Traqula: Providing a Foundation for The Evolving SPARQL Ecosystem Through Modular Query Parsing, Transformation, and Generation

Jitse De Smet [0009-0002-6513-5013], Ruben Taelman [0000-0001-5118-256X]

IDLab, Department of Electronics and Information Systems, Ghent University – imec.

Abstract. SPARQL engines continue to diverge in the versions and language extensions they support. With SPARQL 1.2 nearing completion and the working group's plans toward future maintenance, this divergence is likely to grow. As a result, language tools that aim to support multiple SPARQL versions and dialects face increasing maintenance costs. Furthermore, areas such as SPARQL federation become harder to manage. To address these challenges, we introduce Traqula, a modular TypeScript-based framework for parsing, transforming, and generating SPARQL queries. This article presents Traqula's architecture and its builder-based dependency-injection design, and we show how these components provide modular support for SPARQL 1.1 and 1.2 parsing and generation. Traqula enables round-tripping, supports generic algebraic and AST transformations, and delivers significantly faster parsing compared to current JavaScript-based SPARQL parsing. Traqula's modular architecture enables flexible experimentation for researchers, and it is production-ready thanks to extensive testing and modern development techniques. Looking forward, Traqula lays the groundwork for supporting additional query languages, developing generic rewriting modules for SPARQL federation engines, and building robust tooling for an increasingly diverse SPARQL ecosystem.

Keywords SPARQL, query, algebra, modularity, parser, generator

Canonical version: https://traqula-resource.jitsedesmet.be/

Resource type: software package

Licence: MIT

DOI: 10.5281/zenodo.17793334

URL: https://github.com/comunica/traqula

1. Introduction

The SPARQL query language [1] is the standard way to query RDF [2] data. While SPARQL endpoints [3] are commonly used to expose Knowledge Graphs through a highly expressive query API, alternative APIs have been proposed [4, 5, 6, 7, 8, 9] that offer different trade-offs between client and server effort for query execution. This heterogeneity in server APIs introduces challenges when executing federated SPARQL queries [10, 11, 12, 13, 14] over multiple of these APIs. While this API-based heterogeneity has been an active field of research [15, 16, 17, 18] in recent years, the heterogeneity of the SPARQL language itself lacks understanding. In practice, many SPARQL dialects exist [19, 20, 21, 22] each introducing their own extensions or limitations. Virtuoso, for example, extends SPARQL with full-text search capabilities [23], Apache Jena adds support for constructing quads [24], and Oxigraph provides an additional built-in function, `ADJUST` [25, 26]. Furthermore, some SPARQL endpoints might limit the SPARQL language by removing expensive operators such as OPTIONAL [27]. In the near future, this heterogeneity is expected

to increase even further as the RDF and SPARQL W3C Working Group enters its maintenance mode [28, 29], which could deliver more rapid successions of SPARQL once SPARQL 1.2 is finalized. As a result, the growing diversity of SPARQL versions and dialects introduces several challenges:

1. **Query evaluation**: A user query written in one version might not be executable by a SPARQL engine supporting another version.
2. **Tooling**: Linters, formatters, editors, and language servers often assume a specific SPARQL version.
3. **Maintainability**: Tools that support multiple SPARQL versions typically do so by maintaining multiple software versions, or one version with many conditions, making maintainability highly challenging.

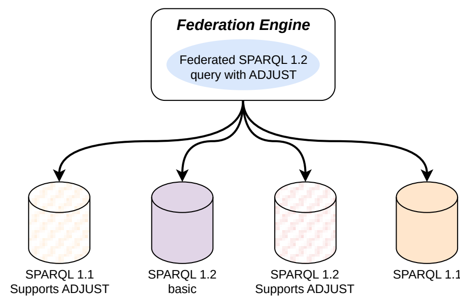

Fig. 1: The federated SPARQL query (blue) uses SPARQL 1.2 features such as triple terms and uses the non-standard builtin function ADJUST [25]. The query targets four SPARQL endpoints, all supporting different SPARQL version, RDF profiles [30], and language extensions. To allow query engines to integrate this heterogeneity, frameworks such as Traqula are necessary to bridge between these SPARQL dialects.

The query evaluation problem within a heterogeneous SPARQL ecosystem becomes especially visible in federated SPARQL query execution. While language dialects are common in many technologies, SQL [31] being a well-known example, the RDF data model [2] is explicitly designed for seamless integration of distributed datasets. SPARQL reflects this distributed nature through support for federated queries, where a single query may involve multiple endpoints, each with its own capabilities, limitations, and language features.

In such a setting, a SPARQL query written in one SPARQL version or dialect may not be executable on all federation members. For instance, a query formulated in SPARQL 1.2 might rely on features, functions, or syntactic constructs that an older endpoint does not support, or that are only available as vendor-specific extensions. Fig. 1 illustrates this scenario: a query is written in SPARQL 1.2 and uses the non-standard ADJUST function [25], yet the endpoints differ in their supported SPARQL versions, RDF profiles [30], and feature sets. Although the SPARQL Service Description specification [32] allows endpoints to declare their supported features, it does not offer mitigation paths to resolve these language mismatches. To solve this problem, there is a need for a parsing, transformation, and generation framework such as Traqula that can handle various SPARQL dialects. Such a framework provides a foundation for future SPARQL federation research towards new techniques and algorithms to manage these dialects.

In prior work [29], we introduced our vision of a modular SPARQL parser to address the growing heterogeneity of SPARQL dialects using builder-based dependency injection [33]. We envisioned a design with a prototype implementation demonstrating parser composability, showing how grammar modules could be combined to modularly define multiple SPARQL parsers covering different versions and dialects. In this article, we fully realize that vision with a complete implementation as well as applying the modular architecture to query generation and transformation. These features allow queries to be parsed, transformed, and regenerated reliably, while preserving their structure and semantics across different dialects.

This article presents Traqula, a modular SPARQL toolkit implementing these ideas. Traqula has already been integrated into the widely used Comunica SPARQL querying framework [15], demonstrating its maturity and practical applicability. Traqula's modular and composable architecture enables researchers and practitioners to: 1. Experiment with grammar changes, including adding, modifying, or removing rules; 2. tackle the complexities of federated SPARQL querying in a heterogeneous SPARQL ecosystem; and 3. lay the foundation for future query formatting and rewriting tools. Traqula is implemented in TypeScript and is available as open-source software under the MIT license on GitHub and npm, providing the community with a robust and flexible resource for building tools and engines for a heterogeneous SPARQL environment.

The remainder of this paper is structured as follows. Section 2 details the requirements of Traqula, Section 3 reviews related work, Section 4 presents the high-level architecture of Traqula, and Section 5 discusses its implementation. Section 6 evaluates its performance, and Section 7 concludes the paper.

