A Tale of Two Graphs: Property Graphs as RDF in Oracle

Souripriya Das Jagannathan Srinivasan Matthew Perry

Eugene Inseok Chong Jayanta Banerjee

Oracle
One Oracle Drive, Nashua, NH 03062
firstname.lastname@oracle.com

ABSTRACT

Graph Databases are gaining popularity, owing to pervasiveness of graph data in social networks, physical sciences, networking, and web applications. A majority of these databases are based on the property graph model, which is characterized as key/value-based, directed, and multi-relational. In this paper, we consider the problem of supporting property graphs as RDF in Oracle Database. We introduce a property graph to RDF transformation scheme. The main challenge lies in representing the key/value properties of property graph edges in RDF. We propose three models: 1) named graph based, 2) subproperty based, and 3) (extended) reification based, all of which can be supported with RDF capabilities in Oracle Database. These models are evaluated with respect to ease of SPARQL query formulation, join complexities, skewness in generated RDF data, query performance, and storage overhead. An experimental study with a real-life Twitter social network dataset on Oracle Database 12c demonstrates the feasibility of representing property graphs as RDF and presents a quantitative performance comparison of the proposed models.

Categories and Subject Descriptors

H.2.m [**Database Management**]: Miscellaneous

General Terms

Algorithms, Performance, Design, Experimentation, Theory.

Keywords

Property Graph, RDF, SPARQL, Graph Database, Social Network.

1. INTRODUCTION

Graph Databases are gaining popularity, owing to pervasiveness of graph data in social networks, physical sciences, networking, and web applications. A majority of these databases (such as Neo4j [6], DEX [7], InfiniteGraph [8]) are based on the *property graph* model [2].

In a property graph, each vertex is identified with a unique identifier (unique within the graph). Each (directed) edge, identified with a unique identifier and labeled with a string, connects a source vertex to a destination vertex. A vertex or an edge may also be associated with a collection of key/value properties. Figure 1 shows a sample property graph, which is *key/value-based*, *directed*, and *multi-relational*.

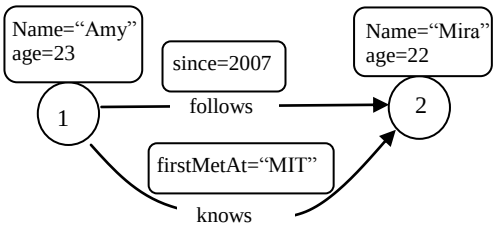

Figure 1. A sample property graph.

Property graph data is typically accessed through the de facto standard Blueprints Java API [3] or some proprietary query language. The query languages over property graphs (such as Cypher [14]) have typically focused on finding paths once the start node, or qualifying start nodes identified with certain key/values are specified. The edges themselves can be traversed by considering the associated key/value pairs. Throughout the paper, we use the words node and vertex interchangeably.

In contrast, RDF [1] provides a way of specifying directed, labeled graphs. Each directed, labeled edge is represented by a triple: <subject, predicate, object> where the predicate is the label for a directed edge from the subject node to the object node. The use of quads, instead of triples, is becoming popular in practice and is included in the new W3C RDF1.1 Recommendation [33]. A quad extends a triple by allowing an optional *named graph* component and is represented as: <subject, predicate, object, graph>. Each of the components of a triple or a quad must be an RDF term. RDF terms can be of three types: Internationalized Resource Identifier (IRI), blank node, or literal. Restrictions on types of RDF terms that can be used in a component position are: 1) subject must be an IRI or a blank node; 2) predicate must be an IRI; 3) object must be an IRI, a blank node, or a literal; 4) graph, if present, must be an IRI or a blank node.

RDF graphs are typically queried using the standard SPARQL query language [9, 16] by specifying a graph pattern, which returns matching subgraphs. Although pattern matching has been the primary focus, the W3C SPARQL 1.1 Recommendation has provided options for querying for paths using the *property path* construct.

The RDF representation has been adopted by several graph databases including RDBMS-based RDF stores (such as Oracle Database Spatial and Graph Option [10], Openlink Virtuoso [11]) and native RDF stores (such as AllegroGraph [12], OntoText OWLIM [13]). Typically, most RDF stores, including Oracle, natively support standard entailment regimes (such as RDFS [4], OWL 2 RL [15]).

On a cursory look, it appears that the property graph model is more general than RDF because key/values can be associated with both vertices and edges in the property graph model. In contrast, RDF

10.5441/002/edbt.2014.82

supports asserting *datatype* and *object* properties[1] for resources (i.e., nodes or vertices), but does not allow directly asserting any properties for a specific edge, that is, for a triple itself. For example, one may conclude that the edge properties `since` and `firstMetAt` shown in the sample property graph of Figure 1 are not expressible in the RDF representation.

We argue, however, that a deeper look would show that RDF is fully capable of representing any property graph. Our specific goal is to enable RDF stores to become an alternate platform for storing and processing property graphs.

The key benefits of supporting property graphs as RDF are:

- RDF stores can benefit from the ability to associate key/value pairs with edges.
- Semantically rich property graph applications can be developed that link with ontologies and domain-specific knowledge bases and exploit native inference capabilities.
- Property graph data can easily be published as RDF linked data on the web.
- RDF stores can serve as backend storage for large property graph datasets. Note that RDF stores have virtually no limit on scale of data and have already exhibited handling of a trillion triples [34], whereas graph databases rely on in-memory graph analysis engines and hence have limits on the graph sizes they can support (for example, Neo4j has a limit of 34 billion edges).
- RDF stores are based on W3C RDF and SPARQL standards, and property graph users can benefit from this standardization.
- Standards based RDF stores offer mature tools and products. In addition, RDBMS backed RDF stores such as Oracle have the advantage of built-in transactional support, which becomes available by default for property graph applications.

The basic idea for supporting property graphs in RDF is to introduce *edge-IRIs* – IRIs to identify individual edges in a property graph. Specifically, we propose three schemes 1) *named graph based*, 2) *subproperty based*, and 3) *(extended) reification* [35] *based*, that use edge-IRIs in different ways for representing property graphs as RDF. All of these schemes can be supported with the capabilities of an RDF store (in our case, Oracle). Furthermore, the proposed representations can be queried using standard SPARQL constructs. That is, the user does not have to learn yet another query language.

Use of edge-IRIs in RDF allows more flexible modeling than property graphs because both datatype and object properties may be asserted for edge-IRIs. In property graphs, key/value properties for edges can only be scalars. RDF, in contrast, allows edges to be associated with object properties as well. That is, in property graph terms, a key/value can link an edge to another vertex. For example, in the property graph of Figure 1, the `knows` edge has a key/value pair `firstMetAt/"MIT"`, where the key's value `"MIT"` is a scalar. An RDF graph model would let the value part be an IRI resource (a vertex), `:MIT`, which universally represents the resource.

Although less expressive than RDF, property graph implementations do allow for a compact representation, since vertex and edge identifiers are local to a graph, and key/values can also be efficiently encoded. A characteristic that is often quoted as a criterion for property graph implementations, or for graph databases in general, is the notion of *index-free adjacency*. That is, every element contains a direct pointer to its adjacent elements and no index lookups are necessary. Such pointers are useful for in-memory graph analysis, which typically is the case for most property graph implementations.

As an alternative, one could consider directly storing graph data in relational tables. However, requiring a user to express graph oriented queries in SQL is cumbersome, so a graph oriented query language is necessary. The SPARQL syntax allows simpler query formulation for RDF data because it was designed with the assumption that the underlying store is a single table (or logical structure) with just four columns (considering quads). Specifically, use of variables or constants in any of the four positions of a triple-pattern, optionally enclosed in a GRAPH clause, implicitly identifies the column being referred to and multiple uses of the same variable specifies equi-join. SQL cannot provide such simple syntax because it was designed for the general case where multiple tables with many columns are being queried. As the number of equi-joins and use of constants increase in a query, the SQL query becomes increasingly complex to specify and difficult to read and understand. To see the relative simplicity of SPARQL queries, consider the following query that involves a 4-way join and uses 5 constants: *find the company that John's uncle works for*.

