GQL Datatypes as an RDF Vocabulary

Dominik Tomaszuk[1], Daniel Szałaj[1]

[1]Faculty of Computer Science, University of Bialystok, Poland

Abstract

The ISO GQL standard introduces a rich value-type system for property graphs, including scalar, temporal, byte-string, list, record, and dynamic union datatypes. The vocabulary presented here additionally covers a vector extension. At the same time, RDF-based knowledge graph systems identify literal datatypes by IRIs and interpret typed literals through value spaces, lexical spaces, and lexical-to-value mappings. This paper presents the *GQL Datatypes RDF Vocabulary*, a lightweight RDF vocabulary that represents selected GQL datatypes and the vector extension as RDF datatype IRIs and describes their hierarchy, aliases, and selected constraints using RDF, RDFS, OWL, and XML Schema Datatypes vocabulary terms. The vocabulary is intended to support interoperability between GQL-oriented property graph systems and RDF-based knowledge graph systems. We describe its namespace and datatype model, summarize its datatype coverage, and show how the accompanying generator and validator use the vocabulary. In the reference implementation, RDF literals retain GQL datatype IRIs; XSD mappings declared in the vocabulary support scalar validation, while lists, records, and vectors are encoded directly as JSON lexical forms of vocabulary-typed literals.

Keywords

GQL, RDF, datatypes, property graphs, knowledge graphs, RDF literals, interoperability

1. Introduction

Knowledge graph systems increasingly need to exchange data across heterogeneous graph models. RDF systems represent information as triples or datasets and identify literal datatypes by IRIs. Property graph systems, in contrast, usually expose values through a host-language or database-language type system in which node and edge properties may contain booleans, strings, numbers, dates, lists, records, vectors, or other structured values. The standardization of GQL as a database language for property graphs makes this type-system gap more visible: GQL provides a common language-level vocabulary for property graph values, whereas RDF provides a common Web-level mechanism for identifying and interpreting typed literals [1, 2].

This gap is important for data management for knowledge graphs. Many workflows combine RDF-based KGs with property graph systems, for example, when RDF graphs are loaded into graph databases, when property graph data is exported as RDF, when graph embeddings are attached to KG entities, or when validation and schema artifacts are exchanged between graph ecosystems. In these settings, datatype information is often either lost, flattened into generic strings, or approximated by existing XML Schema datatypes. Such approximations are sufficient for common scalar values, but they are less satisfactory for GQL-specific or constructed values such as lists, records, vectors, dynamic union values, and implementation-defined numeric ranges.

This paper presents a short description of the *GQL Datatypes RDF Vocabulary*. The vocabulary assigns RDF datatype IRIs to selected GQL value datatypes and records datatype relationships and constraints using standard RDF vocabulary terms. It is not a replacement for the GQL language specification and does not define a complete GQL parser. Instead, it provides a machine-readable bridge between the GQL value-type vocabulary and RDF's typed literal mechanism. As such, it can support graph model interoperability, datatype-aware validation, schema exchange, and documentation of datatype assumptions in hybrid RDF/property-graph data management pipelines.

The contribution of this paper is threefold:

1. we describe a compact RDF representation of GQL datatype names as datatype IRIs;

DMKG'26: 2nd International Workshop on Data Management for Knowledge Graphs, October 2026, Bari, Italy

dominik.tomaszuk@uwb.edu.pl (D. Tomaszuk)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-09

CEUR-WS.org/Vol-4322/paper_5.pdf

2. we outline a datatype model based on value spaces, lexical spaces, and lexical-to-value mappings, aligned with RDF and XML Schema Datatypes;

3. we present a validation-oriented representation strategy for GQL-typed RDF literals, using ontology-derived XSD mappings for scalar checks and direct JSON lexical forms for constructed datatypes.

The remainder covers the motivation (Section 2), vocabulary design (Section 3), datatype coverage and mapping (Section 4), implementation and evaluation (Section 5), related work (Section 6), and conclusions (Section 7).

2. Motivation

RDF provides a general mechanism for typed literals. A literal such as `"42"^^xsd:integer` is interpreted using the datatype IRI, its lexical space, its value space, and the lexical-to-value mapping defined for that datatype. This design is extensible: applications may introduce additional datatype IRIs, provided that their interpretation is specified.

However, datatype interoperability between RDF and property graphs remains incomplete. Many GQL scalar values can be mapped naturally to existing XSD datatypes, for example, booleans to `xsd:boolean`, signed integers to `xsd:integer`, unsigned integers to `xsd:nonNegativeInteger`, strings to `xsd:string`, and dates to `xsd:date`. Other GQL values have no complete XSD counterpart, including dynamic union values, records, lists, vectors, and approximate numeric values whose precision and range may not be preserved by narrowing them to `xsd:float` or `xsd:double`.

The issue is also practical: datatypes influence validation, query evaluation, indexing, storage layout, and data quality. In large-scale KGs, inconsistent handling of datatypes may lead to inefficient queries, failed joins, or subtle semantic errors. A shared datatype vocabulary can help systems document and exchange datatype assumptions, for example, by preserving that a source value was a `gql:INT8` even if the materialized RDF literal is represented as `xsd:integer`.

Recent work on composite RDF literals for lists and maps argues that modeling composite values using RDF collections or containers is verbose and difficult to query and update in SPARQL, and proposes dedicated literal datatypes for compact list and map values [3, 4]. The GQL Datatypes RDF Vocabulary follows a similar intuition, but with a different scope: it provides an RDF vocabulary for identifying and constraining GQL value datatypes and validating RDF literals that retain those datatype IRIs.

3. Vocabulary Design

3.1. Namespace and Core Terms

The prototype vocabulary uses the namespace http://example.org/gql-types# with the preferred prefix `gql`. Each datatype represented by the vocabulary is identified by an IRI in this namespace. For example, `gql:BOOLEAN`, `gql:INT64`, `gql:STRING`, `gql:DATE`, `gql:LIST`, `gql:RECORD`, and `gql:VECTOR` are datatype IRIs.

