



CASE STUDY · ARCHITECTURAL SPECIFICATION

json-query-lite: SQL-Shaped Search for Big JSON in Small Places

“Give JSON arrays a real query language and the UI stops being the database.”

ROLE

Author & Maintainer

TIMELINE

Open source · independently main-
tained

DOMAIN

Search & Retrieval Systems

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Open-source hobby work, MIT-licensed, built on my own time: json-query-lite (TypeScript, zero dependencies) plus a Rust/Wasm sibling for high-cardinality indexing. The engine parses SQL-shaped queries - comparisons, LIKE/IN/BETWEEN, CONTAINS/REGEX/FUZZY, nested AND/OR/NOT, multi-column ORDER BY with BM25 relevance - into an AST evaluated over inverted indexes and typed arrays. It ships as ESM, CJS, and a minified IIFE that loads in browsers and LWC static resources. The demo dashboard exercises 50,000 generated rows with 50+ example queries. Nothing here touches employer or client data; the fixtures are generated and the lineage is public on GitHub.

I built this for myself first: every quoting and catalog experiment I ran needed the same thing - fast, expressive filtering over large JSON without standing up a search cluster. Transactional stores answer point queries well and analytical scans poorly; browsers answer neither well at 500K rows. A small dedicated engine fills the gap.

Quoting-style UIs need multi-attribute, nested-boolean filtering with ranking - the kind of query that is painful in SOQL-like point lookups and lethal as a full client-side scan. Dumping the dataset into the tab makes the UI the search engine, and the UI loses.

CONSTRAINTS

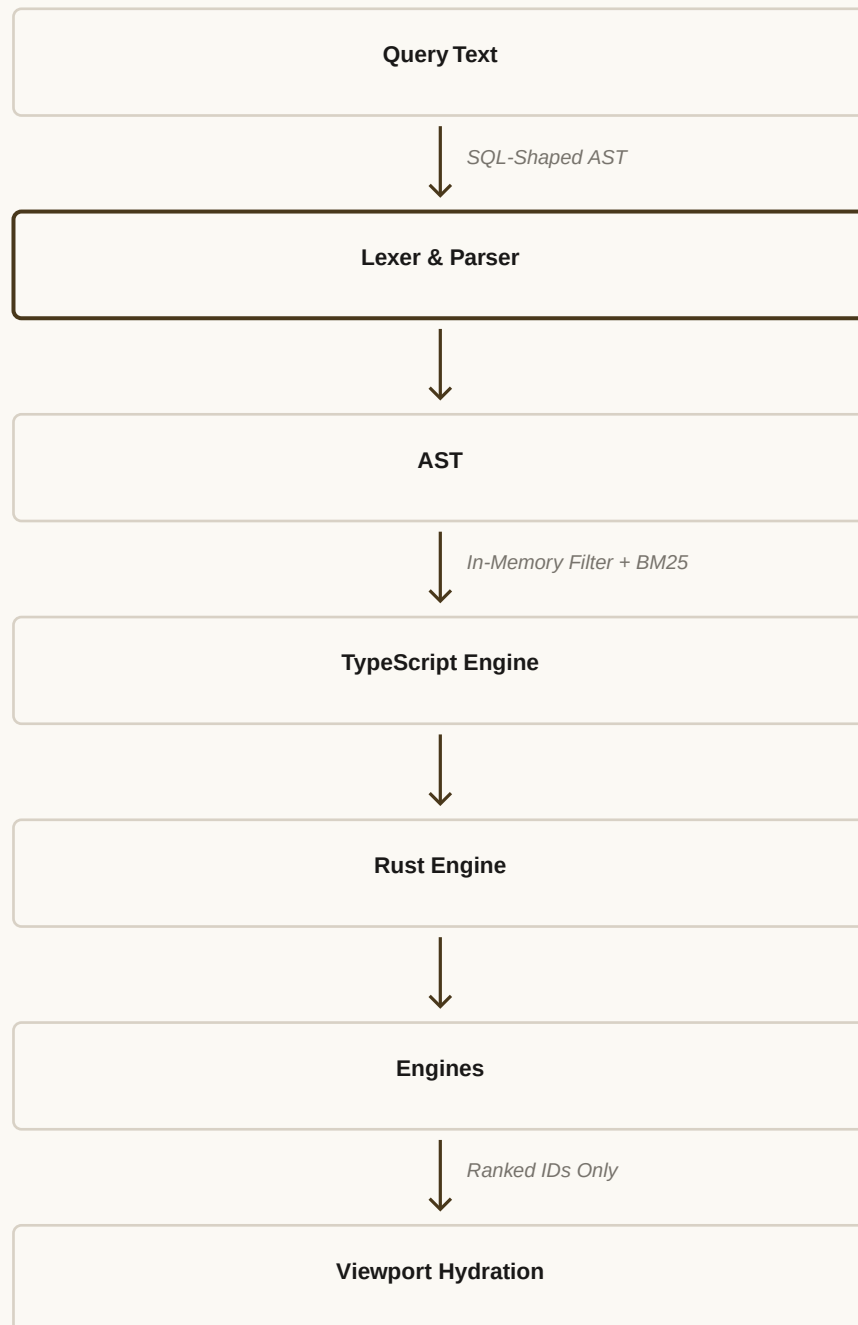
- Expressive grammar: nested booleans, ranges, text operators, sorting with relevance
- Millisecond answers over tens of thousands of in-memory rows
- Zero dependencies and a bundle that fits LWC static resources
- Same semantics in Node, browsers, and the component - tested, not asserted

