A Stitch in Time Saves Nine – SPARQL querying of Property Graphs using Gremlin Traversals

Harsh Thakkar
Smart Data Analytics,
University of Bonn
Bonn, Germany 53117
thakkar@cs.uni-bonn.de

Dharmen Punjani
National and Kapodistrian University of Athens
Athens, Greece 45678
dpunjani@di.uoa.gr

Yashwant Keswani
DA-IICT
Gandhinagar, India 382007
201301047@daiict.ac.in

Jens Lehmann
Smart Data Analytics,
University of Bonn
Bonn, Germany 53117
lehmann@cs.uni-bonn.de

Sören Auer
TIB, Leibniz University of Hannover
Hannover, Germany 30167
soeren.auer@tib.eu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference format:
Harsh Thakkar, Dharmen Punjani, Yashwant Keswani, Jens Lehmann, and Sören Auer. 2016. A Stitch in Time Saves Nine – SPARQL querying of Property Graphs using Gremlin Traversals. In Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference’17), 13 pages. DOI: 10.1145/nmmnnn.nmmnnn

1 INTRODUCTION

Knowledge graphs have become increasingly popular over the past decade. The two most popular data models for storing knowledge graphs are property graphs (PG) and the Resource Description Framework (RDF). In this article, we present an approach for executing SPARQL queries, the standard query language for RDF, over graph databases. This provides several benefits: (1) Applications based on existing W3C Semantic Web standards, like SPARQL and RDF, can integrate property graph databases in a non-intrusive fashion. (2) It lays the foundation for a hybrid use of RDF triple stores and property graph DMS – where a particular query can be dispatched to the DMS capable to answer the query more efficiently [9]. In particular, property graph databases have been shown to work very well for a wide range of queries which benefit from locality in a graph. Rather than performing expensive joins, property graph databases use micro indices to perform traversals. (3) Users familiar with the W3C standardised SPARQL query language can avoid going through a steep learning curve of learning another query language.

We realize our approach through a conversion of SPARQL graph pattern queries to Gremlin graph traversals. To the best of our knowledge, this is the first work addressing the translation problem. Related work (cf. Section 2) mostly covers the area of SPARQL to SQL conversion and vice versa. In contrast to those previous efforts, we have to overcome the challenge of mediating between two very different execution paradigms: While SPARQL uses pattern matching techniques, Gremlin is based on performing graph traversals. More specifically, previous efforts applied query rewriting techniques between formalisms, which are ultimately rooted in relational algebra operations, whereas we had to bridge more disparate query paradigms. While this is a significant challenge, it is also the reason why substantial performance improvements can be made depending on the query load: Whereas direct SPARQL query

ABSTRACT

Knowledge graphs have become popular over the past decade and frequently rely on the Resource Description Framework (RDF) or Property Graph (PG) databases as data models. However, the query languages for these two data models – SPARQL for RDF and the property graph traversal language Gremlin – are lacking interoperability. We present Gremlinator, a novel SPARQL to Gremlin translator. Gremlinator translates SPARQL queries to Gremlin traversals for executing graph pattern matching queries over graph databases. This allows to access and query a wide variety of Graph Data Management Systems (DMSs) using the W3C standardized SPARQL and avoid the steep learning curve of a new Graph Query Language (GQL).

Gremlin is a graph computing system agnostic traversal language (covering both OLTP graph database or OLAP graph processors), making it a desirable choice for supporting interoperability for querying Graph DMSs. We present an empirical evaluation for the validity of our approach by formalising the graph pattern matching construct of Gremlin and illustrate its mapping to corresponding SPARQL queries. Moreover, we also present a proof-of-concept implementation of our approach, demonstrate its validity and applicability by executing SPARQL queries on top of leading Graph stores (Neo4j, Sparksee and Apache TinkerGraph) and compare their performances with RDF stores (Openlink Virtuoso, 4Store and JenaTDB). The results indicate that, for complex queries (such as Star-shaped), Gremlin pattern matching traversals out-perform their corresponding SPARQL queries significantly, including their translation time. Gremlinator currently covers a subset of the SPARQL 1.0 specification, specifically the SELECT queries.

KEYWORDS

SPARQL, Gremlin, RDF Graph, Property Graph, Pattern Matching, Graph Traversal, Query Translator, Gremlinator
execution can be expected to be suitable for large analytical joins over the entire dataset, the Gremlin conversion can significantly speed up all queries exploiting graph locality.

We selected Gremlin as target language, since it is supported by a wide range of property graph databases. Moreover, it provides support for both declarative and imperative constructs. Together with the Apache TinkerPop abstraction framework (which resembles a Java-JVM analogy) it provides support to access both OLTP graph databases or OLAP graph processors. Furthermore, in Gremlin it is possible to switch between the imperative (graph traversal) and declarative (graph pattern matching) style [26].

We map SPARQL queries to (only the) pattern matching Gremlin traversals (i.e. we map SPARQL queries to declarative construct of Gremlin and not imperative). Thus, ensuring a level fairness when comparing the performance of both Graph Query Languages (GQLs). Furthermore, instead of translating SPARQL queries to a specific dialect of Gremlin (e.g. Gremlin-Java8, Gremlin-Python etc.), we map each corresponding operation within a SPARQL basic graph pattern (BGP) to its corresponding traversal step in the Gremlin instruction library (i.e. a single step traversal operation). Using this, we build complex pattern matching traversals, in an analogous fashion to SPARQL style querying wherein multiple BGPs form complex graph patterns (CGPs). Thus, for each SPARQL query, it is possible to construct a corresponding Gremlin traversal representation.

Overall, we make the following contributions:

- A novel approach for mapping SPARQL queries to Gremlin pattern matching traversals, which is (to the best of our knowledge) the first work converting an RDF to a property graph query language.
- An extensible and openly available implementation for executing SPARQL queries over various graph DMSs such as Sparksee and Neo4j.
- A description of the relationship between the different query execution paradigms – pattern matching for SPARQL and traversal for Gremlin.
- An evaluation of the translation approach using an experimental study comprising a state-of-the-art property graph databases and triple stores on the Northwind and BSBM datasets.

The remainder of the article is organized as follows: Section 2 covers related query conversion efforts. Section 3 introduces preliminary notions. Section 4 describes the relationship between SPARQL graph pattern matching and Gremlin traversal steps. Section 5 explains our mapping approach. Section 6 evaluates the approach on two datasets and discusses the results. Finally, Section 7 concludes and describes future work.

2 RELATED WORK

In this section we briefly survey the related work with regard to techniques and tools that support the translation and execution of GQLs.

SPARQL → SQL: There is a substantial amount of work been done for conversion of SPARQL queries to SQL queries [5, 6, 10, 22, 29, 34]. Ontop [5, 6] exposes relational databases as virtual RDF graphs by linking the terms (classes and properties) in the ontology to the data sources through mappings. This virtual RDF graph can then be queried using SPARQL by dynamically and transparently translating the SPARQL queries into SQL queries over the relational databases. The work presented in [29] generates SQL that is optimized and also provides a well-defined specification of the SPARQL semantics used in the translation. In addition, Ontop also supports R2RML mappings over general relational schemas. The authors show that their implementation can outperform other well known SPARQL-to-SQL systems, as well as commercial triple stores by large margin. In [10] a SPARQL-to-SQL translation technique is introduced, that focuses on the generation of efficient SQL queries. It relies on a mapping language that lacks support for URI templates and is less expressive than R2RML. [6] proposes a translation function that takes a query and two many-to-one mappings: (i) a mapping between the triples and the tables, and (ii) a mapping between pairs of the form (triple, pattern, position) and relational attributes. In addition, the approach in [6] assumes that the underlying relational DB is denormalized, and stores RDF terms. The two semantics deviate in the definition of the OPTIONAL algebra operator. The work in [22] is the extension of work in [6] to include R2RML mappings. [34] makes use of non-standard SQL constructs for SPARQL–SQL translation and lacks the formal proof that the translation is correct and an empirical evaluation with realistic data is missing.

