.rete: Browser-Native SPARQL over Static, Range-Addressable RDF Files

Carlos Vivar Rios[1,2]

[1]Swiss Data Science Center, Lausanne, Switzerland
[2]EPFL, Lausanne, Switzerland

Abstract

Publishing an RDF knowledge graph for interactive access normally means either operating a SPARQL endpoint or shipping the whole dataset to the client. We present *.rete*, which packages a graph together with its dictionary, permutation indexes, structural summaries and descriptive metadata into one immutable file whose sections are addressed by byte range. Such a file is served from ordinary static HTTP infrastructure and queried from a Web browser through a WebAssembly client. Our demonstration puts the result of a query next to what that query physically cost. Visitors open an unfamiliar dataset through its embedded card, which on a 2.09 GB, 215M-triple Zenodo graph takes 49 KB in three range requests and never touches the triple index. They then run selective SPARQL queries while the interface reports the requests issued, the bytes transferred, and the index permutation and byte ranges that produced the answer; editing a query shows how the access plan and the network cost move with it. The demonstration is equally explicit about where the model stops paying off, since broader query shapes read more of the file. It currently serves 88 public graphs totalling 144 GB.

Keywords

RDF, SPARQL, knowledge graph publishing, cloud-optimized formats, WebAssembly, Linked Data Fragments

Submission type: Demonstration. **Live demo:** https://demo.graphplaza.com **Code:** https://github.com/caviri/rete (Apache-2.0).

1. Introduction

Making an RDF graph queryable on the Web leaves a publisher with three unappealing options. A *SPARQL endpoint* has to be kept running indefinitely, and a decade of monitoring shows that public endpoints often fail basic availability expectations [1, 2]. A *complete download* moves the burden onto every consumer, which is hard to justify when the question concerns a single record. An intermediate interface such as Triple Pattern Fragments [3] reduces the server's share of the work, but there is still a server to deploy and maintain.

Static hosting, by contrast, is cheap and hard to break, and several data communities have learned to exploit it by changing the *artifact* instead of the server. Parquet [14], Cloud-Optimized GeoTIFF [15], Zarr [16] and PMTiles [17] are all internally addressable: the client reads a directory from the file header, then fetches only the byte ranges it needs over HTTP `Range` [18]. DuckDB-Wasm closed the loop for SQL by moving the engine into the browser [12]. RDF's nearest equivalent, HDT [8], predates this pattern. It compresses a graph into a portable file, but ships its additional indexes separately [9], assumes the file is downloaded whole, and leaves Web-scale querying to a Fragments server layered on top.

Research question. *Can an RDF graph be published like a static Web asset while retaining practical, interactive SPARQL access, without a database server and without transferring the complete dataset?*

Contributions. We claim three. **(C1)** A *range-addressable RDF snapshot format*: one immutable file containing a term dictionary, six permutation indexes, structural summaries and descriptive metadata, every section reachable through a typed directory in a 1 KB header (§2). **(C2)** A *progressive client-side*

ISWC 2026 Posters and Demos Track, October 25–29, 2026, Bari, Italy
carlos.vivarrios@epfl.ch (C. V. Rios)
https://github.com/caviri/rete (C. V. Rios)
0000-0002-8076-2034 (C. V. Rios)

