Proceedings of the Second International Workshop on Contextualized Knowledge Graphs (CKG 2019)
https://wiki.foodmedy.com/index.php?title=CKG2019

Singleton Property Graph: Adding A Semantic Web Abstraction Layer to Graph Databases

Vinh Nguyen[1], Hong Yung Yip[2], Harsh Thakkar[3], Qingliang Li[1], Evan Bolton[1], Olivier Bodenreider[1]

[1] National Library of Medicine, National Institute of Health, Maryland, USA
[2] University of South Carolina, USA
[3] University of Bonn, Germany

Abstract. Property graph databases provide efficient implementations of graph traversal operations, while Semantic Web technologies provide expressive symbolic representation, querying, and reasoning tasks. Despite the differences between the goals of the two data models, they do share similar graph characteristics.
In this paper, we attempt to combine the benefits of each model into a single graph abstraction layer called Singleton Property Graph (SPG). The SPG layer sits on top of the RDF and simulates the property graph model. We describe the SPG model and its queries, which are Semantic Web-compliant, to be executed inside property graph databases such as TinkerPop. We have tested the prototype and evaluated the experiments with the two datasets BKR and PubChem.

1 Introduction

Although property graphs and RDF are the most popular graph models supported by several graph databases, a single database engine implementing both graph models and their query languages remains to be developed. Graph databases such as AllegroGraph [1], OrientDB [6], and GraphDB [3] implement RDF graphs with the SPARQL query language. Graph databases such as Neo4J [5], Apache TinkerPop [7], and JanusGraph [4] support the property graphs with their own native query languages, e.g., Apache TinkerPop Gremlin [13], PGQL [18], or Cypher. Graph databases such as Amazon Neptune [2] support both graph models, but only one model can be active for a database. In practice, we do not have a single data model that natively support both query languages.

Due to the similarity in the graph characteristics between the property graph and the RDF graph, a common graph model simulating both graph models is feasible, and it can combine the advantages of both worlds, graph databases and Semantic Web. The simulation enables the RDF datasets and their SPARQL queries to be loaded and executed in a property graph. This common graph model will provide the capability to run Semantic Web tasks on top of a property graph database and hence, provide the bridge to connect the two worlds.

In this paper, we propose such a common graph model. Here we use the example from Figure 1 as the motivating example for demonstrating our graph model throughout the paper.

1.1 Motivating Example

A Property Graph (PG) is a directed labeled graph with a set of nodes and a set of edges in which every edge is unique and connects an ordered pair of nodes. A node represents an entity, and an edge represents a relationship between two entities. Each node or edge has properties associated with it in the form of key-value pairs. Figure 1 shows an example of a property graph taken from the Apache TinkerPop Gremlin documentation [7]. This graph contains six nodes numbered 1-6 and six edges numbered 7-12. Indeed, every node or edge has an identifier with the key `id` and a label with the key `label` in the form of key-value pairs. For example, the node 1 actually has `id: 1` and `label: person`.

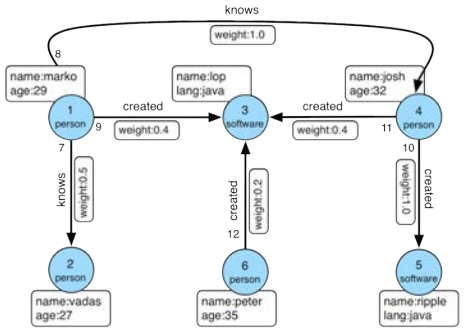

Fig. 1: A sample property graph.

Each blue node represents an entity of type `person` or `software`. A `person` node has two property keys: `name` and `age`. A `software` node has two property keys: `name` and `lang`. Each edge represents one unique relationship `knows` between two `person` entities or one unique relationship `created` between one `person` entity and one `software` entity.

Next, we will present our approach to representing a property graph model and its graph characteristics using RDF.

1.2 Our approach

Compared to the RDF graph model, the distinct characteristics of the property graph model described above are: 1) the edges have their own properties just like the nodes, and 2) every edge or node has a unique identifier. In the running example, the relationship `created` has a property key `weight` showing the contribution of each `person` to the creation of the `software`. The nodes have identifiers 1-6 and the edges have identifiers 7 -12.