The vocabulary uses `rdfs:Datatype` to type every datatype IRI, `rdfs:subClassOf` to represent direct datatype generalization, `owl:equivalentClass` to represent aliases or equivalent datatype names, and `owl:withRestrictions` to encode validator-oriented XSD-style constraints. The ontology also uses `owl:onDatatype`; these terms are consumed as validation metadata and do not establish complete OWL 2 semantics for GQL. A representative fragment is shown in Listing 1.

Listing 1: Illustrative fragment of the GQL datatype vocabulary.

```
@prefix gql:  <http://example.org/gql-types#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
```

```
gql:BOOLEAN
    a rdfs:Datatype ;
    rdfs:label "BOOLEAN"@en ;
    rdfs:subClassOf gql:BOOLEAN_DATA ;
    owl:withRestrictions (
        [ xsd:pattern "TRUE|FALSE|true|false|1|0" ]
    ) .

gql:BOOL
    a rdfs:Datatype ;
    rdfs:label "BOOL"@en ;
    rdfs:subClassOf gql:BOOLEAN_DATA ;
    owl:equivalentClass gql:BOOLEAN .
```

3.2. Datatype Model

Each datatype is described, where applicable, with reference to the usual three-part datatype model:

Value space. The set of abstract values denoted by well-formed literals of the datatype.

Lexical space. The set of Unicode strings accepted as lexical forms.

Lexical-to-value mapping. The mapping from accepted lexical forms to values in the value space.

The vocabulary records selected validator-oriented constraining facets using OWL and XML Schema Datatypes vocabulary terms. These facets are interpreted conjunctively by the reference validator. Thus, if a datatype is associated with both a pattern and a numeric bound, a valid lexical form must satisfy the pattern and denote a value satisfying the bound. For scalar datatypes, these constraints can often be expressed precisely using XSD facets [5]. For constructed datatypes, such as lists, records, and vectors, the implementation uses JSON lexical forms on literals whose datatype remains the corresponding GQL datatype IRI; the inner parsing is implemented in the validator rather than as RDF or OWL datatype semantics.

For the purposes of this paper, the vocabulary can be described as a tuple

$$\mathscr{V} = (D, H, A, B, R),$$

where D is the finite set of GQL datatype IRIs, $H \subseteq D \times D$ is the hierarchy relation encoded with `rdfs:subClassOf`, $A \subseteq D \times D$ is the alias relation between GQL datatypes, B is a partial mapping from GQL datatypes to XSD validation bases, and R assigns lexical or value restrictions to selected datatypes.

This distinction is deliberate. The vocabulary does not attempt to provide a complete RDF datatype semantics for all GQL datatypes. In particular, constructed GQL datatypes such as lists and records, and extension types such as vectors, are treated as vocabulary-typed literals whose lexical forms are validated by the reference validator. Thus, the vocabulary should be understood as a lightweight datatype identification and validation profile rather than as a full formal reconstruction of the GQL value system in RDF semantics.

4. Datatype Coverage and Mapping

4.1. Datatype Coverage

The current vocabulary contains 87 `rdfs:Datatype` definitions. They can be grouped into several families: abstract base datatypes, dynamic union datatypes, Boolean datatypes, numeric datatypes, character string datatypes, binary string datatypes, temporal datatypes, list datatypes, record datatypes, and vector datatypes. The vector family is introduced by this prototype to represent numeric embeddings; it is not defined by ISO GQL.

Table 1
Representative datatype families in the RDF vocabulary.

Family	Examples	RDF role
Boolean	`gql:BOOLEAN`, `gql:BOOL`	typed scalar values
Integer	`gql:INT8`, `gql:INT64`, `gql:INT256`	exact numeric values
Unsigned integer	`gql:UINT8`, `gql:UINT64`	non-negative exact values
Decimal	`gql:DECIMAL`, `gql:DEC`	exact decimal values
Approximate numeric	`gql:FLOAT`, `gql:DOUBLE`	precision-sensitive values
String	`gql:CHAR`, `gql:VARCHAR`, `gql:STRING`	Unicode strings
Binary	`gql:BINARY`, `gql:VARBINARY`, `gql:BYTES`	octet sequences
Temporal	`gql:DATE`, `gql:ZONED_DATETIME`	temporal values
Constructed	`gql:LIST`, `gql:RECORD`, `gql:VECTOR`	structured values
Dynamic	`gql:ANY`, `gql:ANY_VALUE`	union-like values

Its coverage can be summarized by the family tuple

$$\mathscr{C} = (D_{\text{base}}, D_{\text{dynamic}}, D_{\text{boolean}}, D_{\text{numeric}}, D_{\text{string}}, D_{\text{binary}}, D_{\text{temporal}}, D_{\text{list}}, D_{\text{record}}, D_{\text{vector}}),$$

where each component is a subset of D and the vocabulary-level coverage is their union. Table 1 summarizes representative datatype families and their intended RDF role.

The hierarchy is useful for at least two reasons. First, it makes GQL datatype organization available to RDF tools. For instance, a validator can discover that `gql:INT8` is an exact numeric datatype by following `rdfs:subClassOf` links. Second, it enables schema-level mappings that preserve more information than direct conversion to generic RDF literals. For example, even if a value is materialized as an `xsd:integer` literal for compatibility, the source schema may still state that the originating GQL datatype was `gql:INT8`.

4.2. Mapping GQL Values to RDF Literals

The reference implementation does not normalize generated values to XSD literals or `rdf:JSON` literals. Instead, RDF literals retain GQL datatype IRIs, and the ontology supplies XSD mappings and restrictions used during validation. Constructed values are represented as JSON lexical forms on GQL-typed literals. Table 2 summarizes the validation targets and lexical conventions used by this implementation.