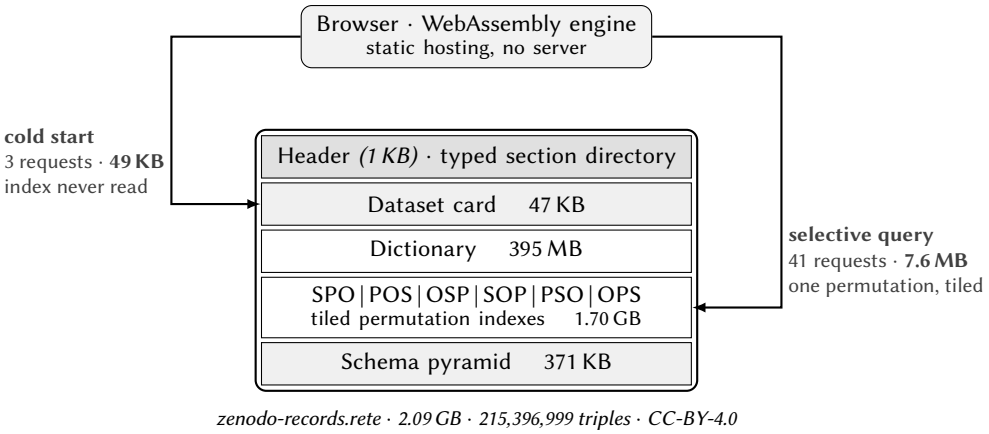


Figure 1: A `.rete` snapshot is one immutable file on static hosting. The header's typed directory makes every later read a bounded HTTP `Range` fetch, so a browser can answer questions about a 2.09 GB graph by reading kilobytes of it. All figures are measured (§5).

query model that distinguishes card-answerable queries that never read the triple indexes, selectively routed triple patterns, and broader SPARQL queries that may read more (§3). **(C3)** A *browser-based demonstration* that presents query results *alongside the physical execution evidence*: requests issued, bytes transferred, index permutation selected, byte ranges read, and whether the main triple index was touched at all (§4).

2. Architecture and File Format

One file, addressed by ranges. A `.rete` file opens with a 1 KB header holding a *typed section directory*: the offset and length of every section. Nothing later in the file is ever located by scanning. RDF terms are dictionary-encoded to integers, and the dictionary is front-coded in chunks with per-chunk restart tables, so any one chunk can be fetched and decoded on its own. Triples are stored as integer rows in *six* permutation indexes (SPO, POS, OSP, SOP, PSO, OPS), quad-aware for named graphs and cut into tiles addressed by a small tile directory. Because all six exist, any triple pattern resolves to a contiguous range with its bound components leading, which is what keeps remote routing bounded. Two further sections describe the data rather than store it: a *dataset card* and a leveled *schema pyramid* over `rdf:type` classes. Both sit before the dictionary and are readable without touching a single index byte.

The dataset card. The card is a small metadata section, written at build time from the graph itself, that answers what a person wants to know before writing a query: title, licence, provenance and content hash, exact counts of triples and distinct terms, the vocabularies in use, exact per-predicate and per-class histograms, and a few runnable starter queries. Being derived from the data rather than maintained beside it, it cannot drift from the file it ships in; and being placed early in the file, it is read in two or three range requests without opening the index. For 49 KB of a 2.09 GB file that is enough to render a dataset landing page, or to answer a question about predicate frequencies exactly (§3).

Compression. Each section is compressed independently with zstd (level 9), so decompression stays local to whatever range was fetched; the dictionary additionally front-codes shared IRI prefixes, and the indexes store integer IDs rather than terms. Compression is thus a property of each addressable unit rather than a wrapper around the file, which is what lets a compressed file stay queryable in place. On the ETH Research Collection graph (11.6M triples, CC0), 1.50 GB of N-Triples become a 206 MB `.rete`, a 7.3× reduction; `gzip -9` reaches 188 MB and `zstd -19` 126 MB, but neither answers a query without a full download and decompression pass. Two thirds of the file is the six index permutations and one third the dictionary, and across the published corpus a triple costs between 6.9 and 13.5 bytes with both included.

Deployment. The build is offline and single-writer; the output is immutable, content-hashed and versioned like a software release, hence trivially cacheable. The same Rust core is compiled natively

and to WebAssembly, where the remote reader implements the range interface with coalesced, retried HTTP `Range` requests. No application server is involved: the demonstrated files sit in object storage behind a CDN, and work equally from GitHub Pages.

3. Progressive Querying in Place

The engine covers the practical SPARQL 1.1 surface [19] — the four query forms, joins, OPTIONAL, UNION, MINUS, FILTER, property paths, aggregates, subqueries and named graphs — evaluated over integer IDs with a lazy pipeline and adaptive join execution. What matters here is not that coverage but *how much of a remote file a query has to touch.* The client distinguishes three regimes, whose measured costs appear in Table 1.

