Postgres-Specific Features: Beyond Standard SQL

Why This Matters

If you're using Postgres (and you probably are), these features are **game-changers**. They're not in standard SQL, but they solve real problems elegantly.

Trade-off: Using them locks you into Postgres. But if you're already on Postgres, use them.

JSONB: Semi-Structured Data

What It Is

JSONB is a binary JSON format. Unlike TEXT, it's:

- **Parsed and indexed**
- **Faster to query**
- **Validated** (invalid JSON is rejected)

Basic Usage

```sql
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  attributes JSONB NOT NULL DEFAULT '{}'
);

INSERT INTO products (name, attributes) VALUES
  ('Laptop', '{"brand": "Dell", "ram_gb": 16, "ssd_gb": 512}'),
  ('Mouse', '{"brand": "Logitech", "wireless": true}');
```

Querying JSONB

Extract Values (`->` and `->>`)

```sql
-- -> returns JSONB
SELECT attributes->'brand' FROM products;
-- Result: "Dell" (JSONB string)

-- ->> returns TEXT
SELECT attributes->>'brand' FROM products;
-- Result: Dell (plain text)
```

Nested Access

```sql
SELECT attributes->'specs'->>'cpu' FROM products;
```

Check for Key Existence

```sql
SELECT * FROM products WHERE attributes ? 'wireless';
-- Returns products with a "wireless" key
```

Containment (`@>`)

```sql
SELECT * FROM products WHERE attributes @> '{"brand": "Dell"}';
-- Returns products where attributes contains {"brand": "Dell"}
```

Indexing JSONB

GIN Index (General Inverted Index)

```sql
CREATE INDEX idx_products_attributes ON products USING GIN(attributes);
```

What it does: Indexes all keys and values in the JSONB.

Use case: Fast containment queries (`@>` , `?` , `?&` , `?|`).

Example:

```sql
SELECT * FROM products WHERE attributes @> '{"brand": "Dell"}';
-- Uses idx_products_attributes
```

Expression Index

Index a specific key:

```sql
CREATE INDEX idx_products_brand ON products ((attributes->>'brand'));
```

Use case: Fast lookups on a specific JSONB key.

```sql
SELECT * FROM products WHERE attributes->>'brand' = 'Dell';
-- Uses idx_products_brand
```

When to Use JSONB

✅ **Use when:**

- **Schema varies** (e.g., product attributes differ by category)
- **Frequent schema changes** (adding columns is expensive)
- **Storing API responses** (already JSON)

❌ **Don't use when:**

- **Core business logic** (use proper columns)
- **Need strong typing** (JSONB is loosely typed)
- **Complex queries** (joins on JSON are awkward)

JSONB Functions

Build JSON

```
SELECT JSONB_BUILD_OBJECT('id', id, 'name', name) FROM products;
-- Result: {"id": 1, "name": "Laptop"}
```

Aggregate into JSON

```
SELECT JSONB_AGG(JSONB_BUILD_OBJECT('id', id, 'name', name)) FROM products;
-- Result: [{"id": 1, "name": "Laptop"}, {"id": 2, "name": "Mouse"}]
```

Merge JSONB

```
SELECT attributes || '{"color": "black"}'::JSONB FROM products;
-- Adds "color" key to attributes
```

Arrays: Multi-Value Columns

Basic Usage

```
CREATE TABLE posts (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  tags TEXT[] NOT NULL DEFAULT '{}'
);

INSERT INTO posts (title, tags) VALUES
  ('SQL Tutorial', ARRAY['sql', 'database', 'postgres']),
  ('JS Guide', ARRAY['javascript', 'web']);
```

Querying Arrays

Check Containment (`@>`)

```
SELECT * FROM posts WHERE tags @> ARRAY['sql'];
-- Posts containing 'sql' tag
```

Check Overlap (`&&`)

```
SELECT * FROM posts WHERE tags && ARRAY['sql', 'javascript'];
-- Posts with at least one of the tags
```

ANY / ALL

```
SELECT * FROM posts WHERE 'sql' = ANY(tags);
-- Same as @> ARRAY['sql']
```

```sql
SELECT * FROM posts WHERE 'sql' = ALL(tags);
-- All tags must be 'sql' (rare)
```

Array Length

```sql
SELECT title, ARRAY_LENGTH(tags, 1) AS tag_count FROM posts;
```

Indexing Arrays

GIN Index

```sql
CREATE INDEX idx_posts_tags ON posts USING GIN(tags);
```

Use case: Fast containment / overlap queries.

When to Use Arrays

✅ **Use when:**

- **Fixed set of related values** (tags, categories)
- **Order doesn't matter** (or use ARRAY for ordered lists)

❌ **Don't use when:**

- **Need to JOIN on array elements** (use a junction table instead)
- **Array could grow unbounded** (performance degrades)

Alternative: Junction table with foreign keys (more normalized, easier to query).

UPSERT: INSERT ... ON CONFLICT

The Problem

Scenario: Insert a user, but if email exists, update their name.

