



# Spatial Querying and Geodatabases

Understanding Spatial Relationships through SQL and Intelligent GIS Tools



**Yifan Yang, Ph.D. Student**

Advisor: Dr. Lei Zou (lzou@tamu.edu)

Department of Geography, Texas A&M University

GEAR Lab Website: <https://www.geearlab.com>

# Self Introduction

University of Southern California   
Master of Science, Spatial Data Science  
August 2022- May 2024

Hainan University   
Bachelor of Engineering, Software Engineering  
September 2018 - June 2022

## Research Interests

### 1. Spatial Data Science & GeoAI Integration

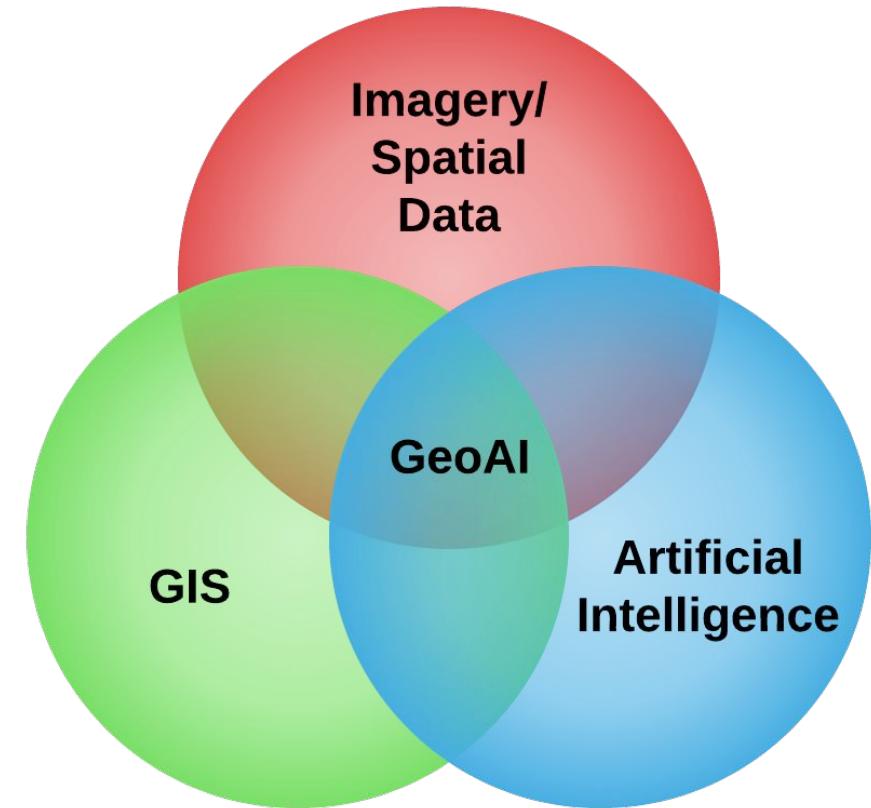
To integrate spatial data science with artificial intelligence for understanding and modeling geographic patterns.

Disaster Resilience: Predicting, assessing, and mitigating natural disasters using multi-source spatial data such as remote sensing, GIS, and street-view imagery.

### 2. Responsible and Autonomous GeoAI

To develop responsible and autonomous geographical AI systems that can perceive, reason, and act in complex real-world environments.

<https://rayford295.github.io/>



**GeoAI = GIS + Artificial Intelligence**

## Why I Am Taking This Course

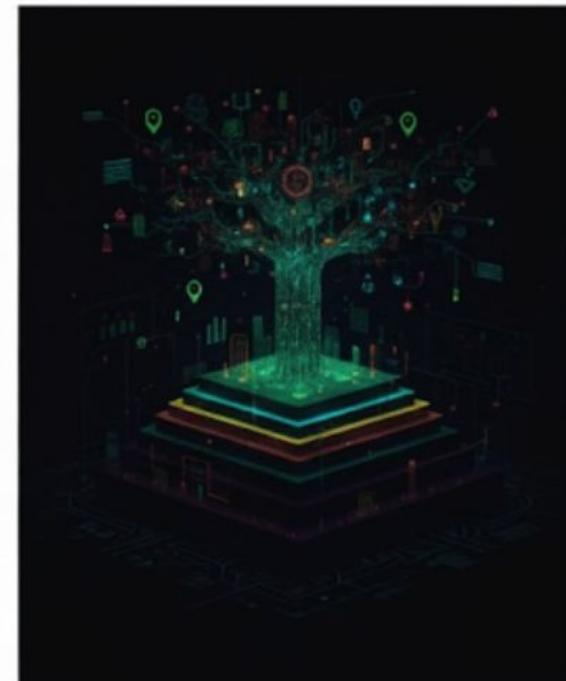
I have studied database-related courses during my **Bachelor (Software Engineering)**, **Master (Spatial Data Science)**, and **Ph.D. (Geography)**. *three times*



Bachelor (Software Engineering)



Master (Spatial Data Science)



Ph.D. Geography

# Learning Objectives

By the end of this session, students will be able to:

1. Define and explain spatial queries and describe their importance in geodatabases for geospatial analysis.
2. Use spatial SQL functions (e.g., ST\_Within, ST\_Intersects, ST\_Distance) to query and analyze geographic data.
3. Perform hands-on spatial analysis using a self-built geospatial database hosted on Supabase (PostGIS) to practice real-world spatial SQL operations.
4. Visualize and interpret query results interactively through the Spatial Query Lab (HTML sandbox).
5. Explore the concept of Agentic GIS, understanding how large language models (LLMs) and AI agents can translate natural language into spatial queries automatically.

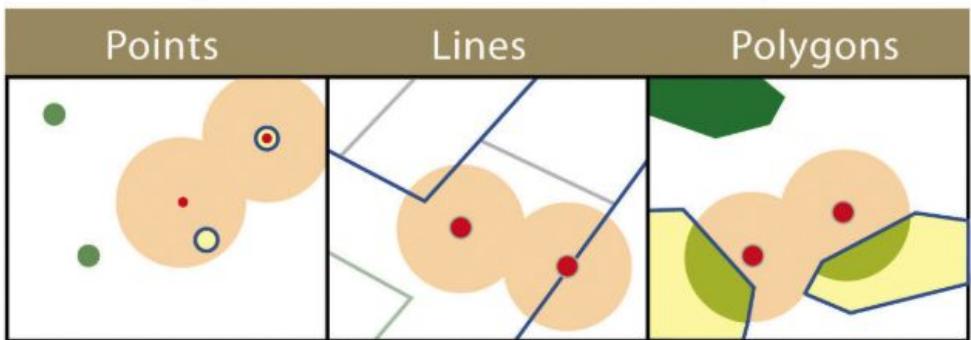
# What Are Spatial Queries?

## Definition

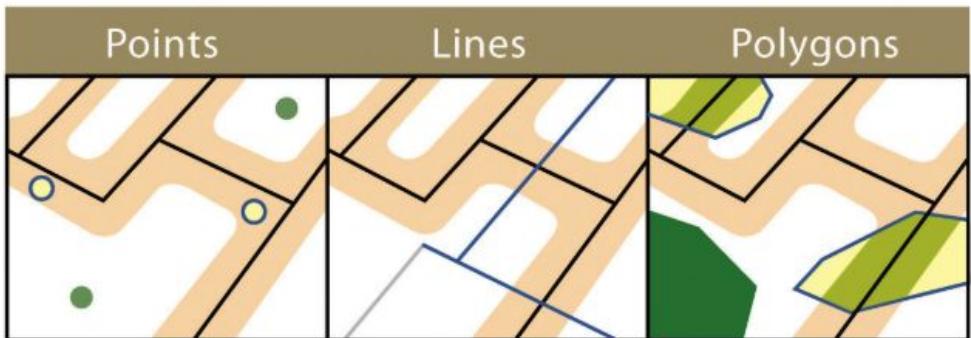
- A **spatial query** retrieves or analyzes geographic features based on their **location, shape, or spatial relationship** to other features.
- Unlike traditional SQL queries (which focus on attributes), spatial queries use **geometry and spatial relationships** as part of the condition.



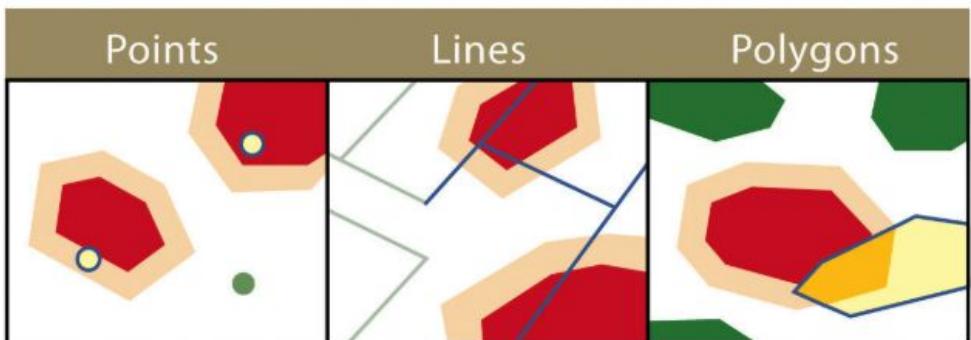
When finding features that are within a distance of point features



When finding features that are within a distance of line features



When finding features that are within a distance of polygon features



## Distance-Based Spatial Queries

Spatial queries can identify **features located within a specific distance** of other features (points, lines, or polygons).

The *distance relationship* depends on the geometry types involved:

- **Point-based queries:** find nearby features around point locations (e.g., fire hydrants near schools).
- **Line-based queries:** find features close to linear elements (e.g., buildings within 100 m of a road).
- **Polygon-based queries:** find features surrounding or overlapping area features (e.g., houses within a flood zone).

# How Are Points, Lines, and Polygons Stored in a Spatial Database?

## 1. Vector Geometry in Databases

- In a **spatial (PostGIS) database**, geographic features are stored as **geometries** — mathematical representations of shapes using coordinate pairs (x, y).
- Every geometry is stored inside a special column (usually named **geom**) with a specific type and Spatial Reference ID (**SRID**).
- These geometries follow the **OGC Simple Features** standard.

```
CREATE TABLE campus_safety (
    id SERIAL PRIMARY KEY,
    name TEXT,
    geom GEOMETRY(Point, 4326)
);
```

The geometry type here is **Point**, and SRID 4326 = WGS 84 (latitude/longitude).

# How Are Points, Lines, and Polygons Stored in a Spatial Database?

## a) Points

```
INSERT INTO campus_safety (name, geom)
VALUES ('Emergency Call Box', ST_SetSRID(ST_Point(-96.34, 30.62), 4326));
```

## b) Lines

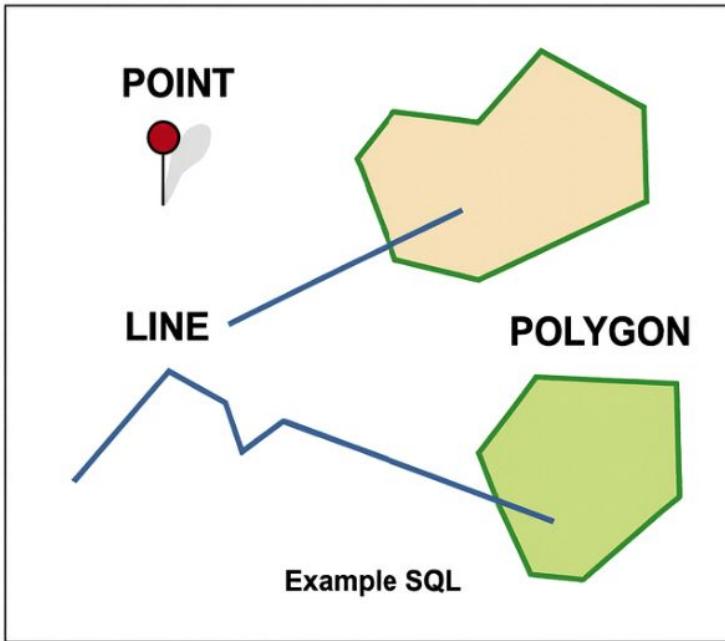
```
INSERT INTO roads (name, geom)
VALUES ('University Dr', ST_GeomFromText(
    'LINESTRING(-96.35 30.62, -96.34 30.621, -96.33 30.623)', 4326));
```

## c) Polygons

```
INSERT INTO buildings (name, geom)
VALUES ('Library', ST_GeomFromText(
    'POLYGON((-96.3400 30.6184, -96.3392 30.6184, -96.3392 30.6178,
              -96.3400 30.6178, -96.3400 30.6184))', 4326));
```

 *Notice:* polygons start and end with the same coordinate — that's how the boundary closes.

# How Are Points, Lines, and Polygons Stored in a Spatial Database?



Geometry Type	Dimension	Description	Example SQL
Point	0-D	A single coordinate (x,y).	<code>ST_Point(-96.34, 30.62)</code>
LineString	1-D	Sequence of connected points.	<code>ST_MakeLine(p1, p2, p3)</code>
Polygon	2-D	Closed ring of points forming an area.	<code>ST_GeomFromText('POLYGON((x1 y1, ...))', 4326)</code>



**Why only “points, lines, and polygons”?**

# The deeper reason — Geometry in coordinate space

GIS is grounded in **computational geometry**, which means features are stored as coordinate sets:

- A **point** = one coordinate pair (x, y)
- A **line** = ordered list of points
- A **polygon** = closed ring of points

When we perform spatial queries like `ST_DWithin`, `ST_Intersects`, or `ST_Contains`, the database needs mathematically defined shapes that can be compared using **Euclidean geometry** (distance, intersection, containment).

These three — point, line, polygon — are the only **geometrically complete** and **computationally stable** shapes defined in the OGC (Open Geospatial Consortium) standard.

# What about “other shapes” like circles or 3D surfaces?

- A **circle** in GIS is actually a **polygon** with many small straight edges — a “buffer” around a point.
- A **curve** is stored as a line made up of many tiny segments.
- **3D surfaces (TINs, DEMs)** are handled in specialized 3D or raster models, not in 2D vector geometry.
- **Raster data** (grids, pixels) are another model entirely — used for continuous surfaces (elevation, temperature), not discrete features.

So in **vector GIS**, everything can be expressed in these three types — and that’s why spatial queries refer to *points, lines, and polygons*.

# Spatial SQL Functions in Geodatabases

## What Are Spatial SQL Functions?

- Spatial SQL functions extend standard SQL with **geometry-based operations**.
- They allow databases (like PostGIS) to understand and process **spatial relationships** such as *distance*, *containment*, and *overlap*.
- These functions are the foundation for spatial queries and geospatial analysis.



*They don't just compare numbers — they compare shapes, locations, and boundaries.*

# Common Spatial SQL Functions

Function	Description	Example SQL	Result
<b>ST_Within(A, B)</b>	Checks if geometry A is <i>completely inside</i> geometry B.	<code>ST_Within(school.geom, city.geom)</code>	Returns TRUE/FALSE
<b>ST_Intersects(A, B)</b>	Returns TRUE if A and B <i>share any space</i> .	<code>ST_Intersects(road.geom, flood.geom)</code>	Overlap detection
<b>ST_Distance(A, B)</b>	Measures the distance between two geometries.	<code>ST_Distance(house.geom, road.geom)</code>	Returns numeric value (meters)
<b>ST_Buffer(A, d)</b>	Creates a buffer zone around geometry A with distance d.	<code>ST_Buffer(park.geom, 100)</code>	Polygon output
<b>ST_Area(A)</b>	Calculates area ( $m^2$ ) of polygon geometry.	<code>ST_Area(zone.geom)</code>	Returns numeric area

 **ST** = *Spatial Type* prefix, indicating a standardized **spatial operation** on geometries defined by OGC SQL/MM Spatial standards.