Some questions the card already answers, and those never open the index at all: the per-predicate histogram of a 215M-triple graph is exact from the card, its 36 counts summing to precisely the 215,396,999 triples in the file. *Routed* triple patterns resolve their bound terms through dictionary chunks, pick the permutation whose prefix is bound, and fetch the tiles covering the resulting range. *General* SPARQL opens the file lazily, reading tile directories only, and faults in index tiles as scans reach them. The boundary of the claim runs between these regimes: bounded, kilobyte-scale access is guaranteed for the first and typical for the second, while an unselective aggregate over the whole graph must read the index and will move a large fraction of the file.

4. The Demonstration

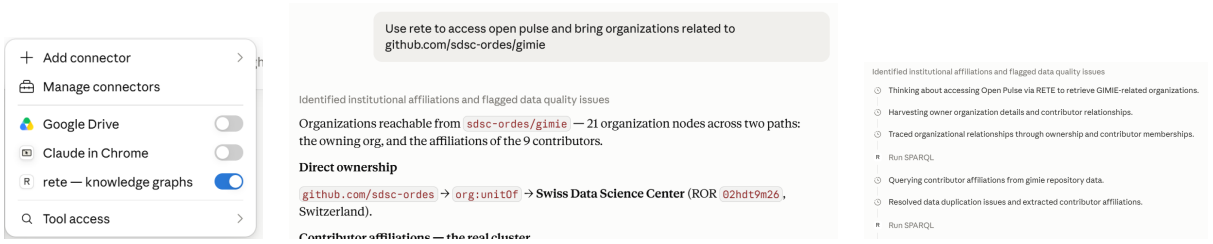
Because the file is the interface, anything that can issue an HTTP range request can query it. The demonstration therefore runs at three stations, none of which has a server behind it.

(a) In the browser. A static HTML page embeds the WebAssembly engine. Visitors work a short loop on a graph they did not build, by default `zenodo-records.rete`, every published Zenodo record (7.76M records, 215M triples, CC-BY-4.0). They *open* it: the card renders title, licence, counts, histograms and starter queries after 49 KB in three requests, with the index unopened. That entry cost barely grows with the graph: the same three requests return 2.6–2.8 KB on the 17.5 GB ORCID and 35.9 GB OpenCitations files. They ask something the card already answers (`SELECT ?p (COUNT(*) AS ?n) WHERE { ?s ?p ?o } GROUP BY ?p`), which returns exact counts for all 36 predicates while the panel reports *index accessed: no.* They then ask something *selective*: a bound pattern returns one record's 19 triples for 7.6 MB in 41 requests, and the panel shows the plan behind them, the dictionary IDs (`s=8520298 p=1 o=32192396`), the permutation (SPO), the tile and byte range (`SPO/15295`, bytes 708,796,070–708,821,740). This step is the point of the demonstration: not that SPARQL works, but that a visitor sees which physical parts of a remote graph an answer required. Editing the query moves plan and cost together (Table 1), and the loop is the same on much larger files: over `crossref.rete`, the Crossref citation graph as 3.78 billion triples in 60.2 GB, a cited-by lookup returns 100 citing works for 96 MB, 0.17% of the file (Figure 3a). The demonstration site runs the loop over *Open Pulse*,[1] the EPFL/SDSC research-software graph and a third-party adopter, asking which laboratories publish the most-starred software from a 51 MB file on static hosting.

(b) In a notebook. The same files are reached from Python in a JupyterLite notebook (Figure 3b): `%pip install rete-graph` into a Pyodide kernel, no local install and no server behind the page. It also goes the other way, building a `.rete` from an rdflib graph in the tab and dissecting the result, so publishing and querying are both client-side.

