Lectura sesión Optimización de consultas a una RDBMS

Introducción

Diseñar una base de datos no es simplemente definir tablas y columnas. Es un proceso intelectual que parte del análisis del dominio del problema, la definición de entidades y sus relaciones, y continúa con decisiones técnicas que impactarán el rendimiento, la integridad y la escalabilidad del sistema.

Una base de datos mal diseñada puede ocasionar problemas como:

- Duplicidad de datos
- Dificultad para escalar
- Inconsistencias
- Baja eficiencia en consultas

Por lo tanto, una buena planificación es clave para asegurar una base sólida.

R **por Kibernum Capacitación S.A.**

Etapas clave en el diseño

1. Análisis del dominio: Entender lo que se quiere representar (personas, ventas, productos, etc.).
2. Identificación de entidades: Definir las cosas importantes del sistema.
3. Relaciones entre entidades: ¿Cómo interactúan entre ellas?
4. Normalización: Aplicar reglas para evitar redundancia y mejorar integridad.
5. Elección de tipos de datos y restricciones.

Uso de Claves en Bases de Datos

Las claves son fundamentales para garantizar la unicidad, referencialidad y la integridad dentro de una base de datos relacional. Sin claves, los registros serían ambiguos y difíciles de relacionar.

Hay dos tipos principales:

- Claves Primarias (Primary Keys)
- Claves Foráneas (Foreign Keys)

Claves Primarias (Primary Keys)

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.

Características:

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

Ejemplo en PostgreSQL:

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

Claves Foráneas (Foreign Keys)

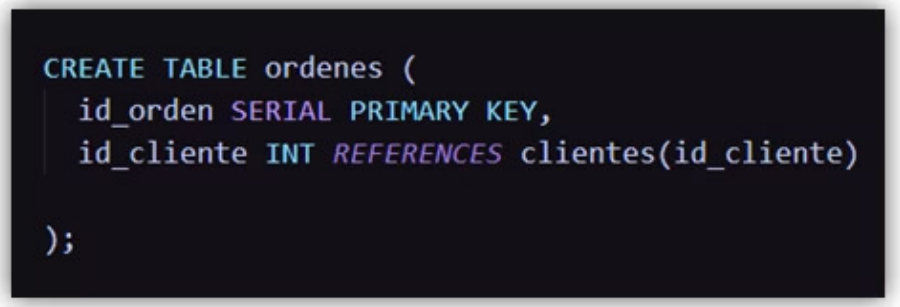
Las claves foráneas conectan registros de diferentes tablas, estableciendo relaciones entre ellas. Permiten asegurar que los datos referenciados realmente existen.

Características:

- Enlazan con una clave primaria de otra tabla.
- Pueden tener acciones asociadas: ON DELETE CASCADE, SET NULL, etc.

Ejemplo en PostgreSQL:

Esto asegura que no se pueda insertar una orden con un cliente inexistente.



Esto asegura que no se pueda insertar una orden con un cliente inexistente.

Uso y Optimización de Índices

Los índices son estructuras especiales que permiten acelerar el acceso a los datos. Sin índices, PostgreSQL tendría que escanear toda la tabla para encontrar un valor (Sequential Scan), lo cual se vuelve lento con grandes volúmenes de datos.

Un buen uso de índices mejora el rendimiento de las consultas, pero un exceso de ellos puede degradar la velocidad de inserciones, actualizaciones y borrados.

Uso de Índices

Crear un índice simple:

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

Ahora PostgreSQL puede buscar por nombre más rápido.

Índices únicos

Aseguran que no haya duplicados:

Optimización de Índices

Para optimizar su uso, considera lo siguiente:

- Crea índices solo en columnas utilizadas en búsquedas, filtros o joins.
- Usa índices compuestos si filtras por varias columnas juntas:

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

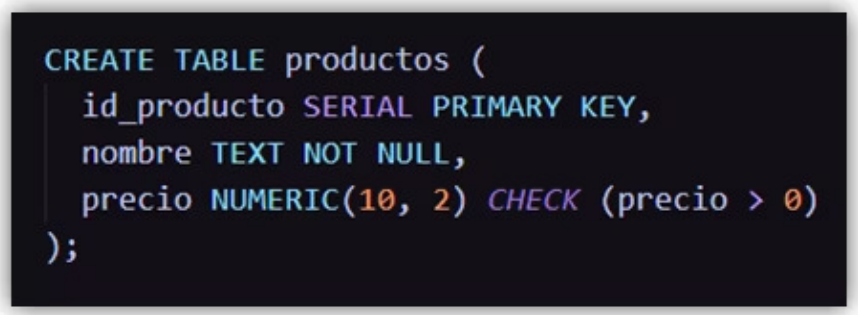
- El orden de las columnas en índices compuestos importa.
- Utiliza EXPLAIN para verificar si una consulta está usando un índice (más adelante lo vemos en profundidad).

