ORM DE DJANGO PARA FANS DE SQL

EDICION 2020

RAUL ENRIQUE TORRES REYES

raultr@gmail.com

@raultr

ORM de Django para fans de SQL

Esta es una guía rápida sobre el ORM de Django y las instrucciones SQL que se generan, el propósito de esto es que quienes vienen del mundo SQL puedan introducirse al mundo del ORM por medio de las instrucciones SQL que ya conocemos, también que sirva de ayuda para que quienes nacieron en el mundo ORM puedan saber que sucede a nivel SQL con las instrucciones que usan y de esta manera poder tener un mejor control de el.

En muchos de los temas se mostrará un enlace para que vean en acción en un video el tema en especifico

18/12/2020

Tabla de contenido

1 Preparación de un entorno para probar el ORM de Django

1.1 Programas requeridos

Python 3.6 o mayor

Django 2.2.7 o mayor

Python VirtualEnv

SQLite

IPython

1.2 Preparación de un ambiente virtual en Python

Instalamos virtualenv

```
pip install virtualenv
```

Ejecutamos el ambiente, podemos especificar la versión de Python como en este caso o sino se especifica tomara la versión por defecto

```
virtualenv –p python3.6  .venv
```

Activar el entorno virtual

Linux: `source .venv/bin/actívate`

Windows: `.venv\Scripts\activate`

1.3 Instalacion de Django y extensiones

```
pip install Django=2.2.7
pip install django-extensions
pip install ipython
```

1.4 Creación de un proyecto de Django

```
django-admin startproject nombre_proyecto
```

Inicializar el servidor de Django

```
python manage.py runserver
```

Nos solicitara realizar la primera migracion

```
python manage.py migrate
```

Crear la primera aplicacion de nuestro proyecto

```
python manage.py startapp nombre_aplicacion
```

1.5 Registro de la aplicación y las extensiones en settings.py

Las aplicaciones que creamos dentro del proyecto y las extenciones que instalamos se registrar en el archivo settings.py para que puedan ser usadas.

```
INSTALLED_APPS = [
    'django_extensions',
    'libreria',
]
```

2 Lenguaje de Definición de Datos (DDL)

2.1 Primer modelo básico

```python
from django.db import models

# Create your models here.

class Autor(models.Model):
    nombre = models.CharField(max_length=70)
```

2.2 Migraciones para convertir un modelo a Tabla

`makemigrations`	Le indica a Django que se han realizado cambios en alguno de sus modelos, y este creara un nuevo archivo de migración dentro de la carpeta migrations.	
`showmigrations`	Muestra una lista de las migraciones que se han hecho y cuales ya se han aplicado a la base de datos y cuales no.	
`sqlmigrate modulo NumMigracion` `Ejemp: sqlmigrate libreria 0002`	Solo muestra la sentencia sql que generara esa migración no la ejecuta en la base de datos.	
`migrate`	Aplica los cambios de las migraciones faltantes en la base de datos, aquí se convierte un modelo a una tabla.	

2.3 Ejemplo de un modelo con mas campos y tipos de datos.

```python
from django.db import models

class Libro(models.Model):

    isbn = models.CharField(max_length=13)
    titulo = models.CharField(max_length=70, blank=True)
    paginas = models.PositiveIntegerField()
    fecha_publicacion = models.DateField(null=True)
    imagen = models.URLField(max_length=85, null=True)
    desc_corta = models.CharField(max_length=2000)
    estatus= models.CharField(max_length=1)
    categoria = models.CharField(max_length=50)
```

SQL

```sql
CREATE TABLE "libreria_libro" (
"id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"isbn" varchar(13) NOT NULL,
"titulo" varchar(70) NOT NULL,
"paginas" integer NOT NULL,
"fecha_publicacion" date NOT NULL,
"imagen" varchar(85) NOT NULL,
"desc_corta" varchar(2000) NOT NULL,
"estatus" varchar(1) NOT NULL,
"categoria" varchar(50) NOT NULL
);
```

Al realizar la migración a sql django nos creara una llave primaria por defecto con el nombre de id de tipo entero y que se incrementa automáticamente.

2.4 Definir una llave primaria

Para establecer otra columna como la llave primaria se especifica de esta manera

```
isbn = models.CharField(max_length=13, primary_key=True)
```

SQL

```
"isbn" integer NOT NULL PRIMARY KEY AUTOINCREMENT
```

2.5 ¿Qué migraciones no se han aplicado en la base de datos?

¿Cómo sabe django que migraciones se han aplicado en la base de datos?

Además de aplicar los cambios en la base de datos, el comando mígrate también inserta un registro de esa migración en la tabla django_migrations, que es donde django revisa cuando usamos el comando `showmigrations`.

```
1  SELECT * FROM public.django_migrations
2  ORDER BY id ASC
```

Data Output | Explain | Messages | Notifications

	id [PK] integer	app character varying (255)	name character varying (255)	applied timestamp with time zone
14	14	auth	0009_alter_user_last_name_m...	2020-09-09 20:35:20.516501-05
15	15	auth	0010_alter_group_name_max...	2020-09-09 20:35:20.5335-05
16	16	auth	0011_update_proxy_permissi...	2020-09-09 20:35:20.551498-05
17	17	libreria	0001_initial	2020-09-09 20:35:20.671491-05

2.6 Algunos tipos de datos en Django y su equivalente a SQL

Django Field	***SQLite***	***PostgreSQL***
AutoField()	id integer PRIMARY KEY AUTOINCREMENT	id serial PRIMARY KEY
BigIntegerField()	bigint	
BinaryField()	BLOB	bytea
BooleanField()	bool	boolean
CharField(max_length=100)	varchar(100)	
DateField()	date	
DateTimeField()	datetime	timestamp with time zone
DecimalField(max_digits=5 decimal_places=2)	decimal	numeric(5 2)
DurationField()	bigint	interval
EmailField()	varchar(254)	
FileField()	varchar(100)	
FilePathField()	varchar(100)	
FloatField()	real	double precision
ImageField()	varchar(100)	
IntegerField()	integer	
GenericIPAddressField()	char(39)	inet
NullBooleanField()	bool NULL	boolean NULL
PositiveIntegerField()	integer unsigned **CHECK ("PositiveIntegerField" >= 0)**	integer **CHECK ("PositiveIntegerField" >= 0)**
PositiveSmallIntegerField()	smallint unsigned **CHECK ("PositiveSmallIntegerField" >= 0)**	smallint **CHECK ("PositiveSmallIntegerField" >= 0)**
SlugField()	varchar(50)	
SmallIntegerField()	smallint	
TextField()	text	
TimeField()	time	
URLField()	varchar(200)	
UUIDField()	char(32)	uuid
• **Todos se crean como NOT NUL a excepción de NullBooleanField**		

2.7 Validación de campos

2.7.1 A nivel del modelo

Para crear este tipo de validaciones primero creamos una función que es la que va a realizar la validación, en este ejemplo estamos validando que no se pueda insertar la palabra cobol, y en caso de que se inserte nos mandara un error.

```
from django.core.exceptions import ValidationError

def validar_titulo(titulo):
    if 'cobol' in titulo:
        raise ValidationError(f'{titulo} no se vende mucho')
    return titulo
```

Después indicamos que columna de nuestro modelo es la que va a tener la validación, en este caso la columna de titulo

```
titulo = models.CharField(max_length=70, blank=True,
validators=[validar_titulo,])
```

SQL

```
"titulo" varchar(70) NOT NULL,
```

Recuerda que este tipo de validación es solo a nivel del modelo, ya que como podemos ver no se crea ninguna validación a nivel de la base de datos.

Además para revisar este tipo de validaciones siempre debemos de ejecutar el método full_clean() para saber que errores de validación existen.

2.7.2 A nivel de la base de datos

Este tipo de validaciones se registran dentro de la clase Meta de nuestro modelo, en este ejemplo validamos que no se pueda insertar la palabra cobol en la columna de título.

```
class Libro(models.Model):
    ...

    class Meta:
        constraints =
[models.CheckConstraint(check=~models.Q(titulo='cobol')
, name='titulo_no_permitido_chk')]
```

SQL

```
CONSTRAINT "titulo_no_permitido_chk" CHECK (NOT
("titulo" = 'cobol')
```

Como se puede ver este tipo de validaciones crea una restricción a nivel de la base de datos.

2.8 Convertir una tabla a modelo

Si tenemos una tabla ya creada en nuestra base de datos y queremos transformarla a un modelo usamos el comando inspectdb.

Si tuviéramos una tabla llamada editorial dentro de nuestra base de datos para pasarla a modelo en un archivo llamado editorial.py:

```
python manage.py inspectdb editorial > editorial.py
```

```
CREATE TABLE editorial(
id INTEGER NOT NULL PRIMARY KEY
AUTOINCREMENT,
nombre varchar(100) NOT NULL
)
```

ORM

```
from django.db import models

class Editorial(models.Model):
    nombre = models.CharField(max_length=100)

    class Meta:
        managed = False
        db_table = 'editorial'
```

2.8.1 Generar una migración de un modelo creado con inspectdb

Si queremos generar una migración de nuestro modelo para que cuando otra persona baje nuestro código le genere la tabla, necesitamos cambiar el managed a True.

```
from django.db import models

class Editorial(models.Model):
    nombre = models.CharField(max_length=100)

    class Meta:
        managed = True
        db_table = 'editorial'
```

2.8.2 Hacer una migración falsa

Si quisiéramos que nuestra migración se registrara en nuestra base de datos pero que no se generara la tabla ya que si ya existe nos generaría error podemos hacer una migración falsa.

```
python manage migrate --fake
```

2.9 Cambiar el nombre de una tabla

Para cambiar el nombre de una tabla esto se realiza desde la clase meta en la opción db_table, y ejecutamos makemigration.

```python
from django.db import models

class Editorial(models.Model):
    nombre = models.CharField(max_length=100)

    class Meta:
       managed = True
        db_table = 'libreria_editorial'
```

SQL

```
ALTE TABLE "editorial" RENAME TO "libreria_editoria"
```

2.10 RELACIONES

2.10.1 ENTRE LA MISMA TABLA

Este tipo de relaciones se utilizan cuando una tabla necesita tener una relación consigo misma, por ejemplo un libro necesita tener una referencia a su edición anterior.

```python
class Libro(models.Model):
    isbn = models.CharField(max_length=13,
primary_key=True)
    ....
    edicion_anterior =
models.ForeignKey('self', null=True,
default=None, on_delete=models.PROTECT)
```

SQL

```
CREATE TABLE "libreria_libro" (
"isbn" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
..
..
"edicion_anterior_id" varchar(13)
NOT NULL
REFERENCES "libreria_libro" ("isbn")
DEFERRABLE INITIALLY DEFERRED;
```

2.10.2 UNO A UNO

Este tipo de relaciones permite que un solo registro de una tabla se relacione con un único registro de otra tabla,

la relación se realiza entre las llaves primarias de dos tablas, para especificar este tipo de relaciones en el modelo usamos **models.OneToOneField**.

```python
from django.db import models

from .libros import Libro


class LibroCronica(models.Model):
    descripcion_larga = models.TextField(null=True)
    libro = models.OneToOneField(Libro, on_delete=models.CASCADE, primary_key=True)
```

SQL

```sql
CREATE TABLE "libreria_librocronica"(
"descripcion_larga" text NULL,

"libro_id" varchar(13) NOT NULL PRIMARY KEY REFERENCES "libreria_libro"("isbn")
);
```

2.10.3 UNO A MUCHOS

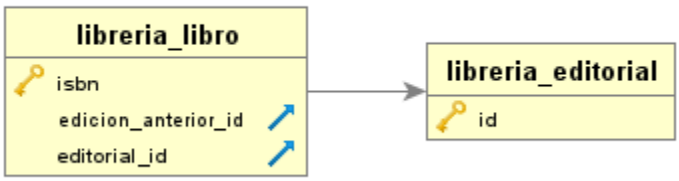
En este tipo de relaciones un registro de una tabla se asocia a uno o varios registros de otra tabla, por lo que una llave primaria se enlaza a la llave foránea de otra tabla.

Para especificar esta relación en el modelo usamos models.ForeignKey y especificamos el modelo con el que tendrá la relación.

```python
class Libro(models.Model):
    isbn = models.CharField(max_length=13, primary_key=True)
    ...
    editorial = models.ForeignKey(Editorial, on_delete=models.PROTECT)
```

SQL

```sql
CREATE TABLE "libreria_libro" (
"isbn" varchar(13) NOT NULL PRIMARY KEY,
...
"editorial_id" integer NOT NULL REFERENCES "libreria_editorial" ("id")
);
```



2.10.4 MUCHOS A MUCHOS

2.10.4.1 BASICA

Este tipo de relaciones uno o varios registros de una tabla se relacionan con uno o varios registros de otra tabla.

En realidad, la relación no es directa entre las dos tablas y se necesita de una tabla intermedia o también llamada tabla pivote que es la que contiene la información de las llaves primarias de las dos tablas que se están relacionando.

Para especificar estas relaciones en django usamos **ManyToManyField** en una de las dos tablas que queramos relacionar.

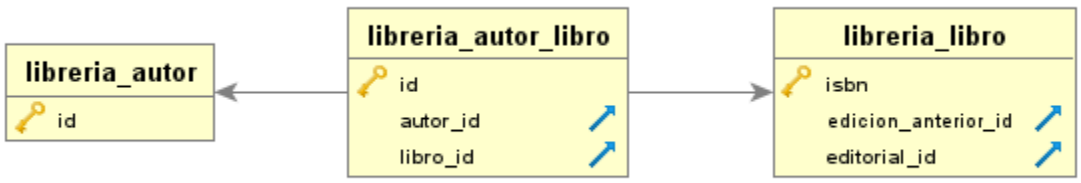
En este ejemplo estamos relacionando Autor y Libro la propiedad **ManyToManyField** la estamos especificando en el modelo Autor, pero se pudo haber especificado en Libro, es muy importante que también le pongamos un nombre a la relación usando **related_name** para poder hacer consultas de la relacion desde el modelo Libro.

```
from django.db import models
from .libros import Libro


class Autor(models.Model):
    ...
    libro = models.ManyToManyField(
Libro,
related_name='libros_autores')
```

SQL

```
CREATE TABLE "libreria_autor_libro" (
"id" integer NOT NULL PRIMARY KEY
AUTOINCREMENT,
"autor_id" integer NOT NULL REFERENCES
"libreria_autor" ("id"),
"libro_id" varchar(13) NOT NULL REFERENCES
"libreria_libro" ("isbn")
)
```

Como se puede notar **ManyToManyField** no modifica la tabla Autor sino que crea una tabla intermedia llamada libreria_autor_libro que es la que hace posible la relación muchos a muchos.



En este ejemplo si quisiéramos insertar libros a una Autor podemos hacerlo usando Add y se insertarían automáticamente los datos en la tabla librería_autor_libro
autor1.libro.add(libro1)
autor1.libro.add(libro2)

Para consultar los libros que ha escrito un autor
autor1.libro.all()

Para consultar los autores que escribieron un libro usamos el nombre de la relacion que es autores
libro1.autores.all()

2.10.4.2 EXTENDIDA

La forma básica de especificar una relación muchos a muchos nos es suficiente en la mayoría de casos, pero si necesitáramos extenderla es decir agregar mas datos a la relación, en el ejemplo anterior si quisiéramos especificar cuantos capítulos escribió un autor en el libro al que está relacionado necesitaríamos agregar un campo más a la tabla intermedia, por lo que tendremos que crear el modelo intermedio es decir que ya no lo genere automáticamente el ORM.

Los pasos esto son:

- Crear el modelo intermedio en este caso le llamaremos AutorCapitulo

```
from django.db import models


class AutorCapitulo(models.Model):
    autor = models.ForeignKey('Autor',
on_delete=models.SET_NULL, null=True)
    libro = models.ForeignKey('Libro',
on_delete=models.SET_NULL, null=True)
    numero_capitulos =
models.IntegerField(default=0)
```

SQL

```
CREATE TABLE “libreria_autorcapitulo”(
"id" integer NOT NULL PRIMARY KEY
AUTOINCREMENT,
"autor_id" integer NULL REFERENCES
"libreria_autor" ("id"),
"libro_id" varchar(13)  NULL REFERENCES
"libreria_libro" ("isbn")
"numero_capitulos" integer NOT NULL,
)
```

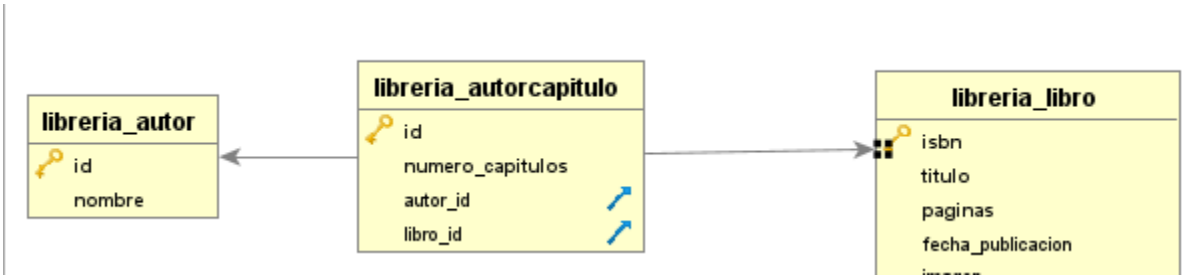
- Especificamos la relación dentro de alguno de nuestros dos modelos, en este caso usamos Autor, y lo hacemos por medio de tres propiedades

  Through: Indica el nombre del Modelo intermedio de la relación

  related_name: El nombre que le pondremos a la relación para que desde Libro también podamos acceder.

  through_fields: Las columnas que intervienen en la relación

```
class Autor(models.Model):
    nombre = models.CharField(max_length=70)
    libro = models.ManyToManyField(Libro, related_name='autores')
    libro = models.ManyToManyField(Libro,
            through='AutorCapitulo',
            related_name='libros_autores',
            through_fields=('autor','libro'))
```



2.11 REGLAS DE ELIMINACION EN UNA RELACION

Para especificar las reglas de eliminación en una relación se utiliza la propiedad ***on_delete*** del método ***ForeignKey*** y tiene las siguientes opciones:

CASCADE : Elimina las filas de referencia en la tabla secundaria, esta eliminación en cascada no es un ON DELETE CASCADE nativo de la base de datos, sino que crea las sentencias delete comenzando desde las tablas secundarias hasta las primarias, si nosotros intentáramos borrar un registro directamente desde la base de datos no lo permitiría porque tiene una restricción NO ACTION allí.

PROTECT: No permite eliminar una fila de la tabla primaria si esta tiene una relación.

SET_NULL: Pone null en las filas de referencia de la tabla secundaria, siempre y cuando esta columna permita nulos.

SET_DEFAULT: Pone el valor por defecto en las filas de referencia de la tabla secundaria.

SET() Pone un valor que nosotros especifiquemos en las filas de referencia de la tabla secundaria.

3 Lenguaje de manipulación de datos DML(Data Manipulation Language)

3.1 Probar nuestros comandos del ORM desde una consola

Podemos utilizar una consola para probar nuestros comandos del ORM e inclusive ver las sentencias SQL que generan

```
python manage.py shell_plus --print-sql
```

3.1.1 Recargar los cambios automáticamente en nuestra consola

Para que no tengamos que salirnos de nuestra consola (shell_plus) cada que realizamos un cambio a nuestros modelos podemos especificar que automáticamente tome los cambios que vamos realizando a nuestro código.

