ShEx2SPARQL: Translating Shape Expressions into SPARQL Queries

Christoph Göpfert[1][0000-0001-6659-8947], Sheeba Samuel[0000-0002-7981-8504] and Martin Gaedke[0000-0002-6729-2912]

[1] Technische Universität Chemnitz, 09111 Chemnitz, Germany
{christoph.goepfert,sheeba.samuel,martin.gaedke}
@informatik.tu-chemnitz.de

Abstract. The Shape Expressions (ShEx) Language provides a powerful tool for describing and validating structures in RDF knowledge graphs. While Shape Expressions are primarily used for validation, they also describe graph structures, enabling knowledge graph exploration. However, existing ShEx engines focus on validation rather than data exploration. In this paper, we introduce ShEx2SPARQL, an approach to systematically translate shape expressions into corresponding CONSTRUCT, SELECT, or ASK SPARQL queries. This enables knowledge graph exploration based on already available ShEx schemas. Our approach imposes certain restrictions, notably the exclusion of recursive shape references, as SPARQL lacks sufficient support for recursive expressions. To evaluate our approach, we selected 292 Wikidata Entity Schemas, translated them into corresponding SPARQL queries and executed them against the Wikidata SPARQL endpoint. The results confirm the feasibility of our approach, but also reveal performance issues when executing complex SPARQL queries resulting from complex shapes with a multitude of constraints.

Keywords: Shape Expressions, ShEx, SPARQL, Knowledge Graph, Exploration, Linked Data, Semantic Web.

1 Introduction

Knowledge graphs (KGs) have become a key technology for the structured organization and integration of information in a machine-readable format. KGs store knowledge as a collection of entities and their relationships using the Resource Description Format (RDF) [1] data model. RDF data can be queried and manipulated using standardized query language SPARQL [2]. However, constructing effective queries requires knowledge of the graph's structure and its underlying ontologies to construct correct query patterns. This structure is typically described in ontologies such as the Web Ontology Language (OWL) [3]. If information about the structure of a KG is not available, it can be derived either by analyzing the ontologies used or by using exploratory queries, before being able to construct the desired query.

The Shape Expressions Language (ShEx) [4] offers a schema language for formally describing and validating the structure of RDF data by defining "shapes". Shapes define

expected properties of RDF nodes, including required predicates in a triple, permitted datatypes for a specified literal, cardinality constraints, or logical combinations of graph patterns. Although designed as a validation language, ShEx shapes encapsulate rich structural information offering a blueprint for SPARQL query patterns.

We introduce ShEx2SPARQL, a novel approach leveraging ShEx schemas for KG exploration. ShEx provides rich, declarative descriptions of shapes which our approach repurposes to guide data discovery in KGs. ShEx2SPARQL translates Shape Expressions into CONSTRUCT, SELECT, or ASK SPARQL queries. Our approach enables the easy retrieval and inspection of data in a KG that conforms to a respective shape, eliminating the need for extensive, manual query construction. Furthermore, it can be applied for other use cases, such as identifying incomplete or inconsistent data, or revealing latent relationships among entities. However, not all graph structures describable with ShEx can be translated into SPARQL, leading to certain restrictions. We evaluated our approach using Wikidata's [5] knowledge base. Wikidata uses Entity Schemas expressed using ShEx to validate graph structures. We selected 292 of these, translated them into queries which executed to demonstrate the feasibility of our approach.

2 Related Work

A template-based query generation approach is presented by Cocco et al. [6], using a training set of natural language questions with associated SPARQL queries to answer natural language questions. A tagger is used to identify entities and relations in a question. Then, predefined query templates are selected based on similarity and presented iteratively to the user to select. Light-QAWizard [7] applies an RNN-based multi-label classification to map a natural language question to templates. These are then used for query generation, attempting to exclude irrelevant clauses to reduce query complexity. More recently, Taffa and Usbeck [8] presented a few-shot LLM-based approach using Vicuna-13B to retrieve similar question-query template pairs. However, queries are associated with question embeddings, which are also used for finding similar questions.

Tools such as Sparklis [9] and the Wikidata Query Builder[1] employ an interactive, visual approach to enable users to construct queries by providing a faceted view. Similarly, the tools Protégé [10] and QueryVOWL [11] enable users to visualize and edit ontologies. Protégé allows users to execute SPARQL queries in its user interface but does not provide pre-defined templates. QueryVOWL allows SPARQL query construction, but is limited to SELECT queries.

