Optimización de Consultas a una RDBMS

El diseño eficiente de bases de datos relacionales es fundamental para garantizar un rendimiento óptimo en aplicaciones. Esta presentación aborda las consideraciones clave en el diseño, uso de claves, optimización de índices, eficiencia en consultas y normalización de datos en sistemas de gestión de bases de datos relacionales.

Exploraremos desde los fundamentos del diseño hasta técnicas avanzadas de optimización, con ejemplos prácticos en PostgreSQL que te permitirán implementar estas estrategias en tus propios proyectos.

por Kibernum Capacitación

Preguntas de Activación

Claves primarias y foráneas

¿Por qué crees que es importante definir correctamente las claves primarias y foráneas en una base de datos?

Tablas grandes y rendimiento

¿Has trabajado alguna vez con tablas grandes? ¿Cómo afectaba el rendimiento de tus consultas?

Uso de vistas

¿Qué opinas sobre el uso de vistas? ¿Crees que pueden facilitar el trabajo de usuarios o desarrolladores?

Consideraciones al Momento del Diseño

Análisis del dominio

Entender lo que se quiere representar (personas, ventas, productos, etc.) es el primer paso para un diseño efectivo.

Identificación de entidades

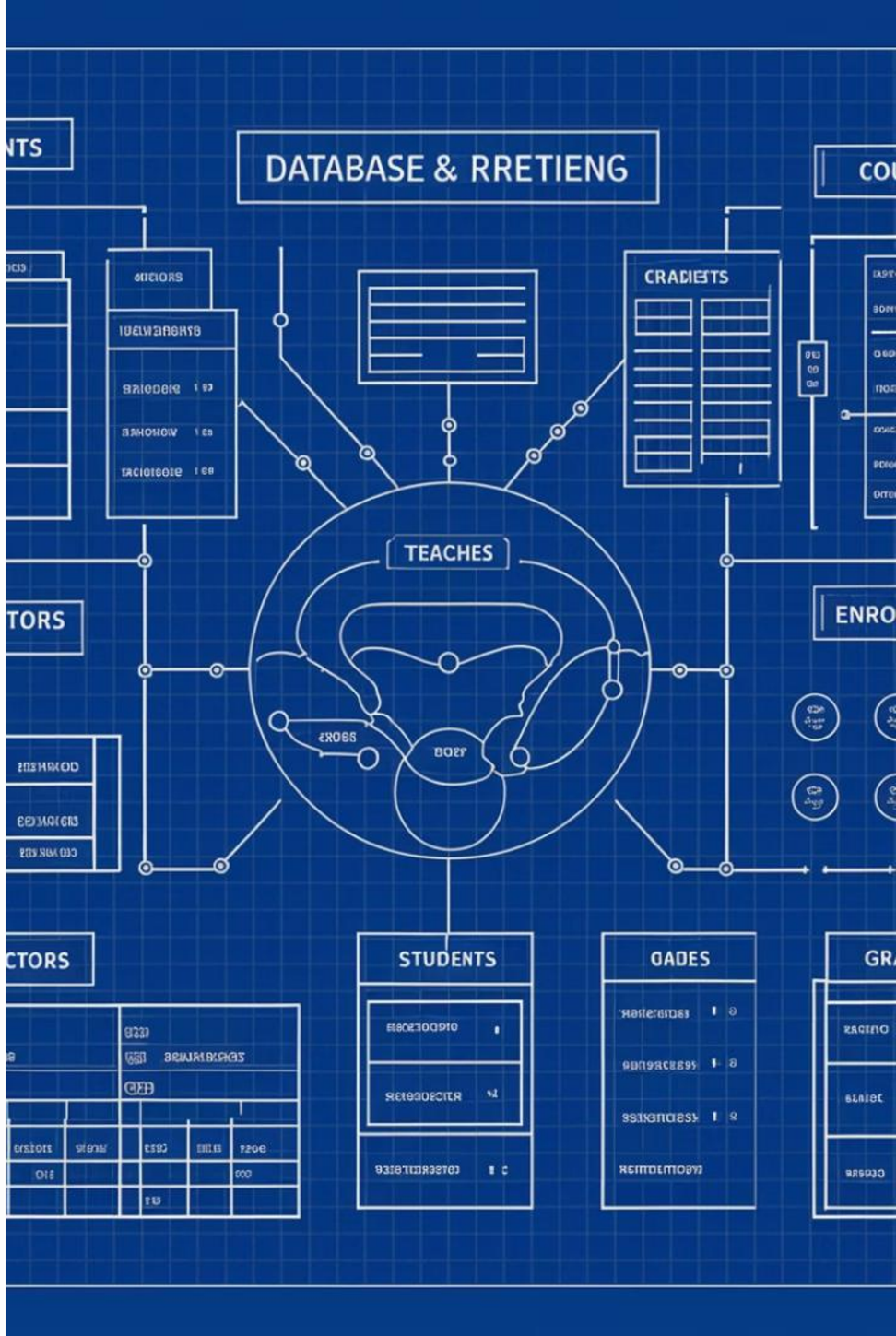
Definir las cosas importantes del sistema y cómo interactúan entre ellas establece la estructura básica.