Dentro de nuestra consola especificamos esto:

```
%load_ext autoreload
%autoreload 2
```

3.2 Insertar datos desde el ORM

Inserción básica

```
Autor.objects.create(nombre=”Autor desde el ORM”)
```

SQL

```
INSERT INTO "libreria_autor" ("nombre")
VALUES ("Autor desde el ORM")
```

3.2.1

Inserción masiva

```
Autor.objects.bulk_create([
Autor(nombre="Autor 1M"),
Autor(nombre="Autor 2M"),
Autor(nombre="Autor 3M"),
])
```

SQL

```
INSERT INTO "libreria_autor"
("nombre")
SELECT "Autor 1M"
UNION ALL SELECT "Autor 2M"
UNION ALL SELECT "Autor 3M"
```

Insertar datos desde una migración

Si queremos cargar datos desde una migración realizaremos los siguiente:

1. Creamos un archivo nuevo, en el ejemplo lo llame migracion.sql y lo guarde dentro de una carpeta nueva llamada sql, este archivo contendrá las sentencias para insertar los datos y puede contener cualquier sentencia sql que necesitemos.

migracion.sql

```
-- Eliminamos los datos al inicio
delete from libreria_autor;
-- Ponemos el numero que se autoincrementa en 0
UPDATE SQLITE_SEQUENCE SET SEQ=0 WHERE NAME='libreria_autor';

-- Insertamos los datos

-- Autores
INSERT INTO libreria_autor VALUES (1, 'Jason R. Weiss');
INSERT INTO libreria_autor VALUES (2, 'Peter Small');
INSERT INTO libreria_autor VALUES (3, 'Spencer Salazar');
INSERT INTO libreria_autor VALUES (4, 'Ahmed Sidky');
INSERT INTO libreria_autor VALUES (5, 'Jonathan Anstey');
INSERT INTO libreria_autor VALUES (6, 'Leo S. Hsu');
INSERT INTO libreria_autor VALUES (7, 'Dmitry Babenko');
INSERT INTO libreria_autor VALUES (8, 'Brandon Goodin');
```

2. Creamos una migración vacía

   python manage.py makemigrations libreria –empty

```
from django.db import migrations

def cargar_datos_desde_sql():
    from orm_sqlfan.settings import BASE_DIR
    import os
    sql_script =
open(os.path.join(BASE_DIR,'libreria/sql/migracion.sql'),'r').read()
    return sql_script


class Migration(migrations.Migration):

    dependencies = [
        ('libreria', '0016_autor_libro'),
    ]

    operations = [
        migrations.RunSQL(cargar_datos_desde_sql(),)
    ]
```

3. Dentro del archivo de migración que nos genera es donde le especificamos que corra el archivo migración.sql
4. Aplicamos la migración
   python manage.py migrate

3.3 Consultas Básicas desde el ORM

3.3.1 Consultar todos los registros

```
Autor.objects.all()
```

SQL

```
SELECT "libreria_autor"."id",
       "libreria_autor"."nombre"
  FROM "libreria_autor"
```

3.3.2 Ver datos en lugar de ver el objeto

Cuando hacemos una consulta vemos en pantalla el nombre del objeto, si queremos ver los datos relacionados con uno o mas campos del modelo lo podemos especificar dentro del método especial __str__

```
from django.db import models

# Create your models here.

class Autor(models.Model):
    nombre = models.CharField(max_length=70)

     def __str__(self):
         return f'Yo soy {self.nombre}'
```

3.3.3 Consultar un único registro

Si conocemos la clave primaria del registro que estamos buscando podemos usar el método get.

```
Libro.objects.get(isbn='1935182080')
```

```
Libro.objects.get(pk='1935182080')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       "libreria_libro"."fecha_publicacion",
       "libreria_libro"."imagen",
       "libreria_libro"."desc_corta",
       "libreria_libro"."estatus",
       "libreria_libro"."categoria",
       "libreria_libro"."edicion_anterior_id",
       "libreria_libro"."editorial_id",
       "libreria_libro"."detalles",
       "libreria_libro"."dsc_corta_token"
  FROM "libreria_libro"
 WHERE "libreria_libro"."isbn" = '1935182080'
```

Recuerda que get solo funciona para traer un solo registro, si tu búsqueda no devuelve registros o devuelve mas de un registro entonces get te lanzara un error, es por eso que se usa mas para buscar por la llave primaria aunque podrías hacer la búsqueda por cualquier campo.

3.3.4 Uso del Manager para agregar nuestras propias consultas 

Si quisiéramos agregar nuestros propios métodos al modelo, para de esta forma poder usar las consultas que mas utilizamos tenemos que hacer uso de los Managers, los pasos para esto son:

1. Crear una clase que contendrá nuestros propios métodos de búsqueda, por ejemplo nuestro primer método es **buscar_por_isb,** que permite buscar por el isbn pero si no encuentra ningún libro lanzara una excepción que hemos personalizado

```
class LibroManager(models.Manager):
    def buscar_por_isbn(self, isbn):
        try:
            buscado = self.get(pk=isbn)
        except ObjectDoesNotExist:
            buscado = f'No existe el libro {isbn}'
        finally:
            return buscado
```

2. Registrar dentro de nuestro modelo el Manager, en este caso LibroManager, puedes registrar varios Manager dentro de un Modelo, en este caso solo registramos uno dentro de objects.

```
class Libro(models.Model):
    isbn = models.CharField(max_length=13, primary_key=True)
    .. .

    objects = LibroManager()
```

3. Ahora podemos usar nuestra función como parte del modelo

```
Libro.objects.buscar_por_isbn("1935182080")
```

3.3.5 Obtener solo el primer resultado

```
Libro.objects.all().first()
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
ORDER BY "libreria_libro"."isbn" ASC
LIMIT 1
```

3.3.6 Obtener solo el ultimo resultado

```
Libro.objects.all().last()
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
ORDER BY "libreria_libro"."isbn" DESC
LIMIT 1
```

3.3.7 Obtener los primeros N resultados

Ejemplo de obtener los primeros 5 libros

```
Libro.objects.all()[:5]
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
ORDER BY "libreria_libro"."isbn" DESC
LIMIT 5
```

3.3.8 Consultar coincidencias por el inicio

Ejempo de obtener los libros cuyo isbn comience con un 16

```
Libro.objects.filter(isbn__startswith="16")
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
 WHERE "libreria_libro"."isbn"::text LIKE '16%
'
```

3.3.9 Consultas por mayor que

Ejemplo de los libros que tienen mas de 200 paginas

```
Libro.objects.filter(paginas__gt=200)
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
 WHERE ("libreria_libro"."paginas" > 200
```

3.3.10 NOT IN

Ejemplo de libros que tienen mas de 200 paginas pero cuyo isbn no sea ninguno de estos dos ('1933988592','1884777600')

```
Libro.objects.filter(paginas__gt=200).exclude
(isbn__in=('1933988592','1884777600'))
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
 WHERE ("libreria_libro"."paginas" > 200
AND NOT ("libreria_libro"."isbn" IN ('193
3988592', '1884777600')))
```

3.3.11 Consultas por mayor o igual que

Ejemplo de libros que tienes 200 o mas paginas

```
Libro.objects.filter(paginas__gte=200)
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
 WHERE ("libreria_libro"."paginas" >= 200
```

3.3.12 Seleccionar las columnas a mostrar

Ejemplo de una consulta de los libros que tienen 200 o mas paginas, pero solo muestra las columnas isbn y paginas

```
Libro.objects.filter(paginas__gte=200)
.values('isbn','paginas')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."paginas"
  FROM "libreria_libro"
 WHERE "libreria_libro"."paginas" >= 200
```

3.3.13 Consultas por menor que

Ejemplo de los libros que tienen menos de 200 paginas

```
Libro.objects.filter(paginas__lt=200)
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
 WHERE ("libreria_libro"."paginas" < 200
```

3.3.14 Consultas por menor o igual que

Ejemplo de libros que tienes 200 o menos paginas

```
Libro.objects.filter(paginas__lte=200)
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas",
       ...
  FROM "libreria_libro"
 WHERE ("libreria_libro"."paginas" <= 200
```

3.3.15 Contar COUNT

Ejemplo de contar los libros que tienen menos de 200 paginas

```
Libro.objects.filter(paginas__lt=200).count()
```

SQL

```
SELECT COUNT(*) AS "__count"
  FROM "libreria_libro"
 WHERE "libreria_libro"."paginas" < 200
```

3.3.16 OR (forma larga)

Ejemplo de consulta de los libros con 200 paginas o con 300 paginas

```
consulta1 = Libro.objects.filter(paginas=200)

consulta2 =  Libro.objects.filter(paginas=300)
(consulta1 | consulta2).values('isbn','paginas')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."paginas"
  FROM "libreria_libro"
 WHERE (
      "libreria_libro"."paginas" = 200
      OR
      "libreria_libro"."paginas" = 300
      )
```

3.3.17 Consultar el por año de una fecha

Ejemplo de una consulta que muestra los libros cuya fecha de publicación es 2012

```
Libro.objects.filter(fecha_publicacion__year=2012
).values('isbn','fecha_publicacion')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."fecha_publicacion"
  FROM "libreria_libro"
 WHERE "libreria_libro"."fecha_publicacion"
 BETWEEN '2012-01-01'::date
     AND '2012-12-31'::date
```

3.3.18 Filtrar usando expresiones regulares

Consultar los libros cuyo isbn comience con un 19 seguido de 8 digitos

```
Libro.objects.filter(isbn__regex=r'19\d{8}$')
.values('isbn')
```

```
SELECT "libreria_libro"."isbn"
  FROM "libreria_libro"
 WHERE
"libreria_libro"."isbn"::text ~ '19\d{8}$'
```

3.3.19 UNION

Unir en una sola consulta el nombre los Autores que contengan la palabra hill con las Editoriales cuyo nombre contenga también la palabra hill.

```
a1 = Autor.objects.filter(nombre__contains='hill').values('nombre')

e1 = Editorial.objects.filter(nombre__contains='hill').values('nombre')

a1.union(e1)
```


```
 (
    SELECT "libreria_autor"."nombre"
    FROM "libreria_autor"
    WHERE "libreria_autor"."nombre"::text LIKE '%hill%'
 )
 UNION
(
    SELECT "libreria_editorial"."nombre"
    FROM "libreria_editorial"
    WHERE "libreria_editorial"."nombre"::text LIKE '%hill%'
```

3.3.20 El cuarto libro con mas paginas

Obtener el cuarto libro con mas paginas

```
Libro.objects.values('isbn','paginas').order_by('-paginas')[3]
```


```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."paginas"
  FROM "libreria_libro"
 ORDER BY "libreria_libro"."paginas" DESC
 LIMIT 1
OFFSET 3
```

OFFSET especifica cuantos registros debe de saltarse dentro del resultado, en este caso ordena por numero de paginas en forma descendente y se salta tres registro y LIMIT indica que solo traiga un registro, por lo que nos mostrara el cuarto libro con mas paginas.

3.3.21 El cuarto y quinto libro con mas paginas

Obtener el cuarto y quinto libro con mas paginas

```
Libro.objects.values('isbn','paginas')
.order_by('-paginas')[3:5]
```


```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."paginas"
  FROM "libreria_libro"
 ORDER BY "libreria_libro"."paginas" DESC
 LIMIT 2
OFFSET 3
```

3.3.22 Paginando a mano

No es una buena idea mostrar todos los registros en una consulta, es por eso que una solución es dividir los registros en paginas de un cierto numero de registros, una forma de hacer esto es crear una función como la siguiente en la que las paginas son 5 registros aunque la ultima puede tener 5 o menos registros.

```
    def LibroPorPaginas(self, pagina):
        import math

        total_filas = Libro.objects.count()
        # Cantidad de filas a mostrar por pagina
        filas_por_pagina = 5
        total_paginas = math.ceil(total_filas / filas_por_pagina)

        final = (pagina * filas_por_pagina)
        inicial = final - filas_por_pagina

        print(f'Pagina {pagina} / {total_paginas}')
        consulta = Libro.objects.all().order_by('isbn')[inicial:final]

        return consulta
```

Usando la función podemos seleccionar por ejemplo la pagina 3

```
Libro.objects.LibroPorPaginas(3)
```

SQL

```
Pagina 3 / 78
```

```
SELECT COUNT(*) AS "__count"
  FROM "libreria_libro"
```

```
SELECT "libreria_libro"."isbn",
       ...
  FROM "libreria_libro"
 ORDER BY "libreria_libro"."isbn" ASC
 LIMIT 5
OFFSET 10
```

Como podemos ver se realizaron dos consultas una para contar el total de libros y con eso podemos determinar la ultima pagina, y la segunda consulta utiliza OFFSET y LIMIT para mostrar los registros de esa pagina.

3.3.23 Paginando con ayuda de Django 

Django nos provee de una clase para paginar y no tener que hacer los cálculos a mano como lo hicimos en el ejemplo anterior, en la siguiente función usamos la clase Paginator de django.

```
    def LibroPorPaginasDjango(self,pagina):
        from django.core.paginator import Paginator
        # Recibe los registros a paginas y la cantidad de registros por pagina
        p = Paginator(Libro.objects.all().order_by('isbn'), 5 )
        # y podemos obtener el numero de paginas
        print(f'Pagina {pagina} / {p.num_pages}')
        # o los registros de una determinada pagina
        pag = p.page(pagina)
        return pag.object_list
```

Usando la función vamos a volver a seleccionar la pagina 3 , y veremos como obtenemos las mismas consultas sql que con la función que hicimos a mano, solo que en esta ocasión los cálculos los realizo Django por nosotros.

```
Libro.objects.LibroPorPaginasDjango(3)
```

SQL

```
Pagina 3 / 78
```

```
SELECT COUNT(*) AS "__count"
  FROM "libreria_libro"
```

```
SELECT "libreria_libro"."isbn",
       ...
  FROM "libreria_libro"
 ORDER BY "libreria_libro"."isbn" ASC
 LIMIT 5
OFFSET 10
```

3.3.24 Índices 

Conforme nuestra base de datos va creciendo, nuestras consultas pueden comenzar a ser mas lentas, los índices nos pueden ayudar a acelerar las consultas pero debemos tener cuidado sobre que columnas los creamos ya que un índice se parece mucho al índice de un libro pero en este caso debe de estarse actualizando cada que insertamos, modificamos o eliminando un registro en el que se encuentre el índice, por lo que debes de escoger bien que columnas son candidatas a índices, generalmente son las columnas por las que mas estamos filtrando nuestra información.

Siempre puede apoyarte viendo el plan de ejecución de una consulta y lo mejor de todo es que esto también lo puedes hacer desde el ORM de Django

```
Libro.objects.filter(paginas__gte=200).explain
```

y en resultado del pland e ejecución podemos ver que esta buscando en toda la tabla cuando filtramos por página **“SCAN TABLE librería_libro”**

```
EXPLAIN SELECT "libreria_libro"."isbn",
  ...
  FROM "libreria_libro"
 WHERE "libreria_libro"."paginas" >= 200

Execution time: 0.003965s [Database: default]

Out[2]: '0 0 0 SCAN TABLE libreria_libro'
```

Para crear un índice usando el ORM debemos ir a nuestro modelo y sobre la columna que se aplicara índice ponemos la propiedad **db_index=True** y realizamos nuestras migraciones para ver los cambios en la base de datos.

En este ejemplo pusimos un índice sobre la columna paginas del modelo Libro y aplicamos nuestras migraciones.

```
class Libro(models.Model):
    isbn = models.CharField(max_length=13, primary_key=True)
    ...
    paginas = models.PositiveIntegerField(db_index=True,)
```

Al migrar los datos la parte del índice en SQL se vería asi:

```
CREATE INDEX "libreria_libro_paginas_f8982400" ON "libreria_libro" ("paginas");
```

Ahora cuando filtremos por página nuestra base de datos podrá usar el índice para realizar la búsqueda de forma mas rápida.

3.4 Consultas de Agregación y Agrupado

3.4.1 MIN

Calcular cual es el numero minimo de paginas que puede tener un libro, en esta consulta no tomamos en cuenta los libros que no se especifico su numero de paginas por lo cual es 0 y este seria el mínimo numero de paginas.

```
Libro.objects.filter(paginas__gt=0)
.aggregate(Min('paginas')
```

SQL

```
SELECT MIN("libreria_libro"."paginas") AS
"paginas__min"
  FROM "libreria_libro"
 WHERE "libreria_libro"."paginas" > 0
```

```
Resultado: {'paginas__min': 180}
```

3.4.2 MAX

Calcular cual es el numero máximo de paginas que puede tener un libro, aquí no necesitamos filtrar, este seria el máximo numero de paginas.

```
Libro.objects.aggregate(Max('paginas')
```

SQL

```
SELECT MAX("libreria_libro"."paginas") AS
"paginas__max"
  FROM "libreria_libro"
```

```
Resultado: {'paginas__max': 1101}
```

3.4.3 AVG

Calcular cual es el numero medio de paginas que puede tener un libro, en esta consulta no tomamos en cuenta los libros que no se especifico su numero de paginas para así solo considerar los libros con paginas.

```
Libro.objects.filter(paginas__gt=0)
.aggregate(Avg('paginas')
```

SQL

```
SELECT AVG("libreria_libro"."paginas") AS
"paginas__avg"
  FROM "libreria_libro"
 WHERE "libreria_libro"."paginas" > 0
```

```
Resultado: {'paginas__avg': 464.08
```

3.4.4 SUM

Sumar el total de paginas de todos los libros que tenemos de Python

```
Libro.objects.filter(categoria__icontains='python').aggregate(Sum('paginas'))
```


```
SELECT SUM("libreria_libro"."paginas") AS "paginas__sum"
  FROM "libreria_libro"
 WHERE UPPER("libreria_libro"."categoria"::text) LIKE UPPER('%python%')
```

```
Resultado: {'paginas__sum': 2894
```

3.4.5 GROUP BY

Para poder agrupar nuestros datos usamos el método **annotate** y dentro de el podemos contar o usar cualquiera de los métodos de agregación que vimos antes, a continuación te muestro algunos ejemplo de agrupado.