SPARQL query:

```
PREFIX : <http://x/> .
SELECT ?company WHERE {
?x :name "John"   . ?x :hasFather ?f .
?f :hasBrother ?b . ?b :worksFor ?company}
```

SQL query (against a 3-column triples(sub, pred, obj) table):

```
SELECT t4.obj company
FROM triples t1, triples t2,
     triples t3, triples t4
WHERE t1.sub = t2.sub AND t2.obj = t3.sub AND
      t3.obj =  t4.sub AND
      t1.pred = '<http:/x/name>' AND
      t1.obj = '"John"' AND
      t2.pred = '<http://x/hasFather>' AND
      t3.pred = '<http://x/hasBrother>' AND
      t4.pred = '<http://x/worksFor>';
```

To demonstrate the feasibility of our proposed scheme for implementing property graphs on an RDF store, we evaluate the approach with a real-life Twitter social network dataset [23] on Oracle Database.

The key contributions of the paper are as follows:

- To the best of our knowledge, this is the first proposal to support storing property graphs as standard RDF and querying using standard SPARQL. The entire scheme can be supported with capabilities of an RDF store (e.g., Oracle Database).
- Named graph based, subproperty based, and (extended) reification based schemes which have varying query performance and storage implications
- The SPARQL query formulation and comparison of various schemes
- An experimental evaluation of all the aspects with a real-life Twitter social network dataset

Note that although we show evaluation of our proposal with RDF capabilities of Oracle, the proposed scheme is general enough to be supported on other RDF stores as well.

1.1 Related Work

In the wake of emerging applications such as social networks, and big data analytics, where fast graph traversal is required, graph databases have recently started to gain traction. Many graph

[1] Note that *datatype* and *object* properties differ in that the range of the former is literals, whereas the range of the latter is resources (IRIs and blank nodes).

databases (Neo4j [6], DEX [7], InfiniteGraph [8]) for property graphs [2] have been introduced implementing the Blueprints API [3]. Graph traversal languages such as Cypher [14] and Gremlin [22] have also been implemented.

Graph databases use indexless adjacency access to process graph traversal queries efficiently and are basically main-memory based. Some graph databases claim to be disk-based, but true indexless adjacency requires in-memory based access. There have been some studies to compare the performance of graph databases and RDF databases [20, 21]. While in-memory graph databases achieve impressive performance in graph traversal, especially in reachability queries, RDF databases show acceptable performance and perform better for data types other than string [21].

RDF [1] related products (Oracle Spatial and Graph Option [10], Virtuoso [11], AllegroGraph [12], OWLIM [13]) have appeared much earlier and are quite mature. Unlike its property graph counterparts, RDF has a standard query language SPARQL [9, 16] and has the capability of inference based on standard rules [4, 5, 15]. The SPARQL language is declarative, which enables users to focus on specification and lets the task of optimal execution be left to the query engine.

There have been efforts to incorporate graph traversal capability and support for edge attributes into RDF databases. One such effort is G-SPARQL [19]. The authors try to use special symbols to distinguish graph attributes and path information from triple pattern queries. Our approach differs from their approach as standard SPARQL (that is, without any changes) can be used for querying both paths and attributes allowing seamless merging of triple pattern queries with graph traversal capabilities, including handling of vertex and edge attributes. Our approach achieves the objective by reformulating the SPARQL query to handle property graphs. However, this reformulation is required only if the graph contains edge attributes.

The rest of the paper is organized as follows. Section 2 introduces the key concepts, and provides qualitative evaluation of the proposed schemes. Section 3 describes how a property graph is supported using Oracle's RDF capabilities. Section 4 describes a performance evaluation using a real-life Twitter social network dataset. Section 5 provides further discussion of a couple aspects related to supporting property graphs in RDF, and Section 6 concludes the paper.

2. KEY CONCEPTS

This section presents the key concepts.

2.1 Representing Property Graphs as RDF

A Property Graph associates three pieces of information with an edge (besides the start and end vertices): a unique identifier, a label, and a possibly empty set of key/value pairs. RDF on the other hand associates only a single piece of information with an edge: a label called the predicate. Thus translating an edge in a Property Graph to an equivalent RDF representation requires more than just an edge (or triple) in RDF.

We will consider the Property Graph of Figure 1 (excluding the `knows` edge and associated key/value) to illustrate three different ways of transforming it to an equivalent RDF graph (shown in Figure 2). For each of these translations, we will show a SPARQL query to find *"who follows whom since when?"*

Translation using reification: In order to accommodate the id, label, and the key/value pairs for an edge, reification in RDF can create a new resource `pg:e3` (based on the edge 3) to represent the reified RDF statement "v1 follows v2". `The pg:e3` is the subject of three triples, with predicates being `rdf:subject, rdf:predicate and rdf:object,` with values `pg:v1, rel:follows` and `pg:v2` respectively.

```
SELECT ?xname   ?yname  ?yr WHERE {
  ?r rdf:subject   ?x .
  ?r rdf:predicate rel:follows .
  ?r rdf:object    ?y .
  ?r key:since     ?yr .
  ?x key:name      ?xname .
  ?y key:name      ?yname }
```

Translation using unique RDF properties for edges: Id, label, and key/values for an edge can be modeled by creating a unique RDF property for each edge to represent the edge id, creating an RDF triple with that property as the predicate, associating the key/value pair with that property, and then making the property a subproperty of another property created based on the edge label. In this example, the unique property `pg:e3` (based on edge 3) is created, an RDF triple `<pg:v1, pg:e3, pg:v2>` is created, and `pg:e3` is made a subproperty of the property `rel:follows` (created from edge label `"follows"`), and two resources `pg:v1` and `pg:v2` (corresponding to vertices with id 1 and 2) are the subject and object, respectively.

```
SELECT ?xname ?yname ?yr WHERE {
  ?x  ?p                ?y .
  ?p  rdf:subPropertyOf rel:follows .
  ?p  key:since         ?yr .
  ?x  key:name          ?xname .
  ?y  key:name          ?yname   }
```

Translation using RDF named graphs: This alternative involves the use of quads (as opposed to triples) to create a unique named graph IRI for each edge. Then the label and the key/value properties of the edge are associated with the graph IRI. In this example, the property `rel:follows` (based on the edge label `"follows"`) and the RDF named graph IRI `pg:e3` (based on edge id 3) have been used to create the quad: `<pg:v1, rel:follows, pg:v2, pg:e3>`, and the graph IRI `pg:e3` has been associated with the key/value properties.

Note that the triples representing edge key values have been included in the corresponding named graph to allow for clustering edge key/values with the corresponding edge.

```
SELECT ?xname   ?yname   ?yr WHERE   {
   GRAPH ?g {?x rel:follows   ?y .
             ?g   key:since   ?yr }
    ?x   key:name   ?xname .
    ?y   key:name   ?yname   }
```

For the rest of the paper, we only consider the subproperty based and named graph based approaches as they lead to compact storage and can be queried using simpler SPARQL graph patterns

Discussion. Although, for reification and subproperty-based schemes, `pg:v1 rel:follows pg:v2` can be implicitly derived, we propose to have it explicitly asserted as a triple, thereby allowing traditional SPARQL (e.g., `?x rel:follows ?y`) to be used for querying when no edge key values (such as `key:since`) are referenced.

