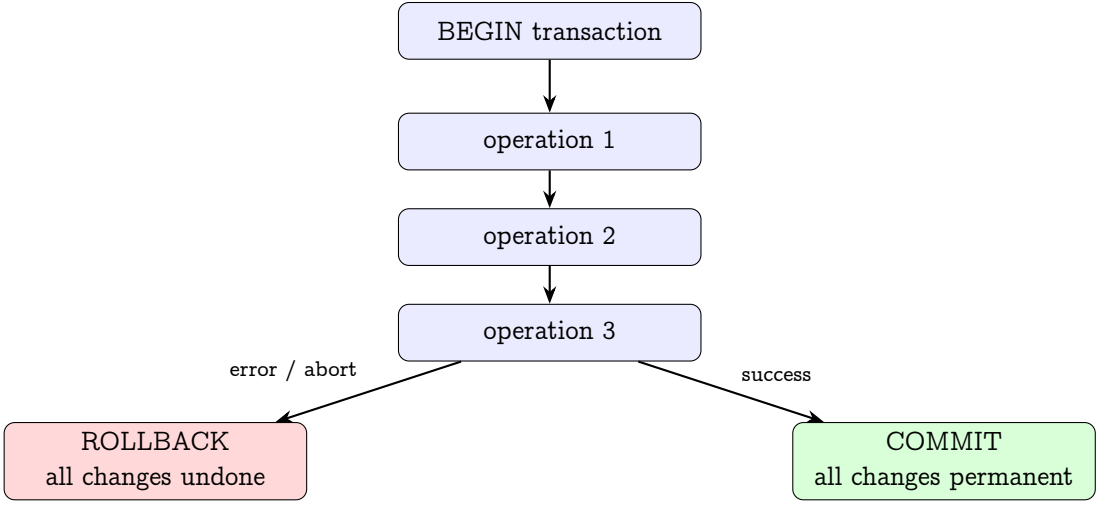
Isolation Concurrent transactions do not see each other's intermediate state. Each transaction behaves as if it ran alone.

Plus two more (collectively: ACID):

Durability Committed changes survive crashes.

Consistency Your responsibility: each transaction must leave the database in a valid state.

Atomicity: commit and rollback



Isolation: concurrent anomalies

Without isolation, concurrent transactions can interfere in three ways:

Dirty read T_1 reads data written by T_2, which has not yet committed. If T_2 rolls back, T_1 has acted on data that never officially existed.

Non-repeatable read T_1 reads the same row twice. Between the two reads, T_2 updates and commits that row. T_1 sees two different values for the same data.

Phantom read T_1 runs the same range query twice. Between the two executions, T_2 inserts rows that satisfy the condition. T_1 sees a different set of rows.

SQL isolation levels

The SQL standard defines four isolation levels (weakest to strongest):

Level	Dirty read	Non-repeatable	Phantom
`READ UNCOMMITTED`	possible	possible	possible
`READ COMMITTED`	no	possible	possible
`REPEATABLE READ`	no	no	possible
`SERIALIZABLE`	no	no	no

- MySQL default: `REPEATABLE READ`
- Higher isolation $\Rightarrow$ stronger guarantees, potentially lower throughput
- `SERIALIZABLE`: transactions behave as if executed one after another
- *How* these levels are implemented (locks, multi-version concurrency control) is a topic for another course

Outline

autocommit and manual commit

- By default, DB-API 2.0 connections are in manual commit mode
- Every `execute()` call is part of an open transaction
- `conn.commit()`: make all changes since the last commit permanent
- `conn.rollback()`: undo all changes since the last commit

To commit each statement automatically:

Python

```
conn = mysql.connector.connect(..., autocommit=True)
```

- With `autocommit=True`, each statement is its own transaction
- Convenient for simple scripts with independent writes; dangerous when multiple writes must succeed together

Using commit and rollback

Python

```python
try:
    cursor.execute(          # Step 1: debit sender
        "UPDATE account SET balance = balance - %s WHERE id = %s",
        (100, "A")
    )
    cursor.execute(          # Step 2: credit receiver
        "UPDATE account SET balance = balance + %s WHERE id = %s",
        (100, "B")
    )
    conn.commit()            # both changes made permanent together
except mysql.connector.Error as e:
    conn.rollback()          # neither change takes effect
    raise
```

Either *both* steps are visible to other connections, or *neither* is.