We observed that this property model shares distinct characteristics with the singleton property (SP) model [11]. Specifically, while the PG model has a unique identifier for each edge, the SP model also has a unique identifier for each singleton property. Furthermore, while the PG model can have key-value properties for each edge, each singleton property can also be associated with additional metadata triples. Therefore, the similarities between the singleton properties of the SP model and the edges of the PG model may provide the foundation for developing a common data model between them. Here we show how edge number 9 in Figure 1 can be represented in the SP model with URIs created by concatenating the `label` and the `id` of each node as follows:

T_1 :	person#1	created#9	software#3 .
T_2 :	created#9	singletonPropertyOf	created .
T_3 :	created#9	weight	0.4 .

Although the SP model can represent the PG edges intuitively as shown above, its SPARQL query pattern `?sub ?sp ?obj .` (TP1) and `?sp singletonPropertyOf ?p .` (TP2) cannot be used to efficiently traverse this PG model. The singleton properties are unknown in most cases and are represented as variables in this SP query pattern. Because the singleton properties are usually unknown, if they are used to query the edges of the PG model, the PG traversal algorithm's performance may suffer severely because of the all-variable triple pattern `?sub ?sp ?obj .` [15,16].

Furthermore, a singleton property can be associated with a metadata value which turns out to be another entity or node. For example, in the SP patterns with `?sp derives_from PMID_1 .` (TP3) and `PMID_1 type Article .` (TP4), the singleton property `?sp` is associated with the metadata value `PMID_1` (in TP3), and this metadata value is also an entity of `Article` (in TP4). This feature makes the SP model more expressive, but unfortunately it is not supported in the PG model. A PG edge can only take the property value from a data type; it does not accept another entity node like the `PMID_1`. As a result, the PG model cannot support the join between the edge's property values and the nodes to simulate the join between the singleton property's metadata value `PMID_1` (in TP3) and the subject `PMID_1` (in TP4).

Therefore, to develop a common graph model for both RDF and PG models and their query languages, we identify three requirements: (R1) consider the intrinsic similarities between the singleton properties and the PG edges, (R2) resolve the potential degraded performance caused by the SP all-variable query pattern (in TP1) applied to the PG whole-graph traversals, and (R3) enable support for the singleton property's additional metadata values as entity nodes (in TP3 and TP4).

1.3 Our contribution

In this paper, we propose the SPG, a common graph model that meets the three requirements analyzed above. Our contribution for the SPG model includes:

- a graph model as abstraction graph layer on top of the RDF singleton property that can simulate the two distinct characteristics of the PG model,
- a graph query pattern that can express the PG traversals to the key-value properties of the nodes and edges, a SPARQL-compliant querying mechanism that can be executed in PG databases, and
- an implementation of this SPG model for two use cases, BKR and PubChem. Two SPG models with their sets of SPG queries generated from the BKR and PubChem inputs are loaded and evaluated in the PG databases.

The rest of the paper is organized as follows. Section 2 describes our SPG model. Section 3 describes the SPG queries and the SPARQL-compliant querying

mechanism with two use cases from the BKR and PubChem datasets. Section 4 demonstrates the feasibility of our implementation for representing and querying the SPG model in the PG databases such as Apache TinkerPop and Neo4j. We provide the related work in Section 5 and conclude with Section 6.

2 Singleton Property Graph Model

Here we explain how the SPG model can be constructed to be compatible with both the RDF and PG models and to meet the three requirements analyzed in Section 1.2.

Given the motivating example from the property graph in Figure 1 , the SP triples T_1, T_2, and T_3 annotate the semantics of the edge property using the SP model. As this annotation is straightforward, Requirement R1 can be met easily with the adoption of the SP model as the foundation for the new common model SPG.

Here we address the Requirements R2 and R3 for the new SPG model.

Mapping PG Edges and Singleton Properties to SPG Property Nodes. We observe that the two issues discussed in Requirements R2 and R3 only occur when the PG edges and the singleton properties are mapped into the edges of a basic graph. In other words, mapping the SP and the PG edges into the edges of a graph is the cause of the two issues.

