Let’s build Bridges not Walls: SPARQL Querying of TinkerPop Graph Databases with sparql-gremlin

Harsh Thakkar
OSTHUS GmbH, Germany
harsh.thakker@osthus.com

Renzo Angles
University of Talca, Chile
rangles@utalca.cl

Marko Rodriguez and Stephen Mallette
Apache TinkerPop, USA
{okram,spmallette}@apache.org

Jens Lehmann
University of Bonn, Germany
jens.lehmann@cs.uni-bonn.de

Abstract—This article presents sparql-gremlin, a compiler that translates SPARQL queries to Gremlin traversals. Currently, sparql-gremlin is a plugin of the Apache TinkerPop graph computing framework, thus the users can run queries expressed in the W3C SPARQL query language over a wide variety of graph data management systems, including both OLTP graph databases and OLAP graph processing frameworks. With sparql-gremlin, we take the first step towards bridging the query interoperability gap between the Semantic Web and Graph database communities. The plugin has received adoption from both academia and industry research in its short timespan.

I. INTRODUCTION

Knowledge graphs have become increasingly popular over the past years. The two most popular data models for representing knowledge graphs are Property Graphs (PG) and Linked Data graphs adhering to the W3C Resource Description Framework (RDF). Both approaches have distinct and complementary characteristics: RDF is suited for distributed data integration with built-in world-wide unique identifiers and vocabularies; PGs on the other hand support horizontally scalable storage and querying, and are widely used for modern data analytics applications.

The standard query language for RDF databases is SPARQL [1], whereas for PG databases there are several languages, including the popular Gremlin traversal language [2]. Currently, SPARQL and Gremlin lack interoperability, i.e. there is no standard and formal methods to translate queries between these languages [3]. The query interoperability problem mentioned above is addressed in this paper by presenting sparql-gremlin, a tool that allows the execution of SPARQL queries over graph databases by translating them to Gremlin traversals. We chose Gremlin as a favorable option to support query interoperability, due to its popularity in the graph database community. Gremlin allows querying using both declarative and imperative constructs, as well provides coverage to a plethora of popular graph databases (OLTP) and graph processing frameworks (OLAP), including AWS Neptune, Azure Cosmos DB, Neo4J, JanusGraph, OrientDB, Apache Hadoop & Spark, among others.

Related Work. To the best of our knowledge, there is no formally published work or openly available software that addresses the query interoperability issue between SPARQL and any Property graph query language on a broader scale [4]. On the other hand, Commercial graph databases, such as AWS Neptune, BlazeGraph, and Stardog are also not comparable to our approach as they provide support for querying using Gremlin traversals directly, without translating it from SPARQL. A similar argument holds for other TinkerPop-enabled databases. Therefore, the proposed sparql-gremlin approach is the first effort that has been implemented [5] and successfully integrated in the industry as a plugin of the Apache TinkerPop open-source project.

A research topic related to our work is the query interoperability between SPARQL and SQL, which was investigated in e.g. [6], [7], [8], [9]. However, we do not elaborate on these as they are orthogonal to the current context of our work. In contrast to SPARQL-SQL translation, we have to overcome the challenge of mediating between two very different execution paradigms. More specifically, those efforts applied query rewriting techniques between languages, which are rooted in relational algebra operations, whereas we had to bridge more disparate query paradigms. While this poses a significant challenge, it is also the reason why substantial performance differences can be observed depending on the use case and query characteristics (feature composition).

Contributions. Overall, we make the following contributions:
- We present a novel method to execute SPARQL queries over Property graph databases by translating them to Gremlin pattern matching traversals.

Note: The presented sparql-gremlin resource in this paper is the final and only maintained version of the early proof-of-concept by Daniel Kuppitz – (https://github.com/dkuppitz/sparql-gremlin). To avoid confusion, this has been clearly stated by Kuppitz in the documentation of the repository.
• A mature implementation of this approach – *sparql-gremlin* which is openly available as a plugin of the popular Apache TinkerPop graph computing framework (version 3.4.0-onwards).
• We also deliver *sparql-gremlin* as an open and independent implementation of the transformation method. It can be used for native integration within custom use cases.

