SQLGraph: An Efficient Relational-Based Property Graph Store

Wen Sun[†], Achille Fokoue[‡], Kavitha Srinivas[‡], Anastasios Kementsietsidis[§], Gang Hu[†], Guotong Xie[†]

[†]IBM Research - China
{sunwenbj, hugang, xieguot}@cn.ibm.com

[‡]IBM Watson Research Center
{achille, ksrinivs}@us.ibm.com

[§]Google Inc.
akement@google.com

ABSTRACT

We show that existing mature, relational optimizers can be exploited with a novel schema to give better performance for property graph storage and retrieval than popular noSQL graph stores. The schema combines relational storage for adjacency information with JSON storage for vertex and edge attributes. We demonstrate that this particular schema design has benefits compared to a purely relational or purely JSON solution. The query translation mechanism translates Gremlin queries with no side effects into SQL queries so that one can leverage relational query optimizers. We also conduct an empirical evaluation of our schema design and query translation mechanism with two existing popular property graph stores. We show that our system is 2-8 times better on query performance, and 10-30 times better in throughput on 4.3 billion edge graphs compared to existing stores.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Property graphs; Gremlin

Keywords

Property graphs; Relational Storage; Gremlin

1. INTRODUCTION

There is increased interest in graph data management recently, fueled in part by the growth of RDF data on the web, as well as diverse applications of graphs in areas such as social network analytics, machine learning, and data mining. The dominant focus in the academic literature has been on RDF data management (e.g., [39, 18, 27, 24, 6, 25, 16]). Much of this work targets support of the graph data model over native stores or over distributed key-value stores. Few target relational systems because of concerns about the efficiency of storing sparse graph adjacency data in relational storage (e.g., Jena SDB [38], C-store [1]). Yet relational systems offer significant advantages over noSQL or native systems because they are fully ACID compliant, and they have industrial strength support for concurrency, locking, security and query optimization. For graph workloads that require these attributes, relational databases may be a very attractive mechanism for graph data management. In fact, in a recent paper, Bornea et al. [5] show that it is possible to shred RDF into relational storage in a very efficient manner with significant gains in performance compared to native graph stores.

Significant progress has been made as well on query optimization techniques for querying RDF graph data. The standard graph query language for RDF is called SPARQL, which is a declarative query language for specifying subgraphs of interest to the user. Many techniques have been proposed for optimizing SPARQL queries (e.g., [33, 34]), with some specifically targeting the problem of translating SPARQL to SQL (e.g., [7, 9, 13]). As a result, it has been possible to use relational databases for RDF data.

However, RDF is just one model for graph data management. Another model which is rapidly gaining popularity for graph storage and retrieval is the so-called property graph data model, which differs from RDF in two important ways: (a) it has an object model for representing graphs, (b) it has an accompanying query language called Gremlin which varies significantly from SPARQL.

The data model for property graphs is a directed labeled graph like RDF, but with attributes that can be associated with each vertex or edge (see Figure 2a for an example of a property graph). The attributes of a vertex such as *name* and *lang* are encapsulated in an object as key-value pairs. In RDF, they would be modeled as extra edges from the vertex to literals with the labels of *name* and *lang*, respectively. Similarly, the attributes of an edge are associated with an object, with attributes such as *weight* and its value represented as key-value pairs. One mechanism to model edge attributes in RDF is using reification, where four new edges are added to the graph to refer to the pre-existing edge. Thus, to model the edge between vertex *4* and *3*, there would be a new vertex *5* added to the graph. This vertex would have an edge labeled *subject* to *4*, an edge labeled *object* to *3*, and an edge label labeled *predicate* to *created*. The fourth edge would link *5*'s type to a *Statement*, to indicate that this vertex reflects metadata about another RDF statement (or edge). While this method of modeling edge attributes is very general, it is verbose and inefficient in terms of storage and retrieval. Other techniques in RDF

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
ACM 978-1-4503-2758-9/15/05.
http://dx.doi.org/10.1145/2723372.2723732.

include adding a fourth element to each edge (such that each edge is now described with four elements), and this fourth element (*5* in our example) can now be used as a vertex in the graph, so that edge attributes such as *weight* can be added as edges from it. Property graphs provide a simplified version of this latter form of reification, by adding an object for every edge, and encapsulating edge attributes as key values. That is, the notion of one level of reification is built in to the notion of a property graph. How to deal with reification in RDF is however, not standard (see [14]), and hence, most of the literature directed at the study of RDF data management has ignored the issue of how to efficiently store RDF edge attributes.

Another important difference between property graphs and RDF is in the query language. While SPARQL, the query language for RDF, is declarative, Gremlin is a procedural graph traversal language, allowing the programmer to express queries as a set of steps or 'pipes'. For example, a typical query might start at all vertices filtered by some vertex attribute *p*, traverse outward from that vertex along edges with labels *a*, and so on. Each step produces an iterator over some elements (e.g., edges or vertices in the graph). In Gremlin, it is possible to have arbitrary code in some programming language such as Java or Groovy act as a pipe to produce side effects. This poses a significant challenge to query optimization, because much of the prior work on SPARQL cannot be re-applied for Gremlin.

Because the data model and query language for property graphs reflect an object-oriented view of graphs, they seem to be gaining popularity with Web programmers, as seen in the growing number of stores aimed at this model. Examples include Apache Titan[1], Neo4j[2], DEX [21], OrientDB[3], InfiniteGraph[4] to name a few. To our knowledge, all of them are built using either native support or other noSQL stores. For instance, Apache Titan supports BerkeleyDB (a key-value store), Cassandra, and HBase (distributed column stores). OrientDB is a document database like MongoDB, but doubles as a graph store. Neo4j and DEX support graphs natively. The question we ask in this paper, is whether one can provide efficient support for property graphs over relational stores. Specifically, we examine alternative schema designs to efficiently store property graphs in relational databases, while allowing the Web programmer access to a query language such as Gremlin. Note that because Gremlin is a procedural language, it may include side effects that make it impossible to translate into SQL. In this paper, we focus on Gremlin queries with no side-effects or complex Java code embedded in the query, to examine if such an approach is even feasible. We outline a generic mechanism to convert Gremlin queries into SQL and demonstrate that this approach does in fact produce efficient queries over relational storage.

There are two key ideas in Bornea et al. [5] that we examine in detail with respect to their applicability to property graphs: (a) the adjacency list of a vertex in a graph is accommodated on the same row as much as possible, (b) to deal with sparsity of graphs and uneven distribution of edge labels in the graph, each edge label is 'hashed' to a small set of columns and each column is overloaded to contain multiple edge labels. The hashes are optimized to minimize conflicts, by analysis of the dataset's characteristics. Bornea et al. [5] demonstrate the efficacy of these ideas to store the adjacency information in a graph, but the property graph model presents additional challenges in terms of storage of edge and vertex information. One option is to store edge or vertex information in another set of tables analogous to those described in [5]. Another option is to examine whether these additional attributes of a property graph model can be stored more efficiently in non-relational storage such as JSON storage since most commercial and open source database engines now support JSON. We examined both options to make informed decisions about the schema design for property graphs, and empirically evaluated their benefits. The outcome is a novel schema that combines relational with non-relational storage for property graphs, because as we show in a series of experiments, non-relational storage provides advantages over relational storage for lookup of edge and vertex attributes.

Our contributions in this paper are fourfold: (a) We propose a novel schema which exploits both relational and non-relational storage for property graphs, (b) We define a generic technique to efficiently translate a useful subset of Gremlin queries into SQL, (c) We modify two very different graph benchmarks (i.e., the DBpedia SPARQL benchmark and the LinkBench) to measure property graph performance because there are no accepted benchmarks yet for this query language[5]. Our benchmarks include graphs in the 300M-4.3 billion edge graphs. We are not aware of any comparison of property graph stores for graphs of this size. (d) We show that our ideas for property graph data management on relational databases yield performance that is 2-8X better than existing stores such as Titan, Neo4j and OrientDB on read only, single requester workloads. On concurrent workloads, that advantage grows to about 30X over existing stores.

2. RELATED WORK

Graph data management is a broad area and falls into three categories: (a) graph stores targeting the RDF data model, (b) graph stores targeting the property graph data model, (c) graph processing frameworks. The first two target subgraph queries over graph data, or selective graph traversal, whereas the third targets global graph manipulations where each step performs some complex logic at each vertex or edge. Our focus is on the graph data management in the first two cases in this paper.