SQL → SPARQL: The work in [24] presents a formal semantics preserving the translation from SQL to SPARQL. RETRO [24] deals only with schema mapping and query mapping rather than to transform the data physically. Schema mapping derives a domain-specific relational schema from RDF data. Query mapping transforms an SQL query over the schema into an equivalent SPARQL query, which in turn is executed against the RDF store.

SQL → CYPERH: CYPERH is the graph query language used to query the Neo4j graph database. There has been no work yet aiming to convert the RDBMS to CYPERH. However, there are some examples that show the equivalent CYPERH queries for certain SQL queries.

3 PRELIMINARIES

In this section, we recall and summarize the mathematical concepts which will be used in this article. Our notation closely follows [1] and extends [28] by introducing the notion of vertex labels.

3.1 Graph Data Models

3.1.1 Edge-labeled Graphs. The Resource Description Framework (RDF) is a well-known W3C standard, which is used for data modeling and encoding machine readable content on the Web [7] and within intranets. An RDF graph can be seen as a set of triples, roughly analogous to nodes and edges in a graph database. However, RDF is more specific in defining disjoint vertex-sets of blank nodes, literals and IRIs. In the rudimentary form, an RDF graph is a directed, edge-labeled, multi-graph or simply an edge-labeled
We adapt the definition provided by [1], which is the closest to our context of SPARQL and Gremlin. We focus on the pattern matching features throughout this article.

Apache TinkerPop modern crew graph. In our current context, we do not consider blank nodes. Edge-labeled graphs can be used to encode complex information despite their elementary structure [1]. Edge-labeled graphs have been formally defined in a wide variety of texts, such as [1, 8, 20, 21, 25]. We adapt the definition provided by [1], which is the closest to our current context:

**Definition 3.1 (Edge-labeled Graph).** An edge-labeled graph is defined as \( G = (V, E, \lambda, \mu) \); where:
- \( V \) is the set of vertices,
- \( E \) is the set of directed edges such that \( E \subseteq (V \times \lambda \times V) \) where \( \lambda \) is the set of Labels,
- \( \lambda \) is a function that assigns labels to the edges and vertices (i.e. \( \lambda : V \cup E \rightarrow \Sigma^* \)),
- \( \mu \) is a partial function that maps elements and keys to values (i.e. \( \mu : (V \cup E) \times \lambda \rightarrow S \)) i.e. properties (key \( r \in R \), value \( s \in S \)).

3.1.2 Property Graphs. Property graphs, also referred to as directed, edge-labeled, attributed multi-graphs, have been formally defined in a wide variety of texts, such as [1, 12, 17, 23, 27]. We adapt the definition of property graphs presented by [27]:

**Definition 3.2 (Property Graph).** A property graph is defined as \( G = (V, E, \lambda, \mu) \); where:
- \( V \) is the set of vertices,
- \( E \) is the set of directed edges such that \( E \subseteq (V \times \lambda \times V) \) where \( \lambda \) is the set of Labels,
- \( \lambda \) is a function that assigns labels to the edges and vertices (i.e. \( \lambda : V \cup E \rightarrow \Sigma^* \)),
- \( \mu \) is a partial function that maps elements and keys to values (i.e. \( \mu : (V \cup E) \times \lambda \rightarrow S \)) i.e. properties (key \( r \in R \), value \( s \in S \)).

Figures 1 and 2, present different data model visualizations of the Apache TinkerPop modern crew graph. We use these as running examples throughout this article.

3.2 Graph Pattern Matching

The Graph Pattern Matching (GPM) problem is generally perceived as a subgraph matching problem (aka subgraph isomorphism problem) [25]. GPM can be done on both undirected and directed graphs respectively. Traditionally GPM is a computational task that can be defined as the evaluation of graph patterns over a graph database [1]. The most trivial form of a graph pattern is the basic graph pattern (BGP). A BGP coupled with features such as projection, union, difference and optional forms a complex graph pattern (CGP). We illustrate these concepts in brief with respect to context of SPARQL and Gremlin GQLs in Section 4. Detailed information on GPM is available in [1, 12, 25].

GPM is carried out by matching (also referred to as evaluation), a sub-graph pattern over a graph \( G \). Matching has been formally defined in various texts and we summarize a formal definition in our context which closely follows the definition provided by [1, 12].

**Definition 3.3 (Match of a Graph Pattern \([P](G)\)).** A graph pattern \( P = (V_p, E_p, \lambda_p, \mu_p) \) is matching the graph \( G = (V, E, \lambda, \mu) \), iff the following conditions are satisfied:

1. There exist mappings \( \mu_p \) and \( \lambda_p \) such that, all variables are mapped to constants, and all constants are mapped to themselves (i.e. \( \lambda_p \in \lambda, \mu_p \in \mu \)),
2. Each edge \( e \in E_p \) in \( P \) is mapped to an edge \( e \in E \) in \( G \), each vertex \( v \in V_p \) in \( P \) is mapped to a vertex \( v \in V \) in \( G \), and
3. The structure of \( P \) is preserved in \( G \) (i.e. \( P \) is a sub-graph of \( G \)).

The definition for matching for edge-labeled graphs is analogous to that of the property graph (ref. Def. 3.3): (i) a mapping \( m \) maps the constants to themselves and variables to constants; and (ii) the structure of \( P \) is preserved in \( G \) (example ref. Figure 2).

3.3 SPARQL Query Language

SPARQL is a declarative GQL which is a W3C recommendation and the query language of the RDF triple stores. The building blocks of SPARQL are RDF triple patterns, consisting of subject, predicate, and object, and where either of it can be a variable, literal value or IRI. In this work, we do not consider the blank node semantics.

**Definition 3.4 (SPARQL BGP).** A SPARQL query defines a graph pattern to be matched against a given RDF graph. A basic graph pattern \( P \) is a sub-graph of \( G \)
A step (analogous to how Scala compiles to the JVM). As a result, there is a step function, which maps the declared variables (elements and keys) to their respective values.

The evaluation (also know as matching, ref. Def. 3.3) of a graph pattern in Gremlin is taken care by two functions:

1. the recursively defined match() function, which evaluates each constituting graph pattern and keeps track of the traverser’s location in the graph (i.e. path history), and,
2. the bind() function, which maps the declared variables (elements and keys) to their respective values.

The evaluation of a graph pattern in Gremlin is carried out by the match()-step. We borrow the notation of the evaluation of a graph pattern $([\mathcal{O}_G])$ from [21] for representing the evaluation of a Gremlin traversal $\Psi$ over a graph $G$, i.e. $[\Psi]_{G}^{gm}$. Details of execution of the match()-step in Gremlin are described in [26].

4 SPARQL ↔ GREMLIN HOMOLOGY

In this section we present the correspondence between SPARQL BGP/CGP with Gremlin pattern matching path traversals. In doing so we devise a formal analogy borrowing the evaluation semantics of a SPARQL query [1, 21, 30] (referring to the well established bag semantics) and put them in context of Gremlin traversals [26–28]. Furthermore, we illustrate the applicability of these concepts with respect to the running examples (as shown in Figures 1 and 2).

4.1 Graph Pattern Matching via Traversing

A SPARQL query consists of several BGP’s which when used with features such as projection or union, form a CGP (as we discussed in section 3.3). BGP’s (ref. Definition 3.4) are comprised of triple patterns, which match to RDF triples that constitute the RDF dataset. Moreover, the RDF data model resembles essentially a directed, edge-labeled, multi-graph or RDF graph. It is, therefore, possible to traverse an RDF graph with Gremlin (i.e. construct traversals), regardless of it being an edge-labeled graph or a property graph as the core-concept of traversing remains unaltered.