Normalización

Aplicar reglas para evitar redundancia y mejorar integridad de los datos es crucial para la eficiencia.

Elección de tipos y restricciones

Seleccionar los tipos de datos adecuados y establecer restricciones apropiadas garantiza la integridad.



Problemas de un Mal Diseño

Duplicidad de datos

La información repetida innecesariamente ocupa espacio adicional y crea problemas de consistencia cuando se actualiza solo una parte de los datos duplicados.

Dificultad para escalar

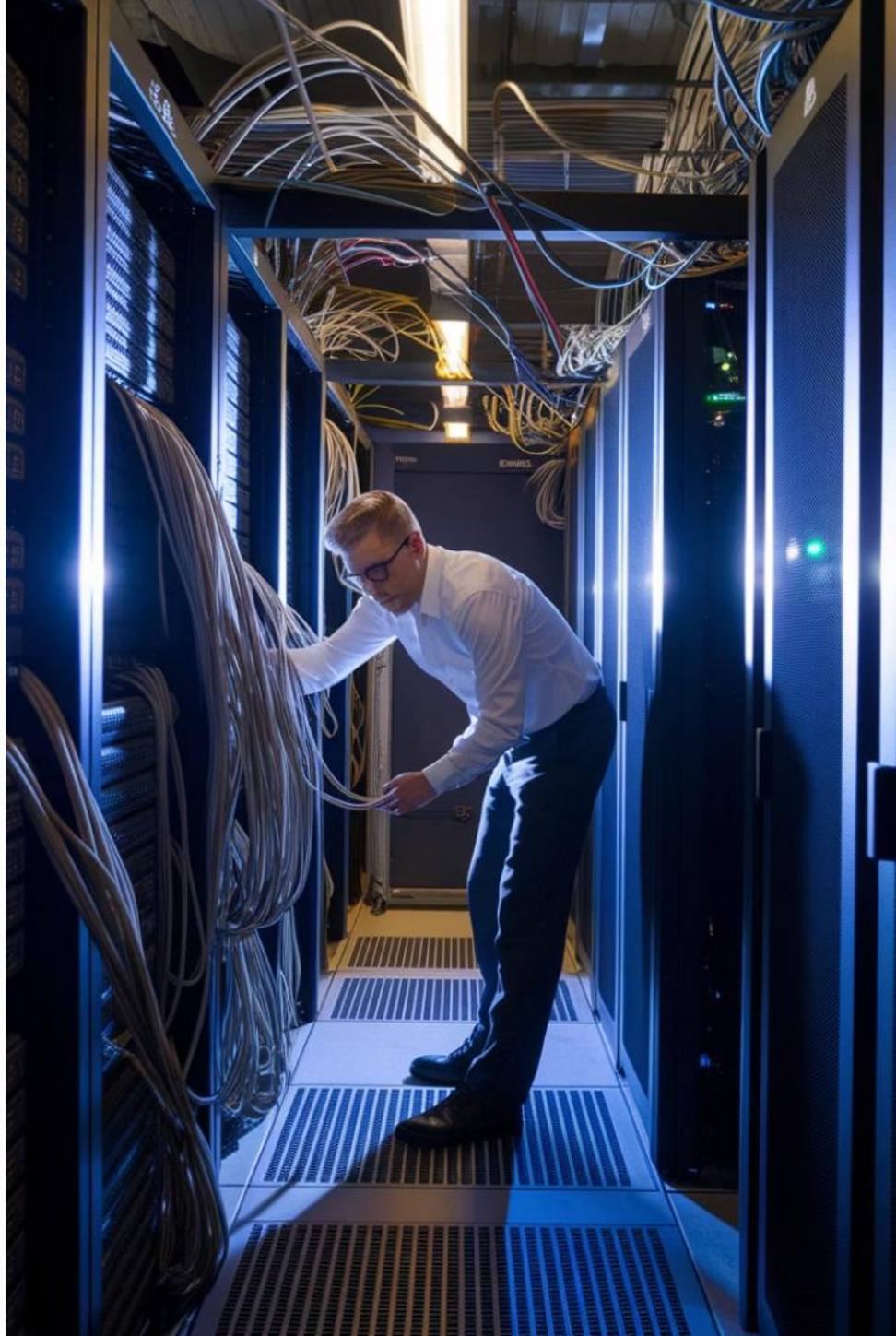
Un diseño deficiente limita la capacidad de crecimiento del sistema, generando cuellos de botella a medida que aumenta el volumen de datos o transacciones.