A GQL-typed RDF literal is modeled here as a pair

$$l = (x, d),$$

where x is the lexical form and $d \in D$ is a GQL datatype IRI. Validation is represented by the partial decision function

$$\text{valid}(x, d) = \begin{cases} \text{jsonShape}(x, d), & d \in D_{\text{list}} \cup D_{\text{record}} \cup D_{\text{vector}}, \\ \text{booleanLexical}(x), & d \in D_{\text{boolean}}, \\ \text{scalarValid}(x, d, B(d)) \wedge \text{facets}(x, R(d)), & B(d) \text{ is defined}, \\ \bot, & \text{otherwise}. \end{cases}$$

The cases cover JSON structures, case-insensitive Boolean lexical forms, and other scalars with ontology-derived XSD validation bases and restrictions. Here, $\text{facets}(x, R(d))$ is true when x satisfies every applicable pattern, length constraint, and numeric bound in $R(d)$.

The use of JSON is limited to the lexical representation of constructed values [6]. JSON was chosen because RDF 1.2 recognizes JSON as a datatype through `rdf:JSON` [2], making it a natural notation for structured literal values. However, this design deliberately differs from representing such values directly as `rdf:JSON` literals. The datatype IRI remains, for example, `gql:ARRAY`, `gql:VECTOR`, or

Table 2
Representative validation targets for GQL-typed RDF literals.

GQL datatype or family	Validator target
`gql:BOOLEAN`, `gql:BOOL`	`xsd:boolean`
`gql:INT8`, `gql:INT16`, `gql:INT32`	`xsd:byte`, `xsd:short`, `xsd:int`
`gql:INT64`, `gql:INT128`, `gql:INT256`	`xsd:long` or `xsd:integer`
`gql:INTEGER`, `gql:BIGINT`, `gql:SMALLINT`	`xsd:integer`
`gql:SIGNED_INTEGER`, `gql:SIGNED_BIG_INTEGER`	`xsd:integer`
`gql:UINT8`, `gql:UINT16`, `gql:UINT32`	`xsd:unsignedByte`, `xsd:unsignedShort`, `xsd:unsignedInt`
`gql:UINT64`, `gql:UINT128`, `gql:UINT256`	`xsd:unsignedLong` or `xsd:nonNegativeInteger`
`gql:UNSIGNED_INTEGER`, `gql:UBIGINT`, `gql:USMALLINT`	`xsd:nonNegativeInteger`
`gql:DECIMAL`, `gql:DEC`	`xsd:decimal`
`gql:FLOAT`, `gql:REAL`	`xsd:double`
`gql:DOUBLE`, `gql:DOUBLE_PRECISION`	not mapped in validator
`gql:FLOAT16`, `gql:FLOAT32`, `gql:FLOAT64`	`xsd:float` or `xsd:double`
`gql:FLOAT128`, `gql:FLOAT256`	`xsd:double`
`gql:CHAR`, `gql:VARCHAR`, `gql:STRING`	`xsd:string`
`gql:BINARY`, `gql:VARBINARY`, `gql:BYTES`	`xsd:hexBinary`
`gql:DATE`	`xsd:date`
`gql:LOCAL_TIME`, `gql:TIME_WITHOUT_TIME_ZONE`	`xsd:time`
`gql:ZONED_TIME`, `gql:TIME_WITH_TIME_ZONE`	`xsd:time`
`gql:LOCAL_DATETIME`, `gql:TIMESTAMP`	`xsd:dateTime`
`gql:TIMESTAMP_WITHOUT_TIME_ZONE`	`xsd:dateTime`
`gql:ZONED_DATETIME`, `gql:TIMESTAMP_WITH_TIME_ZONE`	`xsd:dateTime`
`gql:DURATION_YEAR_TO_MONTH`	`xsd:duration`
`gql:DURATION_DAY_TO_SECOND`	`xsd:duration`
`gql:LIST`, `gql:ARRAY`, `gql:GROUP_LIST`	JSON array lexical form
`gql:GROUP_ARRAY`	JSON array lexical form
`gql:RECORD`, `gql:ANY_RECORD`, `gql:UNIT_RECORD`	JSON object lexical form
`gql:VECTOR`	JSON numeric array lexical form

Algorithm 1: Validation of GQL-typed RDF literals in the reference implementation.

Input: A typed RDF literal l and the GQL datatype ontology
Output: A validation result
$d \leftarrow \textsc{LocalName}(\textsc{Datatype}(l))$
$x \leftarrow \textsc{LexicalForm}(l)$
if *d is a list, record, unit record, or vector datatype* **then**
  $j \leftarrow \textsc{ParseJSON}(x)$
  return $\textsc{CheckExpectedJSONShape}(j, d)$
if *d is BOOLEAN or BOOL* **then**
  return $\textsc{Lowercase}(x) \in \{\texttt{true}, \texttt{false}, 1, 0\}$
$t \leftarrow \textsc{OntologyXSDMapping}(d)$
if *t exists* **then**
  return $\textsc{CheckScalarLexicalForm}(x, d, t) \wedge \textsc{CheckOntologyFacets}(x, d)$
return $\textsc{UnknownDatatypeError}(d)$

`gql:UNIT_RECORD`, so that the source datatype is preserved; the JSON text itself is the lexical form. The validation procedure is summarized in Algorithm 1.

The validator first handles datatypes whose lexical forms need structural parsing. Lists, arrays, group lists, group arrays, and vectors must parse as JSON arrays; vectors are additionally restricted to numeric entries. Records and any-records must parse as JSON objects, and unit records must parse as an empty JSON object. Booleans use a case-insensitive lexical check. Other scalar datatypes

are checked through the ontology: `owl:equivalentClass` supplies the XSD base datatype, while `owl:withRestrictions` supplies patterns, numeric bounds, and length constraints. Datatypes that are neither explicitly structural nor mapped to an XSD datatype are reported as unknown by the reference validator.

4.3. Examples

Consider a property graph row or property record containing a Boolean flag, a small integer score, a timestamp, a list of tags, and a vector embedding. The RDF fragment in Listing 2 illustrates how such values can be represented using the GQL datatype IRIs.