2. Requirements

Traqula was designed with a set of concrete requirements in mind. In this section, we outline the most important ones:

1. **Flexibility**: allowing composition of parser, transformer, or generator out of small components that can be easily added, removed, or replaced.
2. **Round-tripping**: enable parsing into an AST and generating back to exactly the same string. When manipulations are made, only the manipulated parts change.
3. **Web-based**: harnessing web technologies and enabling execution within and outside browsers, to ensure wide usability.
4. **Language-agnostic**: by providing a generic core, Traqula facilitates the support of many query languages.

We elaborate on each of these requirements in more detail hereafter.

2.1. Flexibility

Given the increasing heterogeneity of SPARQL versions and dialects, Traqula must support a highly flexible and composable architecture. As outlined in the introduction and Fig. 1, differences in supported features, functions, and syntactic constructs already complicate query evaluation and tooling, and these discrepancies are expected to grow as SPARQL enters a phase of more rapid evolution through the SPARQL

working group's maintenance mode [29]. Addressing this landscape requires a toolkit in which parsers, generators, and transformers can be assembled from small, reusable modules rather than fixed monolithic grammars.

Flexibility is essential for two main reasons. First, researchers and practitioners need the ability to experiment with the language itself, by adding, modifying, or removing grammar rules, and evaluating the impact of those changes. Traqula's modular design aims to make such experimentation routine rather than burdensome, supporting rapid prototyping of new language features or alternative syntactic constructs. This capability has already been demonstrated in practice: Traqula's adoption within the modular Comunica query engine allowed us to contribute effectively to the standardization work for SPARQL 1.2.

Second, SPARQL federation engines require a way to mediate between heterogeneous dialects of the SPARQL endpoints they federate over. A single query may need to be adapted to the capabilities of multiple endpoints, each exposing different SPARQL versions or dialects. Traqula's modularity is therefore not merely a convenience but a prerequisite for constructing reliable transformations between dialects, since each dialect will require a parser and generator. By decoupling parsing, generation, and algebraic transformation into flexible and composable components, Traqula positions itself as a key tool for SPARQL rewriting across versions and dialects.

2.2. Round-tripping

Because SPARQL is a structured language that is both written and inspected by humans, it benefits from the same ecosystem of tooling found around programming languages, which includes editors, code highlighters, linters, refactoring tools, and formatters. Many of these tools rely on *round-tripping*, the ability to parse a query into an internal representation, apply a transformation, and regenerate a query string that is identical except for the intentional change. Achieving this property requires the parser and generator to preserve all user-visible details, such as comments, spacing, punctuation, or keyword capitalization, even when these details have no semantic impact on query evaluation.

Round-tripping is essential for practical tooling. A linter that renames a variable, for instance, should not unexpectedly rewrite unrelated parts of the query or normalize stylistic elements that the user intended to keep. A simple example of SPARQL reformatting is rewriting the variable *'s'* to *'subject'* in `SelECT * { ?s   ?p ?o }`, which should result in `SelECT * { ?subject   ?p ?o }`, without altering the spacing or keyword casing. Fig. 2 illustrates the different representations involved: the query string; its abstract syntax tree (AST) as received from the parser, and used by linters and formatters; and its algebraic form, constructed from the AST [1], and used by query engines. Transformations may occur at any of these levels, but we only require the round-tripping property for AST-level transformations.

Supporting round-tripping therefore shapes Traqula's design. The parser must retain sufficient information to reproduce the original query string, while the generator must reconstruct that structure without introducing unintended changes. This requirement ensures that Traqula can serve as a foundation for reliable SPARQL editing, formatting, and rewriting tools.

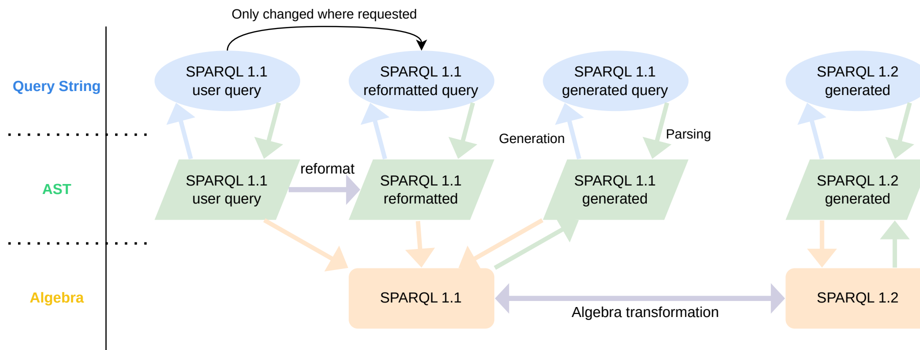

Fig. 2: Various representations of an algebraically equivalent SPARQL 1.1 query and its SPARQL 1.2 counterpart. The figure illustrates three layers: the concrete query string (blue), the Abstract Syntax Tree (AST, green), and the algebraic representation (orange). Round-tripping requires that parsing a query string into an AST and regenerating it produces the identical string. Transforming the leftmost query at the AST level preserves all syntactic information, so regenerating the transformed AST changes only those parts intentionally modified. By contrast, converting the AST to algebra discards syntactic detail: algebraically equivalent queries map to the same algebra and thus to the same canonicalized query string. Finally, the figure shows how transformations between SPARQL 1.1 and SPARQL 1.2 can be performed at the algebra level, where engines operate and where syntactic information is no longer preserved.

2.3. Web-based

Web-based technologies offer a natural foundation for widely adoptable SPARQL tooling. A Web-first JavaScript (or TypeScript after transpilation) implementation can run natively in the browser, where many SPARQL editors [34, 35], and even query engines [15] operate, while also executing on backend environments such as Node.js, Bun, or Deno, and inside application frameworks like Electron or Tauri. Moreover, the Web ecosystem lowers the barrier to implementing a SPARQL language server, similar to Qlue-ls [36], via the Language Server Protocol (LSP) using Microsoft's *vscode-languageserver-node*. Such an LSP or more broadly any Traqula derived (system) service could directly leverage the modular architecture, enabling rule-level extensibility and round-tripping capabilities within interactive editors, especially since Traqula's APIs comunicate using JSON Data Transfor Objects (DTOs), which simplifying crossing language boundries. In principle, WebAssembly (WASM) provides the same portability, enabling code written in languages such as Rust (wasm-pack), C++ (Emscripten), or even JavaScript (Javy) to run across the Web platform.
However, for Traqula we prioritise not only portability, but also accessibility, quick prototyping, and ease of contribution. TypeScript provides the same core advantages as WASM, execution across browser and server environments, while remaining far more familiar to the broad developer community, and without requiring ahead-of-time compilation. As the most widely used language on GitHub in 2025 [37], TypeScript maximises the likelihood that researchers and practitioners can both use and extend Traqula without specialised compilation toolchains or language-specific runtimes.

