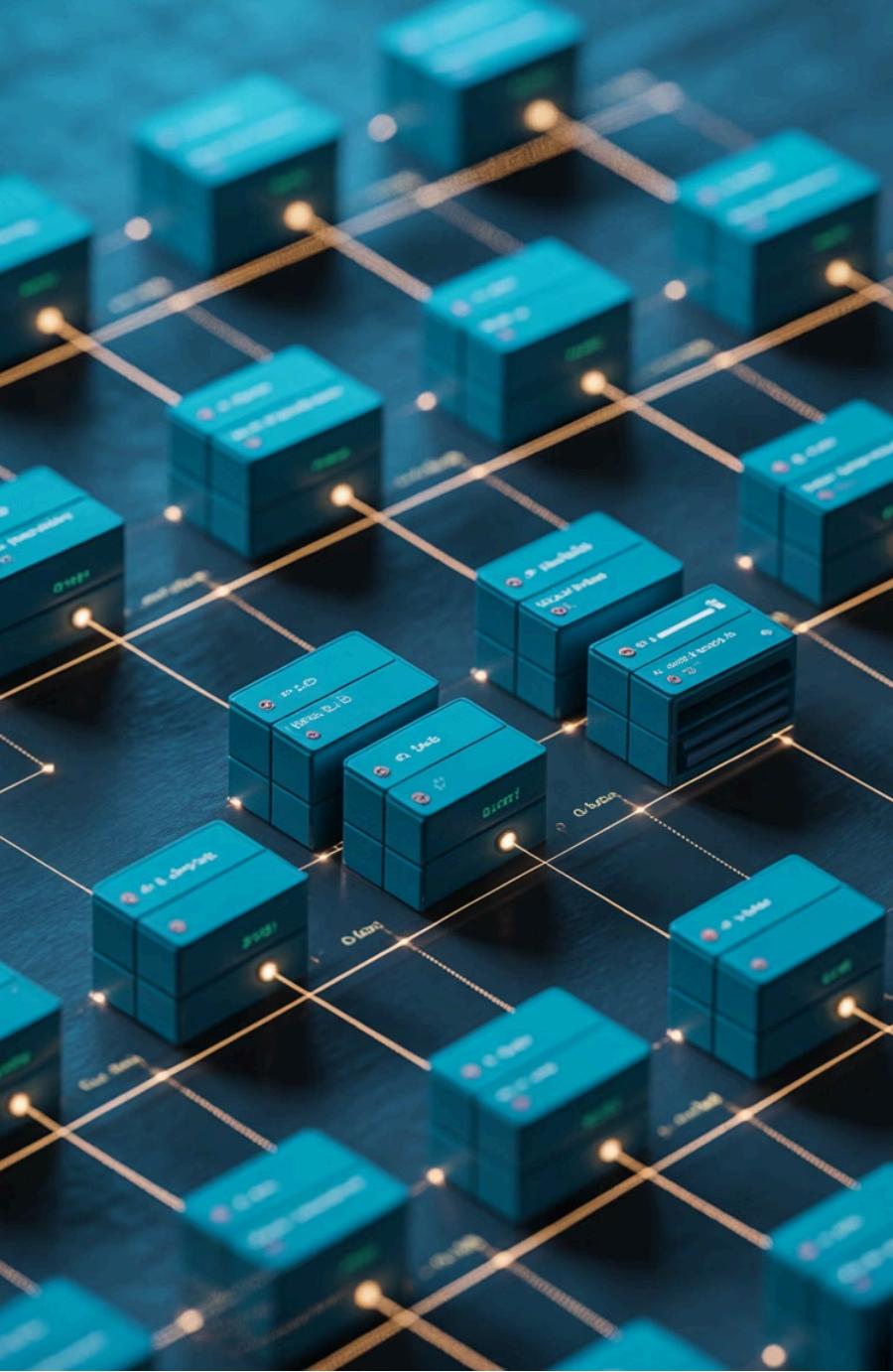


Consultas de datos y SQL personalizado en Django

En esta sesión exploraremos técnicas avanzadas para recuperar y manipular datos en aplicaciones Django. Aprenderás a dominar el ORM para consultas eficientes, aplicar filtros precisos, ejecutar SQL personalizado cuando sea necesario, y optimizar el rendimiento de tus consultas mediante índices, anotaciones y otras estrategias profesionales.





Introducción al ORM de Django

¿Qué es el ORM?

El Object Relational Mapper (ORM) es el puente entre tu código Python y la base de datos. Cada modelo Django representa una tabla completa, y cada instancia del modelo corresponde a una fila específica en esa tabla.

Esta abstracción te permite trabajar con objetos Python en lugar de escribir SQL manualmente, mejorando la productividad y legibilidad del código.