1. Agrupar los libros que son de Python por categoría y contar cuantos libros de cada categoría hay.

```
Libro.objects.filter(categoria__contains='python').values('categoria').annotate(NumeroLibros=Count('*'))
```


```
SELECT "libreria_libro"."categoria",
       COUNT(*) AS "NumeroLibros"
  FROM "libreria_libro"
 WHERE UPPER("libreria_libro"."categoria"::text) LIKE UPPER('%python%')
 GROUP BY "libreria_libro"."categoria"
```

	categoria character varying (50)	NumeroLibros bigint
1	["Programming", "Python"]	1
2	["Python"]	5

2. Agrupar los libros que son de Python por categoría y por el nombre de la editorial y contar cuantos libros hay, a diferencia del ejemplo anterior en este ejemplo se involucran dos modelos Libro y Editorial, por lo que estamos hablando también de una union.

```
Libro.objects.filter(categoria__icontains='python').values('categoria'
,'editorial__nombre').annotate(NumeroLibros=Count('*'))
```

SQL

```
SELECT "libreria_libro"."categoria",
       "libreria_editorial"."nombre",
       COUNT(*) AS "NumeroLibros"
FROM "libreria_libro"
INNER JOIN "libreria_editorial"
  ON ("libreria_libro"."editorial_id" = "libreria_editorial"."id")
WHERE UPPER("libreria_libro"."categoria"::text) LIKE UPPER('%python%')
GROUP BY "libreria_libro"."categoria",
         "libreria_editorial"."nombre"
```

	categoria character varying (50)	nombre character varying (100)	NumeroLibros bigint
1	["Programming", "Python"]	PirÃ¡mide	1
2	["Python"]	PirÃ¡mide	2
3	["Python"]	Siglo XXI	2
4	["Python"]	Tirant lo Blanch	1

3.4.6 HAVING (Filtrar agrupados)

Para poder filtrar lo que agrupamos utilizamos **filter** solo que ahora utilizaremos alguna de las columnas que especificamos dentro de annotate, en este ejemplo agrupamos los libros por fecha_publicacion y filtramos solo las tengan mas de 5 libros publicados en esa fecha.

```
Libro.objects.values('fecha_publicacion').annotate(cant_fec_pub=Count('fecha_publicacion')
).filter(cant_fec_pub__gte=5)
```

SQL

```
SELECT "libreria_libro"."fecha_publicacion",
       COUNT("libreria_libro"."fecha_publicacion") AS "cant_fec_pub"
FROM "libreria_libro"
GROUP BY "libreria_libro"."fecha_publicacion"
HAVING COUNT("libreria_libro"."fecha_publicacion") >= 5
```

	fecha_publicacion date	cant_fec_pub bigint
1	2002-03-01	5
2	2005-03-01	12

Si quisiéramos obtener el detalle de los libros de la consulta anterior podemos hacer lo siguiente

```
Consulta_fechas =Libro.objects.values('fecha_publicacion').annotate(cant_fec_pub=Count('
fecha_publicacion')).filter(cant_fec_pub__gte=5).values_list('fecha_publicacion')

Libro.objects.filter(fecha_publicacion__in= consulta1).values('isbn')
```

SQL

```
SELECT "libreria_libro"."isbn"
  FROM "libreria_libro"
 WHERE "libreria_libro"."fecha_publicacion" IN (
       SELECT U0."fecha_publicacion"
         FROM "libreria_libro" U0
        GROUP BY U0."fecha_publicacion"
       HAVING COUNT(U0."fecha_publicacion") >= 5
      )
```

	isbn [PK] character varying (13)
1	1932394524d-e
2	1932394524e-e
3	1932394524f-e
4	1932394524g-e

3.4.7 DISTINCT

Devolver valores únicos de una para evitar ver valores duplicados, en este ejemplo usamos distinct sobre paginas ya que muchos libros tienen 0 paginas.

```
Libro.objects.values('paginas').filter(
paginas__lt=200).distinct()
```

SQL

```
SELECT DISTINCT "libreria_libro"."paginas"
  FROM "libreria_libro"
 WHERE "libreria_libro"."paginas" < 200
```

	paginas integer
1	0
2	180

3.5 Funciones escalares

Al igual que cualquier lenguaje de programación SQL también dispone de muchas funciones de tipo aritméticas, de cadena de caracteres, conversión, manejo de fechas etc. A diferencia de las funciones de agregación que nos devuelven el calculo de los valores de una columna, las funciones escalares nos devuelven un valor en base al valor de entrada que le proporcionemos.

El ORM de Django nos permite utilizar estas funciones, aquí podemos ver las que están disponibles para nosotros dentro de la librería **django.db.models.functions**

```
In [3]: from django.db.models.functions import
    Abs             Concat          ExtractDay      FirstValue      Log             NullIf          Right           text            TruncTime
    ACos            ConcatPair      ExtractHour     Floor           Lower           Ord             Round           Trim            TruncWeek
    ASin            Cos             ExtractIsoYear  Greatest        LPad            PercentRank     RowNumber       Trunc           TruncYear
    ATan            Cot             ExtractMinute   Lag             LTrim           Pi              RPad            TruncDate       Upper
    ATan2           CumeDist        ExtractMonth    LastValue       math            Power           RTrim           TruncDay        window
    Cast            datetime        ExtractQuarter  Lead            mixins          Radians         Sin             TruncHour       __all__
    Ceil            Degrees         ExtractSecond   Least           Mod             Rank            Sqrt            TruncMinute     __doc__
    Chr             DenseRank       ExtractWeek     Left            Now             Repeat          StrIndex        TruncMonth      __file__
    Coalesce        Exp             ExtractWeekDay  Length          NthValue        Replace         Substr          TruncQuarter    __name__
    comparison      Extract         ExtractYear     Ln              Ntile           Reverse         Tan             TruncSecond     __package__
```

En esta guía solo veremos el uso de algunas de estas funciones pero si necesitas un detalle de todas estas puedes ir a la documentación oficial de Django https://docs.djangoproject.com/en/3.1/ref/models/database-functions/

3.5.1 Left

Queremos que nuestra desc_corta solo muestre los primeros 15 caracteres

```
from django.db.models.functions import Left

Libro.objects.annotate(desc_resumida=
Left('desc_corta',15)).values('isbn',
'desc_resumida')
```

SQL

```

SELECT "libreria_libro"."isbn",
   LEFT("libreria_libro"."desc_corta", 15)
 AS "desc_resumida"
  FROM "libreria_libro"
```

3.5.2 Concat y Value

Queremos concatenar tres puntos (…) al fina de los valores de nuestra desc_resumida, para poder concatenar usamos la función **Concat** y para especificar la cadena a concatenar usamos **Value.**

```
from django.db.models import  Value as V
from django.db.models.functions import Left
, Concat

Libro.objects.annotate(desc_resumida=
Concat(Left('desc_corta',15),V('...')
)).values('isbn','desc_resumida')
```

SQL

```

SELECT "libreria_libro"."isbn",
       CONCAT(LEFT("libreria_libro"."desc_corta"
, 15), '...') AS "desc_resumida"
  FROM "libreria_libro"
```

	isbn [PK] character varying (13)	desc_resumida text
1	1933988592	Sin resumen...
2	1884777791	Sin resumen...
3	193239415X	"2005 Best Java...
4	1884777589	"Distinctive bo...
5	1933988495	Hello World! pr...

3.5.3 Case 

En el ejemplo anterior le pusimos los tres puntos (…) a todas las filas, en este ejemplo solo se los pondremos a las cadenas cuya longitud sea mayor de 15.

```
from django.db.models import CharField, Case, Value as V
from django.db.models.functions import Left, Concat
```

```
Libro.objects.annotate(longitud = Length('desc_corta')).annotate(
             desc_resumida = Case(
                 When(longitud__gt=50,
                    then = Concat(Left('desc_corta',15), V('...')))
                    ,
                    default=('desc_corta'),
                    output_field=CharField(),
                 )).values('isbn','desc_resu mida','longitud')
```

SQL 

```
SELECT "libreria_libro"."isbn",
       LENGTH("libreria_libro"."desc_corta") AS "longitud",
       CASE WHEN LENGTH("libreria_libro"."desc_corta") > 15 THEN CONCAT(LEFT("libreria_
libro"."desc_corta", 15), '...')
            ELSE "libreria_libro"."desc_corta"
             END AS "desc_resumida"
  FROM "libreria_libro"
```

	isbn [PK] character varying (13)	longitud integer	desc_resumida character varying
1	1933988592	11	Sin resumen
2	1884777791	11	Sin resumen
3	193239415X	50	"2005 Best Java...
4	1884777589	92	"Distinctive bo...
5	1933988495	94	Hello World! pr...

3.5.4 Comparar columnas del mismo modelo (F)

Queremos saber que libros tienen un titulo igual a su desc_corta dentro de sus primeros 50 caracteres, para especificar que lo que va a comparar es una columna del modelo y no una cadena de texto usamos F(nombre_columna)

```
from django.db.models import  F
```

```
from django.db.models.functions import  Left

Libro.objects.annotate(tit50= Left('titulo',50), desc50= Left('desc_corta',50))
.filter(tit50 = F('desc50')).values('isbn','tit50','desc50')
```

SQL

```
SELECT "libreria_libro"."isbn",
```

```
       LEFT("libreria_libro"."titulo", 50) AS "tit50",
       LEFT("libreria_libro"."desc_corta", 50) AS "desc50"
  FROM "libreria_libro"
 WHERE LEFT("libreria_libro"."titulo", 50) = (LEFT("libreria_libro"."desc_corta", 50))
```

3.5.5 Replace

Queremos quitarle las comillas a los nombres de nuestra categoría y remplazarlas por un *

```
from django.db.models.functions import  Replace
from django.db.models import Value as V

Libro.objects.annotate(categoria_sin_comillas = Replace('categoria', V('"'),V('*'))).
values('isbn','categoria','categoria_sin_comillas')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       REPLACE("libreria_libro"."categoria", '"', '*') AS "categoria_sin_comillas"
  FROM "libreria_libro"
```

	isbn [PK] character varying (13)	categoria character varying (50)	categoria_sin_comillas text
1	1617291005	[]	[]
2	1884777848	["Java"]	[*Java*]
3	193011009X	["Java", "Internet"]	[*Java*, *Internet*]

Recuerda que no estamos afectando la base de datos solo estamos mostrando los datos, si quisieras afectar la base de datos y cambiar las comillas por * tendrías que hacer lo siguiente:

```
Libro.objects.filter(categoria='[]').update(categoria = Replace('categoria', V('"'),V('*')))
```

SQL

```
UPDATE "libreria_libro"
   SET "categoria" = REPLACE("libreria_libro"."categoria", '"', '*')
 WHERE "libreria_libro"."categoria" = '[]'
```

3.6 Consultas avanzadas

3.6.1 OR mejorado (usando Q)

Ya habíamos visto una forma larga de realizar un OR (forma larga) vamos a ver una forma mas corta de poder utilizarlo.

Para poder hacer esto utilizamos el operador Q(consulta) después podemos aplicar el OR y seguir haciendo condiciones.

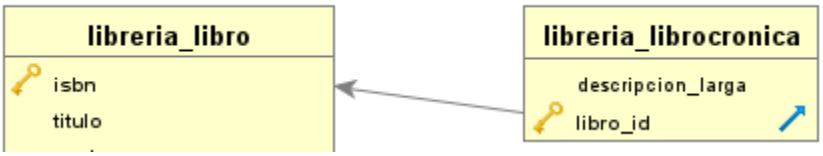
En este ejemplo queremos las categorías que sean sobre Python o Java o net y que no tengan paginas igual a 0

```
from django.db.models import Q

Libro.objects.filter(
(Q(categoria__contains='python') |
Q(categoria__contains='java') |
Q(categoria__contains='net')) &
~Q(paginas=0))
```


```
 SELECT "libreria_libro"."isbn",
        ...
  FROM "libreria_libro"
 WHERE ((
"libreria_libro"."categoria"::text LIKE '%python%' OR
 "libreria_libro"."categoria"::text LIKE '%java%' OR
 "libreria_libro"."categoria"::text LIKE '%net%')
AND NOT  ("libreria_libro"."paginas" = 0))
```

3.6.2 LEFT JOIN (en relaciones 1 a 1)

Queremos hacer una consulta entre el modelo Libro y LibroCronica, nos interesa saber que libros no tienen crónica, recordemos como esta nuestra relación 1 a 1.



La relación la podemos hacer directamente desde nuestro Libro usando el nombre de la tabla librocronica, nuestra consulta quedaría de esta manera:

```
Libro.objects.filter(librocronica__descripcion_larga__isnull=True)
.values('isbn','titulo','librocronica__descripcion_larga')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_librocronica"."descripcion_larga"
  FROM "libreria_libro"
  LEFT OUTER JOIN "libreria_librocronica"
    ON ("libreria_libro"."isbn" = "libreria_librocronica"."libro_id")
 WHERE "libreria_librocronica"."descripcion_larga" IS NULL
```

3.6.3 LEFT JOIN (usando select_related relaciones 1 a 1)

Existe otra forma de hacer la relacion uno a uno de Libro y LibroCronica y es usando select_related(modelo_relacion).

```
Libro.objects.select_related('librocronica').filter(categoria__contains='python')
```

SQL

```
SELECT "libreria_libro"."isbn",
       ...
  FROM "libreria_libro"
  LEFT OUTER JOIN "libreria_librocronica"
    ON ("libreria_libro"."isbn" = "libreria_librocronica"."libro_id")
 WHERE "libreria_libro"."categoria"::text LIKE '%python%'
```

3.6.4 Beneficios de usar select_related en relaciones 1 a 1

En el ejemplo anterior hicimos las consultas desde el modelo Libro, que pasaría si lo hiciéramos ahora desde el modelo, vamos a consultar primero nuestro modelo LibroCronica.

```
LibroCronica.objects.all()[:3]
```


```
SELECT "libreria_librocronica"."descripcion_larga",
       "libreria_librocronica"."libro_id"
  FROM "libreria_librocronica"
 LIMIT 3
```

```
SELECT "libreria_libro"."isbn",
       ...
  FROM "libreria_libro"
 WHERE "libreria_libro"."isbn" = '1933988673'
```

```
SELECT "libreria_libro"."isbn",
       ...
       "libreria_libro"."dsc_corta_token"
  FROM "libreria_libro"
 WHERE "libreria_libro"."isbn" = '1935182722'
```

```
SELECT "libreria_libro"."isbn",
       ...
  FROM "libreria_libro"
 WHERE "libreria_libro"."isbn" = '1933988746'
```

Como puedes ver cuando hacemos la consulta sobre el que se le especifico la relación UNO A UNO, el ORM realiza la consulta primero sobre este modelo para obtener los ids en este caso los isbn después realiza una consulta por cada isbn sobre el modelo al que esta relacionado, en este caso solo fueron 3 registros, como podemos ver este tipo de consultas no es optima si aumentara el numero de registros que estamos consultando

Vamos a ver que pasa si utilizamos select_related

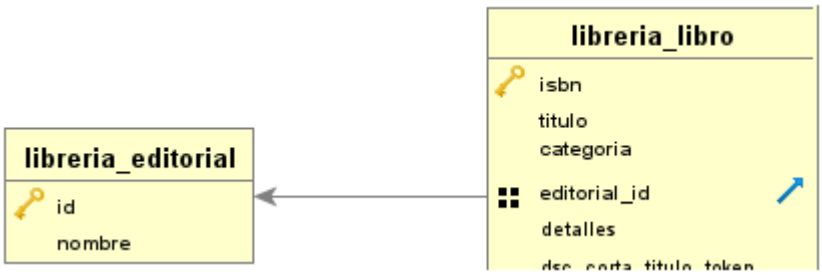
```
LibroCronica.objects.select_related('libro').all()[:3]
```


```
SELECT "libreria_librocronica"."descripcion_larga",
       "libreria_librocronica"."libro_id",
       "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       ...
  FROM "libreria_librocronica"
 INNER JOIN "libreria_libro"
    ON ("libreria_librocronica"."libro_id" = "libreria_libro"."isbn")
 LIMIT 3
```

Mucho mejor no!!!

3.6.5 Consultas de relaciones 1 a muchos

Para poder hacer los ejemplo de consultas en relaciones uno a muchos vamos a usar la relación entre Libro y Editorial UNO A MUCHOS, recordemos como esta nuestra relación.



Queremos consultar los libros cuya categoría sea Python y entonces imprimir el nombre de su editorial el cual se encuentra en el modelo Editoria, así que una primera forma de hacerlo seria la siguiente:

```python
categorias = Libro.objects.all().filter(
categoria__icontains='python')

for libro in categorias:
    print(libro.editorial.nombre)
```

SQL

Una consulta por cada editorial, en total 6 no lo se!!

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."categoria",
        ...
FROM "libreria_libro"
 WHERE UPPER("libreria_libro"."categoria"::text) LIKE UPPER('%pyt
hon%')
```

```
SELECT "libreria_editorial"."id","libreria_editorial"."nombre"
  FROM "libreria_editorial" WHERE "libreria_editorial"."id" = 8

SELECT "libreria_editorial"."id"","libreria_editorial"."nombre"
  FROM "libreria_editorial" WHERE "libreria_editorial"."id" = 6

SELECT "libreria_editorial"."id","libreria_editorial"."nombre"
  FROM "libreria_editorial" WHERE "libreria_editorial"."id" = 7

SELECT "libreria_editorial"."id","libreria_editorial"."nombre"
  FROM "libreria_editorial" WHERE "libreria_editorial"."id" = 8

SELECT "libreria_editorial"."id","libreria_editorial"."nombre"
  FROM "libreria_editorial" WHERE "libreria_editorial"."id" = 6

SELECT "libreria_editorial"."id","libreria_editorial"."nombre"
  FROM "libreria_editorial" WHERE "libreria_editorial"."id" = 8
```

3.6.6 Beneficios de usar select_related en relaciones 1 a muchos

Vamos a realizar la consulta anterior pero ahora agregándole select_related especificándole que la relación es sobre el modelo editorial

```
categorias = Libro.objects.all().
select_related('editorial').filte
r(categoria__icontains='python')
```

```
for libro in categorias:
    print(libro.editorial.nombre)
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."categoria",
        ...
       "libreria_editorial"."id",
       "libreria_editorial"."nombre"
  FROM "libreria_libro"
 INNER JOIN "libreria_editorial"
    ON ("libreria_libro"."editorial_id" = "libre
ria_editorial"."id")
 WHERE UPPER("libreria_libro"."categoria"::text)
 LIKE UPPER('%python%')
```

No se tu pero yo prefiero mis consultas en una sola sentencia SQL.

¿Y si no quiero utilizar un for para esto?

Pues:

```
consulta = Libro.objects.all().select_related('editorial').filter(categoria__icontains
='python')
```

```
dic_libros = dict(consulta.values_list('isbn','editorial__nombre'))
print(dic_libros)
```

La consulta sql que se genera es la misma que la anterior

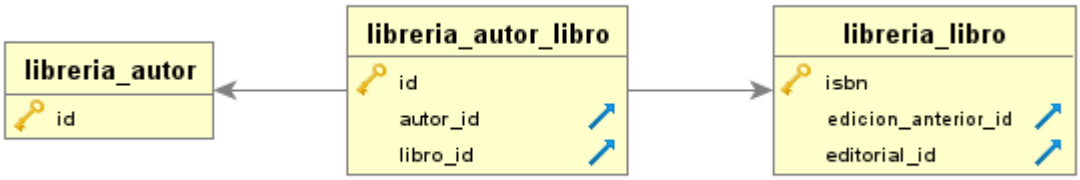
3.7 Consultas de relación muchos a muchos

Para realizar las consultas en una relación muchos a muchos es importante saber cual es el modelo que tiene especificada la relación, en los ejemplo que veremos la relación esta en el modelo Autor MUCHOS A MUCHOS.

La Consulta en la relación BASICA de nuestro ejemplo se especifico en el modelo Autor

```
class Autor(models.Model):
    nombre = models.CharField(max_length=70
)
    libro = models.ManyToManyField(Libro,
 related_name='autores')
```

```
class Libro(models.Model):
    isbn = models.CharField(max_length=13,
primary_key=True)
```



Si quisiéramos consultar los Autores y listar los libros que cada uno ha escrito tenemos las siguientes formas de hacerlo

3.7.1 Forma inviable 

En este ejemplo vamos a consultar 2 Autores y después vamos a buscar todos los libros que han escrito, como veremos esta forma no es optima debido a que primero realiza una consulta para traer los datos de los dos autores y a continuación realiza una consulta por cada autor dentro de la tabla de libros.