The example can be written as a property tuple with an associated datatype tuple:

$$\begin{aligned} r &= (\mathit{active}, \mathit{score}, \mathit{created}, \mathit{tags}, \mathit{vector}), \\ \tau(r) &= (\texttt{gql:BOOLEAN}, \texttt{gql:INT8}, \texttt{gql:ZONED_DATETIME}, \\ &\qquad \texttt{gql:LIST}, \texttt{gql:VECTOR}). \end{aligned}$$

Listing 2 materializes this tuple as RDF predicate–object pairs whose literal datatypes are GQL datatype IRIs. Predicates identify the fields; triple order is immaterial. Recovering column order requires an external schema or explicit position metadata. JSON arrays retain their element order within each literal.

Listing 2: Typed GQL values represented as RDF literals.

```
@prefix ex:  <https://example.org/resource/> .
@prefix gql: <http://example.org/gql-types#> .

ex:row1
    ex:active  "true"^^gql:BOOLEAN ;
    ex:score   "42"^^gql:INT8 ;
    ex:created "2026-05-05T12:30:00Z"^^gql:ZONED_DATETIME ;
    ex:tags    "[\"alpha\", \"beta\"]"^^gql:LIST ;
    ex:vector  "[0.125, -2.5, 9.0]"^^gql:VECTOR .
```

This representation is useful when preserving the original GQL datatype is more important than immediately normalizing every value to a standard RDF datatype. It may be used by schema exchange tools, RDF exports of property graph data, and validation pipelines that need to retain the source datatype vocabulary. A separate normalization step may still map selected values to XSD or `rdf:JSON` targets for compatibility with RDF systems that do not recognize the GQL datatype IRIs directly.

5. Implementation and Evaluation

5.1. Implementation

The reference implementation is intended to demonstrate and test the proposed GQL datatype vocabulary rather than to introduce a separate data-generation framework. It consists of three main artifacts: the Turtle ontology `gql_datatypes.ttl`, the generator `generator_real.py`, and the validator `validator.py`, all available in the public repository.[1] The ontology declares 87 datatype IRIs, including the vector extension, in the namespace http://example.org/gql-types#. It records hierarchy, aliases, XSD bases, and selected lexical and value constraints with the RDF and OWL terms described above.

The generator creates synthetic patient records with `rdflib` and `Faker`; its workflow is shown in Figure 1. It writes Turtle triples for resources of type `schema:Patient`, assigns GQL datatype IRIs directly to literal values, and supports configurable field-presence probabilities. Faker supplies names

[1] https://github.com/szalajdaniel/Data_types_for_GQL_in_RDF_letters

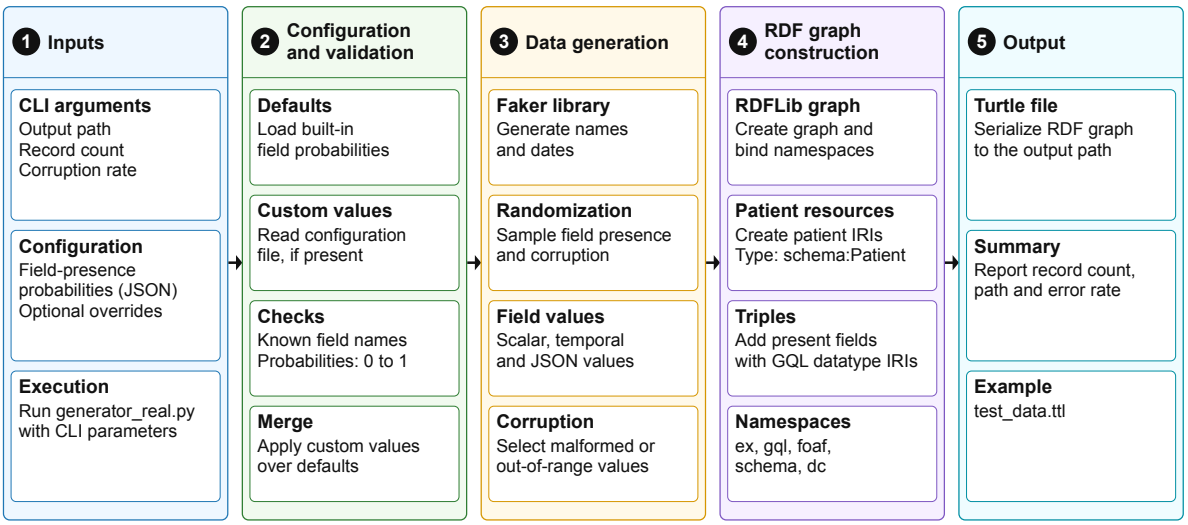


Figure 1: Synthetic RDF data generation workflow in the reference implementation.

and dates, without modeling clinical distributions or correlations. A command-line error-rate parameter selects corruption candidates, including out-of-range integers, malformed dates and booleans, wrong-length binary strings, JSON objects in place of arrays or vectors, and nonempty unit records. Some candidates are accepted by the validator, such as uppercase TRUE.

In Figure 1, (1) command-line parameters select the output, record count, error rate, and probability configuration. The generator then (2) merges field probabilities, (3) generates field values and corruption candidates, (4) adds them to the `rdflib` graph, and (5) serializes the graph as Turtle.