If we do not map the PG edges and SPs into the edges of a graph, indeed, we are left with another choice, which is to map them to the nodes of that graph. We have explored this choice in our prior work [12]. This choice is irregular because we are used to the idea that properties are equivalent to edges or links connecting the nodes. However, here we need to justify the nature of these PG edges and SPs. On the other hand, we also investigate this case to verify if mapping the PG edges and SPs to nodes will resolve the two issues.

First, comparing the edges and the nodes in a PG, we observe that both of them share the same characteristic that both of them can carry their own properties. However, the edges carry one extra connectivity characteristic that the nodes do not. In the SP triple, the subject/object and the singleton property also share the same characteristic that all of them can be asserted in any triple. The singleton property itself can also carry the unique connection between the subject and the object. Therefore, from this point of view, we believe that the PG edges and SPs do carry the characteristics of both nodes and edges of a graph, and it is reasonable to map them to a special type of nodes which we refer to as property nodes.

Second, if the mapping is to the nodes, then we have all three disconnected nodes. Requirement R3 is satisfied because PG nodes can be connected to other nodes via edges by the design of the PG model. Here we show how Requirement R2 with all-variable SP query pattern can be address indirectly.

For the three disconnected nodes, we create the first edge with `id: e1` and `label: in` connecting the first and the second nodes, and the second edge with `id: e2` and `label: out` connecting the second node and the third node as shown

in Figure 2. The second node is the property node, and it carries the properties from the original PE edge. If the second node is mapped from the singleton property, then its `id` has the `UUID` of the SP, and its `label` has the value from the generic property. In either case, the property node and the two edges `e1` and `e2` always have a label. When the query for the SP pattern is formed, no variable is needed for the predicate, and that resolves Requirement R2 in the SP all-variable query pattern. Section 3 will discuss this issue in more detail. Therefore, mapping the PG edges and SPs into property nodes satisfies the two remaining Requirements R2 and R3.

As a consequence, the resulting graph meets the three requirements for a common graph model. This resulting graph is called the SPG.

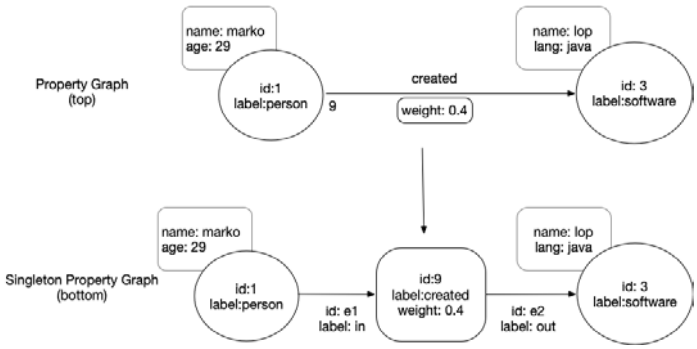

Fig. 2: The original Property Graph sample (top) and the corresponding SPG subgraph (bottom).

3 Loading and Querying SPG Model in Property Graphs

The SPG model described previously is compliant with the RDF representation, and the SPG queries can be expressed in SPARQL. However, here we focus on the implementation of the SPG model and the execution of SPG queries in property graph databases.

We start this section by showing how the SPG model is implemented in the two datasets, PubChem and BKR. We then explain how the SPG queries are constructed and executed.

3.1 Similarity Scores in the PubChem

We collected the data generated by PubChem 3-D similarity algorithm [4], measuring two similar compounds using 3-D Shape and Color Tanimoto scores [9].

[4] `ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound_3D/similar_conformers/`

This repository contains 16995 zipped files of the total size 798 GB. We generated a small portion of this PubChem 3D similarity scores by filtering the files with all rows that have both ST (shape) and CT (color) scores greater than or equal to 90.

Given a pair of compounds `CUI_1` and `CUI_2`, we represent the similarity scores between them as the has_ST_score and the has_CT_score. We created a singleton property `has_sim_score` between the two compounds and associate with it the two meta scores. We loaded the PubChem 3-D similarity scores into two models, M0 and M1 datasets. The difference between the PubChem-M0 and PubChem-M1 datasets is that the PubChem-M0 maps the SPs to edges while PubChem-M1 maps the SPs to property nodes as shown in Figure 3.