```
autores = Autor.objects.filter(pk__in
=(398,523))
```

```
for autor in autores:
    print(f'Autor: {autor}')
    print('Libros escritos:')
    for libro in autor.libro.all():
        print(libro.titulo)
```

SQL

```
SELECT "libreria_autor"."id",
       "libreria_autor"."nombre"
  FROM "libreria_autor"
 WHERE "libreria_autor"."id" IN (398, 523)
```

```
Autor: Yo soy Vikram Goyal
Libros escritos:
SELECT "libreria_libro"."isbn",
         ...
FROM "libreria_libro"
 INNER JOIN "libreria_autorcapitulo"
    ON ("libreria_libro"."isbn" = "libreria_auto
rcapitulo"."libro_id")
 WHERE "libreria_autorcapitulo"."autor_id" = 398

Autor: Yo soy Don Jones
Libros escritos:
SELECT "libreria_libro"."isbn",
         ...
  FROM "libreria_libro"
 INNER JOIN "libreria_autorcapitulo"
    ON ("libreria_libro"."isbn" = "libreria_auto
rcapitulo"."libro_id")
 WHERE "libreria_autorcapitulo"."autor_id" = 523
```

3.7.2 Forma optima usando prefetch_related

Vamos a realizar la misma consulta anterior pero ahora usaremos el método prefetch_related para especificarle que Autor tiene una relación con libros, y como veremos ahora solo realizara 2 consultas no importa el numero de autores que consultemos.

```
autores = Autor.objects.filter(pk__in=
(398,523)).prefetch_related('libro')
```

```
for autor in autores:
    print(f'Autor: {autor}')
    print('Libros escritos:')
    for libro in autor.libro.all():
        print(libro.titulo)
```

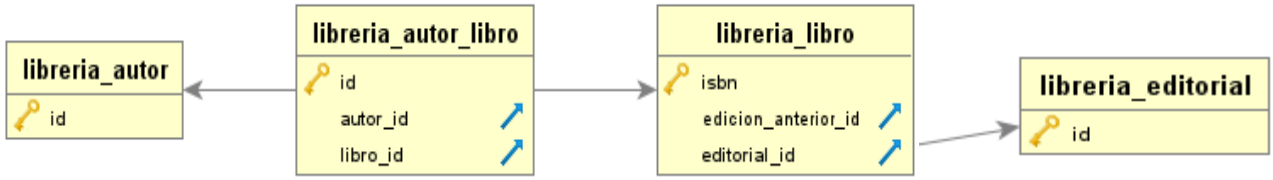
SQL

```
SELECT "libreria_autor"."id",
       "libreria_autor"."nombre"
  FROM "libreria_autor"
 WHERE "libreria_autor"."id" IN (398, 523)
```

```
SELECT "libreria_libro"."isbn",
       ...
  FROM "libreria_libro"
 INNER JOIN "libreria_autorcapitulo"
    ON ("libreria_libro"."isbn" = "libreria_a
utorcapitulo"."libro_id")
 WHERE "libreria_autorcapitulo"."autor_id" IN
 (398, 523)
```

3.7.3 Consultas muchos a muchos profundas (forma inviable)

Si quisiéramos hacer una consulta más profunda por ejemplo buscar el Autor y sus libros asi como el nombre de la editorial de cada libro, aquí tenemos una relación muchos a muchos entre autor y libro y una relación uno a muchos entre editorial y libro



Nuestro primer intento será consultar los Autores utilizando prefetch_related como lo vimos en el ejemplo anterior, así podemos listar cada uno de sus libros, pero ahora debemos mostrar el nombre de la editorial por cada uno de los libros

```
autores = Autor.objects.filter(pk__in=(398
,523)).prefetch_related('libro')
```

```
for autor in autores:
    print(f'Autor: {autor}')
    print('Libros escritos:')
    for libro in autor.libro.all():
        print(f'{libro.isbn} Editorial: {
libro.editorial.nombre}')
```

SQL

```
SELECT "libreria_autor"."id",
       "libreria_autor"."nombre"
  FROM "libreria_autor"
 WHERE "libreria_autor"."id" IN (398, 523)
```

```
SELECT "libreria_libro"."isbn",
       ...
  FROM "libreria_libro"
 INNER JOIN "libreria_autorcapitulo"
    ON ("libreria_libro"."isbn" = "libreria_a
utorcapitulo"."libro_id")
 WHERE "libreria_autorcapitulo"."autor_id" IN
 (398, 523)
```

```
SELECT "libreria_editorial"."id",
       "libreria_editorial"."nombre"
  FROM "libreria_editorial"
 WHERE "libreria_editorial"."id" = 6
1932394524d-e Editorial: Siglo XXI
SELECT "libreria_editorial"."id",
       "libreria_editorial"."nombre"
  FROM "libreria_editorial"
 WHERE "libreria_editorial"."id" =
1932394524e-e Editorial: Siglo XXI
SELECT "libreria_editorial"."id",
       "libreria_editorial"."nombre"
  FROM "libreria_editorial"
 WHERE "libreria_editorial"."id" = 3
1932394524f-e Editorial: Tecnos

Y aun hay mas en total son 17 consultas!!
```

Ahora el problema es que por cada libro esta realizando una consulta a la tabla de editorial, en total son 17 consultas a esta tabla, no se ha ustedes pero a mi me parecen muchas consultas.

3.7.4 Consultas muchos a muchos profundas (forma viable)

Vamos a realizar la misma consulta que en el ejemplo anterior solo que esta vez en el prefetch_related le indicaremos que también existe una relación entre el libro y la editorial.

```
autores = Autor.objects.filter(pk__in=(398
,523)).prefetch_related(libro__editorial)
```

```
for autor in autores:
    print(f'Autor: {autor}')
    print('Libros escritos:')
    for libro in autor.libro.all():
         print(f'{libro.isbn} Editorial: {
libro.editorial.nombre}')
```

SQL

Tres consultas una por cada tabla Libro,Autor,Editorial, mucho mejor, aunque estoy seguro que cualquiera con un poco de conocimientos en sql pudiera hacer una mejor consulta que esta.

```
SELECT "libreria_autor"."id",
       "libreria_autor"."nombre"
  FROM "libreria_autor"
 WHERE "libreria_autor"."id" IN (398, 523)
```

```
SELECT "libreria_libro"."isbn",
       ...
  FROM "libreria_libro"
 INNER JOIN "libreria_autorcapitulo"
    ON ("libreria_libro"."isbn" = "libreria_a
utorcapitulo"."libro_id")
 WHERE "libreria_autorcapitulo"."autor_id" IN
 (398, 523)
```

```
SELECT "libreria_editorial"."id",
       "libreria_editorial"."nombre"
  FROM "libreria_editorial"
 WHERE "libreria_editorial"."id" IN
(1, 2, 3, 5, 6, 7, 8, 9, 10)
```

3.7.5 Prefetch al rescate 

Existe una clase llamada Prefetch que podemos usar para controlar un poco mas las operaciones que va a realizar prefetch_related, en esta consulta utilizamos el método select_related que ya habíamos visto en las consultas uno a muchos y esta consulta se la especificamos dentro del Prefectch en su propiedad queryset.

```
from django.db.models import Prefetch

libro_y_editorial = Libro.objects.select_related('editorial')

autores = Autor.objects.filter(pk__in=(398,523)).prefetch_related(
Prefetch('libro', queryset=libro_y_editorial))
   for autor in autores:
       print(f'Autor: {autor}')
       print('Libros escritos:')
       for libro in autor.libro.all():
       print(f'{libro.isbn} Editorial
```

Solo dos consultas, de esto es de lo que hablábamos, mucho mejor ¿no es así?!!

SQL

```
SELECT "libreria_autor"."id",
       "libreria_autor"."nombre"
  FROM "libreria_autor"
 WHERE "libreria_autor"."id" IN (398, 523)

SELECT ("libreria_autorcapitulo"."autor_id") AS "_prefetch_related_val_autor_id",
       "libreria_libro"."isbn",
       ..
       "libreria_editorial"."id",
       "libreria_editorial"."nombre"
  FROM "libreria_libro"
 INNER JOIN "libreria_autorcapitulo"
    ON ("libreria_libro"."isbn" = "libreria_autorcapitulo"."libro_id")
 INNER JOIN "libreria_editorial"
    ON ("libreria_libro"."editorial_id" = "libreria_editorial"."id")
 WHERE "libreria_autorcapitulo"."autor_id" IN (398, 523)
```

3.7.6 Usando los atributos de Prefetch 

Prefetch nos permite darle un nombre de atributo usando to_attr con lo cual en la consulta anterior podríamos usar este nombre, en este ejemplo pueden ver que no necesitamos llamar autor.libro.all() sino que llamamos autor.libaut, esto no modifica la consulta sql, es la misma del ejemplo anterior.

```
libro_y_editorial = Libro.objects.filter(titulo__contains='u').select_related(
'editorial')
```

```
autores = Autor.objects.filter(pk__in=(398,523)).prefetch_related(Prefetch('li
bro', queryset=libro_y_editorial, to_attr='libaut'))
   for autor in autores:
       print(f'Autor: {autor}')
       print('Libros escritos:')
       for libro in autor.libaut:
       print(f'{libro.isbn} Editorial: {libro.editorial.nombre}')
```

3.7.7 Relación inversa 

Hasta el momento hemos hecho las consultas muchos a muchos teniendo como base el modelo en el cual se especifico la relación, en este caso tomamos como base el modelo Autor y de allí buscamos sus Libros, pero que pasa si queremos realizar la consulta sobre el modelo en el que no esta especificada la relación en este caso es Libro.

Para realizar esto hacemos uso del nombre que le pusimos a nuestra relación en related_name='libros_autores', en este ejemplo consultamos algunos Libros y mostramos su Editorial y el Autor o Autores que los escribieron.

```
libros = self.filter(isbn__in=('1617290475', '1935182048')).select_related('editorial')
.prefetch_related('libros_autores')
```

```
for p in libros:
    print(f'{p.isbn} - {p.titulo} Editorial: {p.editorial.nombre}  Escrito por:')
    for q in p.libros_autores.all():
        print(f'{q.nombre} ')
```

Y como se que ya te esta gustando ver la sentencia SQL que genera, aquí te la dejo

```
SELECT "libreria_libro"."isbn",
       ...
       "libreria_editorial"."id",
       "libreria_editorial"."nombre"
  FROM "libreria_libro"
 INNER JOIN "libreria_editorial"
    ON ("libreria_libro"."editorial_id" = "libreria_editorial"."id")
 WHERE "libreria_libro"."isbn" IN ('1617290475', '1935182048')
```

```
SELECT ("libreria_autorcapitulo"."libro_id") AS "_prefetch_related_val_libro_id",
       "libreria_autor"."id",
       "libreria_autor"."nombre"
  FROM "libreria_autor"
 INNER JOIN "libreria_autorcapitulo"
    ON ("libreria_autor"."id" = "libreria_autorcapitulo"."autor_id")
 WHERE "libreria_autorcapitulo"."libro_id" IN ('1617290475', '1935182048')
```

3.8 Usando Vistas SQL 

Muchas veces necesitamos crear vistas directamente desde el SQL y llamarlas desde el ORM de Djando, esto porque podemos tener las vistas ya creadas y tardaríamos mas tiempo en traducirlas al ORM, también puede ser que algunas funciones de nuestro motor de base de datos aun no estén soportadas por el ORM o incluso que no sepamos como se utiliza una determinada función de SQL en el ORM y necesitamos implementarla de inmediato, sea cual sea nuestro caso afortunadamente Djando nos permite llamar a nuestras vistas que ya tenemos creadas en nuestra base de datos.

En este ejemplo queremos hacer una consulta que nos muestre el isbn y titulo de nuestro libro y los autores que lo escribieron pero estos los queremos en una sola columna y que estén separados por comas.

Esto es lo que queremos

isbn character varying (13	**titulo** character varying (70)	**autores** text
132612313	Object Technology Cent...	Timothy D. Korson,Vijay K. Vaishnavi
1617290394	Linked Data	David Wood,Marsha Zaidman,Luke Ruth,with Michael Ha...
1933988355	jQuery in Action	Bear Bibeault,Yehuda Katz
1932394621	wxPython in Action	Noel Rappin,Robin Dunn
1932394583	POJOs in Action	Chris Richardson

Postgres nos provee de una función llamada **string_agg** que nos pasa las filas a una cadena concatenada por el carácter que le especifiquemos, y esto es justo lo que necesitamos, desafortunadamente no sabemos hacerlo directamente con el ORM de Django y no tenemos el tiempo de investigarlo porque la consulta debe de ser mostrada a la voz de ya!

Como si sabemos utilizar la función en SQL vamos a crear una vista

```
CREATE OR REPLACE VIEW public.v_libroautores
 AS
 SELECT lib.isbn AS id,
    lib.isbn,
    lib.titulo,
    string_agg(aut.nombre::text, ','::text) AS autores
   FROM libreria_libro lib
     LEFT JOIN libreria_autorcapitulo rel ON rel.libro_id::text = lib.isbn::text
     LEFT JOIN libreria_autor aut ON aut.id = rel.autor_id
  GROUP BY lib.isbn, lib.titulo;
```

Ya que tenemos creada nuestra vista, transformamos su estructura a un modelo

Python manage.py inspectdb v_libroautores

Podemos ya crear un modelo con la estructura que nos arrojó ese comando, y notaremos que db_table anota al nombre de nuestra vista, también podrías crear el modelo a mano si gustas, pero recuerda que la consulta urge!

```python
class VLibroautores(models.Model):
    isbn = models.CharField(max_length=13, blank=True, null=True)
    titulo = models.CharField(max_length=70, blank=True, null=True)
    autores = models.TextField(db_column='Autores', blank=True, null=True)  # Field name made lowercase. This field type is a guess.

    class Meta:
        managed = False  # Created from a view. Don't remove.
        db_table = 'v_libroautores'
```

Listo podemos consultar nuestra vista como cualquier otro modelo de Django

```python
VLibroautores.objects.all()
```

Y nuestra sentencia SQL queda asi

```sql
SELECT "v_libroautores"."id",
       "v_libroautores"."isbn",
       "v_libroautores"."titulo",
       "v_libroautores"."autores"
  FROM "v_libroautores"
```

Ahora que ya entregamos nuestra consulta, podemos darnos el tiempo de investigar como realizar la misma consulta pero utilizando solo el ORM de Djando, resulta que la función en el ORM se llama StringAgg, después de un par de minutos nuestra consulta queda asi:

```python
from django.contrib.postgres.aggregates.general import StringAgg

libros = Libro.objects.filter(isbn__in=('1617290475', '1935182048')).prefetch_related('libros_autores').annotate(autores=StringAgg('libros_autores__nombre',delimiter=','))

for p in libros:
    print(f'{p.isbn} - {p.titulo} Autores: {p.autores}')
```

Y la sentencia SQL que nos genera es muy parecida a la de nuestra vista

```sql
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       ...
       STRING_AGG("libreria_autor"."nombre", ',') AS "autores"
  FROM "libreria_libro"
  LEFT OUTER JOIN "libreria_autorcapitulo"
    ON ("libreria_libro"."isbn" = "libreria_autorcapitulo"."libro_id")
  LEFT OUTER JOIN "libreria_autor"
    ON ("libreria_autorcapitulo"."autor_id" = "libreria_autor"."id")
 WHERE "libreria_libro"."isbn" IN ('1617290475', '1935182048')
 GROUP BY "libreria_libro"."isbn"
```

Me sigue gustando mas la sentencia SQL que escribirla en la forma ORM, en fin soy un generación X masoquista lo se.

3.9 Funciones de Ventana (Window Functions)

¿Qué son las funciones de ventana? 

Las funciones de ventana son parte del estándar de SQL desde el 2003, son muy útiles en el análisis de datos donde se realizan consultas complejas que involucran operaciones de clasificación, ordenación y agregación, a diferencia de las funciones agregadas las funciones de ventana operan sobre un conjunto de filas y nos devuelven un valor de agregación por cada fila .

Las funciones de ventana se clasifican en:

Funciones de Agregación: COUNT, SUM, MAX, MIN, AVG

Funciones de Clasificación: ROW_NUMBER, RANK, DENSE_RANK

Funciones de Valores: LAG, LEAD, FIRST_VALUE, LAST_VALUE

Las funciones de ventana dentro de Django se encuentran dentro de la librería **django.db.models.functions,** además deberemos de importar también la clase Window **django.db.models import Window**

Veamos como utilizar estas funciones con algunos ejemplos:

3.9.1 COUNT

Queremos mostrar los libros que tienen 600 paginas o mas y en una nueva columna contar cuantos libros hay con esas paginas.