Inconsistencias

Sin las restricciones adecuadas, los datos pueden volverse inconsistentes, comprometiendo la integridad y fiabilidad de la información almacenada.

Baja eficiencia en consultas

Las consultas se vuelven lentas y consumen más recursos del servidor, afectando el rendimiento general de la aplicación.



Claves Primarias (Primary Keys)

Características esenciales

- No puede ser NULL
- Debe ser única
- Generalmente es un número incremental (SERIAL, BIGSERIAL) o un UUID

La clave primaria identifica de forma única cada fila en una tabla. Su existencia asegura que no haya duplicados y sirve como referencia en otras tablas.

En este ejemplo, se define una clave primaria para la tabla clientes, garantizando que cada cliente tenga un identificador único que no puede ser nulo.

Ejemplo en PostgreSQL

```
CREATE TABLE clientes (
  id_cliente SERIAL PRIMARY KEY,
  nombre TEXT NOT NULL
);
```

Claves Foráneas (Foreign Keys)

Definición

Las claves foráneas conectan registros de diferentes tablas, estableciendo relaciones entre ellas.

Función

Permiten asegurar que los datos referenciados realmente existen en la tabla relacionada.

Integridad

Mantienen la integridad referencial con acciones como ON DELETE CASCADE o SET NULL.

```
CREATE TABLE ordenes (
  id_orden SERIAL PRIMARY KEY,
  id_cliente INT REFERENCES clientes(id_cliente)

);
```

Uso y Optimización de Índices

Definición de índices

Estructuras especiales que permiten acelerar el acceso a los datos

Beneficios

Mejoran el rendimiento de las consultas evitando escaneos secuenciales

Consideraciones

Un exceso de índices puede degradar inserciones, actualizaciones y borrados

Creación de Índices

Índice simple

```
CREATE INDEX idx_nombre_cliente ON clientes(nombre);
```

Índice único

```
CREATE UNIQUE INDEX idx_correo_cliente ON clientes(correo);
```

Índice compuesto

```
CREATE INDEX idx_pedidos_cliente_fecha ON pedidos(id_cliente, fecha_pedido);
```

Índice parcial

```
CREATE INDEX idx_pedidos_grandes ON pedidos(total) WHERE total > 1000;
```

Optimización del Esquema

Tipos de datos adecuados

Seleccionar el tipo más eficiente para cada columna

Evitar redundancia

Aplicar normalización para eliminar duplicidad

Monitoreo continuo

Evaluar y ajustar según el rendimiento

Restricciones correctas

Implementar CHECK, NOT NULL y UNIQUE

Uso de Vistas

Vistas Estándar

Las vistas son consultas predefinidas que pueden usarse como si fueran tablas. Son ideales para simplificar consultas complejas o restringir el acceso a ciertos datos.

Ejemplo:

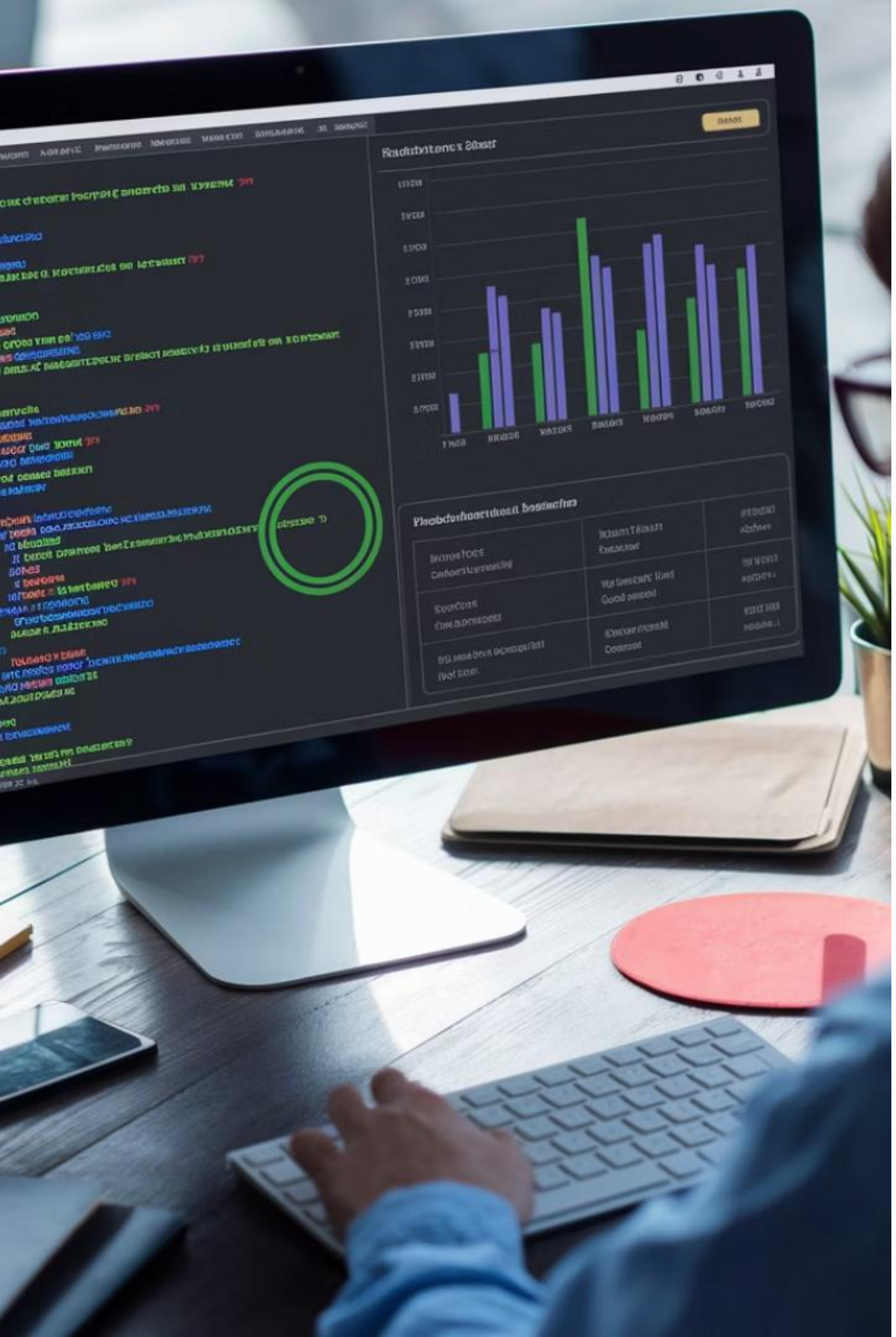
```
CREATE VIEW resumen_ordenes AS
SELECT c.nombre, COUNT(o.id_orden) AS total_ordenes
FROM clientes c
JOIN ordenes o ON c.id_cliente = o.id_cliente
GROUP BY c.nombre;
```

Vistas Materializadas

A diferencia de las vistas normales, las materializadas almacenan físicamente los datos. Mejoran el rendimiento en informes, dashboards y consultas frecuentes.

Ejemplo:

```
CREATE MATERIALIZED VIEW productos_populares AS
SELECT id_producto, COUNT(*) AS total_ventas
FROM orden_detalle
GROUP BY id_producto;
```



Eficiencia en la Ejecución de Consultas

Evita SELECT *

Selecciona solo las columnas necesarias para reducir la transferencia de datos.

Filtra con WHERE bien definido

Usa condiciones específicas que puedan aprovechar los índices existentes.

Usa JOIN en lugar de subconsultas

Los JOIN suelen ser más eficientes que las subconsultas en muchos casos.

Aplica agregaciones correctamente

Utiliza funciones como SUM, COUNT y AVG de manera óptima.

Análisis del Plan de Ejecución

Utilizar EXPLAIN

```
EXPLAIN ANALYZE
SELECT * FROM ordenes WHERE fecha > NOW() - INTERVAL '1 month';
```

Este comando muestra cómo PostgreSQL ejecutará la consulta sin realmente ejecutarla.