Numerous schemes have been proposed for storage of graph data over relational and non-relational storage (see [31], [26] for surveys) in both centralized and distributed settings. These schemes include storage of edges in (a) vertical tables with extensive use of specialized indexes for performance [25], (b) predicate-oriented column stores to deal with sparsity [1], or to enable scale out [28], (c) different tables for each RDF type [38], (d) a relational hash table to store adjacency lists for each vertex [5]. To our knowledge, ours is the first work to explore combining relational with non-relational storage to address the problem of storing a graph along with metadata about each edge or vertex.

[1]`http://thinkaurelius.github.io/titan/`
[2]`http://www.neo4j.org`
[3]`http://www.orientechnologies.com/orientdb/`
[4]`http://www.objectivity.com/infinitegraph`

[5]Standardization efforts are underway, but the benchmarks are still in draft form as of today [2]

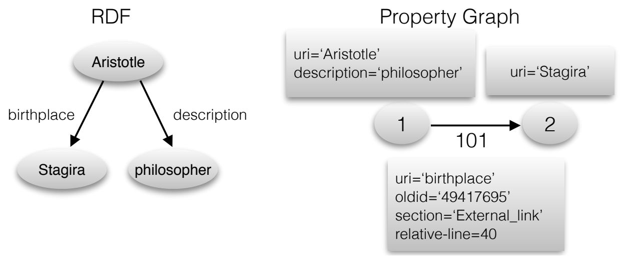


Figure 1: Conversion of RDF to Property graphs

As we stated earlier, compilation of declarative query languages such as SPARQL into SQL is a well-studied problem [7, 9, 13], both in terms of mapping SPARQL's semantics to SQL, and in terms of providing SPARQL views over legacy relational schemas [10, 29, 30]. There does not appear to be any work targeting the translation of Gremlin to SQL, perhaps due to the fact that it is a procedural language. Yet, for many graph applications, Gremlin is used to traverse graphs in a manner that can be expressed declaratively. In fact, one recent attempt contrasts performance on Gremlin with performance on other SPARQL-like declarative query languages on Neo4j [15].

There are numerous benchmarking efforts in the RDF space targeting query workloads for graphs. Examples include SP2Bench [32], LUBM [12], UOBM [19], BSBM [4] and DBpedia [22], but none of them can be easily modeled as property graphs because they do not have edge attributes, except for DBpedia. Other graph benchmarks such as HPC Scalable Graph Analysis Benchmark [11] and Graph500 [23] largely target graph processing frameworks, and once again have no edge attributes, or even edge labels. Ciglan et al. [8] proposed a general graph benchmark that evaluates the performance of 2-hop and 3-hop BFS kernals over a property graph, but the benchmark is not publicly accessible. PIG [20] is a benchmark for tuning or comparison of different property graph specific operations, but it targets a Blueprints API which performs atomic graph operations. A complex graph traversal can use these atomic graph operations in sequence, but the performance overhead is very serious in normal client-server workloads. An ongoing property graph benchmark project is the Linked Data Benchmark Council [2], where a Social Network Benchmark is under development targeting interactive graph workloads, but the current queries are still vendor-specific, and do not support Gremlin. LinkBench [3] is a benchmark for evaluating the performance of different databases on supporting social graph query workloads. It was initially designed for MySQL and HBase, and it generates synthetic datasets and queries based on traces of Facebook's production databases. Since none of the current benchmarks support Gremlin natively, we chose to adapt DBpedia and LinkBench as our target benchmarks for two different type of workloads. DBpedia's queries are more complex, but target a read only benchmark. LinkBench focuses on atomic graph operations like PIG, but has very good support for measuring concurrent read write workloads.

3. SCHEMA DESIGN

Bornea et al. [5] outlined a novel schema layout for storage of RDF data in relational databases, and demonstrated its efficacy against other native RDF systems. In this paper, we evaluate the generality of this design for the property graph data model. Recall that in property graphs, each vertex or edge can have multiple key-value pairs that serve as attributes on the vertex or edge. An important point to note is that access to these key-value pairs associated with an edge or vertex is usually through the vertex or edge identifier, unless specialized indexes have been created by a user on specific keys or values for the vertex or edge. In other words, access in property graph models tends to be like very much like a key-value lookup. To help make informed decisions on schema design for this specific model and its access patterns, we created a micro benchmark (a) to empirically examine whether adjacency information is best stored in a schema layout outlined for RDF as in [5] or whether a backend store supporting key-value lookups was more appropriate, and (b) to evaluate whether it is better to store vertex and edge attributes in key-value type stores or shredded within a relational model as in [5].

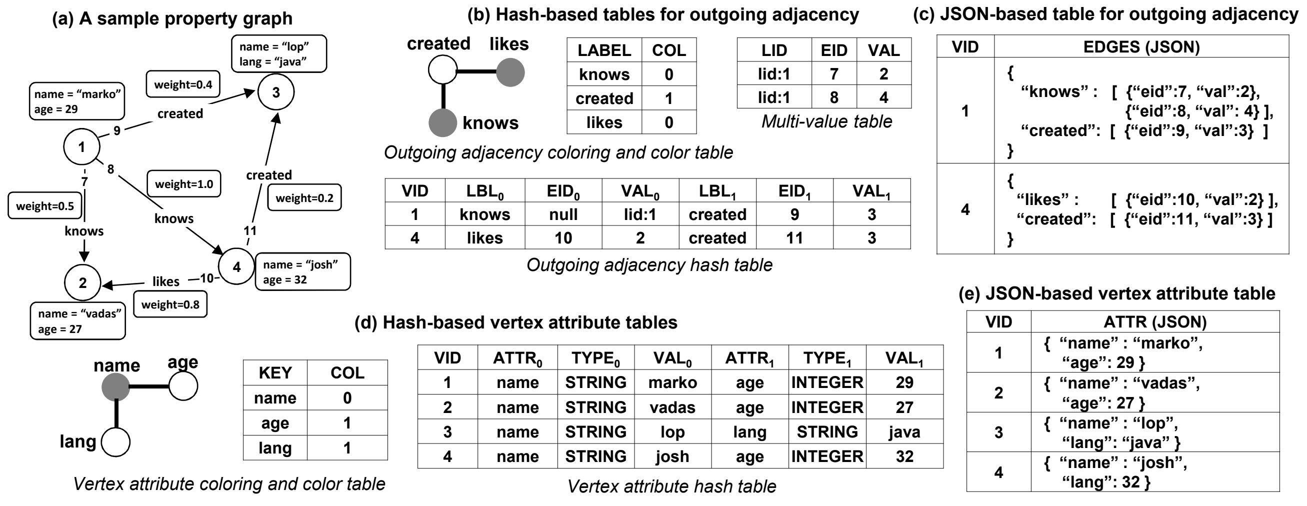
3.1 Micro Benchmark Design

For this micro-benchmark, we needed (a) fairly complex graph traversal queries to contrast differing approaches to storing adjacency and (b) simple vertex or edge attribute lookups to contrast different approaches for storing edge or vertex metadata. As stated earlier, there is a dearth of realistic graph datasets for property graphs. Some of the synthetic datasets that exist such as LinkBench are clearly not designed to study graph traversal performance, although they do provide a nice benchmark for edge attribute lookups. As a result, we turned to real graphs in the RDF space to adapt them for use as a micro-benchmark. This allows us to re-use the exact same graph for both studies, by just varying the queries.

To adapt the DBpedia 3.8 RDF data model into a property graph data model, we translated each triple in the RDF dataset into a property graph using the following rules: (a) any subject or object node in RDF became a vertex with a unique integer ID in the property graph, (b) object properties in RDF were specified as adjacency edges in the property graph, where the source and the target of the edge were vertex IDs, and the edge was identified by an integer ID, (c) datatype properties in RDF were specified as vertex attributes in the property graph, (d) provenance or context information, encoded in the DBpedia 3.8 dataset as n-quads were converted into edge attributes. Figure 1 shows the conversion of DBpedia from an RDF data model to property graphs. Note that URIs are abbreviated for succinctness. This conversion helped us study characteristics of query performance such as *k hop* traversal or vertex attribute lookup on the same real graph data, without having to revert to the creation of new synthetic datasets for each aspect of our study. We define the set of queries we used for each study on this same graph in the sections below.