Analogous to SPARQL, Gremlin also provides the GPM construct, using the Match()-step. This enables the user to represent the query using multiple individually connected or disconnected graph patterns. Each of these graph patterns can be perceived as a simple path traversal, to-and-from a specific source and destination, over the graph.

\[ p \in E \text{ and } E \subseteq (V \times V) \]

\[ \mathbb{E}_0 = [e] \text{, } e \text{ is the identity/empty element.} \]
In Gremlin, each traversal can be perceived as a path query, starting from a particular source (A) and terminating at a destination (B) by visiting vertices \( v \in V \) and edges \( e \in E(V \times V) \). Each path query is composed of one or more single-step traversals (SST) as shown by [26]. Through function composition and currying, it is possible to define a query of arbitrary length [26]. Furthermore, just as multiple BGP s form a CGP in SPARQL, the corresponding SSTs can be coupled with features such as projection, union, optional, etc. to form a complex path traversal query in Gremlin. These path queries can be a combination of either a source, destination, labeled traversal or all of them in a varying fashion, depending on the information need of the user.

### 4.2 SPARQL BGP s as Gremlin Single Step Traversals

In this section we establish the exact analogy between SPARQL BGP s and Gremlin SSTs. In SPARQL, GPM is conducted by matching BGP s which consist of triple patterns (TP), that form a sub-graph, against an RDF graph \( G \) (i.e. checking whether a sub-graph is contained in \( G \)). We can represent BGP s notationally as:

\[
BGP = \{ TP . \}^+ ; \quad TP = \{ s \ p \ o . \}^{14}
\]

In this unique representation, each subject and object (i.e. nodes) in a triple is connected through only one predicate relation (i.e. edges). Figure 1 presents an example of the graph representation of a sample RDF dataset.

In Gremlin, GPM is conducted by the \texttt{match(\()\)-step, wherein each above graph patterns, represented as pattern matching traversals are evaluated against a graph \( G \). We already know that Gremlin allows a user to form traversals of arbitrary length using function currying and composition. Due to this functionality and given the nature of the information represented in a triple, it is possible to represent the underlying traversal operation using at most one SST, which represented by its predicate/edge.

For instance, consider the BGP in listing 1, where the information need is to find what marko has created. This pattern, i.e. a subgraph formed by the BGP will be matched against a graph (ref. Figure 1) to bind the values of the variables labeled as “\( x \)” to “marko”, and “\( y \)” to the name property of the node connected by the edge/predicate labeled “created” by “\( x \)”. Listing 1 represents the SPARQL BGP as shown in Figure 3(a).

| Listing 1: What has marko created? |
|-----------------------------------|
| \{ ?x a: name "marko" . ?x a: Created ?y. \} |

Now consider its a Gremlin traversal for the corresponding SPARQL BGP, as shown in listing 1 from Figure 2. Here the underlying SSTs required are \( .\text{has(‘name’, ‘marko’) and .out(‘Created’)} \), that map to the HasStep() and VertexStep() instructions in the Gremlin instruction-set library [26, 28] respectively.

| Listing 2: An outgoing traversal from the vertex "marko" via an edge labeled "created" |
|-----------------------------------|
| \texttt{g.V().as(’x’).has(’name’, ’marko’).out(’Created’).as(’y’)} |

14The * operator symbolizes that a BGP can also be an empty graph pattern.

### 4.3 SPARQL Queries as Gremlin Pattern Matching Traversals

In SPARQL query language, as we have mentioned in earlier sections, a query comprises of one or more CGP s which in turn are formed by a combination of BGP s.
Where, \( \sigma \) and \( \psi \) are the relational operators we also present the SPARQL query language constructs and their representation, Union, Diff. and Filter are the relational operators defined on the BGPs.

Similarly, in Gremlin traversal language, a pattern matching traversal comprises of one or more path traversals which in turn are comprised of a combination of several SSTs.

From the already well established semantics of SPARQL query language [1, 3, 8, 14, 21], a query (Q) can be notationally represented as:

\[
Q = ([\text{PROJ.}] \ BGP \ [\text{UNION/DIFF./OPT.}] \ BGP \ [\text{Filter (c)}])^+ \tag{3}
\]

Where, Proj., Union, Diff. and Filter are the relational operators defined on the BGPs.

We have already established from Table 1, that each BGP can be mapped to a corresponding Gremlin single step traversal (\( \sigma(BGP) = \psi_s \)). Thus, from equations (2, 3), we can create a mapping function \( \sigma \), such that:

\[
\sigma(BGP) = \psi_s \tag{4}
\]

\[
\sigma(Q) = \psi \sigma([\text{PROJ.}] \ BGP \ [\text{UNION/DIFF./OPT.}] \ BGP \ [\text{Filter (c)}])^+ = \psi \sigma([\text{PROJ.}]) \psi_s \sigma([\text{UNION/DIFF./OPT.}]) \psi_s \sigma([\text{Filter (c)}])^+ = \psi \tag{5}
\]

Where, \( \sigma([\text{PROJ.}]) \), \( \sigma([\text{UNION/DIFF./OPT.}]) \) and \( \sigma([\text{FILTER (C)}]) \) represent the respective Gremlin instruction steps for the Projection, Union, etc operators. We present a consolidated summary of the correspondence between SPARQL features/keywords and their corresponding Gremlin instruction steps in Table 2. Furthermore, we also present the SPARQL query language constructs and their corresponding Gremlin traversal language constructs in Table 2.

The evaluation of a SPARQL query \( Q \) is carried out by matching or evaluating the graph patterns within \( Q \), against a graph \( G \) (an RDF graph in this case), denoted as \( P^\text{sparql}_G \). Similarly, in Gremlin traversal language and machine, the evaluation of a pattern matching traversal \( \Psi \) is carried out by the \( \text{match}() \) step, by matching or evaluating the SSTs within \( \Psi \) against a graph \( G \) (a property graph in this case). We borrow the same notation from [8, 21] to fit our purpose and denote as \( \Psi^\text{gremlin}_G \).

We display brevity in constructing our arguments by quick examples instead of re-inventing the wheel by re-defining formal concepts and proofs (which already have been addressed in the works [1, 3, 15, 19]). Moreover, we illustrate using examples, the semantic analogy between the evaluation of Gremlin traversal features in a homologous fashion to that of the multi-set semantics of SPARQL queries defined by [3] who extend the work of [8, 21]. We show, by structural analogy created with the evaluation semantics of SPARQL, that:

\[
[Q]_G^{\text{sparql}} \equiv \Psi_G^{\text{gremlin}} (: \sigma(Q) = \Psi, \text{Eqn : 5}) \tag{6}
\]

Projection. The \( \text{projection} \) operator projects/selects the values of a specific set of variables \((x_1, \ldots, x_n)\), from the solution of a matched graph pattern \( P \), against the graph \( G \). Furthermore, it is possible to declare variables in Gremlin using \( \text{as()} \) steps, which serve as syntactic sugars. For instance, in the CGP as shown in Figures 3(b) and 4(b), we project only variable \( ?a \) despite using \( ?a, ?b \) & \( ?c \) in the query, since we are only interested knowing the value binded to it. It is carried out using the \( \text{SELECT} \) keyword in SPARQL, and corresponding \( \text{.select()} \) step in Gremlin. The corresponding evaluation of a select step in Gremlin can be illustrated as:

\[
[\text{SELECT} \ ?x1 \ ?x2 \ {BGP}]_G^{\text{sparql}} = \sigma([\text{SELECT} ?x1 ?x2 \ {BGP}])_G^{\text{gremlin}} = \Psi
\]

\[
[\text{SELECT} \ ?x1 \ ?x2 \ {BGP}]_G^{\text{sparql}} = \sigma([\text{SELECT} ?x1 ?x2 \ {BGP}])_G^{\text{gremlin}} = \Psi_G^{\text{gremlin}} \tag{7}
\]

Here, \( \psi_s \) (SST) and \( \text{SelectStep}[\{x1,x2\}] \) collectively form the final pattern matching traversal (analogous to a collection of BGPs and SSTs forming). Moreover, \( \psi_s \) is mapped from Table 1, depending on the case it corresponds.

