ORM

Bases de Datos 1 (CC3088) - 2026

ORM

Semestre 01, 2026

El problema sin ORM

Una aplicación necesita buscar un producto por su ID.

```
import psycopg2

conn = psycopg2.connect("dbname=tienda user=app password=clave")
cursor = conn.cursor()

cursor.execute("SELECT id, nombre, precio, stock FROM productos WHERE id = %s", (producto
fila = cursor.fetchone()

if fila:
    producto = {
        "id": fila[0],
        "nombre": fila[1],
        "precio": fila[2],
        "stock": fila[3]
    }

cursor.close()
conn.close()
```

Funciona. Pero hay que repetir esta estructura para cada tabla, cada operación.

Lo que se repite siempre

- Abrir y cerrar la conexión.
- Escribir SQL para cada operación: SELECT, INSERT, UPDATE, DELETE.
- Mapear manualmente cada columna a un campo del diccionario o objeto.
- Gestionar errores y transacciones de forma manual.
- Sincronizar el esquema SQL con la estructura de datos en el código.

Con 10 tablas, el código de acceso a datos se vuelve masivo.

La mezcla de responsabilidades

El código de negocio y el código SQL quedan mezclados.

```
# ¿Esto es lógica de negocio o SQL?
def aprobar_pedido(pedido_id):
    cursor.execute(
        "UPDATE pedidos SET estado = 'aprobado', fecha_aprobacion = NOW() WHERE id = %s",
        (pedido_id,)
    )
    cursor.execute(
        "UPDATE inventario SET reservado = reservado - cantidad FROM pedido_detalle WHERE
        (pedido_id,)
    )
    conn.commit()
```

La lógica de negocio y el SQL están entrelazados.

Cambiar la base de datos implica reescribir la lógica.

¿Qué es un ORM?

ORM — **Object Relational Mapping** (Mapeo Objeto-Relacional).

Es una capa de abstracción que traduce entre dos mundos:

| Mundo relacional | Mundo orientado a objetos | | --- | --- | | Tabla | Clase | | Fila | Objeto / instancia | | Columna | Atributo | | Clave primaria | Identificador del objeto | | Clave foránea | Referencia a otro objeto | | JOIN | Acceso a objeto relacionado |

La idea central

En lugar de escribir SQL, se trabaja con objetos del lenguaje de programación.

El ORM traduce automáticamente esas operaciones en SQL.

```
# Sin ORM
cursor.execute("SELECT * FROM productos WHERE precio < %s", (100,))

# Con ORM
productos = Producto.query.filter(Producto.precio < 100).all()
```

Mismo resultado, distinta abstracción.

El mapeo en la práctica

Se define la tabla como una clase:

```
class Producto(Base):
    __tablename__ = "productos"

    id      = Column(Integer, primary_key=True)
    nombre  = Column(String(200), nullable=False)
    precio  = Column(Numeric(10, 2), nullable=False)
    stock   = Column(Integer, default=0)
```

A partir de esa definición, el ORM sabe cómo construir el SQL necesario.

ORMs más usados

SQLAlchemy — Python

El ORM más completo y flexible para Python.

Compatible con PostgreSQL, MySQL, SQLite, Oracle, Microsoft SQL Server y más.

Tiene dos modos: ORM completo (con clases) y SQL Expression Language (más cercano al SQL puro).

Ampliamente usado en aplicaciones Flask, FastAPI y scripts de análisis de datos.

Django ORM — Python

ORM integrado en el framework Django.

No requiere configuración adicional — viene incluido.

Muy productivo para desarrollo web: generación automática de migraciones y panel de administración.

Compatible con PostgreSQL, MySQL, SQLite y Oracle.

Eloquent — PHP / Laravel

ORM del framework Laravel.

Usa el patrón Active Record: cada modelo es una clase que representa una tabla.

Sintaxis muy fluida y expresiva.

Compatible con MySQL, PostgreSQL, SQLite y SQL Server.

Hibernate — Java

El ORM más maduro del ecosistema Java.

Base de JPA (Jakarta Persistence API), el estándar de persistencia en Java.

Ampliamente usado en aplicaciones empresariales.

Compatible con prácticamente todos los motores relacionales.

Prisma — Node.js / TypeScript

ORM moderno para el ecosistema JavaScript/TypeScript.

Genera un cliente tipado a partir del esquema — errores detectados en tiempo de compilación.

Compatible con PostgreSQL, MySQL, SQLite, MongoDB y SQL Server.