The remainder of the article is organized as follows: Section (II) describes the query transformation method used by *sparql-gremlin* and its implementation; Section (III) presents the evaluation methodology, results and the current limitations of *sparql-gremlin*; Section (IV) discusses the impact of *sparql-gremlin* and presents a few use cases that reuse it; Section (V) presents information on the reusability, technical quality and design, and the availability of our resource; Finally, Section (VI) concludes the paper and discusses future work.

II. SPARQL TO GREMLIN TRANSLATION

This section describes the data models (RDF and property graphs), the query languages (SPARQL and Gremlin) and outlines the transformation method pursued by *sparql-gremlin*. Due to space restrictions, we do not present detailed description. However, we present the minimum concepts to make the paper self-contained.

A. RDF and SPARQL.

*RDF* [10], acronym of Resource Description Framework, is a W3C standard that defines a graph data model for describing resources in the Semantic Web.

An RDF graph is a set of RDF triples \((s,p,o)\) where \(s\) is called the *subject*, \(p\) is the *predicate* and \(o\) is the *object*, each of which can be an IRI, subjects and objects can alternatively be blank nodes and objects can also represent literal data values.

Figure 1 shows an RDF graph describing information about people related to the TinkerPop Project. Each node in the graph is labeled with either an IRI (e.g. \(\text{http://tinkerpop.apache.org/people/2}\)), a blank node (e.g. \(\_x1\)) or a literal ("marko"), and each edge is labeled with an IRI (e.g. \(\text{a:Created}\)). It is important to note that the IRIs are presented using their simplified representation, i.e. \(p:n\), where \(p\) is the prefix and \(n\) is the name of the resource. For instance, the extended version of \(b:2\) can thus be expanded to the IRI \(\text{http://tinkerpop.apache.org/people/2}\). We will use abbreviated IRIs in order to make the examples legible.

*SPARQL* is the standard query language for RDF recommended by the W3C consortium. SPARQL is a declarative query language which is based on graph pattern matching. SPARQL 1.0 [1] defines basic types of graph patterns, filter conditions (e.g. equalities), solution modifiers (e.g. order by) and query forms (e.g. select). SPARQL 1.1 [11] extends the first version with operators for aggregation, subqueries and path queries.

In general terms, a SPARQL query is a collection of triple patterns grouped according to different clauses and operators. The result of evaluating a triple pattern is a set of bindings (i.e. a variable \(\rightarrow\) value assignment), which is called a binding table. The evaluation of a SPARQL query is defined by operations over binding tables.

For example, consider the SPARQL query shown in Listing 1. The expression \(\{ ?p1 a : name ?n1 \}\) is a triple pattern and \(?n1 = \text{"marko"}\) is a filter constraint. The operators *SELECT*, *FILTER*, *AND* and *OPTIONAL* allow to execute the operations of projection, selection, join and left-outer join over binding tables. The result of evaluating the sample query over the RDF graph shown in Figure 1 is a binding table with two solutions: \(\{ ?n2 \rightarrow \text{"josh"}, ?cn \rightarrow \text{"ripple"} \}\) and \(\{ ?n2 \rightarrow \text{"vadas"} \}\). Note that the query looks for people (?p2) that "marko" (?p1) knows, and returns the name (?n2) of such people, and the name (?cn) of the resources (?c) created by such people.

```sql
SELECT ?n2, ?cn WHERE {
    \{ ?p1 a :name ?n1 \} FILTER (?n1 = "marko") AND
} OPTIONAL \{ ?p2 a :Created ?c \} AND \{ ?c a :name ?cn \}
```

Listing 1. Example of SPARQL query

B. Property Graphs and Gremlin

A *Property Graph* is a directed, labeled, multigrapth, whose main characteristic is that nodes (or vertices) and edges can contain a (possibly empty) set of key-value pairs. Figure 2 shows a property graph that describes the same information...
A path traversal (denoted by $\Psi$) is composed of an ordered list of steps called the single-step traversals. A single-step traversal (SST, denoted by $\psi$) is an atomic operation over the elements in the target graph (i.e., nodes and edges). In the above example, the underlying SSTs are $\text{.has}(...)$, $\text{.out}(...)$ and $\text{.as}(...)$.