Linked Data Objects (LDO) [12] represents RDF data as JavaScript objects. Modifications to objects can automatically be translated into SPARQL update queries. SELECT, ASK or CONSTRUCT queries are not supported.

In contrast, ShEx2SPARQL translates formal constraints specified in a ShEx schema directly into executable SPARQL graph patterns. No intermediate steps – such as template completion or mapping of entities and relations – are required. Further, by reusing schemas, ShEx2SPARQL provides an alternative approach to knowledge graph

[1] https://query.wikidata.org/querybuilder/

exploration with SELECT, ASK or CONSTRUCT queries, without the need for training data or manual query construction.

3 Mapping ShEx to SPARQL

Shape expressions provide a formal schema language designed to validate RDF data by defining shapes, specifying expected node properties. These may include constraints on predicates, permitted datatypes, cardinality, and logical combinations of graph patterns. There are three types of ShEx syntaxes. In the following, we use the terminology of the JSON-LD-based ShExJ grammar in ShEx version 2. Its building blocks include **Schema**, **Shape Expression** (shapeExpr), **NodeConstraint** and **TripleConstraint**. Below, we provide a shortened (indicated by "(…)") excerpt of the ShExJ syntax [13]:

Schema { start:shapeExpr? shapes:[ShapeDecl+]? (...) }
ShapeDecl { id:shapeExprLabel shapeExpr:shapeExpr (...) }
shapeExpr = ShapeOr | ShapeAnd | ShapeNot | NodeConstraint | Shape | ShapeExprRef ;
ShapeOr { shapeExprs:[shapeExpr{2,}] }
ShapeAnd { shapeExprs:[shapeExpr{2,}] }
ShapeNot { shapeExpr:shapeExpr }
Shape { expression:tripleExpr? (...) }
NodeConstraint { id:shapeExprLabel? nodeKind:(...) datatype:IRIREF? (...) }
tripleExpr = EachOf | OneOf | TripleConstraint | tripleExprRef ;
EachOf { id:tripleExprLabel? expressions:[tripleExpr{2,}] (...) }
OneOf { id:tripleExprLabel? expressions:[tripleExpr{2,}] (...) }
TripleConstraint { id:tripleExprLabel? predicate:IRIREF valueExpr:shapeExpr? (...)}

Schema represents the main building block, optionally specifying a start shape and a collection of shape declarations. A shape declaration (*ShapeDecl*) consists of an identifier and a shape expression (*shapeExpr*), which in turn can take various forms: logical combinations of two or more shapes (*ShapeOr* and *ShapeAnd)*, the negation of a shape (*ShapeNot*), define structural constraints in triple expressions (*Shape)*, impose constraints on a subject or object of a triple (*NodeConstraint)*, and reference another shape expression in the schema (*ShapeExprRef)*. A shape element may contain a triple expression (*tripleExpr*), which can be logically combined (*EachOf* and *OneOf)*, impose triple constraints (*TripleConstraint)*, or reference a triple expression (*tripleExprRef)*. The following challenges need to be addressed when mapping ShEx to SPARQL:

Recursion: ShEx allows recursive references of shapes. This enables the definition of nested or hierarchical relationships at arbitrary depth. However, SPARQL does not provide the means to express constrained traversal. Consequently, recursive relationships cannot be expressed in SPARQL and must either be omitted or simplified.

Start Shape: In ShEx, the presence of a start shape is optional. Without a specified start shape, however, a definite entry point when translating the schema may not be determinable. Selecting a start shape at random may result in different queries depending on the selected shape, which we believe to be undesirable for exploration. Instead,

a “root shape” would need to be identified, i.e., a shape not referenced by any other shape. Further, a schema may contain isolated clusters of shapes. If a start shape is specified, this may not represent an issue, as other clusters will be disregarded in the generated queries. In the absence of a specified start shape, it would not be possible to identify a definitive root shape, as several options may be available, possibly resulting in differing queries. This can be solved by moving shape clusters into distinct schemas.

Variable Naming: A shape describes the structure of a resource. Consequently, a resource represents the subject of the graph patterns generated to express the constraints of a shape. However, shape ids may contain characters that are not valid in SPARQL variable names. To avoid this, shape ids should not be used in the constructed query. Shape ids may be hashed to create SPARQL syntax compliant variable names.

