Building a Client-Side Spatial SQL Explorer

DuckDB-WASM, MapLibre GL, and CodeMirror 6 — all in the browser

Pukar Bhandari

pukar.bhandari@outlook.com

2026-02-25

Table of contents

Most of my day-to-day GIS work happens in R or Python — loading a shapefile, writing a few spatial queries, maybe making a quick map to sanity-check the results. That workflow is fast when you're already in a scripting environment, but it has real friction when you just want to hand someone a tool to explore a dataset without asking them to install anything, configure a Python environment, or deal with a server.

I wanted something simpler: drop a file in, write a SQL query, see results on a map. No backend, no dependencies, no accounts. That itch eventually became Spatial SQL Explorer.

1 What it is

Spatial SQL Explorer is a single-page application that runs entirely in the browser. You load a GeoJSON, CSV, TSV, or Parquet file; it registers as a DuckDB table; you write SQL against it; results render on a MapLibre GL map alongside a sortable, filterable results table. Everything — the query engine, the spatial extension, the editor, the map — runs client-side via WebAssembly. Your data never leaves your machine.

The live app is embedded below. Load the demo data (Nepal district boundaries) and try running a few queries to get a feel for it.

2 Why DuckDB in the browser?

The core technology here is DuckDB-WASM — a WebAssembly build of the DuckDB analytical engine. When the app starts, it downloads the WASM binary, spins up a worker thread, and gives you a fully capable SQL engine with support for window functions, CTEs, joins, aggregations, and — critically — the spatial extension that enables geometry operations like `ST_Within`, `ST_Intersects`, and `ST_AsGeoJSON`.

```
const bundle = await duckdb.selectBundle(duckdb.getJsDelivrBundles());
const worker = new Worker(/* inline worker script */);
const db = new duckdb.AsyncDuckDB(new duckdb.VoidLogger(), worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
const conn = await db.connect();

await conn.query(`INSTALL spatial; LOAD spatial;`);
```

DuckDB can read files registered directly from browser memory — no server upload needed. When you drop a GeoJSON file onto the drop zone, the app reads it as text, registers it with DuckDB, and runs `ST_Read` on it to parse the geometries into DuckDB's native binary format:

```
await db.registerFileText(`${tableName}.geojson`, text);
await conn.query(`
  CREATE TABLE "${tableName}" AS
  SELECT * FROM ST_Read('${tableName}.geojson')
`);
```

Parquet files get the same treatment via `registerFileBuffer`, which handles the binary format efficiently. The fact that DuckDB can query Parquet in-browser is something I still find genuinely impressive — you can drop a multi-million-row Parquet file and run aggregate queries on it without leaving the browser tab.

3 Geometry in the query pipeline

One design decision that took some iteration was how to handle geometry columns between DuckDB and MapLibre. DuckDB's spatial extension stores geometry as WKB (Well-Known Binary), which Arrow serializes as a binary column. MapLibre, on the other hand, expects GeoJSON.

The approach I landed on: detect WKB columns by their Arrow type ID after the query runs, then re-run the query wrapped in a `ST_AsGeoJSON` call to convert geometries on the fly:

```
const geomCols = result.schema.fields.filter(f =>
  f.type?.toString().toLowerCase().includes('binary') || f.typeId === 12
);

if (geomCols.length > 0) {
```

```
  const col = geomCols[0].name;
  result = await conn.query(
    `SELECT * EXCLUDE ("${col}"), ST_AsGeoJSON("${col}") AS "${col}"
     FROM (${cappedSql})`
  );
}
```

This keeps the API clean — you write `SELECT * FROM districts` and get geometry on the map automatically, without having to wrap every query in `ST_AsGeoJSON` yourself.

4 The SQL editor

The editor is CodeMirror 6, which I chose because its extension-based architecture makes it possible to do schema-aware SQL autocomplete with very little boilerplate. Whenever a table is loaded, the schema gets pushed into CodeMirror's SQL language extension:

```
const schema = {};
loadedTablesMeta.forEach(t => { schema[t.name] = t.columns; });
editorView.dispatch({
  effects: sqlSchemaCompartment.reconfigure(
    cmSql({ dialect: StandardSQL, schema, upperCaseKeywords: true })
  )
});
```

The `Compartment` abstraction here was something I had to dig into. CodeMirror 6's architecture is immutable-state-first: rather than mutating editor configuration, you replace it via dispatched transactions. The `Compartment` wraps a specific piece of the extension tree — in this case, the SQL language configuration — so you can hot-swap it without tearing down and rebuilding the entire editor. This keeps the cursor position, undo history, and scroll position intact when a new table is loaded.