Métodos principales

.all() - Recupera todos los registros

.filter() - Obtiene registros que cumplen condiciones

.exclude() - Excluye registros específicos

```
from clientes.models import Cliente
clientes = Cliente.objects.all()
for c in clientes:
    print(c.nombre)
```

Ventajas del ORM de Django



Abstracción Elegante

Evita escribir SQL repetitivo y tedioso. El ORM traduce automáticamente tus llamadas Python a consultas SQL optimizadas, permitiéndote concentrarte en la lógica de negocio.



Seguridad Incorporada

Previene automáticamente ataques de inyección SQL mediante el uso de consultas parametrizadas. Los datos del usuario nunca se interpolan directamente en las consultas.



Compatibilidad Multiplataforma

Cambia entre PostgreSQL, MySQL, SQLite u otros motores sin modificar tu código. El ORM adapta las consultas al dialecto SQL específico de cada motor.



Mayor Productividad

Las consultas son más legibles, mantenibles y fáciles de depurar. El código orientado a objetos se integra naturalmente con el resto de tu aplicación Django.

Recuperando registros con ORM

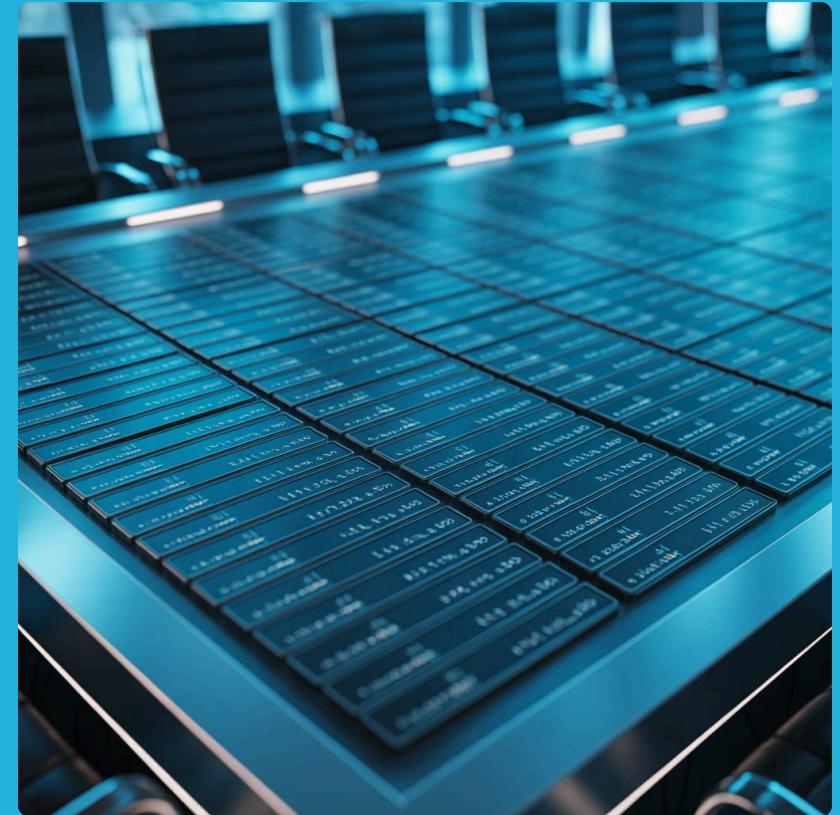
Método .all()

El método `.all()` es la forma más directa de recuperar todos los registros de una tabla. Devuelve un QuerySet iterable que puedes recorrer con un bucle `for` estándar de Python.

Este QuerySet es `lazy`, lo que significa que la consulta a la base de datos no se ejecuta hasta que realmente necesitas los datos, optimizando el rendimiento de tu aplicación.

```
clientes = Cliente.objects.all()
for cliente in clientes:
    print(cliente.nombre, cliente.email)
```

Cada iteración del bucle accede a los atributos del modelo como propiedades Python normales, haciendo el código intuitivo y fácil de leer.



Filtrando registros con .filter()

El método `.filter()` es tu herramienta principal para obtener subconjuntos específicos de datos. Acepta argumentos de palabra clave que coinciden con los campos de tu modelo, y solo devuelve los registros que cumplen todas las condiciones especificadas.

Ejemplo práctico

```
from ordenes.models import Orden

ordenes_pagadas = Orden.objects.filter(
    estado='pagada'
)

for orden in ordenes_pagadas:
    print(orden.id, orden.cliente.nombre)
```

Puedes combinar múltiples condiciones en un solo `filter()`, y todas deben cumplirse (operación AND implícita).