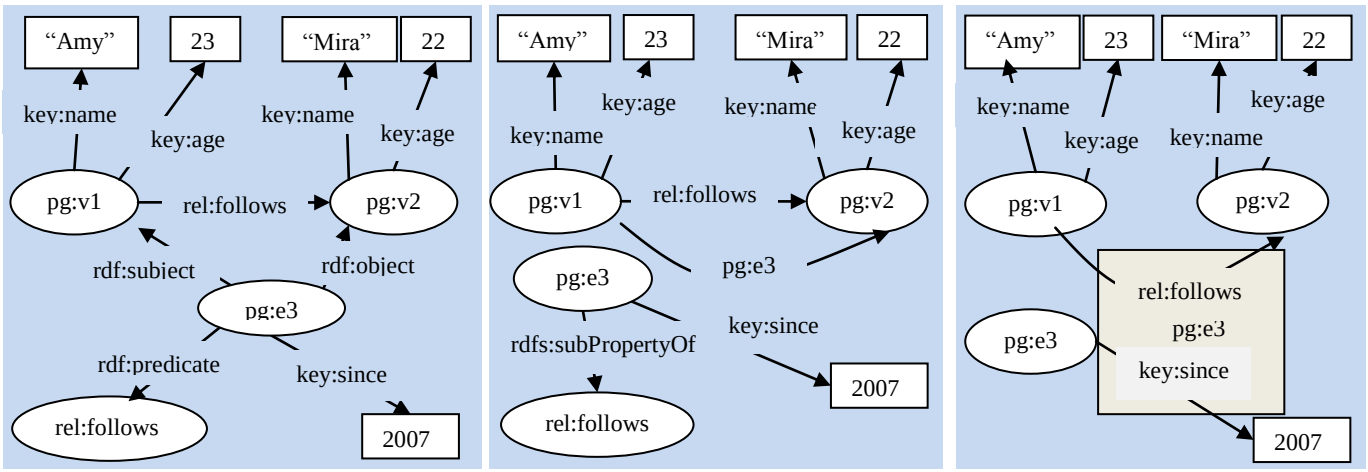

Figure 2. a) (Extended) Reification-based, b) RDF subproperty-based c) Named graph based representation of a property graph

The proposed representation does blur the distinction between topology triples and KVs triples. Thus, for a predicate variable occurring in a SPARQL query, if the context does not provide sufficient information, the `isUri()` and `isLiteral()` built-in functions can be used to distinguish between object and datatype properties (see Section 2.3 for details). The SPARQL 1.1 Update [36] defines an update language for RDF graphs. The simplicity of the RDF model dictates a DELETE and INSERT pattern for updates. The <subject, property, object, graph> quad forms a four part key, so any update basically creates a new quad. In terms of incremental DML operations, the key performance metric that distinguishes the three approaches is time taken to locate existing quads to delete, which is tied to query performance. We will consequently focus on query performance and leave a detailed study of DML performance for future work.

2.2 Transforming Property Graphs to RDF

This section describes the vocabulary used to transform a property graph to RDF. We assume property graph data is available in a representative relational schema consisting of Edges and ObjKVs tables. Figure 3 shows the relational representation of the property graph in Figure 1.

Edges

StartVertex	Edge	Label	EndVertex
1	3	follows	2
1	4	knows	2

ObjKVs

ObjId	Key	Type	Value
1	name	VARCHAR	Amy
1	age	NUMBER	23
...	...	...	...
3	since	NUMBER	2007

Figure 3. A sample property graph in relational format.

The vertices and edges map to RDF resources. For example, `vertex 1` maps to `<http://pg/v1>` and `edge 3` maps to `<http://pg/e3>`. Similarly, labels and keys get mapped to predicate IRIs. For example, label `follows` maps to `<http://pg/r/follows>` and key `age` maps to `<http://pg/k/age>`. No distinction is made between edge and node keys as a key may be common to an edge and a node. The namespace prefixes `rel:` and `key:` denote `<http://pg/r/>` and `<http://pg/k/>`, respectively.

The value component is mapped to an RDF literal by taking the data type into account (e.g., value `23` mapped to `"23"^^<http://www.w3.org/2001/XMLSchema#int>`). Using the URIs and literals generated in this manner, RDF triples or quads, as appropriate, are generated for various schemes.

2.3 Analysis of Various Schemes

This section provides an analysis of the various ways a property graph can be transformed to and modeled as RDF triples or quads and then queried using SPARQL.

Overview of PG-as-RDF models: The three RDF representations for property graphs illustrated in Section 2 differ based on how the *id* and *label* (type of relationship) of an edge in a property graph are modeled in RDF. To characterize them, we use the following notations:

- The form `b-i-r-d` denotes a property graph edge with (unique) id `i` and label `r` connecting source vertex `b` to destination vertex `d`.
- The symbols `s`, `e`, `p`, and `o` are used to denote the IRIs generated from `b`, `i`, `r`, and `d`, respectively (by augmenting them with some prefix and suffix strings).
- The form `e-s-p-o` denotes an RDF quad with `e` as the named graph and `-s-p-o` denotes an RDF triple (i.e., not in a named graph).
- The forms `-n-K-V` and `-e-K-V` denote RDF triples for key/value properties of a property graph vertex or an edge, respectively, where `n`, `e`, and `K` are IRIs generated from the vertex id, edge id, and key in a property graph, respectively, and `V` is an RDF literal corresponding to the value of a key/value property.

Using the above notations, the three PG-as-RDF models can be described as follows (also summarized in Table 1):

- **RF**: The traditional reification based approach (excluding the "rdf:type rdf:Statement" triple) represents `b-i-r-d` using the following three triples: `-e-rdf:subject-s`, `-e-`

rdf:predicate-p, and -e-rdf:object-o. In addition, the triple -s-p-o is included as well to allow posing SPARQL property path queries.

- **NG**: The named graph based approach represents b-i-r-d using a single RDF quad: e-s-p-o. Although not essential for the model, any KVs for the edge are placed in the same named graph e, for clustering of all the triples for the edge.
- **SP**: The subproperty based approach represents b-i-r-d using the triples: -s-e-o and -e-rdfs:subPropertyOf-p. In addition, the (derivable) triple -s-p-o is included as well for the same reasons as in the RF case.

Table 1. RDF representation for three models

PG-as-RDF model	RDF quads/triples for PG element type		
	Topology edge	**EdgeKV**	**NodeKV**
RF	-e-rdf:subject-s -e-rdf:predicate-p -e-rdf:object-o **-s-p-o**	-e-K-V	-n-K-V
NG	e-s-p-o	**e**-e-K-V	-n-K-V
SP	-s-e-o -e-rdfs:subPropertyOf-p **-s-p-o**	-e-K-V	-n-K-V

Special case: If a property graph has a vertex v with no KVs or inbound or outbound edges, then we use the following RDF triple for all the models: -v-rdf:type-rdf:Resource.

Skewness in generated RDF data: RDF datasets generated for the PG-as-RDF models exhibit different characteristics compared to traditional RDF datasets. Relevant cardinalities of a property graph and those for the RDF datasets generated using the different PG-as-RDF models are shown in Table 2.

Table 2. Property graph vs. RDF cardinalities

Property Graph cardinalities			
E edges (E^1 of them with >=1 edge-KVs), *V* vertices, *eKV* edge-KVs, *nKV* node-KVs, *eL* distinct edge-labels (rel. types), *eK* distinct keys for edge-KVs, and *nK* distinct keys for node-KVs			
RDF cardinalities for models	RF	NG	SP
Named Graphs	*0*	***E***	*0*
Obj-prop triples/quads	***4 * E***	***E (quads)***	***3 * E***
Data-prop triples (quads in NG)	*eKV+nKV*		
Distinct sub/obj count	*V+E*	***V+E***^{1}	*V+E*
Distinct obj-properties	*eL+3*	*eL*	***eL+E+1***
Distinct data-properties	*Distinct (eK UNION nK)*		

Since the three PG-as-RDF models differ mainly in how they map a property graph edge into RDF, an important measure to examine is the count of object-property triples or quads. In the NG model, the mapping is straightforward because all four components of a property graph edge are accommodated in a single quad. On the other hand, for the RF and SP models, multiple triples are needed. Thus, the number of object-property triples are 4*E, E (quads), and 3*E for the RF, NG, and SP models, respectively. (As we show later, this difference affects the number of joins in the SPARQL queries for accessing edges and their edge-KVs.) Note that if a property graph edge does not have any edge-KVs, then it is possible to represent it in RDF using just a single -s-p-o triple. We have not accounted for this optimization in the cardinalities shown in the Table 2.

Several aspects of the generated RDF datasets have differences with respect to traditional RDF datasets: 1) In the NG model, the number of distinct named graphs, E, is same as the number of object-property quads. The proportion of one object-property quad per named graph is very low when compared to traditional RDF datasets. 2) In the SP model, the number of distinct object-properties, eL+E+1, is much larger than a typical RDF dataset. Also, the proportion of object-property triples and distinct object-properties, 3*E / (eL + E +1), is less than 3. In contrast, LUBM datasets [25] have only a handful of distinct object properties and those are used for hundreds of millions or billions of triples.

