Datos Geoespaciales

Bases de Datos 1 (CC3088) - 2026

Datos Geoespaciales

Semestre 01, 2026

El problema con coordenadas en columnas normales

Una aplicación de delivery guarda la ubicación de sus restaurantes.

```
CREATE TABLE restaurantes (
    id      INT PRIMARY KEY,
    nombre  VARCHAR(200),
    lat     DECIMAL(9, 6),
    lng     DECIMAL(9, 6)
);
```

Funciona para almacenar. Pero las preguntas reales no son de almacenamiento.

Las preguntas reales

- ¿Cuáles restaurantes están a menos de 3 km del usuario?
- ¿Qué repartidores están dentro de la zona de cobertura?
- ¿Cuál es la ruta más corta entre dos puntos?
- ¿Este punto está dentro de este polígono (zona de despacho)?

Con columnas `lat` y `lng` separadas, cada consulta requiere cálculos trigonométricos complejos.

El cálculo de distancia sin extensión geoespacial

```
-- Distancia aproximada entre dos puntos (fórmula de Haversine)
SELECT nombre,
    6371 * 2 * ASIN(SQRT(
        POWER(SIN(RADIANS(lat - -14.6435) / 2), 2) +
        COS(RADIANS(-14.6435)) * COS(RADIANS(lat)) *
        POWER(SIN(RADIANS(lng - -90.5133) / 2), 2)
    )) AS distancia_km
FROM restaurantes
ORDER BY distancia_km
LIMIT 5;
```

Este código es propenso a errores, difícil de mantener y no usa índices eficientemente.

Las bases de datos relacionales tienen una solución mejor.

Bases de datos geoespaciales

Las bases de datos geoespaciales extienden el motor relacional con:

- **Tipos de dato** para representar geometrías (puntos, líneas, polígonos).
- **Funciones** para cálculos espaciales (distancia, área, intersección).
- **Índices** optimizados para búsquedas espaciales.

Todo integrado en SQL estándar.

El estándar OGC/ISO

La industria usa el estándar **Simple Features** del Open Geospatial Consortium (OGC).

Define un conjunto de tipos de geometría y funciones con prefijo `ST_` (Spatial Type).

Tanto PostgreSQL/PostGIS como MySQL implementan este estándar.

Las funciones `ST_Distance`, `ST_Contains`, `ST_Intersects` funcionan igual en ambos.

PostgreSQL: PostGIS

PostGIS es una extensión para PostgreSQL.

Es el motor geoespacial más completo y usado del ecosistema open source.

Soporta geometrías 2D, 3D, geografía esférica, raster y topología.

```
CREATE EXTENSION postgis;
```

Una sola línea habilita cientos de funciones geoespaciales.

MySQL: Spatial

MySQL tiene soporte geoespacial integrado desde la versión 5.7.

No requiere extensión adicional.

Más limitado que PostGIS, pero suficiente para casos de uso comunes.

```
-- MySQL ya tiene soporte espacial integrado
-- Solo hay que usar los tipos correctos al crear las tablas
```

Tipos de geometría

Los tipos siguen el estándar WKT (Well-Known Text) para representarse.

POINT — Punto

La geometría más básica: una sola ubicación en el espacio.

```
-- Un punto en coordenadas (longitud, latitud)
POINT(-90.5133 14.6435)
```

Casos de uso: ubicación de un restaurante, posición de un vehículo, dirección de un cliente.

LINESTRING — Línea

Una secuencia de puntos conectados que forman una línea.

```
-- Una ruta entre tres puntos
LINESTRING(-90.5133 14.6435, -90.5200 14.6500, -90.5300 14.6600)
```

Casos de uso: rutas de entrega, calles, ríos, tuberías.

POLYGON — Polígono

Una región cerrada definida por un anillo exterior (y opcionalmente anillos interiores para huecos).

```
-- El primer y último punto deben ser iguales para cerrar el polígono
POLYGON((-90.50 14.64, -90.52 14.64, -90.52 14.66, -90.50 14.66, -90.50 14.64))
```

Casos de uso: zona de cobertura de un restaurante, departamento, delimitación de un barrio.