The expression $\text{g.V()}$ returns the set of all nodes (or vertices) in the graph and defines the starting point of the traversal. The $\text{as()}$ operator allows to define variables that can be used in any part of the Gremlin expression; in this case it is used to denote the start ($'x'$) and the end ($'y'$) of the traversal. The $\text{has}$ operator allows to filter vertices and edges based on their properties; in this case, the nodes whose property $'name$' has the value $'marko'$.

Gremlin includes a large list of traversal operators whose syntax and use is described in the TinkerPop3 documentation\(^\text{7}\). Next, we describe the operators that have been used to implement the translation from SPARQL to Gremlin:

\begin{itemize}
  \item **match** allows expressing pattern matching in a declarative form. It contains a collection of traversal patterns that must hold true.
  \item **union** allows the merging of the results of an arbitrary number of traversals.
  \item **optional** allows to return the result of the specified traversal if it yields a result, else it returns the calling element.
  \item **where** allows filtering the current object based on either the object itself or the path history of the object. It can include operators like eq and neq to evaluate equalities or inequalities. The operators and, or and not can be used to introduce boolean conditions.
  \item **group.by** allows organizing (or group) the objects according to some function of the object (e.g., a property).
  \item **order.by** allows sorting the objects.
  \item **range(begin,end)** allows restricting the number of objects obtained by a traversal.
  \item **limit** is analogous to **range()**, save that the lower end range is set to 0.
  \item **select** allows specifying the object returned by the traversal.
  \item **dedup** allows removing duplicated objects for the traversal stream.
\end{itemize}

\section*{C. From SPARQL Queries to Gremlin Traversals}

We have seen that SPARQL and Gremlin are two ways to express pattern matching over graphs. SPARQL follows a full declarative approach, whereas Gremlin uses path traversals. Now, we describe the query transformation used by sparql-gremlin.

Consider the function $\gamma(P)$ which takes a SPARQL graph pattern $P$ as input and returns a Gremlin expression. In order to show the idea of the transformation function, we will use the example shown in Figure 3.

**Triple patterns.** Given a triple pattern $(v_1, v_2, v_3)$, the transformation function generates a different Gremlin expression depending if $v_2$ refers to a property, or it refers to a relationship. In both cases the result is a simple traversal expression. In our sample transformation, the triple pattern $\text{?person} \text{v:label } "person"$ is translated to the Gremlin expression $\text{as('person').hasLabel('person')}$.

**AND graph patterns.** A graph pattern $\{P_1 \land P_2\}$ implies a natural join between the binding tables obtained from $P_1$ and $P_2$. This behavior is simulated in Gremlin by using the operator match, as it allows the join of a set of traversals. It is important to mention that a match can occur inside another match, in any level of nesting, so recursive matching is supported.

**FILTER graph patterns.** The FILTER operator is used to restrict the results obtained after evaluating a graph pattern. Several types of filter conditions are supported, including equalities, inequalities and boolean conditions (in our example, $\text{FILTER } (?\text{age} < 30)$). Filter conditions are expressed in Gremlin using the operator $\text{.where(C), where } C$ is a
constraint. Gremlin provides several operators to implement simple, complex filter conditions.

SELECT. This clause allows projecting the variables in the binding table obtained by the graph pattern matching step. This feature is implemented in Gremlin by using the .select() operator (in our example, SELECT ?age).

Aggregates. Aggregates apply expressions over groups of solutions that enable a user to group the solutions in specific groups as specified by the calculated aggregate values for a solution. Gremlin also provides several types aggregates like in SPARQL, (in our example, GROUP BY(?age)).

Solution modifiers. The solution set returned by the evaluation of a graph pattern is not de-duplicated or ordered by default, as both languages operate on bag semantics. Therefore, solution modifiers are used to sort, filter or limit objects in the solution (in our example LIMIT 2 maps to range(0,2)). Each SPARQL query modifier considered in this paper has a corresponding operator in Gremlin.

Table I presents an itemization of the equivalences between SPARQL operators and Gremlin expressions. We point the interested reader to [14], [15], [16], where we discuss the translation process using denotational semantics.