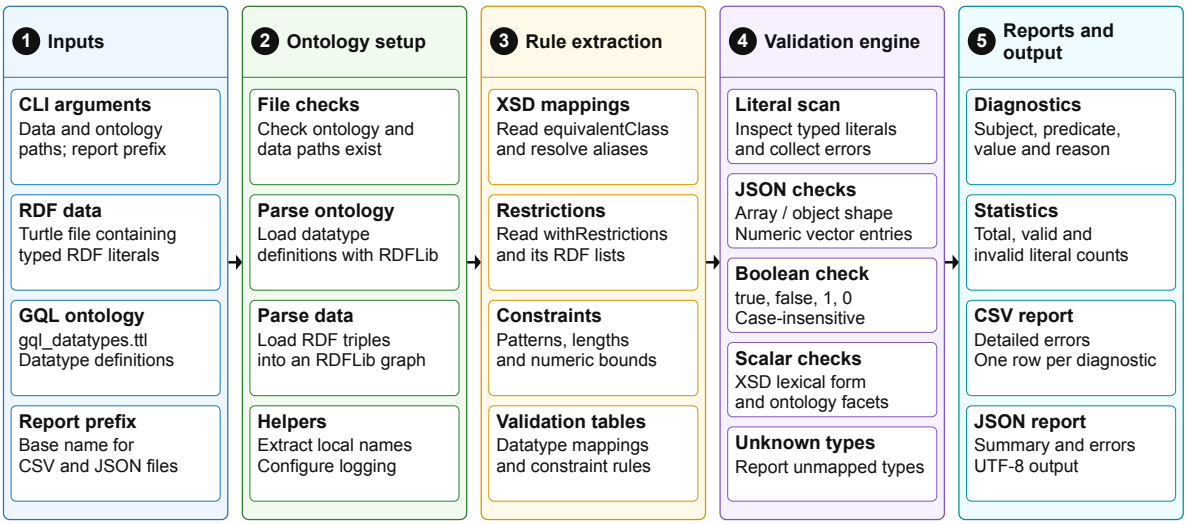


Figure 2: Validation workflow for GQL-typed RDF literals in the reference implementation.

The validator loads the ontology and data graph at runtime, as summarized in Figure 2. It derives XSD validation bases from `owl:equivalentClass`, follows aliases whose targets already have an XSD mapping, extracts patterns, bounds, and length constraints from `owl:withRestrictions`, and validates each typed literal. Validation errors are printed and exported as CSV and JSON reports containing the subject, predicate, value, original GQL datatype, base datatype, and error reason. Thus, the implementation should be read as an executable validation profile for the vocabulary, while the vocabulary itself remains the main contribution of the paper.

In Figure 2, ❶ the validator receives the data, ontology, and report paths; ❷ parses the ontology; ❸ extracts XSD mappings, aliases, and restrictions; and ❹ checks each typed literal. Finally, ❺ it prints diagnostics and writes CSV and JSON reports. Both tools are standalone Python command-line programs using RDFLib for graph parsing and serialization. Validation is application code, not a registered RDFLib datatype or SHACL extension. Neither workflow executes GQL queries.

5.2. Evaluation

We measured execution time and run-to-run variation using datasets of 10,000–50,000 patient records in steps of 5,000. Each size was tested at 10%, 45%, and 90% error-injection rates with ten repetitions, producing 270 runs and 8.1 million records. Field-presence probabilities determine the datatype mix: age and health-plan flag 0.90; birth date and gender 0.85; weight and identifier 0.80; height 0.75; admission datetime 0.70; prescription 0.65; arrival time 0.60; duration 0.50; salary 0.40; vector and unit record 0.30; unknown type 0.10. Names are mandatory. The test system had an Intel Core i7-1360P, 62 GiB of memory, Ubuntu 22.04, Python 3.10.12, RDFLib 7.6.0, and Faker 37.0.2. We measured generation and validation time, throughput, and valid and invalid typed literals [7].

Across all runs, the validator evaluated 84,234,831 typed literals, or approximately 10.40 literals per generated record. Table 3 summarizes the validation outcomes. The configured error rate is an independent corruption probability applied to eligible generated fields; it is not a target percentage of all RDF literals. The denominator also includes mandatory and non-corrupted literals, while the optional unknown-datatype field has its own presence probability. Consequently, the observed invalid literal rates are lower than the configured injection rates, but increase monotonically from 8.94% to 72.79%.

Table 3
Validation outcomes aggregated by configured error-injection rate.

Configured rate	Evaluated literals	Valid literals	Invalid literals	Invalid rate
10%	28,081,282	25,570,969	2,510,313	8.94%
45%	28,079,569	17,724,368	10,355,201	36.88%
90%	28,073,980	7,640,268	20,433,712	72.79%

Figure 3 summarizes generation time, while Figure 4 shows its throughput. Mean generation time increased from 8.58–8.68 s for 10,000 records to 44.33–44.74 s for 50,000 records. The differences between the mean and median were small for most configurations, and the min–max ranges were narrow apart from isolated runs. The linear models in Figure 3 explain more than 99.96% of the variation in the mean times ($R^2 = 0.99992$, 0.99992, and 0.99967 for the three error rates). Their slopes correspond to approximately 9.01, 8.91, and 8.98 s per additional 10,000 records. Thus, changing the error-injection rate has little effect on generation cost.

Generator throughput remains nearly constant across the tested workloads. The mean values over all dataset sizes were 1,135.1, 1,142.6, and 1,146.8 records/s for the 10%, 45%, and 90% configurations, respectively. The small differences between these values show that producing an invalid lexical candidate is not materially more expensive than producing a valid one. The modest decrease visible for some larger datasets does not alter the linear scaling trend.

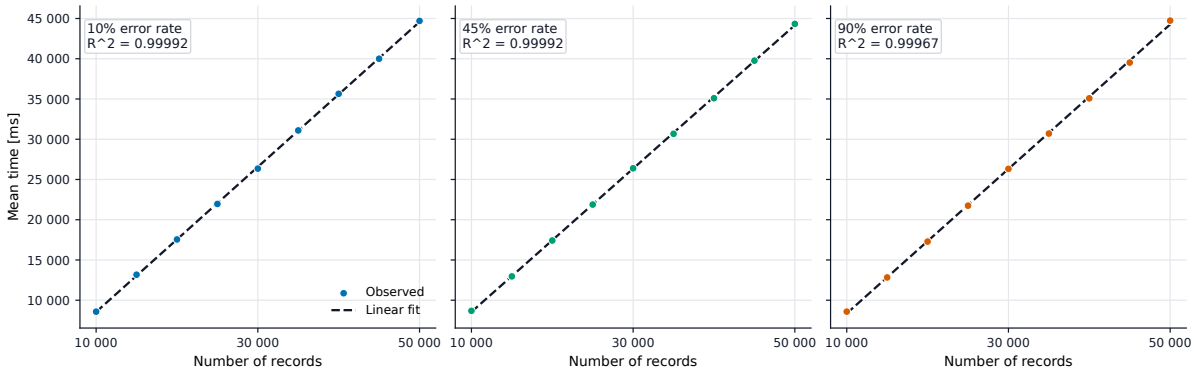


Figure 3: Mean generation time and linear fit as a function of the number of generated records.