Tipos compuestos

| Tipo | Descripción | Caso de uso | | --- | --- | --- | | `MULTIPOINT` | Colección de puntos | Sucursales de una empresa | | `MULTILINESTRING` | Colección de líneas | Red de calles | | `MULTIPOLYGON` | Colección de polígonos | Municipios de un departamento | | `GEOMETRYCOLLECTION` | Mezcla de geometrías | Datos heterogéneos |

SRID: Sistema de referencia de coordenadas

Cada geometría lleva asociado un SRID (Spatial Reference ID).

El SRID 4326 es el más común: coordenadas geográficas (longitud, latitud) en el sistema WGS 84.

Es el sistema que usa GPS y Google Maps.

```
-- Punto en Guatemala City con SRID 4326
ST_SetSRID(ST_MakePoint(-90.5133, 14.6435), 4326)
```

Mezclar geometrías de distintos SRID produce resultados incorrectos.

Tablas con columnas geoespaciales

PostgreSQL (PostGIS)

```
CREATE TABLE restaurantes (
    id       SERIAL PRIMARY KEY,
    nombre   VARCHAR(200)   NOT NULL,
    telefono VARCHAR(20),
    ubicacion GEOMETRY(POINT, 4326) NOT NULL
);
```

`GEOMETRY(POINT, 4326)` — tipo punto, sistema WGS 84.

MySQL

```
CREATE TABLE restaurantes (
    id       INT AUTO_INCREMENT PRIMARY KEY,
    nombre   VARCHAR(200)   NOT NULL,
    telefono VARCHAR(20),
    ubicacion POINT          NOT NULL SRID 4326
) ENGINE=InnoDB;
```

Insertar un punto

PostgreSQL:

```
INSERT INTO restaurantes (nombre, ubicacion)
VALUES (
    'Restaurante El Portal',
    ST_SetSRID(ST_MakePoint(-90.5133, 14.6435), 4326)
);
```

MySQL:

```
INSERT INTO restaurantes (nombre, ubicacion)
VALUES (
    'Restaurante El Portal',
    ST_GeomFromText('POINT(-90.5133 14.6435)', 4326)
);
```

Leer una geometría como texto

```
-- PostgreSQL
SELECT nombre, ST_AsText(ubicacion) AS coordenadas
FROM restaurantes;
```

```
-- POINT(-90.5133 14.6435)

-- También como GeoJSON (compatible con mapas web)
SELECT nombre, ST_AsGeoJSON(ubicacion) AS geojson
FROM restaurantes;
-- {"type":"Point","coordinates":[-90.5133,14.6435]}
```

Funciones de distancia

ST_Distance

Calcula la distancia entre dos geometrías.

```
-- PostgreSQL: distancia en grados (no útil en práctica)
SELECT ST_Distance(
    ST_SetSRID(ST_MakePoint(-90.5133, 14.6435), 4326),
    ubicacion
) AS distancia
FROM restaurantes;
```

Para distancias reales en metros, usar el tipo `GEOGRAPHY` en PostGIS:

```
-- PostGIS: distancia real en metros usando geografía esférica
SELECT nombre,
    ST_Distance(
        ubicacion::geography,
        ST_SetSRID(ST_MakePoint(-90.5133, 14.6435), 4326)::geography
    ) AS distancia_metros
FROM restaurantes
ORDER BY distancia_metros
LIMIT 5;
```

ST_DWithin — buscar dentro de un radio

La función más usada en aplicaciones de geolocalización.

```
-- Restaurantes a menos de 3 km del usuario
-- PostGIS (geografía esférica — metros reales)
SELECT nombre
FROM restaurantes
WHERE ST_DWithin(
    ubicacion::geography,
    ST_SetSRID(ST_MakePoint(-90.5133, 14.6435), 4326)::geography,
    3000  -- 3000 metros
);
```

```
-- MySQL: usar ST_Distance_Sphere para metros
SELECT nombre,
    ST_Distance_Sphere(
        ubicacion,
        ST_GeomFromText('POINT(-90.5133 14.6435)', 4326)
    ) AS distancia_metros
FROM restaurantes
HAVING distancia_metros < 3000
ORDER BY distancia_metros;
```