Optional. The \( \text{optional} \) operator in corresponds to a left-join operation (in relational sense). The optional graph patterns in a query are declared using this operator. For instance, given the CGP: \( \text{BGP1 \ optional} \). \( \text{BGP2} \); if the optional \( \text{BGP2} \) does not match with graph \( G \), then the results of \( \text{BGP1} \) are returned unchanged, else additional bindings of \( \text{BGP2} \) are added to the solution. It is present in both SPARQL (as \( \text{OPTIONAL} \)) and Gremlin (as \( \text{.optional} \)) keyword which corresponds to \( \text{ChooseStep}() \) in the Gremlin instruction

| S.S.T. [26] | Basic Graph Pattern (BGP) | Corresponding Gremlin Traversal Step \( \sigma(BGP) = \psi_s \) | Case |
|-------------|---------------------------|-------------------------------------------------|------|
| L\(_v\)     | ( ? \( x \) :\text{label} "person" .) | [\text{MatchStartStep}(x), \text{HasStep}([-\text{label.eq(person)}]), \text{MatchEndStep}] | L    |
| L\(_e\)     | ( ? \( x \) :\text{label} "knows" .) | [\text{MatchStartStep}(x), \text{HasStep}([-\text{label.eq(knows)}]), \text{MatchEndStep}] | L    |
| P\(_v\)     | ( ? \( v \) :\text{name} "marko" .) | [\text{MatchStartStep}(x), \text{HasStep}([\text{name.eq(marko)}]), \text{MatchEndStep}] | P1   |
| P\(_e\)     | ( ? \( e \) :\text{weight} "0.8" .) | [\text{MatchStartStep}(x), \text{HasStep}([\text{weight.eq(0.8)}]), \text{MatchEndStep}] | P2   |
| \( \psi \)  | \( \sigma \)  | [\text{MatchStartStep}(x), \text{PropertiesStep}([\text{name}], \text{value})], \text{MatchEndStep(y)}] | E    |
| \( \varnothing \) | \( \psi \)  | [\text{MatchStartStep}(x), \text{VertexStep}(\{\text{OUT}, \text{[knows], vertex}\}], \text{MatchEndStep(y)}] | E    |

Table 1: Mapping between the SPARQL BGPs, Gremlin SSTs and their corresponding Traversal steps. Each case of a SPARQL BGP can be mapped one-to-one to a Gremlin SST as described in Sect. 4.2.
with respect to specified equality/inequality/regular expressions (i.e. constraints). It is present in both SPARQL (as \(\text{FILTER}(C)\), where \(C\) is the declared constraint) and Gremlin (as \(.\text{where}(C)\), where \(C\) is the constraint). In Gremlin the \(.\text{where}(C)\) keyword corresponds to the WhereTraversalStep() from the instruction set library

\[
[BGP_1 \text{ OPT. } BGP_2]_{G}^{\text{sparql}} = [\sigma(BGP_1 \text{ OPT. } BGP_2)]_{G}^{\text{gml}}
= [\sigma(BGP_1) \text{ ChooseStep}(\sigma(BGP_2))]_{G}^{\text{gml}}
= [\psi_{s_1}, \text{ChooseStep}(\psi_{s_2})]_{G}^{\text{gml}} = \Psi_{G}^{\text{gml}}
\]

\[
[BGP_1 \text{ UNION } BGP_2]_{G}^{\text{sparql}} = [\sigma(BGP_1 \text{ UNION } BGP_2)]_{G}^{\text{gml}}
= [\text{UnionStep}(\sigma(BGP_1)), \sigma(BGP_2))]_{G}^{\text{gml}}
= [\text{UnionStep}(\psi_{s_1}, \psi_{s_2})]_{G}^{\text{gml}} = \Psi_{G}^{\text{gml}}
\]

\[
[BGP \text{ FILTER } C]_{G}^{\text{sparql}} = [\sigma(BGP \text{ FILTER } C)]_{G}^{\text{gml}}
= [\sigma(BGP) \sigma(\text{FILTER } C)]_{G}^{\text{gml}}
= [\psi_{s}, \text{WhereTraversalStep}(\psi_{c})]_{G}^{\text{gml}}
= \Psi_{G}^{\text{gml}}
\]

Union. The union operator combines the solution sets of the two input graph patterns. In SPARQL, union occurs between two BGP or CGPs, analogously in Gremlin, it occurs between two SSTs and Traversals (i.e. the result set of two traversers). The solution set returned after the union operation is not de-duplicated by default, because of the governing bag semantics. Thus, all possible solutions are returned. Formally, the evaluation of a union can be illustrated as:

For instance, consider the sample SPARQL CGP with UNION over the graph G (ref. Figure 2) as illustrated in the example below. The idea is to find the all the software created by “marko” which are in “java” language.

Illustration of a CGP with UNION

| SPARQL CGP | Gremlin Traversal (\(\sigma(BGP)\)) |
|------------|----------------------------------|
| \{ ?soft v:lang "java" UNION \{ ?person v:name "marko" \} \} | \([\text{MatchStartStep(a)}, \text{PropertiesStep([name],value), \text{EndStep}]}, \text{[StartStep([soft]), \text{PropertiesStep([lang],value), \text{EndStep}]})\) |

FILTERs. The filter keyword (or a group of operators) is used to restrict the results based on user-defined criteria. Filters declare one or more constraints on the variables in the query, depending on the need of the user, and limit the solution of the overall group of BGP with respect to specified equality/inequality/regular expressions (i.e. constraints). It is present in both SPARQL (as \(\text{FILTER}(C)\), where \(C\) is the declared constraint) and Gremlin (as \(.\text{where}(C)\), where \(C\) is the constraint). In Gremlin the \(.\text{where}(C)\) keyword corresponds to the WhereTraversalStep() from the instruction set library

Illustration of a CGP with FILTER

| SPARQL CGP | Gremlin Traversal (\(\sigma(BGP)\)) |
|------------|----------------------------------|
| \{ ?a v:name \_ \_ \_ \_ FILTER {?d?30} \} | \([\text{MatchStartStep(a)}, \text{PropertiesStep([age],value), \text{EndStep}]}, \text{[MatchEndStep(d)], \text{WhereTraversalStep(\text{[WhereStartStep(d), \text{IsStep(\text{lt(30)})}]})}\) |

Like in SPARQL, it is possible to declare multiple constraints inside a single FILTER clause:

FILTER(C1 & C2) \rightarrow \text{WhereTraversalStep(\text{AndStep(\{C1, C2\})})}
FILTER(C1 || C2) \rightarrow \text{WhereTraversalStep(\text{OrStep(\{C1, C2\})})}

For brevity we skip the illustration of this step, as it being perceptible.

Query Modifiers. The solution set returned by the evaluated graph patterns is NOT de-duplicated or ordered by default, as both the languages operate on bag semantics. Therefore, query modifiers or solution sequence modifiers are used for presenting the results in the desired order. We list out query modifiers, their corresponding keywords and language constructs in Table 2. Examples of query modifiers include \(\text{DISTINCT}\) (for result de-duplication), \(\text{LIMIT} \& \text{OFFSET}\) (for restricting no. of results), \(\text{GROUP BY}\) (for grouping manipulation of result stream), \(\text{ORDER BY}\) (for ordering manipulation of result stream). For brevity we skip the formal definitions of each modifier, rather illustrate their correspondence and applicability in Table 2.