Old way:

```sql
-- Check if exists
SELECT id FROM users WHERE email = 'alice@example.com';
-- If exists, UPDATE; else INSERT
```

Race condition: Another transaction might insert between SELECT and INSERT.

The Solution: ON CONFLICT

```sql
INSERT INTO users (email, name)
VALUES ('alice@example.com', 'Alice')
ON CONFLICT (email) DO UPDATE
  SET name = EXCLUDED.name;
```

What it does:

- Tries to INSERT
- If email conflict (UNIQUE constraint), UPDATE instead
- `EXCLUDED` refers to the row that would have been inserted

Atomic and race-free.

ON CONFLICT DO NOTHING

```
INSERT INTO users (email, name)
VALUES ('alice@example.com', 'Alice')
ON CONFLICT (email) DO NOTHING;
```

What it does: If conflict, skip the insert (no error, no update).

Use case: "Insert if not exists" without locking.

RETURNING Clause

```
INSERT INTO users (email, name)
VALUES ('alice@example.com', 'Alice')
ON CONFLICT (email) DO UPDATE
  SET name = EXCLUDED.name
RETURNING id, email, name;
```

What it does: Returns the inserted/updated row.

Use case: Get the ID without a separate SELECT.

When to Use

✅ **Use when:**

- **Idempotent writes** (repeat same INSERT safely)
- **Caching / session storage** (update if exists)

❌ **Don't use when:**

- **Need to distinguish insert vs update** in application logic (check `xmax` system column, or use separate queries)

RETURNING: Get Data Back from Writes

Basic Usage

```
INSERT INTO users (email, name) VALUES ('bob@example.com', 'Bob') RETURNING id;
-- Returns: id = 123

UPDATE users SET name = 'Robert' WHERE id = 123 RETURNING id, name;
-- Returns: id = 123, name = 'Robert'
```

```sql
DELETE FROM users WHERE id = 123 RETURNING *;
-- Returns: all columns of deleted row
```

Benefits:

- **No extra SELECT** (one round-trip instead of two)
- **Atomic** (no race condition)

Advanced Usage

```sql
WITH deleted AS (
  DELETE FROM users WHERE active = false RETURNING *
)
SELECT COUNT(*) FROM deleted;
-- Count how many users were deleted
```

When to Use

✅ **Always, when you need the result:**

- After INSERT (get the generated ID)
- After UPDATE (verify what changed)
- After DELETE (audit / logging)

LATERAL: Correlated Subqueries in FROM

The Problem

Goal: For each user, show their 3 most recent orders.

Naive approach (doesn't work):

```sql
SELECT u.name, o.id
FROM users u
JOIN (
  SELECT * FROM orders WHERE user_id = u.id ORDER BY created_at DESC LIMIT 3
) o;
-- ERROR: u.id not available in subquery
```

The Solution: LATERAL

```sql
SELECT u.name, o.id, o.created_at
FROM users u
LEFT JOIN LATERAL (
  SELECT * FROM orders WHERE user_id = u.id ORDER BY created_at DESC LIMIT 3
) o ON true;
```

What LATERAL does: Allows the subquery to reference `u` (the outer query).

Result: For each user, join their top 3 orders.

When to Use

✅ **Use when:**

- **Top-N per group** (most recent orders, top products, etc.)
- **Correlated subqueries in FROM**

❌ **Don't use when:**

- **Window functions work** (they're often simpler)

Example (equivalent with window functions):

```
SELECT u.name, o.id, o.created_at
FROM users u
LEFT JOIN (
  SELECT *, ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY created_at DESC) AS rn
  FROM orders
) o ON u.id = o.user_id AND o.rn <= 3;
```

Both work. LATERAL is more explicit about the correlation.

Full-Text Search

Basic Usage

```
-- Create a tsvector column
ALTER TABLE posts ADD COLUMN search_vector TSVECTOR;

-- Populate it
UPDATE posts SET search_vector = TO_TSVECTOR('english', title || ' ' || body);

-- Create a GIN index
CREATE INDEX idx_posts_search ON posts USING GIN(search_vector);

-- Search
SELECT * FROM posts WHERE search_vector @@ TO_TSQUERY('english', 'sql & tutorial');
```

What it does:

- **Stemming:** "running" matches "run"
- **Stop words:** Ignores "the", "and", etc.
- **Ranking:** Can sort by relevance

Generated Column for Search

```
CREATE TABLE posts (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  body TEXT NOT NULL,
  search_vector TSVECTOR GENERATED ALWAYS AS (
    TO_TSVECTOR('english', title || ' ' || body)
```

```
  ) STORED
);

CREATE INDEX idx_posts_search ON posts USING GIN(search_vector);
```

Automatically updated on INSERT/UPDATE.