The same pattern is used for the theme: wrapping `EditorView.theme(...)` in its own `Compartment` so that toggling between light and dark mode instantly updates gutter colors, selection highlights, and autocomplete styles without a page refresh.

5 Map styling

The style panel supports three modes — Single, Graduated (choropleth), and Categorical — all of which translate into MapLibre GL paint expressions at apply time.

Graduated styling uses `step` expressions, which MapLibre evaluates at the tile rendering level rather than in JavaScript. This means the coloring logic runs in the GPU pipeline, not in the main thread, so it scales well to large feature sets:

```
const stepExpr = ['step', ['get', col], ramp[0]];
for (let i = 1; i < n; i++) {
```

```
  stepExpr.push(breaks[i]);
  stepExpr.push(ramp[i]);
}
map.setPaintProperty('query-polygons', 'fill-color', stepExpr);
```

For classification, the app supports Quantile, Equal Interval, and Natural Breaks (Jenks). I implemented Jenks from scratch since there's no obvious lightweight library for it — it's the standard dynamic programming approach, running in $O(n \cdot k^2)$ time where n is the number of values and k is the number of classes. For the sizes of datasets this tool handles, it's fast enough.

Categorical styling has one subtle issue worth mentioning: MapLibre's `match` expression is strictly typed, so a numeric property value `1` never matches the string `"1"`. Since the same column might be typed as integers in DuckDB but come through as numbers in properties, I coerce values at match time:

```
const match = ['match', ['to-string', ['get', col]]];
unique.forEach((v, i) => {
  match.push(String(v));
  match.push(CATEGORICAL_PALETTE[i % CATEGORICAL_PALETTE.length]);
});
match.push('#aaaaaa'); // fallback
```

6 Session persistence and URL sharing

The app uses IndexedDB to persist loaded tables across sessions. When a file is loaded, its raw content is stored in IndexedDB alongside its format. On next visit, the app reads all stored tables back into DuckDB before rendering the UI — so your data and last query are waiting for you when you return.

URL sharing works by encoding the current SQL query and style settings as a base64 JSON blob in the URL hash:

```
const state = { sql, style: { ...styleSettings }, tables, hasDemo };
const encoded = btoa(unescape(encodeURIComponent(JSON.stringify(state))));
history.replaceState(null, '', `#state=${encoded}`);
```

The share link encodes the *query and style*, not the data itself — recipients need to load the same file. This is a reasonable tradeoff for a fully client-side tool. For tables that are just the demo dataset, the link includes a `hasDemo` flag so the app can auto-load the demo data when the link is opened.

7 The SharedArrayBuffer constraint

DuckDB-WASM uses SharedArrayBuffer for its WASM threading model, which requires the page to be served with `Cross-Origin-Embedder-Policy: require-corp` and `Cross-Origin-Opener-Policy: same-origin` headers. GitHub Pages doesn't let you set custom response headers.

The workaround is coi-serviceworker: a service worker that intercepts the page's own requests and injects these headers. One constraint that burned me during development: the service worker's scope is limited to its own directory. `coi-serviceworker.min.js` must live in the project root alongside `index.html` — if you move it into a subdirectory, it can't intercept the root page and DuckDB fails silently with an infinite reload loop.

8 Demo data: Nepal districts

The included demo dataset is Nepal's 77 district boundaries from the Survey Department of Nepal. Nepal has three tiers of administrative boundaries — provinces, districts, and municipalities — and the district level is the one most commonly used for regional analysis. The dataset includes attributes like province, district name, area, and population that make it a reasonable playground for trying graduated and categorical styling.

It's also, admittedly, a personal choice. I grew up in Nepal, and most of my professional work is on transportation planning in the US. This project sits in neither domain directly, but it felt right to use a dataset from home for the demo.

9 What I'd do differently

A few things I'd reconsider if starting over:

State management. Style state, table metadata, selection state, and filter state all live as module-level `let` variables. It works, but a simple reducer pattern would have made the URL sharing and session restore logic considerably cleaner.

The `demo` table name assumption. The URL-sharing logic has to know that the demo table is called `nepal_districts` to correctly reconstruct state from a shared link. This name is hardcoded in several places. A better design would treat the demo like any other loaded file and infer the table name dynamically.

10 Wrapping up

This was a genuinely fun project to build. Working with DuckDB-WASM pushed me to understand more about the WebAssembly execution model than I expected to, and CodeMirror 6's architecture — while steep at first — is elegant once you understand the `Compartment`/transaction model.

If you're doing spatial data work and want a scratchpad that doesn't require spinning up a server, give it a try. The code is on GitHub — issues and PRs welcome.