By choosing a Web-based, TypeScript implementation, Traqula satisfies the requirement of broad, frictionless adoption while integrating naturally into both front-end applications and backend query engines.

2.4. Language-agnostic

To address query language heterogeneity, Traqula provides a generic core that can be reused to implement modular parsers, generators, and transformers for a variety of query languages. The SPARQL 1.1 and 1.2 parser, generator, and algebra transformations all build on this single core library, which enables the creation of modular systems beyond SPARQL. For example, the SHACL Compact Syntax developed by W3C's Data Shapes Working Group [38] shares syntactic constructs with SPARQL, allowing it to reuse some components of Traqula's SPARQL grammar. Similarly, mapping languages such as RML [39] for Knowledge Graph constructions allow for representation in SPARQL algebra [40]. Another example is the LDQL query language [41], which enables directed web navigation for Link Traversal Query Processing [42, 43] over Linked Data documents. In contrast, the newly standardized ISO Graph Query Language (GQL) [44] defines a completely different syntax, but could still leverage Traqula's language-agnostic core to implement modular parsers, generators, and transformations.

In summary, Traqula's design is guided by four core requirements: flexibility, round-tripping, Web-based execution, and language-agnosticism. Together, these requirements ensure that Traqula can support modular, maintainable tooling for heterogeneous SPARQL dialects while remaining extensible to other query languages and deployment environments. With these principles in place, we now turn to existing approaches in parser design and query language tooling to position Traqula within the broader landscape.

3. Related Work

This section first covers the related work on parsers and compilers from a theoretical perspective; afterward, we list the most relevant existing parsers for SPARQL and other query languages. After covering the parsing itself, we take a look at common abstract syntax tree (AST) structures, specifically focussing on approaches that support round-tripping.

3.1. Parsers

Parsing a structured language involves transforming it from a string into a desired data structure. Conceptually, parsing can be divided into three steps [45]:

1. **Lexical analysis** (or *scanning*) performed by a *lexer*: the source character stream is read and broken into non-overlapping sequences called lexemes. Each lexeme is classified by a token type, often defined using regular expressions. Lexemes may be represented as strings, ranges in the source, or both. The lexer outputs a stream or list of tokens, which are the lexeme representation together with its token type.
2. **Syntax analysis** (or *parsing*) performed by a *parser*: after having generated a 'flat' stream or list of tokens, the next part is to create a data structure, typically a tree, often called the Abstract Syntax Tree (AST), representing the syntactic

structure of the language. Some parsers can automatically parse into a Concrete Syntax Tree (CST) which traces all grammar information without generalization or abstraction.

3. **Semantic analysis**: the AST is checked for non-structural constraints. In SPARQL, for example, the variable assigned in a `BIND` clause cannot be used in the immediately preceding tripleBlock within the same group. In programming languages, type checking is a common example of semantic analysis.

One example of joined execution is the case of a streaming parser where the output data structure of the parser is itself a stream. Streaming parsers are specifically interesting when parsing large amounts of data, where semantic analysis is either not required or required in limited form. Example data formats that tailor themselves to streaming, specifically RDF data, are Jelly [46] and n-triples [47]. Another way of joined execution is where syntax analysis and semantic analysis are joined, allowing the parser to fail fast in case a semantic constraint would be broken, an approach partially taken by Traqula.

The **parser construction technique**, i.e., the actual programmatic definition of the different parsing steps, can happen in various ways; we identify the following three:

1. **Generated**: Code generation tools come with their Domain Specific Language (DSL) that will typically share similarities with Extended Backus–Naur Form (EBNF) and some Regular Expression dialect. The build process of the software that uses the custom parser should then compile the parser definition file (in the custom DSL) to the desired target language, which is typically the same language as the software being built. Example parser generators are Bison [48] and ANTLR [49].
2. **Hand-built**: Code generation can come with optimization limitations, when the grammar allows for optimizations not taken by the code generator. Writing a handwritten parser is powerful but very challenging to get right, because compilers often know powerful, very specific, non-trivial optimizations. On top of programming language-specific optimizations, other powerful optimizations exist for specific sets of grammars, such as LL(1) and LL(k) [45, 49].
3. **Toolkits / libraries**: A compromise between hand-built parsers and generators exists in the form of toolkits / libraries. Parser building toolkits, e.g. Chevrotain [50], are software libraries that provide an API that facilitates the construction of parsers within a specific programming language. The benefit of constructing a parser within the programming language the parser would be called from is that the project's code can be more coherent, allowing better integration, while also providing abstraction for powerful optimizations that can be made for specific grammars such as LL(1) and LL(k). Additionally, it allows the usage of language-specific features, e.g. a type system, which are often not present in DSLs used by parser generators, since DSLs are often limited in complexity. However, toolkits miss out on compiler-based optimizations since they cannot fully optimize the user's code, and similarly miss out on certain optimizations that could be possible in hand-built parsers.

In the next section, we argue that parser-building toolkits offer a practical middle ground between generated and hand-built parsers, achieving good performance *(https://chevrotain.io/performance/)* while providing flexibility for modular, language-specific systems like Traqula.

3.2. Existing Query Parsers

Table 1 provides an overview of the parsing frameworks used by popular open-source query language parsing software and query engines. The table extends upon our previous work [29], extending our analysis to open-source implementations of the SQL [31], GraphQL [51], GQL [44], and Neo4J's cypher [52] query languages. We can clearly see that parser generators are the dominant approach, with 13 of the 16 systems listed using generated parsers. Only one implementation uses a handwritten parser, and two employ a parser toolkit, namely Chevrotain.

Parsing Software	Query Language	Parsing Framework	Construction Technique	Req: F	R	W	L
Apache Jena - arq(v5.6.0) [53]	SPARQL	JavaCC	Generated	·	·	·	·
Blazegraph (commit 829ce82) [54]	SPARQL	JavaCC	Generated	·	·	·	·
Oxigraph (v0.5.2) [55]	SPARQL	rust-peg	Generated	·	·	·	·
QLever (commit 6ec0a5e) [56]	SPARQL	ANTLR	Generated	▯	·	▯	·
RDF4J (v5) [57]	SPARQL	JavaCC	Generated	·	·	·	·
SPARQL.js (v3.7.1) [58]	SPARQL	Jison	Generated	·	·	✔	·
Stardog - Millan (commit 6109984) [59]	SPARQL	Chevrotain	Toolkit	✔	·	✔	·
Virtuoso opensource (commit 23cff67) [60]	SPARQL	Bison	Generated	·	·	·	·
Yasgui (v4.0.113) [34]	SPARQL	SWI Prolog	Generated	?	·	✔	·
Traqula (v1.0.0)	SPARQL	Chevrotain	Toolkit	✔	✔	✔	✔
DuckDB (v1.4.2) [61]	SQL	Bison	Generated	·	·	·	·
PostgreSQL (v18) [62]	SQL	Bison	Generated	·	·	·	·
SQLite (v3.51.0) [63]	SQL	Lemon	Generated	·	·	·	·
GraphQL-js (v16.12.0) [64]	GraphQL		Hand written	✔	·	✔	·
opengql grammar (v1.9.0) [65]	GQL	ANTLR	Generated	▯	·	▯	·
Neo4J (v25) [52]	Cypher	ANTLR	Generated	▯	·	▯	·

Table 1: List of parsing software packages for various query languages, including the underlying parsing framework and construction technique. The table also indicates how each package satisfies the requirements established in Section 2: *flexibility (F)*, *round-tripping (R)*, *web-based (W)*, and *language agnosticism (L)*. Each requirement is marked as fully met (✔), partially met ( ), or not met (·); question marks indicate uncertainty.