Optimización de Bases de Datos

Optimizar una base de datos no es únicamente agregar índices. Es también una tarea que implica diseñar un esquema inteligente, aplicar restricciones correctas, y usar recursos como vistas o vistas materializadas que mejoren la eficiencia.

Optimización del Esquema

Usa tipos de datos adecuados, evita redundancia y aplica restricciones:



```
CREATE TABLE productos (
  id_producto SERIAL PRIMARY KEY,
  nombre TEXT NOT NULL,
  precio NUMERIC(10, 2) CHECK (precio > 0)
);
```

Esto garantiza que los precios sean positivos.

Uso de Vistas y Vistas Materializadas

Las vistas son consultas predefinidas que pueden usarse como si fueran tablas. Ideal para simplificar consultas o restringir el acceso.

```sql
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 rendimiento en informes, dashboards, etc.

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

Para actualizarla:

```sql
REFRESH MATERIALIZED VIEW productos_populares;
```

Eficiencia en la Ejecución de Consultas

Una consulta lenta puede ser el resultado de mala escritura SQL, falta de índices o problemas de diseño. Aprender a escribir consultas eficientes y analizar su ejecución es clave para bases de datos de alto rendimiento.

Escritura Eficiente de SQL

Buenas prácticas:

- Evita SELECT * si no necesitas todas las columnas.
- Filtra con WHERE bien definido.
- Usa JOIN en lugar de subconsultas si es más eficiente.
- Aplica agregaciones correctamente (SUM, COUNT, AVG).

Ejemplo:

```sql
SELECT p.nombre, SUM(d.cantidad)
FROM productos p
JOIN orden_detalle d ON p.id_producto = d.id_producto
GROUP BY p.nombre;
```

Análisis del Plan de Ejecución

PostgreSQL permite ver cómo se ejecuta una consulta con EXPLAIN:

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

¿Qué verás?

- Seq Scan: Escaneo completo (puede ser lento)
- Index Scan: Uso de índice (mejor)
- Rows, Cost, Time: Indicadores de eficiencia

Puedes crear un índice para mejorar la consulta:

```
CREATE INDEX idx_fecha_orden ON ordenes(fecha);
```

Tabla resumen de conceptos

Tema	Descripción breve
Clave primaria	Identifica una fila de forma única
Clave foránea	Relaciona una fila con otra en otra tabla
Índice	Acelera búsquedas
Vista	Consulta almacenada como tabla virtual
Vista materializada	Consulta almacenada físicamente
EXPLAIN	Analiza cómo se ejecuta una consulta

Formas Normales en Bases de Datos Relacionales

Primera Forma Normal (1NF)

Definición:

- Cada celda tiene un solo valor (atómico)
- No existen grupos repetidos ni listas separadas por comas

Tabla NO normalizada (violando 1NF)

id_estudiante	nombre	cursos
1	**Ana**	**MAT101, ALG202**
2	**Luis**	**ALG202, HIS301**

Solución en 1NF – Tablas Separadas

```sql
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)
);
```

Tablas visuales en 1NF

id_estudiante	nombre
1	Ana
2	Luis

Tabla cursos

cod_curso	nombre	profesor
MAT101	Matemáticas	Sra. Ana
ALG202	Álgebra	Sr. Jorge
HIS301	Historia	Sr. Pablo

Tabla EstudiantesCursos

id_estudiante	cod_curso
1	MAT101
1	ALG202
2	ALG202
2	HIS301

Segunda y Tercera Forma Normal (2NF y 3NF)

Segunda Forma Normal (2NF)

Definición:

- Cumple con 1NF
- Todos los atributos no clave dependen de toda la clave primaria (no de solo una parte)

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)
);
```

Tablas visuales en 2NF

Cursos (con profesor separado)

cod_curso	nombre	profesor
MAT101	Matemáticas	Sra. Ana
ALG202	Álgebra	Sr. Jorge
HIS301	Historia	Sr. Pablo

Tercera Forma Normal (3NF)

Definición:

- Cumple 2NF
- No existen dependencias transitivas (un campo no clave no depende de otro campo no clave)

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) -- ❌
);
```

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);
```

Tablas visuales en 3NF

Profesores

Nombre	Departamento
Sra. Ana	Ciencias
Sr. Jorge	Matemáticas
Sr. Pablo	Humanidades

Cursos (con referencia a profesor)

cod_curso	nombre	profesor
MAT101	Matemáticas	Sra. Ana
ALG202	Álgebra	Sr. Jorge
HIS301	Historia	Sr. Pablo

Resumen de las 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)
);
```

2 Insertar registros de prueba

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

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

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

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