Context managers

Python

```
with mysql.connector.connect(
    host="localhost", user="student",
    password="s3cr3t", database="movies"
) as conn:
    with conn.cursor() as cursor:
        cursor.execute(
            "INSERT INTO director (name, nationality) "
            "VALUES (%s, %s)", ("Agnès Varda", "French")
        )
    conn.commit()
# cursor and connection closed automatically on exit
```

- "`with conn.cursor() as cursor:`" ensures `cursor.close()` is always called, even on exception
- `conn.commit()` must still be called explicitly; on exception, call `conn.rollback()` in the `except` branch

Outline

Working with raw DB-API is verbose

Python

```
cursor.execute("""
    SELECT m.title, d.name, m.year
    FROM movie m JOIN director d
      ON m.director_id = d.director_id
    WHERE m.genre = %s
""", ("Drama",))
cols = [d[0] for d in cursor.description]
movies = [dict(zip(cols, row)) for row in cursor.fetchall()]
```

- SQL is a plain string: no type checking, no IDE completion
- Rows are tuples by default; `cursor(dictionary=True)` gives dicts, but no named attributes or type annotations
- Schema changes silently break SQL strings throughout the codebase

Object-Relational Mapper (ORM)

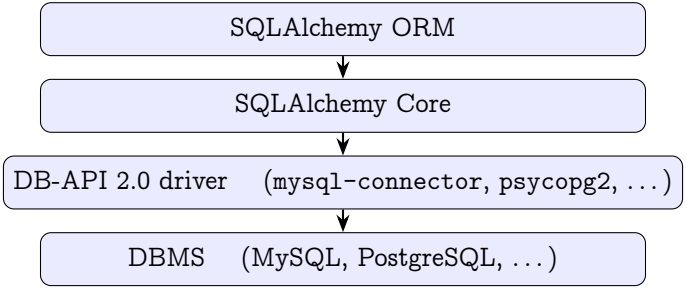
An ORM maps tables to Python classes, rows to instances, and foreign keys to

SQLAlchemy

- The most widely used Python database toolkit
- Two distinct layers:

  Core SQL Expression Language: a Pythonic, composable, database-agnostic way to build and execute SQL

  ORM Declarative mapping: Python classes represent tables; a *Session* manages object persistence automatically
- Both layers sit on top of any DB-API 2.0 driver
- Can be used independently or mixed freely



Outline

Engine: the entry point

Python

```python
from sqlalchemy import create_engine

engine = create_engine(
    "mysql+mysqlconnector://student:s3cr3t@localhost/movies",
    echo=True       # print generated SQL to stdout (for debugging)
)
```

- `create_engine()` takes a connection URL: `dialect+driver://user:password@host/database`
- Creates a connection pool: connections are reused efficiently
- Switch databases by changing the URL only: `postgresql+psycopg2://...`, `sqlite:///movies.db`
- `echo=True` is invaluable during development

Connection pooling

Opening a new connection to the DBMS for every query is expensive (authentication, handshake, memory allocation on the server). SQLAlchemy's engine maintains a connection pool automatically:

Python

```
engine = create_engine(
    "mysql+mysqlconnector://student:s3cr3t@localhost/movies",
    pool_size=5,        # keep 5 connections open
    max_overflow=10,    # allow up to 10 extra under load
    pool_timeout=30,    # wait at most 30 s for a free connection
    pool_recycle=3600   # discard connections older than 1 hour
)
```

- `engine.connect()` borrows a connection from the pool; the `with` block returns it when done
- Critical for web apps: many requests share a small pool rather than each opening their own connection; raw DB-API has no built-in pooling

Executing SQL with Core

Python

```python
from sqlalchemy import text
with engine.connect() as conn:
    result = conn.execute(
        text("SELECT title, year FROM movie "
             "WHERE genre = :genre ORDER BY year"),
        {"genre": "Drama"}
    )
    for row in result:
        print(row.title, row.year)    # attribute access by name
    conn.commit()
```

- `text()` wraps a raw SQL string; Core uses `:name` placeholders (translated to driver syntax)
- Rows support `row.title` and `row[0]`
- `engine.connect()` returns the connection to the pool on exit

SQL Expression Language

Core also provides a composable language that generates SQL:

Python

```
from sqlalchemy import Table, MetaData, select
metadata = MetaData()
movie = Table("movie", metadata, autoload_with=engine)
stmt = (
    select(movie.c.title, movie.c.year)
    .where(movie.c.genre == "Drama")
    .order_by(movie.c.year)
    .limit(10)
)
with engine.connect() as conn:
    for row in conn.execute(stmt):
        print(row.title, row.year)
```

- `autoload_with=engine` queries the database to discover columns and types automatically – no need to declare them in Python
- Composable: clauses can be added or modified before execution

Inspecting generated SQL

It is always possible to see exactly what SQL a statement will produce:

Python

```python
from sqlalchemy.dialects import mysql
stmt = (
    select(movie.c.title, movie.c.year)
    .where(movie.c.genre == "Drama")
    .order_by(movie.c.year)
)
print(stmt.compile(dialect=mysql.dialect(),
                   compile_kwargs={"literal_binds": True}))
# SELECT title, year FROM movie
# WHERE genre = 'Drama' ORDER BY year
```

- `echo=True` on the engine prints every executed statement to stdout at runtime
- Indispensable for debugging unexpected results or performance issues: always verify that the ORM generates the SQL you expect