Some parser generators support limited flexibility through grammar composition. ANTLR [49], for example, supports grammar imports, enabling composition at the parser-level, but not at the rule-level. Parsers can be extended in a manner similar to object-oriented class inheritance, but individual rules cannot be deleted or patched while preserving references to the original implementation. Furthermore, mainstream parser generators, like ANTLR require a compilation step and typically produce only a Concrete Syntax Tree (CST), which reflects the full grammar without abstracting to

a higher-level AST. Producing an AST therefore requires an additional traversal and transformation pass, effectively splitting parsing into two stages and increasing complexity for tools that operate on the AST. While ANTLR theoretically supports targeting multiple host languages, including JavaScript and thus the Web, this generally produces only a CST without an AST. Crucially, none of these frameworks provides the combination of fine-grained extensibility, AST-level round-tripping, Web based execution, and query language agnostic design demanded by Traqula, motivating the need for a new modular approach.

To address these requirements, we selected Chevrotain [50] as the foundation for Traqula. Chevrotain can uniquely satisfy our requirements for flexibility, round-tripping, Web-based execution, and language-agnostic extensibility, while maintaining a sufficiently high-level abstraction. As a parsing toolkit, Chevrotain supports rule-level modularity: parser construction is expressed directly in TypeScript rather than a separate grammar language, enabling dynamic manipulation of individual rules, runtime extensions, and distribution of language-agnostic helpers as ordinary TypeScript modules. Unlike most parser generators, Chevrotain can parse directly into AST rather than CSTs, allowing round-trippable ASTs without an intermediate transformation. Finally, Chevrotain is highly optimised for Web-based execution environments, producing fast parsers that run efficiently in both browsers and server-side JavaScript. Together, these properties make Chevrotain a particularly strong match for Traqula's design goals.

3.3. AST Structures for Round-Tripping

To support Traqula's requirement for AST-level round-tripping, we examine two popular tools designed for reformatting and rewriting: Babel [66] and ESLint [67].

Babel [66] is a compiler that enables developers to write next-generation JavaScript and transpile it to earlier versions, ensuring compatibility with older environments. To support transformations while preserving the original source structure, Babel annotates its AST nodes with source location information, specifying the range of offsets each node represents in the original string. When a node is replaced, it can either be substituted with a raw source string, preserving user-visible syntax exactly, or with a new AST node, whose string representation is automatically generated. Auto-generation produces a valid string for the node; for example, a SPARQL string literal '`a`' could be generated as `'a'`, `"a"`, `'''a'''`, `"""a"""`, `'a'^^xsd:string`', etc. Similarly, SPARQL keywords are case-insensitive, so an auto-generated keyword may have any casing.

ESLint [67] is a widely used linter that also supports automatic fixes. Like Babel, ESLint requires AST nodes to carry source location information. Unlike Babel, fixes are applied externally through a helper that patches specific source ranges with new strings, rather than modifying the AST nodes directly. Despite this difference, the key principle of associating each node with its position in the source text remains central to enabling precise, reliable transformations.

4. Architecture

In this section, we go into more detail on the architecture implemented in Traqula, and how we extended the architectural vision towards query generation and transformation. We explain how Traqula exists of many small, interlinkable packages

and provide more detail on the core package, which contains the language-agnostic builders and transformers.

4.1. Builder-based Dependency Injection

At the core of Traqula's flexibility is its builder-based dependency injection architecture. Dependency injection [33] is a software design that promotes flexible software by creating objects or functions that receive the components they rely on, rather than instantiating them directly, allowing them to treat their dependencies more conceptually. In Traqula each functional element, such as a parser rule, generator rule, or transformation step, is declared under a symbolic name, and refers to other rules by name rather than by concrete implementation. This indirection enables users to replace or patch functionality simply by adjusting the mapping from names to functions. Fig. 3 (left) illustrates this idea, while Fig. 3 (right) shows Traqula's TypeScript definition of the `iri` parser and generator rules, each depending on named subrules.

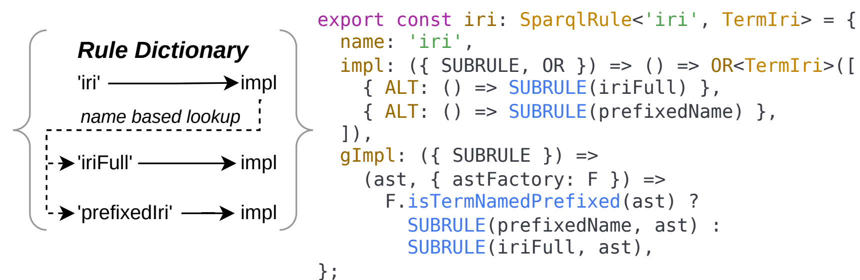

Fig. 3: Left: A mapping from rule names to their implementations, as registered by the user. In this example, the `iri` rule depends on the named rules *`iriFull`* and *`prefixedIri`*. At execution time, the implementation of `iri` consults the user-provided map to resolve these names to their concrete implementations.
Right: Traqula's TypeScript declaration of the same `iri` rule. The rule is defined under the name *iri*, and provides both a parser implementation (`impl`) and a generator implementation (`gImpl`). The use of *'SUBRULE'* reflects the same *name-based lookup* shown in the left figure, delegating execution to the implementations registered under the names `iriFull` and `prefixedIri`.

To manage and share these rule dictionaries, Traqula adopts the builder design pattern *(https://refactoring.guru/design-patterns/builder)*. Builders construct complex, modular objects step by step, and expose a uniform API for composing, extending, and patching rules. By implementing the builder pattern directly in the host language, Traqula avoids the need for additional compilation stages and allows builders to be referenced, extended, and combined programmatically. This choice also makes it possible to integrate language-specific features, most notably static type checking, into the dependency-injection mechanism itself. Traqula provides three task-specific builders: a parser builder, a generator builder, and a general indirection builder. All share the same underlying rule dictionary, differing only in the `build` artifact they produce, and the supplementary dependencies they inject.
Each builder exposes the same operations to manage the rule dictionary. Fig. 4 illustrates the core management functions, creating a builder, adding a rule, patching a rules, and building the final artifact .

```
const iriGeneratorBuilder = GeneratorBuilder
  .create([iriFull, prefixedIri])
  .addRule(iri);
const specialIriGenerator = GeneratorBuilder
  .create(iriGeneratorBuilder)
  .patchRule(alternativePrefixedRule)
  .build();
```

Fig. 4: Construction of a 'special IRI generator' based on an existing builder for building a IRI generator, patching it with an alternative rule implementation for the *'prefixedIri'* rule.

