Two for One – Querying Property Graph Databases using SPARQL via Gremlinator

Harsh Thakkar
University of Bonn
Bonn, Germany
thakkar@cs.uni-bonn.de

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

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

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

ABSTRACT

In the past decade Knowledge graphs have become very popular and frequently rely on the Resource Description Framework (RDF) or Property Graphs (PG) as their data models. However, the query languages for these two data models – SPARQL for RDF and the PG traversal language Gremlin – are lacking basic interoperability. In this demonstration paper, we present **Gremlinator**, the first translator from SPARQL – the W3C standardized language for RDF – to Gremlin – a popular property graph traversal language. Gremlinator translates SPARQL queries to Gremlin path traversals for executing graph pattern matching queries over graph databases. This allows a user, who is well versed in SPARQL, to access and query a wide variety of Graph databases avoiding the steep learning curve for adapting to a new Graph Query Language (GQL). Gremlin is a graph computing system-agnostic traversal language (covering both OLTP graph databases and OLAP graph processors), making it a desirable choice for supporting interoperability for querying Graph databases. Gremlinator is planned to be released as an Apache TinkerPop plugin in the upcoming releases.

KEYWORDS

Property Graph, SPARQL, Gremlin, Graph Traversal, Gremlinator

ACM Reference format:
Harsh Thakkar, Dharmen Punjani, Jens Lehmann, and Sören Auer. 2018. Two for One – Querying Property Graph Databases using SPARQL via Gremlinator. In *Proceedings of 1st Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10–15, 2018 (GRADES-NDA'18),* 5 pages.
https://doi.org/10.1145/3210259.3210271

1 INTRODUCTION

Knowledge graphs model the real world in terms of entities and relations between them. They became popular as they are an intuitive and simple data model, which allows to execute many types of queries efficiently and can serve as a foundation for a range of Artificial Intelligence applications. The Resource Description Framework (RDF) and Property Graphs (PGs) are popular languages for knowledge graphs. For RDF, the SPARQL query language was standardized by W3C, whereas for PGs several languages are frequently used, including Gremlin [10].

PGs and RDF have evolved from different origins and still have largely disjoint user communities. RDF is part of the Semantic Web initiative with a focus on expressive data modeling as well as data publication and linking. PGs originate from the database community with a focus on efficient execution of graph traversals.

With *Gremlinator*, we build a bridge between both communities and research the *interoperability* of RDF and PG query languages [15]. Moreover, we allow combining the best of both worlds: Powerful modeling capabilities as well as data publication and interlinking methods combined with efficient graph traversal execution. In particular, Gremlinator has the following advantages: (1) Existing SPARQL-based applications can switch to Property graphs in a non-intrusive way. (2) It provides the foundation for a hybrid use of RDF triple stores and Property graph store – a system could detect which one is more efficient for answering a particular query [5] and redirect the query accordingly. 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. Rather than performing expensive joins, property graph databases use micro indices to perform traversals. (3) Users familiar with the W3C standardized SPARQL query language do not need to learn another query language.

Contributions. Overall, we make the following contributions:

- A novel approach for mapping SPARQL queries to Gremlin pattern matching traversals, Gremlinator, which is the first work bridging the RDF-PG *query interoperability* gap at a broader scale, to the best of our knowledge.
- An openly available implementation for executing SPARQL queries over a plethora of Property graph stores such as *Neo4J*, *Sparksee*, *OrientDB*, etc. using the *Apache TinkerPop* framework.

Organization. The remainder of the article is organized as follows: Section (2) summarizes the related work. Section (3) sheds light on the importance of Gremlin, briefly discusses the Gremlinator approach and its limitations. Section (4) presents the demonstration details, experimental observations and the value Gremlinator

GRADES-NDA'18 , June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5695-4/18/06.
https://doi.org/10.1145/3210259.3210271

will cater to its users. Finally, Section (5) concludes the article and describes the future work.