# Example: Finding Points Within a Polygon

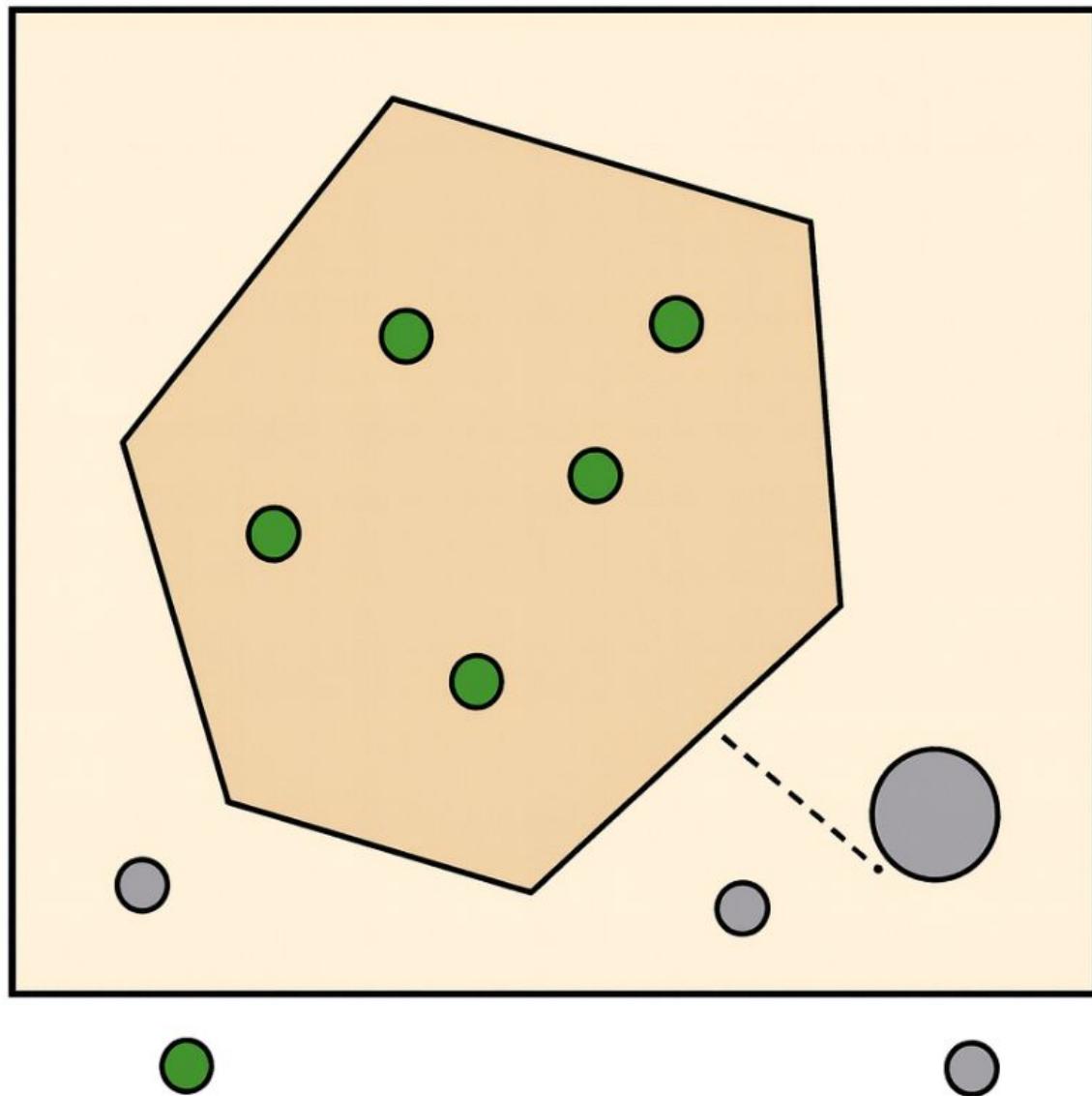
## ST\_Within Example

```
SELECT id, name  
FROM buildings  
WHERE  
ST_Within(buildings.geom,  
campus_boundary.geom);
```

- Find all buildings located inside the campus boundary.

Visualization idea:

- Show campus boundary as a polygon.
- Highlight points (buildings) inside it in green, outside in gray.



# Example: Finding Features that Intersect

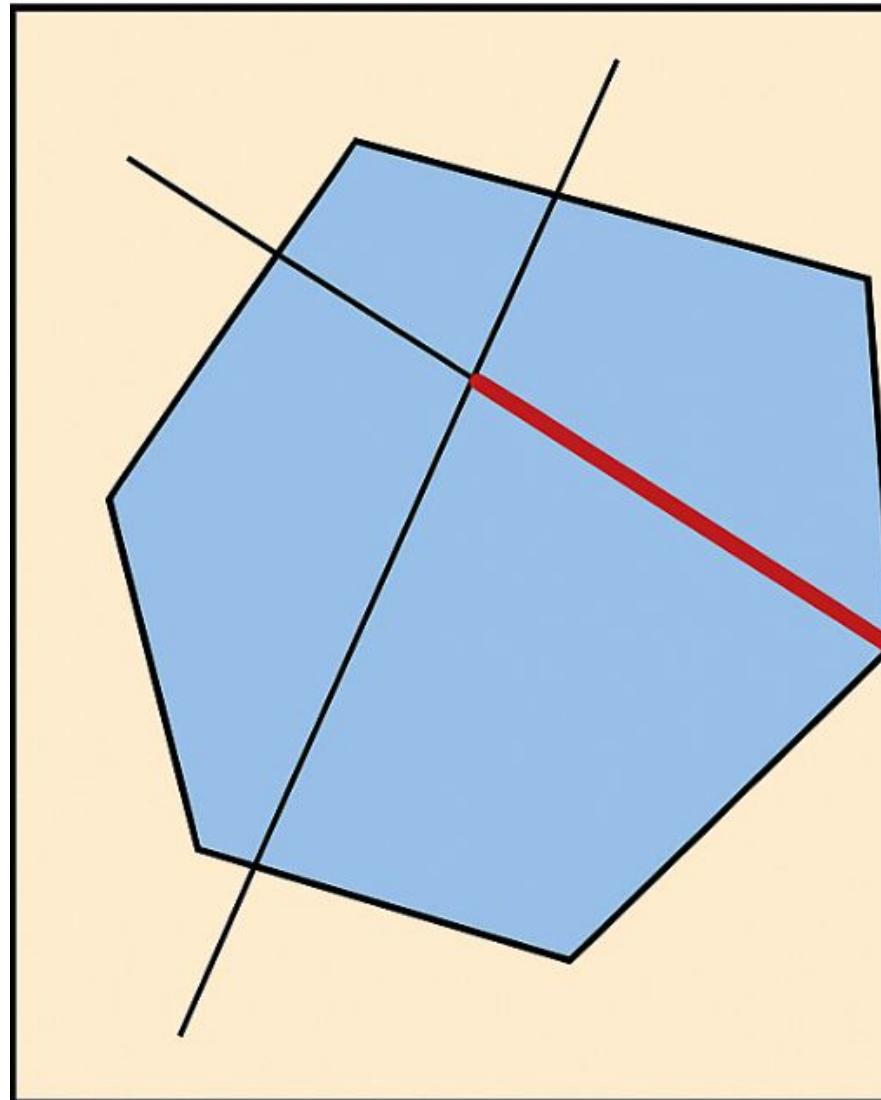
## ST\_Intersects Example

```
SELECT road.name,  
flood_zone.name  
FROM road, flood_zone  
WHERE ST_Intersects(road.geom,  
flood_zone.geom);
```

🚧 Identify roads affected by flooding.

Visualization idea:

- Flood zone polygon in **blue**
- Roads crossing it highlighted in **red**



# Example: Measuring Distance

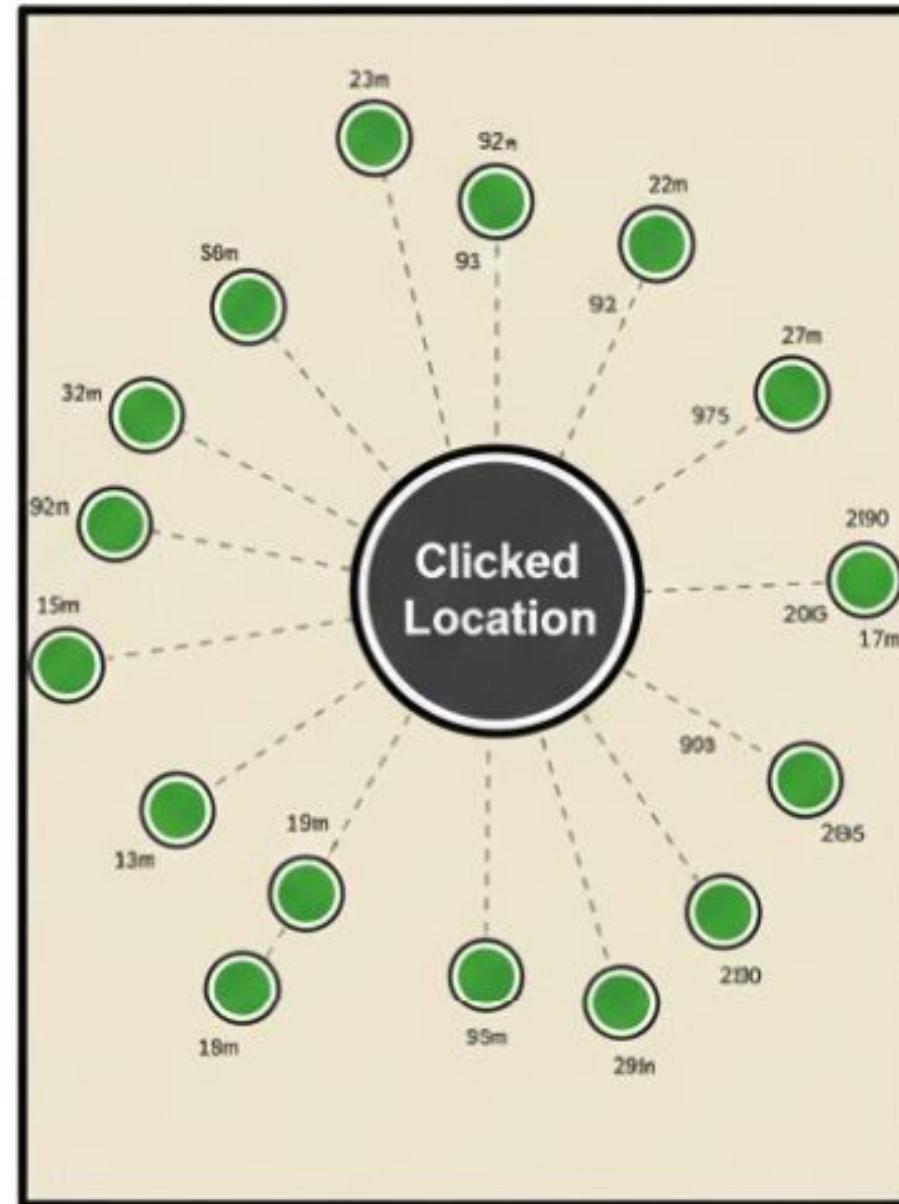
# ST\_Distance Example

```
SELECT id, name,
       ST_Distance(geom,
ST_SetSRID(ST_Point(-96.34, 30.62), 4326))
AS dist_m
FROM campus_safety
ORDER BY dist_m ASC
LIMIT 10;
```

-  Find the 10 nearest emergency call boxes to a location.

## Visualization idea:

- Map showing your clicked location + nearest points with distances labeled.



# Spatial vs. Non-Spatial Databases

Aspect	Regular Database	Spatial Database (PostGIS, SpatiaLite, etc.)
Data Type	Stores text, numbers, dates, etc.  Example: A customer table with columns <code>name</code> , <code>age</code> , <code>email</code> .	Stores <b>geometry types</b> (points, lines, polygons).  Example: A <code>buildings</code> table with <code>geom</code> column storing building footprints.
Query Language	Standard SQL for attribute filtering.  Example: <code>SELECT * FROM sales WHERE amount &gt; 1000;</code>	<b>Spatial SQL</b> uses geometry functions.  Example: <code>SELECT * FROM parks WHERE ST_Area(geom) &gt; 50000;</code>
Relationships	Based on <b>keys or attributes</b> .  Example: <code>JOIN</code> tables by <code>customer_id</code> .	Based on <b>spatial relationships</b> (distance, overlap, containment).  Example: <code>ST_Within(house.geom, flood_zone.geom)</code> finds houses inside flood zones.
Indexes	B-Tree for fast numeric/text lookups.  Example: Searching for names alphabetically.	<b>GIST / R-Tree</b> optimize geometry queries.  Example: Quickly find the 10 nearest hospitals ( <code>ST_Distance</code> ).
Visualization	Output is tabular or textual.  Example: A table of sales records.	Output can be <b>mapped and visualized</b> .  Example: Display all roads intersecting a flood zone.
Applications	Used in <b>finance, inventory, CRM</b> systems.  Example: Tracking product orders or customers.	Used in <b>urban planning, environmental monitoring, transportation, and disaster management</b> .  Example: Finding evacuation routes during a hurricane.





**Is SQL programming? How  
can I verify that I have  
mastered this skill?**

# Is SQL a Programming Language?

## Is SQL Programming?

- SQL (**Structured Query Language**) is **not** a *full* programming language like Python or Java.
- It is a **declarative language** — meaning you tell the database *what you want*, not *how to do it*.

## How can I verify that I have mastered this skill?

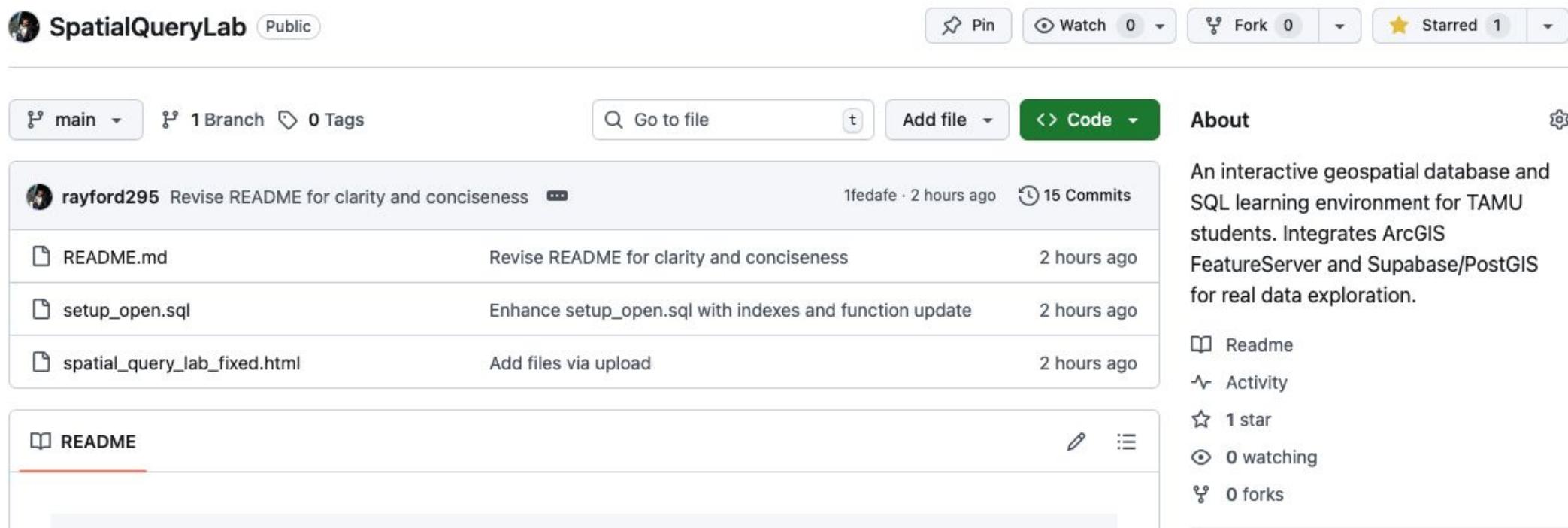
### 1. Learn by Doing — Build a Spatial Project

- The most effective way to confirm mastery is to **create your own database-based project**.
- Designing and managing a spatial database lets you practice **data creation, querying, and visualization** in one workflow.
- Projects turn knowledge into skill — they prove you can apply concepts to solve real problems.

# Project Overview: Spatial Query Lab

The **Spatial Query Lab** is an interactive platform that allows students to learn spatial SQL concepts through real-time experimentation.

It connects a **PostGIS-enabled Supabase database (backend)** with a **web-based visualization interface (frontend)**.



A screenshot of a GitHub repository page for "SpatialQueryLab". The repository is public and has 1 star, 0 forks, and 1 commit. The README file has been edited recently. The repository description explains it's an interactive geospatial database and SQL learning environment for TAMU students, integrating ArcGIS FeatureServer and Supabase/PostGIS.

**SpatialQueryLab** Public

Pin Watch Fork Starred

main · 1 Branch · 0 Tags Go to file Add file Code

**rayford295** Revise README for clarity and conciseness · 1fedafe · 2 hours ago · 15 Commits

README.md · Revise README for clarity and conciseness · 2 hours ago

setup\_open.sql · Enhance setup\_open.sql with indexes and function update · 2 hours ago

spatial\_query\_lab\_fixed.html · Add files via upload · 2 hours ago

**README**

About

An interactive geospatial database and SQL learning environment for TAMU students. Integrates ArcGIS FeatureServer and Supabase/PostGIS for real data exploration.

