Análisis Lab 3 Bases de datos
Roberto Nájera

1. ¿Por qué eligieron ese ORM y qué beneficios o dificultades encontraron?
Se optó por el ORM de Django porque viene integrado, facilita la definición declarativa de esquemas y gestiona migraciones. Los beneficios son el mapeo automático entre clases Python y tablas SQL, generación y aplicación de migraciones sin escribir DDL manual, validaciones a nivel de modelo y abundantes helpers. Las dificultades fueron que el manejo de relaciones muchos-a-muchos con datos extra requiere usar through e formsets inline, lo cual añade complejidad, y que los campos EnumField necesitan un workaround (con ChoiceField) para mostrarse bien en formularios.

2. ¿Cómo implementaron la lógica master-detail dentro del mismo formulario?
Usamos un ModelForm para el Product (maestro) y un InlineFormSet para los ProductTag (detalle). En la vista, al POST, se valida el formulario principal (commit=False), se instancia el formset con esa misma instancia "pendiente", se guardan primero el master (.save()), luego el detalle (formset.save()), asegurando las FK.

3. ¿Qué validaciones implementaron en la base de datos y cuáles en el código?

- NOT NULL y UNIQUE en campos críticos (name, sku, title).
- UniqueConstraint(product, tag) para evitar duplicados en la intermedia.
- CheckConstraint en relevance para forzar un rango 1–10.
- ENUMs nativos para status y category.
- ModelForm y ChoiceField para controlar formatos y opciones válidas.
- min_num=1, validate_min=True en el formset para exigir al menos una etiqueta.
- Manejo de errores de formularios y re-renderizado con mensajes amigables.

4. ¿Qué beneficios encontraron al usar tipos de datos personalizados?
Al definir ENUMs (StatusChoices, CategoryChoices), el dominio de valores queda explícito. Hay más seguridad porque Postgres impide valores fuera de ese conjunto, reduciendo datos corruptos. Tambien en DB y en código queda claro qué estados/categorías están permitidos.

5. ¿Qué ventajas ofrece usar una VIEW como base del índice en vez de una consulta directa?
Da simplicidad en la ui porque el frontend hace un unió select a la tabla del índice, además que cualquier cambio en la lógica se hace en la vista de BD sin tocar Python, además que se puede optimizar la vista para evitar hacer múltiples joins.

6. ¿Qué escenarios podrían romper la lógica actual si no existieran las restricciones?

- Sin la UniqueConstraint(product, tag), un mismo tag podría duplicarse varias veces para un producto.
- Si no se validara relevance con CHECK, podrían guardarse valores fuera de rango, alterando cálculos o visualizaciones.
- Sin ENUMs, un typo en “publsihed” entraría sin control.

7. ¿Qué aprendieron sobre la separación entre lógica de aplicación y lógica de persistencia?

- La lógica de negocio vive en Django.
- La lógica de datos vive en Postgres.
- Mantenerlas separadas hace cada capa más mantenible y especializada.

8. ¿Cómo escalaría este diseño en una base de datos de gran tamaño?

- Convertir la VIEW en MATERIALIZED VIEW si los datos cambian con poca frecuencia, mejorando consultas de índice.
- Añadir índices en columnas usadas en filtros/joins (product_id, tag_id).
- Paginar los resultados en la UI para no traer miles de filas de golpe.

9. ¿Consideran que este diseño es adecuado para una arquitectura con microservicios?
Cada servicio podría exponer su propia View sin sobrecargar otros dominios, lo cual es algo bueno, pero la view directa en la BD comparte el mismo esquema, entonces convendría exponer un API de agregación en lugar de un SELECT directo.

10. ¿Cómo reutilizarían la vista en otros contextos como reportes o APIs?
Para reportes internos, materializar la VIEW y generar ETL nocturno. Para API REST, Crear un endpoint DRF que consulte ProductIndex.objects.all().

11. ¿Qué decisiones tomaron para estructurar su modelo de datos y por qué?
- Definir tabla intermedia explícita (ProductTag) para poder añadir el campo extra relevance.
- Mantener Tag como entidad separada para promover la reutilización y evitar duplicados de texto.
- Usar ENUMs para campos de estado y categoría, haciendo el modelo más robusto.

12. ¿Cómo documentaron su modelo para facilitar su comprensión por otros desarrolladores?

Dejamos un README con toda la información relevante para el proyecto y con todas las explicaciones necesarias

13. ¿Cómo evitaron la duplicación de registros o errores de asignación en la tabla intermedia?

- Con la UniqueConstraint (product, tag) a nivel BD.
- Validando en el formset que no se repitiera un mismo tag (Django lanza error al colisionar).
- Ocultando correctamente el checkbox de DELETE y procesándolo en el formset para actualizaciones.