Although, among the three models, the NG model has the lowest storage cost in terms of number of triples/quads and number of distinct values, the NG model has a drawback if one wants to extend or augment the generated RDF data with traditional RDF data or combine data generated from multiple property graphs. Consider an e-s-p-o RDF quad created for a property graph edge. If one wants to keep all the RDF generated from the property graph in a separate named graph g (probably because he/she wants to put the content from multiple property graphs and organize them into different named graphs), then the quads such as e-s-p-o generated for NG have to be inserted into quads g-s-p-o and the SPARQL graph patterns for querying will get complicated. The complication arises from presence of the triple -s-p-o in the named graph e as well as in the named graph g.

SPARQL Query formulation for the PG-as-RDF models: We illustrate the mapping of property graph queries to SPARQL graph patterns using queries shown in Table 3.

Property graph queries may be broadly divided into two categories based on whether or not a query accesses any edge-KVs. Queries that do not access edge-KVs can be expressed in SPARQL the same way regardless of whether we use the RF, NG, or SP scheme. This is due to the presence of -s-p-o triple (in RF and SP) or e-s-p-o quad (in NG) that represents each edge. Query Q1 shows an example of such a query. Queries that need to access edge-KVs, however, will differ for RF, NG, and SP schemes due to differences in how edge-KVs are represented in each scheme. Query Q2 shows an example of such a query.

An aspect of a property graph query that affects SPARQL query formulation, regardless of the use of RF, NG, or SP, is when only the KVs or only the outbound topological edges of a node need to be retrieved, but not both. To select only the intended triples, the use of a FILTER clause to check if the object component of triple is a literal (using isLiteral()) or if it is an IRI (using isIRI()) is needed. Queries Q3 and Q4 show the use of such filters.

Rules for constructing SPARQL graph patterns for property graph queries may be formulated as follows:

1. If a query needs access to an edge but not any of its edge-KVs then the SPARQL graph pattern is quite simple.
   a) If the edge label is bound, then simply use a triple-pattern of the form ?x <label> ?y (see Q1).
   b) If the edge label is a variable (and the query does not want to retrieve node-KVs of the source vertices), then use Filter clause to restrict (see Q4).

2. If an edge-KV is accessed, then for each different PG-as-RDF model we need to use the appropriate set of triple-patterns to access the edge resource first (see Q2).

3. If a node-KV needs to be accessed and
   a) the key is bound, then simply use a triple-pattern of the form ?x <key> ?V (see Q1 and Q3).
   b) the key is a variable, and the query does not want to retrieve outbound topological edges from the vertices, then the use of a Filter clause to impose that exclusion is needed (see Q3).

Table 3. SPARQL graph patterns for property graph queries

PG-as-RDF model	SPARQL query graph pattern for property graph queries
Q1. Get triangles (three edge cycles) of "follows" edges	
All	{?x rel:follows ?y . ?y rel:follows ?z . ?z rel:follows ?x}
Q2. Get vertex pairs and *all KVs of edges* with "follows" label	
RF	{?e rdf:subject ?x; rdf:predicate rel:follows; rdf:object ?y . ?e ?k ?V}
NG	{GRAPH ?e {?x rel:follows ?y . ?e ?k ?V}}
SP	{?x ?e ?y . ?e rdfs:subPropertyOf rel:follows . ?e ?k ?V}
Q3. Get *all KVs of vertices* matching a given KV: name = "Amy"	
All	{?x key:name "Amy" . ?x ?k ?V FILTER isLiteral(?V)}
Q4. Get source and destination vertices of all edges	
All	{?x ?p ?y FILTER isIRI(?y)}

As can be seen in Q2, if an edge is to be accessed along with edge-KVs, then the number of joins is maximum in RF. Specifically, for each such edge, RF requires a 3-way join. Also, the storage cost is the highest for the RF model. For these reasons, in the rest of the paper we omit RF and focus on the NG and the SP models only.

3. SUPPORTING PROPERTY GRAPHS AS RDF IN ORACLE

This section presents the relevant capabilities in Oracle RDF Semantic Graph and their use in supporting the PG-as-RDF models outlined in the last section. Note that although we show evaluation of our proposal with RDF capabilities of Oracle, the proposed scheme is general enough to be supported on other RDF stores as well.

3.1 Oracle RDF capabilities

Currently, Oracle has, among many others, the following RDF capabilities:

- Allows creating one or more semantic models each of which can hold an RDF dataset (containing triples in a default (unnamed) graph and, optionally, quads in named graphs). Individual models or merges of multiple models can be independently queried.
- Supports fast bulk load of RDF data supplied in N-Quads format into a semantic model.
- Supports querying using SPARQL 1.1 directly from Java or from SQL using the SEM_MATCH table function.
- Supports semantic network indexes (among the typical six combinations on triple `s-p-o` or 24 combinations in quad `g-s-p-o`) to improve the performance of SPARQL query processing.
- Supports creation and querying of virtual semantic models defined as a UNION (or UNION ALL) of existing semantic models.

3.2 What we will use

We assume property graph data is available in relational format as described in Section 2.2. The property graph data available in a relational table is converted into RDF with IRIs generated as discussed in Section 2.2 and triples and/or quads formed as discussed in Section 2.3. The resulting RDF data is loaded into a semantic model. This semantic model is then queried using SPARQL to perform property graph queries.

Partitioned Storage for generated RDF: Often SPARQL queries may access only some of the various forms of RDF triples or quads (see Table 1) generated for a PG-as-RDF model. Thus, separating the storage of the triples or quads into different partitions based on the specific form, may lead to better query performance. In Oracle Database, one can create separate semantic models (as partitions in a partitioned table with local indexes) to store these different forms of triples.

For example, one possible configuration could be to create three separate partitions: 1) edge quads or triples partition, 2) node-KV triples partition, and 3) the edge-KV triples (for SP, this would include the `-s-e-o` and `-e-sPO-p` triples as well).

Node attributes can be converted naturally from property graph into RDF for both NG and SP models. However, for edges with edge attributes, NG models do not incur any additional storage overhead (except for their use of the fourth component). SP models need an anchor triple `-e-sPO-p` to associate the edge attributes with the edge. Therefore, it would increase the number of triples proportional to the total number of edges. The storage overhead caused by these additional triples can be mitigated by the use of compression and partitioning schemes.

Topology information, edge properties, and node properties can be kept in separate partitions to maximize compression on prefixes (`:e :sPO`) and (`:e :k`) for edge attributes and prefix (`:s :k`) for vertex attributes. Each partition in the current Oracle RDF store is implemented as a separate model. Therefore, if more than one partition is accessed, a virtual model containing all those partitions is used.

Table 4 shows the query type and forms of triples accessed. By using Oracle's capability of querying individual semantic models (partitions) or a union of one or more semantic models, a user can choose the appropriate RDF dataset for each query.

Table 4. Property graph query types

Query Type	Forms of quads/triples to be queried	
	NG	SP
edge traversal, no edge-KV	e-s-p-o	-s-p-o
edge + edge-KV	e-s-p-o e-e-K-V	-s-e-o -e-sPO-p -e-K-V
Node-KV	-n-K-V	-n-K-V

For example, query Q1 in Table 3 (edge traversal only) may be posed against a single semantic model that holds the `e-s-p-o` (for NG) or the `-s-p-o` triples (for SP). Similarly, for NG, query Q2 (edge + edge-KV) may be posed against the union of semantic models

holding the `e-s-p-o` triples and the `e-e-K-V` triples. For SP, the target dataset resides in a single partition.

Indexing: SPARQL query processing in Oracle Database typically involves accessing only the indexes built on the semantic models; the tables are rarely accessed. Users may create any permutation of the following letters to specify the key for an index that he/she wants to build on a semantic model: S (subject), P (predicate), C (canonical object), G (named graph), and M (semantic model). All of these columns hold numeric identifiers, not lexical values, because they are ID-based. M is included in any indexes that are built to allow identification of the partition of a table that holds the target semantic model.