Interpretar los resultados

Identificar si se está usando Seq Scan (escaneo completo) o Index Scan (uso de índice), así como el número estimado de filas y costos.

Optimizar según el análisis

Crear índices o modificar la consulta basándose en la información del plan de ejecución para mejorar el rendimiento.

Primera Forma Normal (1NF)

Definición

Una tabla está en 1NF si cada celda tiene un solo valor (atómico) y no existen grupos repetidos ni listas separadas por comas.

Esta forma normal elimina los grupos repetitivos y asegura que cada campo contenga solo un valor, estableciendo la base para un diseño relacional adecuado.

Ejemplo de violación de 1NF

```
CREATE TABLE Estudiantes (
    id_estudiante SERIAL PRIMARY KEY,
    nombre VARCHAR(50),
    cursos TEXT -- ❌ contiene lista
);
```

En este ejemplo, la columna "cursos" contiene múltiples valores separados por comas, lo que viola el principio de atomicidad de la 1NF.

Solución en 1NF – Tablas Separadas

```
CREATE TABLE Estudiantes (
    id_estudiante SERIAL PRIMARY KEY,
    nombre VARCHAR(50)
);

CREATE TABLE Cursos (
    cod_curso VARCHAR(10) PRIMARY KEY,
    nombre VARCHAR(50),
    profesor VARCHAR(50)
);

CREATE TABLE EstudiantesCursos (
    id_estudiante INT,
    cod_curso VARCHAR(10),
    FOREIGN KEY (id_estudiante) REFERENCES Estudiantes(id_estudiante),
    FOREIGN KEY (cod_curso) REFERENCES Cursos(cod_curso)
);
```

Solución en Primera Forma Normal

Tablas Separadas

Para cumplir con la 1NF, se divide la información en tablas relacionadas. Se crea una tabla para estudiantes, otra para cursos, y una tabla intermedia que relaciona estudiantes con sus cursos.

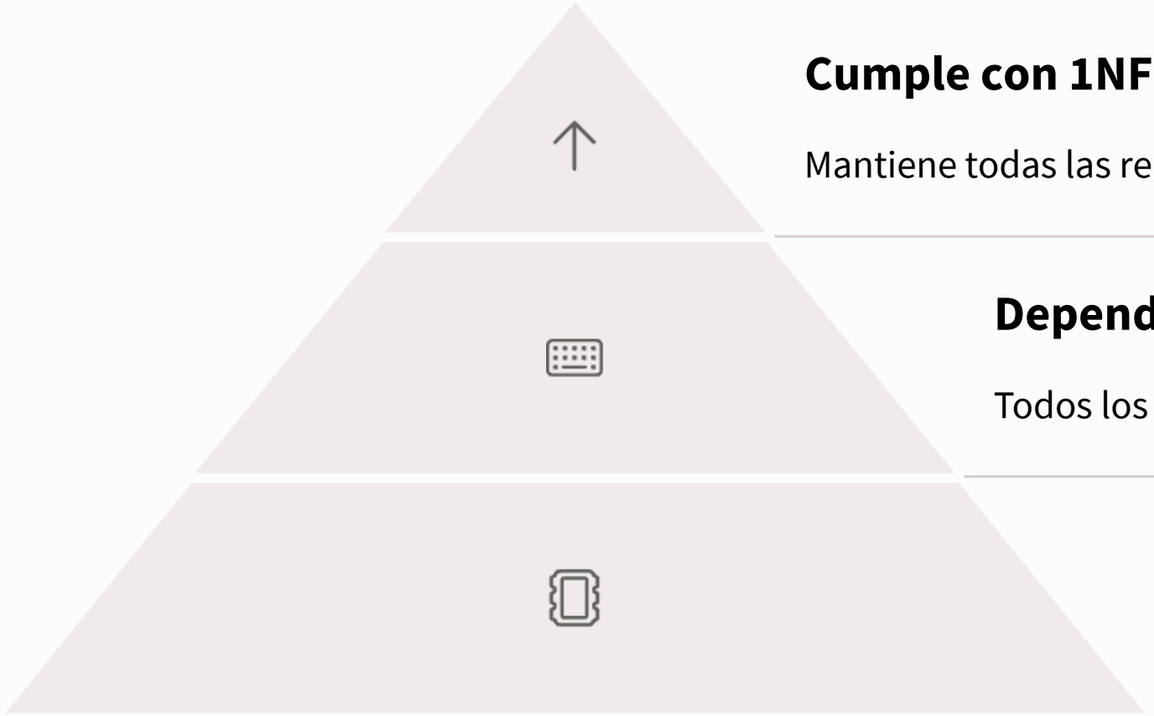
Estructura Resultante

La estructura normalizada permite almacenar los datos de manera más eficiente, eliminando la redundancia y facilitando las consultas y actualizaciones de la información.

