PostgreSQL Avanzado

Documento Técnico Explicado en Profundidad

Documento técnico

14 de enero de 2026

Índice

1. Introducción

PostgreSQL es un sistema gestor de bases de datos objeto–relacional (ORDBMS) diseñado bajo tres principios fundamentales:

- **Corrección de los datos por encima del rendimiento**
- **Cumplimiento estricto del estándar SQL**
- **Extensibilidad del motor**

A diferencia de otros SGBD, PostgreSQL no delega la integridad de los datos a la aplicación. La sintaxis que veremos a lo largo del documento responde a esta filosofía: *las reglas viven en la base de datos, no en el código cliente.*

Este enfoque lo hace especialmente adecuado para arquitecturas como MVC, microservicios y sistemas distribuidos, donde múltiples aplicaciones consumen el mismo esquema de datos.

2. Creación y Gestión de Tablas

2.1. Definición básica de tablas

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT now()
);
```

Análisis profundo de la sintaxis:

- `CREATE TABLE` define una estructura persistente en disco. PostgreSQL crea automáticamente páginas, metadatos y entradas en el catálogo del sistema.
- `SERIAL` no es un tipo real, sino un atajo sintáctico: crea internamente una `SEQUENCE` y un `DEFAULT nextval()`. Se usa por simplicidad, aunque en sistemas modernos suele preferirse `GENERATED AS IDENTITY`.
- `PRIMARY KEY` crea simultáneamente:
  - Una restricción de unicidad
  - Un índice B-Tree
  - Una garantía semántica de identidad
- `DEFAULT now()` delega al servidor la responsabilidad de asignar el timestamp, evitando inconsistencias entre clientes.

Esta sintaxis refleja una decisión de diseño: la base de datos controla el estado mínimo coherente de la entidad.

2.2. Restricciones avanzadas

```
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(id),
    amount NUMERIC(10,2) CHECK (amount > 0)
);
```

Por qué esta sintaxis es importante:

- `REFERENCES users(id)` implementa una **clave foránea**. PostgreSQL valida esta relación en cada operación de inserción, actualización o borrado, garantizando integridad referencial incluso bajo concurrencia.
- `NUMERIC(10,2)` se elige frente a `FLOAT` porque PostgreSQL evita errores de precisión en operaciones financieras.
- `CHECK` se evalúa en el motor, no en la aplicación. Esto previene datos corruptos incluso si múltiples servicios escriben simultáneamente.

La sintaxis de restricciones refleja una filosofía defensiva: *la base de datos no confía en el cliente.*

3. Tipos de Datos Avanzados

3.1. Arrays

```
CREATE TABLE products (
    id SERIAL,
    tags TEXT[]
);
```

Justificación técnica:
PostgreSQL permite arrays porque su modelo interno soporta tipos complejos. Esta sintaxis evita tablas intermedias cuando:

- El conjunto es pequeño
- No se consulta de forma relacional
- No requiere claves externas

Internamente, el array se almacena como un único valor con metadatos de dimensión.

3.2. JSONB

```
CREATE TABLE events (
    id SERIAL,
    payload JSONB
);
```

```
SELECT payload->>'type'
FROM events
WHERE payload ? 'user_id';
```

Por qué JSONB y no JSON:

- JSONB se almacena en formato binario
- Permite indexación GIN
- Elimina claves duplicadas

Los operadores `->` y `?` forman parte de una sintaxis especializada que permite tratar documentos como estructuras consultables sin abandonar SQL.

4. Consultas SQL Avanzadas

4.1. JOINs

```
SELECT u.username, o.amount
FROM users u
LEFT JOIN orders o ON o.user_id = u.id;
```

Razón del LEFT JOIN:
El optimizador de PostgreSQL garantiza que:

- No se pierdan filas de la tabla izquierda
- Se preserva la cardinalidad original

Esto es fundamental en reportes donde la ausencia de datos también es información.

4.2. Subconsultas

```
SELECT *
FROM users
WHERE id IN (
    SELECT user_id FROM orders WHERE amount > 100
);
```

Esta sintaxis expresa dependencia lógica entre conjuntos. PostgreSQL puede reescribir internamente esta consulta como un `JOIN` o un `SEMI-JOIN` según estadísticas y costos.

5. CTE y Consultas Recursivas

```
WITH RECURSIVE tree AS (
    SELECT id, parent_id, name
    FROM categories
    WHERE parent_id IS NULL
    UNION ALL
    SELECT c.id, c.parent_id, c.name
    FROM categories c
    JOIN tree t ON c.parent_id = t.id
)
SELECT * FROM tree;
```

La palabra clave `RECURSIVE` instruye al motor a evaluar la CTE iterativamente hasta que no se produzcan nuevas filas, resolviendo estructuras jerárquicas sin bucles externos.

6. Funciones de Ventana

```
SELECT
    user_id,
    amount,
    SUM(amount) OVER (PARTITION BY user_id) AS total_user
FROM orders;
```

La cláusula `OVER` define una ventana lógica sin colapsar filas. Esto es posible gracias al planificador de PostgreSQL, que separa fases de agregación y proyección.

7. Índices Avanzados

7.1. Índice parcial

```
CREATE INDEX idx_active_users
ON users(id)
WHERE active = true;
```

Esta sintaxis permite crear índices más pequeños y selectivos, mejorando el rendimiento sin penalizar escrituras innecesarias.

7.2. Índice en expresión

```
CREATE INDEX idx_lower_email
ON users (lower(email));
```

El índice almacena el resultado de la expresión, evitando evaluaciones repetidas en tiempo de consulta.

7.3. GIN para JSONB

```
CREATE INDEX idx_payload
ON events
USING GIN (payload);
```

GIN (Generalized Inverted Index) permite indexar múltiples claves por fila, algo imposible con índices B-Tree tradicionales.

8. Transacciones

```
BEGIN;
-- operaciones
COMMIT;
```

Esta sintaxis activa el modelo ACID completo, usando MVCC y WAL para garantizar atomicidad, aislamiento y durabilidad.

8.1. Rollback

```
ROLLBACK;
```

Cancela la transacción descartando versiones no confirmadas.

9. Vistas

9.1. Vista

```
CREATE VIEW user_orders AS
SELECT u.username, o.amount
FROM users u
JOIN orders o ON o.user_id = u.id;
```

Una vista no almacena datos, sino una definición lógica reutilizable.

9.2. Vista materializada

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT user_id, SUM(amount) total
FROM orders
GROUP BY user_id;
```

Aquí la sintaxis indica persistencia física, sacrificando frescura por rendimiento.