```
contar = Window(
            expression=Count('*'),
            partition_by=[F('paginas')],
        )

filtro_row_number =(Libro.objects
            .annotate(num_libros=contar)
            .filter(paginas__gte=600)
   .values('isbn','paginas','num_libros')
)
```

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."paginas",
       COUNT(*) OVER (
         PARTITION BY "libreria_libro"."
paginas"
        ) AS "num_libros"
  FROM "libreria_libro"
 WHERE "libreria_libro"."paginas" >= 600
```

SQL

isbn [PK] character varying (13)	paginas integer	num_libros bigint
1884777775	600	6
1932394761	600	6
1935182056	600	6
1935182617	600	6
193239480X	600	6
1935182420	600	6
1932394621	620	1
1933988401	624	2
1932394052	624	2
1933988231	632	1

3.9.2 SUM, MAX, MIN, AVG

Queremos mostrar los libros agrupados por su editorial, solo se mostraran los libros que pertenecen a las editoriales Siglo XXI y Tecnos, y solo los que tienen 600 paginas o mas, vamos a agregar 5 columnas a nuestra consulta:

col_suma: Sumara el total de páginas de los libros de la editorial agrupada.

col_promedio: El promedio de páginas de los libros de la editorial agrupada.

col_minimo: El número mínimo de páginas de los libros de la editorial agrupada.

col_maximo: El número máximo de páginas de los libros de la editorial agrupada.

col_suma_acum: Sumara el número de páginas y acumulara la suma agrupada por editorial.

```
        ventana_conf = {
            'partition_by': [F('editorial__nombre')],
        }
        ventana_conf_acum = {
            'partition_by': [F('editorial__nombre')],
            'order_by': [F('isbn')],
        }
        expresion_suma = Window( expression=Sum('paginas'), **ventana_conf)
        expresion_min = Window( expression=Min('paginas'), **ventana_conf)
        expresion_max = Window( expression=Max('paginas'), **ventana_conf)
        expresion_prom = Window( expression=Avg('paginas'), **ventana_conf)
        expresion_suma_acum = Window( expression=Sum('paginas'), **ventana_conf_acum)

        filtro =(Libro.objects.select_related('editorial')
        .annotate(
                col_suma=expresion_suma,  col_promedio = expresion_prom,
                col_minimo = expresion_min, col_maximo = expresion_max,
                col_suma_acum = expresion_suma_acum
            )
            .filter(paginas__gte=600,editorial__nombre__in=('Siglo XXI','Tecnos'))
            .values('isbn','paginas','editorial__nombre',
            'col_suma','col_promedio','col_minimo','col_maximo','col_suma_acum'))
```

isbn character varyir	paginas integer	nombre character va	col_suma bigint	col_promedio numeric	col_minimo integer	col_maximo integer	col_suma_acum bigint
1884777775	600	Siglo XXI	2492	623.0000000000000000	600	672	600
1932394249	672	Siglo XXI	2492	623.0000000000000000	600	672	1272
1932394621	620	Siglo XXI	2492	623.0000000000000000	600	672	1892
1932394761	600	Siglo XXI	2492	623.0000000000000000	600	672	2492
1932394125	744	Tecnos	4240	706.6666666666666667	600	848	744
1932394222	656	Tecnos	4240	706.6666666666666667	600	848	1400
1932394613	680	Tecnos	4240	706.6666666666666667	600	848	2080
1933988347	712	Tecnos	4240	706.6666666666666667	600	848	2792
1935182048	848	Tecnos	4240	706.6666666666666667	600	848	3640
1935182056	600	Tecnos	4240	706.6666666666666667	600	848	4240

Casi lo olvido, aquí esta la sentencia SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."paginas",
       "libreria_editorial"."nombre",
       SUM("libreria_libro"."paginas") OVER (PARTITION BY "libreria_editorial"."nombre")
 AS "col_suma",
       AVG("libreria_libro"."paginas") OVER (PARTITION BY "libreria_editorial"."nombre")
 AS "col_promedio",
       MIN("libreria_libro"."paginas") OVER (PARTITION BY "libreria_editorial"."nombre")
 AS "col_minimo",
       MAX("libreria_libro"."paginas") OVER (PARTITION BY "libreria_editorial"."nombre")
 AS "col_maximo",
       SUM("libreria_libro"."paginas") OVER (PARTITION BY "libreria_editorial"."nombre"
ORDER BY "libreria_libro"."isbn") AS "col_suma_acum"
  FROM "libreria_libro"
 INNER JOIN "libreria_editorial"
    ON ("libreria_libro"."editorial_id" = "libreria_editorial"."id")
 WHERE ("libreria_editorial"."nombre" IN ('Siglo XXI', 'Tecnos') AND "libreria_libro"."p
aginas" >= 600)
```

3.9.3 ROW_NUMBER, RANK, DENSE_RANK

Queremos mostrar los libros agrupados por su editorial, solo se mostraran los libros que pertenecen a las editoriales Siglo XXI y Tecnos, y solo los que tienen 600 paginas o mas, vamos a agregar 3 columnas a nuestra consulta:

col_rownumber: Pondrá un numero consecutivo a las filas y estará agrupado por la editorial.

col_rank: Pondrá un ranking dependiendo del numero de paginas, si hay un empate el siguiente numero no será consecutivo.

col_denserank: Pondrá un ranking dependiendo del numero de paginas, a diferencia de Rank si hay un empate el siguiente numero si será consecutivo.

```
from django.db.models.functions import RowNumber, Rank, DenseRank
```

```
ventana_conf = {
    'partition_by': [F('editorial__nombre')],
    'order_by': [F('paginas')],
}

expresion_rownumber = Window( expression=RowNumber(), **ventana_conf)
expresion_rank = Window( expression=Rank(), **ventana_conf)
expresion_denserank = Window( expression=DenseRank(), **ventana_conf)

filtro =(Libro.objects.select_related('editorial')
.annotate(
        col_rownumber=expresion_rownumber,
        col_rank=expresion_rank,
        col_denserank = expresion_denserank
    )
    .filter(paginas__gte=600,editorial__nombre__in=('Siglo XXI','Tecnos'))
    .values('isbn','paginas','editorial__nombre',
    'col_rownumber','col_rank','col_denserank'))
```

*Si quisiéramos ordenar las páginas de manera descendente quedaría asi:

```
ventana_conf = {
    'partition_by': [F('editorial__nombre')],
    'order_by': [F('paginas').desc()],
}
```

Y aquí esta nuestra consulta SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."paginas",
       "libreria_editorial"."nombre",
       ROW_NUMBER() OVER (PARTITION BY "libreria_editorial"."nombre"
          ORDER BY "libreria_libro"."paginas") AS "col_rownumber",
       RANK() OVER (PARTITION BY "libreria_editorial"."nombre"
         ORDER BY "libreria_libro"."paginas") AS "col_rank",
       DENSE_RANK() OVER (PARTITION BY "libreria_editorial"."nombre"
       ORDER BY "libreria_libro"."paginas") AS "col_denserank"
  FROM "libreria_libro"
INNER JOIN "libreria_editorial"
   ON ("libreria_libro"."editorial_id" = "libreria_editorial"."id")
WHERE ("libreria_editorial"."nombre" IN ('Siglo XXI', 'Tecnos')
AND "libreria_libro"."paginas" >= 600)
```

isbn character varyin	paginas integer	nombre character varyir	col_rownumber bigint	col_rank bigint	col_denserank bigint
1932394761	600	Siglo XXI	1	1	1
1884777775	600	Siglo XXI	2	1	1
1932394621	620	Siglo XXI	3	3	2
1932394249	672	Siglo XXI	4	4	3
1935182056	600	Tecnos	1	1	1
1932394222	656	Tecnos	2	2	2
1932394613	680	Tecnos	3	3	3
1933988347	712	Tecnos	4	4	4
1932394125	744	Tecnos	5	5	5
1935182048	848	Tecnos	6	6	6

3.9.4 LAG, LEAD, FIRST_VALUE, LAST_VALUE

Queremos mostrar los libros agrupados por su editorial, solo se mostraran los libros que pertenecen a las editoriales Siglo XXI y Tecnos, y solo los que tienen 600 paginas o mas, vamos a agregar 3 columnas a nuestra consulta:

col_lag: Pondrá el numero de paginas de la fila anterior, agrupada por la editorial.

col_lead: Pondrá el numero de paginas de la fila siguiente, agrupada por la editorial.

col_first: Pondrá el numero de paginas de la primer fila, agrupada por la editorial

col_last: Pondrá el numero de paginas de la ultima fila, agrupada por la editorial.

```
from django.db.models.functions import Lag, Lead,FirstValue,LastValue
```

```
ventana_conf = { 'partition_by': [F('editorial__nombre')],'order_by': [F('isbn')],}

ventana_conf2 = {'partition_by': [F('editorial__nombre')],}

expresion_lag = Window( expression=Lag('paginas',1), **ventana_conf)
expresion_lead = Window( expression=Lead('paginas',1), **ventana_conf)
expresion_first = Window( expression=FirstValue('paginas'), **ventana_conf2)
expresion_last = Window( expression=LastValue('paginas'), **ventana_conf2)

filtro =(Libro.objects.select_related('editorial')
.annotate(
        col_lag=expresion_lag,
        col_lead=expresion_lead,
        col_first=expresion_first,
        col_last=expresion_last,
    )
    .filter(paginas__gte=600,editorial__nombre__in=('Siglo XXI','Tecnos'))
    .values('isbn','paginas','editorial__nombre',
    'col_lag','col_lead','col_first','col_last'))
```

isbn character varyin	paginas integer	nombre character v	col_lag integer	col_lead integer	col_first integer	col_last integer
1884777775	600	Siglo XXI	[null]	672	600	600
1932394249	672	Siglo XXI	600	620	600	600
1932394621	620	Siglo XXI	672	600	600	600
1932394761	600	Siglo XXI	620	[null]	600	600
1932394125	744	Tecnos	[null]	656	744	600
1932394222	656	Tecnos	744	680	744	600
1932394613	680	Tecnos	656	712	744	600
1933988347	712	Tecnos	680	848	744	600
1935182048	848	Tecnos	712	600	744	600
1935182056	600	Tecnos	848	[null]	744	600

Y finalmente la consulta SQL

```sql
SELECT "libreria_libro"."isbn",
       "libreria_libro"."paginas",
       "libreria_editorial"."nombre",
       LAG("libreria_libro"."paginas", 1)
            OVER (PARTITION BY "libreria_editorial"."nombre"
          ORDER BY "libreria_libro"."isbn") AS "col_lag",
       LEAD("libreria_libro"."paginas", 1)
            OVER (PARTITION BY "libreria_editorial"."nombre"
          ORDER BY "libreria_libro"."isbn") AS "col_lead",
       FIRST_VALUE("libreria_libro"."paginas")
            OVER (PARTITION BY "libreria_editorial"."nombre") AS "col_first",
       LAST_VALUE("libreria_libro"."paginas")
            OVER (PARTITION BY "libreria_editorial"."nombre") AS "col_last"
  FROM "libreria_libro"
 INNER JOIN "libreria_editorial"
    ON ("libreria_libro"."editorial_id" = "libreria_editorial"."id")
 WHERE ("libreria_editorial"."nombre" IN ('Siglo XXI', 'Tecnos')
```

4 Transacciones

4.1.1 Transacciones en todas nuestras vistas

La primera forma de utilizar transacciones dentro del orm de django es activando las transacciones automáticamente para todas nuestras vistas, esto se hace utilizando atomic_requests dentro de nuestro archivo settings.py en la parte de la cadena de conexión de la base de datos.

settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'orm_sql',
        'USER':  os.getenv('ORM_DB_USER'),
        'PASSWORD': os.getenv('ORM_DB_PASSWORD'),
        'HOST': 'localhost',
        'ATOMIC_REQUESTS': True,
    },
}
```

Ya con esto todas nuestras vistas automáticamente se ejecutaran dentro de una transacción.

Si quisiéramos que alguna de nuestras vistas no se ejecute dentro de una transacción utilizaremos el atributo **@transaction.non_atomic_requests**

views.py

```
from django.db import transaction
from django.shortcuts import render
from libreria.services import actualizar_libro_y_autor

# Create your views here.
@transaction.non_atomic_requests
def get_actualizar_libro_autor(request):
            actualizar_libro_y_autor()
            return render(request, "Libro y autor")
```

4.1.2 Usando el decorador de transacción

La siguiente manera de usar transacciones, es utilizando el decorador @transaction.atomic de esta manera la función que elijamos se ejecutara dentro de una transacción.

```
from django.db import transaction

@transaction.atomic
def actualizar_libro_y_autor():
    autor = Autor.objects.get(pk=273)
    libro = Libro.objects.get(isbn='1617291269')
    autor.nombre = "Autor de cobol"
    libro.titulo = "cobol"
    autor.save()
    libro.save()
    enviar_correo()
```

Debes de tener en cuenta que cualquier cosa que falle dentro de la función que fue decorada con @transaction.atomic, hará que la transacción se revierta, en el ejemplo si el envió de correo marcara algún error la transacción se revertiría aunque los datos se hubieran guardado bien, lo cual muchas veces no es lo que queremos.

4.1.3 Transacciones por bloques de código

Para poder especificar que bloques del codigo estarán involucrados dentro de la trasaccion utilizamos with transaction.atomic().

```
from django.db import transaction

def actualizar_libro_y_autor():
    autor = Autor.objects.get(pk=273)
    libro = Libro.objects.get(isbn='1617291269')
    autor.nombre = "Autor de cobol"
    libro.titulo = "cobol"
    with transaction.atomic():
        autor.save()
        libro.save()
    enviar_correo()
```

A diferencia del decorador @transaction.atomic, aquí nosotros especificamos que parte del código esta involucrada en la tracción, en este ejemplo si enviar_correo fallara no se revertiría la transacción ya que esta fuera de la transaccion, de esta manera tenemos un mejor control de nuestras transacciones.

4.1.4 Try/Except en las transacciones 

Podemos utilizar try/except dentro de nuestras transacciones para capturar los errores específicos que puedan suceder.

```
from django.db import transaction, IntegrityError

def actualizar_libro_y_autor():
    autor = Autor.objects.get(pk=273)
    libro = Libro.objects.get(isbn='1617291269')
    autor.nombre = "Autor de cobol"
    libro.titulo = "cobol"
     try:
         with transaction.atomic():
             autor.save()
             libro.save()
     except IntegrityError:
         "Error de integridad no puedes guardar ese titulo"
```

4.1.5 Realizar operaciones si la transacción fue exitosa 

En ocasiones queremos realizar alguna operación una vez que nuestra transacción fue exitosa, para eso utilizamos el método transaction.on_commit, en este ejemplo una vez que se guarden los datos se llamara a la función de envia_correo.

```
from django.db import transaction, IntegrityError

def actualizar_libro_y_autor():
    autor = Autor.objects.get(pk=273)
    libro = Libro.objects.get(isbn='1617291269')
    autor.nombre = "Autor de cobol"
    libro.titulo = "cobol"
    try:
        with transaction.atomic():
            transaction.on_commit(envia_correo)
            autor.save()
            libro.save()
    except IntegrityError:
        "Error de integridad no puedes guardar ese titulo"

def envia_correo():
    print("Correo enviado")
```

4.1.6 Savepoints 

Los savepoints permiten crear puntos de retorno dentro de un grupo de transacciones y poder regresar a ese punto si se necesita informar que la transacción fue exitosa o si existiera algún error poder revertir solo esa transacción sin tener que revertir la transacción completa, para utilizarlos dentro del orm de django usamos transacciones anidadas como en este ejemplo.

```
autor = Autor.objects.get(pk=273)
```

```
libro = Libro.objects.get(isbn='1617291269')
autor.nombre = "Autor de python"
libro.titulo = "aprende python"

with transaction.atomic():
    transaction.on_commit(lambda: print("primera transacción exitosa"))
    autor.save()
    print("Se guardo el autor")
    with transaction.atomic():
        transaction.on_commit(lambda: print("Segunda transacción exitosa"))
        libro.save()
        print("Se guardo el libro")
```

Y al ver el orden en que se ejecuta, nos daremos cuenta que hasta que no terminan las dos transacciones es cuando se comienzan a llamar a las funciones que imprimen que la transacción fue exitosa.

```
Se guardo el autor
```

```
SAVEPOINT "s3304_x1"

Se guardo el libro
RELEASE SAVEPOINT "s3304_x1"

primera transacción exitosa
Segunda transacción exitosa
```

5 NOSQL datos tipo Json

En el presente capitulo se mostrara como almacenar datos tipo Json dentro de nuestra base de datos de postgres o si estas usando Django 3.1 ya existe el soporte para otras bases de datos.

5.1 Definiendo un tipo de datos JSON

Para definir un campo de tipo Json dentro de nuestro modelo deberemos usar el tipo JSONField de la librería django.contrib.postgres.fields.

```
from django.contrib.postgres.fields
import JSONField

class Libro(models.Model):
    isbn = models.CharField(max_length=13
, primary_key=True)
    ...
    detalles = JSONField(null=True)
```

SQL

```
BEGIN;

--
-- Add field detalles to libro
--
ALTER TABLE "libreria_libro" ADD COLUMN
"detalles" jsonb NULL;
COMMIT;
```

En postgres existen dos tipos de datos para manejar json, uno es json que guarda los datos como texto, y el otro es jsonb que es el que usamos en el ejemplo y que guarda los datos como binarios lo cual permite que las consultas sean mas rápidas

Si estas usando Django 3.1 no es necesario importar la librería de postgres, y para definir tu tipo de datos json solo tendrías que poner models.JSONField de esta manera puedes usarlo en otras bases de datos que lo soportan como son MariaDB 10.2.7+, MySQL 5.7.8+, Oracle, PostgreSQL y SQLite 3.9.0+.

5.2 Guardando valores tipo Json

Una vez que tenemos definido un campo de tipo JSONField podemos almacenar datos en formato json, en este ejemplo vamos a consultar un libro y vamos a guardar en su campo detalles algunos datos en formato tipo json.

```
    libro = Libro.objects.filter(isbn='1933988495').update(
    detalles = {'genero': [
        'Programming', 'Python'], 'idioma': 'Ingles', 'Disponible': None, 'Estrellas': 5})
```

SQL

```
UPDATE "libreria_libro"
 SET "detalles" = '{"genero": ["Programming", "Python"], "idioma": "Ingles", "Disponible":
 null, "Estrellas": 5}'
 WHERE "libreria_libro"."isbn" = '1933988495'
```

5.3 Consultando valores Nulos

Si queremos saber si en nuestro campo tipo Json algún campo tiene valores nulos, en este ejemplo consultamos los libros que en su campo detalle y dentro de el en su campo disponible es igual a nulo.

```
libros = Libro.objects.values(
        'isbn', 'categoria', 'detalles')
.filter(detalles__Disponible=None)
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE ("libreria_libro"."detalles" ->
  'Disponible') = 'null'
```

isbn [PK] character v	categoria character varying (50)	detalles jsonb
1933988495	["Programming", "Python"]	{"genero": ["Programming", "Python"], "idioma": "Ingles", "Estrellas": 5, "Disponible": null}

5.4 Consultar si no existe un determinado campo

En este ejemplo queremos saber dentro de la columna tipo json Detalles cuales filas no incluyen el campo BestSeller.

```
Libro.objects.values('isbn', 'categoria',
'detalles').filter(
        detalles__BestSeller__isnull=True,
        isbn__in=('161729134X', '132632942
', '1933988495'))
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE (("libreria_libro"."detalles"
-> 'BestSeller') IS NULL
 AND "libreria_libro"."isbn" IN ('161
729134X', '132632942', '1933988495'))
```

isbn [PK] character va	categoria character varying (50)	detalles jsonb
132632942	["Java", "Business"]	{"genero": ["Java", "Business"], "idioma": "Español", "Disponible": 3}
1933988495	["Programming", "Python"]	{"genero": ["Programming", "Python"], "idioma": "Ingles", "Estrellas": 5, "Disponible": null}

5.5 Igual A

Dentro de un campo de tipo JSONField podemos hacer la mayoría de las consultas normales que solemos usar con el orm, en este ejemplo queremos saber que libros en detalles en su columna de idioma son igual a español.

```
Libro.objects.values('isbn', 'categoria',
 'detalles').filter(
        detalles__idioma='Español')
```

SQL

```
 SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE ("libreria_libro"."detalles" ->
'idioma') = '"Espa\u00f1ol"'
```

isbn [PK] character va	categoria character varying (50)	detalles jsonb
161729134X	[]	{"idioma": "Español", "BestSeller": true, "Disponible": 4}
132632942	["Java", "Business"]	{"genero": ["Java", "Business"], "idioma": "Español", "Disponible": 3}

5.6 Mayor o igual a

Queremos saber que libros en su campo detalle en su campo de disponible son mayor o igual a 3.

```
Libro.objects.values('isbn', 'categoria', 'detalles').filter(detalles__Disponible__gte=3)
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE ("libreria_libro"."detalles" -
> 'Disponible') >= '3'
```

isbn [PK] character v	categoria character varying (5	detalles jsonb
161729134X	[]	{"idioma": "Español", "BestSeller": true, "Disponible": 4}
132632942	["Java", "Business"]	{"genero": ["Java", "Business"], "idioma": "Español", "Disponible": 3}

5.7 OR

También podemos usar el condicional OR, en este ejemplo buscamos los libros cuyo BestSeller sea True o que su idioma sea Ingles

```
Libro.objects.values('isbn', 'categoria', 'detalles').filter(
  Q(detalles__BestSeller=True)
| Q(detalles__idioma='Ingles'))
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE (("libreria_libro"."detalles" ->
'BestSeller') = 'true' OR
("libreria_libro"."detalles" ->
'idioma') = '"Ingles"')
```

isbn [PK] character v	categoria character varying (50)	detalles jsonb
161729134X	[]	{"idioma": "Español", "BestSeller": true, "Disponible": 4}
1933988495	["Programming", "Python"]	{"genero": ["Programming", "Python"], "idioma": "Ingles", "Estrellas": 5, "Disponible": null}

5.8 Contains

Contains nos permite buscar que el valor de una o mas columnas coincida con un criterio de busqueda, en el siguiente ejemplo lo usamos para buscar los libros que sean del idioma Español y sean BestSeller.