- ❑ Actividad: Filtra órdenes creadas en los últimos 30 días usando `fecha__gte` con `datetime.now() - timedelta(days=30)`



Tipos de filtros útiles en Django

Django proporciona una amplia variedad de lookups (búsquedas) que se añaden a los nombres de campo usando doble guion bajo. Estos lookups permiten consultas sofisticadas sin escribir SQL complejo.

Comparaciones exactas

exact - Coincidencia exacta (predeterminado)

```
Cliente.objects.filter(  
    nombre_exact='Juan'  
)
```

Búsquedas de texto

contains - Contiene texto

startswith - Comienza con

endswith - Termina con

```
Cliente.objects.filter(  
    nombre_contains='Juan'  
)
```

Comparaciones numéricas

gte - Mayor o igual que

lte - Menor o igual que

gt - Mayor que

lt - Menor que

```
Orden.objects.filter(  
    total_gte=100  
)
```

Pertenencia a listas

in - Valor en lista

```
Orden.objects.filter(  
    estado_in=[  
        'pagada',  
        'enviada'  
    ]  
)
```

Consultas SQL personalizadas con .raw()

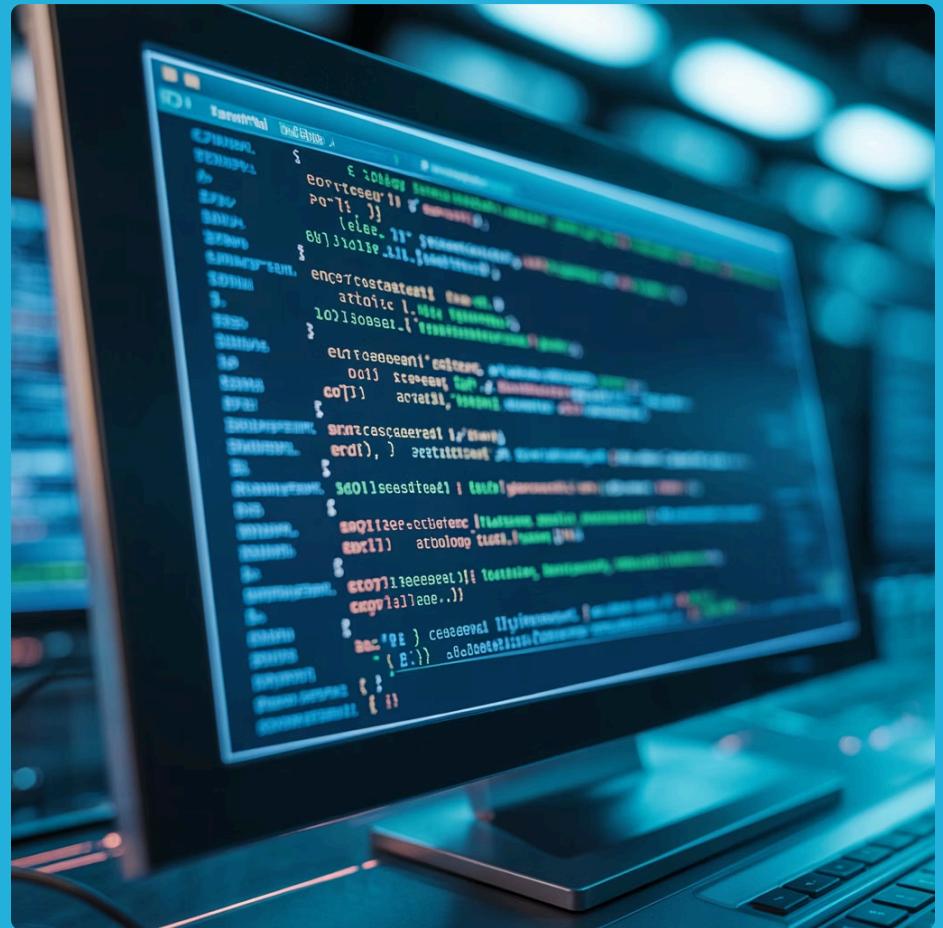
Cuándo usar SQL directo

Aunque el ORM de Django es poderoso, hay ocasiones donde necesitas ejecutar SQL personalizado: consultas muy complejas, funciones específicas de la base de datos, o cuando necesitas optimización máxima para casos especiales.

El método `.raw()` te permite ejecutar SQL personalizado mientras mantiene el mapeo automático a instancias de modelo, combinando lo mejor de ambos mundos.