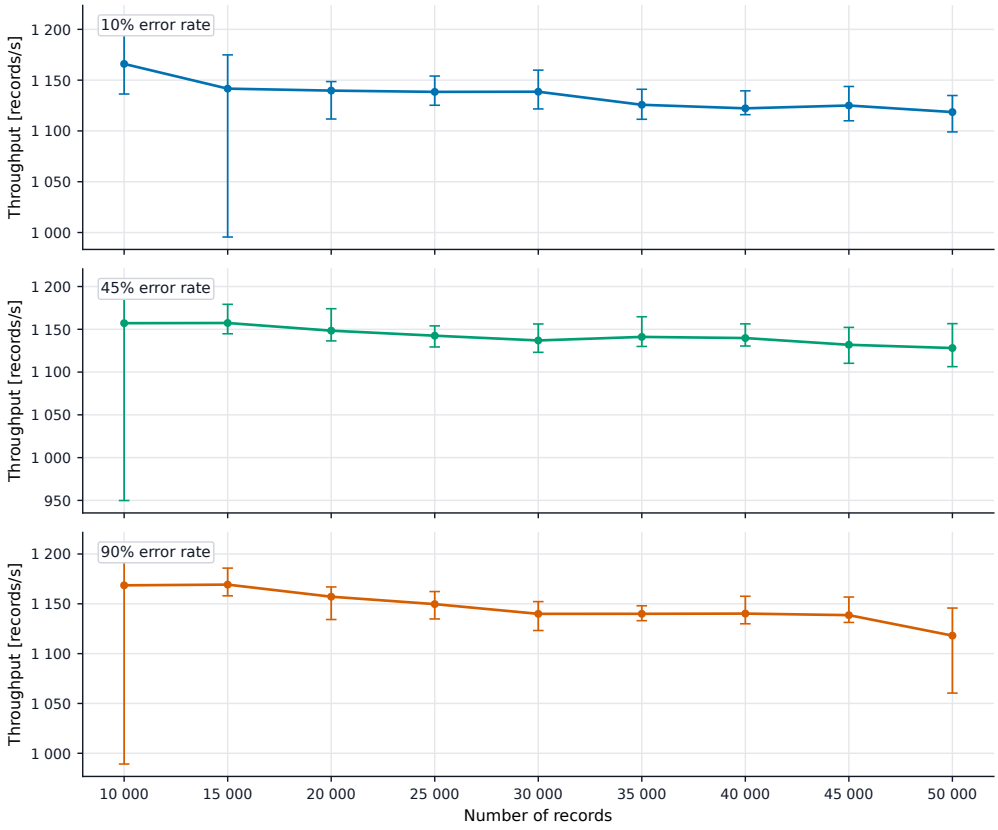


Figure 4: Generator throughput by dataset size and configured error rate. Error bars show the minimum–maximum range over ten runs.

Validation time is shown in Figures 5 and 6, and its throughput is shown in Figure 7. At 10,000 records, mean validation time was 7.46 s at the 10% error rate, 8.02 s at 45%, and 8.88 s at 90%. At 50,000 records, the corresponding times were 37.83, 41.69, and 45.91 s. The fitted models remain strongly linear, with $R^2 = 0.99991$, 0.99993, and 0.99986, while their slopes increase from 7.56 to 8.40 and 9.18 s per additional 10,000 records.

Unlike generation, validation throughput depends on the number of detected errors. Its mean value decreased from 1,333.3 records/s at the 10% configuration to 1,218.1 records/s at 45% and 1,105.2 records/s at 90%. For the largest dataset, the corresponding throughputs were 1,321.8, 1,199.4, and 1,089.2 records/s. This behavior is consistent with the additional work required to construct diagnostics