```
Libro.objects.values('isbn', 'categoria',
 'detalles').filter(
 detalles__contains={'idioma': 'Español',
 'BestSeller': True})
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE "libreria_libro"."detalles" @> '
{"idioma": "Espa\u00f1ol", "BestSeller"
: true}'
```

isbn [PK] character	categoria character	detalles jsonb
161729134X	[]	{"idioma": "Español", "BestSeller": true, "Disponible": 4}

5.9 Consultando en Arrays

En un campo de tipo Json podemos guardar algunos datos en un array como en el caso de la columna género, en esta consulta buscamos los libros que en su array de genero tengan Python y Programming, aquí no importa el orden en el que estén guardados.

```
Libro.objects.values('isbn', 'categor
ia', 'detalles').filter(
detalles__genero__contains=['Python',
'Programming'])
```

SQL

```
 SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE ("libreria_libro"."detalles" -
> 'genero') @> '["Python", "Programming"]'
```

isbn [PK] character	categoria character varying (50)	detalles jsonb
1933988495	["Programming", "Python"]	{"genero": ["Programming", "Python"], "idioma": "Ingles", "Estrellas": 5, "Disponible": null}

5.10 Consulta por llave

Si quisiéramos saber si un campo json contiene una llave en específico podemos usar has_key, en el siguiente ejemplo buscamos los libros que tienen en su json de detalles el campo género.

```
Libro.objects.values('isbn', 'categoria', 'detalles').filter(
            detalles__has_key='genero')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE "libreria_libro"."detalles" ? 'genero'
```

isbn [PK] character	categoria character varying (50)	detalles jsonb
1933988495	["Programming", "Python"]	{"genero": ["Programming", "Python"], "idioma": "Ingles", "Estrellas": 5, "Disponible": null}
132632942	["Java", "Business"]	{"genero": ["Java", "Business"], "idioma": "Español", "Disponible": 3}

5.11 Consultar por cualquier llave

Si queremos buscar si un campo json contiene algunos campos usamos has_any_keys, en el siguiente ejemplo buscamos los libros que tengan el campo BestSeller o el campo Estrellas.

```
Libro.objects.values('isbn', 'categoria', 'detalles').filter(
detalles__has_any_keys=['BestSeller', 'Estrellas'])
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE "libreria_libro"."detalles" ?|
 ARRAY['BestSeller','Estrellas']
```

isbn [PK] character v	categoria character varying (50)	detalles jsonb
161729134X	[]	{"idioma": "Español", "BestSeller": true, "Disponible": 4}
1933988495	["Programming", "Python"]	{"genero": ["Programming", "Python"], "idioma": "Ingles", "Estrellas": 5, "Disponible": null}

5.12 Consulta por todas las llaves

Por ultimo cuando queremos buscar si el campo json tiene todas las llaves que le especifiquemos usamos has_keys, en el siguiente ejemplo buscamos los libros que tengan el campo genero, idioma, disponible y estrellas.

```
Libro.objects.values('isbn', 'categoria', 'detalles'
).filter(
detalles__has_keys=['genero', 'idioma', 'Disponible'
,'Estrellas'])
```


```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."categoria
",
       "libreria_libro"."detalles"
  FROM "libreria_libro"
 WHERE "libreria_libro"."detalles"
 ?& ARRAY['genero',
       'idioma',
       'Disponible',
```

isbn [PK] character var	categoria character varying (50)	detalles jsonb
1933988495	["Programming", "Python"]	{"genero": ["Programming", "Python"], "idioma": "Ingles", "Estrellas": 5, "Disponible": null}

6 Búsquedas de texto Completo (Full Text Search)

Cuando se trata de búsqueda de texto dentro de nuestra base de datos, por lo general utilizamos el operador de igualdad o el uso de like para buscar si el texto contiene una determinada palabra , estas búsquedas funcionan muy bien si el texto de la columna no es muy grande por ejemplo el nombre, pero si la columna contiene por ejemplo el texto de un articulo completo es aquí donde este tipo de búsquedas se quedan cortas, es aquí donde entran a nuestro rescate las consulta de texto completo (full text search) las cuales estas diseñadas para buscar palabras dentro de documentos.

En mundo de las bases de datos NoSQL existen varias alternativas como por ejemplo Elasticsearch o Apache Solr, afortunadamente en el mundo SQL muchas bases de datos ya lo están integrando entre ellas postgresql que es la que usaremos para los ejemplos además de que el orm ya nos permite integrarla.

6.1 Búsqueda básica

Este ejemplo es la forma de búsqueda mas básica dentro de full text search, aquí vamos a buscar dentro de la columna titulo la palabra network.

```
Libro.objects.filter(titulo__search='network').values('isbn','titulo')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo"
  FROM "libreria_libro"
 WHERE to_tsvector(COALESCE("libreria_libro"."titulo", '')) @@ (plainto_tsquery('network')
) = true
```

isbn [PK] character varying (13)	titulo character varying (70)
188477749X	Java Network Programming, Second Edition
1884777295	Building Secure and Reliable Network Applications

6.2 SearchVector para su funcionamiento

Para realizar este tipo de consultas postgresql necesita que los datos de la columna a consultar se pasen un tipo de datos llamado tsvector el cual contiene la lista de las palabras separadas, quitando las palabras que no tienen significado y pasando las palabras al singular, en este ejemplo vamos a convertir los datos de la columna titulo a un tsvector para ello usamos SearchVector.

```
Libro.objects.annotate(search=SearchVector('titulo',config='english')
).filter(search='network').values('isbn','titulo','search')
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       to_tsvector('english'::regconfig, COALESCE("libreria_libro"."titulo", '')) AS "search"
  FROM "libreria_libro"
 WHERE to_tsvector('english'::regconfig, COALESCE("libreria_libro"."titulo", '')) @@ (plainto
_tsquery('english'::regconfig, 'network')) = true
```

isbn [PK] character	titulo character varying (70)	search tsvector
013319955X	Comprehensive Networking ...	'acronym':5 'comprehens':1 'glossari':3 'guid':6 'network':2
188477749X	Java Network Programming,...	'edit':5 'java':1 'network':2 'program':3 'second':4
1884777295	Building Secure and Reliable...	'applic':6 'build':1 'network':5 'reliabl':4 'secur':2

6.3 Buscando en Español

Para poder tener mejores resultados en nuestras búsquedas, debemos de especificar el lenguaje en el que se realizaran, para ello se usa el atributo config=idioma, en este ejemplo usaremos el idioma español.

```
Libro.objects.annotate(search=SearchVector('titulo',config='spanish'),)
.filter(search='nade').values('isbn','titulo','search')
```

isbn [PK] character	titulo character varying (70)	search tsvector
013268327X	Diana nado con nadadores en la compentencia de natacion	'compentent':7 'dian':1 'nad':2 'nadador':4 'natacion':9

6.4 Buscando en mas de una columna

Nuestras búsquedas pueden realizarse en mas de una columna, en el siguiente ejemplo vamos a buscar la palabra mapreduce dentro de las columnas titulo y desc_corta.

```
(Libro.objects.annotate(search=SearchVector('titulo','desc_corta',config='english'))
        .filter(search='mapreduce').values('isbn','titulo','desc_corta','search'))
```

y este es nuestro pequeño monstruo de consulta en sql:

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."desc_corta",
       to_tsvector('english'::regconfig, COALESCE("libreria_libro"."titulo", '') || ' '
|| COALESCE("libreria_libro"."desc_corta", '')) AS "search"
  FROM "libreria_libro"
 WHERE to_tsvector('english'::regconfig, COALESCE("libreria_libro"."titulo", '') || ' '
|| COALESCE("libreria_libro"."desc_corta", '')) @@ (plainto_tsquery('english'::regconfig
, 'mapreduce')) = true
```

y como podemos ver la palabra mapreduce la encontró en la columna desc_corta:

isbn [PK] character	titulo character varying (70)	desc_corta character varying (2000)
1935182196	Hadoop in Action Book	Hadoop in Action teaches readers how to use Hadoop and write MapReduce progr...

6.5 Usando SearchQuery para consultas mas complejas

Hasta el momento para nuestras consultas postgres ha utilizado plain_tsquery con lo cual solo podemos hacer búsquedas de texto plano, si queremos realizar consultas mas complejas en las que podamos utilizar operadores como and, or, not e inclusive utilizar comodines, para ello el ORM nos proporciona la clase SearchQuery en la que podemos especificarle el tipo de búsqueda que queremos, existen 4 tipos:

plain	Texto plano
phrase	Busca frases
raw	Nos permite utilizar comodines y operadores como and,or,not
websearch	Nos permite realizar búsqueda con operadores como los que se usan en los motores de búsqueda web.

6.6 Usando comodines

Podemos realizar búsquedas de tipo like, para ello utilizamos el tipo de búsqueda raw y el comodin * , en este ejemplo buscamos las filas que tengan cualquier palabra que comience con netwo.

```
from django.contrib.postgres.search import  SearchQuery

 (Libro.objects.annotate(search=SearchVector('titulo',config='english'),)
.filter(search=SearchQuery('netwo:*', search_type='raw')).values('isbn','titulo'))
```

Y como podemos ver nuestra consulta sql ahora utiliza to_tsquery en lugar de plainto_tsquery:

```
 SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo"
  FROM "libreria_libro"
 WHERE to_tsvector('english'::regconfig, COALESCE("libreria_libro"."titulo", ''))
@@ (to_tsquery('netwo:*')) = true
```

isbn [PK] character	titulo character varying (70)
013319955X	Comprehensive Networking Glossary and Acronym Guide
188477749X	Java Network Programming, Second Edition
1884777295	Building Secure and Reliable Network Applications
137353006	Client/Server Applications on ATM Networks
131271687	SNA and TCP/IP Enterprise Networking

6.7 Usando OR

En este buscaremos la palabra program o la palabra programmer

```
from django.contrib.postgres.search import  SearchQuery,SearchVector

(Libro.objects.annotate(search=SearchVector('titulo',config='english'),)
.filter(search=SearchQuery('program | programmer', search_type='raw')).values('isbn',
'titulo'))
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo"
  FROM "libreria_libro"
 WHERE to_tsvector('english'::regconfig, COALESCE("libreria_libro"."titulo", ''))
 @@ (to_tsquery('program | programmer')) = true
```

isbn [PK] character	titulo character varying (70)
1884777554	PFC Programmer's Reference Manual
131723979	Visual Object Oriented Programming
1930110405	LDAP Programming, Management and Integration
1884777813	Python and Tkinter Programming
1930110197	Microsoft.NET for Programmers

6.8 Usando AND

En este ejemplo buscamos los títulos que tengan la palabra program y la palabra network

```
from django.contrib.postgres.search import  SearchQuery,SearchVector

(Libro.objects.annotate(search=SearchVector('titulo',config='english'),)
.filter(search=SearchQuery('program & network', search_type='raw')).values('isbn'
,'titulo'))
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo"
  FROM "libreria_libro"
 WHERE to_tsvector('english'::regconfig, COALESCE("libreria_libro"."titulo", ''))
@@ (to_tsquery('program & network')) = true
```

isbn [PK] character varying (13)	titulo character varying (70)
188477749X	Java Network Programming, Second Edition

6.9 Usando SearchVectorField como columna del modelo

Hasta el momento nuestras consultas las realizamos convirtiendo nuestra columna a un tsvector en el mismo momento de la consulta, esto hace que sea mucho mas lenta si tenemos muchos datos, para solventar esto vamos a agregar una nueva columna donde se pueda almacenar este vector, para ello el orm nos proporciona el tipo de datos SearchVectorField.

```
class Libro(models.Model):
    isbn = models.CharField(max_length=13, primary_key=True)
    ...
    dsc_corta_token = SearchVectorField(null=True)
```

SQL

```
--
-- Add field dsc_corta_token to libro
--
ALTER TABLE "libreria_libro" ADD COLUMN "dsc_corta_token" tsvector NULL;
```

Ahora solo nos que pasar los datos de la columna desc_corta a la nueva columna pero convertidos en un tsvector.

```
from django.contrib.postgres.search import SearchVector

Libro.objects.all().update(dsc_corta_token=SearchVector(F('desc_corta'),config='english'))
```

SQL

```
UPDATE "libreria_libro"
   SET "dsc_corta_token" = to_tsvector('english'::regconfig, COALESCE("libreria_libro"."desc_corta", ''))
```

Y ahora ya podemos realizar nuestras consultas de full text search sin necesidad de estar convirtiendo nuestra columa a vector en cada consulta que hagamos, y nuestra consulta ahora quedaría asi:

```
(Libro.objects.filter(dsc_corta_token=SearchQuery('introduction', search_type='raw', config='english')).values('isbn','desc_corta'))
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."desc_corta"
  FROM "libreria_libro"
 WHERE "libreria_libro"."dsc_corta_token" @@ (to_tsquery('english'::regconfig, 'introduction')) = true
```

6.10 Manteniendo actualizados los datos del vector 

Nos queda un problema por resolver y es que si insertamos o modificamos un dato de la columna desc_corta este no se vera reflejado en la columna dsc_corta_token, así que necesitamos una forma de hacerlo, tenemos tres opciones, la primera es crear un disparador que cada que suceda un cambio en esa columna lo convierta a tsvector y lo guarde en la columna dsc_corta_token, la segunda es realizar esa misma operación pero desde el modelo esto usando el evento save, y la tercera opción es crear una columna que automáticamente genere el valor, es como un tipo de columna calculada pero con la diferencia de que los datos si se persistirán en la base de datos, esta ultima opción es la que vamos a usar y se puede utilizar a partir de postgres 13.

Para hacer esto podemos hacer una migración que contenga la sentencia sql donde se especifica que esa columna cada que exista un cambio generara un tsvector de la columna desc_corta y lo guardara en la columna dsc_corta_token.

Nuestro archivo de migración quedaría de la siguiente manera:

```
ALTER TABLE public.libreria_libro  drop COLUMN IF EXISTS dsc_corta_token;
```

```
ALTER TABLE public.libreria_libro
ADD COLUMN dsc_corta_token tsvector
               GENERATED ALWAYS AS (to_tsvector('english', coalesce(titulo, '') || ' '
|| coalesce(desc_corta, ''))) STORED;
```

Con esto cada que exista un cambio nuestra columna dsc_corta_token tendrá los datos actualizados, solo recuerda que no podrás actualizar directamente tu la columna dsc_corta_token.

6.11 Buscando frases 

Podemos buscar frases completas, esto lo podemos realizar de 3 maneras:

En este ejemplo estamos buscando la frase advance features:

```
 (Libro.objects.filter(dsc_corta_token=SearchQuery('advance <-> features',
search_type='raw', config='english')).values('isbn','desc_corta'))
```

o buscar que las palabras se encuentren con un poco de distancia

```
(Libro.objects.filter(dsc_corta_token=SearchQuery('advance <2> features', search_type='raw
',config='english')).values('isbn','desc_corta'))
```

otra manera es especificarle que el tipo de búsqueda es de tipo frase

```
(Libro.objects.filter(dsc_corta_token=SearchQuery('advance features', search_type='phrase'
,config='english')).values('isbn','desc_corta'))
```

6.12 Búsqueda y Ranking 

La función de rango ts_rank pone un ranking de la búsqueda en función del numero de veces en que se encuentra cada termino y la posición dentro del documento.

En el ejemplo estamos buscando la palabra Python , pero filtramos solo las filas cuyo ranking de esa columna sea mayor a 0.05.

```
    query = SearchQuery('python',config='english')
```

```
    libros = (Libro.objects.annotate(
        rank=SearchRank("dsc_corta_token", query)
        )
        .filter(rank__gte=0.05)
        .order_by('-
rank')).values('isbn','dsc_corta_token','rank',)
    print(libros)
```

6.13 Búsquedas de similaridad

Las búsquedas de similaridad o también conocidas como Trigram nos permiten comparar dos cadenas y determinar que tan similares son.

En este ejemplo estamos buscando los autores en donde el nombre ‘maicol barlota’ tengan una similaridad de mas del 0.3

```
from django.contrib.postgres.search import  TrigramSimilarity

Autor.objects.annotate(similarity=TrigramSimilarity('nombre', 'maicol barlota'),).
filter(similar ity__gt=0.3).values('nombre','similarity')
```

SQL

```
SELECT "libreria_autor"."nombre",
       SIMILARITY("libreria_autor"."nombre", 'maicol barlota') AS "similarity"
  FROM "libreria_autor"
 WHERE SIMILARITY("libreria_autor"."nombre", 'maicol barlota') > 0.3
```

Y estos son los nombre de autores que mas se parecen a ‘maicol barlota’

nombre character varying (70)	similarity real
Michael J. Barlotta	0.30769232
MIchael Barlotta	0.33333334
Michael Barlotta	0.33333334

7 ¿Cómo lo hago?

7.1 Ejecutar un procedimiento almacenado

En la actualidad la tendencia ya no es poner la lógica del negocio dentro de los procedimientos almacenados, sino en el backend, sin embargo, aun existen algunos casos donde los procedimientos almacenado todavía nos pueden ser útiles, por ejemplo:

- Implementar funciones de nuestro motor de base de datos que aún no han sido implementadas en el orm.
- Procesar una gran cantidad de información.
- Integrar aplicaciones donde la lógica del negocio se encuentra en procedimientos almacenados.

En este ejemplo tenemos dos funciones en postgres, las cuales hacen una consulta de las editoriales dependiendo del valor a buscar.