Ejemplo de uso

```
ordenes = Orden.objects.raw(  
    "SELECT * FROM ordenes_orden  
    WHERE cliente_id = %s",  
    [1]  
)  
  
for o in ordenes:  
    print(o.id, o.total)
```



 **Importante:** Siempre usa parámetros (`%s`) en lugar de interpolación de strings para prevenir inyección SQL.

Notas importantes sobre .raw()

Coincidencia de columnas

Los nombres de columnas en tu consulta SQL deben coincidir exactamente con los nombres de campo definidos en tu modelo Django. Si los nombres difieren, Django no podrá mapear correctamente los resultados a las instancias del modelo.

Parámetros seguros

Siempre pasa los parámetros como el segundo argumento en forma de lista o tupla. Django los escapará automáticamente, previniendo ataques de inyección SQL. Nunca uses formateo de strings (f-strings o .format()).

```
Orden.objects.raw(  
    "SELECT * FROM ordenes WHERE  
    id = %s",  
    [orden_id] # ✓ Correcto  
)
```

Consideraciones de rendimiento

Los QuerySets de .raw() siguen siendo lazy y soportan iteración eficiente. Sin embargo, no puedes aplicar métodos adicionales del ORM como .filter() o .order_by() sobre ellos.

Optimizando consultas: solo campos necesarios

Una de las optimizaciones más efectivas es recuperar únicamente los datos que realmente necesitas. Cargar campos innecesarios consume memoria y tiempo de transferencia de la base de datos.

Método .values()

Devuelve diccionarios en lugar de instancias de modelo, conteniendo solo los campos especificados. Ideal cuando no necesitas los métodos del modelo.

```
clientes = Cliente.objects.values(  
    "nombre",  
    "email"  
)  
  
for cliente in clientes:  
    print(cliente['nombre'])
```

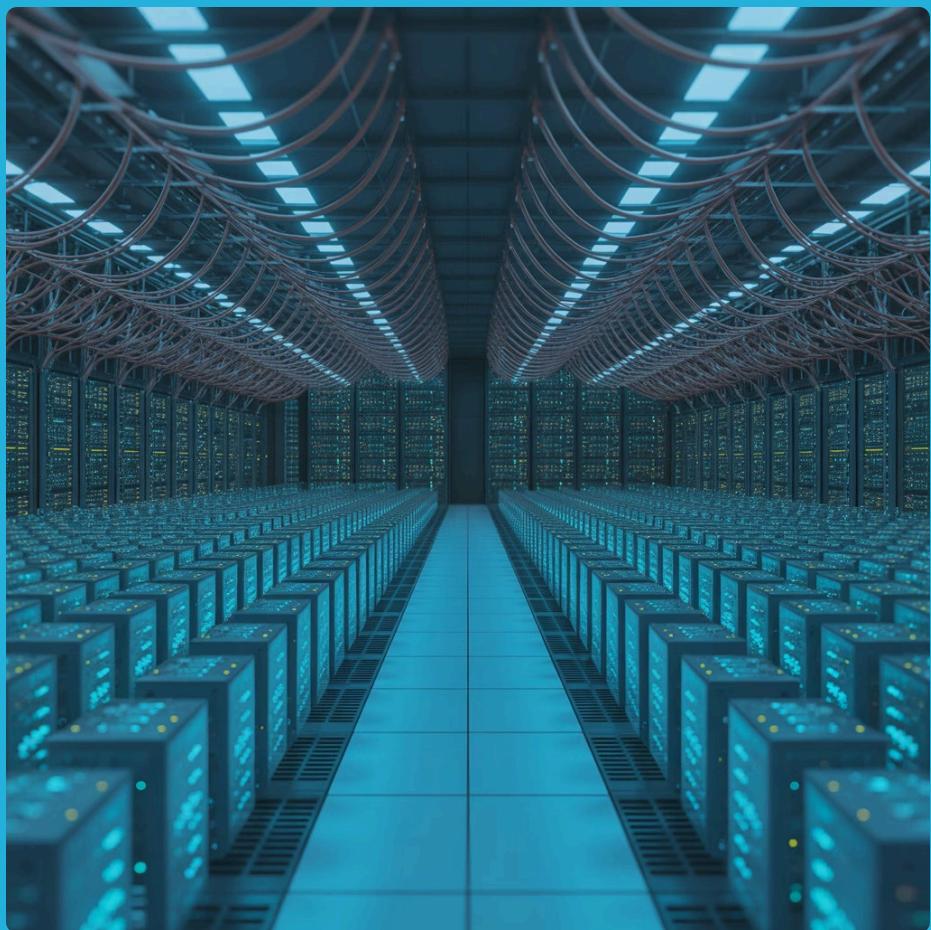
Método .only()

Similar a values(), pero devuelve instancias del modelo completas. Los campos no especificados se cargarán bajo demanda si se acceden.

```
clientes = Cliente.objects.only(  
    "nombre",  
    "email"  
)
```



Excluir campos pesados con .defer()



Optimización inversa

Mientras que `.only()` especifica qué campos cargar, `.defer()` hace lo contrario: especifica qué campos NO cargar inicialmente. Esto es especialmente útil cuando tienes campos muy grandes como texto largo, JSON o datos binarios.