4.2. Modular Packages

To maximize independent evolution and minimise unnecessary dependencies, Traqula is composed of many small, interlinked packages rather than a single big monolithic package. A user only need to depend on what packages they actually use. For example, if a user only requires SPARQL 1.1 parsing, they can depend solely on *'@traqula/parser-sparql-1-1'*, without needing to pull in, and consider the SPARQL 1.2 parser, the generators, or the transformers Traqula maintains. Beyond flexibility and extensibility, this modular architecture is crucial for controlling bundle size, a key requirement for modern Web applications. By allowing developers to import only the necessary components, Traqula avoids the overhead of shipping unused functionality to the browser, improving load times and performance.
Modularity also reinforces well-defined and stable package interfaces: because Traqula itself is built from these packages, its APIs must remain open for extension. For maintainability and version control, all packages are managed as a monorepo *(https://monorepo.tools/)* in a single GitHub repository under the Comunica organisation: https://github.com/comunica/traqula.

4.3. Language-Agnostic Core Package

Following the modular package design, Traqula provides a central language-agnostic core package containing the generic builders and transformers. These components are designed to be reusable across different query languages, whether for dialects of SPARQL or entirely different languages such as SHACL-compact or GQL. The core package underpins Traqula's flexibility, enabling parsers, generators, and transformations to be composed in a language-independent manner.
A key feature of the core package is the generic transformers and visitors, which facilitate AST- and algebra-level transformations/ visiting. This transformer is type-safe, ensuring correctness, leveraging TypeScript's generics to adapt dynamically to the AST or algebra representation being manipulated. Beyond enabling rewrites and translations between query languages, the transformers also support AST-level reformatting, since reformatting can be expressed as a transformation on the AST level in the case of Traqula.
To use the transformer, one instantiates a new transformer object, specifying the types of AST nodes or algebra operations, and optionally providing a default transformation context. Transformations can be guided through preVisitors, allowing to 1. stop traversal completely; 2. continue traversal into descendants; 3. skip specified properties of the current node; 4. copy certain keys without traversal; and 5. shallowly copy the current node. The transformer maintains a stack of nodes to be transformed, ensuring that descendant nodes are processed before their parents. Fig. 5 illustrates a

simple algebraic transformation that wraps a *'distinct'* operation around the first *'project'* node it encounters. In addition, we provide a catalogue of known existing transformations powered by Traqula *(https://github.com/comunica/traqula/blob/main/docs/transformation-catalogue.md)*, thereby also demonstrating how Traqula can be used for traditional query optimizing or rewriting.

```
const transformed = new TransformerTyped<Sparql11Nodes>()
  .transformNode({ // A query as AST
    type: Algebra.Types.SLICE,
    input: {
      type: Algebra.Types.PROJECT,
      input: {
        type: Algebra.Types.JOIN,
        input: [{ type: Algebra.Types.PROJECT }, { type: Algebra.T
      },
    },
  }, {
    [Algebra.Types.PROJECT]: { // Transformation of projections
      preVisitor: () => ({ continue: false }),
      transform: projection => algebraFactory.createDistinct(proje
    },
  });
```

Fig. 5: Example usage of the transformer to wrap a *'distinct'* node around the first *'project'* node in an algebraic expression. Providing input query `SELECT * { { Select * { ?s ex:a ?o } } . ?s ex:b ?o . } LIMIT 20` would result in rewritten query `SELECT DISTINCT * { { Select * { ?s ex:a ?o } } . ?s ex:b ?o . } LIMIT 20`.

5. Implementation

This section outlines the technological foundations of Traqula, the SPARQL languages and dialects currently supported, and the maintenance plan that ensures the system remains reliable and extensible as query languages continue to evolve.

5.1. TypeScript

Traqula is implemented in TypeScript, following the rationale outlined in Section 2: TypeScript offers broad accessibility, quick prototyping, browser-server portability, and a familiar development environment for a large set of developers. Within Traqula, TypeScript empowers the dynamic, and generic APIs used by the builders and transformers, through its expressive type system, allowing IDEs to surface context-specific function signatures, prevent common integration mistakes, and support safe extensibility.
Traqula is developed openly on GitHub under the MIT licence, making the project straightforward to adopt, audit, modify, and extend. Documentation is provided through: 1. documentation pages maintained by the development team as part of exposed package READMEs (e.g. @traqula/parser-sparql-1-1 *(https://www.npmjs.com/search?q=traqula%20parser%20sparql%201)*), 2. dedicated

documentation guides *(https://github.com/comunica/traqula/blob/main/docs/index.md)*, and 3. JSDoc annotations that are directly visible within IDEs while also being exposed through an automatically generated documentation website *(https://comunica.github.io/traqula/)*. Together, these choices reduce the barrier for integrating Traqula into research prototypes, production systems, and community-driven extensions.

5.2. Current language support

Traqula currently supports SPARQL 1.1 [1], SPARQL 1.2 [68], and SPARQL 1.1 + ADJUST [25], providing the ability to transform queries to an AST and to their algebraic representation. Given an algebra, a corresponding AST can be created that produces the same algebra, and given an AST, a matching query string can be generated. Both the AST and algebra for all supported query languages can be transformed using type-safe functions from the core library. When an AST includes source location information, the generator produces a query string that preserves user-visible details, ensuring that semantically irrelevant information, such as spacing or keyword capitalization, is maintained.
To ensure correctness, Traqula includes extensive integration tests. The project currently incorporates 6269 tests, achieving 94% line coverage, combining tests authored specifically for Traqula with those inherited from SPARQL.js [58] and SPARQLAlgebra.js [69], two widely used projects that have been deprecated in favor of Traqula. In addition, Traqula leverages the RDF Tests Community Group's test suites, which provide a broad set of positive and negative tests for implementers of RDF technologies. To further increase maturity, reliability, and facilitate future contributions, the authors are committed to further expanding Traqula's test coverage to 100% through additional unit and integration tests.

5.3. Maintenance and sustainability plan

A clear maintenance plan is essential for both the authors of a software project and for users who rely on it. Traqula's source code is openly available as free software under the MIT license, which allows anyone to contribute, fork, or maintain their own version. This ensures that, even if the original maintainers step away, the project can continue to evolve.
However, unmaintained software remains a concern for dependent projects regardless of licensing. To address this, Traqula is maintained under the Comunica association, which distributes responsibility across multiple contributors rather than relying on a single individual. Furthermore, the association provides a mechanism for financial incentives, such as bounties, allowing organizations or users to fund the implementation of specific features or changes. This structured support increases the trustworthiness and long-term sustainability of the project, providing confidence to both researchers and practitioners who depend on Traqula.