Comparación: columnas numéricas vs geometría

| Criterio | lat/lng separados | Columna GEOMETRY | | --- | --- | --- | | Cálculo de distancia | Fórmula de Haversine manual | `ST_Distance` / `ST_DWithin` | | Índice para búsqueda espacial | No posible eficientemente | Índice GIST / R-Tree | | Verificar si está dentro de un polígono | Algoritmo point-in-polygon manual | `ST_Contains` | | Estándar de la industria | No | Sí (OGC Simple Features) |

Funciones de relación espacial

ST_Contains — ¿el punto está dentro del polígono?

```
-- ¿La dirección del cliente está dentro de la zona de cobertura?
SELECT z.nombre AS zona
FROM zonas_cobertura z
WHERE ST_Contains(
    z.area,
    ST_SetSRID(ST_MakePoint(-90.5200, 14.6500), 4326)
);
```

Caso de uso: verificar si una dirección de entrega está dentro del área de despacho de un restaurante.

ST_Within — inverso de ST_Contains

```
-- ¿El restaurante está dentro de la zona habilitada?
SELECT r.nombre
FROM restaurantes r, zonas_habilitadas z
WHERE ST_Within(r.ubicacion, z.area);
```

`ST_Within(A, B)` equivale a `ST_Contains(B, A)`.

ST_Intersects — ¿dos geometrías se tocan?

```
-- ¿La ruta del repartidor pasa por la zona de alta demanda?
SELECT r.nombre AS repartidor
FROM repartidores r, zonas_alta_demanda z
WHERE ST_Intersects(r.ruta_actual, z.area);
```

Caso de uso: detectar repartidores cuya ruta atraviesa una zona específica.

ST_Overlaps — ¿dos polígonos se superponen?

```
-- ¿Hay zonas de cobertura que se solapan entre restaurantes?
SELECT a.nombre AS zona_a, b.nombre AS zona_b
FROM zonas_cobertura a, zonas_cobertura b
```

```
WHERE a.id < b.id
  AND ST_Overlaps(a.area, b.area);
```

Caso de uso: detectar conflictos de cobertura entre sucursales.

Tabla de relaciones espaciales

| Función | ¿Qué verifica? | | --- | --- | | `ST_Contains(A, B)` | A contiene completamente a B | | `ST_Within(A, B)` | A está completamente dentro de B | | `ST_Intersects(A, B)` | A y B tienen al menos un punto en común | | `ST_Overlaps(A, B)` | A y B se superponen parcialmente | | `ST_Touches(A, B)` | A y B se tocan solo en el borde | | `ST_Disjoint(A, B)` | A y B no tienen ningún punto en común |

Funciones de geometría

ST_Buffer — zona de influencia

Crea un polígono alrededor de una geometría a una distancia dada.

```
-- Zona de cobertura de 2 km alrededor de un restaurante (PostGIS)
SELECT ST_Buffer(
    ubicacion::geography,
    2000  -- 2000 metros
)::geometry AS zona_cobertura
FROM restaurantes
WHERE id = 1;
```

Caso de uso: calcular y almacenar el área de cobertura de cada sucursal.

ST_Centroid — punto central de un polígono

```
-- Centro geográfico de cada zona
SELECT nombre, ST_AsText(ST_Centroid(area)) AS centro
FROM zonas;
```

Caso de uso: encontrar el punto óptimo para colocar un depósito en el centro de una zona.