Ejemplo práctico

```
clientes = Cliente.objects.defer(  
    "direccion",  
    "historial_compras"  
)  
  
for cliente in clientes:  
    # Estos campos se cargan normalmente  
    print(cliente.nombre)  
  
    # Este campo se carga solo si se accede  
    # (genera consulta adicional)  
    if necesario:  
        print(cliente.direccion)
```

Los campos diferidos se cargarán automáticamente si los accedes, pero la consulta inicial es más rápida y ligera.

Índices para mejorar rendimiento

Los índices de base de datos son estructuras que aceleran dramáticamente las búsquedas en campos específicos. Sin índices, la base de datos debe escanear toda la tabla para encontrar registros.



Identificar campos frecuentes

Analiza qué campos usas más frecuentemente en filtros, búsquedas y ordenamientos. Estos son candidatos ideales para indexación.



Añadir db_index=True

En tu modelo Django, añade el parámetro db_index=True a los campos que necesitan búsquedas rápidas.

```
class Cliente(models.Model):
    nombre = models.CharField(
        max_length=100,
        db_index=True
    )
    email = models.EmailField(
        db_index=True
    )
```



Ejecutar migraciones

Genera y ejecuta las migraciones para que Django cree los índices en la base de datos.

```
python manage.py makemigrations
python manage.py migrate
```



Verificar mejora

Los índices pueden acelerar las consultas de 10x a 1000x en tablas grandes, especialmente en campos únicos o con alta cardinalidad.

Anotaciones y agregaciones

El método `.annotate()` permite añadir campos calculados a cada objeto del `QuerySet` mediante funciones de agregación. Esto es increíblemente útil para realizar cálculos complejos directamente en la base de datos en lugar de en Python.

Funciones de agregación

Count - Cuenta registros relacionados

Sum - Suma valores numéricos

Avg - Calcula promedio

Max/Min - Valores máximo/mínimo

Ejemplo con Count

```
from django.db.models import Count

clientes = Cliente.objects.annotate(
    total_orden=Count("ordenes")
)

for c in clientes:
    print(
        f"{c.nombre}: {c.total_orden}"
    )
```



- ❑ Actividad: Usa `.filter()` después de `.annotate()` para obtener solo clientes con más de 3 órdenes: `filter(total_orden__gt=3)`

Ejecutando SQL directamente con cursos

Para operaciones que requieren control total o que no mapean a modelos, Django proporciona cursos de base de datos. Estos son especialmente útiles para operaciones de escritura masivas, procedimientos almacenados o consultas que no devuelven modelos.

1

Importar connection

El objeto `connection` provee acceso directo a la base de datos configurada en `settings.py`

2

Crear cursor

Usa context manager (`with`) para manejar automáticamente el cierre del cursor

3

Ejecutar SQL

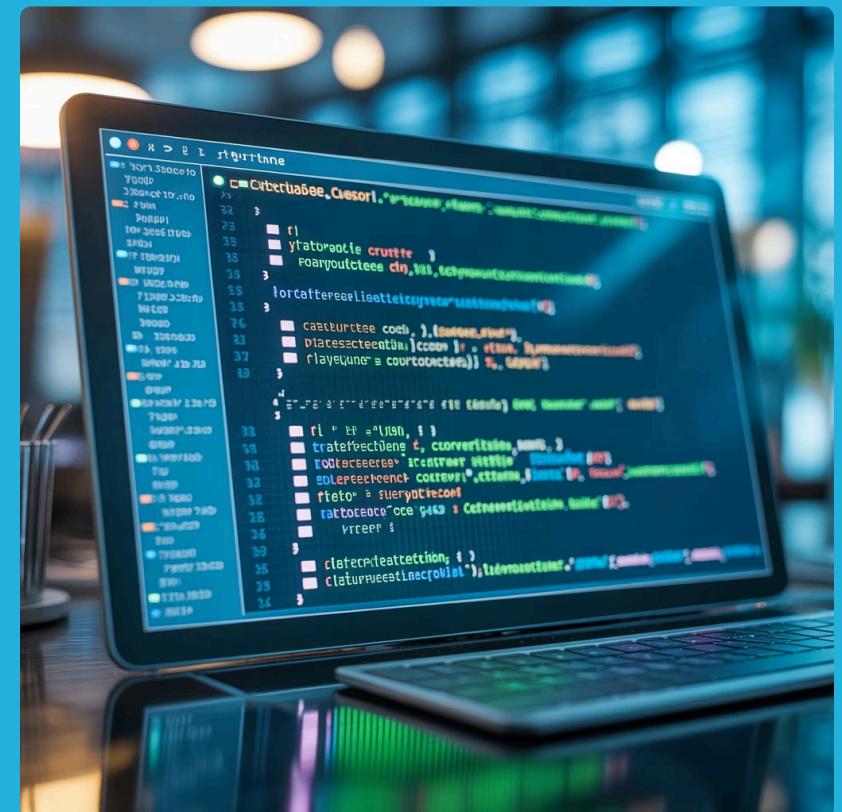
Llama a `cursor.execute()` con tu consulta SQL y parámetros