Beneficios

Esta estructura facilita la búsqueda, actualización y eliminación de datos, además de permitir relaciones más complejas entre las entidades del sistema.

Segunda Forma Normal (2NF)



Cumple con 1NF

Mantiene todas las reglas de la Primera Forma Normal

Dependencia de clave completa

Todos los atributos no clave dependen de toda la clave primaria

Elimina dependencias parciales

Separa atributos que dependen solo de parte de la clave

Segunda Forma Normal (2NF)

Error común en 2NF

```sql
-- Aquí el profesor depende solo del curso
CREATE TABLE EstudiantesCursos (
    id_estudiante INT,
    cod_curso VARCHAR(10),
    profesor VARCHAR(50) -- ❌ dependencia parcial
);
```

Solución en 2NF – profesor pasa a la tabla Cursos

```sql
-- Ya creada en 1NF
CREATE TABLE Cursos (
    cod_curso VARCHAR(10) PRIMARY KEY,
    nombre VARCHAR(50),
    profesor VARCHAR(50)
);
```

Tercera Forma Normal (3NF)

Definición

Una tabla está en 3NF si cumple con 2NF y no existen dependencias transitivas (un campo no clave no depende de otro campo no clave).

Esta forma normal elimina las dependencias transitivas, mejorando aún más la estructura de la base de datos y reduciendo la redundancia.

Ejemplo con dependencia transitiva

```sql
-- El departamento depende del profesor, no directamente del curso
CREATE TABLE Cursos (
    cod_curso VARCHAR(10),
    nombre VARCHAR(50),
    profesor VARCHAR(50),
    departamento VARCHAR(50) -- ✗
);
```

En este ejemplo, el departamento depende del profesor, y el profesor depende de la clave primaria, creando una dependencia transitiva que debe eliminarse.

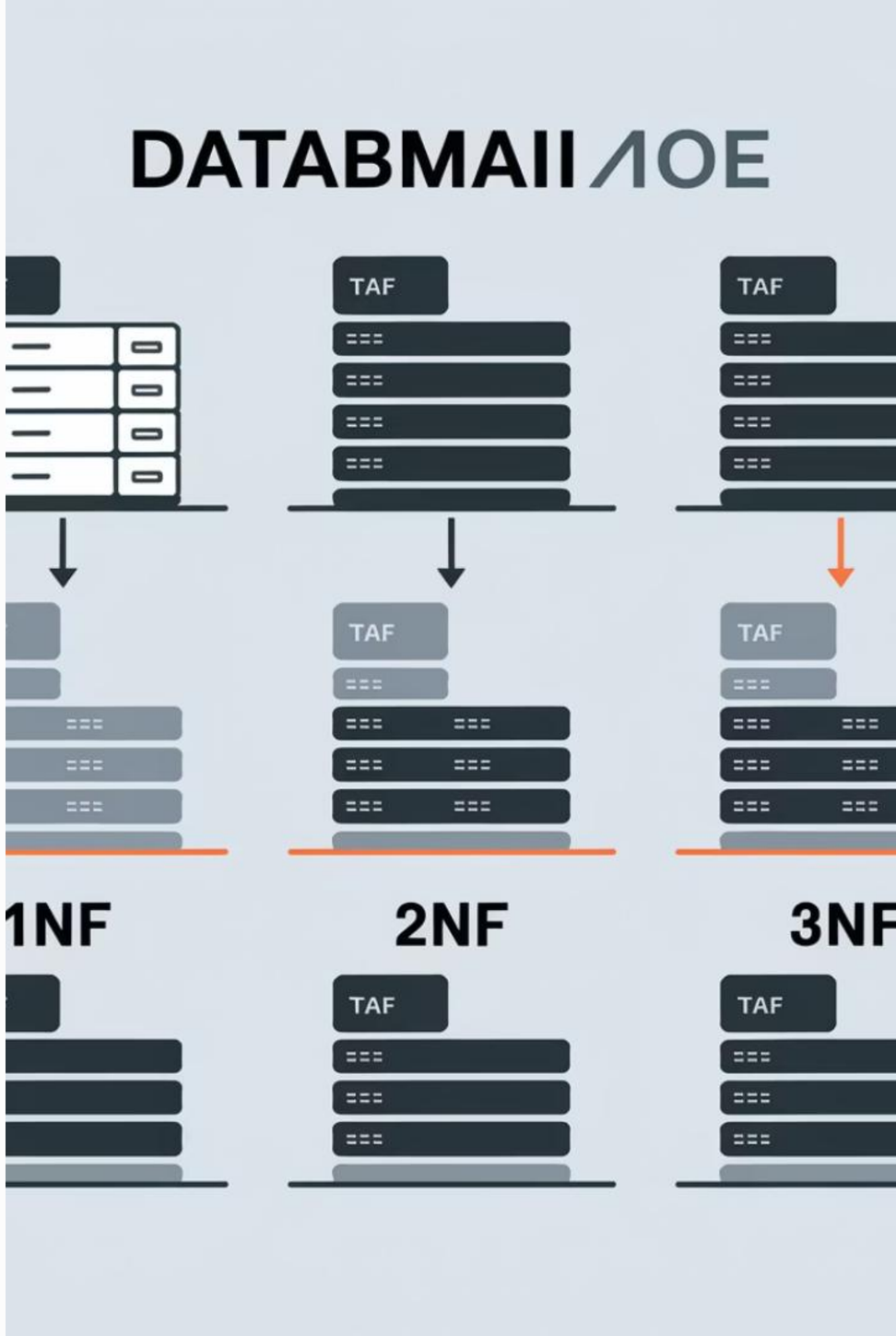
Solución en 3NF – Separar tabla Profesores

```sql
CREATE TABLE Profesores (
    nombre VARCHAR(50) PRIMARY KEY,
    departamento VARCHAR(50)
);

ALTER TABLE Cursos
ADD CONSTRAINT fk_profesor FOREIGN KEY (profesor)
REFERENCES Profesores(nombre);
```