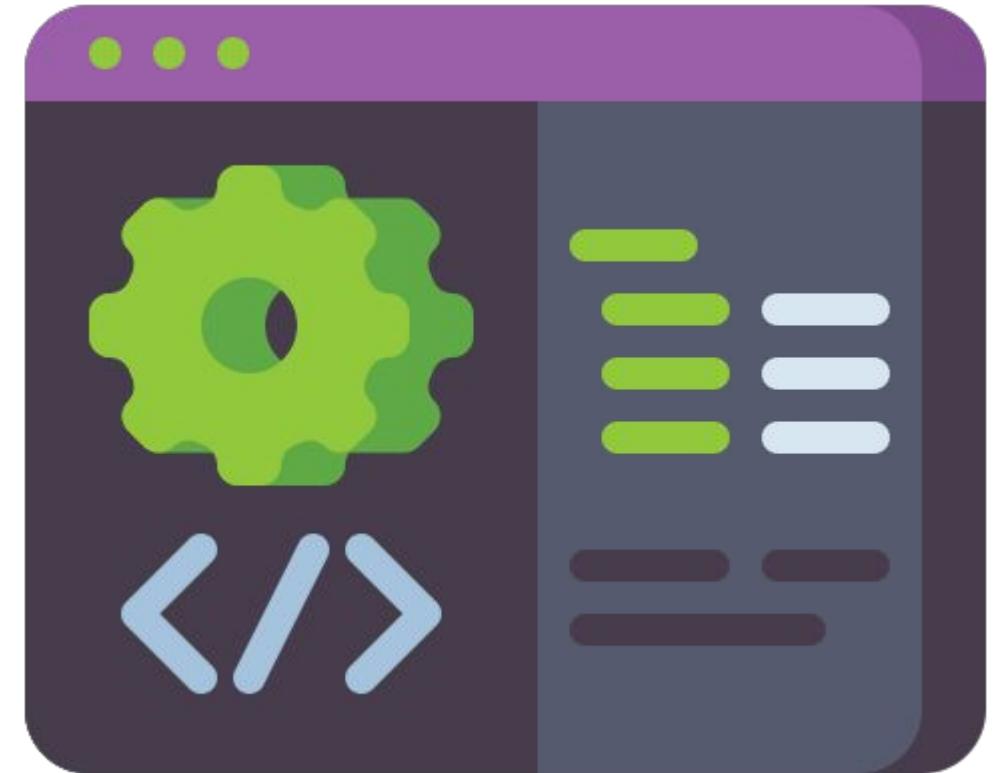
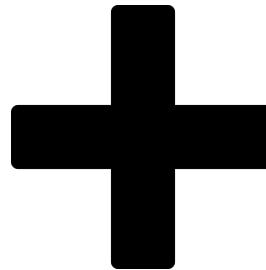
Readme · Activity · 1 star · 0 watching · 0 forks

<https://github.com/rayford295/SpatialQueryLab>

**Project = Frontend + Backend**

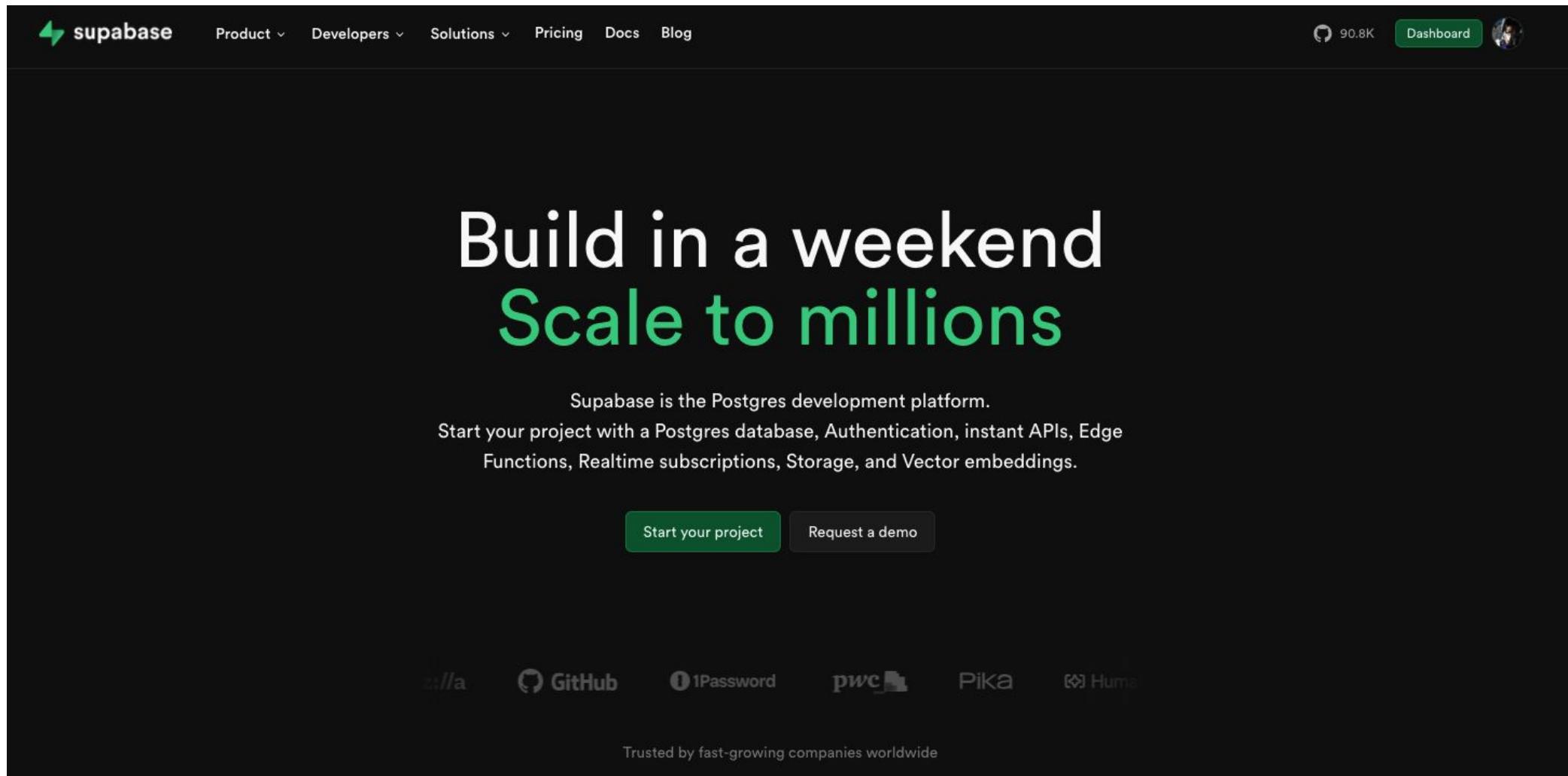


**Frontend (webGIS)**



**Backend (Geodatabase)**

# Backend Design – Supabase + PostGIS



The image shows the Supabase homepage. At the top, there is a navigation bar with links for Product, Developers, Solutions, Pricing, Docs, and Blog. On the right side of the nav bar are icons for user profile, dashboard, and notifications (90.8K). Below the nav bar, the main headline reads "Build in a weekend Scale to millions". A sub-headline below it says "Supabase is the Postgres development platform. Start your project with a Postgres database, Authentication, instant APIs, Edge Functions, Realtime subscriptions, Storage, and Vector embeddings." There are two buttons at the bottom: "Start your project" (green) and "Request a demo" (white). At the very bottom, there's a footer section with logos for Zalando, GitHub, 1Password, PwC, PiKa, and Huma, followed by the text "Trusted by fast-growing companies worldwide".

supabase

Product ▾ Developers ▾ Solutions ▾ Pricing Docs Blog

90.8K Dashboard

# Build in a weekend Scale to millions

Supabase is the Postgres development platform.  
Start your project with a Postgres database, Authentication, instant APIs, Edge Functions, Realtime subscriptions, Storage, and Vector embeddings.

Start your project Request a demo

zalando GitHub 1Password PwC PiKa Huma

Trusted by fast-growing companies worldwide

<https://github.com/supabase/supabase>

# Backend Design – Supabase + PostGIS

Hosted on **Supabase**, an open-source PostgreSQL service with built-in PostGIS extension.

Stores geospatial datasets such as *Campus Safety Points* and *Campus Boundaries*.

Supports spatial SQL functions including `ST_Within`, `ST_Intersects`, `ST_Buffer`, and `ST_Distance`.

Provides a **RESTful API endpoint** for the frontend to query and return geoJSON results.



On the right, the **SQL Editor** is used to set up and populate spatial tables. Here, two key tables are created:

1. **campus\_safety** — stores campus safety features (e.g., emergency call boxes, cameras) as **Point geometries**.
2. **campus\_buildings** — stores building footprints as **Polygon geometries**.

The image shows the Supabase dashboard interface. On the left, there's a sidebar with various project management and database-related icons. The main area has tabs for 'Org' (Free), 'Project' (Production), and 'main'. Below these are sections for 'Authentication', 'Storage', 'Edge Functions', and 'Realtime', each with 'Explore' and 'About' buttons. A central section titled 'Connecting to your new project' provides instructions on how to interact with the database via 'Project API' or 'View API settings'. The 'Project API' section includes a 'Project URL' (a RESTful endpoint) and an 'API Key' (an example value: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e...). Below this, code snippets for 'Javascript' and 'Dart' are provided for connecting to the database. On the right side of the dashboard, a red box highlights the 'SQL Editor' section. This section contains the following SQL code:

```
1 -- setup_open.sql
2 -- PostGIS teaching dataset + open SQL RPC
3 -- WARNING: run_open_sql executes arbitrary SQL. Use only for teaching / sandbox.
4
5 -- 0) Enable PostGIS
6 create extension if not exists postgis;
7
8 --
9 -- 1) Points: campus safety features
10 --
11 create table if not exists public.campus_safety (
12     id serial primary key,
13     name text,
14     type text,
15     geom geometry(Point, 4326)
16 );
17
18 truncate table public.campus_safety;
19 insert into public.campus_safety (name, type, geom) values
20 ('Emergency Call Box - Zachry', 'call_box', st_setsrid(st_point(-96.3407, 30.6197),4326)),
21 ('Security Camera - Library', 'camera', st_setsrid(st_point(-96.3389, 30.6181),4326)),
22 ('Emergency Call Box - Kyle', 'call_box', st_setsrid(st_point(-96.3397, 30.6107),4326)),
23 ('Security Camera - MSC', 'camera', st_setsrid(st_point(-96.3419, 30.6128),4326));
24
25 --
26 -- 2) Buildings (polygons)
27 --
28 create table if not exists public.campus_buildings (
29     id serial primary key,
30     bldg_name text,
31     dept text,
32     geom geometry(Polygon, 4326)
33 );
34
35 truncate table public.campus_buildings;
36 insert into public.campus_buildings (bldg_name, dept, geom) values
run_open_sql
{"?column?":1}
```

This page shows the **Table Editor** in Supabase, where the project's spatial data are stored and managed. Here, the **campus\_safety** table contains the key geospatial features used in the Spatial Query Lab.

The screenshot displays two tables in the Supabase Table Editor:

**campus\_safety** table (highlighted by a red box):

	id	name	type	geom
	19	Emergency Call Box - Zachry	call_box	0101000020E6100000D
	20	Security Camera - Library	camera	0101000020E61000005
	21	Emergency Call Box - Kyle	call_box	0101000020E6100000E
	22	Security Camera - MSC	camera	0101000020E61000002

**geometry\_columns** table (highlighted by a red box):

	f_table_catalog	f_table_schema	f_table_name	f_geometry_column	coord_dimension	srid	type
	postgres	public	campus_safety	geom	2	4326	POINT
	postgres	public	green_spaces	geom	2	4326	POLYGON
	postgres	public	roads	geom	2	4326	LINESTRING
	postgres	public	campus_buildings	geom	2	4326	POLYGON

The **frontend interface** connects to the Supabase backend using the **Project URL** and **anon API key** shown earlier. This enables students to directly run spatial SQL queries and visualize the results interactively on a web map.

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**

Supabase Project URL  
<https://hypqkovnppcwvhmxmtm.supabase.co>

Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpc3MiOiJzdXl

**Connect Supabase** Connected ✓

Backend expects Postgres function  
`run_open_sql(sql_text)` to execute arbitrary SQL.

**2) SQL Editor**

Click the map to set variables `{{lon}}`, `{{lat}}`,  
`{{dist_m}}` (default 200).

```
select id, name, type,  
       st_asgeojson(geom)::jsonb as geom  
  from campus_safety  
 limit 10;
```

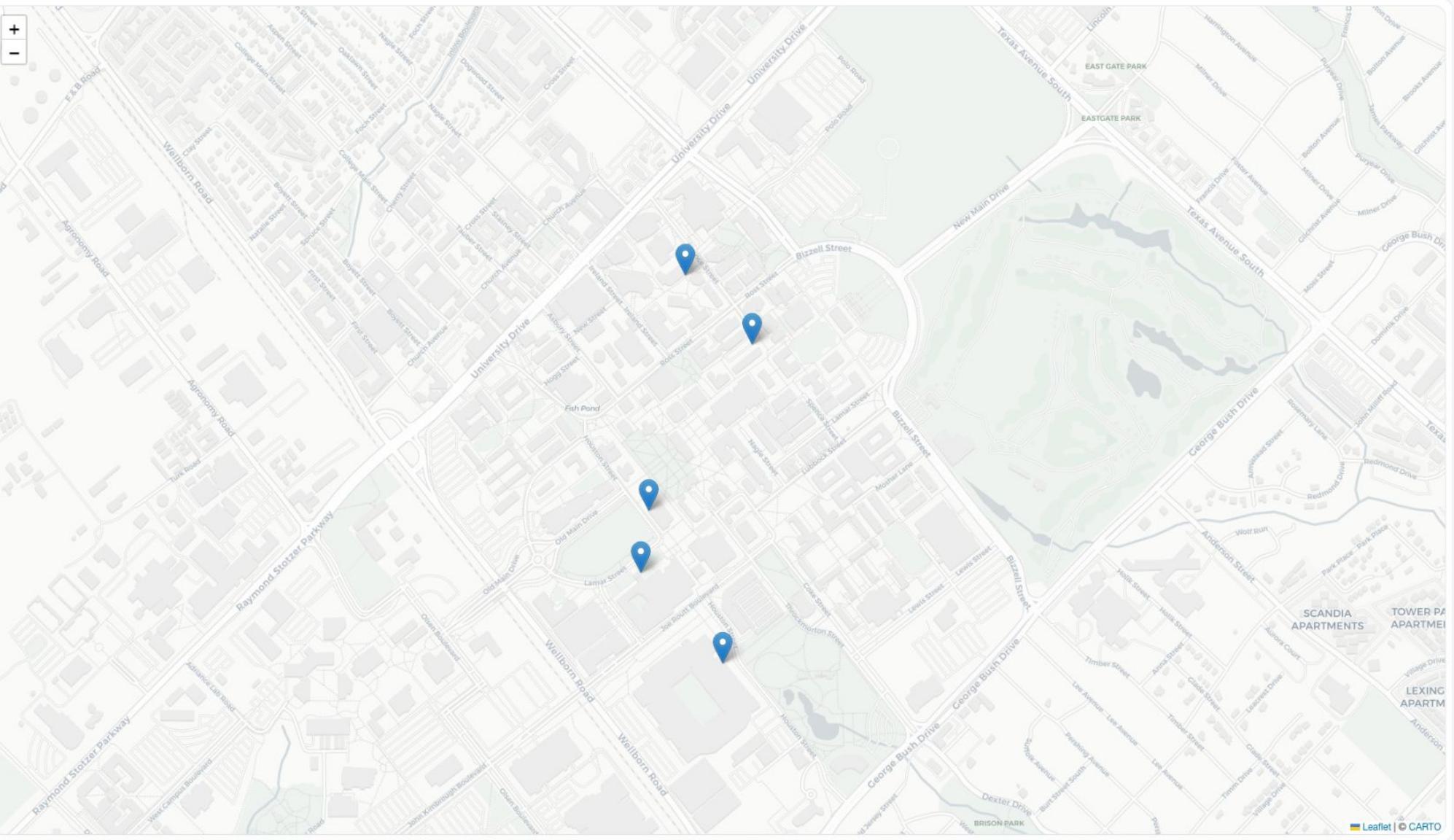
**Run SQL** **Clear Results**

Click set → lat=30.61424, lon=-96.341686, dist=200m

**3) Results**

id	geom	name	type
19	{"type": "Point", "coordinates": [-96.3407, 30.6197]}	Emergency Call Box - Zachry	call_box
20	{"type": "Point", "coordinates": [-96.3389, 30.6181]}	Security Camera - Library	camera
21	{"type": "Point", "coordinates": [-96.3397, 30.6107]}	Emergency Call Box - Kyle	call_box
22	{"type": "Point", "coordinates": [-96.3419, 30.6128]}	Security Camera - MSC	camera

4 row(s)



Leaflet | CARTO

# Frontend Design – HTML + Leaflet.js + Supabase API

- Built as a **web-based sandbox (HTML + JavaScript)** for interactive learning.
- Students can:
  1. Enter spatial SQL queries directly in a text editor panel.
  2. Send the SQL query to Supabase through the **anon key** and project URL.
  3. Visualize the query results dynamically on a **Leaflet.js map**.
- Includes buttons for “Connect”, “Run Query”, and “Clear Results”.