3.2 Storing Adjacency

An interesting aspect of popular stores for storing property graphs is that they are based on noSQL key-value stores or document stores such as Berkeley DB, Cassandra, HBase or OrientDB. In storing sparse RDF graph data, earlier work has shown that [5] shredding vertex adjacency lists into a relational schema provides a significant advantage over other mechanisms such as property tables or vertical stores which


(a) A sample property graph
name = "marko"
age = 29
name = "lop"
lang = "java"
weight=0.4
created
weight=1.0
created
weight=0.2
weight=0.5
knows
knows
name = "josh"
age = 32
likes
weight=0.8
name = "vadas"
age = 27
(b) Hash-based tables for outgoing adjacency
created
likes
knows
Outgoing adjacency coloring and color table
Multi-value table
Outgoing adjacency hash table
(c) JSON-based table for outgoing adjacency
(d) Hash-based vertex attribute tables
name
age
lang
Vertex attribute coloring and color table
Vertex attribute hash table
(e) JSON-based vertex attribute table

LABEL	COL
knows	0
created	1
likes	0

LID	EID	VAL
lid:1	7	2
lid:1	8	4

VID	LBL_0	EID_0	VAL_0	LBL_1	EID_1	VAL_1
1	knows	null	lid:1	created	9	3
4	likes	10	2	created	11	3

VID	EDGES (JSON)
1	{ "knows" : [{"eid":7, "val":2}, {"eid":8, "val": 4}], "created": [{"eid":9, "val":3}] }
4	{ "likes" : [{"eid":10, "val":2}], "created": [{"eid":11, "val":3}] }

KEY	COL
name	0
age	1
lang	1

VID	$ATTR_0$	$TYPE_0$	VAL_0	$ATTR_1$	$TYPE_1$	VAL_1
1	name	STRING	marko	age	INTEGER	29
2	name	STRING	vadas	age	INTEGER	27
3	name	STRING	lop	lang	STRING	java
4	name	STRING	josh	age	INTEGER	32

VID	ATTR (JSON)
1	{ "name" : "marko", "age": 29 }
2	{ "name" : "vadas", "age": 27 }
3	{ "name" : "lop", "lang": "java" }
4	{ "name" : "josh", "lang": 32 }

Figure 2: Hash-based and JSON-based schema for graph adjacency and attributes.

store all edge information in a single table. However, given the somewhat stylized access patterns in property graphs, it is unclear whether storing adjacency lists in non-relational key-value stores would provide more efficient storage. A relevant research question then is whether such stores provide more efficient access for property graphs.

Most modern relational databases such as DB2, Oracle or Postgresql have features to support both relational and non-relational storage within the same database engine, making it possible to perform an empirical comparison of the utility of relational versus non-relational storage structures for property graphs. Our first study was to compare relational versus non-relational methods for the storage of adjacency lists of a vertex.

For the relational schema, we re-used the approach specified in [5]; i.e., the adjacency list of an edge was stored in a relational table by hashing each edge label to a specific column pair, where one column in the pair stored the edge label, and the other column in the pair stored the value as shown in Figure 2b. In this schema, a given column is overloaded in terms of the number of different edge labels it can store to minimize the use of space. Figure 2 shows this column overloading, such that *likes* and *knows* edges are stored in the same column *0*, both having hashed to column *0*. RDF graphs can have thousands of edge labels, so overloading columns reduces sparsity in the relational table structure. However, this mechanism can also result in conflicts if one uses a hashing function that does not capitalize on the characteristics of the data. Bornea et al. [5] introduced a hashing function based on an analysis of the dataset characteristics. Specifically, the technique involves building a graph of edge label co-occurrences where two edge labels share an edge if they occur together in an adjacency list (e.g., *knows* and *created* in 2b). A graph coloring algorithm is then applied to this graph of edge label co-occurences, to ensure that two predicates that co-occur together in an adjacency list never get assigned to the same color. Because the color represents a specific column in the store, this hashing function minimizes conflicts by assigning predicates that co-occur together in a dataset to different columns. In the example, this means that *knows* and *created* would be assigned to different columns. With this type of hashing, Bornea et al. [5] showed that across multiple benchmarks, one can accomodate most adjacency lists on a single row, and moreover, this schema layout has significant advantages for query performance on many different RDF benchmarks[6].

We contrasted this relational schema to an approach where the entire adjacency list was stored as a JSON object. Our choice of JSON was driven by the fact that most modern relational engines support JSON stores in an efficient way, and this support co-exists with relational storage in the same database engine. A comparison can therefore be made between the two approaches in a more controlled setting. In a later section, we perform an experimental evaluation of our approach against other popular property graph stores, which rely on different key-value stores to rule out the possibility that any of the differences we see in are purely due to implementation specific differences within the engine for relational versus non relational data.

Our queries shown in Table 1 to study adjacency storage were focused around graph traversal, because these sorts of queries can highlight inefficiencies in adjacency storage. We created a set of queries on the DBpedia 3.8 property graph to vary (a) the number of hops that had to be traversed in the query, (b) the size of the starting set of vertices for the traversal, (c) the result size which reflects query selectivity as shown in Table 1. All the queries shown in Table 1 involved traversal over *isPartOf* relations between places, or *team* relationships between soccer players and their teams[7] In this and all other experiments, we always discarded the first run, so we could measure system performance with a warm cache. We ran each query 10 times, discarded the first run, and report the mean query time in our results.

The results shown in Figure 3 were unequivocal. Storing adjacency lists by shredding them in a relational table has significant advantages over storing them in a non-relational store such as JSON. Query times were significantly faster for the relational shredded approach (mean: 3.2 s, standard deviation: 2.2 s) compared to the non-relational JSON ap-

[6]Source code for ideas described in Bornea et al. is available at `https://github.com/Quetzal-RDF/quetzal`
[7]In the case of *team* relations, we traversed these relations ignoring the directionality of the edge.

Query	Query ID	Num. Hops	Input Size	Result Size
isPartOf	1	3	16000	257K
	2	6	16000	257K
	3	9	16000	257K
	4	5	100	4K
	5	5	1000	30K
	6	5	10000	196K
team	7	4	1	61K
	8	6	1	234K
	9	8	1	267K
	10	6	10	255K
	11	6	100	266K

Table 1: Adjacency queries

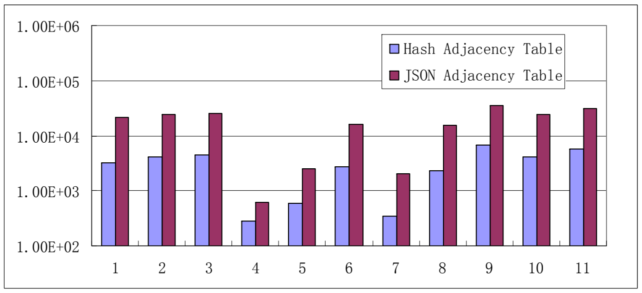


Figure 3: Results of the adjacency micro-benchmark.

proach (mean: 18.0 s, standard deviation: 11.9). These results suggest that there is value in re-using the relational shredded approach to store the adjacency information in a property graph model. Our next question was how to extend the shredded relational schema approach to store edge and vertex attributes in the property graph data model, which we address in the next section.

3.3 Storing Vertex and Edge Attributes

Query Type	ID	Attribute	Filter	Result Size
String	1	national	not null	239
	2	national	like %en	218
	3	genre	not null	28K
	4	genre	like %en	27K
	5	title	not null	231K
	6	title	like %en	222K
	7	label	not null	10M
	8	label	like %en	10M
Numeric	9	regionAffiliation	not null	223
	10	regionAffiliation	1958	3
	11	populationDensitySqMi	not null	28K
	12	populationDensitySqMi	100	32
	13	longm	not null	205K
	14	longm	1	3K
	15	wikiPageID	not null	11M
	16	wikiPageID	29899664	1

Table 2: Queries of the vertex attribute lookup micro-benchmark.