Ejemplo de actualización masiva

```
from django.db import connection

with connection.cursor() as cursor:
    cursor.execute(
        """UPDATE ordenes_orden
           SET total = total + 100
           WHERE estado = %s""",
        ['pendiente']
    )

    # Obtener filas afectadas
    rows = cursor.rowcount
    print(f"Actualizadas: {rows}")
```



Llamando procedimientos almacenados

Los procedimientos almacenados son funciones que viven en la base de datos y encapsulan lógica compleja. Son ideales para operaciones que requieren múltiples pasos o que deben ejecutarse de forma atómica en el servidor de base de datos.

Ventajas de los procedimientos

- Reutilización de lógica de negocio
- Mejor rendimiento para operaciones complejas
- Seguridad mediante encapsulación
- Reducción de tráfico de red

Ejemplo de invocación

```
from django.db import connection

with connection.cursor() as cursor:
    # Llamar función almacenada
    cursor.execute(
        "SELECT contar_ordenes_cliente(%s)",
        [cliente_id]
    )

    # Obtener resultado
    total = cursor.fetchone()[0]
    print(f"Total órdenes: {total}")
```



❑ Nota: Los procedimientos almacenados son específicos del motor de base de datos y reducen la portabilidad de tu aplicación.

Buenas prácticas: ORM vs SQL

1

Preferir ORM para consultas comunes

El ORM es más legible, mantenible y seguro. Úsallo siempre que sea posible para operaciones CRUD estándar y consultas simples.

2

Parámetros en consultas raw

Nunca concatenes strings para construir SQL. Siempre usa parámetros (%s) para prevenir inyección SQL y garantizar seguridad.

3

Evitar consultas N+1

Usa `select_related()` para relaciones `ForeignKey/OneToOne` y `prefetch_related()` para `ManyToMany`. Esto reduce drásticamente el número de consultas.

4

Indexar campos estratégicos

Añade `db_index=True` a campos que usas frecuentemente en filtros, búsquedas y ordenamientos para mejorar significativamente el rendimiento.