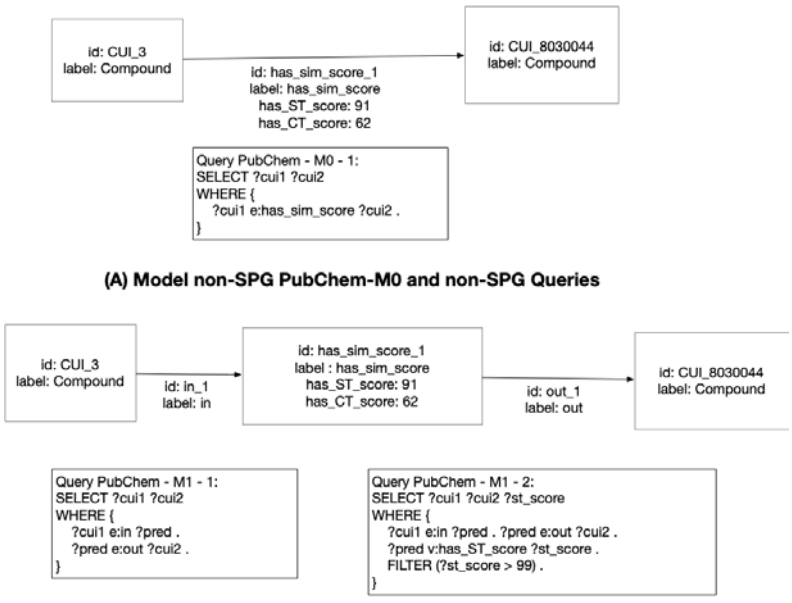

Fig. 3: The PubChem 3-D similarity scores datasets represented in PubChem-M0 and PubChem-M1 data models.

We provide the PubChem-M0 to show the limitation of the SPARQL queries if not using our SPG model. The SPARQL query in this model cannot access the key-value of the edges as we pointed out in Requirement R2.

3.2 Triple Provenance in the BKR

BKR is a biomedical knowledge repository containing over 30 million semantic predications extracted from PubMed abstracts and the Unified Medical Lan-

guage System (UMLS) [11,14]. We collect the original BKR dataset from [11]. It represents the semantic predications using the SP model in NTriple format.

Given a semantic predication (C0007028, PART_OF, C0026969) extracted from the PubMed abstract PUBMED_99992, we represent it in the form of singleton property as follows.

C0007028 PART_OF#1 C0026969 .
PART_OF#1 singletonPropertyOf PART_OF .
PART_OF#1 derives_from PUBMED_99992 .

We transformed this SP dataset into the SPG representation using two models, BKR-M1 and BKR-M2 as shown in Figure 4. The difference between the two models is that in the BKR-M1, we map the singleton properties to a set of property nodes, and the source of the semantic predication is represented as a key-value pair of the property node. Meanwhile, in the BKR-M2, we map the source of the semantic predication to another node and provide additional information about that node, such as the publication date. This BKR-M2 model demonstrates the support for Requirement R3 from Section 1.2.