## Tech Stack:

HTML + JavaScript + Supabase.js + Leaflet + CSS

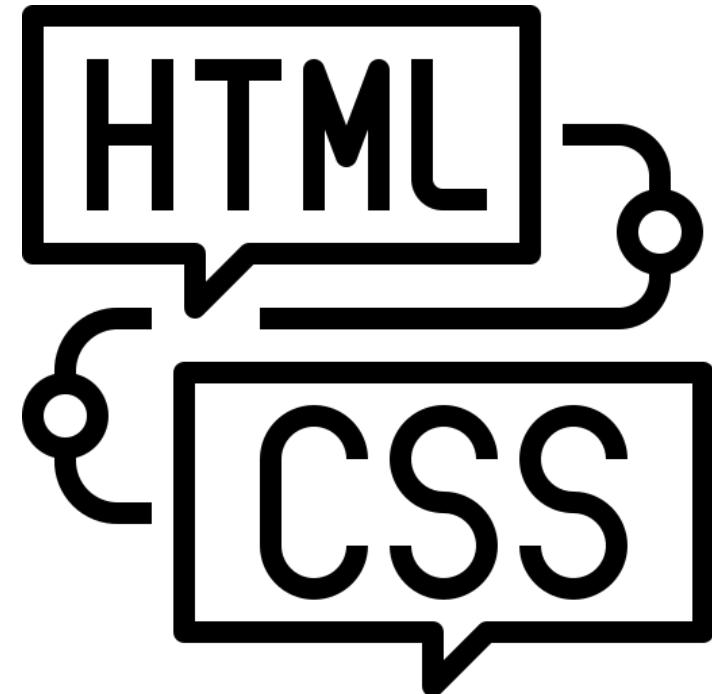
HTML → Structure

CSS → Style

JavaScript → Interaction

Supabase.js → Database connection

Leaflet.js → Map visualization



This interface shows the **Spatial Query Lab (v3)** — an interactive web application designed for learning and practicing **spatial SQL queries** using a real geospatial database.

**Write SQL commands** — in the **SQL Editor** section, students can perform spatial queries such as selecting, buffering, or filtering geographic features.

**View Query Results** — the system outputs query results in JSON format, listing spatial geometries (**geom**) and attributes (**name**, **type**).

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://hypqkovpppcvhmxmtr.supabase.co>  
Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cI6IkpxVCJ9.eyJpc3MiOiJzdXI

**Connect Supabase** Connected ✓  
Backend expects Postgres function  
`run_open_sql(sql_text)` to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables `{{lon}}`, `{{lat}}`, `{{dist_m}}`. (default 200).  

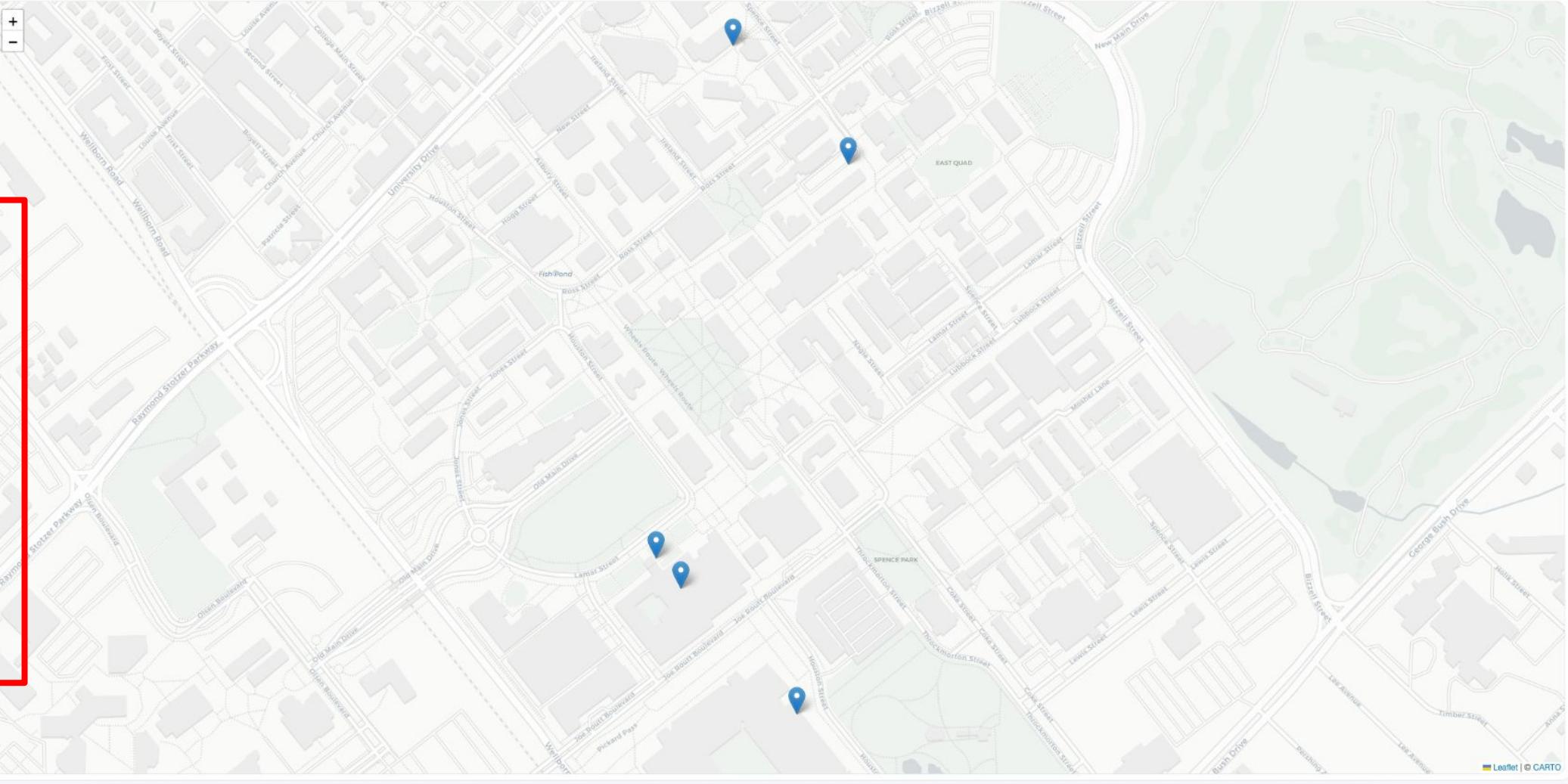
```
select id, name, type,
       st_asgeojson(geom)::jsonb as geom
  from campus_safety
 limit 20;
```

**Run SQL** **Clear Results**  
Click set → lat=30.612394, lon=-96.341515, dist=200m

**3) Results**

id	geom	name	type
19	{"type": "Point", "coordinates": [-96.3407, 30.6197]}	Emergency Call Box - Zachry	call_box
20	{"type": "Point", "coordinates": [-96.3389, 30.6181]}	Security Camera - Library	camera
21	{"type": "Point", "coordinates": [-96.3397, 30.6107]}	Emergency Call Box - Kyle	call_box
22	{"type": "Point", "coordinates": [-96.3419, 30.6128]}	Security Camera - MSC	camera

4 row(s)



Leaflet | © GARTO

## 1. View All Campus Safety Features

```
select id, name, type,  
       st_asgeojson(geom)::jsonb as geom  
  from campus_safety  
 limit 20;
```

**Purpose:** Basic query to verify table structure and visualize all points.

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://hypgk0vppcwvhmxmtm.supabase.co>  
Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJpc3MIoIjzdxI

**Connect Supabase** Connected ✓  
Backend expects Postgres function  
run\_open\_sql(sql\_text) to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables: {{lon}}, {{lat}}, {{dist\_m}} (default 200).  

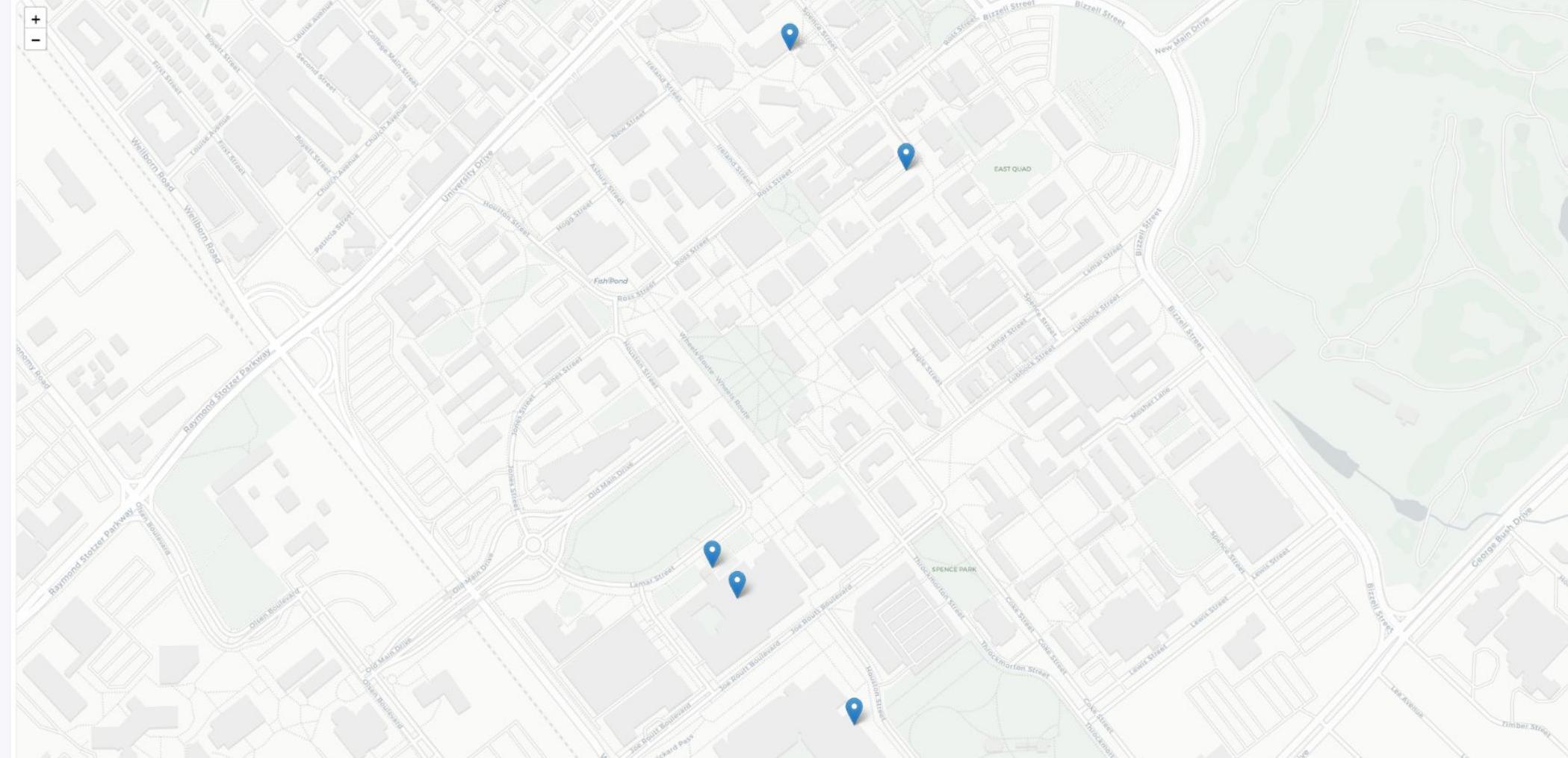
```
select id, name, type,  
       st_asgeojson(geom)::jsonb as geom  
  from campus_safety  
 limit 20;
```

**Run SQL** Clear Results  
Click set → lat=30.612394, lon=-96.341515, dist=200m

**3) Results**

id	geom	name	type
19	{"type": "Point", "coordinates": [-96.3407, 30.6197]}	Emergency Call Box - Zachry	call_box
20	{"type": "Point", "coordinates": [-96.3389, 30.6181]}	Security Camera - Library	camera
21	{"type": "Point", "coordinates": [-96.3397, 30.6107]}	Emergency Call Box - Kyle	call_box
22	{"type": "Point", "coordinates": [-96.3419, 30.6128]}	Security Camera - MSC	camera

4 row(s)



## 2. Buffer Around Each Point (50 meters)

```
select id, name,
       st_asgeojson(st_buffer(geom::geography,
50)::geometry)::jsonb as geom
from campus_safety;
```

**Purpose:** Creates circular buffers around each feature to visualize coverage zones.

2) SQL Editor

Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).

```
select id, name,
       st_asgeojson(st_buffer(geom::geography,
50)::geometry)::jsonb as geom
from campus_safety;
```

**Run SQL**   **Clear Results**

Click set → lat=30.612394, lon=-96.341515, dist=200m

3) Results

id	geom	name
19	{"type": "Polygon", "coordinates": [[[-96.34017887, 30.61968933], [-96.340191295, 30.619601614], [-96.340223263, 30.619517673], [-96.340273552, 30.619440738], [-96.340340229, 30.619373767], [-96.340420731, 30.61931933], [-96.34051965, 30.619279527], [-96.340610426, 30.61925588], [-96.340712328, 30.6192493], [-96.340813757, 30.61926004], [-96.34091088, 30.619287687], [-96.3409997, 30.6193118], [-96.341077206, 30.619388846], [-96.34140147, 30.619458469], [-96.34186173, 30.619537374], [-96.341213515, 30.619622529], [-96.341221124, 30.619710661], [-96.341208706, 30.619798384], [-96.341176739, 30.619882325], [-96.34112645, 30.61995926], [-96.341059774, 30.620026232], [-96.340979271, 30.620080667], [-96.340888036, 30.620120473], [-96.340789575, 30.62014412], [-96.340687671, 30.62015057], [-96.340586242, 30.62013996], [-96.340489184, 30.620112312], [-96.340400227, 30.620068819], [-96.340322791, 30.620011153], [-96.340259951, 30.619941529], [-96.340213826, 30.619862624], [-96.340186484, 30.619777469], [-96.340178877, 30.619689337]]]}	Emergency Call Box - Zachry
20	{"type": "Polygon", "coordinates": [[[-96.33237888, 30.61808933], [-96.338391306, 30.618001608], [-96.338423275, 30.617917667], [-96.338473564, 30.617840733], [-96.338540241, 30.617773763], [-96.3386210742, 30.617719329], [-96.338711976, 30.61769525], [-96.338810435, 30.617655879], [-96.338912336, 30.6176493], [-96.339013763, 30.617660042]]]}	Security Camera - Library

Leaflet | © CARTO

### 3. Find Features Within a Distance (Click-to-Query)

```
select id, name, type,  
       st_asgeojson(geom)::jsonb as geom  
from campus_safety  
where st_dwithin(  
    geom::geography,  
    st_setsrid(st_point({{lon}}, {{lat}}), 4326)::geography,  
    {{dist_m}}  
) ;
```

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL:  
<https://hypqkovppowhmxxmt.supabase.co>  
Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpc3MiOiJzdXI  
  
Connect Supabase Connected ✓  
Backend expects Postgres function  
run\_open\_sql(sql\_text) to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).  