D. Implementation

We now discuss the implementation of the sparql-gremlin resource.

Encoding SPARQL Prefixes. We encode the prefixes of SPARQL queries within sparql-gremlin implementation, in order to aid the translation process. We define the custom namespace–http://tinkerpop.apache.org/traversal/$element$ for the graph elements (i.e. vertex, edge, and properties). For instance, an edge will be represented using the URI–http://tinkerpop.apache.org/traversal/edge. We also define custom prefixes for the IRIs keeping mind the corresponding Gremlin SSTs. For instance, the label prefix (which is a predicate in a SPARQL query - “rdfs:label”) is encoded as e:label or v:label (where e = edge and v = vertex). Similarly, a property-access operation on a vertex or an edge is encoded as v:property_name and e:property_name respectively.

Query Translation Pipeline. We discuss the query translation pipeline employed by the proposed sparql-gremlin resource. A SPARQL query passes through a series of steps as shown in Figure 4, which comprise of the translation pipeline, to obtain the resultant Gremlin traversal.

Step 1 The input SPARQL query is first parsed using the Jena SPARQL processing module (ARQ). This allows: (i) validating the query, i.e. checking whether the input query is a valid SPARQL query, and (ii) generating an abstract syntax tree (AST) representation.

Step 2 After the AST of the parsed SPARQL query is obtained, the opWalker then visits each triple pattern of the SPARQL query and maps or re-writes them to the corresponding Gremlin SSTs, i.e. via the Rewriter module (cf. Figure 4).

Step 3 Thereafter, depending on the operator precedence obtained from the AST of the parsed SPARQL query, each of the corresponding SPARQL operators are mapped to their corresponding instruction steps from the Gremlin instruction library. A final conjunctive traversal ($\Psi$) is then generated by the Translation Writer module, by appending the SSTs and instruction steps.

Step 4 Finally, this final conjunctive traversal ($\Psi$) is used to generate Gremlin Bytecode$^8$ which can be executed on any TinkerPop-enabled graph database.

$^8$Bytecode is a list of primitive-valued, nested arrays of the form: bytecode = [op, arg,*] where an arg can be another chunk of bytecode.
The following questions:

A. Evaluation Methodology

We empirically evaluate sparql-gremlin by answering the following questions:

Q1) Query preservation: Do the sparql-gremlin generated Gremlin traversals return the same results as their SPARQL counterparts? i.e. is the proposed approach preserving the meaning of the input queries?

Q2) Performance analysis: What observations and insights can we obtain upon executing the SPARQL queries and their Gremlin counterparts over three top-of-the-line RDF and Graph databases respectively?

B. Experimental Setup

We describe the setup implemented to conduct experiments next.

Datasets. We used the (i) Northwind dataset, which consists of synthetic data describing an e-commerce scenario between a fictional company “Northwind Traders”, its customers and suppliers, and the, (ii) Berlin SPARQL Benchmark (BSBM) dataset, which consists of synthetic data describing an e-commerce scenario involving a set of products, their vendors, and consumers who review the products. We generated one million triples for the experiment. The respective PG datasets were created using a trivial RDF graph pattern to PG pattern mapping rules. The python scripts created one million triples for the experiment. The respective PG vendors, and consumers who review the products. We generated one million triples for the experiment. The respective PG datasets were created using a trivial RDF graph pattern to PG pattern mapping rules. The python scripts

were performed on the following machine configuration: CPU: Intel® Xeon® CPU E5-2660 v3 (2.60GHz), RAM: 128 GB DDR3, HDD: 512 GB SSD, OS: Linux 4.2-generic. To ensure the reproducibility of our results, we provide the scripts, data and queries here[12], and also provide a persistent URL[12] referring to all the resources used in this paper. All queries were executed in both cold and warm cache settings.