6. Performance analysis

While the primary focus of Traqula is flexibility and expressivity, sufficient performance is still important in broadly used software tools. As such, we evaluate the execution-time performance of Traqula, comparing its different parsing modes,

contrasting it with the SPARQL.js [58] parser plus the algebra transformation software SPARQLAlgebra.js [69], and highlighting the impact of parser reuse. All measurements were conducted on Fedora Linux 43 (Workstation Edition), equipped with an Intel® Core™ Ultra 7 165U × 14 processor, having 16 GiB of RAM. We measured the execution times of parsing 50 real-world SPARQL 1.1 queries from the DBpedia SPARQL Benchmark [70] in each of the tools and provide the scripts for reproducibility *(https://github.com/comunica/traqula/tree/c04d10db2ece3bad5f5924c3d511c1a6cd1fe615/eval)*.

Fig. 7 shows the execution times of Traqula across different targets and parser configurations. Patching a modular SPARQL 1.1 parser to support SPARQL 1.2 results in a small but statistically significant increase in execution time ($p < 0.05$), with a mean increase of 10.8% . Tracking source information to support round-tripping introduces additional overhead due to the more complex lexing process, with mean increases of 18.8% and 16.6% for SPARQL 1.1 and 1.2 parsing, respectively (both $p < 0.05$). Parsing directly into algebra, effectively performing an AST transformation after parsing, further increases execution time, with mean increases of 45.6% for SPARQL 1.1 and 43.0% for SPARQL 1.2 ($p < 0.05$). We also carried out these same experiments for the 199 manually curated SPARQL 1.1 queries within Traqula's unit tests and reach the same conclusions. For space reasons, we omit these results *(https://github.com/comunica/traqula/tree/c04d10db2ece3bad5f5924c3d511c1a6cd1fe615/eval/res-own-bench)* from this paper.

Despite these relative execution time increases, Fig. 7 shows Traqula remains substantially faster than existing solutions. The mean execution time for parsing into an AST using Traqula is 2.5 ms, compared to 23.93 ms for SPARQL.js, a 957.2% difference, even though Traqula generates a more complete and correct AST. Parsing into algebra shows a similar trend: Traqula takes 3.64 ms, while SPARQLAlgebra.js (using SPARQL.js) requires 24.04 ms, a 660.4% difference. Much of this advantage can be attributed to the performance of Chevrotain, the parser toolkit underlying Traqula.

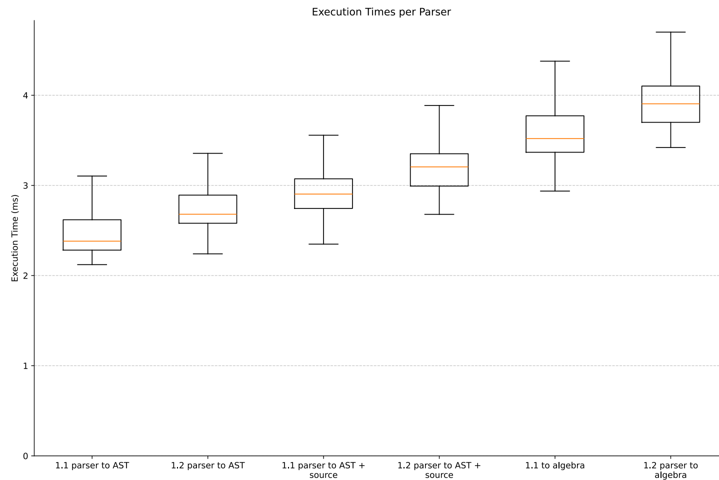

Fig. 7: Variance of execution times (ms) for Traqula's SPARQL 1.1 and 1.2 parsers, when parsing 50 real world SPARQL 1.1 queries [70] using different configurations. Parsing SPARQL 1.2 is always slightly slower than parsing SPARQL 1.1 due to increased grammar complexity, and source tracking or algebra transformation further increase effort.

Finally, it is important to recognise the cost associated with using a parser toolkit in a non-pre-compiled language like TypeScript. Traqula's underlying parser toolkit, Chevrotain, performs its parser optimizations at runtime, making parser construction computationally expensive. To illustrate, parsing the 50 queries with a prebuilt and reused parser has a mean execution time of 2.5 ms, whereas creating a new parser for each query results in a mean execution time of 689.31 ms; a slowdown of 275.7 times. Moreover, because Chevrotain is optimised for JavaScript's V8 engine, many performance benefits rely on JIT compilation. Recreating the parser repeatedly may prevent or invalidate JIT optimizations, further degrading performance. This makes parser reuse essential in practice, as already adopted in systems such as the Comunica query engine.

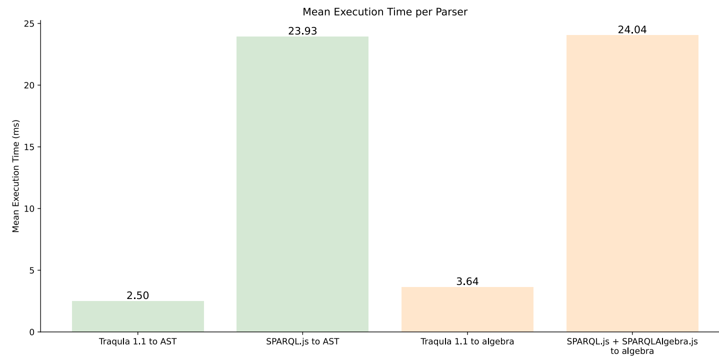

Fig. 7: Execution time (ms) comparison between Traqula and the SPARQL.js [58] parser plus the algebra transformation software SPARQLAlgebra.js [69] when parsing 50 real-world SPARQL 1.1 queries [70]. Parsing to both an AST (green) and to algebra (orange) is significantly faster using Traqula.

7. Conclusion

In this paper, we introduced Traqula, a framework for modular query parsing, generation, and transformation. We argue that a modular parser is essential for the future of SPARQL querying, particularly in increasingly heterogeneous federated environments. By decoupling the various steps in query manipulation, Traqula enables rapid experimentation with new or modified language features. Furthermore, it facilitates query rewriting between different languages and supports the creation of modular query tooling, such as linters and reformatters, through easy AST and algebra manipulation.

Traqula is designed for widespread adoption. It has already been integrated into the modular querying framework Comunica *(https://github.com/comunica/comunica/pull/1614)*. In addition, we demonstrate its usability for extending language features as part of a Comunica tutorial *(https://comunica.dev/docs/modify/getting_started/contribute_new_operation/)*. The framework is openly available on GitHub under the MIT license and is maintained by the Comunica association, ensuring long-term sustainability.