Resumen de Formas Normales

Forma	Qué soluciona	Cómo se aplica
1NF	Datos agrupados / múltiples valores	Celdas atómicas y tabla intermedia
2NF	Atributos que dependen solo de parte de la clave	Separar a otras tablas (como profesor)
3NF	Atributos que dependen de otro atributo no clave	Separar jerarquías como profesor ↔ depto.



Actividad Práctica Guiada: Diseño y Optimización en PostgreSQL

Objetivo:

Aplicar los conceptos de diseño relacional y optimización en PostgreSQL, mediante la creación de un esquema con claves primarias y foráneas, índices, y el uso de vistas, incluyendo la evaluación del rendimiento de una consulta con EXPLAIN.

Paso a Paso Detallado:

1 Crear el esquema de base de datos

Desde tu editor de código o PgAdmin, crea las siguientes tablas:

```sql
CREATE TABLE clientes (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(50),
    correo VARCHAR(100) UNIQUE
);

CREATE TABLE pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INT REFERENCES clientes(id) ON DELETE CASCADE,
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total NUMERIC(10,2) CHECK (total >= 0)
);
```

Actividad Práctica Guiada: Diseño y Optimización en PostgreSQL

2 Insertar registros de prueba

```sql
INSERT INTO clientes (nombre, correo)
VALUES ('Ana', 'ana@mail.com'),
       ('Luis', 'luis@mail.com');

INSERT INTO pedidos (cliente_id, total)
VALUES (1, 150.00), (1, 200.00), (2, 300.00);
```

3 Crear un índice en la columna total de la tabla pedidos

```sql
CREATE INDEX idx_total ON pedidos(total);
```

Esto mejorará las búsquedas por total.

Actividad Práctica Guiada: Diseño y Optimización en PostgreSQL

4 Crear una vista para consultar pedidos con nombre del cliente

```sql
CREATE VIEW vista_pedidos AS
SELECT c.nombre, p.total, p.fecha
FROM clientes c
JOIN pedidos p ON c.id = p.cliente_id;
```

5 Analizar una consulta con EXPLAIN

```sql
EXPLAIN SELECT * FROM pedidos WHERE total > 100;
```

Observa si el plan usa Index Scan.