When to Use

✅ **Use when:**

- **Basic search** (blog posts, documents)
- **Don't need Elasticsearch** (Postgres FTS is good enough)

❌ **Don't use when:**

- **Need advanced features** (fuzzy search, typo tolerance, facets)
- **Huge corpus** (Elasticsearch scales better)

ENUM Types

Basic Usage

```
CREATE TYPE order_status AS ENUM ('pending', 'processing', 'shipped', 'delivered',
'cancelled');

CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  status order_status NOT NULL DEFAULT 'pending'
);

INSERT INTO orders (status) VALUES ('shipped');
INSERT INTO orders (status) VALUES ('invalid');  -- ERROR: invalid input value
```

Benefits:

- **Type-safe** (typos are errors)
- **Self-documenting**
- **Efficient** (stored as integers internally)

Modifying ENUMs

Add a value:

```
ALTER TYPE order_status ADD VALUE 'returned' AFTER 'delivered';
```

Can't remove values (Postgres limitation). Workaround: Use lookup tables.

When to Use

✅ **Use when:**

- **Fixed set of values** that rarely changes

- **Type safety matters**

❌ **Don't use when:**

- **Values change frequently** (use lookup table)

Range Types

Basic Usage

Date ranges:

```
CREATE TABLE events (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  duration DATERANGE NOT NULL
);

INSERT INTO events (name, duration) VALUES
  ('Conference', '[2024-06-01, 2024-06-03)');  -- Inclusive start, exclusive end
```

Check overlap:

```
SELECT * FROM events WHERE duration && '[2024-06-02, 2024-06-04)';
-- Returns events overlapping with this range
```

EXCLUSION Constraints

Prevent double-booking:

```
CREATE TABLE reservations (
  id SERIAL PRIMARY KEY,
  room_id INT NOT NULL,
  duration TSTZRANGE NOT NULL,
  EXCLUDE USING GIST (room_id WITH =, duration WITH &&)
);

INSERT INTO reservations VALUES (1, 1, '[2024-06-01 10:00, 2024-06-01 12:00)');
INSERT INTO reservations VALUES (2, 1, '[2024-06-01 11:00, 2024-06-01 13:00)');
-- ERROR: conflicting key value violates exclusion constraint
```

Enforced at the database level. No race conditions.

When to Use

✅ **Use when:**

- **Time slots** (reservations, appointments)
- **IP address ranges**
- **Need overlap detection**

CTEs with MATERIALIZED Hint

Force Materialization

```
WITH expensive_calc AS MATERIALIZED (
  SELECT user_id, SUM(total) AS total_spent
  FROM orders
  WHERE created_at > NOW() - INTERVAL '1 year'
  GROUP BY user_id
)
SELECT * FROM expensive_calc WHERE total_spent > 10000
UNION ALL
SELECT * FROM expensive_calc WHERE total_spent < 100;
```

What it does: Computes `expensive_calc` once, stores it, reuses it.

When to use: CTE is expensive and referenced multiple times.

Extensions

Postgres has a rich extension ecosystem.

pg_trgm: Fuzzy String Matching

```
CREATE EXTENSION pg_trgm;

-- Similarity search
SELECT * FROM products WHERE name % 'Labtop';  -- Matches 'Laptop'

-- CREATE GIN index for LIKE
CREATE INDEX idx_products_name_trgm ON products USING GIN(name gin_trgm_ops);

-- Fast LIKE queries
SELECT * FROM products WHERE name ILIKE '%lap%';
```

uuid-ossp: UUID Generation

```
CREATE EXTENSION "uuid-ossp";

SELECT uuid_generate_v4();
-- Result: 550e8400-e29b-41d4-a716-446655440000
```

PostGIS: Geospatial Data

```
CREATE EXTENSION postgis;

-- Store geographic coordinates
ALTER TABLE locations ADD COLUMN geom GEOMETRY(POINT, 4326);

-- Find nearby locations
```

```sql
SELECT * FROM locations
WHERE ST_DWithin(geom, ST_MakePoint(-73.935242, 40.730610), 1000);
-- Within 1km of coordinates
```

When NOT to Use Postgres-Specific Features

❌ **Avoid if:**

- **You might switch databases** (MySQL, Oracle, SQL Server)
- **Using an ORM that doesn't support them** (check Prisma/Drizzle docs)
- **Team doesn't know Postgres well** (learning curve)

✅ **Use if:**

- **Committed to Postgres** (most modern stacks are)
- **Features solve real problems** (don't use for novelty)

Key Takeaways

1. **JSONB for semi-structured data.** Index with GIN, query with `@>` , `->` , `->>` .
2. **Arrays for multi-value columns.** Use GIN index, query with `@>` , `&&` .
3. **UPSERT with ON CONFLICT** for idempotent writes.
4. **RETURNING gets data back from writes** (one round-trip).
5. **LATERAL for correlated subqueries in FROM** (top-N per group).
6. **Full-text search with tsvector/tsquery.** GIN index + TO_TSVECTOR.
7. **ENUM types for type-safe fixed values.**
8. **Range types + EXCLUSION constraints** prevent overlapping ranges.
9. **Extensions add powerful features** (pg_trgm, PostGIS, uuid-ossp).
10. **Trade-off:** Postgres lock-in vs powerful features. Usually worth it.

Next up: SQL vs ORM—when to use each.