As we noted earlier, the key-value attributes on the vertices and edges is the only difference between property graphs and RDF graphs in structure. We started by examining whether we could extend the existing relational schema by adding two more tables for the storage of vertex and edge key value properties respectively. To store edge and key value attributes in these tables, we could use the same technique we

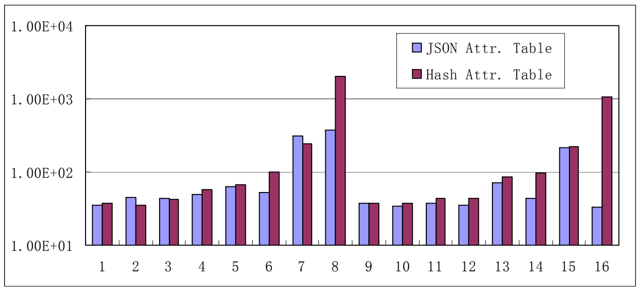


Figure 4: Results of the vertex attribute lookup micro-benchmark.

outlined in [5], by hashing attributes to columns in a standard relational table. However, note that access to these attributes tends to be very much like a simple key value lookup (because it does not involve joins). Shredding the key values into a relational table may be unnecessary. Hence, we compared the choice of a relational or non-relational approaches for storage of these attributes, just as we did in the prior sub-section. Once again, as shown in Figure 2 d, we shredded vertex or edge attributes using a coloring based hash function, and contrasted it with an approach that stored all the vertex or edge information in a single JSON column (see Figure 2 e).

	Vertex Attribute Hash Table	Outgoing Adjacency Hash Table	Incoming Adjacency Hash Table
No. of Hashed Labels	53K	13K	13K
Hashed Bucket Size	106	125	19
Spill Rows Percentage	3.2%	0	0.6%
Long String Table Rows	586K	0	0
Multi-Value Table Rows	49M	244M	243M

Table 3: Comparison of using hash tables for vertex attributes and adjacency.

To evaluate the efficacy of these different storage mechanisms, we used the same DBpedia benchmark but changed the queries so that they were lookups on a vertex's attributes[8]. In property graphs, a user would typically add specialized indexes for attributes that they wanted to lookup a vertex or an edge by. We therefore added indexes for queried keys and attributes both for the shredded relational table and when the vertex's attributes were stored in JSON. Table 2 shows the queries we constructed for this portion of our study. Across queries, we varied (a) whether the queried attribute values were strings or required casts to numeric, (b) whether the query was a simple lookup to check if the key of the attribute existed (the *not null* queries), or whether it required the value as well (these in addition could be equality comparisons such as the lookup to see if *longm* had the value 1, or string functions to evaluate if the query matched some substring), (c) whether the query was selective or not selective. The results are shown in Figure 4. Vertex attribute lookups on the JSON attribute table (mean: 92 ms, standard deviation: 108 ms) were better than the relational shredded table lookups (mean: 265 ms, standard deviation: 537 ms).

[8]We did not test lookup of an edge's attributes because the mechanism is the same.

VID^+	SPILL	......	EID_j	LBL_j	VAL_j	......	EID_K	LBL_K	VAL_K
1	0		null	knows	101		9	created	3
4	0		10	like	2		11	created	3

(a) Outgoing Primary Adjacency (OPA)

VALID^+	EID	VAL
101	7	2
101	8	4

(b) Outgoing Secondary Adjacency (OSA)

VID^+	SPILL	......	EID_p	LBL_p	VAL_p	......	EID_q	LBL_q	VAL_q
2	0		7	knows	1		10	like	4
3	0		null	null	null		null	created	102
4	0		8	knows	1		null	null	null

(c) Incoming Primary Adjacency (IPA)

VALID^+	EID	VAL
102	9	1
102	11	4

(d) Incoming Secondary Adjacency (ISA)

VID^*	ATTR (JSON object)
1	{ "name"="marko", "age"=29 }
2	{ "name"="vadas", "age"=27 }
3	{ "name"="lop", "lang"="java" }
4	{ "name"="josh", "age"=32 }

(e) Vertex Attributes (VA)

EID^*	INV	OUTV	LBL	ATTR (JSON object)
7	1	2	knows	{ "weight"=0.5 }
8	1	4	knows	{ "weight"=1.0 }
9	1	3	created	{ "weight"=0.4 }

(f) Edge Attributes (EA)

Figure 5: Schema of the proposed property graph store. "*" denotes the primary key, and "+" denotes the indexed column.

Another interesting aspect of using JSON in this dataset is shown in the characteristics of the shredded relational hash tables for vertex attributes for DBpedia, as shown in Table 3. Clearly, the relational hash table approach is efficient only to the degree that the entire adjacency list of a vertex is stored in a single row. For outgoing edges (and incoming edges) in DBpedia, if one considers just the adjacency data, this is mostly true with no spill rows in the outgoing adjacency hash table and 0.6% spills in the incoming adjacency hash table. The vertex attribute hash table however has more spills, and has a number of long strings in the attributes which cannot be put into a single row. Storing this data in a shredded relational table thus means more joins in looking up vertex attributes, either because rows have spilled due to conflicts, because long strings are involved, or because a vertex has multiple values for a given key. In JSON, we eliminate joins due to spills, long strings, or multi-valued attributes. Moreover, because the shredded relational table needs a uniform set of columns to store many different data types, it needs casts, which are eliminated in JSON. Thus, as long as these values do not participate in a join again, we see substantial gains in using JSON to store these attribute values. Note that JSON lookups for simple attribute lookups (the *not null*) queries were not different from the shredded relational table, suggesting that when joins are not involved, both storage systems do equally well.

3.4 The Proposed Schema

Given the results of the micro-benchmarks, we designed a novel schema that combined relational storage for adjacency information along with JSON storage for vertex and edge attributes. Figure 5 illustrates the proposed schema with the sample property graph in Figure 2a.

The primary tables for storing outgoing and incoming adjacency (OPA, and IPA) directly apply the coloring and hashing ideas in the RDF store [5] to store edge labels, their values and additionally an edge ID as a key for edge attributes, provided the edge has only a single value. An example in Figure 2 is the *like* edge between *4* and *2*. Ideally, all the outgoing (or incoming) edges of a vertex, will be stored in the same row, where the EID_i, LBL_i, and VAL_i columns are used to store the connected edge id, edge label, and outgoing (or incoming) vertex id respectively, assuming that the hashing function hashes the edge label to the i^th column triad (in our example for *like* this would be the j^th triad. If there are collisions of hashing, the SPILL column will be set for the vertex to indicate multiple rows are required to represent the outgoing adjacency information of the vertex. In addition, if the vertex has multiple outgoing edges with the same label, the outgoing edge ids and outgoing vertices are stored in the OSA (or correspondingly ISA) tables, as shown in the figure for the edge between *1* and its edges to *2* and *4*. Obviously, the effectiveness of this approach to storing adjacency is dependent on minimizing hashing conflicts. Bornea et al. [5] show that hashing based on dataset characteristics is relatively robust if its based on a reprentative small sample of the data. However, if updates change substantially the basic characteristics of the dataset on which the hashing functions were derived, reorganization is required for efficient performance.

The vertex attribute (VA) table as in Figure 5 (e) directly uses JSON column to store vertex attributes. The separate table avoids redundant storage of the attributes, in case vertices span multiple rows. The edge attribute (EA) table not only stores the edge attributes in JSON column, but also keeps a copy of the adjacency information of each edge. We incorporated this feature because it provides significant benefits on certain types of graph queries, as we discuss in the next section. Furthermore, as we discuss in the evaluation section, this redundancy does not actually result in greater storage costs on disk compared to existing systems, because most relational engines have very good compression schemes.

In addition, for VA and EA tables, the vertex and edge ids are used as the primary keys. For the other tables, we built indexes over the VID and VALID columns, to support efficient table joins by using the ids. We also added the equivalent of a combined index on INV and LBL, as well as OUTV and LBL (these are effectively the equivalent of SP and OP indexes in RDF triple stores). In addition, depending on the workloads of the property graph stores, more relational and JSON indexes can be built to accelerate specific query types or graph operations, which is similar to the functionality provided by most property graph stores.