2 RELATED WORK

We now present a brief summary of related work with regard to approaches that support the translation and execution of formal query languages, aiming the interoperability issue.

SPARQL → SQL: A substantial amount of work has been done for translating queries from SPARQL to SQL, such as – *Ontop* [3], *R2RML* [11], Elliot et al. [6], Chebotko et al. [4], Zemke et al. [17], Priyanka et al. [8]. *Ontop* [3], one of the most popular system, exposes relational databases as virtual RDF graphs by linking the terms (classes and properties) in the ontology to the data sources through mappings, which can then be queried using SPARQL.

SQL → SPARQL: *RETRO* [9] presents a formal semantics preserving the translation from SQL to SPARQL. It follows a schema and query mapping approach rather than to transform the data physically. The 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 → CYPHER: CYPHER[1] is the de facto graph query language of the *Neo4j*[2] graph database. There has been no work yet to use SQL on top of CYPHER. However, there are some examples[3] that show the equivalent CYPHER queries for certain SQL queries.

3 GREMLINATOR APPROACH

Next, we discuss why we choose Gremlin as a Property graph query language, briefly describe the Gremlinator approach and its limitations.

3.1 Why Gremlin Traversal Language?

Gremlin is a system-agnostic query language developed by Apache TinkerPop[4]. It supports both – pattern matching (declarative) and graph traversal (imperative) style of querying over Property graphs.

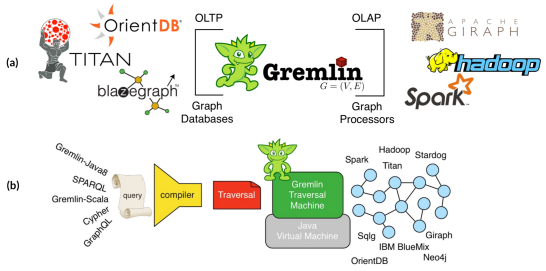

Figure 1: The Gremlin Traversal Language and Machine[4].

Gremlin is more general than, e.g. CYPHER, as it provides in addition to a query language a common execution platform for supporting any graph computing system (including both OLTP and OLAP graph processors), for addressing the querying interoperability issue (cf. Figure 1 (a)). Together, Apache TinkerPop framework and Gremlin, are a language and a virtual machine, which make it possible to design another traversal language that compiles to the Gremlin traversal machine (analogous to how Scala compiles to the JVM), cf. Figure 1 (b). Gremlin provides the declarative (SPARQL style) pattern matching querying construct via the `.match()`-step.

For brevity, we abstain from dwelling on the formal definitions and semantics of Gremlin and SPARQL, rather point the interested reader to the literature [7, 10, 12, 16]. Furthermore, one may also refer to [15], where the complete SPARQL to Gremlin translation approach is discussed in detail.

3.2 Gremlinator Pipeline

We now present the architectural overview of Gremlinator in Figure 2 and discuss each of the four steps of its 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, each basic graph pattern (BGP) is visited, mapping them to the corresponding Gremlin single step traversals (SSTs). An SST in Gremlin is an atomic traversal step (ψ_s). We describe this in [15] in detail.

Step 3. Thereafter, depending on the operator precedence obtained from the AST of the parsed SPARQL query, each of the corresponding SPARQL keywords is mapped to their corresponding instruction steps from the Gremlin instruction library. Thereafter a final conjunctive traversal (Ψ) is generated appending the SSTs and instruction steps. This can be perceived analogous to the SPARQL query language, wherein a set of BGPs form a single complex graph pattern (CGP).

Step 4. This final conjunctive traversal (Ψ) is used to generate bytecode[5] which can be used on multiple language and platform variants of the Apache TinkerPop Gremlin family.