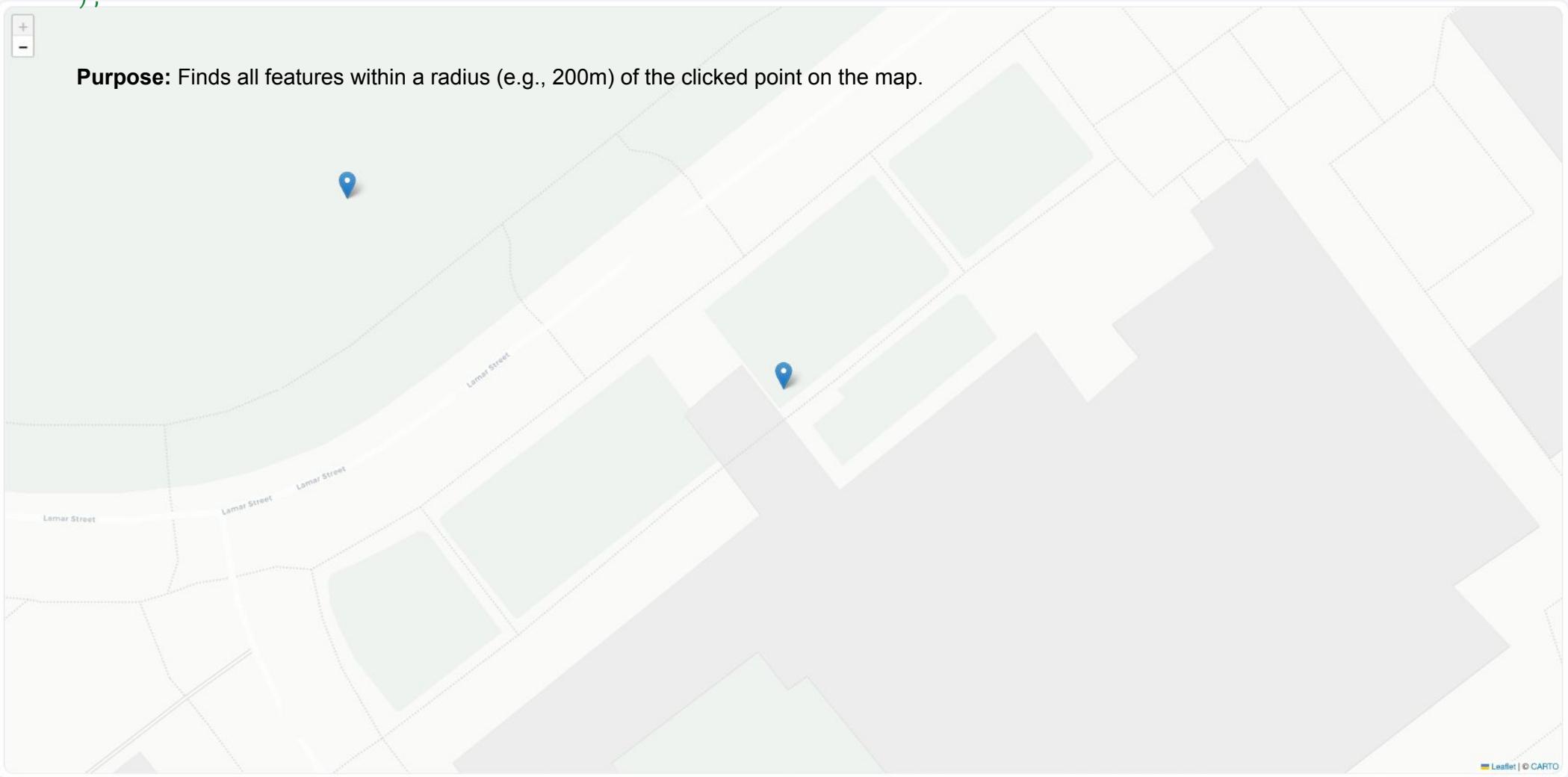
```
select id, name, type,  
       st_asgeojson(geom)::jsonb as geom  
from campus_safety  
where st_dwithin(  
    geom::geography,  
    st_setsrid(st_point({{lon}}, {{lat}}), 4326)::geography,  
    {{dist_m}}  
) ;
```

  
**Run SQL** **Clear Results**  
Click set → lat=30.613123, lon=-96.342759, dist=200m

**3) Results**  

id	geom	name	type
22	{"type": "Point", "coordinates": [-96.3419, 30.6128]}	Security Camera - MSC	camera

  
1 row(s)



#### 4. Find the Nearest Feature to a Clicked Point

```
select id, name, type,
       st_distance(
           geom::geography,
           st_setsrid(st_point({{lon}}, {{lat}}), 4326)::geography
       ) as dist_m,
       st_asgeojson(geom)::jsonb as geom
  from campus_safety
 order by geom <-> st_setsrid(st_point({{lon}}, {{lat}}), 4326)
 limit 1;
```

Spatial Query Lab v3-fixed

1) Supabase (SQL Runner)

Supabase Project URL  
<https://hypqkovnppcwvhmxxtm.supabase.co>

Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpc3MiOiJzdXI

Connect Supabase Connected ✓

Backend expects Postgres function  
run\_open\_sql(sql\_text) to execute arbitrary SQL.

2) SQL Editor

Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).

```
select id, name, type,
       st_distance(
           geom::geography,
           st_setsrid(st_point({{lon}}, {{lat}}), 4326)::geography
       ) as dist_m,
       st_asgeojson(geom)::jsonb as geom
  from campus_safety
 order by geom <->
```

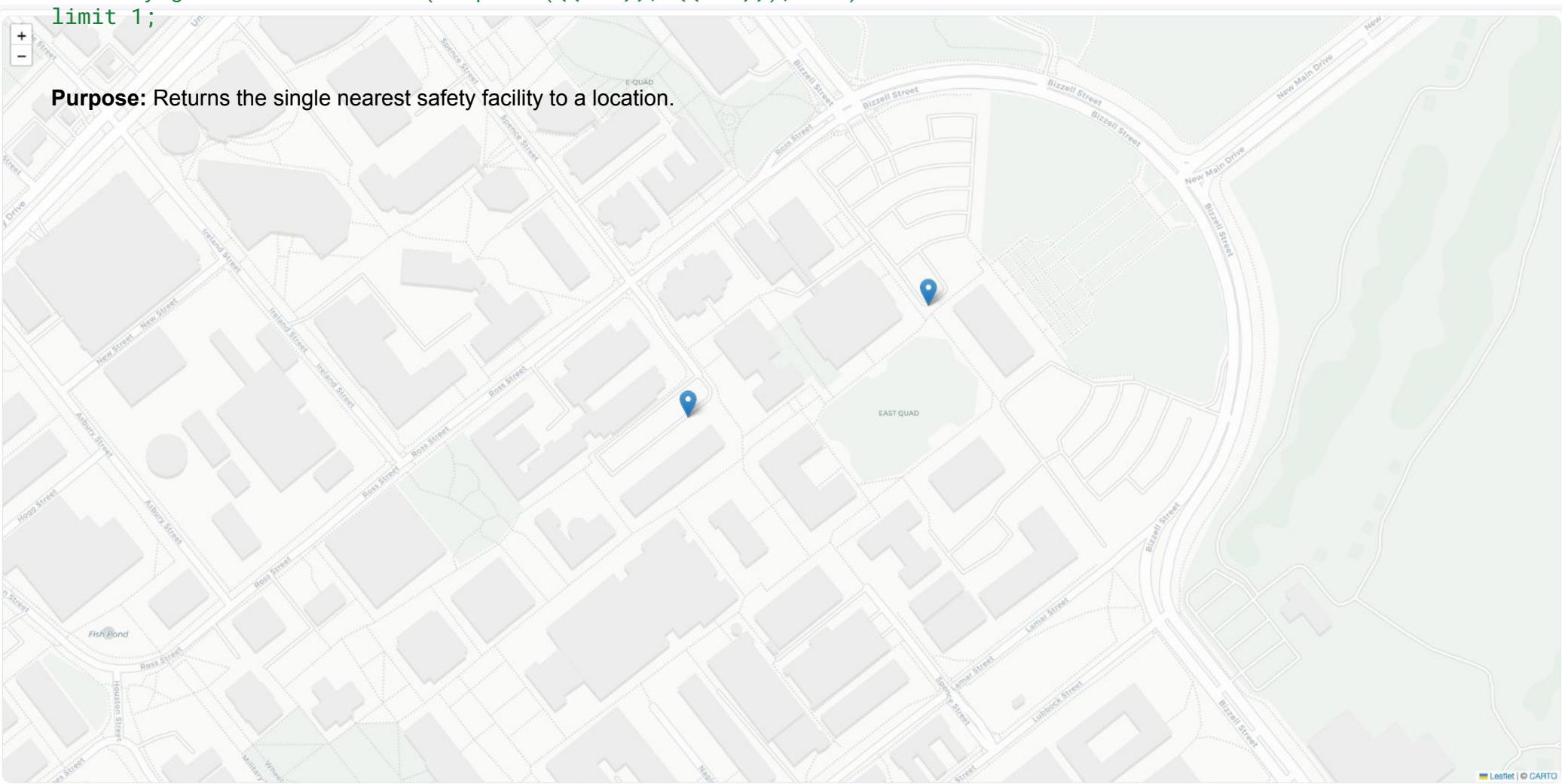
Run SQL Clear Results

Click set → lat=30.618857, lon=-96.337008, dist=200m

3) Results

id	geom	name	type	dist_m
20	{"type": "Point", "coordinates": [-96.3389, 30.6181]}	Security Camera	Camera	199.6
		Library		

1 row(s)



## 5. Intersect Features Between Tables

```
select s.id as safety_id, s.name as safety_name, b.bldg_name,  
       st_asgeojson(s.geom)::jsonb as geom  
  from campus_safety s  
 join campus_buildings b  
  on st_intersects(s.geom, b.geom);
```

**Purpose:** Finds safety features located *within or touching* campus buildings.

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://hypqk0nppcwvhmxmtr.supabase.co>  
Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXI  
  
**Connect Supabase** Connected ✓  
Backend expects Postgres function  
run\_open\_sql(sql\_text) to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).  
  

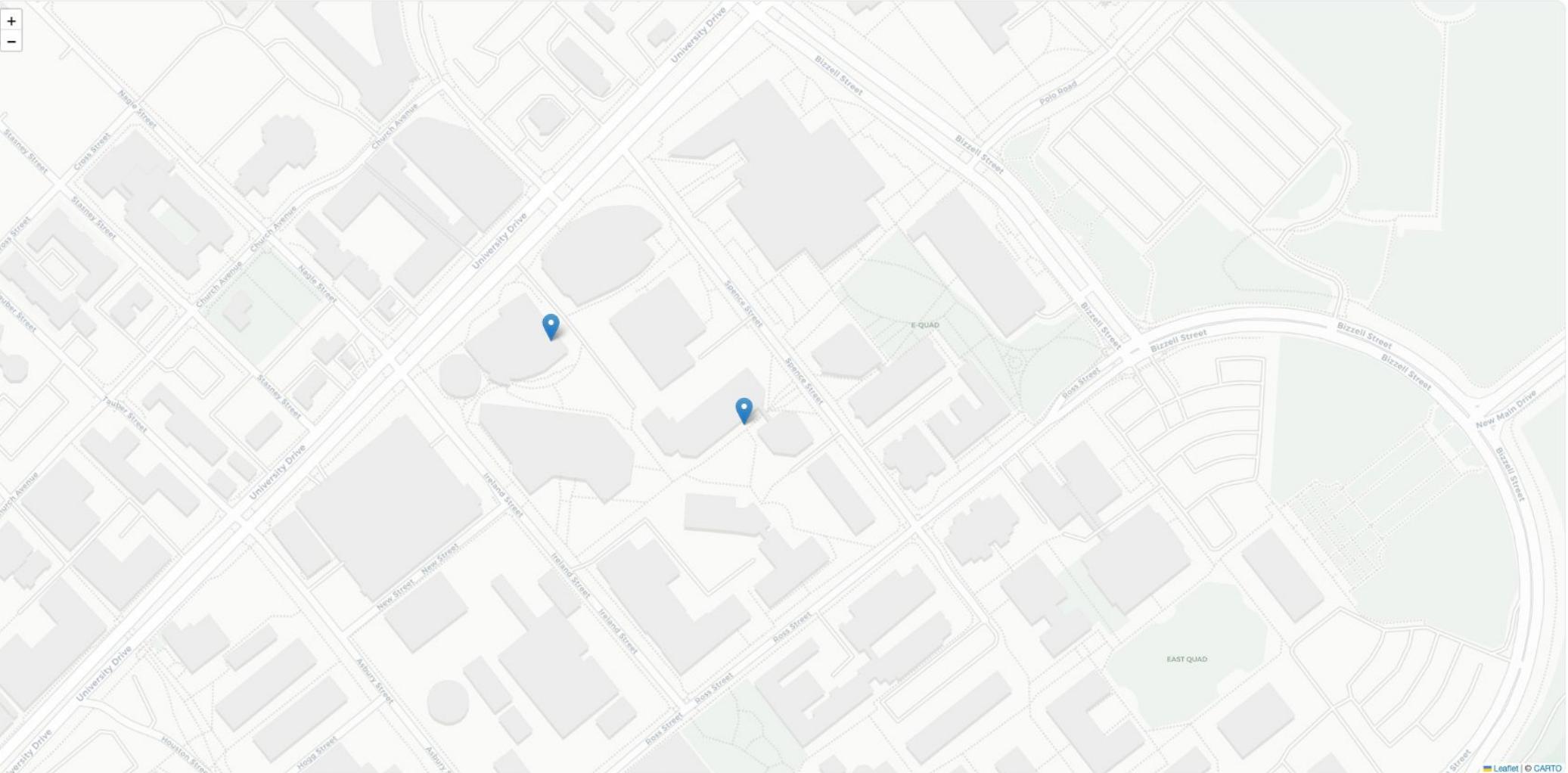
```
select s.id as safety_id, s.name as safety_name, b.bldg_name,  
       st_asgeojson(s.geom)::jsonb as geom  
  from campus_safety s  
 join campus_buildings b  
  on st_intersects(s.geom, b.geom);
```

  
**Run SQL** Clear Results  
Click set + lat=30.620265, lon=-96.342205, dist=200m

**3) Results**

geom	bldg_name	safety_id	safe
{"type": "Point", "coordinates": [-96.3407, 30.6197]}	Zachry Engineering Education Complex	19	Eme Call Zach

1 row(s)



Leaflet | © CARTO

## 6. Count Features Per Type

```
select type, count(*) as feature_count  
from campus_safety  
group by type  
order by feature_count desc;
```

**Purpose:** Aggregates the number of cameras and call boxes for a quick summary.

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://hypqkovppcwvhmxmtm.supabase.co>  
Supabase anon public key  
`eyJhbGciOiJIUzI1NltsInR5cCl6IkpxVCJ9.eyJpc3MiOiJzdXI`  
Connect Supabase  Connected ✓  
Backend expects Postgres function  
`run_open_sql(sql_text)` to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables `{{lon}}`, `{{lat}}`,  
 `{{dist_m}}`  (default 200).  

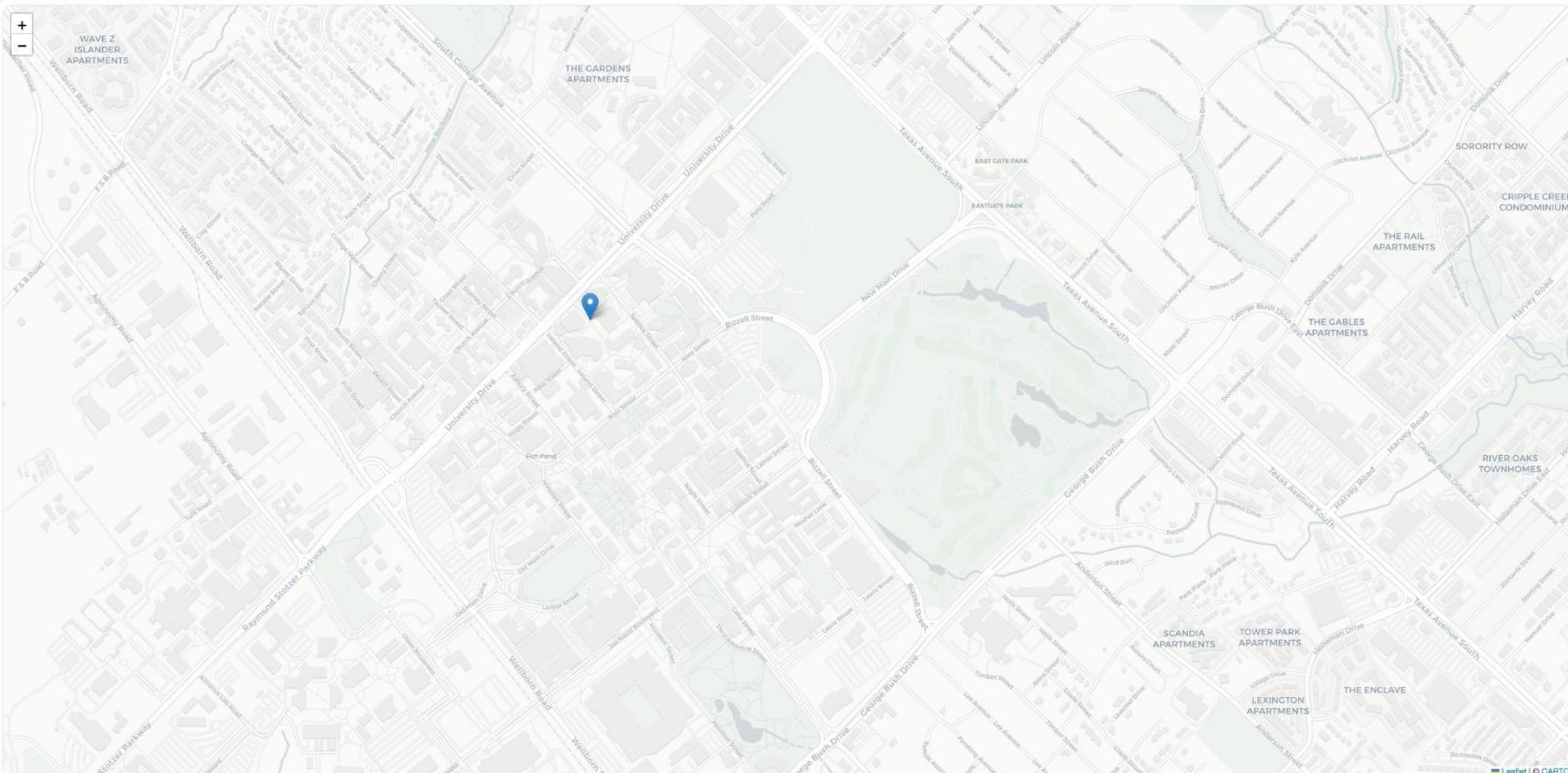
```
select type, count(*) as feature_count  
from campus_safety  
group by type  
order by feature_count desc;
```