Outline

Declaring models

Python

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
from sqlalchemy import String, ForeignKey
class Base(DeclarativeBase): pass   # one per application
class Director(Base):
    __tablename__ = "director"
    director_id: Mapped[int] = mapped_column(primary_key=True)
    name:        Mapped[str] = mapped_column(String(100))
    nationality: Mapped[str] = mapped_column(String(50))
class Movie(Base):
    __tablename__ = "movie"
    movie_id:    Mapped[int] = mapped_column(primary_key=True)
    title:       Mapped[str] = mapped_column(String(200))
    year:        Mapped[int]
    genre:       Mapped[str] = mapped_column(String(50))
    director_id: Mapped[int] = mapped_column(
                     ForeignKey("director.director_id"))
```

Relationships

Python

```python
from sqlalchemy.orm import relationship
class Director(Base):
    __tablename__ = "director"
    director_id: Mapped[int] = mapped_column(primary_key=True)
    name:        Mapped[str] = mapped_column(String(100))
    nationality: Mapped[str] = mapped_column(String(50))
    movies: Mapped[list["Movie"]] = relationship(
                back_populates="director")
class Movie(Base):
    __tablename__ = "movie"
    # ... (columns as before)
    director: Mapped["Director"] = relationship(
                  back_populates="movies")
```

- `director.movies` → list of `Movie`; `movie.director` → `Director`
- Related objects are loaded lazily by default (query issued on first attribute access)

Session: creating objects

Python

```python
from sqlalchemy.orm import Session
Base.metadata.create_all(engine)   # create tables if absent
with Session(engine) as session:
    d = Director(name="Wong Kar-wai", nationality="Hong Kong")
    session.add(d)
    session.flush()        # sends INSERT, assigns director_id
    m = Movie(title="In the Mood for Love",
              year=2000, genre="Drama",
              director_id=d.director_id)
    session.add(m)
    session.commit()       # finalizes both INSERTs permanently
```

- The session tracks all pending changes (unit-of-work pattern)
- `flush()` sends SQL to the DB but does not commit
- `commit()` finalizes and starts a new transaction

Querying with ORM

Python

```python
from sqlalchemy import select
with Session(engine) as session:
    # Fetch a single object by primary key
    m = session.get(Movie, 1)
    print(m.title, "directed by", m.director.name)
    # Query with filter
    stmt = (
        select(Movie)
        .where(Movie.genre == "Drama")
        .order_by(Movie.year)
    )
    dramas = session.scalars(stmt).all()
    for movie in dramas:
        print(movie.year, movie.title, "->", movie.director.name)
```

`session.scalars(stmt).all()` returns ORM objects

The N+1 query problem

Python

```python
# Problem: lazy loading fires one SELECT per movie
movies = session.scalars(select(Movie)).all()
for movie in movies:
    print(movie.title, "->", movie.director.name)
# Fix: eager loading with selectinload (2 queries total)
from sqlalchemy.orm import selectinload
stmt = select(Movie).options(selectinload(Movie.director))
movies = session.scalars(stmt).all()
for movie in movies:
    print(movie.title, "->", movie.director.name)
```

- `selectinload`: one extra "`SELECT ... WHERE id IN (...)`" for all objects at once – 2 queries total instead of $N + 1$
- `joinedload`: uses a `JOIN` in the main query

Updating and deleting

Python

```python
with Session(engine) as session:
    # Update: modify an attribute, commit
    movie = session.get(Movie, 42)
    movie.genre = "Thriller"
    session.commit()      # generates UPDATE automatically
    # Delete
    old_movie = session.get(Movie, 7)
    if old_movie:
        session.delete(old_movie)
        session.commit()  # generates DELETE automatically
```

- The session tracks changes to all loaded objects (dirty tracking)
- Modifying an attribute and calling `commit()` generates the correct `UPDATE` with no SQL to write
- No need to write SQL queries by hand

Choosing your approach

	DB-API 2.0	SQLAlchemy Core	SQLAlchemy ORM
SQL written by	you (strings)	expression language	auto-generated
Results as	tuples / dicts	`Row` objects	mapped classes
Choose for	scripts, full control; no extra dependency	complex queries, bulk operations, fine-grained SQL	domain models, updates, long-term maintainability
Avoid for	large codebases; schema changes silently break strings	simple updates (more verbose than ORM)	complex aggregations, bulk operations

In practice: mix

SQLAlchemy lets you combine Core and ORM freely – use the ORM for object management and drop to Core or `text()` for complex queries or bulk operations.

Conclusion

- Programs access databases through a driver implementing a standard API: ODBC (C), JDBC (Java), DB-API 2.0 (Python)
- DB-API 2.0: `Connection` + `Cursor`; always use parameterized queries to prevent SQL injection
- Transactions guarantee atomicity (all-or-nothing) and isolation (no interference from concurrent transactions)
- SQLAlchemy Core and ORM sit on top of DB-API 2.0; inspect generated SQL with `echo=True`

Licensing