Cardinality Constraints: A triple expression may impose cardinality constraints, such as the minimum or maximum number of times a triple with a specified predicate may occur. Mapping such constraints requires a mechanism to count the occurrence of certain predicate-object pairs for a given subject. An approach to solving this is described by Prud’hommeaux[2], using a counter function generating a query block to count the matching predicate-object pairs. This block then aggregates using the function *COUNT()* combined with a *GROUP BY* clause. Cardinality constraints are then specified using the *HAVING* clause to enforce the count value to be within the specified minimum and maximum limits. Both GROUP BY and HAVING can be represented in the post-modifier of a block, explained in Section 4.

Node Constraints: A Node Constraint imposes restrictions on a node and may specify its datatype, node kind or permitted values. These constraints can be mapped to corresponding SPARQL FILTER operations.

Based on these background considerations and challenges in mapping ShEx schemas to SPARQL queries, we discuss the specifics of our approach Section 4.

4 The ShEx2SPARQL Approach

This section introduces ShEx2SPARQL and describes how it translates key ShEx schema elements into SPARQL query patterns to enable knowledge graph exploration. It supports SELECT, CONSTRUCT and ASK queries and imposes two restrictions on source schemas to ensure executable and valid queries: **1)** A start shape with a provided id must be defined in the source schema. As stated in Section 3, several scenarios exist that hinder effective exploration if no start shape is given. For this reason, we consider the definition of a start shape to be mandatory. **2)** schemas containing recursive shape references cannot be translated, as SPARQL lacks support for constrained traversal.

ShEx2SPARQL begins by parsing a given schema, identifying its start shape. The structure of the query is then constructed by traversing the elements of the schema and converting any constraints into a hierarchical representation of **blocks** and **statements.**

A **block** represents a grouping of statements and blocks nested within the current block. A block may include a “pre”- and “post”-modifier and a list of statements and

[2] https://www.w3.org/2013/ShEx/toSPARQL.html

sub-blocks. Pre- and post-modifiers specify optional or excluded graph patterns and group by variables with cardinality constraints. They also specify cardinality ranges. A **statement** corresponds to a triple pattern, filter, or VALUES condition. Sub-blocks handle nested shape expressions or patterns that are optional or need to be excluded.

After processing the schema, its root block representing the outermost structure of the query is returned. Then, each schema element is processed using a corresponding "visit" function to map its contents to SPARQL constructs. These are described below:

1) Schema

The start shape specified in the schema is identified and visited using the *visitShape* function, returning a new block. This block element represents the root block, which will also be returned by the *visitSchema* function.

2) Shape

As a shape represents an entity, its expressions describe the expected structure of this entity. The shape id is used in the SPARQL translation process as the subject for the triple patterns generated from its expressions. We use hashed (MD5) shape ids to ensure SPARQL compliant variable names. Each shape's shape expression is visited with the *visitShapeExpr* function. This function takes the hashed shape id as a parameter and recursively processes the shape's constraints. It returns a block with statements representing the structure of the shape as SPARQL graph patterns. The *visitShape* function returns this block without any modifications.

3) Shape Expression

Each type of shape expression needs to be processed differently:

Shape: A shape may contain a triple expression which is visited, returning a new block and statements. The new block is added as a sub-block to the current expression. New statements are then appended to the statements of the current triple expression.

ShapeOr: A *ShapeOr* represents two or more shapes linked by logical OR. This is translated by iterating through each shape, visiting their shape expression. Each visit results in a new block and statements. For each new block, the pre-modifier "UNION" is set to translate the OR relation of the shapes to blocks of triple patterns. For the first block, the "UNION" pre-modifier is not set, as only the blocks from the second onward must be joined via "UNION". New blocks and statements are handled as described for *Shape* elements. Statements consisting of SPARQL FILTERs require special handling. Multiple FILTER statements must be merged by logical OR into a single FILTER.

ShapeAnd: A *ShapeAnd* represents an AND relationship between two or more shapes. Similarly to *ShapeOr*, this relation is translated by iterating through each of the shapes, visiting each shape expression. New blocks and statements are processed as described for the *Shape* element. Unlike *ShapeOr*, FILTER constraints do not require any special handling, as an AND condition requires all FILTER constraints to apply.