  
**Run SQL** **Clear Results**  
Click set → lat=30.620279, lon=-96.341954, dist=200m

**3) Results**

type	feature_count
camera	2
call_box	2

2 row(s)



Leaflet © CARTO

## 7. Clip Buildings by Green Space

```
select b.id, b.bldg_name, g.name as park_name,  
       st_asgeojson(st_intersection(b.geom, g.geom))::jsonb as geom  
from campus_buildings b  
join green_spaces g  
on st_intersects(b.geom, g.geom);
```

**Purpose:** Shows which building footprints overlap with green spaces.

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://hypqk0vppcwhmxmtm.supabase.co>  
Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpc3MiOiJzdH  
Connected

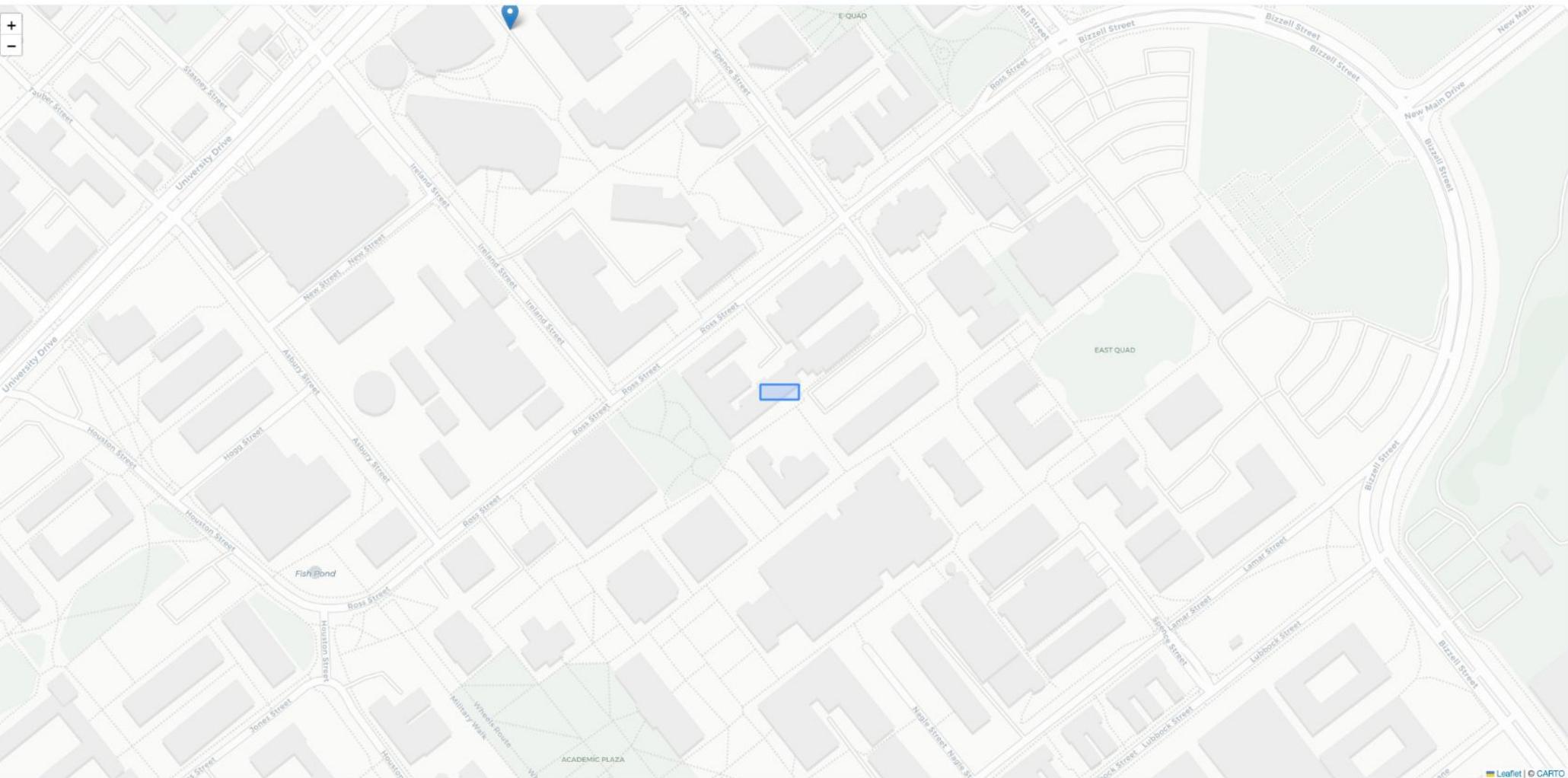
Backend expects Postgres function  
`run_open_sql(sql_text)` to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables: {{lon}}, {{lat}}, {{dist\_m}} (default 200).  
  
select b.id, b.bldg\_name, g.name as park\_name,  
 st\_asgeojson(st\_intersection(b.geom, g.geom))::jsonb as geom  
from campus\_buildings b  
join green\_spaces g  
on st\_intersects(b.geom, g.geom);  
  
**Run SQL** **Clear Results**  
Click set → lat=30.620279, lon=-96.341954, dist=200m

**3) Results**

id	geom	bldg_name	park_name
6	{"type": "Polygon", "coordinates": [[[-96.34, 30.6178], [-96.34, 30.6179], [-96.3397, 30.6179], [-96.3397, 30.6178], [-96.34, 30.6178]]]}}	Evans Library	Simpson Drill Field

1 row(s)



Leaflet | © CARTO

## 8. Compute Distance Between Cameras and Call Boxes

```
select c1.name as camera_name, c2.name as callbox_name,  
       st_distance(c1.geom::geography, c2.geom::geography) as dist_m  
  from campus_safety c1, campus_safety c2  
 where c1.type = 'camera'  
   and c2.type = 'call_box'  
  order by dist_m  
 limit 10;
```

**Purpose:** Measures nearest distances between two different feature types.

Spatial Query Lab v3-fixed

1) Supabase (SQL Runner)

Supabase Project URL  
<https://hypqkovppcwvhmxxtm.supabase.co>

Supabase anon public key  
eyJhbGciOiJIUzI1NlslR5cCl6kpXVCJ9.eyJpc3MiOiJzdXI

Connect Supabase Connected ✓

Backend expects Postgres function  
run\_open\_sql(sql\_text) to execute arbitrary SQL.

2) SQL Editor

Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).

```
select c1.name as camera_name, c2.name as callbox_name,  
       st_distance(c1.geom::geography,  
c2.geom::geography) as dist_m  
  from campus_safety c1, campus_safety c2  
 where c1.type = 'camera'  
   and c2.type = 'call_box'  
  order by dist_m  
 limit 10;
```

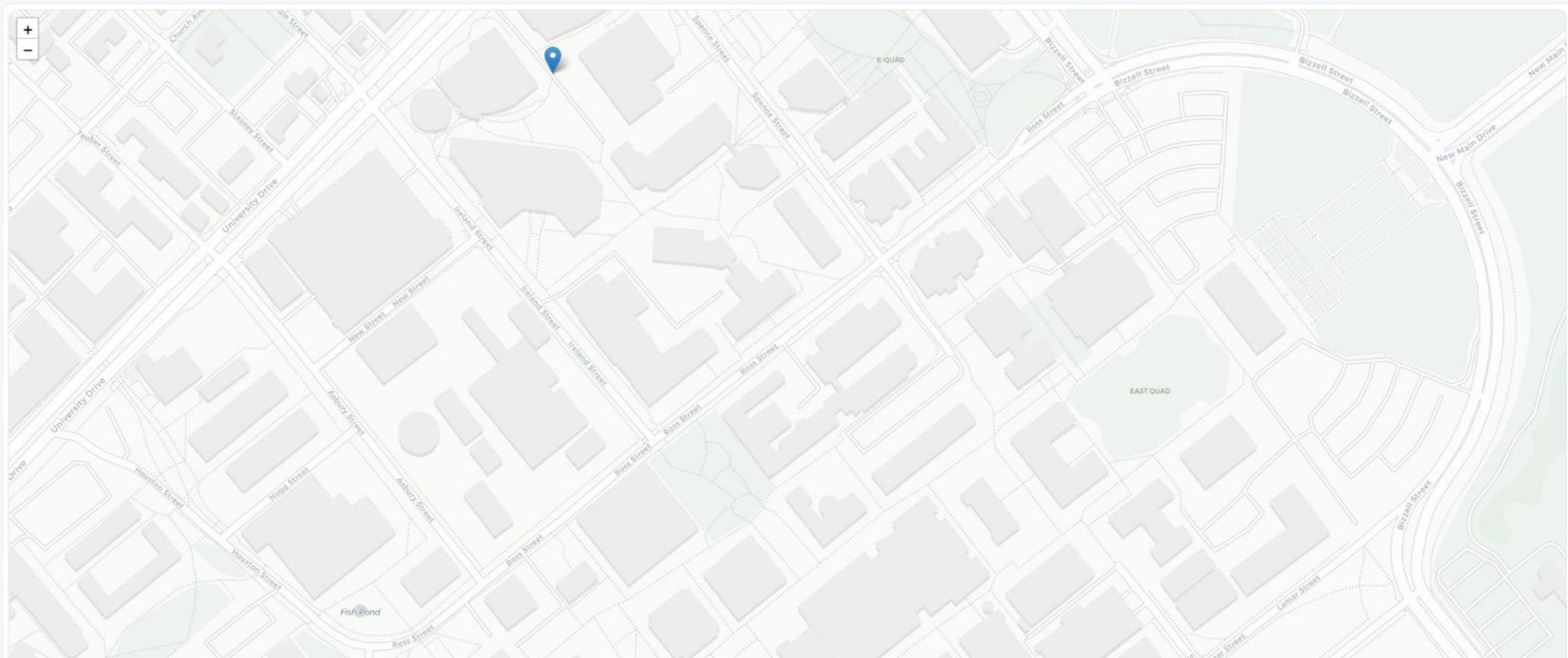
Run SQL Clear Results

Click set → lat=30.620279, lon=-96.341954, dist=200m

3) Results

dist_m	camera_name	callbox_name
247.48811316	Security Camera - Library	Emergency Call Box - Zachry
314.17151267	Security Camera - MSC	Emergency Call Box - Kyle
773.55911606	Security Camera - MSC	Emergency Call Box - Zachry
823.96366442	Security Camera - Library	Emergency Call Box - Kyle

4 row(s)



## 9. Find Buildings Near Roads (Within 30m)

```
select b.id, b.bldg_name, r.road_name,
       st_asgeojson(b.geom)::jsonb as geom
  from campus_buildings b
 join roads r
  on st_dwithin(b.geom::geography, r.geom::geography, 3000);
```

**Purpose:** Identifies buildings located close to major roads.

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://hypqkovanppcwihmxmtm.supabase.co>  
Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpc3MiOiJzdX

**Connect Supabase** Connected ✓  
Backend expects Postgres function  
run\_open\_sql(sql\_text) to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).  

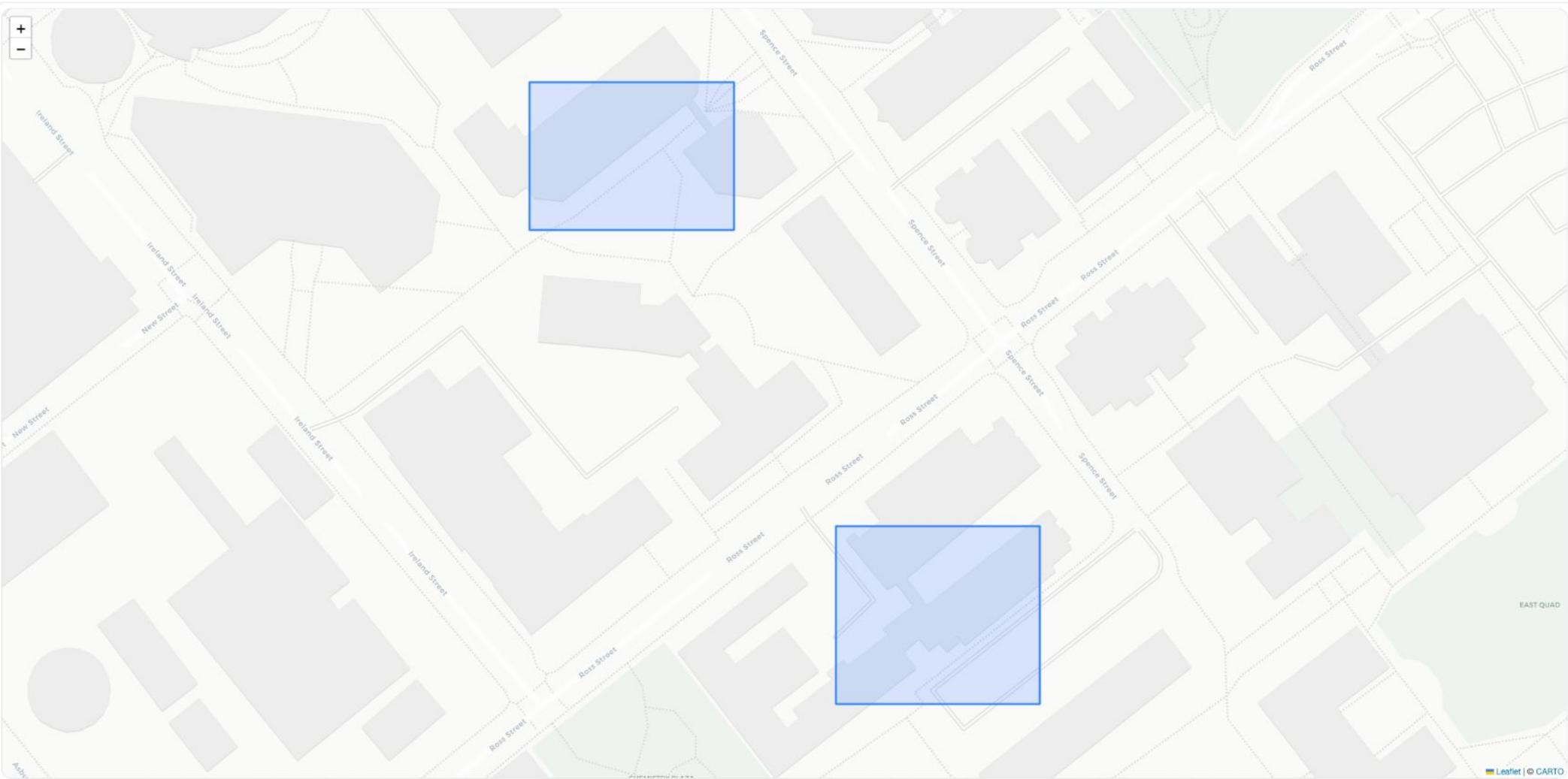
```
select b.id, b.bldg_name, r.road_name,
       st_asgeojson(b.geom)::jsonb as geom
  from campus_buildings b
 join roads r
  on st_dwithin(b.geom::geography,
r.geom::geography, 3000);
```

**Run SQL** Clear Results  
Click set → lat=30.618825, lon=-96.343371, dist=200m

**3) Results**

id	geom	bldg_name	road_name
5	{"type": "Polygon", "coordinates": [[[-96.3412, 30.6199], [-96.3404, 30.6199], [-96.3404, 30.6194], [-96.3412, 30.6194], [-96.3412, 30.6199]]]}	Zachry Engineering Complex	George Bush Dr
6	{"type": "Polygon", "coordinates": [[[{-96.34, 30.6184}, {-96.3392, 30.6184}, {-96.3392, 30.6178}, {-96.34, 30.6178}, {-96.34, 30.6184}]]]}	Evans Library	George Bush Dr

2 row(s)



Leaflet | © CARTO

## 10. Create Combined GeoJSON Layers

```
select 'campus_safety' as layer, st_asgeojson(geom)::jsonb as geom from campus_safety
union all
select 'campus_buildings' as layer, st_asgeojson(geom)::jsonb as geom from campus_buildings
union all
select 'roads' as layer, st_asgeojson(geom)::jsonb as geom from roads;
```

**Purpose:** Combines multiple datasets into a single map layer for visualization.

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL: <https://hypqkovnppcwhmxmtm.supabase.co>  
Supabase anon public key: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJz  
**Connect Supabase** Connected ✓  
Backend expects Postgres function run\_open\_sql({sq\_l\_text}) to execute arbitrary SQL.