Table 2: A consolidated list of SPARQL features/keywords & their corresponding Instruction steps in Gremlin.

| Operation | SPARQL keyword | Gremlin Step | SPARQL construct (Q) | Gremlin construct σ(Q) = Ψ |
|-----------|----------------|--------------|----------------------|-----------------------------|
| Graph Pattern(s) | \{ s p o \} | \ψ (i.e. σ(s p o .)) | BGP | \ψ (single step traversal [list of ψ]) |
| Matching | \WHERE \{ \} | MatchStep(AND, []) \(\Psi \) | WHERE \{ BGP1 \ BGP2 \} | \[MatchStep(\text{AND}, \{[ψ_1], [ψ_2]\})\] |
| Restriction | \FILTER(C) | WhereTraversalStep(p(C)) | BGP1 \ BGP2 | WhereTraversalStep(value(v1), IsStep(lt(30))) |
| Join | JOIN | AndStep() | BGP1 \ BGP2 | AndStep([ψ_1], [ψ_2]) |
| Projection | SELECT | SelectStep() | SELECT \(?v1 \_\_2\) | SelectStep([a, b],) |
| Combination | UNION | UnionStep() | BGP1 UNION BGP2 | UNIONStep(p(BGP1),p(BGP2)) |
| Deduplication | DISTINCT | DedupStep() | DISTINCT \(?v1\) | DedupStep([a, b],) |
| Restriction | LIMIT(M) | RangeStep(0, M) | LIMIT 2 | RangeStep(0, 2) |
| Restriction | OFFSET(N) | RangeStep(N, M+N) | OFFSET 10 | RangeStep(10, 12) |
| Sorting | ORDER BY() | OrderStep() | ORDER BY DESC(?a) | OrderStep([value(a), desc]) |
| Grouping | GROUP BY() | GroupStep() | GROUP BY(?a) | GroupStep(value(a)) |
5 APPROACH

In this section we present our approach: Gremlinator.

5.1 Considerations

We clarify requirements which need to be satisfied for our approach to work.

Dataset Consistency. The data in both the data models should be consistent, i.e., the data in the RDF graph version of the dataset should be the same in its Property Graph representation. We validated the consistency of both the version (ref. Sec. 6.1) and present summarized statistics in Table 3. If the datasets are not consistent, correct results cannot be guaranteed.

Encoding SPARQL prefixes. We encode the prefixes of SPARQL queries within Gremlinator implementation. In order to aid the SPARQL to Gremlin translation process, we define custom prefixes keeping in mind the four categories of SSTs (as stated in sec. 4.2). For instance, the standard rdfs:label prefix (which is generally a predicate) is represented as e:label or v:label (where e = edge and v = vertex). A similar procedure is followed for other three cases.

5.2 Gremlinator Architecture & Algorithm

We present the architectural overview of Gremlinator in Fig. 5. We now discuss in brief the role of each of the four-step execution pipeline.

Step 1. The input SPARQL query is first parsed using the Jena ARQ module, thereby: (i) validating the query and (ii) generating its abstract syntax tree (AST) representation.

Step 2. From the obtained AST of the parsed SPARQL query, Gremlinator then visits each BGP, mapping them to the corresponding Gremlin SSTs ($\psi_s$, ref. Table 1).

Step 3. Thereafter, depending on the operator precedence obtained from the AST of the parsed SPARQL query, each of the corresponding SPARQL keywords are mapped to their corresponding instruction steps from the Gremlin instruction library (ref. Table 2). Thereafter a final conjunctive traversal ($\Psi$) is generated appending the SSTs and instruction steps.

Step 4. This final complex traversal is used to generate bytecode (optional) which can be used in multiple language and platform variants of Apache TinkerPop Gremlin family.

Algorithm. We present the SPARQL to Gremlin translation algorithm in Algorithm 1.

6 EXPERIMENTAL EVALUATION

6.1 Datasets

Northwind – is a synthetic-dataset with an e-commerce scenario between a fictional company "Northwind Traders", its Customers, Suppliers. Originated as a sample dataset shipped with Microsoft Access\textsuperscript{16}, it raised to fame with an enormous demand for e-commerce use cases in benchmarking DMSs. In Figure 6(a) we present the dataset schema. We obtained graph version of the dataset from here\textsuperscript{17}.

Berlin SPARQL Benchmark\textsuperscript{4} (BSBM) – is a synthetic dataset, which is built around an e-commerce use case, between a set of products, their vendors, consumers who review the products. It is a widely famous for benchmarking RDF DMSs as it offers the flexibility of generating graphs of custom size and density. We generated a standard 1M triples dataset using their data generation

Algorithm 1: SPARQL2Gremlin

input :SQ : Sparql Query
output :GT : Gremlin Traversal

1 $GT \leftarrow \emptyset$; $T \leftarrow \emptyset$ \hspace{1cm} // list of single step traversals $T$
2 $\rightarrow AST \leftarrow \text{getAST}(SQ)$; $\rightarrow BGPs \leftarrow \text{getAllBGPs}(AST)$
3 \hspace{1cm} foreach $bgp_i \in BGPs$ do
4 \hspace{2cm} $T \leftarrow T \cup \psi_i$ \hspace{0.5cm} // mapping BGP to Gremlin S.S.T. ($\psi_s = \sigma(k_{bgp_i})$
5 \hspace{1cm} end
6 \hspace{1cm} // We now map the corresponding Gremlinator operators for each SPARQL operator in the A.S.T. as shown in Table 2
7 \hspace{1cm} if $c \leftarrow \text{AST.FILTER}, \exists c \neq \emptyset$ then \hspace{1cm} // where $c$ is a SPARQL condition
8 \hspace{2cm} \text{foreach } c \in AST do \hspace{1cm} // list of single step traversals $T$
9 \hspace{3cm} $T \leftarrow T \cup \text{WhereTraversalStep}(\psi_c)$
10 \hspace{2cm} \end
11 \hspace{1cm} end
12 \hspace{1cm} if $c \leftarrow \text{AST.UNION}$ then
13 \hspace{2cm} $GT \leftarrow \text{UnionStep}(\text{Match}(T))$
14 \hspace{1cm} end
15 \hspace{1cm} if $|T| > 1$ then
16 \hspace{2cm} $GT \leftarrow \text{Match}(T)$
17 \hspace{2cm} else
18 \hspace{3cm} $GT \leftarrow GT \cup T$
19 \hspace{1cm} end
20 \hspace{1cm} if $c \leftarrow \text{AST.ORDERBY}$ then
21 \hspace{2cm} $GT \leftarrow T \cup \text{OrderStep}(\psi_c)$
22 \hspace{1cm} end
23 \hspace{1cm} if $c \leftarrow \text{AST.GROUPBY}$ then
24 \hspace{2cm} $GT \leftarrow T \cup \text{GroupByStep}(\psi_c)$
25 \hspace{1cm} end
26 \hspace{1cm} if $c \leftarrow \text{AST.LIMIT}$ then
27 \hspace{2cm} if $k \leftarrow \text{AST.OFFSET}$ then
28 \hspace{3cm} $GT \leftarrow T \cup \text{RangeStep}(k, c + k)$
29 \hspace{2cm} else
30 \hspace{3cm} $GT \leftarrow T \cup \text{RangeStep}(c)$
31 \hspace{2cm} end
32 \hspace{1cm} end
33 \hspace{1cm} return $GT$

---

\textsuperscript{16}Northwind Database (https://northwinddatabase.codeplex.com/)
\textsuperscript{17}SQL2Gremlin website – (http://www.sql2gremlin.com)
Table 3: Dataset statistics

| Criterion                  | Northwind |      | BSBM |      |
|----------------------------|-----------|------|------|------|
|                            | RDF       | PG   | RDF  | PG   |
| Classes                    | 11        | 159  |      |      |
| Entities & Nodes           | 4413      | 3209 | 71015| 92757|
| Distinct subjects          | 4413      |      |      |      |
| Distinct objects           | 8187      |      | 166384|      |
| Properties                 | 55        | 55   | 40   | 40   |
| Number of Triples & Edges  | 33003     | 6177 | 1000313| 238309|