3.5 Uses for Redundancy in the Schema

One weakness in the proposed schema for the storage of adjacency lists is that it always requires a join between the OSA and OPA (or correspondingly IPA and ISA) tables to find the immediate neighbors of a vertex. In cases where the result set is large in either table, this can be an expensive

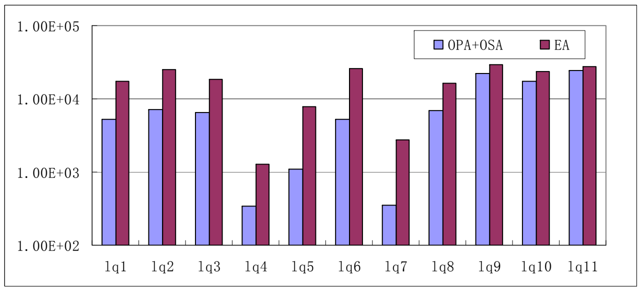


Figure 6: Results of the EA versus OPA-OSA path computation in ms.

operation compared to an index lookup in EA. We tested this hypothesis with another micro-benchmark on DBpedia. The query was to find all neighbors of a given vertex. We varied the selectivity of that query by choosing vertices with a small number of incoming edges, or a large number of incoming edges, as shown in Table 4. As shown in the table, a simple lookup of a vertex's neighbors can degrade if the query is not selective for the adjacency tables, compared to a lookup in EA.

The next question to ask is whether we need the adjacency tables at all? After all, the EA table contains adjacency information as well (it is basically a triple table) and can be used to compute paths. We therefore ran our long path queries using joins on the EA table alone, or using joins on OPA+OSA. The results shown in Figure 6 were once again unequivocal. On average, queries for paths were performed in 8.8 s when OPA+OSA were used to the answer the query compared to 17.8 s when EA was used. OPA+OSA was also somewhat less variable overall (a standard deviation of 8.2 s compared to 9.8 s for EA). The reason for this finding is due to the fact that shredding results in a much more compact table than a typical vertical representation. Thus, the cardinality of the tables involved in the joins is smaller, and it yields better results for path queries. We note that this complements [5]'s work which shows the advantages of shredding for so-called star queries, which are also very common in graph query workloads.

Query ID	ResultSize	EA	IPA+ISA
1	1	38	39
2	21	38	38
3	228	39	40
4	2282	39	40
5	21156	41	42
6	226720	58	77
7	2350906	74	440

Table 4: Comparison of getting vertex neighbors query performance in ms.

4. QUERY TRANSLATION

Gremlin, the de facto standard property graph query traversal language [37], is procedural, and this makes it difficult to compile it into a declarative query language like SQL. Nevertheless, Gremlin is frequently used to express graph traversal or graph update operations alone, and these can be mapped to declarative languages. In this paper, we focus on graph traversal queries and graph update operations with no side effects.

4.1 Gremlin Query Language in a Nutshell

A Gremlin query consists of a sequence of steps, called pipes. The evaluation of a pipe takes as input an iterator over some objects and yields a new iterator. Table 5 shows the different categories of Gremlin operations (or pipes).

Gremlin Operation Types	Description
Transform	Take an object and emit a transformation of it. Examples: both(), inE(), outV(), path()
Filter	Decide whether to allow an object to pass. Examples: has(), except(), simplePath()
Side Effect	Pass the object, but with some kind of side effect while passing it. Examples: aggregate(), groupBy(), as()
Branch	Decide which step to take. Examples: split/merge, ifThenElse, loop

Table 5: Operations supported by Gremlin query language.

The interested reader is referred to [36] for an exhaustive presentation of all Gremlin pipes. Here, we illustrate, on a simple example, the standard evaluation of a Gremlin query. The following Gremlin query counts the number of distinct vertices with an edge to or from at least one vertex that has 'w' as the value of its 'tag' attribute:

g.V.filter{it.tag=='w'}.both.dedup().count()

The first pipe of the query V returns an iterator it_1 over all the vertices in the graph g. The next pipe *filter*{*it.tag* == *'w'*} takes as input the iterator it_1 over all vertices in the graph, and yields a new iterator it_2 that retains only vertices with 'w' as the value of their 'tag' attribute. The *both* pipe then takes as input the iterator it_2 and returns an iterator it_3 containing, for each vertex v in it_2, all vertices u such that the edge (v, u) or (u, v) is in the graph g (note that it_3 may contain duplicated values). The *dedup()* produces an iterator it_4 over unique values appearing in the iterator it_3. Finally, the last pipe *count()* returns a iterator with a single value corresponding to the number of elements in the previous iterator it_4.

4.2 Query Processing Framework

Since Gremlin operates over any graph database that supports the basic set of primitive CRUD (Create Read Update Delete) graph operations defined by the Blueprints APIs [35], a straightforward way to support Gremlin queries is to implement the Blueprints APIs over the proposed schema, as most of the existing property graph stores do. However, this approach results in a huge number of generated SQL queries for a single Gremlin query, and multiple trips between the client code and the graph database server, which leads to significant performance issues when they are not running in the same process on the same machine. For instance, for the example query in the previous section, for each vertex v returned by the pipe *filter*{*it.tag* == *'w'*}, the Blueprints' method $getVertices(Direction.BOTH)$ will be invoked on v to get all its adjacent vertices in both directions, which will result in the evaluation, on the graph database server, of a SQL query retrieving all the vertices that have an edge to or from v.

Compared to random file system access or key-value lookups, SQL query engines are more optimized for set operations rather than for multiple key lookups. Hence, the basic idea of our query processing method is to convert a Gremlin query into a single SQL query. By doing so, we not only eliminate the chatty protocol between the client and the database server, but we also leverage multiple decades of query optimization research and development work that have gone into mature relational database management systems. In other words, by specifying the intent of the graph traversal in one shot as a declarative query, we can leverage the database engine's query optimizer to perform the query in an efficient manner.

The proposed query framework follows the following steps. The input Gremlin query is first parsed into an execution pipeline that is composed of a set of ordered Gremlin operations (i.e., pipes). The pipes are then sent to the query builder, where a set of pre-defined templates, which are of different types including SQL functions, user defined functions (UDFs), common table expression (CTE) fragments and stored procedures (SPs), are used to translate the pipes into SQL queries. Based on the order of the input pipes, the matched templates are composed together and optimized by the query optimizer. Finally, input Gremlin queries are converted into a single SQL query or stored procedure call to send to the relational engine for execution.

4.3 Gremlin Query Translation

In the standard implementation of Gremlin, the input or output of a pipe is an iterator over some elements. In our SQL based implementation, the input or output of a pipe is a table (a materialized table or a named Common Table Expression (CTE)) with a mandatory column named *val* that contains the input or output objects, and an optional column named *path* that represents the traversal path for each element in the *val* column (this path information is required by some pipes such as simplePath or path).

Definition 1. *We define the translation of a gremlin pipe e, denoted* $[e]$*, as a function that maps the input table* t_{in} *of the pipe to a tuple of* (sp, spi, cte, t_{out})*, where*

- t_{out} *(also denoted* $[e].out$*) is the result table of the pipe.*
- *sp (also denoted* $[e].sp$*) is the list of stored procedure definitions used in the translation of e.*
- *spi (also denoted* $[e].spi$*) is the list of stored procedure invocations for a subset of stored procedures in sp.*
- *cte (also denoted* $[e].cte$*) is the list of pairs* $(cteName, cteDef)$ *consisting of the name and the definition of Common Table Expressions (CTEs) used in the translation of e.*

If the translation is done through CTEs, then t_{out} is the name of one of the CTEs in *cte*; otherwise, it is the name of a temporary table created and populated by the invocation of the last element of *spi*.

Table 5 lists the Gremlin operations in different categories. Basically, for the different types of Gremlin operations (pipes), we designed different types of query templates to handle each Gremlin pipe based on operations that are standard in relational databases.