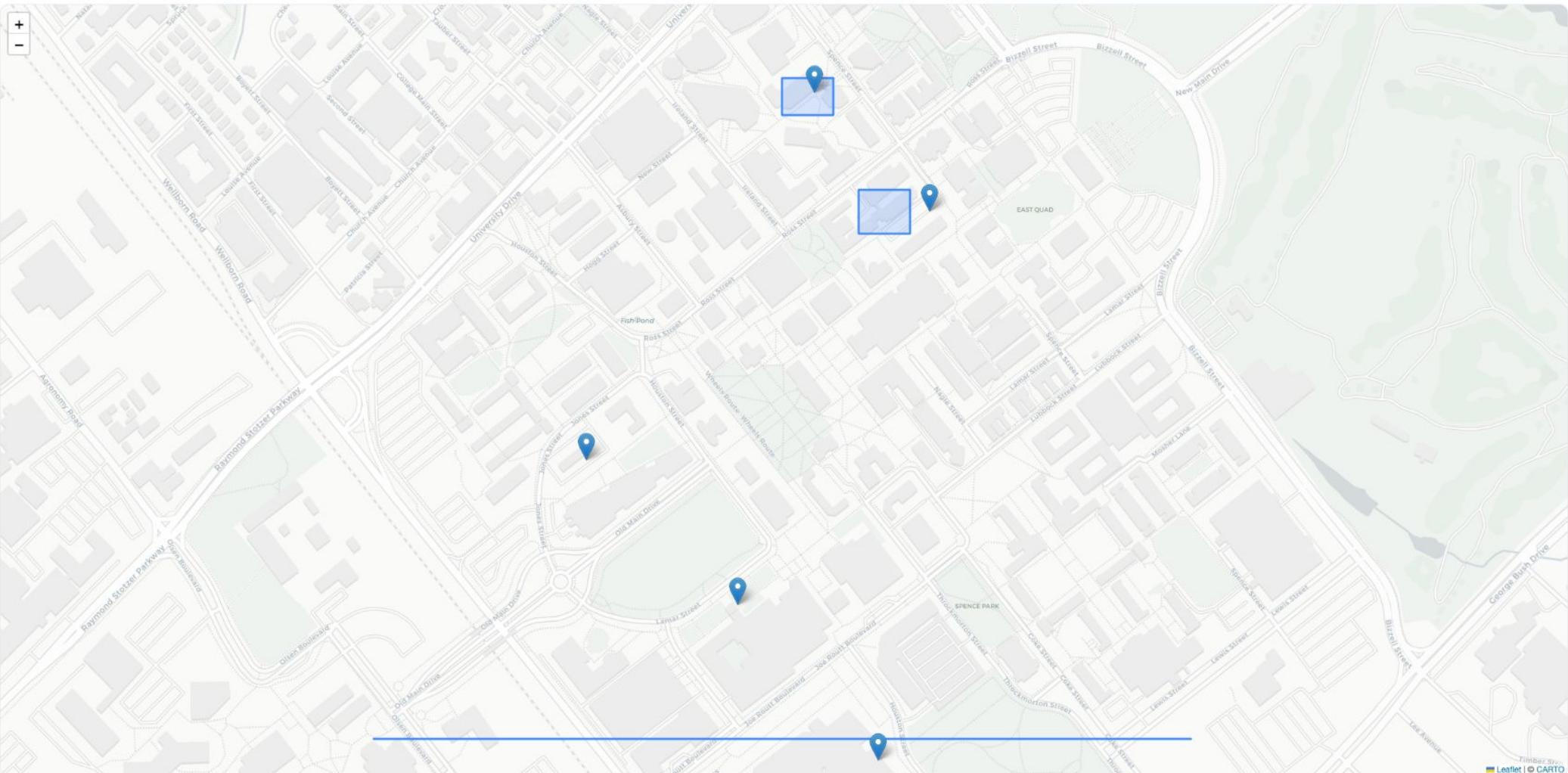
**2) SQL Editor**  
Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).  

```
select 'campus_safety' as layer,
st_asgeojson(geom)::jsonb as geom from
campus_safety
union all
select 'campus_buildings' as layer,
st_asgeojson(geom)::jsonb as geom from
campus_buildings
union all
select 'roads' as layer,
```

**Run SQL** Clear Results  
Click set → lat=30.614748, lon=-96.344272, dist=200m

**3) Results**

geom	layer
{"type": "Point", "coordinates": [-96.3407, 30.6197]}	campus_safety
{"type": "Point", "coordinates": [-96.3389, 30.6181]}	campus_safety
{"type": "Point", "coordinates": [-96.3397, 30.6107]}	campus_safety
{"type": "Point", "coordinates": [-96.3419, 30.6128]}	campus_safety
{"type": "Polygon", "coordinates": [[[-96.3412, 30.6199], [-96.3404, 30.6199], [-96.3404, 30.6194], [-96.3412, 30.6194], [-96.3412, 30.6199]]]}}	campus_buildings
{"type": "Polygon", "coordinates": [[[ -96.34, 30.6184], [-96.3392, 30.6184], [-96.3392, 30.6178], [-96.34, 30.6178], [-96.34, 30.6184]]]}}	campus_buildings
{"type": "LineString", "coordinates": [-96.3476, 30.6111]}	roads



The map displays a detailed street grid of a university campus. Buildings are represented by grey polygons, roads by black lines, and safety points by blue squares with a white dot. The map includes labels for various streets and buildings, such as University Drive, Ross Street, and the EAST QUAD. A legend in the top left corner shows icons for buildings, roads, and safety points.

## 11. Heatmap Preparation – Count Safety Features per Building

```
select b.bldg_name, count(s.id) as safety_count,  
       st_asgeojson(b.geom)::jsonb as geom  
  from campus_buildings b  
left join campus_safety s  
    on st_within(s.geom, b.geom)  
 group by b.bldg_name, b.geom  
order by safety_count desc;
```

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://hypckovnppcwhmxmtm.supabase.co>  
Supabase anon public key  
eyJhbGciOiJUzI1NlIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdjM...  
**Connect Supabase** Connected   
Backend expects Postgres function  
`run_open_sql(sql_text)` to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables `{{lon}}`, `{{lat}}`, `{{dist_m}}` (default 200).  

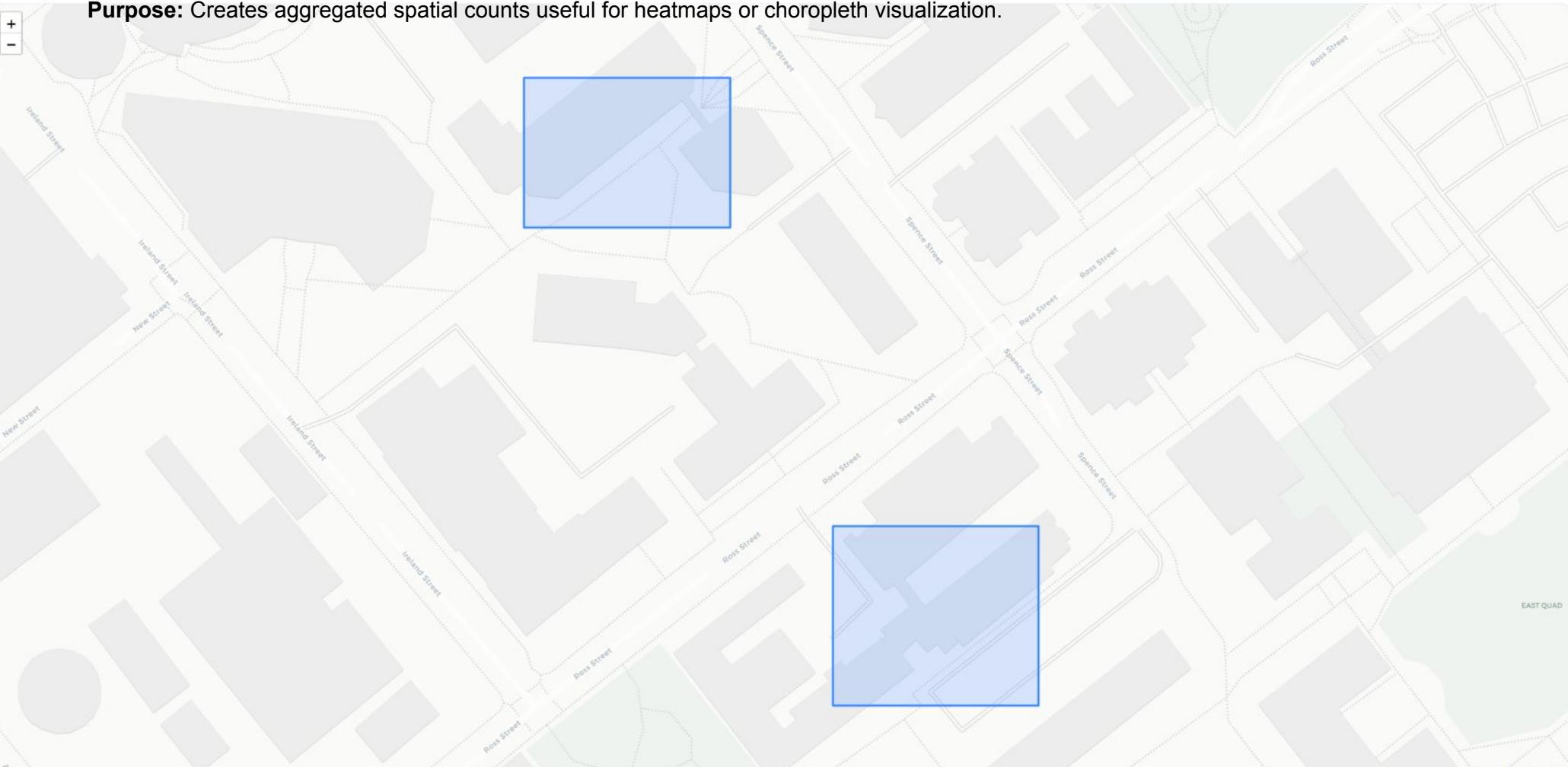
```
select b.bldg_name, count(s.id) as safety_count,  
       st_asgeojson(b.geom)::jsonb as geom  
  from campus_buildings b  
left join campus_safety s  
    on st_within(s.geom, b.geom)  
 group by b.bldg_name, b.geom  
order by safety_count desc;
```

**Run SQL** **Clear Results**  
Click set → lat=30.617675, lon=-96.34704, dist=200m

**3) Results**

geom	bldg_name	safety_count
{"type": "Polygon", "coordinates": [[[-96.3412, 30.6199], [-96.3404, 30.6199], [-96.3404, 30.6194], [-96.3412, 30.6194], [-96.3412, 30.6199]]]}	Zachry Engineering Education Complex	1
{"type": "Polygon", "coordinates": [[[-96.343, 30.6184], [-96.3392, 30.6184], [-96.3392, 30.6178], [-96.34, 30.6178], [-96.34, 30.6184]]]}	Evans Library	0

Purpose: Creates aggregated spatial counts useful for heatmaps or choropleth visualization.



Leaflet | © CARTO

## 12. Multi-Condition Query (Spatial + Attribute)

```
select id, name, type,
       st_asgeojson(geom)::jsonb as geom
  from campus_safety
 where type = 'call_box'
   and st_dwithin(
     geom::geography,
     st_setsrid(st_point({{lon}}, {{lat}}), 4326)::geography,
     3000
   );
```

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://ypk0vppcwvhmxmtm.supabase.co>  
Supabase anon public key  
eyJhbGciOiJUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MI0JzdxI

**Connect Supabase** Connected   
Backend expects Postgres function  
`run_open_sql(sql_text)` to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables `{{lon}}`, `{{lat}}`, `{{dist_m}}` (default 200).  

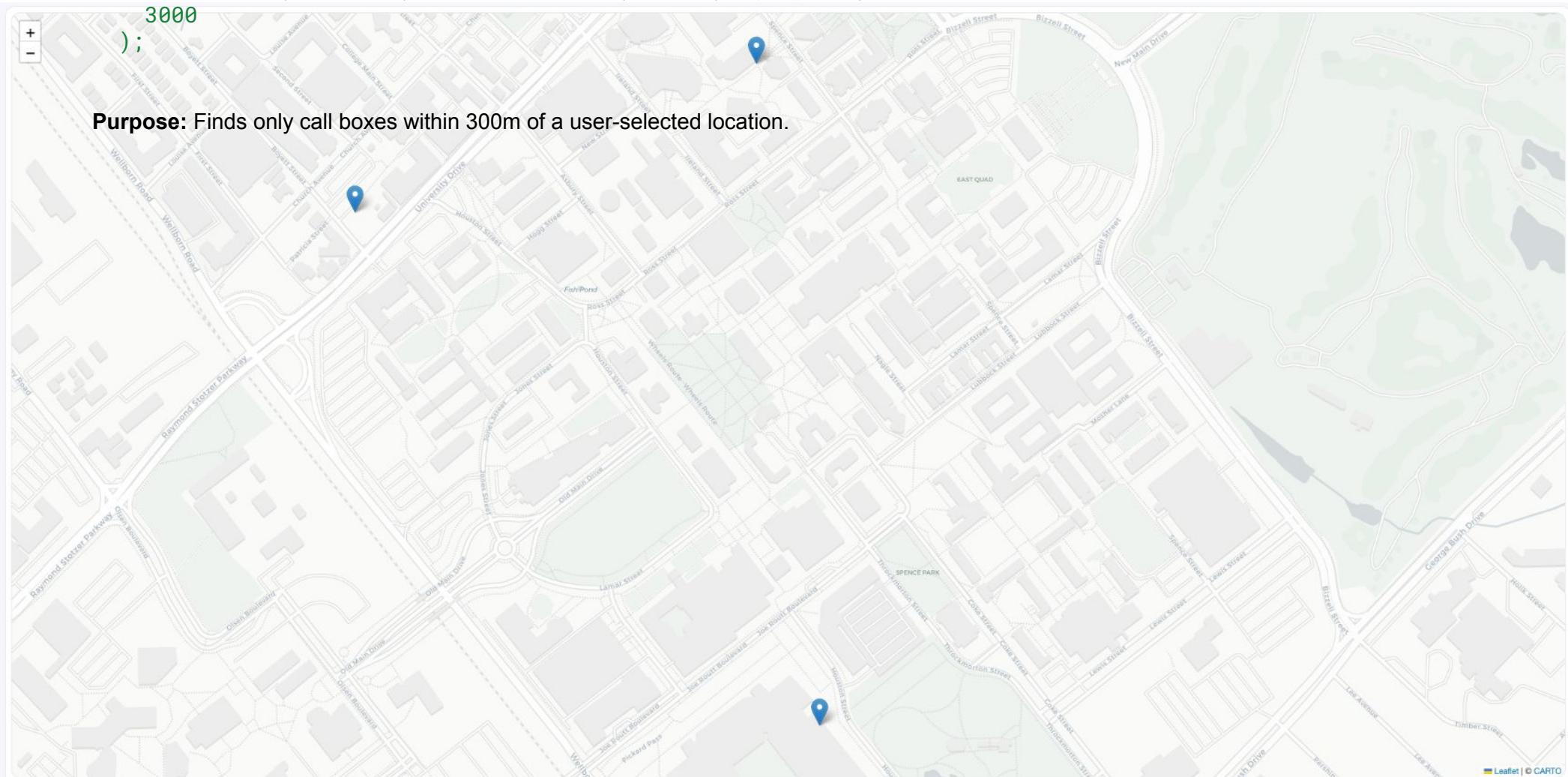
```
from campus_safety
where type = 'call_box'
  and st_dwithin(
    geom::geography,
    st_setsrid(st_point({{lon}}, {{lat}}), 4326)::geography,
    3000
  );
```

**Run SQL** **Clear Results**  
Click set → lat=30.617675, lon=-96.34704, dist=200m

**3) Results**

id	geom	name	type
19	{"type": "Point", "coordinates": [-96.3407, 30.6197]}	Emergency Call Box - Zachry	call_box
21	{"type": "Point", "coordinates": [-96.3397, 30.6107]}	Emergency Call Box - Kyle	call_box

2 row(s)



## ⚙️ 13. Create Bounding Box from All Features

```
select st_asgeojson(st_extent(geom))::jsonb as geom  
from campus_safety;
```