Table 4: Query feature design and description

| Query | Feature      | FILTER | COUNT | LIMIT | DISTINCT | # Patterns | # Proj. Var. |  |
|-------|--------------|--------|-------|-------|----------|------------|-------------|---|
| C1    | CGP          | ✓      | ✓     |       |          | 2          | 2           |   |
| C2    | CGP          |        |       |       |          | 1          | 1           |   |
| C3    | CGP          |        |       | ✓     |          | 1          | 1           |   |
| F1    | CONDITION    | ✓      |       | ✓     |          | 3          | 3           |   |
| F2    | CONDITION    | ✓      |       | ✓     |          | 3          | 3           |   |
| F3    | CONDITION    | ✓      |       | ✓     |          | 2          | 1           |   |
| L1    | RESTRICTION  | ✓      |       | ✓     |          | 4          | 2           |   |
| L2    | RESTRICTION  | ✓      |       | ✓     |          | 2          | 2           |   |
| L3    | RESTRICTION  | ✓      |       | ✓     |          | 2          | 2           |   |
| G1    | GROUP BY     | ✓      |       | ✓     |          | 2          | 2           |   |
| G2    | GROUP BY     | ✓      |       | ✓     |          | 6          | 2           |   |
| G3    | GROUP BY     | ✓      |       | ✓     |          | 1          | 2           |   |
| Gc1   | GROUP COUNT  | ✓      | ✓     |       |          | 3          | 2           |   |
| Gc2   | GROUP COUNT  | ✓      |       | ✓     |          | 2          | 2           |   |
| Gc3   | GROUP COUNT  | ✓      |       | ✓     |          | 1          | 2           |   |
| O1    | ORDER BY     | ✓      |       | ✓     |          | 1          | 1           |   |
| O2    | ORDER BY     | ✓      |       | ✓     |          | 4          | 3           |   |
| O3    | ORDER BY     | ✓      |       | ✓     |          | 1          | 1           |   |
| U1    | UNION        | ✓      | ✓     | ✓     |          | 8          | 1           |   |
| U2    | UNION        | ✓      | ✓     |       |          | 6          | 2           |   |
| U3    | UNION        | ✓      | ✓     | ✓     |          | 4          | 1           |   |
| Op1   | OPTIONAL     | ✓      | ✓     |       |          | 3          | 3           |   |
| Op2   | OPTIONAL     | ✓      | ✓     | ✓     |          | 6          | 2           |   |
| Op3   | OPTIONAL     | ✓      | ✓     | ✓     |          | 8          | 3           |   |
| M1    | MIXED        | ✓      | ✓     | ✓     |          | 3          | 2           |   |
| M2    | MIXED        | ✓      | ✓     | ✓     |          | 2          | 2           |   |
| M3    | MIXED        | ✓      | ✓     | ✓     |          | 4          | 2           |   |
| S1    | STAR         | ✓      | ✓     | ✓     |          | 12         | 11          |   |
| S2    | STAR         | ✓      | ✓     | ✓     |          | 5          | 4           |   |
| S3    | STAR         | ✓      | ✓     | ✓     |          | 10         | 9           |   |

Figure 6: The dataset schema of (a) Northwind and (b) BSBM.

6.2 Queries

We created a set of 30 SPARQL queries, for each dataset, which covers 10 different SPARQL query features (i.e., three queries per feature with a combination of various modifiers). These features were selected after a systematic study of SPARQL query semantics [1, 21, 30] and from BSBM [4] explore use cases. A gold standard set of Gremlin traversals corresponding to the SPARQL queries was created by three Gremlin expert users, for a twofold validation of the traversals generated by our approach. Table 4 summarizes the query feature design which was used for our experiment.

Experimental Correctness.

- SPARQL query & result validation: We validated all the 60 SPARQL queries (30 x 2 datasets) by (i) parsing it through Jena ARQ, thereby verifying the syntactic validating; and (ii) executing the queries against a custom setup endpoint using Openlink Virtuoso for both the datasets, thereby verifying the semantic consistency.

- Gremlin query & result validation: Similarly for all 60 Gremlin queries (expert-curated), we followed the same procedure – (i) using Apache Groovy for verifying the syntactic validity; and (ii) getting each gremlin query tested by 3 different Gremlin experts against TinkerGraph DMS for semantic consistency.

- Translated traversal validation: We validated the translated traversals by – (i) comparing their results with respect to the results returned by the input SPARQL query; and (ii) also with respect to the results returned by the expert-curated corresponding Gremlin traversals. Thus conducting a two-fold validation of results returned by translated Gremlin traversal.

6.3 Experimental Setup

We selected the following DMSs for the experiments: RDF DMS: Openlink Virtuoso [11] [v7.2.4], JenaTDB [20] [v3.2.0], 4Store [13] [v1.1.5]; Graph DMS: TinkerGraph [16] [v3.2.3], Neo4j [21] [v1.9.6].
Sparks22 [v5.1]. All the experiments were performed on a machine with the following configuration: CPU: Intel® Xeon® CPU E5-2660 v3 (20 cores @2.60GHz), RAM: 128 GB DDR3, HDD: 512 GB SSD, OS: Linux 4.2-generic.

Evaluation Metrics. The following conditions and parameters were considered for recording all the results.

- Query execution time (in milliseconds or ms) considered is the average of 10 runs for each query (of both SPARQL and translated Gremlin traversals).
- Queries executed in both cold and warm cache settings for respective DMSs. Where a warm cache: implies that the cache is not cleared after each query run, and cold cache: implies that the cache is cleared using the `echo 3 > /proc/sys/vm/drop_caches` UNIX command after each query execution.
- For Graph DMSs, query execution time is recorded for both with and without creating explicit indices. We elaborate on the reason for the same, next.

Indexing in RDF vs Graph DMSs. It is known that, almost every RDF DMS indexes data as per some pre-defined indices. The same, however, cannot be said for Graph DMSs wherein these indices have to be created manually, depending upon the use case. For instance, Openlink Virtuoso maintains a set of all-purpose 2 full (bitmap indices over PSOG, POGS) and 3 partial indices (over SP, OP GS) in default configuration. Furthermore, 4Store in its default setting maintains a set of three full indices (R, P, M) [13], where the ‘R-index’ is a hash-map index over RDF resources (URIs, Literals, and Blank Nodes); ‘P-index’ consists of a set of two radix tries per predicate, using a 4-bit radix; the ‘M-index’ is a hash-map based indexing scheme over RDF Graphs (G). Lastly, Apache Jena TDB maintains three indices using a custom persistent implementation of B+ Trees.

On the other hand, Graph DMSs seldom maintain any default indexing scheme. They rather offer the possibility of declaring/creating explicit indexes over custom graph elements, using a variety of data structures, depending on the vendor’s implementation. For instance, TinkerGraph supports the creation of regular and composite hash-map indices (multiple key-value pairs) on graph elements (node and edge attributes). Neo4j allows declaring regular indices (composite indices are supported from v3.5 onwards) on graph elements (including labels). It offers a variety of indices ranging from Lucene index (for textual attributes) and as SBTree-based index (numeric ones, such as IDs), which is based on custom implementation of B-Trees with several optimizations related to data insertion and range queries. Lastly, like other Graph DMSs, Sparks2 also offers user-defined indices on attributes. It uses a bitmap index implemented using sorted B-trees.