Although Oracle allows users to create indexes with any of the various permutations (with S, P, C, and G - ignoring M) as key, in practice only a few permutations are necessary. Two indexes are created by default on all the semantic models: (unique) PCSGM and PSCGM. If the RDF data uses named graphs (i.e., contains quads), then to allow access using named graphs one or both of the following two indexes are often necessary: GPCSM and GSPCM. Also, to allow subject-based access, the SPCGM or SCPGM index may be useful as well.

Typically we have three categories of queries as shown in Table 4 (and index use as shown in Table 5):

- Edge traversal without filter on edge attributes: The index PCSGM may be used depending upon the triple pattern and only the topology partition will be accessed (see Q1).
- Edge traversal with filter on edge attributes: The anchor triple `<:e :sPO :p>` needs to be used to get the edge attributes. The index PCSGM may be used, and only the edge KV partition is accessed (see Q2).
- Vertex with attributes filter: No changes are required for filtering vertex attributes. The index PCSGM may be used, and the topology and vertex property partitions are accessed (see Q3)

As seen from Table 5, most of the queries can be satisfied by using one of the five indexes. Note that for a selective pattern, an index range scan is used, and for and unselective filter, a full index scan is used in Oracle Database.

For datasets that could fit in memory, the index blocks get cached in database buffers, and disk access can therefore be completely avoided after the initial load. Also, because indexes are local to a partition, partitioning data into multiple semantic models based on the forms of triples helps achieve better clustering and compression of index data and leads to optimal utilization of the database buffer cache.

Table 5. Property graph query execution using indexes

PG-as-RDF model	**SPARQL query graph pattern for property graph queries and corr. index-based access plans**
Q1. Get triangles (three edge cycles) of "follows" edges	
NG, SP	**?x rel:follows ?y.** 1: [P=rel:follows] PCSGM **?y rel:follows ?z.** 2: [P=rel:follows and S=c1] PSCGM **?z rel:follows ?x** 3: [P=rel:follows and C=s1 and S=c2] PCSGM
Q2. Get vertex pairs and *all KVs of edges* with "follows" label	
NG	**GRAPH ?e {?x rel:follows ?y}** 1: [P=rel:follows] PCSGM **GRAPH ?e {?e ?k ?V}** 2: [G=g1 and S=g1] GSPCM
SP	**?e rdfs:subPropertyOf rel:follows** 1: [P=rdfs:subPropertyOf and C=rel:follows] PCSGM **?x ?e ?y** 2: [P=s1] PCSGM **?e ?k ?V Filter isLiteral(?V)** 3: [S=s1] SCPGM (+filter)
Q3. Get *all KVs of vertices* matching a given KV: name = "Amy"	
NG,SP	**?x key:name "Amy"** 1: [P=key:name and C="Amy"] PCSGM **?x ?k ?V Filter isLiteral(?V)** 2: [S=s1] SCPGM (+filter)

4. Experimental Evaluation

This section describes the experiments conducted to evaluate the performance of various schemes. A Twitter social network dataset, which uses the property graph model, has been converted to RDF using both the SP and NG schemes, and a series of queries against both schemes illustrates the relative performance of each approach.

4.1 Experimental Setup

All experiments were conducted on a Lenovo ThinkPad T430 equipped with a dual-core Intel i5-3320M CPU, 8 GB of RAM and a 120 GB OCZ Vertex2 SSD, and all experiments used Oracle Database 12c configured with the Oracle Spatial and Graph – RDF Semantic Graph option running on 64-bit Oracle Enterprise Linux 6.

The goal of our experimental evaluation is to characterize the performance differences of the SP and NG schemes with respect to query execution time. Note that query execution time is critical for DML operations as well because triples/quads to be updated must be retrieved first. We also want to demonstrate that query execution times with these schemes are fast enough for interactive applications. An evaluation of the general scalability of Oracle RDF Semantic Graph is outside the scope of this paper (see [18] for general Oracle RDF Semantic Graph Performance).

4.2 Property Graph Dataset Characteristics

A Twitter social network dataset [23] (used in the discovery of social circles [24]) was chosen for experiments. It consists of 973 ego networks, where each ego network with ego `a` contains edges of type `b follows c`, which implicitly means `a knows b` and `a knows c`. These form the topological edges. Each node had zero or more features of the form `@keyword` or `#tag`. From these features, the node KVs `n refs @keyword` or `n hasTag #tag` were generated. The edge KVs were generated by taking the intersection of start node KVs with end node KVs, both for edges with `follows` and `knows` labels. For example, for edge `e`: `a follows b`, the `{KVs of e}={KVs of a}∩{KVs of b}`. The generated data characteristics are shown in Table 6. Among 76,245 nodes, 70,097 nodes occur as subjects in the property graph. Also, the edge count is much smaller than the KV count.

Table 6. Twitter dataset characteristics

	Nodes	Edges	Node KVs	Edge KVs
Count	76,245	1,796,085	1,218,763	3,345,982

Figure 4 shows the out-degree and in-degree distribution by count. As expected, the in-degrees are generally higher than out-degrees as the same literal values are often shared between many KVs.

The data characteristics show that it is a highly connected graph, which is a common characteristic of property graphs, and thus serves as a representative dataset for experimentation.

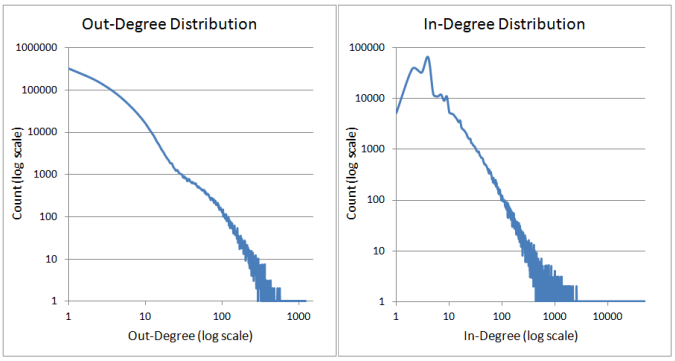

Figure 4. Out-degree and in-degree distribution

4.3 RDF Dataset Characteristics

Table 7 shows the data characteristics for PG-as-RDF model triples. The first row of the table shows the core triples, which are present in both NG and SP models. The SP model has 3,592,170 more triples than NG model due to the addition of `-e-sPO-p` (1,796,085) and `-s-e-o` (1,796,085) triples.

Table 7. Transformed RDF dataset characteristics: triples

	Edges		KVs	
	follows	**knows**	**refs**	**hasTag**
Triples	1,667,885	128,200	3,771,755	792,990
NG	6,360,830			
SP	9,953,000			

Table 8 shows the number of distinct resources of each type. For both models, the number of subject IRI resources increases from 70,097 to 1,019,549 and 1,866,182 respectively due to occurrence of either named graphs (for NG: `e-`***`e`***`-K-V`) or edge IRIs as subjects (for SP: `-`***`e`***`-sPO-p`). The increase is less for NG because it occurs only for edges with at least one or more edge KVs.

Table 8. Transformed RDF dataset characteristics: resources

	NG	SP
Subjects	1,019,549 (70,097 + 949,452)	1,866,182 (70,097 + 1,796,085)
Predicates	4	1,796,090 (4 + 1 + 1,796,085)
Objects	288,392	288,392 + 2
Named Graphs	1,796,085	0

Also, there is an increase in predicates in the SP model as each edge occurs as a predicate IRI (plus one for `rdf:subPropertyOf`). The objects IRI counts are similar. The additional two for SP model corresponds to two properties `rel:knows` and `rel:follows` that occur in the object position for `-e-sPO-`***`p`*** triples.

4.4 Experiments

This section presents a series of experiments that test the performance of 1) node-centric queries, 2) edge-centric queries, 3) aggregate queries and 4) graph traversal queries. These queries are similar to those in [17, 20] with the addition of edge key/value queries. Table 10 shows the queries used in our experiments.