C. Results and Discussion

We present the results with respect to the evaluation methodology described earlier. We executed the SPARQL queries against the three RDF triplestores on both the datasets and retrieved their results. Similarly, we executed the translated Gremlin traversals against the three graph databases on both datasets and retrieved their results. We compared the results returned by the SPARQL queries to those of their Gremlin counterparts. The query execution time (in ms) is reported for

Table I

<table>
<thead>
<tr>
<th>Operation</th>
<th>SPARQL k/w</th>
<th>Gremlin k/w</th>
<th>SPARQL construct</th>
<th>Gremlin construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching</td>
<td>WHERE { ... }</td>
<td>MatchStep(AND,[[],[]])</td>
<td>WHERE { BGp1, BGp2, ..., BGp_n }</td>
<td>[MatchStep(AND,[[ψ1],[ψ2], ..., [ψ_n]])]</td>
</tr>
<tr>
<td>Restriction</td>
<td>FILTER(C)</td>
<td>IsStep(C)</td>
<td>FILTER (?v1 &lt; 30)</td>
<td>IsStep(∧(30))</td>
</tr>
<tr>
<td>Join</td>
<td>JOIN</td>
<td>AndStep()</td>
<td>BGp1 * BGp2 * ... * BGp_n</td>
<td>AndStep([ψ1], [ψ2], ..., [ψ_v])</td>
</tr>
<tr>
<td>Projection</td>
<td>SELECT</td>
<td>SelectStep()</td>
<td>SELECT ?v1 = ?v2 ... ?v_n</td>
<td>SelectStep[a, b, ..., n]</td>
</tr>
<tr>
<td>Combination</td>
<td>UNION</td>
<td>UnionStep()</td>
<td>{BGp1} UNION {BGp2}</td>
<td>UnionStep(ψ1, ψ2)</td>
</tr>
<tr>
<td>Left Join</td>
<td>OPTIONAL</td>
<td>CoalesceStep()</td>
<td>{BGp1} OPTIONAL { BGp2}</td>
<td>ψ1,CoalesceStep(ψ2)</td>
</tr>
<tr>
<td>Deduplication</td>
<td>DISTINCT</td>
<td>DedupStep()</td>
<td>DISTINCT ?v1</td>
<td>DedupStep(ψ1)</td>
</tr>
<tr>
<td>Restriction</td>
<td>LIMIT(M)</td>
<td>RangeStep(0,M)</td>
<td>LIMIT 2</td>
<td>RangeStep(0,2)</td>
</tr>
<tr>
<td>Restriction</td>
<td>OFFSET(N)</td>
<td>RangeStep(N,M+N)</td>
<td>OFFSET 10</td>
<td>RangeStep(10,12)</td>
</tr>
<tr>
<td>Sorting</td>
<td>ORDER BY()</td>
<td>OrderStep()</td>
<td>ORDER BY DESC(?a)</td>
<td>OrderStep([value(a), desc])</td>
</tr>
<tr>
<td>Grouping</td>
<td>GROUP BY()</td>
<td>GroupStep()</td>
<td>GROUP BY(?a)</td>
<td>GroupStep(value(a))</td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>Query Feature</th>
<th>FILTER</th>
<th>COUNT</th>
<th>LIMIT</th>
<th>DISTINCT</th>
<th># TPs.</th>
<th># Proj. vars.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>F3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>L2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>L3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>G1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>G2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>G3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G4</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>G5</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>G6</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>O1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>O2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>O3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>U1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>U2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>U3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Op1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Op2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Op3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>M1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>M3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

TOTAL 30 Q.

an average of 10 runs per query (both SPARQL and translated
Gremlin traversals).

Due to lack of space, we report all the queries, their translations, results, and plots at the following online Google spreadsheet: (https://docs.google.
.com/spreadsheets/d/183aOSeCNR6y7GVv8NVOI16_TELEsIoZ4dR9HKSZVw63jw). Here we only present a summary of the observations from the conducted experimental
evaluation. The average time for translating a SPARQL query to the corresponding Gremlin traversal is 14 ms for BSBM and 12.5 ms for Northwind queries respectively.