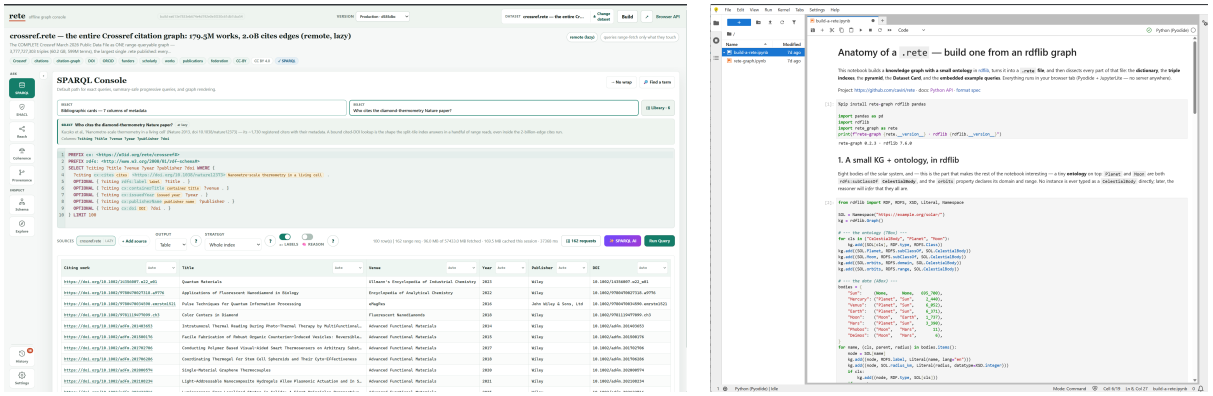
(c) From an agent. The packaged MCP extension (`rete.mcpb`) installs into an MCP-capable assistant and exposes nine tools over the same files, among them the card, the schema and SPARQL. Enabling it is one toggle (Figure 2a), after which a question in plain language is answered from the graph (b) by a chain of `Run SPARQL` calls the user can inspect (c). The agent follows the progressive

[1] https://openpulse.epfl.ch; ontology at https://github.com/sdsc-ordes/open-pulse-ontology.



(a) enable the extension (b) question and answer (c) the tool trace

Figure 2: Station (c): an assistant queries Open Pulse through `rete.mcpb`. Asked which organizations relate to the `sdsc-ordes/gimie` repository, it resolves ownership (`org:unitOf` → Swiss Data Science Center, ROR `02hdt9m26`) and contributor affiliations over four `Run SPARQL` calls (the panel shows three), flagging the identifier inconsistencies it meets.



(a) the browser station (b) the notebook station

Figure 3: Stations (a) and (b). *Left:* a cited-by lookup over `crossref.rete` (3.78 billion triples, 60.2 GB) returns 100 citing works for `96.0 MB of 57433.0 MB fetched` in `162 range req` — 0.17% of the file. *Right:* a JupyterLite notebook builds a `.rete` from an rdflib graph and dissects it, with the client installed by `%pip install rete-graph` into a Pyodide kernel; nothing runs outside the browser tab in either case.

Table 1
Measured network cost on `zenodo-records.rete` (2,094,612,808 bytes; 215,396,999 triples; CC-BY-4.0). Times are end-to-end from a residential link, including process start and TLS.

Interaction	Requests	Bytes read	% of file	Index read	Time
Dataset card (= exact predicate histogram)	3	49,452	0.002%	no	0.7 s
Selective pattern, 19 triples	41	7,602,176	0.36%	partial	5.7 s
Join: version chain → title	30	17,825,792	0.85%	partial	4.3 s
Join: ORCID author → works	50	18,743,296	0.89%	partial	7.3 s
Join: cited-by, empty result	24	4,980,736	0.24%	partial	3.4 s

path a person does, reading the card to learn the vocabulary before querying, which stops a model inventing predicate names.

In use. The playground serves 88 public graphs totalling 144 GB; switching the browser station to `orcid.rete` (1.30 billion triples in 17.5 GB) keeps the loop identical, one researcher's record for 28 MB.