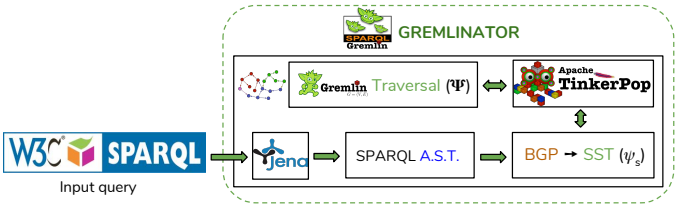

Figure 2: The Gremlinator pipeline architecture.

For a better conceptualization, we present an illustration of the SPARQL → Gremlin translation using a sample query in Figure 3.

3.3 Queries

For demonstrating Gremlinator, we provide a set of 30 pre-defined SPARQL queries for reference, for each dataset, covering 10 different SPARQL query features (i.e. three queries per feature with a combination of various modifiers) as shown in Table 1. These

[1]CYPHER Query Language (https://neo4j.com/developer/cypher-query-language/)
[2]Neo4j (https://neo4j.com/)
[3]SQL to CYPHER (https://neo4j.com/developer/guide-sql-to-cypher/)
[4]Gremlin: Apache TinkerPop's graph traversal language and machine (https://tinkerpop.apache.org/)

[5]Bytecode is simply serialized representation of a traversal, i.e. a list of ordered instructions where an instruction is a string operator and a (flattened) array of arguments.

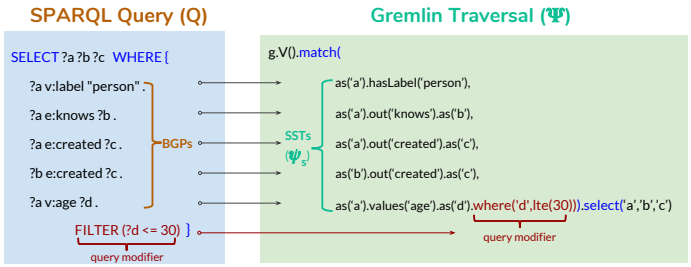

Figure 3: An illustrative example demonstrating the translation of a sample SPARQL query (Q) to its corresponding Gremlin traversal (Ψ) in GREMLINATOR.

Table 1: Query feature and description

Query Id.	Feature	Description
C1-C3	CGPs	Queries with mixed number of BGPs
F1-F3	FILTER	CGPs with a combination of ≥1 FILTER constraints
L1-L3	LIMIT+OFFSET	CGPs with a combination of LIMIT + OFFSET constraints
G1-G3	GROUP BY	CGPs with GROUP BY feature
Gc1-Gc3	GROUP COUNT	CGPs with GROUP BY + COUNT
O1-O3	ORDER BY	CGPs with ORDER BY feature
U1-U3	UNION	CGPs with UNION feature
Op1-Op3	OPTIONAL	CGPs with OPTIONAL BGPs
M1-M3	MIX	CGPs with a combination of all above features
S1-S3	STAR	CGPs forming a STAR shape execution plan (≥10 BGPs)

features were selected after a systematic study of SPARQL query semantics [1, 7, 12] and from *BSBM* [2] explore use cases[6] and *Watdiv Query templates*[7]. Furthermore, we encourage the end user to write and execute custom SPARQL queries for both the datasets, for further exploration.

3.4 Limitations

GREMLINATOR is an on-going effort for achieving seamless translation of SPARQL queries to Gremlin traversals. The current version of GREMLINATOR supports the SPARQL 1.0 SELECT queries with the following exception: GREMLINATOR does not support variables for the property predicate, i.e. the predicate `(p)` in a graph pattern `{s p o .}` has to be defined or known for the traversal to be generated. This is because traversing a graph is not possible without knowing the precise traversal operation to the destination (vertex or edge) from the source (vertex or edge).

4 DEMONSTRATION DETAILS

We next report observations from the experimental evaluation of our approach – GREMLINATOR, followed by the specifics of the demonstration setup, and value GREMLINATOR will cater to its users.