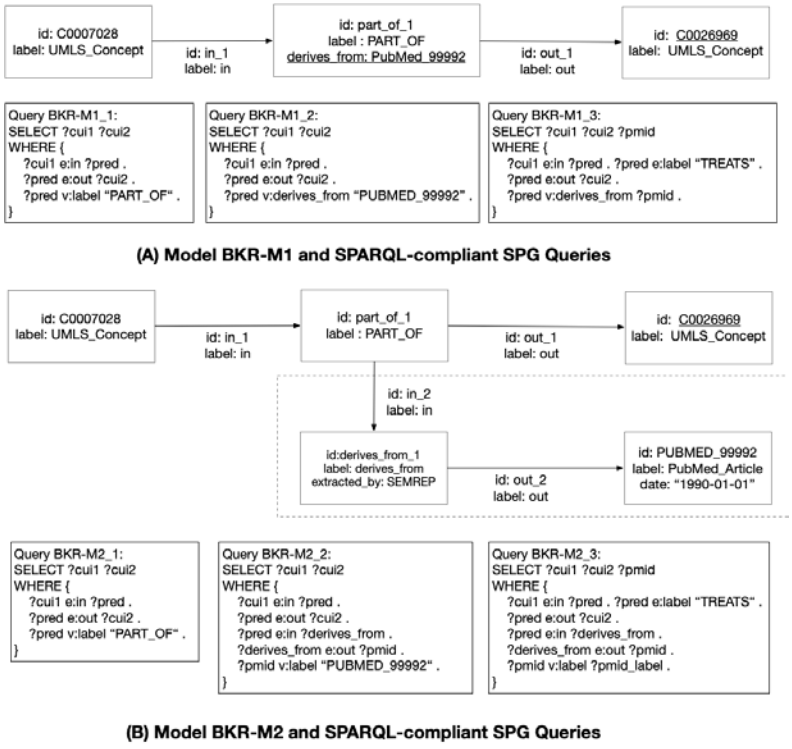

Fig. 4: The BKR dataset loaded into BKR-M1 and BKR-M2 models and their corresponding SPG queries.

3.3 Querying SPG Model in Property Graphs

We loaded the two PubChem and BKR datasets to the Neo4J database using the SPG's M1 and M2 models as shown in Figure 3 and Figure 4, respectively. These models can be queried using SPARQL-compliant SPG queries associated with each model. The SPG queries are executed by using the Sparql-gremlin plugin [15,16] to translate a SPARQL 1.0 query into a Gremlin query that is supported by property graph databases like TinkerPop or Neo4J. This plugin predefines a set of SPARQL 1.0 query patterns for traversing the PG and accessing the key-value properties of a node. The predicates in these query patterns have two parts, a prefix `e:` or `v:` following by a key. The prefix `e:` is for traversing to the edges having the matching key and the prefix `v:` is for retrieving the value for the key from the same node.

SPARQL-compliant SPG query

For every SPG node triple $t = (v_i, v_e, v_j)$, $f_{SPG}(v_i, v_e, v_j) = (e_i, e_o)$, the node triple is connected by the pair of (`in`, `out`) edges. The subject node v_i is connected to the property node v_e by the `label:in` edge e_i, and the property node v_e is connected to the object node v_j by the `label:out` edge e_o. Therefore, the common SPG pattern for accessing any SPG node triple will be in this form: `?sub1 e:in ?pred1 . ?pred1 e:out ?obj1 .` (P1)

For accessing the value from the key `key_m` of any node in the SPG node triple, we use the following pattern: `?sub1 v:key_m ?val`. (P2)

These SPG query patterns P1 and P2 can be used in conjunction with each other to traverse and retrieve the key-value pairs of any node in the SPG model. All SPG queries from Figure 3 and Figure 4 use these two patterns.

For example, considering the queries BKR-M2-1 and BKR-M2-2, two node triple patterns P1 and one key-value pattern P2 are used to construct these queries. The queries BKR-M1-1 and BKR-M1-2 share the same combination of one node triple pattern P1 and one key-value pattern P2.

For the PubChem-M1, the query PubChem-M1-1 uses only one node triple pattern P1, and the query PubChem-M1-2 uses one node triple pattern P1 and one key-value pattern P2. Meanwhile, the PubChem-M0 is not a SPG model. It cannot support the access to the key-value properties of the M0's edges.

Next, we report the use of the data models generated here for the experimental evaluation.

4 Experiments

In this section we report the experiments that demonstrate the proof-of-concept implementation of SPG models serving as a Semantic Web abstraction layer on property graphs with queryable Semantic Web-compliant SPARQL queries. The experiments can be grouped into three main categories: (i) importing the SPG models into property graph database, (ii) comparing the property graph loading and reading times, and (iii) clocking the query execution time and evaluating the query results. In these experiments, we used the Biomedical Knowledge Repository (BKR) and PubChem datasets described in Section 3.