As we pointed out earlier, it is not possible to have a completely index-free RDF DMS. Thus, in order to grasp a better understanding of query execution performance with respect to various factors (such as indexing schemes, query typology and cache configuration) and also for the sake of fairness (towards Graph DMSs) we run all the experiments with two settings of Graph DMSs, i.e. with (i.e. manually created) and without indices.

6.4 Results

The detailed results including the queries, dataset statistics, plots and full configuration settings can be obtained from here. The complete source code of Gremlinator is made publicly available, along with a recorded demonstration of Gremlinator in action, which can be accessed here. The complete setup including all the datasets, scripts, and DMSs can be found here. The average time for translating a SPARQL query to corresponding pattern matching Gremlin traversal is 14 ms for BSBM and 12.5 ms for Northwind queries respectively.

Figure 7, presents the plots of our experimental results, in all four settings, for the BSBM dataset. The plots follow log scale for execution time (in ms). We organize our observations on the performances of participating DMSs as follows, and present our discussion.

Graph DMSs without index: We categorize our findings in two groups – cold cache and warm cache. We observe that for –

1. **Cold cache**: SPARQL queries report a comparative advantage with respect to Gremlin traversals, leveraging the advantage of indexing schemes of RDF DMSs. SPARQL performs moderately faster (1x-2x) for simple queries (C1, C2) and order by (O1, O3); substantially faster (3x-5x) for union and mixed queries (U1-3, M1-3). Whereas, Gremlin traversals benefit from only the graph locality inherent to Graph DMSs. Gremlin traversals perform moderately faster (1x-2x) for restriction (L1, L3), group by (G1-3) and conditional (F1-3) queries; substantially faster (3x-5x) for group count (Gc1-3) and star (S1-3) queries. We also note that aggregation queries (counts, group counts) in Graph DMSs are an order of magnitude faster as compared to RDF DMSs since they do not have to execute multiple inner joins in addition to the aggregation operations. Moreover, for star-shaped queries (queries with bushy plans having >=5 triple patterns, >=1 filter and >=4 projection variables) Gremlin pattern matching traversals outperform their SPARQL counterparts by at least an order of one magnitude for S1, S2 and at least an order of two magnitudes for S3 (with 10 triple patterns, 1 filter and 9 projection variables).

2. **Warm cache**: SPARQL queries reap the most benefits of warm caching from RDF DMSs as compared to the Gremlin traversals from Graph DMSs. We observe that on average, in this setting, the improvement is up to 1x-1.8x for star and mixed queries, 2x-3x for aggregation (counts), condition (filter) and re-ordering (order by, group by) queries, and 3x-5x for CGPs and union queries. We also note that SPARQL queries are almost an order of magnitude faster than the corresponding Gremlin traversals for queries having a union operator, and are comparable for mixed, CGPs, and

22Sparksee – formerly DEX (https://sparity-technologies.com/#sparksee)
23RDF indexing scheme in Virtuoso (http://docs.openlinksw.com/virtuoso/edpffpdfscheme/)
24RDF indexing scheme in Apache Jena TDB (https://jena.apache.org/documentation/tdb/architecture.html#triple-and-quad-indexes)
25Indexing in Neo4j (http://neo4j.com/docs/developer-manual/current/cypher/schema/index/)
26Detailed results can be found at (https://goo.gl/CSSVz2)
27Gremlinator source code (https://github.com/LITMUS-Benchmark-Suite/sparql-to-gremlin)
28Experimental setup (https://github.com/harsh9/EDBT-2016-Experiments)
Figure 7: Performance comparison of SPARQL queries vs Gremlin (pattern matching) traversals for BSBM dataset with respect to RDF and Graph DMSs in different configuration settings.
We now discuss the findings of our experiments with respect to our findings based on the following factors: the factors which influence the query execution performance of a DMS, in warm cache. We report that on average, in this setting, the improvement is up to 1.3x for aggregation (count, group count) and star-shaped queries; up to 1.5x for re-ordering (order-by, group-by) and condition (filters) queries; up to 2x for mixed, union and restriction (limit) queries.

**Graph DMSs with indexing:** We manually created composite indices for each Graph DMS on attributes such as "name", "customerId", "unitPrice", "unitsInStock", "unitsOnOrder" for the BSBM dataset. Similarly, on "type", "productID", "reviewerID", "productTypeID" for the Northwind dataset, on the node attributes (numeric). The indices use the hash-map data structure. We did not re-execute SPARQL queries on RDF DMSs, as there was no change in the indexing setting for the same.

1. **Cold cache:** Gremlin traversals perform significantly faster when executed on Graph DMSs with composite indices. We observe that, as compared to the previous (cold cache + without index) setting, the improvement reported on an average is up to 1x-2x for union, mixed and group by traversals; up to 2x-3x for group-by, order-by traversals; up to 3x-5x for regular and restriction traversals; and >5x for aggregation and star-shaped traversals.

2. **Warm cache:** In this setting the Graph DMSs (i.e. Gremlin traversals) register similar performance gains to that in non-indexed configuration.

### 6.5 Discussion.

We now discuss the findings of our experiments with respect to the factors which influence the query execution performance of a particular DMS and summarize our observations. We categorize our findings based on the following factors:

- **Query typology:** We report that for - (i) simple/linear queries (such as C1-3, F1-3, L1-3) both SPARQL and Gremlin traversal performances are comparable; (ii) SPARQL outperforms corresponding Gremlin traversals for union queries. This is so because in SPARQL a union occurs between two or more sets of triple patterns. Whereas in the declarative construct (pattern matching) of Gremlin, a union occurs between two .match()-steps as a distinct traversal and then executes a union on top of it; (iii) Whereas, for complex queries (such as star-shaped and aggregation based queries), Gremlin traversals outperform their SPARQL counterparts. As mentioned before (ref. 6.4 – cold cache section), this is because Graph DMSs do not have to perform expensive joins (like RDF DMSs) on top of executing aggregation operations. (iv) Lastly, we also observe that for queries with greater number of projection variables (Proj. vars >= 3) and query modifiers (count, distinct, limit + offset, filter), Gremlin traversals show a distinctive advantage (more than an order of magnitude) in terms of performance with respect to corresponding SPARQL queries (e.g. for F1, F2, O2, S1, S2, S3). This advantage, while still exists, is not as pronounced when comparing queries with a fewer number of projection variables and query modifiers.

- **Query caching – Cold vs Warm:** Despite the fact that both DMSs benefit from warm cache query execution (as compared to cold cache), SPARQL queries receive the most advantage as compared to corresponding Gremlin traversals. One reason for this is that Gremlin traversals perform considerably better (except in cases of union queries) by leveraging the advantage of underlying property graph data model (locality) and cannot be optimized further without explicitly creating regular or composite indices. Out of all the three RDF DMSs, Jena shows the most gain in warm execution time, which receives up to 5x boost in cases such as union and CGP queries.

- **Indexing scheme:** It does not go without noticing, the one-sided dominance of Openlink Virtuoso, amongst all the evaluated RDF DMSs. As mentioned earlier, Virtuoso maintains a variety of full and partial indices. Moreover, we also know that virtuoso employs custom partition clustering and caching schemes on top of these indices to provide an adaptable solution to all kinds of workloads. One distinctive advantage in virtuoso is that the indices are column-wise by default, which takes one-third amount of space as compared to the row-wise indices. On the contrary, similar claims cannot be made about other RDF DMSs such as 4Store and JenaTDB. Graph DMSs, have a limited number options in terms of underlying indexing data structures implementation for creating manual indices in the chosen version. One reason can be deduced that there has not been an explicit need for using complex index schemes (as in Virtuoso), since composite indices based on B+ trees and hash-maps provide sufficient performance boost for graph traversal operations.

Thus, based on our findings, we can summarise that for complex queries (such as aggregation, star-shaped, and queries with higher number of projection variables + query modifiers) corresponding Gremlin pattern matching traversals outperform SPARQL queries. Whereas, for union-based queries SPARQL register significant performance advantage.