Experimental evaluation. To test the applicability and validity of our approach, we conducted an empirical evaluation by comparing the results obtained after – (1) executing the SPARQL queries against three state-of-the-art RDF triplestores (i.e. Virtuoso[8] [v7.2.4], JenaTDB[9] [v3.2], and 4Store[10]) [v1.1.5], and (2) executing the translated Gremlin traversals (using GREMLINATOR) against three state-of-the-art graph stores (Neo4J[11] [v1.9.6], Sparksee[12] [v5.1] and TinkerGraph[13] [v3.2.3]), using the two famous Berlin SPARQL Benchmark (BSBM) [2] and Northwind[14] datasets respectively. These systems have been experimented with in vanilla state. The Northwind dataset consists of 33,003 triples, 55 properties in the RDF version and 3,209 nodes, 6,177 edges, 55 properties in the Property graph version. Whereas, the BSBM data consists of 1,000,313 triples, 40 properties in the RDF version and 92,757 nodes, 238,308 edges, 40 properties in the Property graph version. Furthermore, we also asked three Gremlin experts to manually write the corresponding Gremlin traversals for each SPARQL query for conducting a two-fold result validation.

Experimental observations. On comparing the results returned by each SPARQL query and its translated Gremlin traversal, we observe uniformity (i.e. the results are equal). Thus, implying that the translation from SPARQL to Gremlin did not alter the meaning of the input query (i.e. the semantics of the input SPARQL query are preserved). On analyzing the performances of queries with respect to RDF triplestores vs Graph stores (we report the results of a subset of BSBM queries in Figure 5, Appendix A), we observe that –

- For C1-3 (CGPs), O1-3 (order by) and M1-3 (mixed) queries (cf. Table 1), there is no clear winner reported between SPARQL and Gremlin.
- For F1-3 (filter), G1-3 (group by) and U1-3 (union) queries (cf. Table 1), the Gremlin traversals are moderately faster (1.5-3x) than the corresponding SPARQL queries.
- For Gc1-3 (group count), L1-3 (limit+offset) and S1-3 (*star-shaped* queries with ≥10 triples) queries (cf. Table 1), the Gremlin traversals outperform corresponding SPARQL queries by an order of two magnitudes. This is because Graph stores benefit from the neighborhood in a graph and avoid performing expensive join operations as compared to RDF stores.

GREMLINATOR demonstration. For the demonstration of GREMLINATOR, we have prepared – **(a)** an online screencast[15] **(b)** a web application, see Figure 4)[16] **(c)** a desktop application of GREMLINATOR (standalone .jar bundle) which requires Java 1.8 JRE installed on the corresponding host machine, downloadable from the web demo website. The source code of Gremlinator is made publicly available here[17].

The system work-flow for all the above mentioned GREMLINATOR versions is alike, wherein – **(i)** the user selects a dataset (Northwind or BSBM) from the corresponding drop-down menu; **(ii)** the user selects a query (one of the ten SPARQL query features) from the corresponding drop-down menu; **(iii)** the user executes the query; **(iv)** system returns the selected SPARQL query, translated Gremlin traversal, its result and traversal profile after the traversal execution; **(v)** the user can also edit/write custom SPARQL queries and execute using the integrated query editor at will.

[6] BSBM Explore Use Cases (https://goo.gl/y1ObNN)
[7] Watdiv Query Features (http://dsg.uwaterloo.ca/watdiv/basic-testing.shtml)
[8] Openlink Virtuoso (https://virtuoso.openlinksw.com/)
[9] Apache Jena TDB (https://jena.apache.org/documentation/tdb/index.html)
[10] 4Store (https://github.com/4store/4store)
[11] Neo4J (https://neo4j.com/)
[12] Sparksee – formerly DEX (http://sparsity-technologies.com/#sparksee)
[13] TinkerGraph (http://tinkerpop.apache.org/docs/current/reference/#tinkergraph-gremlin)
[14] Northwind Database (https://northwinddatabase.codeplex.com/)
[15] Gremlinator Demo Screencast – https://youtu.be/Z0ETx2IBamw
[16] Gremlinator Web Demo – http://gremlinator.iai.uni-bonn.de:8080/Demo and http://195.201.31.31:8080/Demo/
[17] https://github.com/LITMUS-Benchmark-Suite/sparql-to-gremlin