Q1 - Query preservation: We observe that the results returned by the SPARQL queries and their corresponding Gremlin traversals were same/equal (i.e. they have the same number of results and same values for each corresponding variable in the result set) with the exception of their representation
formats. The SPARQL queries returned the results in a tabular format, whereas the Gremlin traversals returned results in a list (or a set of lists) format, e.g. consider the following:

<table>
<thead>
<tr>
<th>Q.</th>
<th>SPARQL Query</th>
<th>SPARQL Result</th>
<th>Gremlin Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>SELECT (COUNT (DISTINCT (?product) as &quot;total&quot;) WHERE { ?a v:type &quot;review&quot; . ?a e:edge &quot;product&quot; .})</td>
<td>2787</td>
<td>2787</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{pid=1636}</td>
<td>{pid=2295}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bsbm:inst/Reviewer1263</td>
<td>bsbm:inst/Reviewer1263</td>
</tr>
</tbody>
</table>

Thus, based on this empirical evidence we can say that the proposed translation approach is query preserving.

Q2 - Performance analysis: Due to a lack of space, we only report the results of the BSBM dataset. The query execution time in both cold and warm cache settings is shown in Figure 5. It can be observed the Gremlin traversals are competitive in most cases compared to their SPARQL counterparts. Similar performance was also reported in an independent study, which uses our approach, by [20] for querying Openlink Virtuoso vs JanusGraph. However, in queries with star shaped execution plan and neighborhood queries (path queries), Gremlin traversals outperform SPARQL queries by an order of two magnitudes. We suspect this is observed because the RDF triplestores spend a large amount of time in executing joins and forming an execution plan. Whereas, the graph databases take advantage of the micro-indices and graph neighborhood. SPARQL queries with mostly only basic graph patterns and with features such as union report 1.1x to 1.4x faster execution time on average as compared to their Gremlin counterparts. The full results are available at the document pointed out earlier. Finally, this also demonstrates that the proposed sparql-gremlin plugin is successful in translating and executing SPARQL queries (that cover SPARQL 1.0 specification) over the selected TinkerPop-enabled graph databases.

D. SPARQL Coverage and Limitations

The current version of the sparql-gremlin plugin supports the translation of SPARQL SELECT queries. It covers the SPARQL 1.0 specification, including aggregates from SPARQL 1.1 (cf. Table I), with the following exceptions:

L1) SPARQL queries which cannot be parsed by Apache Jena cannot be translated, i.e., if a SPARQL query cannot be parsed in Jena it will not yield a Gremlin traversal (e.g. queries with Non-group key variable in SELECT, etc).

L2) SPARQL queries with regular expressions (regex) are currently not supported. This will be resolved in TinkerPop4 as it will provide native support for regex in Gremlin traversals.

IV. IMPACT

In this section we report the general impact and community-wide adoption that the sparql-gremlin resource has amassed so far.

A. General Impact

In a general sense, the sparql-gremlin resource renders several benefits:

• It enables the users familiar with W3C SPARQL to query a variety of TinkerPop-enabled graph databases, avoiding the need to learn a new graph query language;
• Applications based on Semantic Web standards, like SPARQL and RDF, can use Property graph databases in a non-intrusive fashion;
• The query translation lays the foundation for hybrid use of RDF triple stores and Property graph databases (e.g. as a layer on top of AWS Neptune) wherein a particular query can be dispatched to the database capable to answer the query more efficiently [21]. In particular, property graph databases have been shown to work very well for a wide range of queries, which benefit from the locality in a graph [22], [23]. Rather than performing expensive joins, property graph databases use micro indices to perform traversals;
• It facilitates efforts for bridging the data and query interoperability gap between the Semantic Web and Graph database communities [4].

B. Community Adoption

The sparql-gremlin resource is gaining attention and adoption by both the academic and industry communities. We report a few such use cases:

IBM Research AI use case. In the recent research study [20] published by the IBM Research AI team, sparql-gremlin has been extended and reused in order to support scalable reasoning over large scale Knowledge Graphs.
The translation is embedded in the query layer to execute SPARQL queries over the property graph data stored in JanusGraph. They report better performance of the translated Gremlin traversals as compared to SPARQL counterparts in most cases in the case of Openlink Virtuoso vs JanusGraph.

**Contextualised Knowledge Graphs** This use case [24], in collaboration with the National Library of Medicine - National Institutes of Health, is about adding a semantic web abstraction layer on top of Graph databases by employing sparql-gremlin for querying a Contextualised Knowledge Graph (CKG) model. This project aims to simulate PG-style characteristics (e.g. node and edge properties) to RDF KGs via extending the singleton property semantics [25].