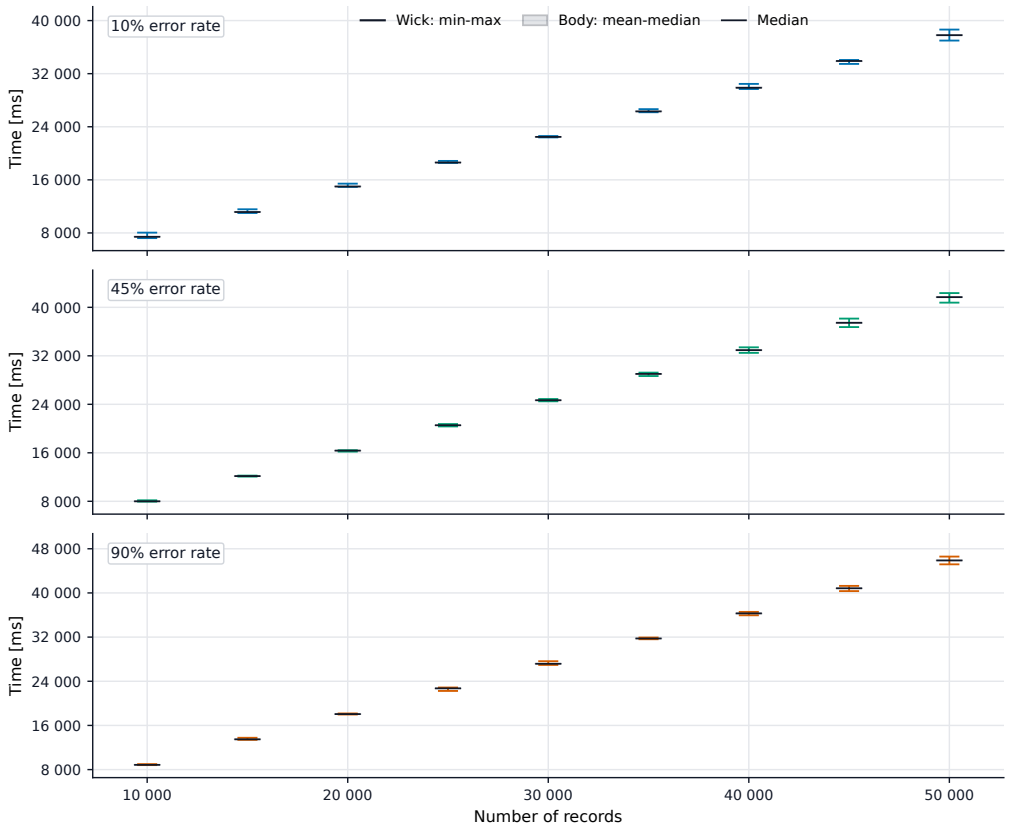


Figure 5: Distribution of validation times over ten runs for each dataset size and configured error rate. Whiskers show the minimum and maximum; the body spans the mean and median.

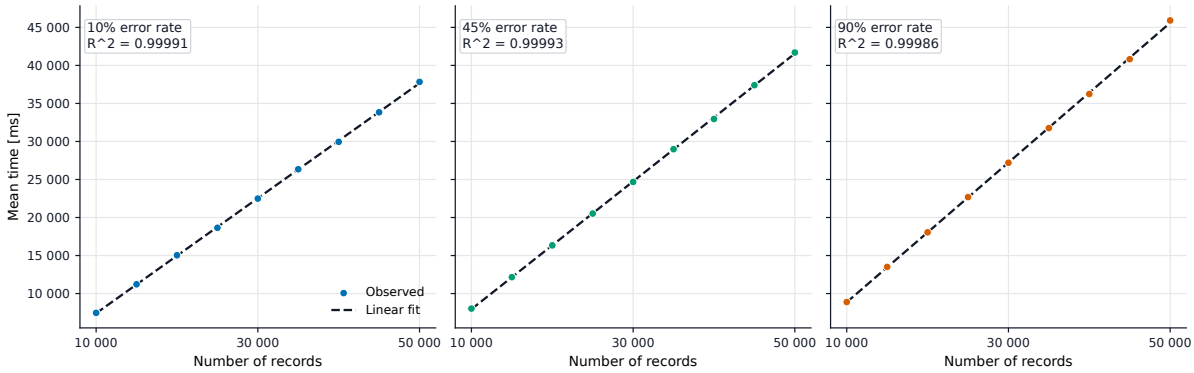


Figure 6: Mean validation time and linear fit as a function of the number of generated records.

and materialize error reports for invalid literals. Nevertheless, throughput within each error level remains comparatively stable as the dataset grows.

Overall, the experiment demonstrates approximately linear scaling for both components over the tested range. The generator is insensitive to the configured error rate, whereas validator cost increases predictably with the volume of reported errors. These measurements characterize the reference implementation on a synthetic workload and a single machine; they should not be interpreted as a benchmark of RDF stores or production property-graph systems. Generator measurements characterize workload preparation cost. The experiment does not measure datatype-specific validation cost, conformance accuracy, or end-to-end RDF/property-graph interoperability.

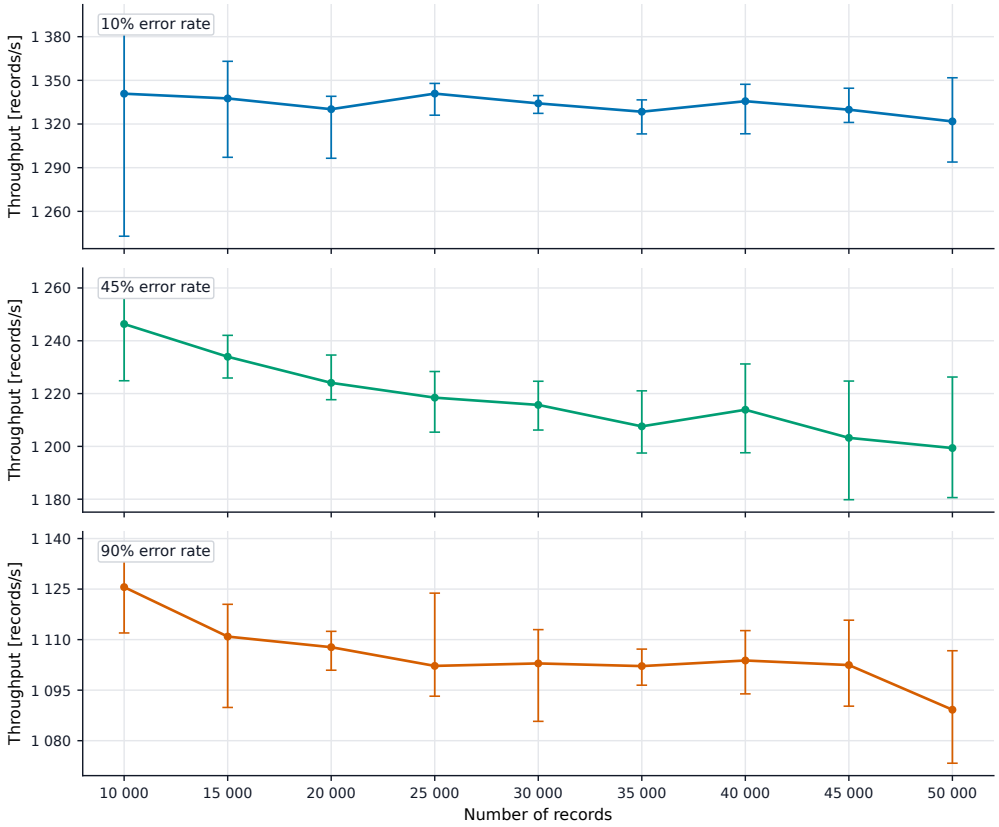


Figure 7: Validator throughput by dataset size and configured error rate. Error bars show the minimum–maximum range over ten runs.