ST_Area — área de un polígono

```
-- Área de cada zona en km²
SELECT nombre,
    ST_Area(area::geography) / 1000000 AS area_km2
FROM zonas
ORDER BY area_km2 DESC;
```

ST_Length — longitud de una línea

```
-- Longitud de rutas en metros
SELECT nombre,
    ST_Length(ruta::geography) AS longitud_metros
FROM rutas_entrega
ORDER BY longitud_metros;
```

ST_Union — unión de geometrías

```
-- Área total de cobertura de todos los restaurantes de una cadena
SELECT ST_AsText(ST_Union(area)) AS cobertura_total
FROM zonas_cobertura
WHERE cadena_id = 5;
```

ST_Intersection — intersección de dos geometrías

```
-- Área compartida entre dos zonas
SELECT ST_AsText(ST_Intersection(a.area, b.area)) AS zona_compartida
FROM zonas a, zonas b
WHERE a.id = 1 AND b.id = 2;
```

Índices espaciales

Sin índice espacial, cada consulta `ST_DWithin` o `ST_Contains` recorre toda la tabla.

Con millones de registros, esto es inaceptable.

PostGIS: índice GiST

GiST (Generalized Search Tree) — el índice espacial de PostgreSQL.

```
-- Crear índice espacial en PostGIS
CREATE INDEX idx_restaurantes_ubicacion
ON restaurantes
USING GIST (ubicacion);
```

El optimizador lo usa automáticamente en consultas con `ST_DWithin`, `ST_Contains`, `ST_Intersects`.

MySQL: índice SPATIAL

```
-- Crear índice espacial en MySQL
-- La columna debe ser NOT NULL para soportar índice SPATIAL
CREATE SPATIAL INDEX idx_restaurantes_ubicacion
ON restaurantes (ubicacion);
```

¿Cómo funciona el índice espacial?

El índice GiST/R-Tree divide el espacio en rectángulos delimitadores (bounding boxes).

Para buscar puntos dentro de un radio, primero filtra por bounding box (muy rápido).

Luego aplica el cálculo exacto solo a los candidatos.

Sin índice: revisar 1,000,000 filas. Con índice: revisar solo los ~20 candidatos en el área.

Verificar que el índice se usa

```
-- PostgreSQL: ver el plan de ejecución
EXPLAIN ANALYZE
SELECT nombre FROM restaurantes
WHERE ST_DWithin(
    ubicacion::geography,
    ST_SetSRID(ST_MakePoint(-90.5133, 14.6435), 4326)::geography,
    3000
);
-- Debe mostrar: Bitmap Index Scan on idx_restaurantes_ubicacion
```

Caso de uso 1: App de delivery

Una app de comida a domicilio necesita mostrar restaurantes cercanos.

```
-- Esquema
CREATE TABLE restaurantes (
    id        SERIAL PRIMARY KEY,
    nombre    VARCHAR(200) NOT NULL,
    categoria VARCHAR(100),
    activo    BOOLEAN DEFAULT TRUE,
    ubicacion GEOMETRY(POINT, 4326) NOT NULL
```

```
);

CREATE INDEX idx_rest_ubic ON restaurantes USING GIST (ubicacion);

-- Restaurantes activos a menos de 2 km del usuario, ordenados por distancia
SELECT id, nombre, categoria,
    ROUND(ST_Distance(
        ubicacion::geography,
        ST_SetSRID(ST_MakePoint(-90.5133, 14.6435), 4326)::geography
    )::numeric) AS distancia_metros
FROM restaurantes
WHERE activo = TRUE
  AND ST_DWithin(
      ubicacion::geography,
      ST_SetSRID(ST_MakePoint(-90.5133, 14.6435), 4326)::geography,
      2000
  )
ORDER BY distancia_metros
LIMIT 20;
```