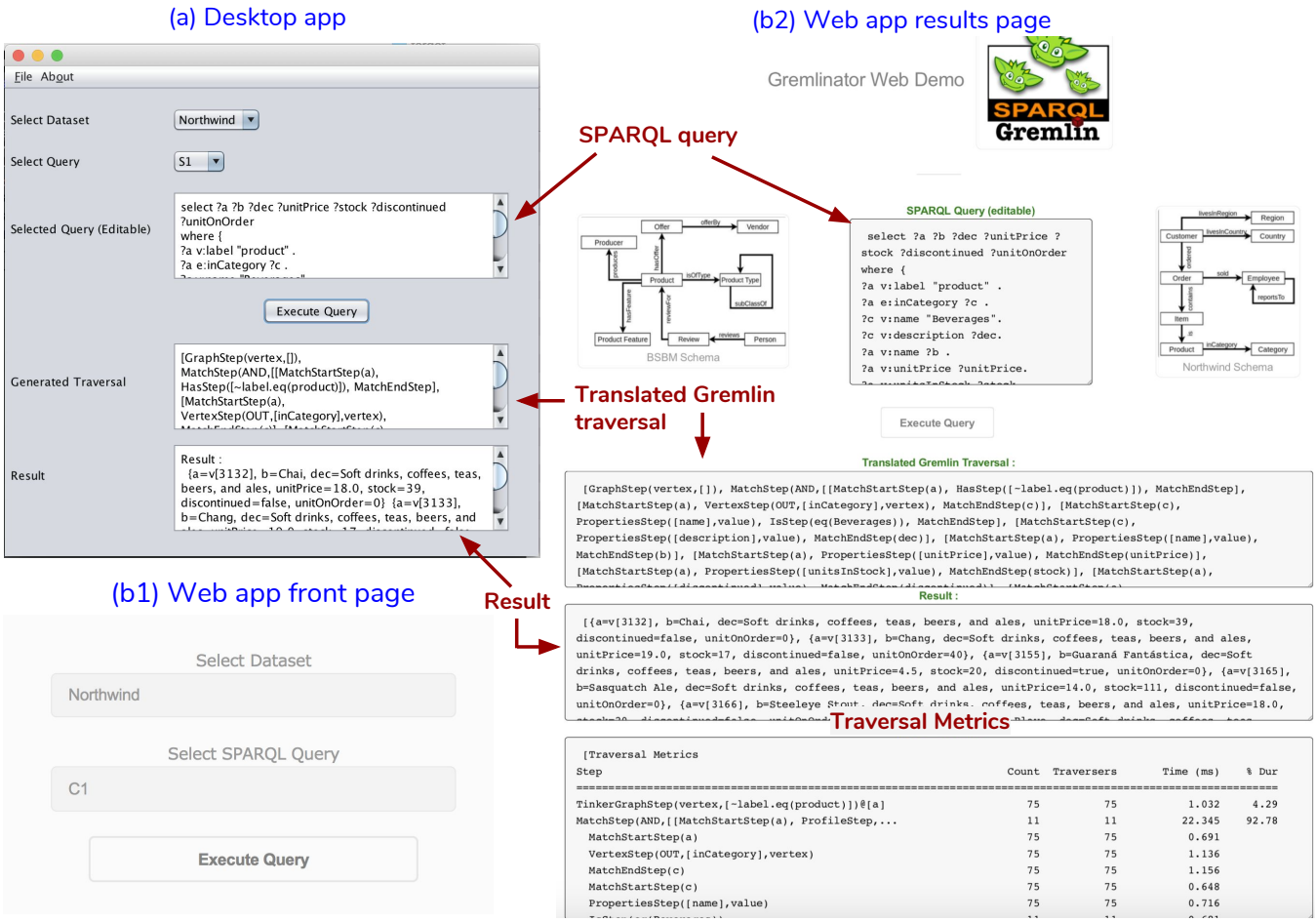

Figure 4: GREMLINATOR approach demonstration snapshots– (a) Desktop application (b1) Web application front page (b2) Web application results page.