**Purpose:** Displays a bounding box around all safety points.

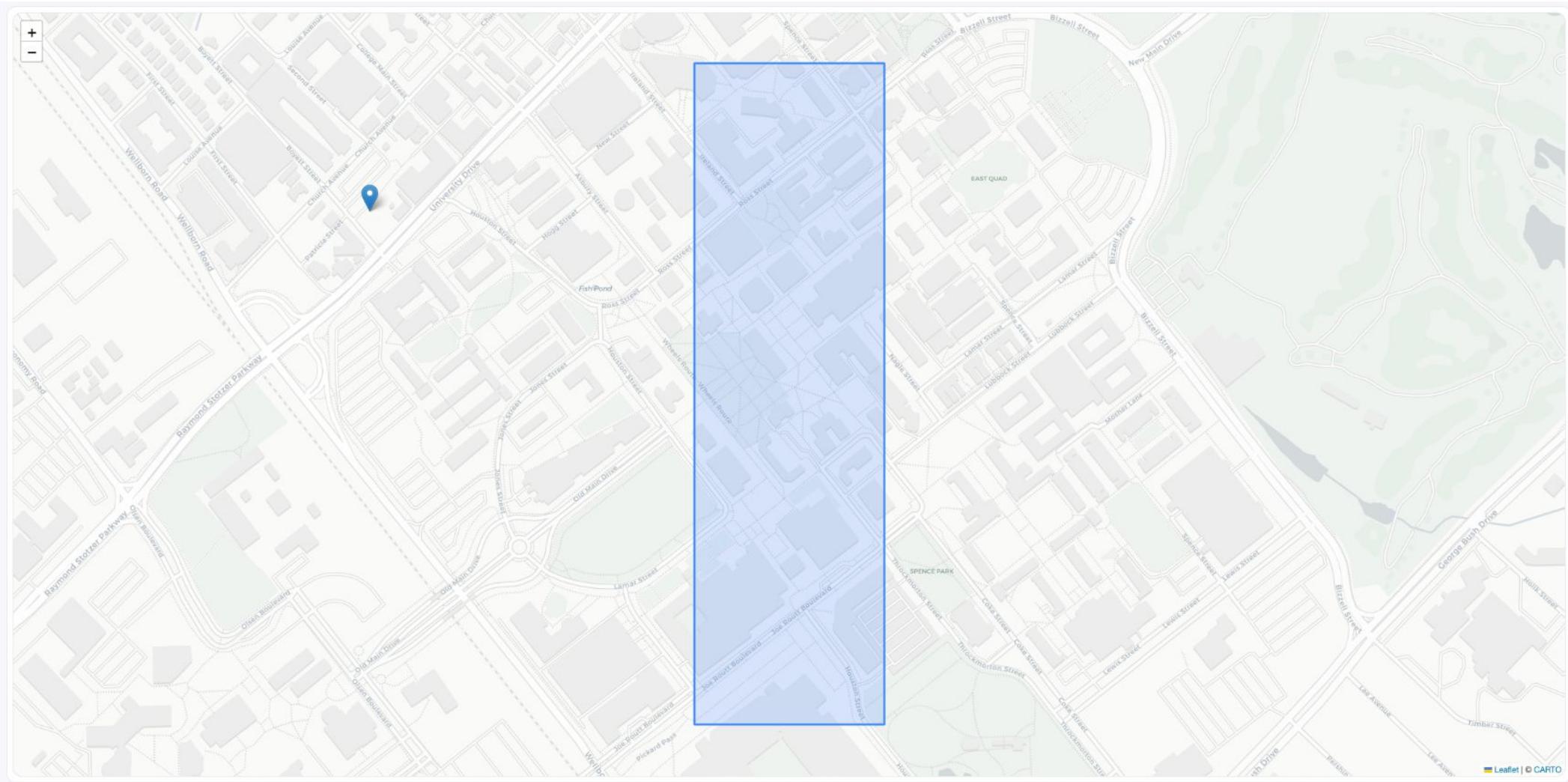
**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL:  
<https://hypqkovppcwhmxmtm.supabase.co>  
Supabase anon public key  
eyJhbGciOiJUzI1NiIsInR5cCI6IkpxVCJ9eyJpc3MIoJzdXI  
Connect Supabase Connected ✓  
Backend expects Postgres function  
run\_open\_sql(sql\_text) to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).  
select st\_asgeojson(st\_extent(geom))::jsonb  
as geom  
from campus\_safety;

**Run SQL** Clear Results  
Click set → lat=30.617675, lon=-96.34704, dist=200m

**3) Results**  
geom  
>{"type": "Polygon", "coordinates": [[[[-96.3419, 30.6107], [-96.3419, 30.6197], [-96.3389, 30.6197], [-96.3389, 30.6107], [-96.3419, 30.6107]]]]}  
1 row(s)



The map displays a detailed street layout of a university campus. A single blue location pin is placed near the center of the campus. A large, solid blue rectangle, representing the bounding box, is drawn over a cluster of buildings in the central area. The map includes labels for various streets, buildings, and landmarks, such as University Drive, Old Main Drive, Raymond Stotter Parkway, and various dormitory and academic buildings.

## 14. Nearest Building to Clicked Point

```
select b.id, b.bldg_name,
       st_distance(
           b.geom::geography,
           st_setsrid(st_point({{lon}}, {{lat}}), 4326)::geography
       ) as dist_m,
       st_asgeojson(b.geom)::jsonb as geom
  from campus_buildings b
 order by geom <-> st_setsrid(st_point({{lon}}, {{lat}}), 4326)
 limit 1;
```

**Purpose:** Returns the closest building to the selected map location.

**Spatial Query Lab v3-fixed**

**1) Supabase (SQL Runner)**  
Supabase Project URL  
<https://hypqkovnppcwvhmxm.supabase.co>  
Supabase anon public key  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpc3MIoUzdXI

Connect Supabase Connected ✓  
Backend expects Postgres function run\_open\_sql(sql\_text) to execute arbitrary SQL.

**2) SQL Editor**  
Click the map to set variables {{lon}}, {{lat}}, {{dist\_m}} (default 200).  

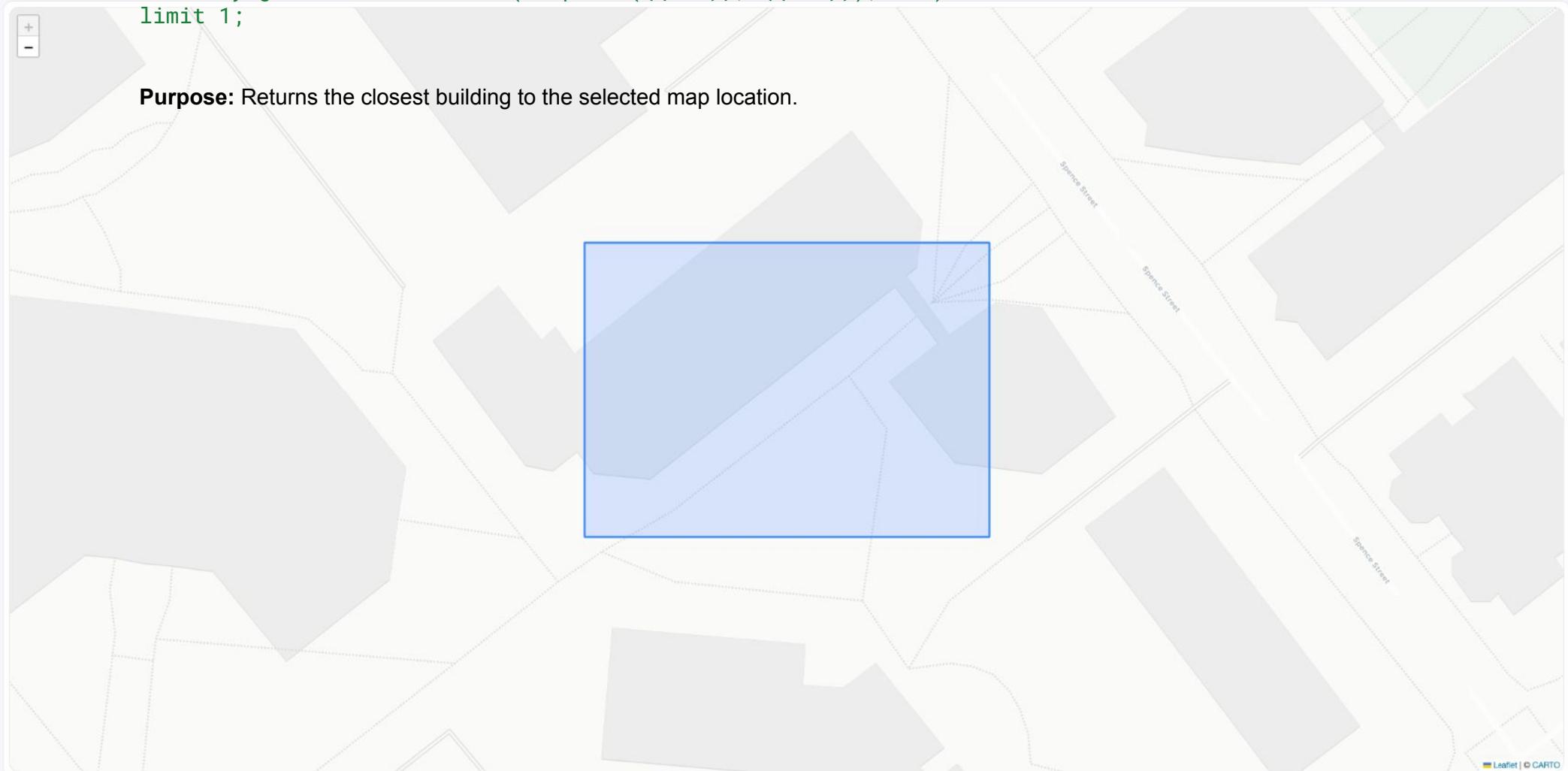
```
select b.id, b.bldg_name,
       st_distance(
           b.geom::geography,
           st_setsrid(st_point({{lon}}, {{lat}}), 4326)::geography
       ) as dist_m,
       st_asgeojson(b.geom)::jsonb as geom
  from campus_buildings b
```

Run SQL Clear Results  
Click set → lat=30.617675, lon=-96.34704, dist=200m

**3) Results**

id	geom	dist_m	bldg_
5	{"type": "Polygon", "coordinates": [[[-96.3412, 30.6199], [-96.3404, 30.6199], [-96.3404, 30.6194], [-96.3412, 30.6194], [-96.3412, 30.6199]]]}	591.70873822	Zachry Engine Educa Compl

1 row(s)



## 5. Agentic GIS: AI-driven spatial analysis with CARTO

The image shows a screenshot of the CARTO AI Agent interface. On the left, there's a dark blue background with the text "Introducing Agentic GIS". On the right, a white rectangular window titled "AI Agent" contains a conversation with an AI agent represented by a small circular icon with a dot.

**AI Agent:** How can I help you?

**Carto AI:** Give me some insights to identify the best locations for our next 5G towers. Ask several questions, something like this, or download a location with key metrics — all based on the current map view.

**Carto AI:** What should we deploy in our next 5G tower?

**Carto AI:** Which areas have the highest connection speed?

**Carto AI:** Can you suggest more locations in this city?

**Carto AI:** Download a report with the latest findings?

[Open in a new tab](#)

# Agentic GIS = GIS + AI Agents + Natural Language → Autonomous Spatial Analytics

## What It Is

**Agentic GIS** is part of CARTO's new platform update that embeds an **AI agent** into GIS.

Instead of manually writing spatial SQL or setting map filters, users can simply **chat with the map** using natural language — like:

“Show me the best locations to place new cell towers.”

“Where are the highest flood-risk zones within this region?”

“Download a report with the current map insights.”

## How It Works

- **LLM Integration:** The AI agent interprets user questions and automatically converts them into spatial queries (PostGIS, BigQuery GIS, etc.).
- **CARTO Cloud Platform:** It uses CARTO's data warehouse connections to access geospatial tables directly (e.g., Google BigQuery, Snowflake, PostgreSQL).
- **Dynamic Visualization:** Results are visualized instantly on the map layer, with options to download insights or create reports.
- **Continuous Learning:** The agent can refine analysis based on context, previous questions, or user feedback — similar to how ChatGPT builds context over a conversation.



**With Agentic GIS, do we still  
need to learn databases?**

That's a great discussion prompt — "With Agentic GIS (or similar AI-driven spatial tools), do we still need to learn databases?"

Here's a refined version of the question — plus some supporting arguments you can use in class or for your own reflection.

### 💡 Refined Question

"In an era of intelligent spatial agents and conversational GIS tools, is it still necessary for spatial analysts, GIS professionals, and students to master databases (e.g., SQL + PostGIS)?"

### ✓ "Yes, we still need to learn databases" — Key Reasons

#### 1. Foundational understanding:

Even when tools wrap complexity behind natural-language interfaces, knowing how databases work (tables, indices, schema design, SQL logic) gives you control, transparency and the ability to troubleshoot. Without that you become a consumer of tools rather than an informed practitioner.

#### 2. Customization, flexibility, and advanced workflows:

AI tools may handle many common tasks but when you need bespoke queries, custom joins, performance tuning for large datasets, or novel spatial operations, only the database layer gives you full flexibility.

#### 3. Performance and scale:

Many spatial data sets are large (satellite imagery, street-view, social sensors, mobility records). For scale you need database architectures, indexing strategies (e.g., GiST, H3, partitioning) and queries tuned for performance. AI agents may still rely on a database underneath, so database competence remains critical.

#### 4. Governance, reproducibility, auditability:

In research or enterprise, you must show your work, trace results, explain logic. SQL queries and database schema are explicit. If everything is done via black-box AI agent, you lose clarity. Knowing databases ensures you can validate, version and audit your workflows.

#### 5. Tool failure and vendor independence:

AI tools evolve rapidly, change provider, have limitations. If your knowledge is only of the tool, you're vulnerable. But if you also know the database core, you can adapt when the tool changes, move to

remains critical.

#### 4. Governance, reproducibility, auditability:

In research or enterprise, you must show your work, trace results, explain logic. SQL queries and database schema are explicit. If everything is done via black-box AI agent, you lose clarity. Knowing databases ensures you can validate, version and audit your workflows.

#### 5. Tool failure and vendor independence:

AI tools evolve rapidly, change provider, have limitations. If your knowledge is only of the tool, you're vulnerable. But if you also know the database core, you can adapt when the tool changes, move to new platforms, or build in-house solutions.

### ⚠ "...but not exclusively" — When databases may matter less in some contexts

- For introductory teaching or non-technical users: Tools like Agentic GIS make spatial insight accessible without writing SQL, lowering the barrier to entry.
- For standard repeatable tasks: If your workflows are templated (e.g., buffer + count within area), and the tool supports them, you may not need to hand-write queries every time.
- For decision-makers who only consume results: If you're a stakeholder reviewing outputs, you might not need deep database skills—but as a practitioner you still benefit from them.

### 🔍 My Position: Both are needed

So, yes — you *still* need to learn databases, but the role shifts. With intelligent spatial tools:

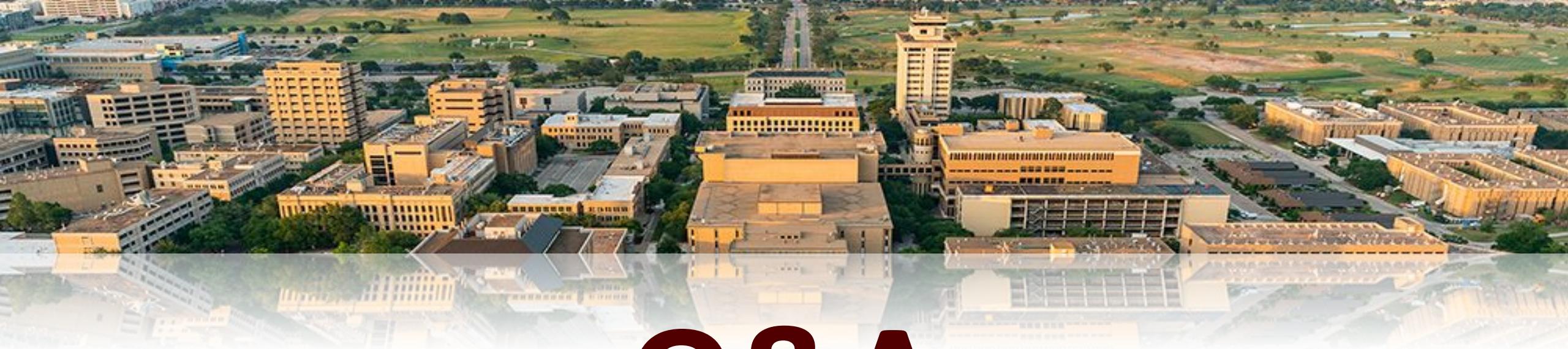
- The **toolset** becomes richer and more responsive (Agentic GIS, AI Agents for spatial analysis).
- The **database skills** become the foundation that enables you to use, extend and critically evaluate those tools effectively.
- In other words: Tools may automate many operations, but **database knowledge** is what lets you ask *why, how, what if, and what else*.

**Absolutely. If you don't take a  
Geodatabase course, you won't get  
the credits, thus preventing you  
from obtaining a degree and  
ultimately earning money.** 😂😂😂



**Of course. It's precisely because you chose to study this Geodatabase course that we met. Learning is not just about learning technology and knowledge, but also about learning how other people think.**





# Q&A

Happy Halloween !



*yyang295@tamu.edu*