Looking forward, Traqula's modular architecture and developer-friendly APIs pave the way for supporting additional query languages, such as SHACL Compact Syntax and GQL.

Acknowledgements

Jitse De Smet is a predoctoral fellow of the Research Foundation – Flanders (FWO) (1SB8525N). Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1202124N).

References

1. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C, https://www.w3.org/TR/2013/REC-sparql11-query-20130321/ (2013).

2. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1: Concepts and Abstract Syntax. W3C, https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (2014).
3. Feigenbaum, L., Todd Williams, G., Grant Clark, K., Torres, E.: SPARQL 1.1 Protocol. W3C, https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/ (2013).
4. Verborgh, R., Sande, M.V., Hartig, O., Herwegen, J.V., Vocht, L.D., Meester, B.D., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. J. Web Semant. 37-38, 184–206 (2016). doi:10.1016/J.WEBSEM.2016.03.003
5. Azzam, A., Fernández, J.D., Acosta, M., Beno, M., Polleres, A.: SMART-KG: hybrid shipping for SPARQL querying on the web. In: Proceedings of The Web Conference 2020. pp. 984–994 (2020).
6. Minier, T., Skaf-Molli, H., Molli, P.: SaGe: Web preemption for public SPARQL query services. In: The World Wide Web Conference. pp. 1268–1278 (2019).
7. Azzam, A., Aebeloe, C., Montoya, G., Keles, I., Polleres, A., Hose, K.: WiseKG: Balanced access to web knowledge graphs. In: Proceedings of the Web Conference 2021. pp. 1422–1434 (2021).
8. Hartig, O., Buil-Aranda, C.: Bindings-restricted triple pattern fragments. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 762–779. Springer (2016).
9. Pham, T.H.T., Montoya, G., Nédelec, B., Skaf-Molli, H., Molli, P.: Passage: Ensuring Completeness and Responsiveness of Public SPARQL Endpoints with SPARQL Continuation Queries. In: Proceedings of the ACM on Web Conference 2025. pp. 47–58 (2025).
10. Aranda, C.B., Prud'hommeaux, E.: SPARQL 1.1 Federated Query. W3C, https://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/ (2013).
11. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization Techniques for Federated Query Processing on Linked Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., and Blomqvist, E. (eds.) The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I. pp. 601–616. Springer (2011). doi:10.1007/978-3-642-25073-6_38
12. Saleem, M., Ngomo, A.-C.N.: HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., and Tordai, A. (eds.) The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings. pp. 176–191. Springer (2014). doi:10.1007/978-3-319-07443-6_13
13. Görlitz, O., Staab, S.: SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. COLD. (2011).
14. Hanski, J., Crum, E., Taelman, R.: Observations on automated client-side query federation over Wikidata SPARQL endpoints. In: Proceedings of the Wikidata Workshop 2025 co-located with 24th International Semantic Web Conference (ISWC 2025) (2025).
15. Taelman, R., Herwegen, J.V., Sande, M.V., Verborgh, R.: Comunica: A Modular SPARQL Query Engine for the Web. In: Vrandecic, D., Bontcheva, K., Suárez-Figueroa, M.C., Presutti, V., Celino, I., Sabou, M., Kaffee, L.-A., and Simperl, E. (eds.) The Semantic Web - ISWC 2018 - 17th International Semantic Web

Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II. pp. 239–255. Springer (2018). doi:10.1007/978-3-030-00668-6_15
16. Heling, L., Acosta, M.: Federated SPARQL Query Processing over Heterogeneous Linked Data Fragments. In: Laforest, F., Troncy, R., Simperl, E., Agarwal, D., Gionis, A., Herman, I., and Médini, L. (eds.) WWW ’22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022. pp. 1047–1057. ACM (2022). doi:10.1145/3485447.3511947
17. Cheng, S., Hartig, O.: FedQPL: A Language for Logical Query Plans over Heterogeneous Federations of RDF Data Sources. In: Indrawan-Santiago, M., Pardede, E., Salvadori, I.L., Steinbauer, M., Khalil, I., and Kotsis, G. (eds.) iiWAS ’20: The 22nd International Conference on Information Integration and Web-based Applications & Services, Virtual Event / Chiang Mai, Thailand, November 30 - December 2, 2020. pp. 436–445. ACM (2020). doi:10.1145/3428757.3429120
18. Montoya, G., Aebeloe, C., Hose, K.: Towards Efficient Query Processing over Heterogeneous RDF Interfaces. In: Verborgh, R., Kuhn, T., and Berners-Lee, T. (eds.) Proceedings of the 2nd Workshop on Decentralizing the Semantic Web co-located with the 17th International Semantic Web Conference, DeSemWeb@ISWC 2018, Monterey, California, USA, October 8, 2018. CEUR-WS.org (2018).
19. Battle, R., Kolas, D.: Geosparql: enabling a geospatial semantic web. Semantic Web Journal. 3, 355–370 (2011).
20. Hartig, O.: Querying trust in rdf data with tsparql. In: European Semantic Web Conference. pp. 5–20. Springer (2009).
21. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: Proceedings of the 18th international conference on World wide web. pp. 1061–1062 (2009).
22. Pelgrin, O., Taelman, R., Galárraga, L., Hose, K.: The Need for Better RDF Archiving Benchmarks. In: Proceedings of the 9th Workshop on Managing the Evolution and Preservation of the Data Web (2023).
23. Virtuoso: Using Full Text Search in SPARQL. https://docs.openlinksw.com/virtuoso/sparqlextensions/#rdfsparqlrulefulltext (2024).
24. Apache Jena: ARQ - Construct Quad. https://jena.apache.org/documentation/query/construct-quad.html#Grammar (2024).
25. Williams, G.T.: SPARQL Dev: Add Support Durations, Dates, and Times. https://github.com/w3c/sparql-dev/blob/7350808a4f0b7ca7c2a8d823b4e7d8958dfb073b/SEP/SEP-0002/sep-0002.md (2025).
26. Oxigraph: SEP 0002: calendar and duration operations. https://github.com/oxigraph/oxigraph/wiki/SPARQL#sep-0002-calendar-and-duration-operations (2024).
27. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34, 16:1–16:45 (2009). doi:10.1145/1567274.1567278
28. Gschwend, A., Lassila, O.: RDF & SPARQL Working Group Charter. (2025).
29. Smet, J.D., Taelman, R.: Towards tackling SPARQL heterogeneity through modular parsing. In: Chaves-Fraga, D., Heibi, I., Garijo, D., Collarana, D., Salatino, A.A., and Vahdati, S. (eds.) Joint Proceedings of Posters, Demos, Workshops, and Tutorials of the 21st International Conference on Semantic Systems co-located with 21st International Conference on Semantic Systems