Table 9. Physical storage characteristics

DB Object	Size (MB)	
	NG	**SP**
Triples Table	248	329
Values Table	56	57
PCSGM Index	259	398
PSCGM Index	338	504
GPSCM Index	366	NA
SPCGM Index	358	506
Total	1,625	1,794

Database Configuration: The database was configured with a `pga_aggregate_target` of 2 GB and an `sga_target` of 4 GB. Four semantic network indexes were created: PCSGM, PSCGM, SPCGM, GPSCM. In addition, an `optimizer_dynamic_sampling` level of 6 was used for EQ11a-e, while a level of 2 was used for all other queries. Loading the quads and triples for the NG and SP models took 5min 16 sec and 6 min 01 sec respectively.

Physical Storage Characteristics: The physical characteristics of the stored data using the NG and SP schemes are shown in Table 9. The increased number of rows (triples/quads) required to store the graph in the SP scheme is reflected in the larger size for each database object, but the total size needed is very similar for each scheme because the GPSCM index is not required in the SP scheme.

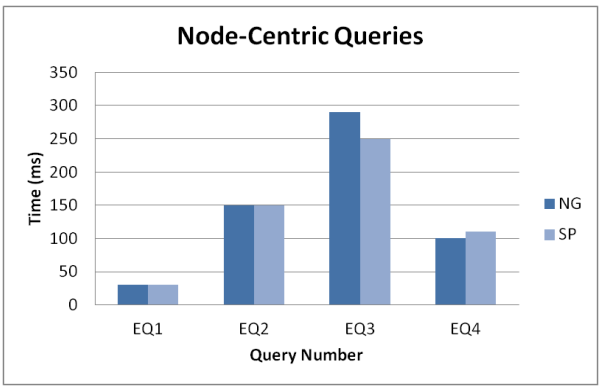

Figure 5. Execution time for node-centric queries.

Methodology: The reported query execution times were obtained by running the queries with SQL*Plus using the `set timing on` option. For each experiment, the queries were run once sequentially to warm up the database buffers, then the queries were run sequentially a second time to obtain the reported times.

Experiment 1 – Node-centric Queries: This experiment tests the performance of EQ1 (find all nodes/edges that have tag "#webseries"), EQ2 (find all nodes that follow nodes with tag "#webseries"), EQ3 (find all 3-hop paths where each node has tag "#webseries"), and EQ4 (find all key/value pairs for nodes/edges with tag "#webseries"). The results of this experiment are shown in in Figure 5. All queries finish within 300 milliseconds, and there

Table 10. Queries for experimentation

Query	Query Text	Number of Results
EQ1	`SELECT ?n` `WHERE { ?n k:hasTag "#webseries" }`	251
EQ2	`SELECT ?nf` `WHERE { ?n k:hasTag "#webseries" . ?nf r:follows ?n }`	1,249

EQ3	`SELECT ?n4`
	`WHERE { ?n k:hasTag ?t . ?n r:follows ?n2 . ?n2 k:hasTag ?t .`
	`?n2 r:follows ?n3 . ?n3 k:hasTag ?t . ?n3 r:follows ?n4 .`
	`?n4 k:hasTag ?t FILTER (?t = "#webseries") }`
		11,440

EQ4	`SELECT ?n ?k ?v` `WHERE { ?n k:hasTag "#webseries" . ?n ?k ?v FILTER (isLiteral(?v)) }`	3,011
EQ5a	`SELECT ?n2` `WHERE { GRAPH ?g1 { ?n r:follows ?n2 . ?g1 k:hasTag "#webseries" } }`	206

EQ5b	`SELECT ?n2`
	`WHERE { ?s ?p ?n2 . ?p rdfs:subPropertyOf r:follows . ?p k:hasTag`
	`"#webseries" }`
		206

EQ6a	`SELECT ?n3`
	`WHERE { GRAPH ?g1 { ?n r:follows ?n2 . ?g1 k:hasTag "#webseries" }`
	`?n2 r:follows ?n3 }`
		13,012

EQ6b	`SELECT ?n3`
	`WHERE { ?s ?p ?n2 . ?p rdfs:subPropertyOf r:follows .`
	`?p k:hasTag "#webseries" . ?n2 r:follows ?n3 }`
		13,012

EQ7a	`SELECT ?n4`
	`WHERE { GRAPH ?g1 { ?n r:follows ?n2 . ?g1 k:hasTag "#webseries" }`
	`GRAPH ?g2 { ?n2 r:follows ?n3 . ?g2 k:hasTag "#webseries" }`
	`GRAPH ?g3 { ?n3 r:follows ?n4 . ?g3 k:hasTag "#webseries" } }`
		11,440

EQ7b	`SELECT ?n4`
	`WHERE { ?s ?p ?n2 . ?p rdfs:subPropertyOf r:follows . ?p k:hasTag`
	`"#webseries" .`
	`?n2 ?p2 ?n3 . ?p2 rdfs:subPropertyOf r:follows . ?p2 k:hasTag`
	`"#webseries" .`
	`?n3 ?p3 ?n4 . ?p3 rdfs:subPropertyOf r:follows . ?p3 k:hasTag`
	`"#webseries" }`
		11,440

EQ8a	`SELECT ?n2 ?k ?v`
	`WHERE { GRAPH ?g1 { ?n r:follows ?n2 . ?g1 k:hasTag "#webseries" .`
	`?g1 ?k ?v FILTER (isLiteral(?v)) } }`
		1,269

EQ8b	`SELECT ?n2 ?k ?v`
	`WHERE { ?s ?p ?n2 . ?p rdfs:subPropertyOf r:follows .`
	`?p k:hasTag "#webseries" . ?p ?k ?v FILTER (isLiteral(?v)) }`
		1,269

EQ9	`SELECT ?inDeg (COUNT(*) as ?cnt)`
	`WHERE { SELECT ?n2 (COUNT(*) as ?inDeg)`
	`WHERE { ?n1 (r:knows|r:follows) ?n2 }`
	`GROUP BY ?n2 } GROUP BY ?inDeg ORDER BY DESC(?inDeg)`
		580

EQ10	`SELECT ?outDeg (COUNT(*) as ?cnt)`
	`WHERE { SELECT ?n1 (COUNT(*) as ?outDeg)`
	`WHERE { ?n1 (r:knows|r:follows) ?n2 }`
	`GROUP BY ?n1 } GROUP BY ?outDeg ORDER BY DESC(?outDeg)`
		412

EQ11a	`SELECT (COUNT(?y) as ?cnt)` `WHERE {<http://pg/n6160742> r:follows ?y }`	21
EQ11b	`SELECT (COUNT(?y) as ?cnt)` `WHERE {<http://pg/n6160742> r:follows/r:follows ?y }`	900
EQ11c	`SELECT (COUNT(?y) as ?cnt)` `WHERE {<http://pg/n6160742> r:follows/r:follows/r:follows ?y }`	52,540
EQ11d	`SELECT (COUNT(?y) as ?cnt)` `WHERE {<http://pg/n6160742> r:follows/r:follows/r:follows/r:follows ?y }`	3,573,916

EQ11e	`SELECT (COUNT(?y) as ?cnt)`
	`WHERE {http://pg/n6160742`
	`r:follows/r:follows/r:follows/r:follows/r:follows ?y}`
		257,861,728

EQ12	`SELECT (COUNT(*) AS ?cnt)` `WHERE { ?x r:follows ?y . ?y r:follows ?z . ?z r:follows ?x }`	20,211,887

is no significant difference between the NG and SP approaches. These results are not surprising because each approach uses the same triples store node key/value pairs. In addition, these queries are evaluated with index-based nested loop join (NLJ), which tends to scale with result set size, so the additional triples stored in the SP approach have little effect on the query execution time.

Experiment 2 – Edge-centric Queries: This experiment tests the performance of EQ5a/b (find all edges with tag "#webseries"), EQ6a/b (find all nodes that are endpoints of an edge with tag "#webseries" and find nodes that are followed by these nodes), EQ7a/b (find all 3-hop paths where each edge has tag "#webseries"), and EQ8a/b (find all edge key value pairs for edges with tag "#webseries"). The results of this experiment are shown in Figure 6. The results show that the NG approach performs better for queries involving multiple edge key/value pair accesses. The improved performance can be attributed to avoiding extra joins required to retrieve edge key/value pairs in the SP approach (i.e. three triples vs. two quads). The performance improvement is most obvious in query EQ7a/b due to a significant difference in number of joins.