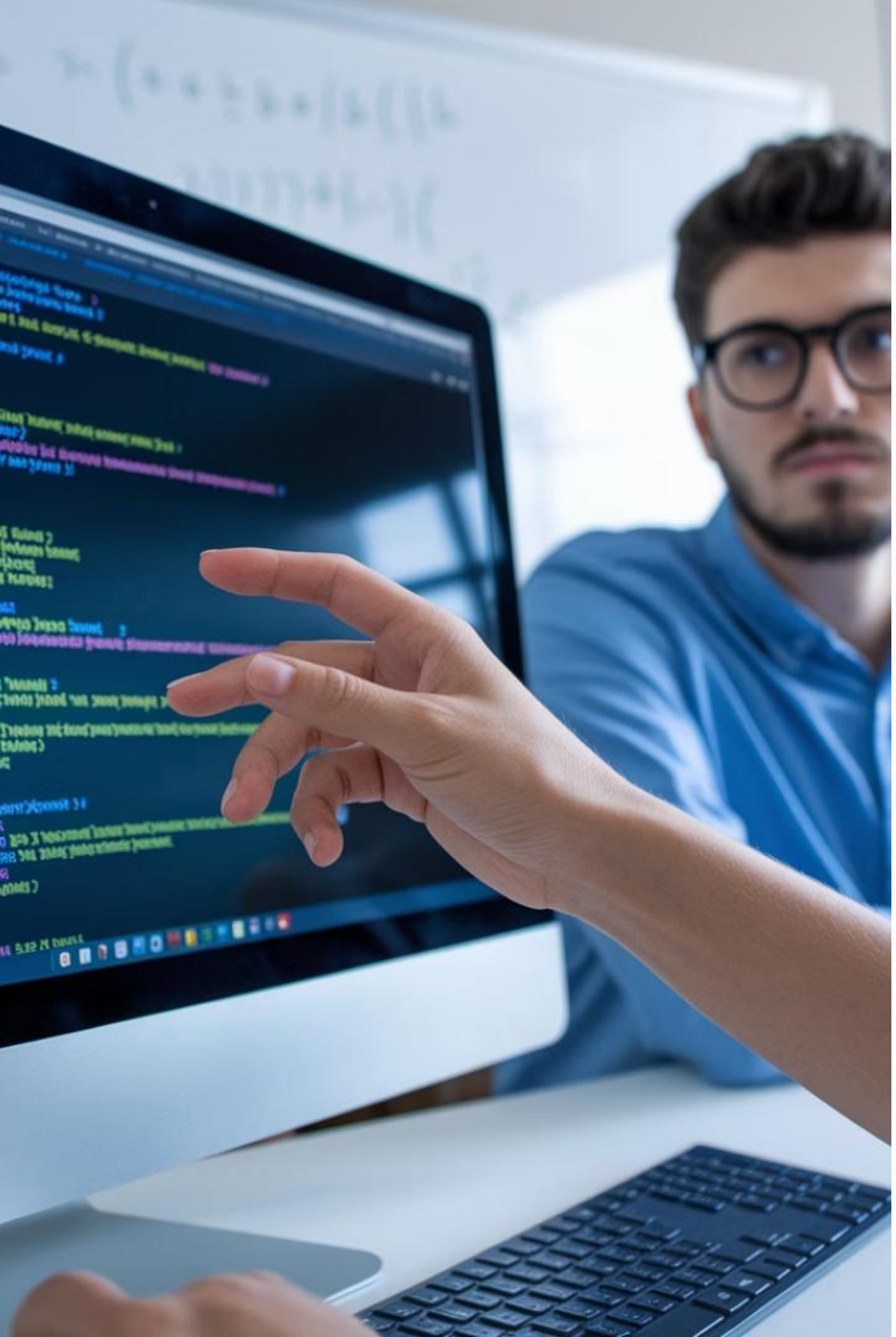
Reflexión en conjunto:

- Qué aprendiste sobre claves, vistas e índices.
- Qué mejoras viste con el índice.
- Qué muestra el plan de ejecución.

Material Complementario

https://www.youtube.com/watch?v=OuJerKzV5T0

🎥 Este video en español ofrece una explicación clara y didáctica sobre claves, normalización, creación de índices y análisis de rendimiento en PostgreSQL. Incluye ejemplos ejecutados en consola, ideales para complementar lo aprendido en clase.



Reflexión sobre Estrategias y Aprendizajes

Mejora del rendimiento de consultas

¿Qué estrategias aplicaste para mejorar el rendimiento de tus consultas? ¿Funcionaron como esperabas?

Índices y claves foráneas

¿Cómo decidiste dónde aplicar índices y claves foráneas? ¿Tu decisión influyó en la eficiencia?

Análisis de plan de ejecución

¿Qué aprendiste al analizar un plan de ejecución? ¿Qué indicadores te resultaron más útiles?.