5. Implementation and Preliminary Evaluation

The core is a single Rust library under Apache-2.0, compiled natively and to WebAssembly, and wrapped by the clients in Table 2. They differ in language and runtime but not in function: each opens a local or remote `.rete` and evaluates SPARQL against it, without a server. Correctness is checked by differential testing against Oxigraph [13]: on a 588k-triple OpenCitations network a 24-query operator matrix returns row-identical results on both engines (24/24), rete winning or tying 20 of 24 shapes; opening

that graph takes ∼16 ms against ∼2.2 s for a store that must parse and index it first, because the indexes are already in the file. The engine passes ∼75% of the 309 W3C SPARQL 1.1 query-evaluation tests (≈89% excluding entailment regimes it omits by design and tests needing a live SERVICE endpoint).

Table 1 reports what a browser actually pays to answer questions about a 2.09 GB remote graph. All numbers were measured on 2026-07-24 against the public files over the commodity Internet, and are reproducible with the shipped `rete card-url`, `sparql-url`, `why-url` and `cost` commands.

Table 2
Clients over one engine. Each reads local files and remote URLs through the same range interface; the three demonstration stations are marked.

Client	Distribution	Runtime
Rust core + CLI	source, Apache-2.0	native
JavaScript / WebAssembly	`rete-graph` (npm)	browser, Node *(station a)*
Python	`rete-graph` (PyPI)	CPython; Pyodide *(station b)*
MCP extension	`rete.mcpb`, 9 tools	MCP host *(station c)*
R	extendr package	native
Java	Maven module	JVM, wasm on Chicory

Limitations. Snapshots are immutable, so updates require a rebuild: this is a publication format, not a transactional one. Not every SPARQL shape gets equally selective remote execution, and an unselective whole-graph aggregate must read the index. The structural summary that ships beside the card is still experimental, and the card is the route we rely on for index-free answers. Entailment is limited to OWL 2 QL query rewriting [20], and latency depends on `Range` support and round-trip time: the times in Table 1 are dominated by request latency rather than bytes. Finally, this evaluation is an access-cost study, not a database benchmark.

6. Related Work

Compressed and self-indexed RDF. HDT [8] pioneered the portable binary graph; its query indexes are a sidecar [9] and the design assumes a complete download. Compact self-indexes such as the Ring [10] optimise in-memory space, not partial remote reads. *The Fragments spectrum.* TPF [3], brTPF [4], SaGe [5], SMART-KG [6] and WiseKG [7] redistribute effort between server and client; rete sits at the extreme client end, where the "server" is unmodified static hosting. *Client-side engines.* Comunica [11] executes SPARQL in the browser over Web interfaces; DuckDB-Wasm [12] is the closest architectural sibling, for SQL over the cloud-optimized formats whose pattern we transfer to RDF. *Dataset description.* VoID [21], DCAT [22] and Croissant [23] are sidecar documents that can drift from the data, whereas the card is generated at build time and travels *inside* the artifact it describes [24]. The contribution is not that any one capability is unprecedented, but their combination in one artifact: a self-contained RDF snapshot with embedded indexes and self-description, published on static HTTP and executed in WebAssembly, with progressive access the user can observe.

7. Conclusion

A knowledge graph can be published like any static Web asset and still be queried from a browser: kilobytes for simple questions, a fraction of a percent for selective ones, and a demonstration that shows which parts each query needed. Keeping a graph queryable then costs no more than hosting a file.

Declaration on Generative AI

The author used Claude (Anthropic) for grammar and spelling checking, paraphrasing and rewording, and for assistance in producing the figures and in running and tabulating the reported measurements.

All measurements were executed against the public artifacts and verified by the author, who reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] C. Buil-Aranda, A. Hogan, J. Umbrich, P.-Y. Vandenbussche, SPARQL web-querying infrastructure: Ready for action?, in: Proc. ISWC, 2013, pp. 277–293.