La primera función solo recibe un parámetro y devuelve las editoriales encontradas de acuerdo al parametro búsqueda.

```
CREATE OR REPLACE FUNCTION mi_funcion(buscar varchar(50))
RETURNS TABLE(id int,nombre varchar(100))
AS $$
BEGIN
RETURN QUERY
    SELECT a.id,a.nombre
    FROM libreria_editorial as a
    WHERE buscar = CASE WHEN buscar='todos' then buscar else a.nombre END;

END;
$$ LANGUAGE plpgsql;
```

Y la forma de llamarlo desde el orm

```
from django.db import connection

c=connection.cursor()

c.callproc('mi_funcion',['Alianza'])
Out: ['Alianza']

c.fetchall()
Out: [(2, 'Alianza')]

c.close()
```

```
from django.db import connection

c=connection.cursor()

c.callproc('mi_funcion',['todos'])
Out: ['todos']

c.fetchall()
Out:
[(1, 'Ariel'), (2, 'Alianza'),
 (3, 'Tecnos'), (4, 'SÃ\xadntesis'),
 (5, 'CÃ\xadvitas'), (6, 'Siglo XXI'),
 (7, 'Tirant lo Blanch'), (8, 'PirÃ¡mide'),
 (9, 'CrÃ\xadtica'), (10, 'McGraw-Hill')]
```

La segunda función es un ejemplo de cuando tenemos dos variables de salida, en este caso son resultado y saludo.

```
CREATE OR REPLACE FUNCTION mi_funcion_out(buscar varchar(50),out resultado json,out saludo
varchar(10))
AS $$
BEGIN
     resultado=array_to_json(array_agg(row_to_json(t)))
    from (
         SELECT a.id,a.nombre
         FROM libreria_editorial as a
         WHERE buscar = CASE WHEN buscar='todos' then buscar else a.nombre END
    ) t;
    saludo='Hola mundo';
END;
$$ LANGUAGE plpgsql;
```

Y la forma de llamarlo desde el orm

```
# Para no tener que estar cerrando la conexión usamos with
with connection.cursor() as cursor:
    cursor.callproc('mi_funcion_out',['todos'])
    # Usamos fetchone porque no estamos devolviendo
    # un conjunto de datos sino solo las variables de salida
    resultado = cursor.fetchone()

# y podemos acceder a nuestras variables por su indice
# [0] resultado
# [1] saludo
resultado[0]
Out:
[{'id': 1, 'nombre': 'Ariel'},
 {'id': 2, 'nombre': 'Alianza'},
 {'id': 3, 'nombre': 'Tecnos'},
resultado[1]
Out:
 'Hola mundo'
```

7.2 Consultas dinámicas

Muchas veces nos encontramos en situaciones donde necesitamos que dependiendo de los parámetros que se le pasen a nuestra función se ejecute una condición diferente en nuestra consulta en cada caso.

Por ejemplo si tenemos una tabla de libros, si a nuestro parámetro de búsqueda le enviamos números la búsqueda será por medio del isbn pero si le enviamos letras la búsqueda será por el titulo, nuestro ejemplo quedaría de la siguiente manera:

```python
def buscar_libro(buscar, con_paginas=False):
    filtro={}
    col_numero = 'isbn'
    col_letras = 'titulo'

    if(con_paginas):
        filtro['paginas__gt'] = 0

    if(buscar.isnumeric()):
        filtro[col_numero] = buscar
    else:
        filtro[f'{col_letras}__contains'] = buscar

    libro = Libro.objects.filter(**filtro).values('isbn','titulo','paginas')
```

Si llamamos nuestra función pasándole solo números:

```python
buscar_libro('1933988592')
```

SQL

```sql
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas"
  FROM "libreria_libro"
 WHERE "libreria_libro"."isbn" = '1933988592'
```

Y si le pasamos letras como parámetros además de indicarle que queremos los que tengan páginas:

```python
buscar_libro('python',True)
```

SQL

```sql
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas"
  FROM "libreria_libro"
 WHERE ("libreria_libro"."paginas" > 0
 AND "libreria_libro"."titulo"
           LIKE '%python%' ESCAPE '\')
```

Podemos complicar un poco más las cosas y hacer que nuestra función pueda buscar por dos columnas cuando le pasemos letras como parámetro, en este ejemplo buscaría por la columna título o la columna categoría:

```
def buscar_libro2(buscar, con_paginas=False):
    from django.db.models import Q

    filtro={}
    col_numero = 'isbn'
    col_letras = ['titulo','categoria']
    query1 = Q()

    if(buscar.isnumeric()):
        filtro[col_numero] = buscar
    else:
        for nom_col in col_letras:
            condicion = {f'{nom_col}__contains': buscar }
            query1 |= Q(**condicion)
    query1.connector = 'OR'

    if(con_paginas):
        query1.add(Q(paginas__gt=0), Q.AND)


    libro = Libro.objects.filter(query1).values('isbn','titulo','paginas')
```

Asi que si ejecutamos nuestra función pasándole letras como parámetro, veremos que busca el valor en la columna titulo o en la columna categoría, además de que si le indicamos que queremos solo los libros con paginas esta condición será independiente.

```
buscar_libro2('python',True)
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       "libreria_libro"."paginas"
  FROM "libreria_libro"
 WHERE (
 ("libreria_libro"."titulo" LIKE '%python%'
                    OR
"libreria_libro"."categoria" LIKE '%python%')

   AND "libreria_libro"."paginas" > 0
  )
```

7.3 Concatenar filas

En ocasiones necesitamos concatenar nuestras filas en una sola columna separadas por algún carácter especial, el orm de django nos provee el acceso a una función que nos permite hacer esto de maner rápida, el único problema es que solo esta disponible para postgres, si estas usando alguna otra base de datos diferente siempre existen funciones de terceros que hacen lo mismo.

La función que nos sirve para realizar esto se llama StringAgg y se encuentra dentro de django.contrib.postgres.aggregates.

En este ejemplo vamos a hacer una relación de los libros con sus autores y el nombre de los autores quera concatenado en una sola columna separado por algún carácter especial, de tal forma que obtendríamos algo similar a esto:

isbn [PK] character	titulo character varying (70)	autores text
1617290475	SQL Server MVP Deep Dives,...	Kimberly Tripp,Kalen Delaney,Louis Davidson,Greg Low,Brad McGehee,Paul Nielsen,Paul R...
1935182048	SQL Server MVP Deep Dives	Contributions from 53 SQL Server MVPs,Edited by Paul Nielsen,Kalen Delaney,Greg Low,A...

El ejemplo usando el orm de django quedaría de la siguiente manera:

```
# Traemos la libreria que nos permite concatenar
```

```
from django.contrib.postgres.aggregates import StringAgg

# Filtramos solo algunos libros de ejemplo
libros = Libro.objects.filter(isbn__in=('1617290475', '1935182048'))
# En este caso tenemos una relacon libro con autores
# por lo que usamos prefetch_relate
libros = libros.prefetch_related('libros_autores')

# a StringAgg le podemos especificar algunos parametros si asi lo necesitamos
parametros = {
    'delimiter': ', ',  # El delimitador de separacion
    'ordering': (F('libros_autores__nombre')), # Si queremos que se ordene el resultado po
r algun campo
    'distinct': True, # Si queremos eliminar los duplicados en el resultado
    'filter': Q(libros_autores__nombre__contains='Paul'), # Si queremos filtrar el resulta
do
    }

libros = libros.annotate(autores=StringAgg('libros_autores__nombre',**parametros))

# Mostramos el resultado de nuestra consulta
for l in libros:
    print(f'Libro: {l.isbn} Autores: {l.autores}')

# La salida seria esta:
#Libro: 1617290475 Autores: Paul Nielsen, Paul Randal
#Libro: 1935182048 Autores: Edited by Paul Nielsen, Paul S. Randal
```

Y nuestra consulta SQL quedaría asi:

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."titulo",
       ...
       STRING_AGG(DISTINCT "libreria_autor"."nombre",
                  ', 'ORDER BY "libreria_autor"."nombre")
                 FILTER (WHERE "libreria_autor"."nombre"::text
                          LIKE '%Paul%') AS "autores"
  FROM "libreria_libro"
  LEFT OUTER JOIN "libreria_autorcapitulo"
    ON ("libreria_libro"."isbn" = "libreria_autorcapitulo"."libro_id")
  LEFT OUTER JOIN "libreria_autor"
    ON ("libreria_autorcapitulo"."autor_id" = "libreria_autor"."id")
 WHERE "libreria_libro"."isbn" IN ('1617290475', '1935182048')
 GROUP BY "libreria_libro"."isbn"
```

Ademas de StringAgg django tambien nos proporciona ArrayAgg y JSONBAgg funcionan de manera similar solo que uno devuelve los resultados en un array y el otro en un formato json, otra diferencia es que JSONBAgg no admite el parametro distinct.

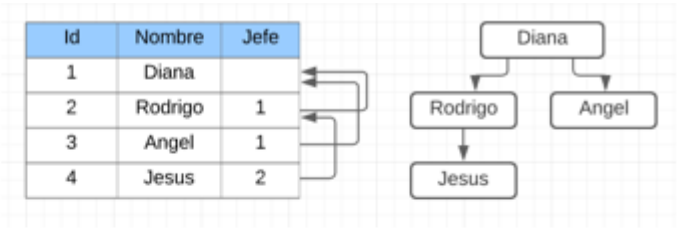
7.4 Consultas recursivas

La recursividad en una técnica muy utilizada en la programación, y consiste en que una función se llame a si misma, los ejemplos mas conocidos donde se puede usar es en el factorial de un numero o la serie Fibonacci.

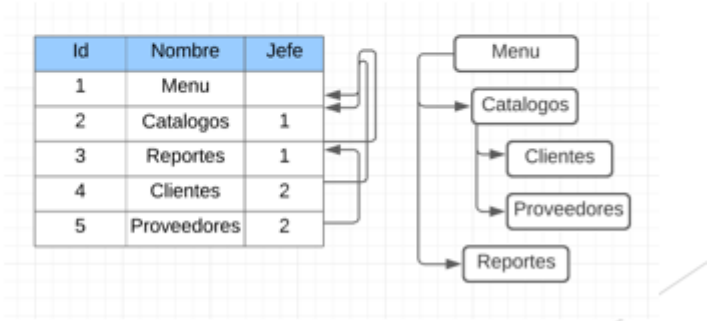
7.4.1 ¿En que casos podríamos usar la recursividad en el modelado de base de datos?

Estos son algunos ejemplos en donde podemos utilizar la recursividad:

Estructura organizacional



Estructura de un menú



Partes y subpartes de artículos

7.4.2 Formas de realizarla

1. **Traer todos los datos y procesarlos desde el backend:** Esta solución nos funciona si los datos a traer no son muchos, por ejemplo en una estructura de un menú.

2. **Hacer una función recursiva en el backend que realice las consultas SQL:** Aquí se realiza una consulta recursiva por cada uno de los nodos, un ejemplo:

Tenemos la siguiente relación y dando el isbn de algún libro debemos obtener toda su jerarquia:

	isbn [PK] character varying (13)	**titulo** character varying (70)	**edicion_anterior_id** character varying (13)
1	1884777740	Python Ed 1	[null]
2	1884777813	Python Ed2	1884777740
3	1932394621	Python Ed3 r1	1884777813
4	1935182080	Python Ed3 r2	1884777813
5	193518220X	Python Ed4	1935182080

Un ejemplo de resolverla de esta manera seria así:

```
def descendientes_dic(isbn=None, padre=None):
    lista = []
    if isbn is not None:
        padre = Libro.objects.values('isbn','edicion_anterior').get(isbn=isbn)

    lista.append(padre)
    # Buscamos sus descendiente inmediatos
    hijos=Libro.objects.values('isbn','edicion_anterior').filter(
        edicion_anterior= padre['isbn'])

    for h in hijos:
        d = descendientes_dic(padre = h)
        lista.append(d)
    return lista
```

Y la forma de llamarla :

```
descendientes_dic('1884777740')
```

```
Resultado:
 [{'isbn': '1884777740', 'edicion_anterior': None},
 [{'isbn': '1884777813', 'edicion_anterior': '1884777740'},
  [{'isbn': '1935182080', 'edicion_anterior': '1884777813'},
   [{'isbn': '193518220X', 'edicion_anterior': '1935182080'}
]],
  [{'isbn': '1932394621', 'edicion_anterior': '1884777813'}]
]]
```

SQL

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."edicion_anterior_id"
  FROM "libreria_libro"
 WHERE "libreria_libro"."isbn" = '1884777740'
SELECT "libreria_libro"."isbn",
       "libreria_libro"."edicion_anterior_id"
  FROM "libreria_libro"
 WHERE "libreria_libro"."edicion_anterior_id" = '18847
77740'
SELECT "libreria_libro"."isbn",
       "libreria_libro"."edicion_anterior_id"
  FROM "libreria_libro"
 WHERE "libreria_libro"."edicion_anterior_id" = '18847
77813'
SELECT "libreria_libro"."isbn",
       "libreria_libro"."edicion_anterior_id"
  FROM "libreria_libro"
 WHERE "libreria_libro"."edicion_anterior_id" = '19351
82080'
SELECT "libreria_libro"."isbn",
       "libreria_libro"."edicion_anterior_id"
  FROM "libreria_libro"
 WHERE "libreria_libro"."edicion_anterior_id" = '19351
8220X'
SELECT "libreria_libro"."isbn",
       "libreria_libro"."edicion_anterior_id"
  FROM "libreria_libro"
 WHERE "libreria_libro"."edicion_anterior_id" = '19323
94621'
```

**Problema: Estamos realizando una consulta al motor de base de datos por cada uno de los elementos descendiente del nodo

3. Utilizar CTE recursivos (expresión de tabla común).

Podríamos realizar la consulta anterior utilizando CTE en SQL, el nuestro en este ejemplo quedaría de la siguiente manera en SQL.

```
WITH RECURSIVE LibroEdiciones(isbn,edicion_anterior,nivel)
AS(
select a.isbn,a.edicion_anterior_id, 0
from libreria_libro AS a
where a.isbn='1884777740'
UNION ALL
select a.isbn,a.edicion_anterior_id, LibroEdiciones.nivel + 1
from libreria_libro AS a
join LibroEdiciones ON a.edicion_anterior_id = LibroEdiciones.isbn
    )

select * from LibroEdiciones
```

El resultado de esta consulta seria:

	isbn character varying (13)	edicion_anterior character varying (13)	nivel integer
1	1884777740	[null]	0
2	1884777813	1884777740	1
3	1935182080	1884777813	2
4	1932394621	1884777813	2
5	193518220X	1935182080	3

La pregunta es ¿Cómo llamar esta consulta desde django?

Ya que el ORM de django por el momento no implementa el uso de CTE, tenemos tres opciones:

- Ejecutar el sql directamente utilizando raw(): Esta es la ultima opción que deberíamos utilizar, por el peligro de inyección de sql.
- Crear un procedimiento almacenado y llamarlo desde django: La desventaja de esta opción es que tendríamos que crear el procedimiento desde una migración para que quien baje nuestros cambios lo pueda utilizar, además de que la sintaxis para crear procedimientos almacenados puede variar dependiendo del motor de base de datos.

- Utilizar alguna librería de terceros para el manejo de CTE.
  Para este ejemplo vamos a utilizar django-cte https://github.com/dimagi/django-cte

Primero lo instalamos como cualquier otro paquete:

```
pip install django-cte
```

y nuestra consulta utilizando esta librería quedaría asi:

```python
def busqueda_recursiva(isbn_buscar):
    def cte_recursivo(cte):
        return Libro.objects.filter(
            isbn=isbn_buscar
        ).values('isbn','edicion_anterior',
                 prof=V(0, output_field=models.IntegerField()),
        ).union(
            cte.join(Libro, edicion_anterior = cte.col.isbn).values(
                'isbn','edicion_anterior',
                prof=cte.col.prof +1,
            ),
            all=True # UNION ALL
        )
    cte = With.recursive(cte_recursivo)
    resultado = cte.queryset().with_cte(cte).annotate(
        path=ExpressionWrapper(
            Concat(
                Repeat(V('|---'),F('prof')),
                F('isbn')
                ),
                output_field=CharField(30)))

    for libro in resultado:
        print(libro)
```

```
Execution time: 0.007972s [Database: default]

{'isbn': '1884777740', 'edicion_anterior': None, 'prof': 0, 'path': '1884777740'}
{'isbn': '1884777813', 'edicion_anterior': '1884777740', 'prof': 1, 'path': '|---1884777813'}
{'isbn': '1935182080', 'edicion_anterior': '1884777813', 'prof': 2, 'path': '|---|---1935182080'}
{'isbn': '1932394621', 'edicion_anterior': '1884777813', 'prof': 2, 'path': '|---|---1932394621'}
{'isbn': '193518220X', 'edicion_anterior': '1935182080', 'prof': 3, 'path': '|---|---|---193518220X'}
```

Y la consulta SQL que nos generaría es:

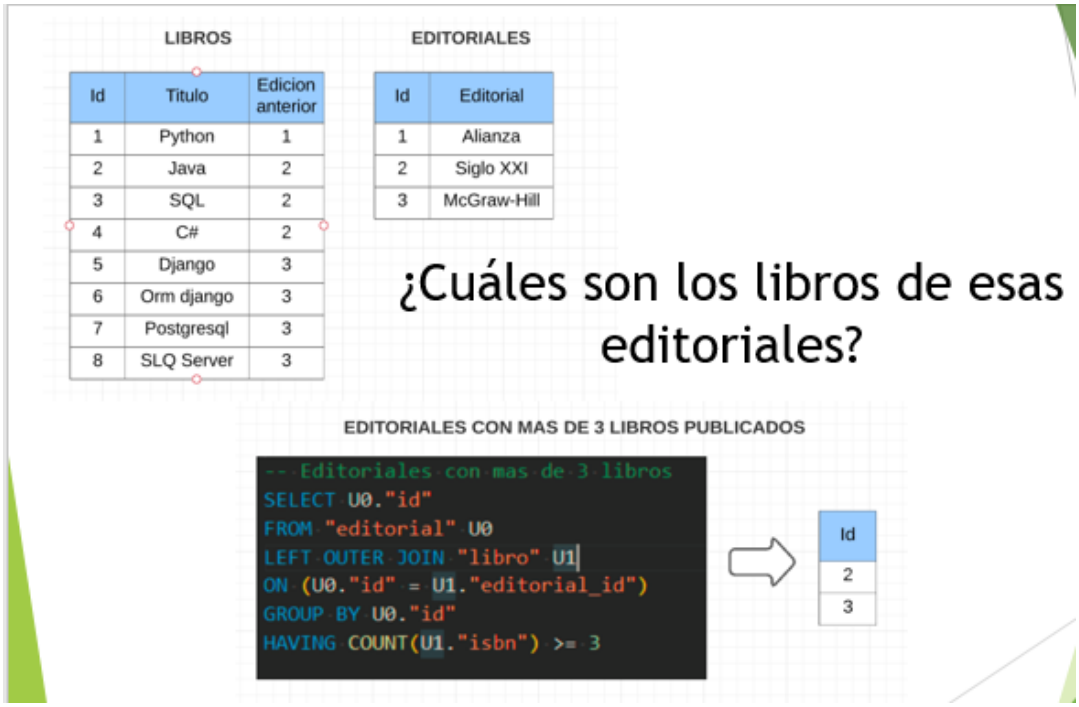
```sql
WITH RECURSIVE cte AS (
   (SELECT "libreria_libro"."isbn", "libreria_libro"."edicion_anterior_id", 0 AS "prof"
   FROM "libreria_libro" WHERE "libreria_libro"."isbn" = '1884777740')
   UNION ALL
   (SELECT "libreria_libro"."isbn", "libreria_libro"."edicion_anterior_id",
           ("cte"."prof" + 1) AS "prof"
    FROM "libreria_libro" INNER JOIN "cte"    ON "libreria_libro"."edicion_anterior_id" = ("cte"."isbn")))
    SELECT "cte"."isbn", "cte"."edicion_anterior_id",
      "cte"."prof" AS "prof",  CONCAT(REPEAT('|---', "cte"."prof"), "cte"."isbn") AS "path"
  FROM "cte"
```

7.5 SubConsultas (subqueries)

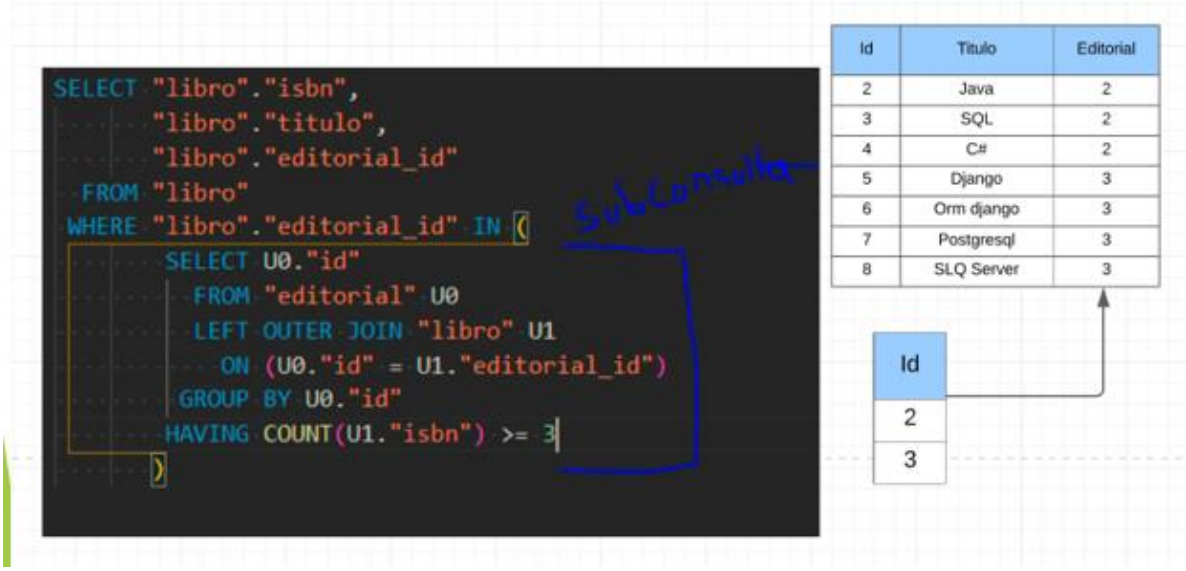
Una subconsulta es una consulta sql que tiene otra consulta sql dentro de su clausula where, o puede ser también parte de una columna.