4.1 Experimental Setup

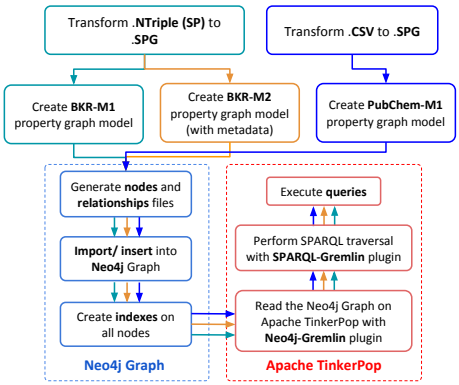

Fig. 5: Experiment Flowchart

The experiments were performed on a server running on CentOS 7 with 126 GB of RAM and 3.84 TB of Samsung PM983 NVMe storage. We used Neo4J version 3.2.3 as the property graph database and Apache TinkerPop Gremlin version 3.4.0 as the graph compute and query engine installed with two plugins: Neo4J-Gremlin version 3.4.1 and SPARQL-Gremlin version 3.4.1. The Neo4J-Gremlin plugin is used to provide the ability to query and traverse a Neo4J graph using Gremlin, whereas the SPARQL-Gremlin is a compiler (also known as Gremlinator) that transforms SPARQL queries into Gremlin traversals.

It uses the Apache Jena SPARQL processor ARQ, which provides access to a syntax tree of a SPARQL query. Together, they provide the necessary interoperability interface between the Semantic Web (SPARQL) and Property Graph (Neo4J) crossover. Next, we describe the experiment processes (Figure 5).

4.2 Importing SPG Models into Neo4J

The BKR SP dataset [11] consists of 33M NTriples with a file size of 17.6 GB. This dataset was first parsed to the SPG representation (.SPG). Two instances of property graph models (BKR-M1 and BKR-M2) were then created from the SPG file. Similarly, the PubChem-M1 model was also generated from its SPG file parsed from its initial CSV files. A set of nodes and relationships files was generated for each of the three models to facilitate the batch insert process into Neo4J using the Neo4J-import tool. The two main criteria that determine the insert performance are the size of the available heap memory and the page cache. A large enough heap space is beneficial to sustain concurrent operations, whereas a large page cache ensures most of the graph data from disk is cached in memory to help avoid costly disk access during import. The Neo4J server is configured to allow a max heap and page cache size of 32 GB respectively, which are more than adequate given the total number of nodes and relationships of our largest model, PubChem-M1. Based on these configurations, we timed the insert speed with and without creating indices. Table 1 shows the corresponding tasks with results for BKR-M1, BKR-M2, and PubChem-M1.

While the SPG representation preserves the same number of triples, it excels with a file size of 6.6 GB, an overall 62.5% reduction in storage space compared to the SP model. The BKR-M1 implementation has a total of 36M nodes, 67M relationships, and 69M properties, whereas the BKR-M2 has a total of 73M nodes,

134M relationships, and 110M properties. The PubChem-M1 implementation has a total of 368M nodes, 682M relationships, and 1.05B properties (Table 1).

Table 1: **BKR-M1 vs BKR-M2 vs PubChem-M1**

Model	BKR-M1	BKR-M2	PubChem-M1
Input file size	17.6 GB (NTriple)		10 GB (CSV)
SPG file size	6.6 GB		58 GB
Number of Unique Nodes	36M	73M	368M
Number of Relationships	67M	134M	682M
Number of Properties	69M	110M	1.05B
Generate nodes and relationships files	3 min 52 sec	7 min 16 sec	32 min 20 sec
Insert into Neo4J (with indices)	2 min 11 sec	3 min 54 sec	19 min 17 sec
Insert into Neo4J (without indices)	3 hours	-	-
Final database size	5.5 GB	11 GB	55 GB

Discussion. Given that the final BKR-M2 database is twice the size of BKR-M1, the difference between the insert performances is relatively marginal. Two plausible reasons are the NVMe drives set-up that read 3 GB/s and write at 1 GB/s, and the optimizations (heap memory and page cache) configured on Neo4J server.

4.3 Loading and Traversing Neo4J Property Graph on Apache TinkerPop Gremlin

Apache TinkerPop Gremlin is used in conjunction with the Neo4J-Gremlin and SPARQL-Gremlin plugins to provide the functionality of running SPARQL queries over a property graph database, since Neo4J does not natively support SPARQL query language. The Neo4J-Gremlin plugin is used to provide API-level access to the BKR-M1, BKR-M2, and PubChem-M1 databases created in Section 4.2. The plugin is configured with the same configurations as the Neo4J server to ensure consistency. Finally, the time taken to read and load the graph into Apache TinkerPop were **4.35**, **9.46**, and **9.83** seconds for BKR-M1, BKR-M2, and PubChem-M1, respectively.