Live demo. We will present the live demonstration of GREMLINATOR using a pre-configured laptop with all the resources including the SPARQL queries and datasets. In order to demonstrate the correctness of our approach, we will provide a custom Docker-based OpenLink Virtuoso SPARQL endpoint, pre-loaded with the datasets, for a one-to-one query result comparison (for interested visitors).

Value. GREMLINATOR will serve as a user friendly medium to – (i) execute SPARQL queries over Property graphs bridging the query interoperability gap; (ii) conduct performance analysis of query results, comparisons of SPARQL vs. Gremlin traversal operations using frameworks such as *LITMUS* [13, 14] ; and (iii) enable querying a spectrum of graph databases via SPARQL 1.0 query fragment, leveraging the advantages of the Apache TinkerPop framework.

5 CONCLUSION & FUTURE WORK

In this paper, we demonstrated GREMLINATOR, a novel approach for supporting the execution of SPARQL queries on Property graphs using Gremlin traversals. It will serve as an important medium to bridge the *query interoperability* gap between the RDF and Graph database communities covering both OLTP and OLAP graph systems. GREMLINATOR has obtained clearance by the Apache Tinkerpop development team and is currently in production phase to be released as a plugin during TinkerPop's next framework cycle. Furthermore, GREMLINATOR is freely available under the Apache 2.0 license for public use.

As *future work*, we are working on – (1) supporting translation of variables in predicate position (our current limitation, cf. Section 3.4), and (2) supporting translation of SPARQL 1.1 query features such as Property Paths, in the next release.

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.

APPENDIX A SPARQL VS GREMLIN

Figure 5 presents the performance comparison of selected SPARQL queries vs the translated Gremlin traversals on three popular RDF and Graph stores.