6. Related Work

RDF collections and containers represent ordered values as graph structures, with different modeling patterns requiring different SPARQL queries. List.MID benchmarks six such patterns using MIDI event sequences and corresponding query templates [8]. In contrast, this work represents a GQL list, array, record, or vector as one typed literal, avoiding a list subgraph but not providing List.MID's element-level graph navigation.

Hartig et al. [3, 4] represent lists and maps as RDF literals and extend SPARQL with constructors, functions, aggregation, and an unfolding operator. Their approach provides generic composite values and query operations. This vocabulary instead covers the wider GQL datatype system, preserves specific GQL datatype IRIs, and provides validation rather than a SPARQL extension.

LINDT uses application-defined RDF datatypes to encode quantities and units from the Unified Code for Units of Measure (UCUM) in compact literals [9]. It defines lexical and value spaces and provides datatype-aware comparison, arithmetic, and casting in Apache Jena [9]. Both approaches use RDF datatype IRIs, but LINDT targets measurement computation, whereas this work models and validates the GQL datatype taxonomy for graph-data exchange.

RDF 1.2 defines `rdf:JSON` with JSON/I-JSON lexical forms and a value space containing strings, numbers, lists, maps, Booleans, and null [2]. It is suitable for generic JSON values. Here, JSON is only the lexical notation, while datatypes such as `gql:ARRAY`, `gql:VECTOR`, and `gql:UNIT_RECORD` preserve source-type information and enable checks not inferable from `rdf:JSON` alone.

7. Conclusion

The GQL Datatypes RDF Vocabulary represents selected GQL value datatypes in RDF. It identifies them by IRIs, records hierarchy and aliasing relations, records selected validator-oriented constraining facets, and supports validation of RDF literals that retain GQL datatype IRIs. By using ontology-derived XSD mappings for scalar datatypes and direct JSON lexical forms for constructed datatypes, the reference implementation preserves GQL datatype information without changing the RDF data model.

The vocabulary supports interoperability between GQL-oriented property graph systems and RDF-based knowledge graph systems. It can be used in data exchange pipelines, RDF exports, schema documentation, and validation scenarios. Since datatype IRIs are RDF resources, they can also be referenced from constraint languages such as SHACL to express GQL-related datatype assumptions. Checking the full GQL-specific constraints, however, requires either generated SHACL shapes, SHACL-SPARQL constraints, entailment-aware validation, or the reference validator described in this paper.

Future work includes three directions. First, the vocabulary should be accompanied by a conformance-oriented test suite that includes positive and negative lexical examples for every datatype. Second, the mapping should be evaluated in end-to-end RDF–property graph exchange scenarios, including round-tripping experiments and datatype preservation metrics. Third, the vocabulary could be integrated more closely with RDF constraint languages and query systems, for example, by generating SHACL shapes that check GQL-compatible lexical forms and by defining reusable datatype profiles for common implementation targets.

Declaration on Generative AI

During the preparation of this work, the authors used OpenAI GPT-5.6 for content drafting, simulated peer review, content refinement, and formatting assistance. All AI-assisted content was carefully reviewed, revised, and validated by the authors, who take full responsibility for the final content of the publication.

References

[1] ISO/IEC, ISO/IEC 39075:2024 Information technology – Database languages – GQL, 2024.

[2] G. Kellogg, O. Hartig, P.-A. Champin, A. Seaborne, RDF 1.2 Concepts and Abstract Data Model, W3C Candidate Recommendation, 2026. URL: https://www.w3.org/TR/rdf12-concepts/, 7 April 2026.

[3] O. Hartig, G. T. Williams, M. Schmidt, O. Lassila, C. M. L. Enriquez, B. Thompson, Datatypes for Lists and Maps in RDF Literals, in: The Semantic Web: ESWC 2024 Satellite Events, volume 15344 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 234–238. doi:10.1007/978-3-031-78952-6_33, presented at ESWC 2024.

[4] O. Hartig, G. T. Williams, M. Schmidt, O. Lassila, C. M. L. Enriquez, B. Thompson, SPARQL CDTs: Representing and Querying Lists and Maps as RDF Literals, Unofficial Draft, 2024. URL: https://awslabs.github.io/SPARQL-CDTs/spec/latest.html.

[5] D. Peterson, S. Gao, A. Malhotra, C. M. Sperberg-McQueen, H. S. Thompson, W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, W3C Recommendation, 2012. URL: https://www.w3.org/TR/xmlschema11-2/, 5 April 2012.

[6] T. Bray, The JavaScript Object Notation (JSON) Data Interchange Format, RFC 8259, 2017. URL: https://www.rfc-editor.org/rfc/rfc8259. doi:10.17487/RFC8259.

[7] D. Szałaj, Performance and Accuracy Validation of Synthetic Medical RDF Graphs with GQL Typing, 2026. URL: https://figshare.com/articles/dataset/Performance_and_Accuracy_Validation_of_Synthetic_Medical_RDF_Graphs_with_GQL_Typing/32874107. doi:10.6084/m9.figshare.32874107.

[8] A. Meroño-Peñuela, E. Daga, List.MID: A MIDI-based benchmark for evaluating RDF lists, in: The Semantic Web – ISWC 2019, volume 11779 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 246–260. doi:10.1007/978-3-030-30796-7_16.

[9] M. Lefrançois, A. Zimmermann, Custom Datatypes: Towards a Web of Linked Datatypes, Unofficial Draft, 2020. URL: https://ci.mines-stetienne.fr/lindt/v3/custom_datatypes, 17 July 2020.