ShapeNot: A *ShapeNot* represents a negated shape expression. The shape expression is visited, creating a new block and statements. These are processed as described for the *Shape* element. However, the pre-modifier of the block is set to "MINUS" to force the exclusion of the specified constraints in SPARQL.

NodeConstraint: A NodeConstraint specifies constraints for individual nodes. Node constraints are translated into statements as opposed to blocks. The created statements are appended directly to the list of statements of the current shape expression.

ShapeExternal: A *ShapeExternal* represents a reference to a shape in an external schema. The referenced shape must be retrieved from the imported schema and is then processed according to the procedure described in **2) Shape**. The resulting new block and statements are integrated as described for the *Shape* element.
shapeExprRef: *shapeExprRef* is a reference to another shape within the current schema. The referenced shape is processed in the same way as an external shape, with the only difference being that the shape does not need to be looked up first.

4) Triple Expression

Each type of triple expression is processed differently. For the type *TripleConstraint*, the triple constraint is visited using the function *visitTripleExpression*, returning a new block and statements. A *TripleExpression* of type *EachOf* or *OneOf* contains multiple linked expressions. In these cases, a new block with statements is created to encapsulate the entire triple expression before iterating through each sub-expression using *visitExpression*. In each visit, a new block and statements are returned. Each new block is added as sub-block to the block of the triple expression. Each *EachOf* triple expression needs to be linked by OR, the pre-modifiers of the corresponding blocks are set to "UNION", as described for the *ShapeOr* element. Similarly to *shapeExprRef*, *tripleExprRef* represents a reference to a triple expression within the schema. The referenced triple expression must be visited and subsequently processed as described in this section. Finally, the block and statements of the triple expression are returned.

5) Node Constraint

A *NodeConstraint* may contain constraints on a triple's subject or object. Object constraints also specify the triple's predicate in a *NodeConstraint*. A randomized hash is used to refer to the object, as a schema may contain multiple node constraints on triples with the same subject and predicate, but differing objects. A constraint will either specify a datatype, values, or node kind, all of which can be translated to FILTER operations as described in Section 3. The *numericFacets* can be translated with comparison operators, and by using the functions STRLEN() or SUBSTR(), etc. Finally, the *visitNodeConstraint* function returns the created statements.

6) Triple Constraint

Triple constraints define a triple's predicate and may include a value expression with a shape expression. The triple's subject is the variable of the shape associated with the triple constraint through a triple expression. The triple's object is described by an optional value expression. A new block and statements are created if either a value expression is specified, or the triple expression is optional or exclusive. ShEx2SPARQL creates a random variable for the constrained triple's object, unless the shape expression is referencing a shape in the schema, where the referenced shape id's hash is used instead. If a triple constraint does include a value expression, the generated variable is passed to the *visitShapeExpr* function as subject, as its shape descriptions relate to the constrained triple's object. The new blocks and statements returned are appended as sub-blocks to the priorly created block and triple constraint's list of statements.

Value expressions of type node constraint are visited, adding statements it returns to the list of statements. These must be moved as statements of the created block if the constraint is optional or excluding, as this block may contain an "OPTIONAL" or "MINUS" pre-modifier. Finally, the created block and statements are returned.

ShEx2SPARQL supports the query methods CONSTRUCT, SELECT and ASK. For CONSTRUCT queries, the graph patterns within the CONSTRUCT clause are populated with all unique statements, omitting FILTER and VALUES statements. Additionally, ShEx2SPARQL provides the option to specify the URI of the start shape, essentially querying a specific instance of the start shape. This is realized by replacing the variable name of the start shape by a provided URI. An ASK query can be constructed to verify whether a resource identified by a provided URI complies with the shape of the schema. Consequently, the provision of a URI is mandatory for ASK queries. For SELECT queries, no further modifications to the constructed query are necessary.

5 Evaluation

We created a proof-of-concept prototype of the ShEx2SPARQL approach, currently providing limited support for cardinality constraints. For parsing schemas, the tool relies on shexjs/parser[3]. Source code is available via **https://purl.org/shex2sparql/code**.

To assess feasibility, we obtained 422 Entity Schemas (ShEx schemas) from Wikidata. We removed schemas that were either 1) empty, 2) invalid or obviously incorrect, 3) made use of schema imports, 4) lacked a start shape, or 5) included recursive shape references. The last three criteria are based on the previously mentioned limitations. The selection based on the above criteria was automated with a script to ensure the reproducibility of our evaluation, available via **https://purl.org/shex2sparql/evaluation**. A total number of 292 schemas remained after filtering.