Discussion. Using the Neo4J-Gremlin plugin eliminates the additional overhead to export the Neo4J graph as GraphML format and subsequently be loaded into Apache TinkerPop. Our experiment of loading BKR-M1 as GraphML format into Apache TinkerPop took hours due to the plausible need to reconstruct the nodes and relationships as well as their properties from scratch. Nonetheless, the Neo4J-Gremlin provided acceptable reading and loading times, especially for

PubChem-M1, with a relatively high number of nodes and relationships compared to BKR-M1 and BKR-M2.

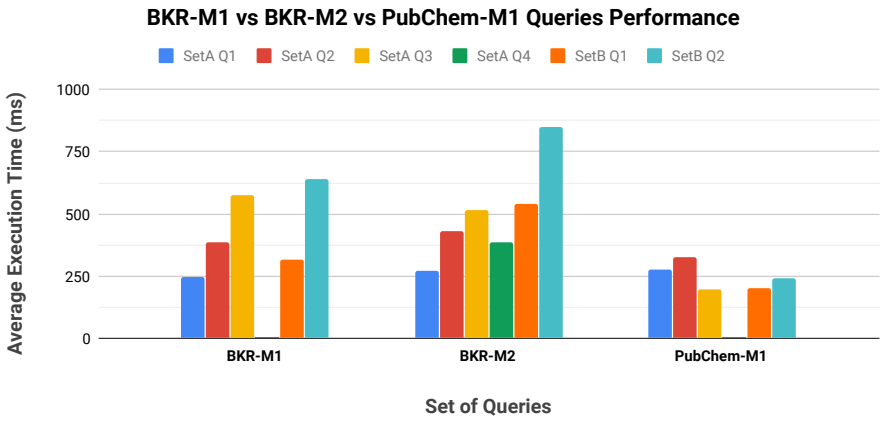

Fig. 6: **Average Query Performance in msec.**

4.4 Queries Execution

We created a set of SPARQL-compliant queries (set A and set B) derived from the [11] that are supported by the current SPARQL-Gremlin version 3.4.1 and performed the queries on both BKR-M1 and BKR-M2. The queries consisted of the basic patterns and simple functions like COUNT, FILTER, GROUP BY, and LIMIT from SPARQL 1.0. The SPARQL queries were performed using the SPARQL-Gremlin plugin loaded on Apache TinkerPop Gremlin. Every query was run for 10 repetitions and started with a cold cache (by restarting the gremlin instance) to provide a fair comparison between short and long queries without the influence of a warm cache from prior queries. The evaluations were quantified by the corresponding average execution time per query using the native Gremlin clock() API and the returned results (Figure 6).

Discussion. Given that the number of nodes and relationships in BKR-M2 are twice the size of BKR-M1, the difference between the query performances were relatively comparable. This suggested BKR-M2 was equally efficient, but at a higher information (metadata) gain. SetA Q4 was not applicable to BKR-M1 and PubChem-M1 as it involved metadata query which BKR-M1 lacked.

4.5 Overall Discussion

Our experiments show that the SPG approach gives a decent performance in terms of number of triples, query size, and query execution time. The results support our proof-of-concept that the SPG queries are indeed SPARQL-compliant and can be used as a Semantic Web abstraction layer on top of graph databases.

Such a layer enables the support of the expressiveness and logic of semantic technologies while providing an efficient implementation of graph traversal operations.

5 Related Work

In this paper, we use the singleton property model proposed by Nguyen et al. [11] as the foundational model for representing our SPG model. However, as the SP all-variable query pattern may cause entire-graph traversals when applied in a graph database, we develop a new querying mechanism for our model. In other words, our work enhances the SP model in that our new querying mechanism provides an alternative implementation for the SP queries.