7.5.1 Subconsultas no correlacionadas:

Dado el siguiente modelo, queremos ver cuales son las editoriales que tienen mas de tres libros publicados.



Si quisiéramos saber cuales son los libros de esas dos editoriales, podríamos hacer una subconsulta como la siguiente:



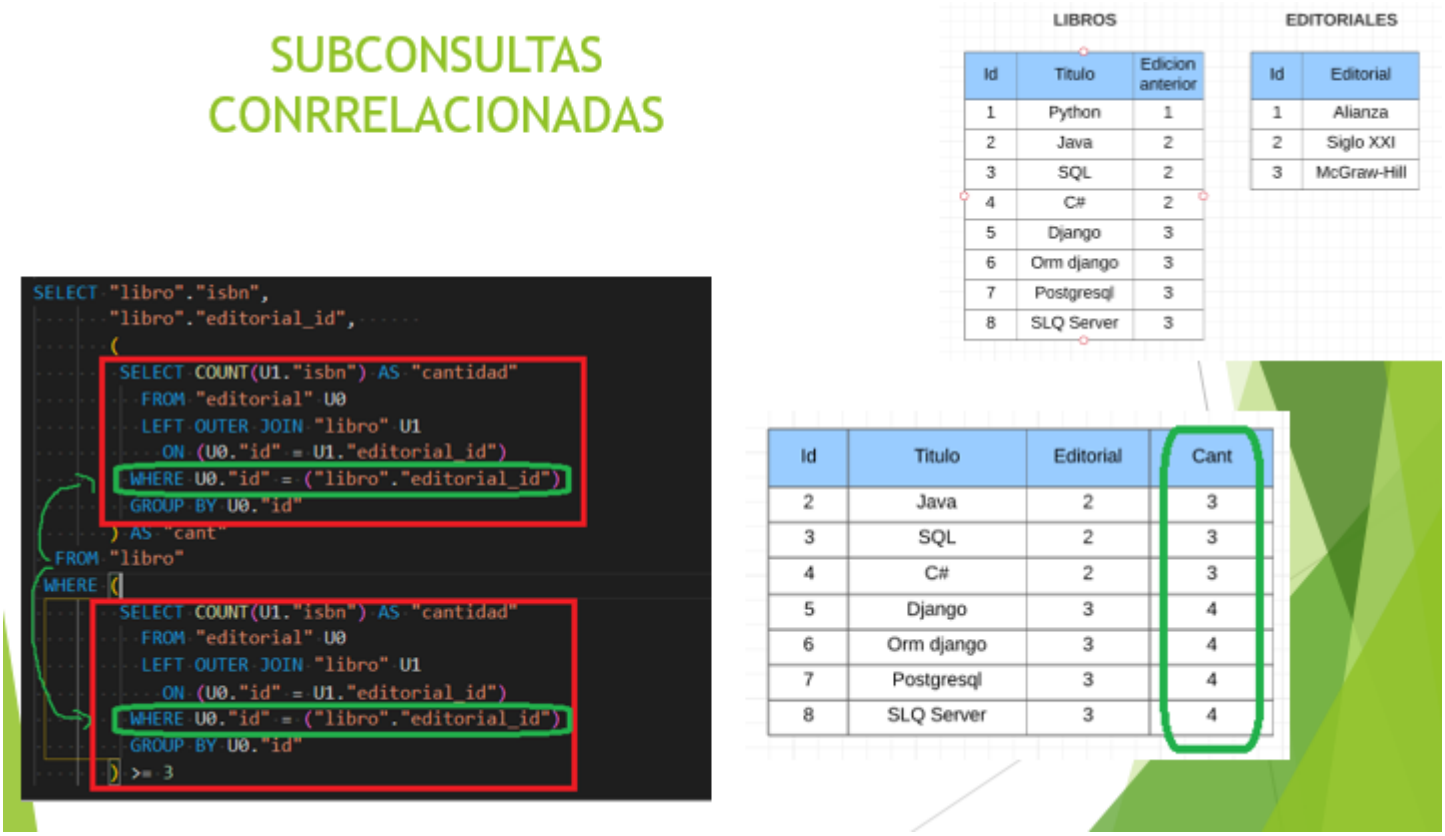
Y nuestra consulta utilizando el orm de django quedaría de la siguiente manera:

```
from django.db.models import Count
from libreria.models.editoriales import Editorial
from libreria.models.libros import Libro

# Esta es la consulta interna
editoriales = Editorial.objects.values('id').annotate(
    cantidad=Count('libro_editorial__isbn'))
editoriales = editoriales.filter(cantidad__gte=40).values('id')

# La consulta principal tiene dentro de ella consulta de arriba
libros = Libro.objects.filter(editorial__in=editoriales)
libros = libros.values('isbn', 'titulo', 'editorial')
libros
```

7.5.2 Subconsultas correlacionadas

Este tipo de consultas se llaman correlacionadas, porque su consulta interna requiere de algún dato de la consulta principal.

En el siguiente ejemplo podemos utilizar este tipo de consultas para generar el valor de una columna mas, en este caso el total de libros publicados de la editorial del cada libro, por lo que la consulta principal necesita pasarle el id de la editorial a la consulta interna, además también en el ejemplo utilizamos la consulta correlacionada para poder filtrar los datos.

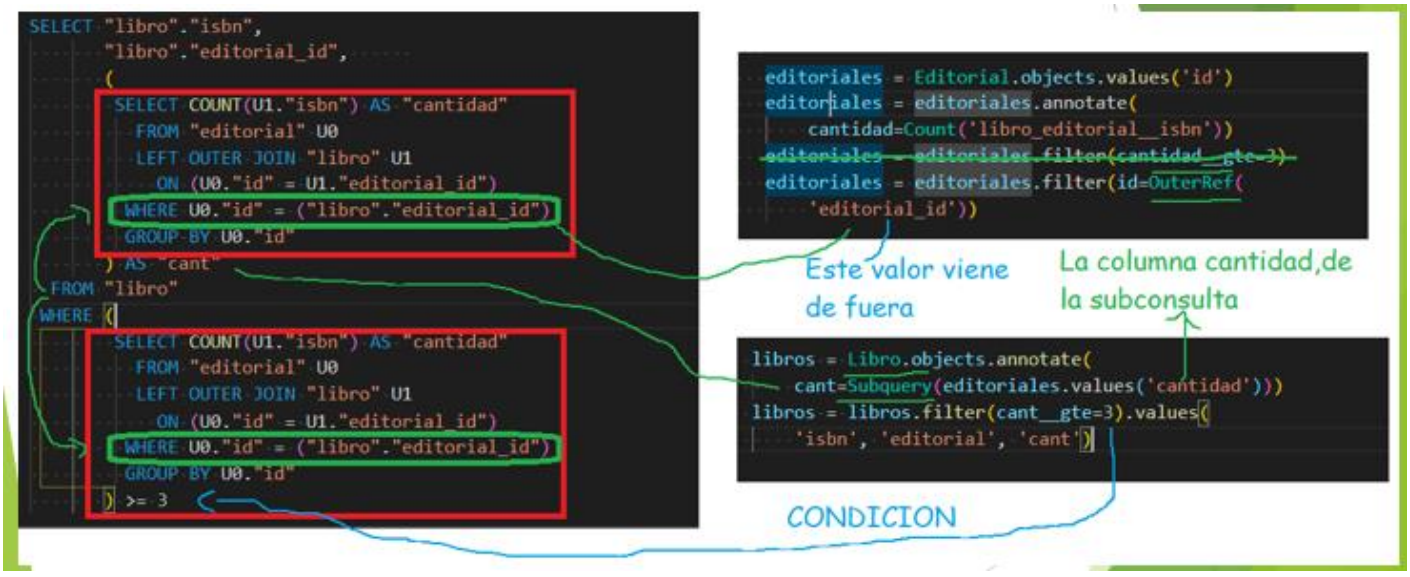


Id	Titulo	Edicion anterior
1	Python	1
2	Java	2
3	SQL	2
4	C#	2
5	Django	3
6	Orm django	3
7	Postgresql	3
8	SLQ Server	3

Id	Editorial
1	Alianza
2	Siglo XXI
3	McGraw-Hill

Id	Titulo	Editorial	Cant
2	Java	2	3
3	SQL	2	3
4	C#	2	3
5	Django	3	4
6	Orm django	3	4
7	Postgresql	3	4
8	SLQ Server	3	4

Para transformar nuestra consulta sql al orm de django necesitamos utilizar **OuterRef y Subquery** las cuales se encuentran dentro de **django.db.models.expressions**



La consulta completa utilizando el orm de django quedaria asi:

```
from django.db.models import Count, IntegerField, FloatField, ExpressionWrapper, Sum, Min,
Max, Avg
```

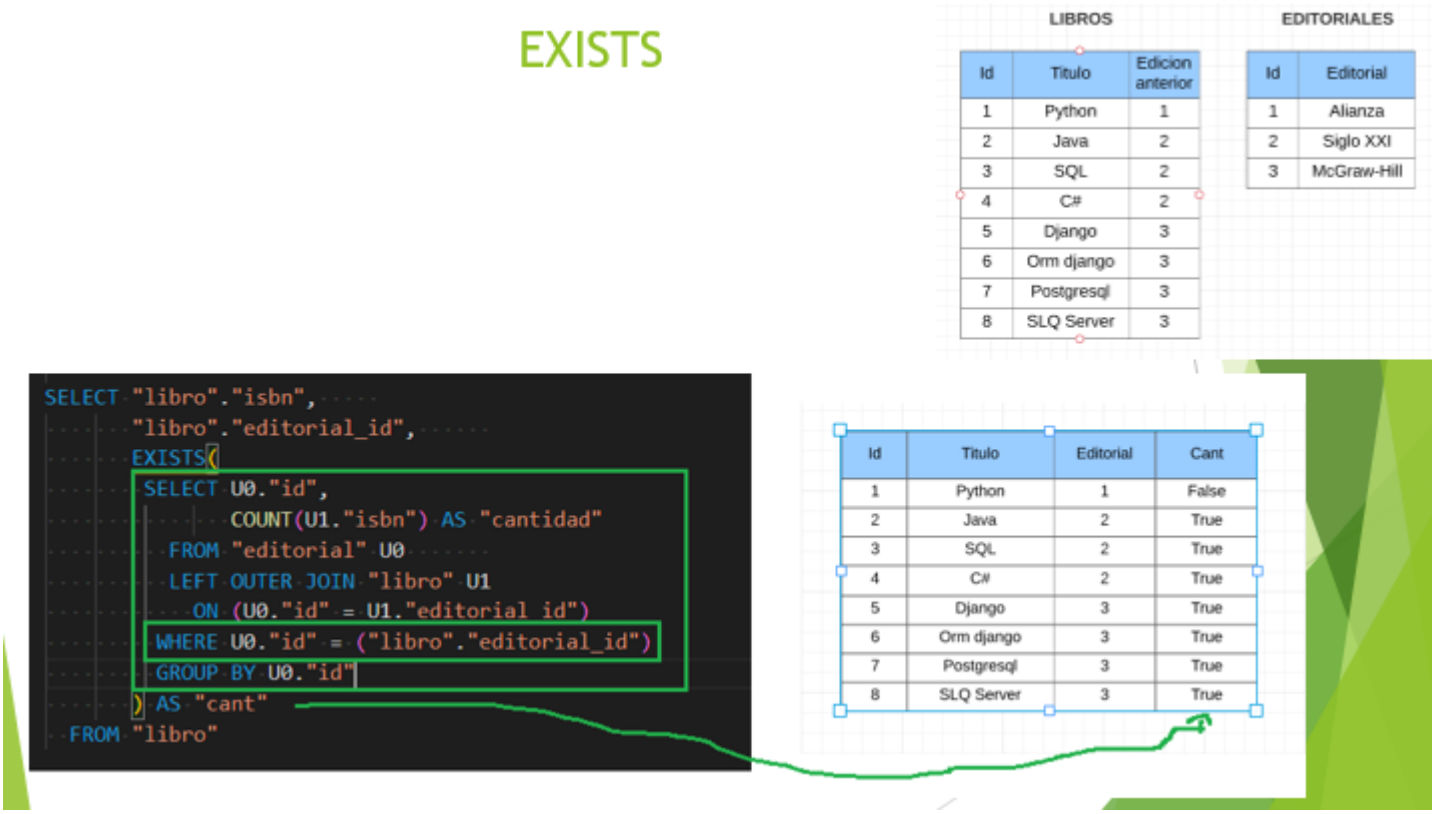
```
from django.db.models.expressions import OuterRef, Exists, OuterRef, Subquery, F
from libreria.models.autores import Autor
from libreria.models.editoriales import Editorial
from libreria.models.libros import Libro

# Nuestra subconsulta correlacionada interna
# Esta sera una nueva columna
editoriales = Editorial.objects.values('id')
editoriales = editoriales.annotate(
    cantidad=Count('libro_editorial__isbn'))
editoriales = editoriales.filter(id=OuterRef(
    'editorial_id'))
# Podemos agregar mas columnas, en este caso
# agregamos el numero de autores que escribieron el libro
num_autores = Autor.objects.values('libro__isbn').annotate(
    num_aut=Count('libro__isbn')).filter(
        libro__isbn=OuterRef('isbn')).values('num_aut')

# y nuestra consulta principal tiene una nueva columna de las editoriales
libros = Libro.objects.annotate(
    cant=Subquery(editoriales.values('cantidad'), output_field=IntegerField()))
# y tiene otra columna del numero de autores
libros = libros.annotate(num_aut=Subquery(num_autores))
# podemos utilizar nuestra subconsulta para filtra
# en este caso usamos la columna que cuenta los libros de las editoriales
libros = libros.filter(cant__gte=40).values(
    'isbn', 'editorial', 'cant', 'num_aut')
```

Y su sentencia sql, luce un poco mas aterradora, tengo que admitirlo.

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."editorial_id",
       (
        SELECT COUNT(U1."isbn") AS "cantidad"
          FROM "libreria_editorial" U0
          LEFT OUTER JOIN "libreria_libro" U1
            ON (U0."id" = U1."editorial_id")
         WHERE U0."id" = ("libreria_libro"."editorial_id")
         GROUP BY U0."id"
       ) AS "cant",
       (
        SELECT COUNT(U1."libro_id") AS "num_aut"
          FROM "libreria_autor" U0
          LEFT OUTER JOIN "libreria_autorcapitulo" U1
            ON (U0."id" = U1."autor_id")
         INNER JOIN "libreria_autorcapitulo" U3
            ON (U0."id" = U3."autor_id")
         WHERE U3."libro_id" = ("libreria_libro"."isbn")
         GROUP BY U1."libro_id"
       ) AS "num_aut"
  FROM "libreria_libro"
 WHERE (
        SELECT COUNT(U1."isbn") AS "cantidad"
          FROM "libreria_editorial" U0
          LEFT OUTER JOIN "libreria_libro" U1
            ON (U0."id" = U1."editorial_id")
         WHERE U0."id" = ("libreria_libro"."editorial_id")
         GROUP BY U0."id"
       ) >= 40
```

7.5.3 Consultas EXITS

Este tipo de consultas solo van a evaluar si la consulta interna cumple o no con la condición, por lo que pueden resultar mas rápidas.



y asi luciría utilizando el orm:

```
editoriales = Editorial.objects.values('id')
editoriales = editoriales.filter(id=OuterRef(
    'editorial_id'))
editoriales = editoriales.annotate(
    cantidad=Count('libro_editorial__isbn'))
editoriales = editoriales.filter(cantidad__gte=40)
libros = Libro.objects.annotate(
    cant=Exists(editoriales)).filter(cant=True)  # ~Exists
libros = libros.values('isbn', 'editorial', 'cant')
```

7.5.4 Combinando exists y subqueries.

Incluso puedes combinar las consultas exists y los subqueries, te dejo un ejemplo de esto junto con su sentencia sql.

```
editoriales = Editorial.objects.values('id')
editoriales = editoriales.filter(id=OuterRef(
    'editorial_id'))
editoriales = editoriales.annotate(
    cantidad=Count('libro_editorial__isbn'))

editoriales = editoriales.filter(cantidad__gte=40)
libros = Libro.objects.annotate(
    cant=Exists(editoriales)).annotate(
    cant_subq=Subquery(editoriales.values('cantidad'), output_field=IntegerField()))
libros = libros.filter(cant=True)
libros = libros.values('isbn', 'editorial', 'cant_subq')
```

SQL 

```
SELECT "libreria_libro"."isbn",
       "libreria_libro"."editorial_id",
       (
        SELECT COUNT(U1."isbn") AS "cantidad"
          FROM "libreria_editorial" U0
          LEFT OUTER JOIN "libreria_libro" U1
            ON (U0."id" = U1."editorial_id")
         WHERE U0."id" = ("libreria_libro"."editorial_id")
         GROUP BY U0."id"
        HAVING COUNT(U1."isbn") >= 40
       ) AS "cant_subq"
  FROM "libreria_libro"
 WHERE EXISTS(
        SELECT U0."id",
               COUNT(U1."isbn") AS "cantidad"
          FROM "libreria_editorial" U0
          LEFT OUTER JOIN "libreria_libro" U1
            ON (U0."id" = U1."editorial_id")
         WHERE U0."id" = ("libreria_libro"."editorial_id")
```