Transform Pipes. The transform pipes control the traversal between the the vertices in a graph. Based on results discussed in section 3.5 on how to best exploit the redundancy in the schema design section, for a transform from a set of vertices to their adjacent vertices, if the transform appears as the only graph traversal step in the query (i.e., for a simple look-up query), the most efficient translation, in general, uses the edge table (EA); otherwise, the translated CTE template joins with the hash adjacency tables. For example, the *out* pipe, which outputs the set of adjacent vertices of each input vertex, is translated by the following template parametrized by the input table t_{in} if the pipe is part of a multi-step traversal query:

```
[out](t_in)=(∅, ∅, cte, t_1)
cte = {
(t_0,SELECT t.val FROM t_in v,OPA p,
   TABLES(VALUES(p.val_0), ... ,(p.val_n))AS t(val)
   WHERE v.val=p.entry AND t.val is not null),
(t_1,SELECT COALESCE(s.val, p.val)  AS  val
    FROM  t_0 p LEFT OUTER JOIN OSA s on p.val=s.id)}
```

Otherwise, if the *out* pipe is the only graph traversal step in the query, the preferred translation uses the edge table (EA) as follows:

```
[out](t_in)=(∅, ∅, cte, t_0)
cte = {(t_0,SELECT p.outv AS val  FROM t_in v, EA p
   WHERE v.val=p.inv )}
```

A more complex transform pipe is the path pipe, which returns the traversal path of each input object (i.e., the path of each object through the pipeline up to this point). For illustration, let us consider the following labeled graph $g = (V = \{1, 2, 3, 4\}, E = \{(1, p, 2), (2, q, 3), (2, r, 4)\})$, the gremlin query $q_1 = g.V(1).out.out$ returns the vertices two hops aways from 1, namely 3 and 4. If we add the path pipe at the end of the previous query, the resulting query $q_2 = g.V(1).out.out.path$ evaluates to the actual traversal path of each result of query q_1 (i.e., a sequence of steps from 1 to a result of query q_1). The result of the evaluation of q_2 consists of the two sequences [1, 2, 3] and [1, 2, 4]. Thus, path pipe requires the system to record the paths of the traversal. Hence, if a path pipe p is present in a query, the additional *path* column has to be added to the CTE templates used to translate all pipes appearing before p to track the path of all output object. The translation of a pipe e that keeps track of the path of each object is denoted $[e]_p$. $[e]_p$ is similar to $[e]$ except that it assumes that the input table t_{in} has a column called *path* and it produces an output table t_{out} with a column named *path* for storing the updated path information. For example, when path information tracking is enabled, the *out* pipe is translated by the following template parametrized by the input table t_{in} (assuming the pipe is part of a multiple step traversal query):

```
[out]_p(t_in)=(∅, ∅, cte, t_1)
cte = {
(t_0,SELECT t.val AS val, (v.path || v.val) AS path
   FROM t_in v, OPA p,
   TABLES(VALUES(p.val_0), ... ,(p.val_n))AS t(val)
   WHERE v.val=p.entry AND t.val is not null),
(t_1,SELECT COALESCE(s.val, p.val)  AS  val, p.path
    FROM  t_0 p LEFT OUTER JOIN OSA s on p.val=s.id)}
```

Filter Pipes. The filter pipes typically filter out unrelated vertices or edges by attribute lookup. Hence, the corresponding CTE templates can simply apply equivalent SQL conditions on JSON attribute table lookup. For the filter conditions not supported by default SQL functions, such as the simplePath() pipe, we define UDFs to enable the filter condition translation.

Side Effect Pipes. Side effect pipes do not change the input graph elements, but generate additional information based on the input. In our current implementation, side effects are ignored, so side effect pipes act as identity functions (i.e., their output is identical to their input).

Branch Pipes. The branch pipes control the execution flow of other pipes. For split/merge pipes and ifElseThen()

pipes, we can simply use CTEs to represent all the possible branches, and use condition filters to get the desired branch.

For example, for a given input table t_{in} and an ifThenElse pipe $e = ifThenElse\{e_{test}\}\{e_{then}\}\{e_{else}\}$, we first translate the test expression e_{test} as a transform expression that yields a boolean value, and we also track provenance information in the path column. Let *test* be the result of the translation: $test = [e_{test}]_p(t_{in})$. Using the output table of *test* (i.e., *test.out*), we then define the CTE $thencte_{in}$ (resp. $elsecte_{in}$) corresponding to the input table for the evaluation of e_{then} (resp. e_{else}):

```
thencte_in=(then_in,SELECT path[0] AS val FROM test.out
   WHERE val=true)
elsecte_in=(else_in,SELECT path[0] AS val FROM test.out
   WHERE val=false)
```

The translation of the ifThenElse expresion e for the input table t_{in} can now be defined by collecting all the stored procedure definitions and invocations, and CTEs produced by the translations of 1) the test condition ($test = [e_{test}]_p(t_{in})$), 2) the then part ($then = [e_{then}](then_{in})$), and 3) the else part ($else = [e_{else}](else_{in})$):

$$\begin{aligned}
[e](t_{in}) &= (sp, spi, cte, t_{out}) \\
sp &= test.sp \cup then.sp \cup else.sp \\
spi &= test.spi \cup then.spi \cup else.spi \\
cte &= test.cte \cup \{thencte_{in}, elsecte_{in}\} \\
&\quad \cup then.cte \cup else.cte \cup \{(t_{out}, \\
&\quad \textbf{SELECT * FROM } then.out \\
&\quad \textbf{UNION ALL SELECT * FROM } else.out)\}
\end{aligned}$$

The result table t_{out} is simply defined as the union of results from the then part and else part.

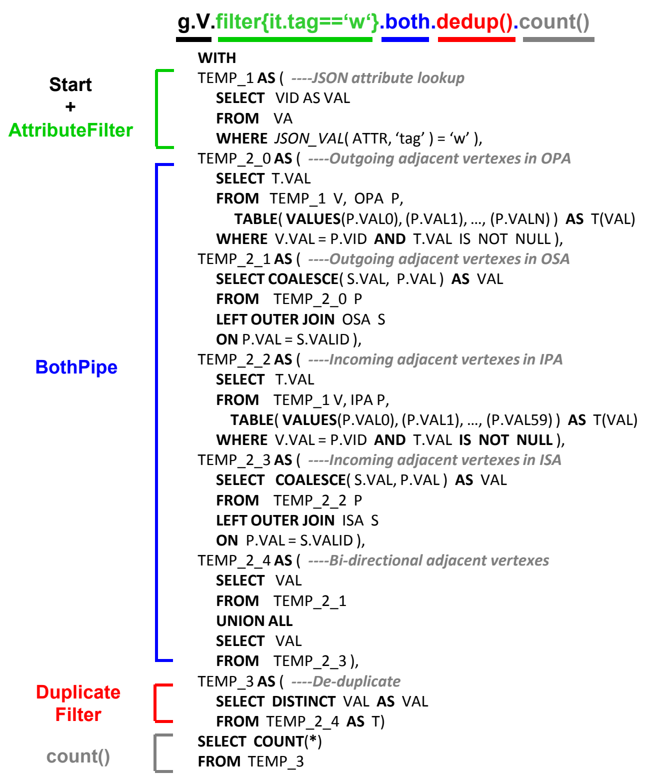
For loop pipes, we evaluate the depth of the loop. For fixed-depth loops, we will directly expand the loop and translate it into CTEs. Otherwise, we translate the loop pipe into a recursive SQL or a stored procedure call, depending on the engine's efficiency in handling recursive SQL.

Figure 7 gives an example of using CTEs to translate our sample Gremlin query.

4.4 Limitations

Our focus in this paper is on graph traversal queries and graph update operations with no side effects. Specifically, side effect pipes are ignored (i.e. they are currently implemented as identity functions: their output is identical to their input). Likewise, pipes with complex Groovy/Java closures or expressions are also ignored because we currently do not perform a static analysis of closures and expressions to allow us to understand, for example, whether a Java method call in a closure or an expression has side effects (e.g., such an expression can appear as the stopping condition in a loop pipe or as the test condition in an IfThenElse pipe). As a result, we currently conservatively ignore pipes containing any expression other than simple arithmetic or comparison operators.