Verificar zona de cobertura

```
CREATE TABLE zonas_cobertura (
    id             SERIAL PRIMARY KEY,
    restaurante_id INT REFERENCES restaurantes(id),
    area           GEOMETRY(POLYGON, 4326) NOT NULL
);

-- ¿La dirección del cliente está dentro de alguna zona de cobertura?
SELECT r.nombre AS restaurante
FROM zonas_cobertura z
JOIN restaurantes r ON r.id = z.restaurante_id
WHERE ST_Contains(
    z.area,
    ST_SetSRID(ST_MakePoint(-90.5200, 14.6500), 4326)
)
AND r.activo = TRUE;
```

Caso de uso 2: Seguimiento de vehículos

Una empresa de transporte monitorea su flota en tiempo real.

```
CREATE TABLE vehiculos (
    id        SERIAL PRIMARY KEY,
    placa     VARCHAR(20) NOT NULL,
    conductor VARCHAR(200)
);

CREATE TABLE posiciones (
    id          SERIAL PRIMARY KEY,
    vehiculo_id INT REFERENCES vehiculos(id),
    posicion    GEOMETRY(POINT, 4326) NOT NULL,
    velocidad   DECIMAL(5, 2),
    timestamp   TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_pos_ubic ON posiciones USING GIST (posicion);
CREATE INDEX idx_pos_tiempo ON posiciones (timestamp);
```

¿Qué vehículos están en una zona restringida?

```
-- Zona restringida (por ejemplo, un aeropuerto)
WITH zona AS (
    SELECT ST_GeomFromText(
        'POLYGON((-90.53 14.58, -90.52 14.58, -90.52 14.59, -90.53 14.59, -90.53 14.58))'
        4326
    ) AS area
)
SELECT v.placa, v.conductor, p.timestamp
FROM posiciones p
JOIN vehiculos v ON v.id = p.vehiculo_id
CROSS JOIN zona
WHERE ST_Within(p.posicion, zona.area)
  AND p.timestamp > NOW() - INTERVAL '5 minutes';
```

Distancia recorrida por un vehículo hoy

```
-- Construir la ruta del vehículo y calcular su longitud
SELECT v.placa,
    ST_Length(
        ST_MakeLine(p.posicion ORDER BY p.timestamp)::geography
    ) AS metros_recorridos
FROM posiciones p
JOIN vehiculos v ON v.id = p.vehiculo_id
WHERE v.id = 7
  AND p.timestamp::date = CURRENT_DATE
GROUP BY v.id, v.placa;
```

Caso de uso 3: Análisis urbano

Una municipalidad analiza cobertura de servicios.

```
CREATE TABLE colonias (
    id        SERIAL PRIMARY KEY,
    nombre    VARCHAR(200),
    zona      INT,
    perimetro GEOMETRY(POLYGON, 4326)
);

CREATE TABLE centros_salud (
    id        SERIAL PRIMARY KEY,
    nombre    VARCHAR(200),
    tipo      VARCHAR(50),
    ubicacion GEOMETRY(POINT, 4326)
);
```

Colonias sin centro de salud a menos de 1 km

```
SELECT c.nombre AS colonia, c.zona
FROM colonias c
WHERE NOT EXISTS (
    SELECT 1
    FROM centros_salud cs
    WHERE ST_DWithin(
        cs.ubicacion::geography,
        ST_Centroid(c.perimetro)::geography,
        1000
    )
)
ORDER BY c.zona, c.nombre;
```

Área total de cobertura de los centros de salud

```
-- Buffer de 1 km alrededor de cada centro, luego unión para evitar duplicar solapamiento
SELECT ST_Area(
    ST_Union(ST_Buffer(ubicacion::geography, 1000)::geometry)::geography
) / 1000000 AS km2_cubiertos
FROM centros_salud
WHERE tipo = 'hospital';
```

Densidad de colonias por km²

```
SELECT nombre, zona,
    ST_Area(perimetro::geography) / 1000000 AS area_km2
FROM colonias
ORDER BY area_km2 DESC;
```