We also use the sparql-gremlin package [15,16] for translating the SPARQL queries to the Gremlin language supported by property graph databases. However, this package does not accept any SPARQL query other than SPARQL 1.0 with predefined patterns for the SPARQL queries to traverse the PG and accessing the key-value properties. It does not support the all-variable queries, and it cannot retrieve the property of the edges. Our work differs from this package in that we define a new data model and use the structures defined by this package to enable the execution of the new queries for our data model. Furthermore, our model can help the Sparql-gremlin to overcome its limitation such as all-variable queries (in case of SP queries) and the retrieval of the edge property.

For the RDF and PG models, several approaches have been proposed for formalizing the PG model and transforming it to other data models such as RDF, and RDF*. Hartig et al. [10] formalizes the PGs and RDF* data models and defines the transformations between them. Our work is different since we are proposing a new graph model that is compatible with both PG and RDF models, and hence, no transformation is needed. [17] proposes YARS as a Cypher-based RDF serialization that is compatible with the PG databases supporting Cypher. Our work is implemented with Gremlin and we use it to translate and execute the SPARQL-compliant SPG queries in PG databases. Das et al. [8] simulates the property graph model using the RDF named graphs and sub-properties for the annotation of the triple metadata. Our work uses the SP model for the simulation.

6 Conclusion

We have presented the SPG model and its implementation showing that this graph model can be the common graph model for both RDF and PG models. Our model and its implementation can also be reused for other datasets and applications. This model is compatible with Semantic Web standards, with the representation in the form of RDF triples and the queries expressed in SPARQL.

Acknowledgement This research was supported in part by the Intramural Research Program of the National Institutes of Health (NIH), National Library of Medicine (NLM). This research was also supported in part by an appointment to

the National Library of Medicine Research Participation Program. This program is administered by the Oak Ridge Institute for Science and Education through an inter-agency agreement between the U.S. Department of Energy and the National Library of Medicine. We are also thankful for the help from Usha Lokala.

References

1. Allegrograph. `https://franz.com/agraph/allegrograph/`. Accessed: 2019-04-10.
2. Amazonneptune. `https://aws.amazon.com/neptune/`. Accessed: 2019-04-10.
3. Graphdb. `http://graphdb.ontotext.com/`. Accessed: 2019-04-10.
4. Janusgraph. `https://janusgraph.org/`. Accessed: 2019-04-10.
5. Neo4j. `https://www.neo4j.com/`. Accessed: 2019-04-10.
6. Orientdb. `https://orientdb.com/`. Accessed: 2019-04-10.
7. Tinkerpop. `http://tinkerpop.apache.org/`. Accessed: 2019-04-10.
8. S. Das, J. Srinivasan, M. Perry, E. I. Chong, and J. Banerjee. A tale of two graphs: Property graphs as rdf in oracle. In *EDBT*, pages 762–773, 2014.
9. G. Fu, C. Batchelor, M. Dumontier, J. Hastings, E. Willighagen, and E. Bolton. Pubchemrdf: towards the semantic annotation of pubchem compound and substance databases. *Journal of cheminformatics*, 7(1):34, 2015.
10. O. Hartig. Reconciliation of rdf* and property graphs. *arXiv preprint arXiv:1409.3288*, 2014.
11. V. Nguyen, O. Bodenreider, and A. Sheth. Don't like rdf reification?: Making statements about statements using singleton property. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 759–770, 2014.
12. V. Nguyen, J. Leeka, O. Bodenreider, and A. Sheth. A formal graph model for rdf and its implementation. *arXiv preprint arXiv:1606.00480*, 2016.
13. M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM, 2015.
14. S. S. Sahoo, V. Nguyen, O. Bodenreider, P. Parikh, T. Minning, and A. P. Sheth. A unified framework for managing provenance information in translational research. *BMC bioinformatics*, 12(1):461, 2011.
15. H. Thakkar, D. Punjani, Y. Keswani, J. Lehmann, and S. Auer. A stitch in time saves nine–sparql querying of property graphs using gremlin traversals. *arXiv preprint arXiv:1801.02911*, 2018.
16. H. Thakkar, D. Punjani, J. Lehmann, and S. Auer. Two for one: querying property graph databases using sparql via g remlinator. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, page 12. ACM, 2018.
17. D. Tomaszuk. Rdf data in property graph model. In *Research Conference on Metadata and Semantics Research*, pages 104–115. Springer, 2016.
18. O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2016.