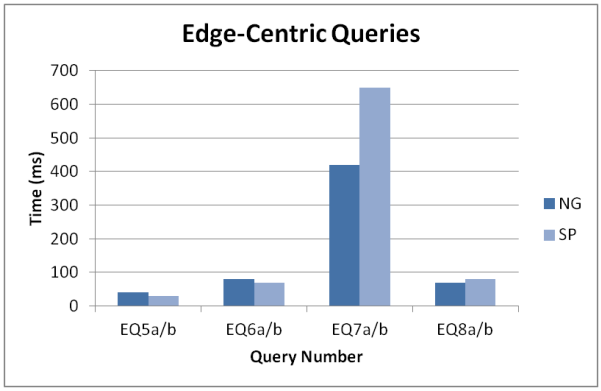

Figure 6. Execution time for edge-centric queries.

Experiment 3 – Aggregate Queries: This experiment tests the performance of aggregate queries over the topological portion of the graph. The specific queries were EQ9 (find the distribution of node in-degree) and EQ10 (find the distribution of node out-degree). The results are shown in Figure 7. Each query finishes in about 9 seconds for both approaches and there is no significant performance difference (< 100ms) between the two approaches, which is not surprising because the same quad (for NG: e-s-p-o) or triple (for SP: -s-p-o) structures are used to store the topological portion of the graph in both approaches.

Experiment 4 – Graph Traversal Queries: This experiment investigated queries EQ11a – EQ11e (count all paths from a specific node ranging in length from 1 to 5). Figure 8 shows the results of this experiment plotted with a log scale. As expected, query execution time rises steeply as path length increases. This steep rise is a consequence of the exponential complexity of the path counting problem, as illustrated by the increase in number of paths found for each query. Nevertheless, over 250 million paths of length 5 were found in less than 4 minutes in both schemes.

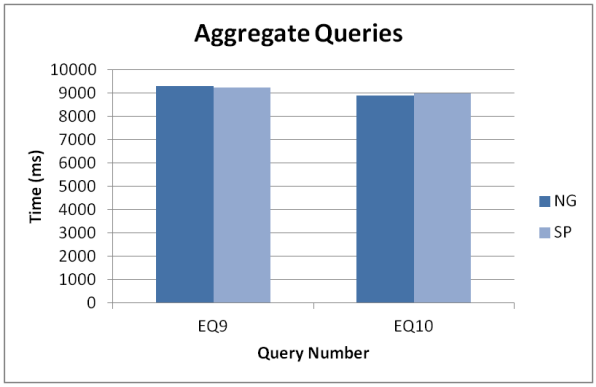

Figure 7. Execution time for aggregate queries.

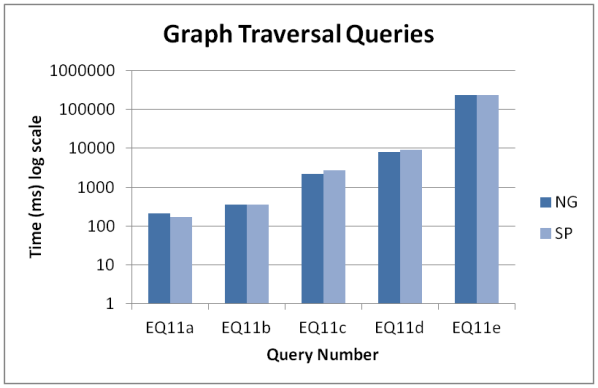

Figure 8. Execution time for graph traversal queries.

In general, the NG approach performs slightly better than the SP approach for these path traversal queries. For the 3, 4 and 5 hop queries, the query optimizer chooses a hash join with a full table scan to access the probe table. In the NG approach, the triples table is smaller, which leads to faster full table scans.

Experiment 5 – Triangle Counting Queries: This experiment investigated query EQ12 (count all follows triangles in the graph). The results of this experiment are shown in Figure 9. In both schemes, over 20 million triangles were found in just over 1 minute: 61 seconds for NG and 65 seconds for SP. Once again, the query optimizer chooses a series of hash joins with full table scans, and the NG approach performs slightly better because of its smaller table size.

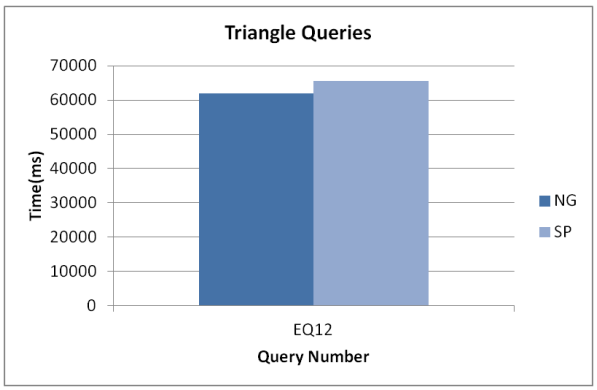

Figure 9. Execution time for triangle counting queries.

4.5 Summary

In general, the SP and NG schemes offer similar performance on a variety of graph queries. Our experiments show that the only significant performance difference occurs when accessing edge key/value pairs. In such cases, the NG approach performs better because fewer joins are required. In addition, the NG approach performs slightly better for large-scale path and triangle counting queries due to a smaller triples table size.

5. Discussion

This section discusses two aspects related to supporting property graphs as RDF.

5.1 Path Queries in SPARQL 1.1

The SPARQL query language is intended primarily for pattern (subgraph) matching rather than path traversal. SPARQL 1.1 introduced property paths, which allow more concise expressions for some SPARQL basic graph patterns and provide the key capability of matching arbitrary length paths [16]. SPARQL 1.1, however, still lacks the ability to reference a path directly in a query (e.g., the ability to store the result of a property path query in a path variable). Without this ability, it is not possible to match an arbitrary length path *and* return the path itself or perform operations based on characteristics of the path, such as path length. Several researchers have proposed extensions of SPARQL that use special variables to reference matched paths to address this shortcoming, for example SPARQLeR [29], SPARQ2L [30], G-SPARQL [19]. Such extensions would effectively allow SPARQL to function as both a pattern matching and graph traversal language.

5.2 Benefits of Modeling Property Graphs using RDF

Using either of the models, once the data becomes available as RDF, there are a few interesting possibilities, which go beyond what one would normally do with property graphs.

The predicate IRIs corresponding to edge label and keys in key/values could be mapped through `owl:equivalentProperty` assertions to properties from existing domain ontologies. Similarly, `owl:sameAs` , which already has a heavy usage in linked data integration [26], can be used to map generated IRIs of resulting RDF data with existing linked data. Once such mapping is established, we can make use of existing OWL Inference Engines (such as the native inference engine of Oracle [27]) to pre-compute entailment that can semantically enrich the transformed RDF data thereby allowing us to do more interesting queries. This step would also make it easier and more useful to publish transformed property graph data as linked data. Note that one could argue that the native property graph data itself could be augmented in a similar fashion. However, this step is easier once the data is transformed to RDF due to readily available domain ontologies and numerous RDF datasets on the web [28].

Example - Linking Twitter Data with WordNet: Consider the Twitter social network dataset used for experiments in Section 4. We loaded the basic version of Wordnet RDF dataset that groups nouns, verbs, adjectives and adverbs into sets of cognitive synonyms (s*ynsets*), each expressing a distinct concept that is characterized by a *word sense* label [31].

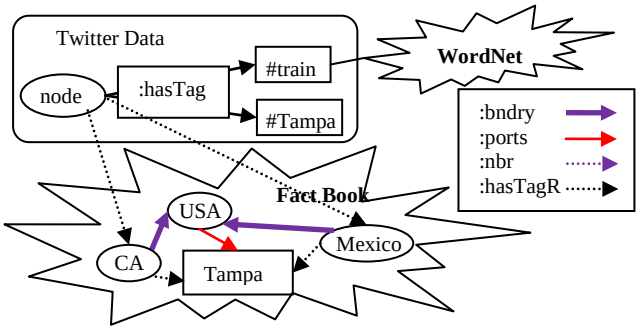

Figure 10. Linking Twitter data with community RDF datasets

Among 33,422 distinct tags used in the sample Twitter data, we found occurrences of 5,993 proper words. Thus, a user can take advantage of the Wordnet ontology to do *query term expansion*, when searching on the `:hasTag` attribute as shown in query pattern below:

```
SELECT ?n
WHERE{?w rdfs:label ?label .
      ?w wn:senseLabel "train"@en-us .
      ?n k:hasTag  ?y
  FILTER(STR(?y)=CONCAT("#",STR(?label)))}
```

For the input word `'train'`, the query returns 6 results with `'#train'` , plus 13 extra results (2 with `'#educate'`, and 11 with `'#prepare'`) due to the query term expansion made possible by Wordnet.