Caso de uso 4: Geofencing

Geofencing = disparar una acción cuando un objeto entra o sale de una zona.

```
CREATE TABLE geofences (
    id          SERIAL PRIMARY KEY,
    nombre      VARCHAR(200),
    descripcion TEXT,
    area        GEOMETRY(POLYGON, 4326) NOT NULL
);

CREATE TABLE eventos_geofence (
    id          SERIAL PRIMARY KEY,
    vehiculo_id INT REFERENCES vehiculos(id),
    geofence_id INT REFERENCES geofences(id),
    tipo        VARCHAR(10) CHECK (tipo IN ('entrada', 'salida')),
    timestamp   TIMESTAMPTZ DEFAULT NOW()
);
```

Detectar vehículos que entraron a una zona en las últimas 24 horas

```
SELECT v.placa, g.nombre AS zona, e.tipo, e.timestamp
FROM eventos_geofence e
JOIN vehiculos v ON v.id = e.vehiculo_id
JOIN geofences g ON g.id = e.geofence_id
WHERE e.timestamp > NOW() - INTERVAL '24 hours'
ORDER BY e.timestamp DESC;
```

Generar evento de entrada al detectar una nueva posición dentro de una zona

```
-- Verificar si la nueva posición está en alguna geofence
INSERT INTO eventos_geofence (vehiculo_id, geofence_id, tipo)
SELECT
    7 AS vehiculo_id,
    g.id AS geofence_id,
    'entrada' AS tipo
```

```
FROM geofences g
WHERE ST_Contains(
    g.area,
    ST_SetSRID(ST_MakePoint(-90.5200, 14.6500), 4326)
);
```

PostGIS vs MySQL Spatial

| Característica | PostGIS (PostgreSQL) | MySQL Spatial | | --- | --- | --- | | Tipos geométricos | Completo (2D, 3D, geografía) | Básico (2D) | | Geografía esférica | Sí (`::geography`) | Parcial (`ST_Distance_Sphere`) | | Funciones disponibles | +500 funciones | ~50 funciones | | Índice espacial | GiST (muy eficiente) | R-Tree (SPATIAL INDEX) | | Rendimiento | Excelente | Bueno para casos básicos | | Soporte raster | Sí | No | | Estándar OGC | Completo | Parcial | | Curva de aprendizaje | Mayor | Menor |

¿Cuándo usar cada uno?

| Caso | Recomendación | | --- | --- | | Distancias y radios simples | MySQL Spatial es suficiente | | Polígonos y relaciones espaciales básicas | MySQL Spatial es suficiente | | Geografía esférica precisa (km reales) | PostGIS obligatorio | | Análisis urbano, GIS completo | PostGIS obligatorio | | Proyecto ya en MySQL | MySQL Spatial | | Proyecto nuevo con requerimientos geoespaciales | PostGIS |

¿Qué puede salir mal?

- **SRID inconsistente** → mezclar geometrías de distintos sistemas de referencia produce resultados incorrectos o error; siempre especificar y verificar el SRID

- **Usar GEOMETRY en lugar de GEOGRAPHY para distancias** → `ST_Distance` con `GEOMETRY` devuelve grados, no metros; usar `::geography` en PostGIS para distancias reales
- **Sin índice espacial** → `ST_DWithin` sobre millones de filas sin índice hace un full scan; crear siempre el índice GIST/SPATIAL
- **Longitud y latitud invertidas** → el estándar es `(longitud, latitud)` en WKT y en ST_MakePoint, pero muchas APIs devuelven `(latitud, longitud)`; verificar la convención
- **Buffer en geometría en lugar de geografía** → `ST_Buffer` sobre GEOMETRY usa grados, no metros; convertir a `::geography` antes
- **Polígono no cerrado** → el último punto debe ser igual al primero; un polígono no cerrado produce error o resultados inesperados
- **No actualizar estadísticas después de cargas masivas** → el planner de PostgreSQL puede no usar el índice espacial si las estadísticas están desactualizadas; ejecutar `ANALYZE`
- **Consultas sin límite sobre datos densos** → `ST_DWithin` en una ciudad puede devolver miles de resultados; usar siempre `LIMIT` y `ORDER BY distancia`