### 7 CONCLUSION AND FUTURE WORK

In this paper, we presented Gremlinator, a novel approach for supporting the execution of SPARQL queries on property graphs using Gremlin pattern matching traversals. Furthermore, we presented an empirical evaluation of our approach using state-of-the-art RDF and Graph DMSs, demonstrating the applicability of our approach. The evaluation demonstrates the enormous performance gain obtained by translating SPARQL queries to Gremlin traversals, especially for complex queries. Furthermore, Gremlinator has obtained clearance by the Apache Tinkerpop development team and is planned to be integrated as a plugin in the coming versions.

---

6 We have provided all the groovy scripts used for creating composite indices in the Github repository pointed earlier.

30 Indexing scheme in Openlink Virtuoso ([http://docs.openlinksw.com/virtuoso/rdfperfstrfscheme/](http://docs.openlinksw.com/virtuoso/rdfperfstrfscheme/))
As future work, we aim to – (i) provide theoretical proofs that the translation is sound and complete which requires substantial foundational work upfront, a part of which is our on-going effort on formalizing Gremlin pattern matching traversals [33]; (ii) extend our current work by enabling support for SPARQL 1.1 featureset, such as Property Paths, Difference of two graph patterns, regex in restrictions (i.e. FILTERs); (iii) integrate Gremlinator within frameworks such as LITMUS [31, 32], to enable automatic execution of SPARQL queries over property graphs for benchmarking diverse RDF and Graph DMSs.

ACKNOWLEDGEMENTS

This work is supported by the funding received from EU-H2020 WDAqua ITN (GA. 642795). We would like to thank Dr. Marko Rodriguez and Mr. Daniel Kuppitz, of the Apache TinkerPop project, for their support and quality insights in developing Gremlinator.

REFERENCES

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2016. Foundations of Modern Graph Query Languages. CoRR abs/1610.06264 (2016). http://arxiv.org/abs/1610.06264
[2] Renzo Angles and Claudio Gutierrez. 2008. The expressive power of SPARQL. In International Semantic Web Conference. Springer, 114–129.
[3] Renzo Angles and Claudio Gutierrez. 2016. The Multiset Semantics of SPARQL Patterns. In The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part 1 20–36. https://doi.org/10.1007/978-3-319-46523-4_2
[4] Christian Bizer and Andreas Schultz. 2009. RDF and Graph DMSs. Of SPARQL queries over property graphs for benchmarking diverse RDF and Graph DMSs.

[13] Steve Harris, Nick Lamb, and Nigel Shadbolt. 2009. 4store: The design and implementation of a clustered RDF store. In 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009), 94–109.

[14] Steve Harris, Andy Seaborne, and Eric Prud’hommeaux. 2013. SPARQL 1.1 query language. W3C recommendation 21, 10 (2013).
[15] Jürgen Hölsch and Michael Grossniklaus. 2016. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ).
[16] Apache Tinker Pop Home. 2016. Apache tinkerpop home. Web Page (2016).

[17] Christian Krause, Daniel Johannsen, Radwan Deeb, et al. 2016. An SQL-based query language and engine for graph pattern matching. In International Conference on Graph Transformation. Springer, 153–169.
[18] Norbert Martinez-Bazan, Sergio Gomez-Villamor, and Francesc Escaleg-Claveras. 2011. DEX: A high-performance graph database management system. In Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on. IEEE, 124–127.

[19] Galbor Szarnyas József Marton. 2017. Formalizing open/cypher Graph Queries in Relational Algebra. Published online on FTSGR archive (2017). http://docs.inf.mit.bme.hu/ingraph/
[20] Vinh Nguyen, Jyoti Leeka, Olivier Bodenreider, et al. 2016. A Formal Graph Model for RDF and Its Implementation. CoRR abs/1606.00480 (2016). http://arxiv.org/abs/1606.00480
[21] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2006. Semantics and Complexity of SPARQL. In International semantic web conference. Springer, 30–43.

[22] Freddy Priyatna, Oscar Corcho, and Juan Sequeira. 2014. Formalisation and experiences of R2ML-based SPARQL to SQL query translation using Morph. In Proceedings of the 23rd international conference on World wide web. ACM, 479–490.
[23] Eric Prud, Andy Seaborne, et al. 2006. SPARQL query language for RDF. Citeseer Online Archive (2006).
[24] Jyothsna Rachapalli, Vaibhav Khadilkar, Murat Kantarcıoglu, and Bhavani Thuraisingham. 2011. RETRO: A Framework for Semantics Preserving SQL-to-SPARQL Translation. The University of Texas at Dallas 900 (2011), 7500–3021.

[25] Juan L. Reutter. 2015. The Gremlin graph traversal machine and language. Invited talk. In Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015, 1–10. https://doi.org/10.1145/2815072.2815077
[26] Marko A. Rodriguez. 2017. Towards an Integrated Graph Algebra for Graph Pattern Matching with Gremlin. In Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I 8, 3 (2017), 35–42.
[27] Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, and Z Meral Ozsoyoglu. 2005. A complete translation from SPARQL to efficient SQL. In Proceedings of the 2009 International Database Engineering & Applications Symposium. ACM, 31–42.

[28] Marko Rodriguez-Muro and Martin Rezk. 2015. Efficient SPARQL-to-SQL with R2RML mappings. Web Semantics: Science, Services and Agents on the World Wide Web 33 (2015), 141–169.
[29] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In Proceedings of the 13th International Conference on Database Theory. ACM, 4–33.

[30] Mariano Rodriguez-Muro and Martin Rezk. 2015. Efficient SPARQL-to-SQL with R2RML mappings. In Proceedings of the 13th International Conference on Database Theory. ACM, 128–131. https://doi.org/10.1109/ICDEW.2011.5767613
[31] Marko A. Rodriguez-Muro and Martin Rezk. 2015. Efficient SPARQL-to-SQL with R2RML mappings. In Proceedings of the 13th International Conference on Database Theory. ACM, 128–131. https://doi.org/10.1109/ICDEW.2011.5767613
[32] Marko A. Rodriguez and Peter Neubauer. 2011. The Graph Traversal Pattern. In Graph Data Management: Techniques and Applications. IGI Global, 29–46. https://doi.org/10.4018/978-1-61520-053-8.ch002
[33] Marko A. Rodriguez and Peter Neubauer. 2011. A path algebra for multi-relational graphs. In Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011. 128–131. https://doi.org/10.1109/ICDEW.2011.5767613

[34] Mariano Rodriguez-Muro and Martin Rezk. 2015. Efficient SPARQL-to-SQL with R2RML mappings. In Proceedings of the 13th International Conference on Database Theory. ACM, 128–131. https://doi.org/10.1109/ICDEW.2011.5767613
[35] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In Proceedings of the 13th International Conference on Database Theory. ACM, 4–33.

[36] Harsh Thakkar. 2017. Towards an Open Extensible Framework for Empirical Benchmarking of Data Management Solutions: LITMUS. In The Semantic Web - 14th International Conference, ESWC 2017, Portoroz, Slovenia, May 28 - June 1, 2017, Proceedings, Part II 256–266. https://doi.org/10.1007/978-3-319-58451-5_20
[37] Harsh Thakkar, Yashwant Keswani, Mohnish Dubey, Jens Lehmann, and Sören Auer. 2017. Trying Not to Die Benchmarking–Orchestrating RDF and Graph Data Management Solution Benchmarks Using LITMUS. (2017).

[38] Harsh Thakkar, Dharmen Punjami, and Sören Auer. 2017. Towards an Integrated Graph Algebra for Graph Pattern Matching with Gremlin. In Proceedings of the 28th International Conference,DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I Springer, 81–91.
[39] F. Zemke. 2006. Converting sparql to sql. Technical Report. Technical Report, October 2006.