Tabla comparativa

| ORM | Lenguaje | Estilo | Compatibilidad principal | | --- | --- | --- | --- | | SQLAlchemy | Python | Data Mapper | PostgreSQL, MySQL, SQLite | | Django ORM | Python | Active Record | PostgreSQL, MySQL, SQLite | | Eloquent | PHP | Active Record | MySQL, PostgreSQL, SQLite | | Hibernate | Java | Data Mapper | PostgreSQL, MySQL, Oracle | | Prisma | TypeScript | Data Mapper | PostgreSQL, MySQL, SQLite |

SQLAlchemy con PostgreSQL y MySQL

Configuración de conexión

```
from sqlalchemy import create_engine

# PostgreSQL
```

```
engine = create_engine("postgresql+psycopg2://usuario:clave@localhost/tienda")

# MySQL
engine = create_engine("mysql+pymysql://usuario:clave@localhost/tienda")
```

Solo cambia la cadena de conexión — el código ORM es idéntico para ambos motores.

Definir un modelo

```
from sqlalchemy import Column, Integer, String, Numeric, ForeignKey
from sqlalchemy.orm import DeclarativeBase, relationship

class Base(DeclarativeBase):
    pass

class Categoria(Base):
    __tablename__ = "categorias"

    id     = Column(Integer, primary_key=True)
    nombre = Column(String(100), nullable=False, unique=True)

    productos = relationship("Producto", back_populates="categoria")


class Producto(Base):
    __tablename__ = "productos"

    id           = Column(Integer, primary_key=True)
    nombre       = Column(String(200), nullable=False)
    precio       = Column(Numeric(10, 2), nullable=False)
    stock        = Column(Integer, default=0)
    categoria_id = Column(Integer, ForeignKey("categorias.id"))

    categoria = relationship("Categoria", back_populates="productos")
```

Insertar un registro

```
from sqlalchemy.orm import Session

with Session(engine) as session:
    nuevo = Producto(
        nombre="Laptop",
        precio=5000.00,
        stock=10,
        categoria_id=1
    )
    session.add(nuevo)
    session.commit()
```

El ORM genera internamente:

```
INSERT INTO productos (nombre, precio, stock, categoria_id)
VALUES ('Laptop', 5000.00, 10, 1)
```

Consultar registros

```
with Session(engine) as session:

    # Todos los productos
    todos = session.query(Producto).all()

    # Por ID
    uno = session.get(Producto, 5)

    # Con filtro
    baratos = session.query(Producto).filter(Producto.precio < 1000).all()

    # Múltiples condiciones
    disponibles = session.query(Producto).filter(
        Producto.stock > 0,
        Producto.precio < 2000
    ).order_by(Producto.precio).all()
```

El SQL que genera el ORM

```
session.query(Producto).filter(
    Producto.stock > 0,
    Producto.precio < 2000
).order_by(Producto.precio).all()
```

Genera:

```
SELECT productos.id, productos.nombre, productos.precio,
       productos.stock, productos.categoria_id
FROM productos
WHERE productos.stock > 0
  AND productos.precio < 2000
ORDER BY productos.precio ASC
```

Actualizar un registro

```
with Session(engine) as session:
    producto = session.get(Producto, 5)

    if producto:
        producto.precio = 4500.00
        producto.stock = 15
        session.commit()
```

El ORM detecta los cambios y genera:

```
UPDATE productos
SET precio = 4500.00, stock = 15
WHERE productos.id = 5
```

Solo se actualiza lo que cambió.

Eliminar un registro

```
with Session(engine) as session:
    producto = session.get(Producto, 5)

    if producto:
        session.delete(producto)
        session.commit()
```

Genera:

```
DELETE FROM productos WHERE productos.id = 5
```

Relaciones: acceder a datos relacionados

```
with Session(engine) as session:
    producto = session.get(Producto, 5)

    # Acceder a la categoría sin escribir un JOIN
    print(producto.nombre)
    print(producto.categoria.nombre)
```

El ORM ejecuta el JOIN automáticamente al acceder a `.categoria`.

```
SELECT categorias.id, categorias.nombre
FROM categorias
WHERE categorias.id = 1
```

Eloquent con MySQL y PostgreSQL

Definir un modelo

```
namespace App\Models;
```

```php
use Illuminate\Database\Eloquent\Model;

class Producto extends Model
{
    protected $table = 'productos';

    protected $fillable = [
        'nombre',
        'precio',
        'stock',
        'categoria_id',
    ];

    public function categoria()
    {
        return $this->belongsTo(Categoria::class);
    }
}
```

Operaciones básicas

```php
// Insertar
$producto = Producto::create([
    'nombre'       => 'Laptop',
    'precio'       => 5000.00,
    'stock'        => 10,
    'categoria_id' => 1,
]);

// Buscar por ID
$producto = Producto::find(5);

// Con filtros
$baratos = Producto::where('precio', '<', 1000)
                   ->where('stock', '>', 0)
                   ->orderBy('precio')
                   ->get();
```

```
// Actualizar
$producto->precio = 4500.00;
$producto->save();

// Eliminar
$producto->delete();
```

Acceso a relaciones

```
$producto = Producto::find(5);

echo $producto->nombre;
echo $producto->categoria->nombre;  // JOIN automático
```

Cambiar de MySQL a PostgreSQL

En el archivo de configuración `.env`:

```
# MySQL
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306

# PostgreSQL
DB_CONNECTION=pgsql
DB_HOST=127.0.0.1
DB_PORT=5432
```

El código de los modelos no cambia.