[2] P.-Y. Vandenbussche, J. Umbrich, L. Matteis, A. Hogan, C. Buil-Aranda, SPARQLES: Monitoring public SPARQL endpoints, Semantic Web 8(6) (2017) 1049–1065.

[3] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, P. Colpaert, Triple Pattern Fragments: a low-cost knowledge graph interface for the Web, J. Web Semant. 37–38 (2016) 184–206.

[4] O. Hartig, C. Buil-Aranda, Bindings-restricted triple pattern fragments, in: Proc. ODBASE, 2016, pp. 762–779.

[5] T. Minier, H. Skaf-Molli, P. Molli, SaGe: Web preemption for public SPARQL query services, in: Proc. WWW, 2019, pp. 1268–1278.

[6] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, A. Polleres, SMART-KG: Hybrid shipping for SPARQL querying on the Web, in: Proc. WWW, 2020, pp. 984–994.

[7] A. Azzam, C. Aebeloe, G. Montoya, I. Keles, A. Polleres, K. Hose, WiseKG: Balanced access to web knowledge graphs, in: Proc. WWW, 2021, pp. 1422–1434.

[8] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, M. Arias, Binary RDF representation for publication and exchange (HDT), J. Web Semant. 19 (2013) 22–41.

[9] M. A. Martínez-Prieto, M. Arias, J. D. Fernández, Exchange and consumption of huge RDF data, in: Proc. ESWC, 2012, pp. 437–452.

[10] D. Arroyuelo, A. Hogan, G. Navarro, J. L. Reutter, J. Rojas-Ledesma, A. Soto, Worst-case optimal graph joins in almost no space, in: Proc. SIGMOD, 2021, pp. 102–114.

[11] R. Taelman, J. Van Herwegen, M. Vander Sande, R. Verborgh, Comunica: a modular SPARQL query engine for the Web, in: Proc. ISWC, 2018, pp. 239–255.

[12] A. Kohn, D. Moritz, M. Raasveldt, H. Mühleisen, T. Neumann, DuckDB-Wasm: Fast analytical processing for the Web, Proc. VLDB Endow. 15(12) (2022) 3574–3577.

[13] T. Pellissier Tanon, Oxigraph: a SPARQL database and RDF toolkit, software, https://github.com/oxigraph/oxigraph, 2024.

[14] Apache Software Foundation, Apache Parquet, columnar storage format, https://parquet.apache.org/, 2013–.

[15] Open Geospatial Consortium, OGC Cloud Optimized GeoTIFF standard, OGC 21-026, 2023.

[16] A. Miles et al., Zarr: chunked, compressed, N-dimensional arrays, Zarr specification v3, https://zarr-specs.readthedocs.io/, 2024.

[17] B. Barrett, PMTiles: a single-file archive format for tiled data, https://github.com/protomaps/PMTiles, 2024.

[18] R. Fielding, M. Nottingham, J. Reschke (Eds.), HTTP semantics, RFC 9110, IETF, 2022.

[19] S. Harris, A. Seaborne (Eds.), SPARQL 1.1 Query Language, W3C Recommendation, 2013.

[20] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, R. Rosati, Tractable reasoning and efficient query answering in description logics: The DL-Lite family, J. Autom. Reason. 39(3) (2007) 385–429.

[21] K. Alexander, R. Cyganiak, M. Hausenblas, J. Zhao, Describing linked datasets with the VoID vocabulary, W3C Interest Group Note, 2011.

[22] R. Albertoni, D. Browning, S. Cox, A. Gonzalez Beltran, A. Perego, P. Winstanley (Eds.), Data Catalog Vocabulary (DCAT) — Version 3, W3C Recommendation, 2024.

[23] M. Akhtar, O. Benjelloun, C. Conforti, et al., Croissant: a metadata format for ML-ready datasets, in: Proc. NeurIPS Datasets and Benchmarks, 2024.

[24] M. D. Wilkinson et al., The FAIR guiding principles for scientific data management and stewardship, Scientific Data 3 (2016) 160018.

[25] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, J. Stat. Mech. (2008) P10008.