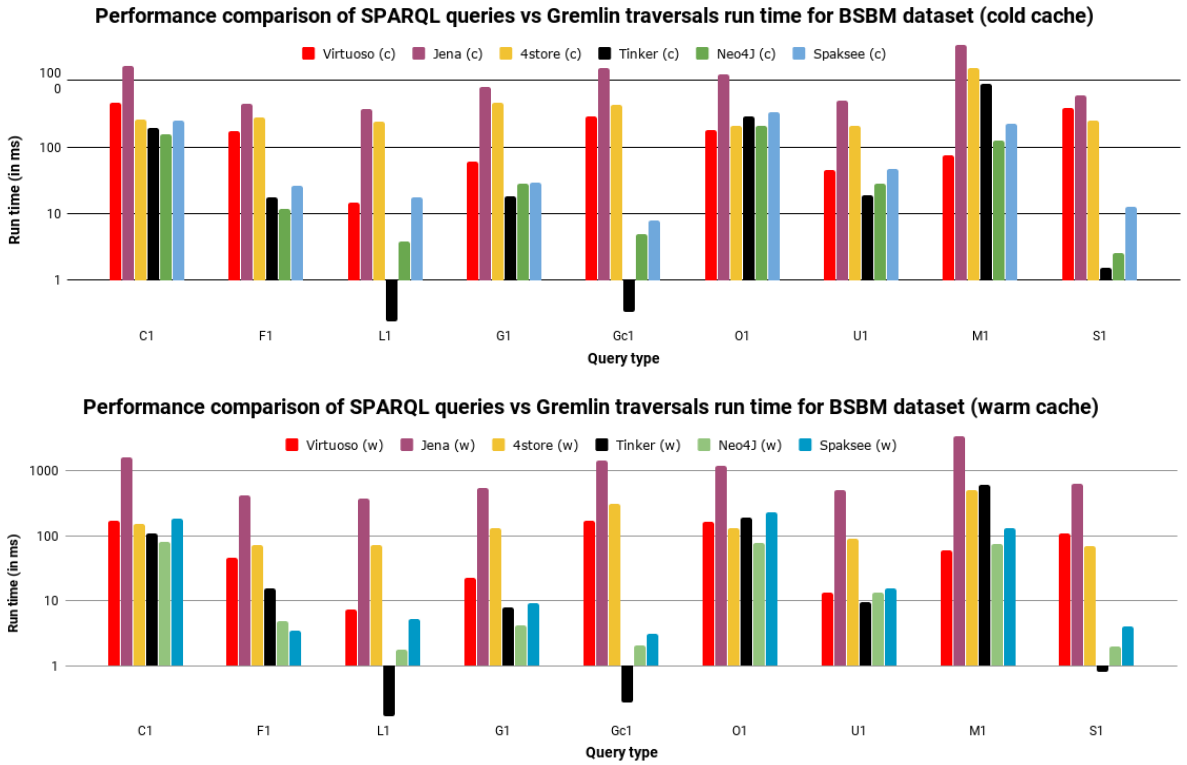

Figure 5: Performance comparison of SPARQL queries vs Gremlin (pattern matching) traversals for BSBM dataset (in milliseconds (ms), log-scale) with respect to RDF and Graph stores in two cache settings. The inverted bars denote <1 ms run-time.

REFERENCES

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, et al. 2016. Foundations of Modern Graph Query Languages. *CoRR* abs/1610.06264 (2016).

[2] Christian Bizer and Andreas Schultz. 2009. The berlin sparql benchmark. (2009).

[3] Diego Calvanese, Benjamin Cogrel, et al. 2017. Ontop: Answering SPARQL queries over relational databases. *Semantic Web* 8, 3 (2017), 471–487.

[4] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. 2009. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering* 68, 10 (2009), 973–1000.

[5] Souripriya Das, Jagannathan Srinivasan, Matthew Perry, et al. 2014. A Tale of Two Graphs: Property Graphs as RDF in Oracle.. In *EDBT*. 762–773.

[6] Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, et al. 2009. A complete translation from SPARQL into efficient SQL. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*. ACM, 31–42.

[7] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2006. Semantics and Complexity of SPARQL. In *International semantic web conference*. Springer, 30–43.

[8] Freddy Priyatna, Oscar Corcho, and Juan Sequeda. 2014. Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph. In *Proceedings of the 23rd international conference on World wide web*. ACM, 479–490.

[9] Jyothsna Rachapalli, Vaibhav Khadilkar, Murat Kantarcioglu, et al. 2011. RETRO: A Framework for Semantics Preserving SQL-to-SPARQL Translation. *The University of Texas at Dallas* 800 (2011), 75080–3021.

[10] Marko A. Rodriguez. 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.

[11] Mariano 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.

[12] 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.

[13] 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, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part II*. 256–266.

[14] 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. In *Proceedings of the 13th International Conference on Semantic Systems, SEMANTICS 2017, Amsterdam, The Netherlands, September 11-14, 2017*. 120–127.

[15] Harsh Thakkar, Dharmen Punjani, Yashwant Keswani, et al. 2018. A Stitch in Time Saves Nine – SPARQL querying of Property Graphs using Gremlin Traversals. *CoRR* abs/1801.02911 (2018).

[16] Harsh Thakkar, Dharmen Punjani, Maria-Esther Vidal, et al. 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.

[17] F Zemke. 2006. *Converting sparql to sql*. Technical Report. Technical Report, October 2006.