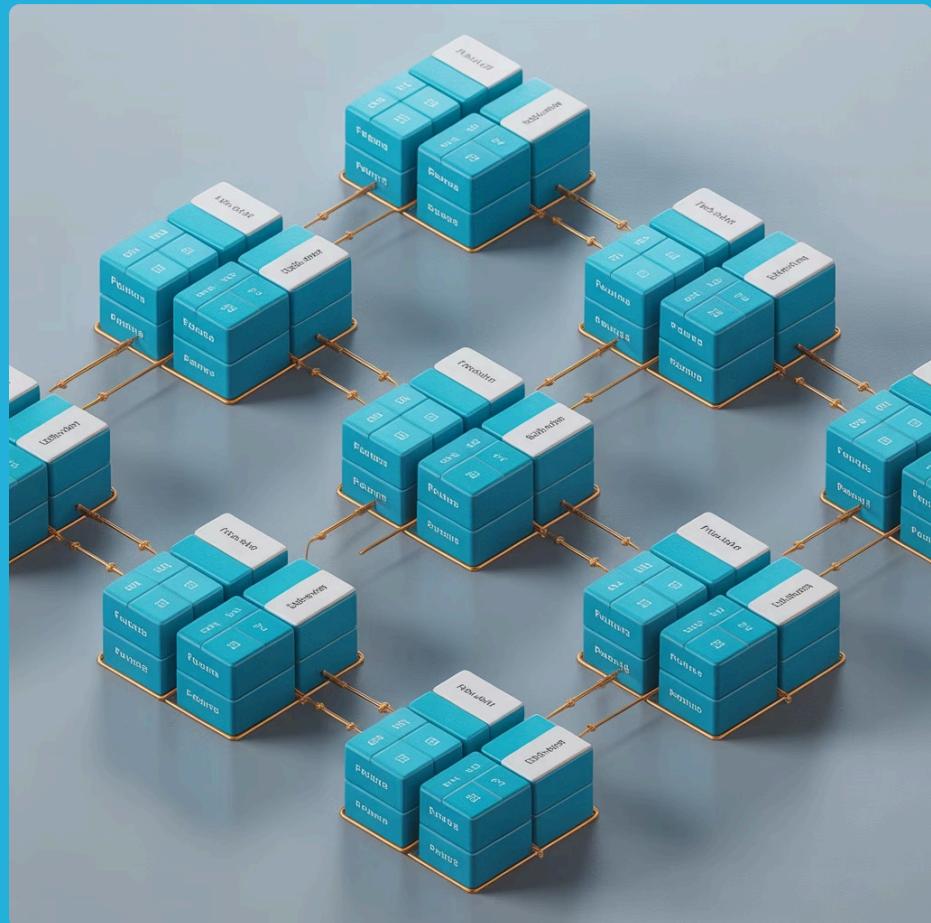
5

Monitorear rendimiento

Usa Django Debug Toolbar o logging de consultas para identificar queries lentas. Optimiza las más frecuentes o costosas primero.

Ejercicio práctico 1

Optimizar carga de relaciones con select_related()



Problema sin optimización

#

Objetivo

Recuperar todos los clientes junto con sus perfiles asociados en una sola consulta SQL, evitando el problema N+1.

Ejercicio práctico 2

Contar registros relacionados con annotate()

Paso 1: Importar Count

Importa la función de agregación Count desde django.db.models para poder contar registros relacionados.

```
from django.db.models import Count
```

Paso 2: Aplicar annotate

Usa annotate() para añadir un campo calculado que cuente las órdenes relacionadas de cada cliente.

```
clientes = Cliente.objects.annotate(  
    total_orden=Count("ordenes")  
)
```

Paso 3: Iterar resultados

Cada cliente ahora tiene un atributo temporal total_orden que puedes usar directamente.

```
for c in clientes:  
    print(  
        f"{c.nombre}: "  
        f"{c.total_orden} órdenes"  
)
```

- ❑ Bonus: Filtra para obtener solo clientes VIP: `.filter(total_orden_gte=5)`

Ejercicio práctico 3

Consulta SQL raw para bicicletas de una orden

Escenario

Necesitas recuperar todas las bicicletas asociadas a una orden específica usando SQL personalizado en lugar del ORM. Esto simula situaciones donde tienes consultas heredadas o muy optimizadas.

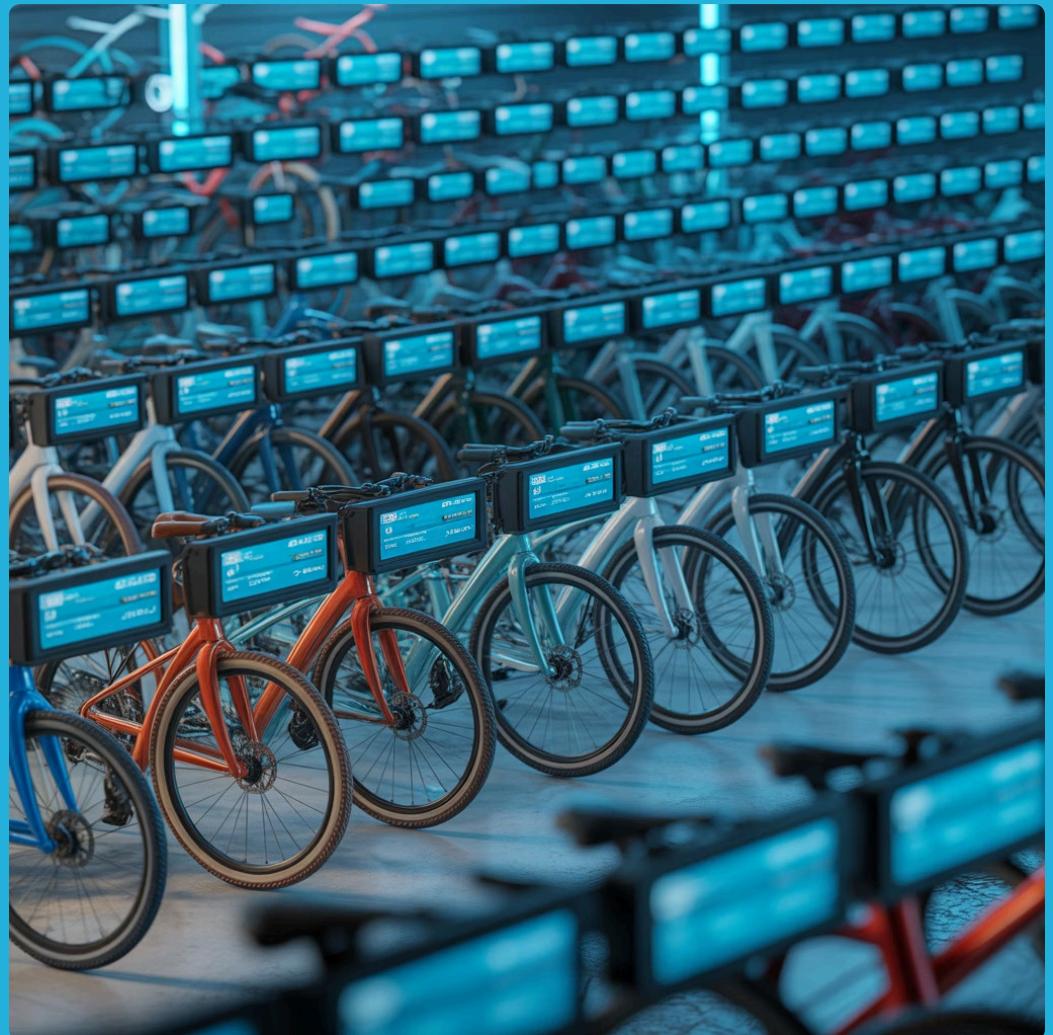
Código guía

```
from bicis.models import Bicicleta

orden_id = 2

bicis = Bicicleta.objects.raw(
    """SELECT *
       FROM bicis_bicicleta
      WHERE orden_id = %s""",
    [orden_id]
)

# Iterar resultados mapeados
for bici in bicis:
    print(
        f"Bici #{bici.id}: "
        f"{bici.modelo} - "
        f"${bici.precio}"
    )
```