PART I · ARCHITECTURE

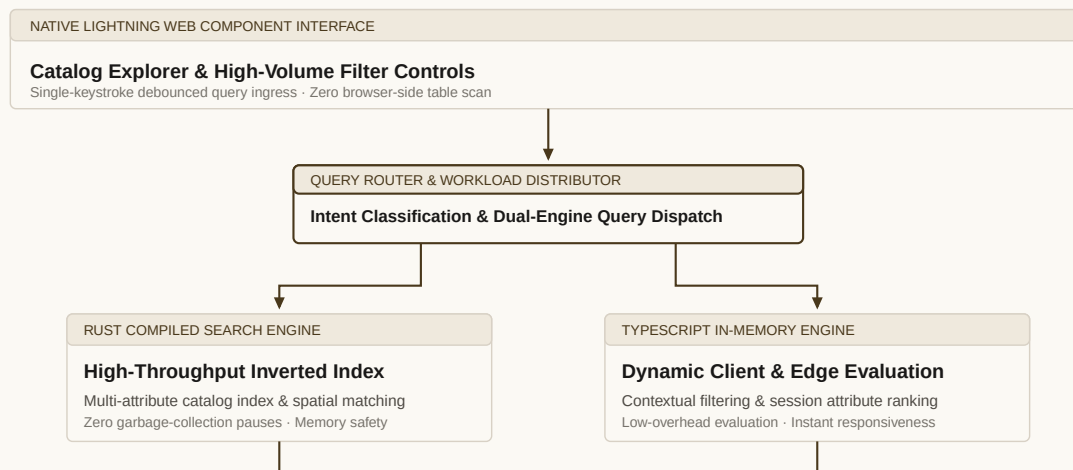
The system as designed

A compact pipeline: lexer with wildcard and modifier support, recursive-descent parser into a typed AST (terms, predicates, AND/OR/NOT, numeric terms), then a two-pass engine - filter and collect term statistics first, BM25-score and sort second. Inverted field indexes serve equality fast paths; a query cache and an adaptive result cache absorb repeated shapes. The Rust sibling mirrors the grammar for bulk indexing behind the same contract.

JSON-QUERY-LITE: QUERY PIPELINE & DUAL RUNTIMES



One grammar, two engines: TypeScript for the edge, Rust for bulk scans, both behind identical contracts.



Full system schematic - json-query-lite: Query Pipeline & Dual Runtimes.

PART I · DECISIONS

Choices, engineering, tradeoffs

Dedicated engine vs. stretching the transactional store

Record stores optimize for consistency and point access, not ranked multi-attribute scans.

CHOSEN

Small purpose-built engines (Rust for bulk, TypeScript for edge) behind one grammar.

WHY

Each workload meets the runtime built for it, and the UI receives ranked IDs instead of raw tables.

Ranked IDs plus viewport hydration vs. full-record streaming

Moving full records per keystroke wastes wire and DOM budgets together.

CHOSEN

Return ranked identifiers; hydrate only visible rows.

WHY

Payloads stay tiny and rendering stays constant-cost no matter the result size.

ENGINEERING NOTES

- Wrote the lexer and parser by hand - arrays, modifiers, wildcards, and multi-column ORDER BY included - with a validate() entry for authoring feedback.
- Implemented two-pass BM25 with dynamic 0–100 normalization so scores read sensibly at any corpus size.

- Built inverted field indexes with explicit `createIndex()` for hot attributes plus automatic smart lookup.
- Added query-AST caching (hundreds of entries) and adaptive result caching tuned for repeated interactive shapes.
- Shipped ESM, CJS, types, and minified IIFE from one source; the demo dashboard runs 50,000 generated rows in-browser.
- Mirrored the grammar in the Rust sibling (serde + wasm-bindgen surface) for bulk scans behind the identical query contract.

PART I · OUTCOMES

Tradeoffs, results, lessons

External index freshness vs. live-store queries

Gain: Millisecond ranked answers with no platform search-limit consumption.

Cost: Indexes must follow source changes.

Mitigation: In my studies, async change streams refresh indexes; staleness windows are explicit per use case.

Two implementations vs. one language

Gain: Bulk throughput where it matters, edge agility where it counts.

Cost: Two toolchains behind one grammar.

Mitigation: One shared grammar and contract tests asserting identical results across engines.

OUTCOMES

- MIT open source with a runnable 50,000-row demo and 50+ example queries
- Millisecond multi-attribute queries over large in-memory fixtures in the harness
- BM25 relevance with 0–100 normalization across corpus sizes
- Dependency-free IIFE that loads in browsers and LWC static resources
- Rust sibling behind the same query contract for bulk indexing

LESSONS

- Two-pass scoring (filter, then rank) keeps relevance honest at any corpus size.
- One grammar across two runtimes is a contract worth testing, not asserting.