Modulo the limitations outlined in the previous paragraph, our translation process is fairly generic and produces CTEs as the result of the translation of most pipes (see Table 8 in the Appendix for the translation in detail). Stored procedures are only used as fallback option in the translation of recursive pipes when the depth of the loop cannot be statically determined.



```
WITH
TEMP_1 AS (  ----JSON attribute lookup
   SELECT  VID AS VAL
   FROM   VA
   WHERE JSON_VAL( ATTR, 'tag' ) = 'w' ),
TEMP_2_0 AS (  ----Outgoing adjacent vertexes in OPA
   SELECT T.VAL
   FROM  TEMP_1 V,  OPA  P,
      TABLE( VALUES(P.VAL0), (P.VAL1), …, (P.VALN) )  AS  T(VAL)
   WHERE V.VAL = P.VID  AND  T.VAL  IS  NOT  NULL ),
TEMP_2_1 AS (  ----Outgoing adjacent vertexes in OSA
   SELECT COALESCE( S.VAL,  P.VAL )  AS  VAL
   FROM   TEMP_2_0  P
   LEFT OUTER JOIN  OSA  S
   ON P.VAL = S.VALID ),
TEMP_2_2 AS (  ----Incoming adjacent vertexes in IPA
   SELECT  T.VAL
   FROM   TEMP_1 V, IPA P,
     TABLE( VALUES(P.VAL0), (P.VAL1), …, (P.VAL59) )  AS  T(VAL)
   WHERE V.VAL = P.VID  AND  T.VAL  IS  NOT  NULL ),
TEMP_2_3 AS (  ----Incoming adjacent vertexes in ISA
   SELECT  COALESCE( S.VAL, P.VAL )  AS  VAL
   FROM   TEMP_2_2  P
   LEFT OUTER JOIN  ISA  S
   ON  P.VAL = S.VALID ),
TEMP_2_4 AS (  ----Bi-directional adjacent vertexes
   SELECT  VAL
   FROM   TEMP_2_1
   UNION ALL
   SELECT  VAL
   FROM   TEMP_2_3 ),
TEMP_3 AS (  ----De-duplicate
   SELECT  DISTINCT  VAL  AS  VAL
   FROM TEMP_2_4  AS  T)
SELECT  COUNT(*)
FROM TEMP_3
```

Figure 7: An example of Gremlin query translation.

4.5 Optimization

In this section, we describe optimizations applied to the query translation and evaluation steps as well as optimizations needed for efficient update operations.

4.5.1 *Query Translation and Evaluation Optimization*

A standard Gremlin query rewrite optimization technique in most property graph stores consists of replacing a sequence of the non selective pipe $g.V$ (retrieve all vertices in g) or $g.E$ (retrieve all edges in g) followed by a sequence of attribute based filter pipes (i.e., filter pipes that select only vertices or edges having specific edge labels, attribute names, or attribute name/value pairs) by a single GraphQuery pipe that combines the non selective pipe $g.V$ or $g.E$ with the potentially more selective filter pipes. A similar rewrite is done to replace a sequence of the potentially non selective pipe *out*, *outE*, *in*, or *inE* followed by a sequence of attribute based filter pipes by a single VertexQuery pipe. This VertexQuery rewrite is particularly efficient for the processing of supernodes (i.e., vertices with large number connections to other vertices). GraphQuery and VertexQuery rewrites allow for a more efficient retrieval of only relevant data by the underlying graph database (e.g., by leveraging indexes on particular attributes). We exploit such merging in our translation as shown in Figure 7, where the non selective first pipe $g.V$ is explicitly merged with the more selective filter $filter\{it.tag == 'w'\}$ in the translation. To some extent, our translation of the whole Gremlin query into a single SQL generalizes the basic idea embodied in GraphQuery and VertexQuery pipes: providing more information about the query to the underlying graph database to enable a more efficient evaluation. However, as opposed to our approach that compiles the whole Gremlin query into a single SQL query, GraphQuery and VertexQuery do not go far enough: they are limited to efficient filtering of a single traversal step (in our example query, the rewrite optimization will yield g.queryGraph(filter{it.tag==‘w’}, V).both.dedup().count()).

In the current implementation of our approach, we rely on the underlying relational database management system to provide the best evaluation strategy for the generated SQL query.

4.5.2 Graph Update Optimization

Basic graph update operations, including addition, update, and deletion of vertices and edges are implemented by a set of stored procedures. For our schema, this is especially important because graph data are stored into multiple tables, and some of these operations involve updates to multiple tables. Furthermore, some update operations, such as the deletion of a single supernode of the graph, can result in changes involving multiple rows in multiple tables, which can significantly degrade performance.

To address this issue, we optimized vertex deletions by setting the ID of the vertices and edges to be deleted to a negative value corresponding to its current ID. To delete a vertex with $ID = i$, we set its VID to $-i - 1$ in the vertex attribute and hash adjacency tables, so the relations of deleted rows are maintained across tables. Corresponding rows in the edge attribute tables are deleted. As a result of this update optimization, we add to each query the additional condition $VID \geq 0$ to ensure that vertices marked for deletion are never returned as answers to a query. An off-line cleanup process can perform the actual removal of the marked vertices, but this is not currently implemented in the system yet.

5. EVALUATION

Our goal in this section was to compare the performance of SQLGraph against two popular open source property graph systems, Titan and Neo4j. To keep systems comparable in terms of features, we focused on a comparison of property graph systems with full ACID support, targeting a single node in terms of scalability. Titan uses existing noSQL stores such as BerkeleyDB, Cassandra and HBase to support graph storage and retrieval, but the latter two backends focus on distributed storage, and do not provide ACID support. We therefore examined Titan with the BerkeleyDB configuration which targets single server, ACID compliant workloads[9]. Neo4j provides native storage for graphs, is fully ACID compliant, and is not based on any existing noSQL stores. We compared the efficacy of our schema and query translation for property graphs by comparing them with these two popular systems for property graphs. The fact that Titan and Neo4j are focused on rather different architectures was an important factor in our choice of these systems for the evaluation. In addition, we also tried to include the document-based graph store OrientDB in our comparisons, but encountered problems in data loading and concurrency support. We include a discussion of the performance of OrientDB where we could.

[9]Titan-BerkeleyDB is also known to have better performance than Titan with distributed store back-ends [17]

There are no explicit benchmarks targeted for Gremlin over property graphs yet. We therefore converted two different graph benchmarks into their property graph equivalents, as described below. We tried to vary the type of workload significantly in our choices for the two benchmarks. As in the micro benchmarks, our first choice was DBpedia, a benchmark which reflects structured knowledge from Wikipedia, as well as extractions from it. The structured knowledge is modeled as RDF, with additional metadata about where the knowledge was extracted from being represented as attributes of quads in RDF. Our second choice was LinkBench, a synthetic benchmark developed to simulate the social graph at Facebook [3].

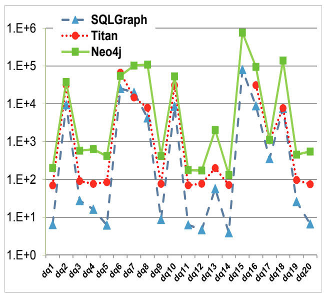
To evaluate the performance of the different property graph stores on existing commodity single-node servers, we conducted our experiments on 4 identical virtual machines (one per system), each with 6-core 2.86GHz CPU, 24GB memory and 1TB storage running 64-bit Linux. All three property graph stores were running in server mode and responding to the requests from clients at localhost. We used a commercial relational engine to implement SQLGraph, and compared it to Neo4j 1.9.4 with its native http server as well as Titan 0.4.0 and OrientDB 1.7.8 with the Rexster http server 2.4.0. During testing, we assigned the same amount of memory (10G except in experiments that manipulated it explicitly) for the graph stores, and used their recommended configurations for best performance (if no recommended parameters found, we used the default ones). In addition, in our LinkBench testing, the largest dataset of billion-node graphs was beyond the storage capacity of the above servers. We conducted that experiment on a virtual machine instance equivalent to an Amazon EC2 hs1.8xlarge machine with 16-core CPU, 117GB memory, and 48TB storage to test it over the three graph stores. We report those experiments separately since they were conducted on different hardware.

5.1 DBPedia

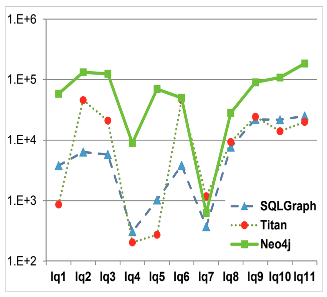
For the DBPedia benchmark, we converted DBPedia 3.8 dataset to a property graph as described in Section 3.1. Two query sets were used to evaluate query performance of our graph store compared with Neo4j and Titan-BerkeleyDB. For the first query set, we converted the SPARQL query set used in [22] into Gremlin queries as described in Appendix B, and compared performance for the three graph stores[10]. The results are shown in Figure 8a, and these are separated from the results for the path queries in figure 8b. As shown in the figure 8a, Titan timed out on query 15 of the DBpedia SPARQL benchmark. We therefore provide two means for each system, one overall mean for the 20 benchmark queries (which for Titan excluded query 15), and an adjusted mean, excluding the times from query 15 for all three systems in figure 8d to allow a more fair comparison.