Puntos clave

- El **SELECT *** es necesario para mapear todos los campos del modelo
- Los nombres de tabla deben coincidir con la estructura real de la DB
- Los resultados se mapean automáticamente a instancias de **Bicicleta**
- Puedes acceder a todos los atributos y métodos del modelo

Ejercicio práctico 4

Optimizar con defer() y medir rendimiento

En este ejercicio compararás el rendimiento de cargar todos los campos versus excluir un campo pesado usando `defer()`. Esta técnica es crítica cuando trabajas con modelos que tienen campos grandes como texto largo, JSON o datos binarios.

Consulta completa

```
import time

# Cargar todos los campos
start = time.time()
clientes = list(
    Cliente.objects.all()
)
tiempo_completo = time.time() - start

print(f"Completo: {tiempo_completo:.3f}s")
```

Análisis

```
# Calcular mejora
mejora = (
    (tiempo_completo - tiempo_defer)
    / tiempo_completo * 100
)

print(f"Mejora: {mejora:.1f}%")
```

1

2

3

Consulta optimizada

```
# Excluir campo pesado
start = time.time()
clientes = list(
    Cliente.objects.defer(
        "historial"
    )
)
tiempo_defer = time.time() - start

print(f"Defer: {tiempo_defer:.3f}s")
```

Resultado esperado: Deberías ver una mejora del 20-50% dependiendo del tamaño del campo `historial` y el número de registros. En tablas grandes, la diferencia puede ser aún más dramática.

Ejercicio práctico 5 y cierre

Actualización masiva con cursor y verificación

Ejercicio final

Utiliza un cursor para actualizar los totales de todas las órdenes pendientes, añadiendo un cargo de procesamiento de \$50. Luego verifica que los cambios se aplicaron correctamente.

```
from django.db import connection

# Actualización masiva
with connection.cursor() as cursor:
    cursor.execute(
        """UPDATE ordenes_orden
        SET total = total + 50
        WHERE estado = 'pendiente'"""
    )
    filas_afectadas = cursor.rowcount
    print(f"Actualizadas: {filas_afectadas}")

# Verificación con ORM
from ordenes.models import Orden
ordenes = Orden.objects.filter(
    estado='pendiente'
)
for o in ordenes:
    print(f"Orden {o.id}: ${o.total}")
```



Conceptos integrados

- ✓ Uso de cursos para SQL directo
- ✓ Operaciones de escritura masivas
- ✓ Verificación con ORM
- ✓ Transacciones implícitas

Resumen de conceptos clave

ORM Django

Abstracción segura y productiva para consultas comunes

Optimización

Índices, `select_related()`, `defer()` y `annotate()` para rendimiento

SQL Personalizado

Flexibilidad total con `.raw()` y cursos cuando se necesita

Seguridad

Siempre usar parámetros, nunca interpolación de strings

- Debate final: ¿Cuándo preferirías usar SQL directo sobre el ORM? ¿Qué factores consideras: rendimiento, complejidad, mantenibilidad, portabilidad?