Eloquent adapta el SQL al motor configurado.

Seguridad

El problema central: SQL Injection

Ya se vio que construir SQL con concatenación de strings es la causa raíz de SQL Injection.

```
# Vulnerable — datos del usuario directamente en el SQL
query = f"SELECT * FROM usuarios WHERE email = '{email}'"
```

Los ORMs eliminan ese patrón por diseño.

Cómo los ORMs previenen SQL Injection

Los ORMs nunca concatenan los valores del usuario directamente en el SQL.

Siempre usan consultas parametrizadas internamente.

```
# El desarrollador escribe esto
session.query(Producto).filter(Producto.nombre == nombre_buscado).all()

# El ORM genera esto (parametrizado)
# SELECT * FROM productos WHERE nombre = $1
# Parámetro: nombre_buscado
```

El valor de `nombre_buscado` nunca se incrusta en el SQL — llega separado al motor.

Demostración: input malicioso

```
nombre_buscado = "'; DROP TABLE productos; --"

# Sin ORM — vulnerable
query = f"SELECT * FROM productos WHERE nombre = '{nombre_buscado}'"
# Ejecuta: SELECT * FROM productos WHERE nombre = ''; DROP TABLE productos; --'

# Con ORM — seguro
```

```
session.query(Producto).filter(Producto.nombre == nombre_buscado).all()
# El ORM genera: SELECT * FROM productos WHERE nombre = $1
# Parámetro: "'; DROP TABLE productos; --"  (tratado como texto, no como SQL)
```

La tabla no se elimina. El input malicioso es buscado literalmente como nombre.

Protección automática en todas las operaciones

Los ORMs parametrizan todos los valores: filtros, inserciones, actualizaciones.

```
# Todo parametrizado automáticamente
session.add(Producto(nombre=nombre, precio=precio))     # INSERT parametrizado
session.query(Producto).filter(Producto.id == id)       # SELECT parametrizado
producto.precio = nuevo_precio; session.commit()         # UPDATE parametrizado
session.delete(producto)                                 # DELETE por PK
```

El desarrollador no necesita recordar parametrizar — el ORM lo hace por defecto.

La excepción: raw queries

Los ORMs permiten escribir SQL crudo cuando es necesario.

Ahí puede reaparecer la vulnerabilidad si no se tiene cuidado.

```
# SQLAlchemy — vulnerable
resultado = session.execute(
    text(f"SELECT * FROM productos WHERE nombre = '{nombre}'")
)

# SQLAlchemy — seguro
resultado = session.execute(
    text("SELECT * FROM productos WHERE nombre = :nombre"),
    {"nombre": nombre}
)
```

```
// Eloquent — vulnerable
$productos = DB::select("SELECT * FROM productos WHERE nombre = '$nombre'");

// Eloquent — seguro
$productos = DB::select("SELECT * FROM productos WHERE nombre = ?", [$nombre]);
```

Las raw queries en un ORM tienen las mismas reglas que el SQL directo.

Beneficios del ORM

1. Productividad

Operaciones CRUD en pocas líneas, sin SQL repetitivo.

La lógica de acceso a datos queda en el modelo — un solo lugar.

Cambios en el esquema se propagan al código automáticamente con las migraciones.

2. Independencia del motor de base de datos

El mismo código funciona con PostgreSQL, MySQL y SQLite.

Cambiar de motor es cuestión de modificar la cadena de conexión.

Facilita desarrollo local (SQLite) y producción (PostgreSQL o MySQL).

3. Seguridad por defecto

Las consultas están parametrizadas automáticamente.

No depende de que cada desarrollador recuerde hacerlo.

La protección contra SQL Injection es estructural, no opcional.

4. Migraciones

Los ORMs generan scripts de migración a partir de los cambios en los modelos.

```
# Django
python manage.py makemigrations
python manage.py migrate

# Laravel
php artisan make:migration add_descuento_to_productos
php artisan migrate
```

El historial de cambios del esquema queda versionado junto con el código.

5. Relaciones como objetos

No hay que escribir JOINs manualmente.