For each schema, we generated a CONSTRUCT query and executed it against Wikidata's SPARQL endpoint. 187 of 292 queries succeeded (HTTP 200), while 105 timed out. Re-executing them with "LIMIT 1" recovered 47 more results. We found no simple correlation between query complexity (number of triple patterns or FILTERs) and timeouts. Four highly complex queries (>164 graph patterns) all timed out, yet even some seemingly trivial queries (containing one graph pattern and a filter) did so too, suggesting endpoint load and queried data volume also play roles. We provide a digital appendix including a detailed list of executed queries and their results at **https://purl.org/shex2sparql/data**. Among the 234 successful queries, 56 returned no data (44 of the original 187, and 12 of the 47 limited). Investigation (cf. digital appendix) showed two causes: 1) some shapes simply had no matching instances in Wikidata, and 2) several schemas contained errors such as invalid IRIs, deprecated or mistyped properties, misused value sets. Thus, the queries were incapable of retrieving data.

6 Conclusion

We presented ShEx2SPARQL, a novel approach leveraging the Shape Expressions Language (ShEx) for knowledge graph exploration. ShEx2SPARQL utilizes a given ShEx schema to systematically translate it into a corresponding SPARQL query. This enables users to explore knowledge graphs by re-utilizing available ShEx schemas. We

[3] https://www.npmjs.com/package/@shexjs/parser

evaluated our approach using real-world ShEx schemas from Wikidata and demonstrated its feasibility by generating SPARQL queries that yielded data conforming to the shapes specified in the schemas. However, we also found limitations, as a subset of queries resulted in timeouts—an issue largely attributable to external endpoint performance and inherent query complexity. Future work will focus on extending the capabilities of ShEx2SPARQL by supporting schema imports and optimizing generated SPARQL queries to better handle complex constraints and to mitigate timeout issues.

Acknowledgements: This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 514664767 – TRR 386 and by the European Commission [grant 101120657 "European Lighthouse to Manifest Trustworthy and Green AI" – ENFIELD].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

[1] G. Klyne, J. J. Carroll, and B. McBride, *RDF 1.1 Concepts and Abstract Syntax*.
[2] S. Harris, A. Seaborne, and E. Prud'hommeaux, "SPARQL 1.1 Query Language."
[3] W3C OWL Working Group, "OWL 2 Web Ontology Language Document Overview (Second Edition)."
[4] E. Prud'hommeaux, I. Boneva, J. Emilio, and G. Kellog, "Shape Expressions Language 2.1."
[5] D. Vrandečić and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Commun ACM*, vol. 57, no. 10, pp. 78–85, Sep. 2014, doi: 10.1145/2629489.
[6] R. Cocco, M. Atzori, and C. Zaniolo, "Machine Learning of SPARQL Templates for Question Answering Over LinkedSpending," in *28th Int. Conf. on Enabling Technol.: Infrastruct. for Collab. Enterp. (WETICE)*, Jun. 2019, pp. 156–161.
[7] Y.-H. Chen, E. J.-L. Lu, and Y.-Y. Lin, "Efficient SPARQL Queries Generator for Question Answering Systems," *IEEE Access*, vol. 10, pp. 99850–99860, 2022.
[8] T. A. Taffa and R. Usbeck, "Leveraging LLMs in Scholarly Knowledge Graph Question Answering".
[9] S. Ferré, "Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language," *Semantic Web*, vol. 8, no. 3, pp. 405–418, Dec. 2016.
[10] M. A. Musen, "The Protégé Project: A Look Back and a Look Forward," *AI Matters*, vol. 1, no. 4, pp. 4–12, Jun. 2015, doi: 10.1145/2757001.2757003.
[11] F. Haag, S. Lohmann, S. Siek, and T. Ertl, "QueryVOWL: Visual Composition of SPARQL Queries," in *The Semantic Web: ESWC 2015 Satellite Events.*, Cham: Springer, May 2015, pp. 62–66.
[12] J. Morgan, "Linked Data Objects (LDO): A TypeScript-Enabled RDF Devtool," in *The Semantic Web – ISWC 2023*, Berlin: Springer, 2023, pp. 230–246.
[13] "Shape Expressions Language 2.next." Accessed: Feb. 19, 2025. [Online]. Available: https://shex.io/shex-next/