(SEMANTiCS 2025), Vienna, Austria, September 3-5, 2025. CEUR-WS.org (2025).
30. Hartig, O., Champin, P.-A., Kellogg, G., Seaborne, A.: SPARQL 1.2 Query Language. https://www.w3.org/TR/rdf12-concepts/ (2025).
31. Information technology - Database languages - SQL. International Organization for Standardization, Geneva, CH, https://www.iso.org/standard/76583.html (2023).
32. Williams, G.: SPARQL 1.1 Service Description. W3C, https://www.w3.org/TR/2013/REC-sparql11-service-description-20130321/ (2013).
33. Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern. https://martinfowler.com/articles/injection.html (2004).
34. Rietveld, L., Hoekstra, R.: The YASGUI family of SPARQL clients. Semantic Web. 8, 373–383 (2017). doi:10.3233/SW-150197
35. Emonet, V., Sima, A.-C., Mendes de Farias, T.: A User-Friendly SPARQL Query Editor Powered by Lightweight Metadata. In: The Semantic Web: ESWC 2025 Satellite Events: Portoroz, Slovenia, June 1–5, 2025, Proceedings. pp. 3–7. Springer-Verlag, Berlin, Heidelberg (2025). doi:10.1007/978-3-031-99554-5_1
36. Nezis, I.: Qlue-ls, a powerful SPARQL language server. https://ad-publications.cs.uni-freiburg.de/theses/Bachelor_Ioannis_Nezis_2025.pdf (2025).
37. GitHub: Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1. https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/ (2025).
38. Alexiev, V., Wright, J.: SHACL Compact Syntax. https://w3c.github.io/data-shapes/shacl12-cs/ (2025).
39. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A generic language for integrated RDF mappings of heterogeneous data. Ldow. 1184, (2014).
40. Min Oo, S., De Meester, B., Taelman, R., Colpaert, P.: Algebraic Mapping Operators for Knowledge Graph Generation. Semantic Web Journal. (2025).
41. Hartig, O., Pérez, J.: LDQL: A query language for the web of linked data. Journal of Web Semantics. 41, 9–29 (2016).
42. Hartig, O., Freytag, J.-C.: Foundations of Traversal based Query Execution over Linked Data. In: Proceedings of the 23rd ACM conference on Hypertext and social media. pp. 43–52. ACM (2012).
43. Taelman, R., Verborgh, R.: Evaluation of Link Traversal Query Execution over Decentralized Environments with Structural Assumptions. CoRR. abs/2302.06933, (2023). doi:10.48550/ARXIV.2302.06933
44. Information technology — Database languages — GQL. International Organization for Standardization, Geneva, CH, https://www.iso.org/standard/76120.html (2024).
45. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, https://www.worldcat.org/oclc/12285707 (1986).
46. Sowinski, P., Bogacka, K., Danilenka, A., Kozlov, N.: Jelly: a Fast and Convenient RDF Serialization Format. In: Chaves-Fraga, D., Heibi, I., Garijo, D., Collarana, D., Salatino, A.A., and Vahdati, S. (eds.) Joint Proceedings of Posters, Demos, Workshops, and Tutorials of the 21st International Conference on Semantic Systems co-located with 21st International Conference on Semantic Systems (SEMANTiCS 2025), Vienna, Austria, September 3-5, 2025. CEUR-WS.org (2025).

47. Carothers, G., Seaborne, A.: RDF 1.1 N-Triples. W3C, https://www.w3.org/TR/2014/REC-n-triples-20140225/ (2014).
48. GNU: GNU Bison. https://www.gnu.org/software/bison/ (2025).
49. Parr, T.J., Quong, R.W.: ANTLR: A Predicated - LL(k) Parser Generator. Softw. Pract. Exp. 25, 789–810 (1995). doi:10.1002/SPE.4380250705
50. Chevrotain, Soel, S.: Chevrotain - Parser Building Toolkit for JavaScript. https://github.com/Chevrotain/chevrotain/ (2025).
51. Facebook: GraphQL. https://spec.graphql.org/September2025/# (2025).
52. Finné, M., Demianenko, M., Melke, P., Nawroth, A.: Neo4j. https://github.com/neo4j/neo4j
53. Seaborne, A.: Apache: Jena. https://github.com/apache/jena
54. Thompson, B.: Blazegraph. https://github.com/blazegraph/database/
55. Pellissier Tanon, T.: Oxigraph. https://doi.org/10.5281/zenodo.7408022 (2025). doi:10.5281/zenodo.7408022
56. Kalmbach, J., Kalmbach, J.: QLever. https://github.com/ad-freiburg/qlever/
57. Ottestad, H.M., Broekstra, J.: RDF4J. https://github.com/eclipse-rdf4j/rdf4j
58. Verborgh, R., Braeckel, S.V., FlorianFV, Taelman, R., Jusevičius, M., Morozov, A., Herwegen, J.V., Arndt, N., Beeke, D., Wright, J., Trame, J., Ermilov, T., Rietveld, L., Rodriguez, E., Aaronsohn, I., Scazzosi, J., DuPont, J.M., Balhoff, J., Werkmeister, L., de Bayser, M., Jaros, P., Reynolds, S., Tanon, T.: RubenVerborgh/SPARQL.js. https://doi.org/10.5281/zenodo.597487 (2025). doi:10.5281/zenodo.597487
59. Barrett, A., Rogers, J.: Stardog: Millan. https://github.com/stardog-union/millan
60. van Kleef, P., Iliev, M.: Virtuoso Open-Source Edition. https://github.com/openlink/virtuoso-opensource
61. Raasveldt, M., Muehleisen, H.: DuckDB. https://github.com/duckdb/duckdb
62. Momjian, B., Eisentraut, P., Lane, T.: PostgreSQL Database Management System. https://github.com/postgres/postgres
63. community, S.Q.L.: SQLite. https://github.com/sqlite/sqlite/
64. Goncharov, I., Byron, L.: GraphQL.js. https://github.com/graphql/graphql-js
65. Burbidge, M., Wileński, D.: OpenGQL: GQL ANTLR. https://github.com/opengql/grammar
66. sebmck, nicolo-ribaudo, hzoo: Babel. https://github.com/babel/babel (2025).
67. Nicholas, C.Z.: ESLint. https://github.com/eslint/eslint/ (2025).
68. Hartig, O., Seaborne, A., Taelman, R., Williams, G., Tanon, T.P.: SPARQL 1.2 Query Language. https://www.w3.org/TR/sparql12-query/ (2025).
69. Herwegen, J.V.: SPARQLAlgebra.js. https://github.com/joachimvh/SPARQLAlgebra.js
70. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.-C.N.: DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., and Blomqvist, E. (eds.) The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I. pp. 454–469. Springer (2011). doi:10.1007/978-3-642-25073-6_29