**SANSA Stack use case.** The Scalable Semantic Analytics (SANSA) Stack [26] exercises distributed computing via Apache Spark and Flink in order to enable scalable machine learning, inference and querying capabilities for large knowledge graphs. The proposed sparql-gremlin translation is employed in the query layer of the SANSA version 0.3\(^{13}\) as an experimental feature. The sparql-gremlin translation executes SPARQL queries in a distributed manner over the Apache Spark and Flink via Gremlin traversals.

\(^{13}\)release (https://github.com/SANSA-Stack/SANSA-Stack/releases/tag/2017-12), changelog (http://sansa-stack.net/sansa-0-3/)

**Open Research Knowledge Graph use case.** In the project, ScienceGRAPH funded by the European Research Council (ERC) an Open Research Knowledge Graph [27] is developed based on an integration of sparql-gremlin translation in order to execute SPARQL queries over a large scholarly communication Knowledge Graph.

This demonstrates that there is a visible engagement of both the research and industry communities. We can expect further adoption of our sparql-gremlin resource over the coming months, due to the popularity of the TinkerPop framework and the involvement of key players such as IBM Research.

V. REUSABILITY, DESIGN AND AVAILABILITY

A. Reusability

To promote reusability of sparql-gremlin, we provide an illustrative documentation in the following manner:

- Apache TinkerPop reference documentation\(^{14}\) – explains the working of the sparql-gremlin plugin and other technical details about it’s installation, use, etc. in the TinkerPop framework;
- Independent implementation documentation\(^{15}\) – which

\(^{14}\)http://tinkerpop.apache.org/docs/current/reference/#sparql-gremlin
\(^{15}\)https://github.com/LITMUS-Benchmark-Suite/sparql-to-gremlin/blob/master/README.md
is the independent source code of the proposed sparql-gremlin translation, which enables easy adoption and extension of our work, for custom use-cases. For instance, the re-use of our work by IBM Research AI [20] (cf. Section IV-B).

B. Technical Quality and Design

Since sparql-gremlin plugin is a part of the Apache TinkerPop project, community software development best practices were followed such as: (i) Apache Maven was used as the project management framework. (ii) Extensive Unit Tests covering a wide variety of test cases were implemented; (iii) Travis CI API\(^\text{16}\) was deployed for continuous automated integration, and (iv) All reference documentation was created using Javadoc

C. Availability and Maintenance

All the artifacts used in this study are permanently made available from https://doi.org/10.6084/m9.figshare.8187110.\(^\text{17}\) In collaboration with the Apache TinkerPop’s large community of contributors\(^\text{17}\), we will continue working on future releases. The Gremlin-users google group\(^\text{18}\) is an active public mailing list for the reporting questions and receiving support for the proposed TinkerPop sparql-gremlin plugin. Furthermore, source code related issues can be raised at the respective Github repository and Apache JIRA\(^\text{19}\).

VI. FINAL REMARKS AND FUTURE WORK

In this article, we presented the sparql-gremlin resource, a plugin of the Apache TinkerPop framework, which allows executing SPARQL queries over property graphs using Gremlin pattern matching traversals. The sparql-gremlin resource is also freely available for reuse and extension for custom use cases. With sparql-gremlin, we aim to take the first steps for supporting query interoperability between the two popular Semantic Web and Graph database communities. Our resource is gained attention from both academia and industry research fraternities so far, and we look forward to improving its visibility in the future.

Acknowledgment. This work supported by the funding recieved from the EU H2020 R&I project BOOST4.0 (GA 780732).

REFERENCES

\[\text{10.} \text{W. W. Consortium et al., “Rdf 1.1 concepts and abstract syntax,” WC3 Archive, 2014.}\]


16Travis CI API (https://docs.travis-ci.com/api/)
17https://github.com/apache/tinkerpop/graphs/contributors
18Gremlin-users (https://groups.google.com/forum/##!forum/gremlin-users)
19TinkerPop JIRA (https://issues.apache.org/jira/projects/TINKERPOP/)