Example – Linking Twitter Data with World Fact Book and Use of Inference: We also loaded the World Fact Book RDF dataset [32], which contains information on the history, people, government, economy, geography, communications, transportation, military, and transnational issues for all countries of the world. Among 33,422 distinct tags used in the sample Twitter data, we found occurrences of 199 proper locations. Thus, a user can use Fact Book properties about countries to perform node selection.

Furthermore, the query processing can be accelerated by pre-computing entailment of CIA Fact Book with Oracle's native RDF/OWL Inference Engine. For example, using OWL2 RL entailment on the Fact Book ontology, we can infer that Mexico and Canada are neighbors to port `'Tampa'` using property chains. Furthermore, using Oracle user-defined rules capability, we can also infer a new `:hasTagR` property that directly links the node with `'#Tampa'` tag to its neighboring countries (see Figure 10). The inferred edges can thus allow refining the filtering on node attributes.

6. Conclusion and Future Directions

The paper examined the problem of supporting property graphs as RDF using the capabilities in Oracle Database. Three models, namely, reification, named graph, and subproperty based, for representing property graphs in RDF were presented. All these models can be supported using RDF. Furthermore, standard SPARQL queries can be used to perform traversal in addition to accessing node and edge key/values.

We demonstrated the feasibility of our approach by implementing the proposals in Oracle Database. An experimental study conducted using a Twitter social network dataset showed that the performance of various types of queries was reasonable. As

expected, the edge traversal queries accessing edge key/values took the longest time. The study also identified the current limitation of SPARQL, especially in property path queries, where length limits cannot be specified, and that could be a problem for large highly connected property graphs. An alternative for such cases is to perform traversal procedurally similar to the approach of Gremlin [22].

Lastly, we identified the possibility of creating hybrid applications that go beyond just representing and querying property graphs by taking advantage of named graphs, OWL inference engines, and the use of global identifiers to publish as linked data.

7. REFERENCES

[1] Resource Description Framework (RDF). 2004, Retrieved Jan. 17, 2014, from http://www.w3.org/RDF/.

[2] Defining a Property Graph. 2012, Retrieved Jan. 17, 2014, from https://github.com/tinkerpop/gremlin/wiki/Defining-a-Property-Graph.

[3] Blueprints Java API. 2013, Retrieved Jan. 17, 2014, from https://github.com/tinkerpop/blueprints/wiki.

[4] RDF Vocabulary Description Language 1.0: RDF Schema. 2004, Retrieved Jan. 17, 2014, from http://www.w3.org/TR/rdf-schema/.

[5] OWL 2 Web Ontology Language Document Overview (Second Edition). 2012, Retrieved Jan. 17, 2014, from http://www.w3.org/TR/owl2-overview/.

[6] Neo4j the graph database. 2014, Retrieved Jan. 17, 2014, from http://www.neo4j.org.

[7] Sparsity Technologies. 2013, Retrieved Jan. 17, 2014, from http://sparsity-technologies.com/dex.

[8] Objectivity: InfiniteGraph. 2013, Retrieved Jan. 17, 2014, from http://www.objectivity.com/infinitegraph.

[9] SPARQL Query Language for RDF. 2008, Retrieved Jan. 17, 2014, from http://www.w3.org/TR/rdf-sparql-query/.

[10] Oracle Database Spatial and Graph Option. 2014, Retrieved Jan. 17, 2014, from http://www.oracle.com/technetwork/database-options/spatialandgraph/overview/index.html.

[11] Openlink Virtuoso Universal Server. 2014, Retrieved Jan. 17, 2014, from http://virtuoso.openlinksw.com/.

[12] Franz Inc. AllegroGraph. 2013, Retrieved Jan. 17, 2014, from http://www.franz.com/agraph/allegrograph/.

[13] OntoText OWLIM. 2013, Retrieved Jan. 17, 2014, from http://www.ontotext.com/owlim.

[14] Neo4j Cypher Query Language. 2014, Retrieved Jan. 17, 2014, from http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html.

[15] OWL 2 Web Ontology Language Profiles (Second Edition). 2012, Retrieved Jan. 17, 2014, from http://www.w3.org/TR/owl2-profiles/.

[16] SPARQL 1.1 Query Language. 2013, Retrieved Jan. 17, 2014, from http://www.w3.org/TR/sparql11-query/.

[17] Social Network Intelligence BenchMark. 2013, Retrieved Jan. 17, 2014, from http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark.

[18] Oracle Spatial and Graph RDF Semantic Graph - Performance. 2014, Retrieved Jan. 17, 2014 from http://www.oracle.com/technetwork/database/options/spatialandgraph/learnmore/performance-086309.html.

[19] Sakr, S., Elnikety, S., and He, Y. 2012. G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs. In *Proceedings of CIKM 2012*, 335-344.

[20] Angles, R., Prat-Perez, A., and Dominguez-Sal, D. 2013. Benchmarking database systems for social network applications. In *Proceedings of GRADES Workshop 2013*.

[21] Vicknair, C. et al. 2010. A Comparison of a Graph Database and a Relational Database. In *Proceedings of ACM Southeast Regional Conference 2010*.

[22] Gremlin Graph Traversal Language, https://github.com/tinkerpop/gremlin/wiki

[23] Snap: Network datasets: Social Circles: Twitter, http://snap.stanford.edu/data/egonets-Twitter.html

[24] McAuley, J. and Leskovec, J. 2012. Learning to Discover Social Circles in Ego Networks. In *Proceedings of NIPS 2012*, 548-556.

[25] Guo, Y., Pan, Z., and Heflin, J. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics*. 3, 2-3 (October 2005), 158-182.

[26] Ding, L., Shinavier, J., Shangguan, Z., and McGuinness, D. 2010. SameAs Networks and Beyond: Analyzing Deployment Status and Implications of owl: sameAs in Linked Data. In *Proceedings of ISWC 2010*, 145-160.

[27] Wu, Z., Eadon, G., Das, S., Chong. E. I., Kolovski, V., Annamalai, M., and Srinivasan, J. 2008. Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In *Proceedings of ICDE 2008*, 1239-1248.

[28] Linked Data: Connect Distributed Data Across the Web. 2014, Retrieved Jan. 17, 2014, from http://linkeddata.org/.

[29] Kochut, K. and Janik, M. 2007. SPARQLeR: Extended SPARQL for semantic association discovery. In *Proceedings of ESWC 2007*, 145–159.

[30] Anyanwu, K., Maduko, A., and Sheth, A. P. 2007. SPARQ2L: towards support for subgraph extraction queries in RDF databases. In *Proceedings of WWW 2007*, 797–806.

[31] WordNet: A lexical database for English. 2013, Retrieved Jan. 17, 2014 from http://wordnet.princeton.edu.

[32] CIA: The World Factbook. 2014, Retrieved Jan. 17, 2014, from https://www.cia.gov/library/publications/the-world-factbook.

[33] RDF 1.1 Concepts and Abstract Syntax. 2014, Retrieved Jan. 17, 2014, from http://www.w3.org/TR/rdf11-concepts.

[34] Franz's AllegroGraph® Sets New Record - 1 Trillion RDF Triples, 2013, Retrieved Jan. 17, 2014, from http://www.franz.com/about/press_room/trillion-triples.lhtml.

[35] RDF Primer. 2004, Retrieved Jan. 17, 2014, from http://www.w3.org/TR/rdf-primer/.

[36] SPARQL 1.1 Update. 2013, Retrieved Jan. 17, 2014 from http://www.w3.org/TR/sparql11-update/.