Se accede a los datos relacionados como atributos del objeto.

```
# El ORM hace el JOIN cuando se accede a .categoria
print(producto.categoria.nombre)
```

6. Validaciones integradas

Los modelos pueden definir reglas de validación antes de llegar a la base de datos.

```
class Producto(Base):
    precio = Column(Numeric(10, 2), nullable=False)
    # nullable=False → el ORM rechaza None antes de intentar el INSERT
```

```
// Laravel — validación en el controlador antes de guardar
$request->validate([
```

```
    'precio' => 'required|numeric|min:0',
    'stock'  => 'required|integer|min:0',
]);
```

Tabla de beneficios

| Beneficio | Sin ORM | Con ORM | | --- | --- | --- | | SQL Injection | Manual — depende del desarrollador | Automático — parametrizado siempre | | Código repetitivo | Alto — SELECT, INSERT, UPDATE, DELETE para cada tabla | Bajo — el ORM genera el SQL | | Cambio de motor DB | Reescribir todas las consultas | Cambiar la cadena de conexión | | Migraciones | Scripts SQL manuales | Generados automáticamente | | Relaciones | JOINs escritos a mano | Acceso como atributos |

Limitaciones del ORM

Rendimiento en consultas complejas

El SQL generado por el ORM no siempre es óptimo.

Para consultas complejas (múltiples JOINs, agregaciones, subconsultas anidadas), el SQL escrito manualmente suele ser más eficiente.

Los ORMs a veces generan N+1 queries: una consulta por cada objeto relacionado.

```
# N+1 — hace 1 SELECT para productos + 1 SELECT por cada producto para su categoría
for producto in session.query(Producto).all():
    print(producto.categoria.nombre)  # query por cada producto

# Correcto — eager loading: 1 JOIN en lugar de N+1
from sqlalchemy.orm import joinedload
productos = session.query(Producto).options(joinedload(Producto.categoria)).all()
```

Curva de aprendizaje

Los ORMs tienen su propia API que hay que aprender.

Se necesita entender el modelo relacional para usar el ORM correctamente.

Un desarrollador que no entiende SQL puede generar consultas muy ineficientes sin saberlo.

Abstracción que oculta lo que sucede

El ORM hace invisible el SQL.

Esto es bueno para la productividad, pero peligroso si no se revisa el SQL generado.

```
# ¿Cuántas queries ejecuta esto?
for pedido in session.query(Pedido).all():
    total = sum(d.cantidad * d.precio for d in pedido.detalles)
```

Puede ejecutar cientos de queries sin que el desarrollador lo note.

No todo se puede expresar con el ORM

Funciones específicas del motor (ventanas, arrays, full-text search) a veces requieren SQL crudo.

Procedimientos almacenados, triggers y funciones complejas están fuera del alcance del ORM.

Para esos casos, los ORMs permiten combinar con SQL directo.

ORM y base de datos en producción

El ORM no reemplaza el conocimiento de SQL

Para depurar problemas de rendimiento, hay que leer el SQL generado.

SQLAlchemy:

```
# Ver el SQL que genera una consulta
print(session.query(Producto).filter(Producto.precio < 1000))
```

Django ORM:

```
print(Producto.objects.filter(precio__lt=1000).query)
```

Un ORM mal usado es más lento que SQL directo.

Separación de responsabilidades

Los modelos del ORM deben mapear el esquema.

Las reglas de negocio complejas pueden seguir viviendo en procedimientos almacenados.

La combinación es válida y a veces la más eficiente.

```
# Llamar a un procedimiento almacenado desde el ORM
session.execute(text("CALL procesar_inventario_diario()"))
```

Índices y rendimiento

El ORM no crea índices automáticamente (salvo la clave primaria).

Los índices deben definirse explícitamente o en la migración.

```
class Producto(Base):
    __tablename__ = "productos"

    id     = Column(Integer, primary_key=True)
    nombre = Column(String(200), index=True)  # crea índice en nombre
```

Sin los índices correctos, las consultas del ORM serán lentas igual que el SQL directo.

¿Qué puede salir mal?

- **Raw queries con concatenación** → SQL Injection exactamente igual que sin ORM
- **N+1 queries sin eager loading** → una query por cada objeto relacionado, rendimiento catastrófico
- **No revisar el SQL generado** → queries ineficientes que pasan desapercibidas
- **Confiar en el ORM para crear índices** → los índices deben definirse explícitamente
- **Modelos que no reflejan el esquema real** → errores silenciosos o datos incorrectos
- **Transacciones implícitas mal manejadas** → datos parcialmente guardados ante un error
- **Usar el ORM para todo, incluyendo consultas complejas** → SQL directo es más claro y eficiente en esos casos
- **No entender SQL porque "el ORM lo hace"** → imposible depurar, optimizar o detectar problemas reales