For the second query set, we used the 11 long-path queries described in Section 3.1 to examine graph traversal over the same dataset. This is a fairly common requirement in graph workloads, but it is not part of the current DBpedia SPARQL benchmark. The results are shown in Figure 8b. As can be seen from figure 8d, SQLGraph achieves the best performance for both path queries and benchmark

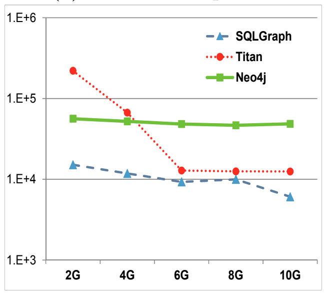
[10]All the queries in this benchmark and in LinkBench and their corresponding translations to Gremlin are available at http://researcher.ibm.com/researcher/files/us-ksrinivs/supp-queries.pdf



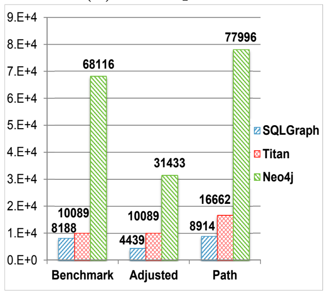
(a) Benchmark queries



(b) Path queries

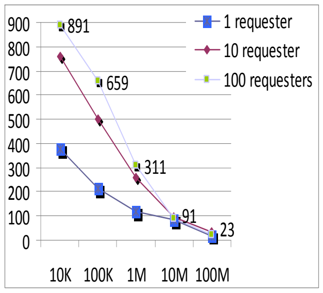


(c) Varying memory usage effects

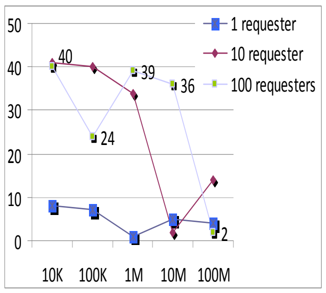


(d) DBpedia performance summary

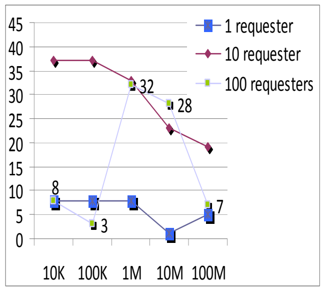
Figure 8: DBpedia benchmark performance in ms.



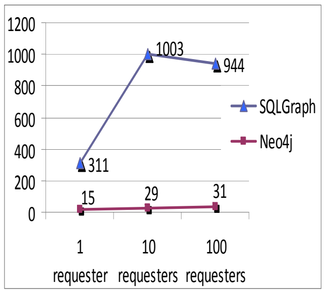
(a) SQLGraph



(b) Titan



(c) Neo4j



(d) 1 Billion node graph

Figure 9: LinkBench workload performance in op/sec.

queries. Specifically, SQLGraph is approximately 2X faster than Titan-BerkeleyDB, and about 8X faster than Neo4j. SQLGraph was also less variable in terms of its query performance; the standard deviation for SQLGraph for the benchmark queries excluding query 15 was 8.3 s, but Titan had a standard deviation of 17.5 s, and Neo4j had a standard deviation of 54.6 s. Path queries showed a similar trend, SQLGraph had a standard deviation of 9.4 s for path queries, Titan had 17.0 s, and Neo4j had 56.7 s.

In addition, we evaluated the performance of the different graph stores while varying the amount of memory available to the system. Our key objective here was to just ensure that the systems we were comparing against were not limited by available resources in the system. Figure 8c shows the comparison results for the 2,4,6,8, and 10 GB memory settings. In plotting the results, we computed the average times for each system across all queries in DBpedia (both the benchmark queries and the path queries), and we omitted query 15 from all systems so they could be compared. As shown in the figure, neither Titan nor Neo4j were showing any perceptible performance benefits when memory increased beyond 8G. We would like to note however, that Titan in particular has some rather aggressive caching strategies, compared to other systems. For instance, on the 10G memory usage case, Titan's mean (excluding query 15) for all the DBpedia queries was 380.1 s on the first run (cold cache), compared to 12.9 s on run 10. SQL graph's corresponding numbers for the first run was 24.1 s compared to 6.2 s on the tenth run, and Neo4j's average was 129.0 s on the first run compared to 55 s on the tenth.

One additional point to make is about the sizes of the database on disk. We pointed out earlier that our schema is redundant, in the sense that we store adjacency information in relational storage and store a copy of it in EA for edge specific queries. We also measured the sizes of the DBpedia dataset on disk for the 3 engines. DBpedia's size on disk for SQLGraph was 66GB, 98GB for Neo4j and 301GB for Titan. For OrientDB, we failed to load the DBPedia dataset after trying various configurations. The loading speed of OrientDB for such large graphs is extremely slow and it seems OrientDB cannot well support URIs as edge labels and property keys.

5.2 LinkBench

LinkBench [3] is a benchmark that simulates a typical social graph workload involving frequent graph updates. While its data model was not specified as a property graph, it can be directly transformed to a property graph by mapping "objects" into graph vertices with vertex attributes *type*, *version*, *update time*, and *data*, and mapping "associations" into graph edges with edge attributes *association type*, *visibility*, *timestamp*, and *data*.

To evaluate the performance of primitive graph CRUD operations, we adapted LinkBench to support the property graph CRUD operations in Gremlin language. The approximate distribution of CRUD operations reflects the distribution described in [3], as shown in Table 6.

We first generated 5 datasets with different scales, with the number of vertices ranging from 10 thousand to 100 million. Figure 9 shows the results of SQLGraph, Neo4j and Titan under different concurrency settings with LinkBench. In the figure, data values are only added for the 100 requesters case for clarity. As can be seen from the figure, SQLGraph's concurrency is much better than the other two graph stores, and it seems to be much less variable than the other two stores. For OrientDB, the throughput ranges from 2 to 9 op/sec for the 1-requester case, which is comparable to Neo4j and Titan. However, for the 10-requester and 100-requester settings, OrientDB reported concurrent update errors due to the lack of built-in locks. We also tested the scalability of these systems against a much larger graph dataset (1 billion nodes, 4.3 billion edges), but that evaluation could not be performed on the same VM configuration because of the scale of the graph. Furthermore, we could not test Titan on the 1 billion node graph because the queries timed out even on a superior VM, and we could not determine the reasons for its failure. We therefore show the results for the 1 billion node graph in a separate panel (d) of the figure. Still, compared with Neo4j, on the billion node graph, SQLGraph showed about 30 times better throughput.

Table 6 shows the distribution of LinkBench operations, and the average performance (as well as the maximum values) for each operation in seconds for the three systems. Note that we do not show the 100 requester case because we tried to find the point where each system was at its maximal performance. This was 10 requesters for all three systems on the 100 million node dataset. As can be seen from the table, SQLGraph is slower than the other two systems on the delete node, add link and update link operations on average, but its maximum time is still well within the maximum time of other systems. These operations constitute about 5% of the overall LinkBench workload. Thus, on write-heavy workloads, SQLGraph may have inferior update performance than Titan and Neo4j on average, because the update affects multiple underlying tables rather than a single operation. However when the dataset gets substantially bigger, one does not observe this behavior. Table 7 shows the same operations on the 1 billion node benchmark for the SQLGraph and Neo4j systems, now with 100 requesters since this was where Neo4j showed maximal performance. As can be seen in the table, SQLGraph now outperforms Neo4j on *all* operations on LinkBench.

Operation	Query Disbn	SQL-Graph	Titan	Neo4j
add node	2.6%	0.04(0.47)	0.30(3.34)	0.96(6.13)
update node	7.4%	0.12(1.75)	0.56(10.25)	1.12(6.87)
delete node	1.0%	2.24(5.81)	0.98(7.45)	0.73(4.06)
get node	12.9%	0.06(1.75)	0.51(9.18)	0.17(4.47)
add link	9.0%	1.68(4.83)	0.72(6.40)	1.02(6.30)
delete link	3.0%	0.12(1.54)	0.63(5.95)	0.28(4.50)
update link	8.0%	1.38(4.70)	0.72(9.29)	1.02(6.33)
count link	4.9%	0.03(0.53)	0.52(7.54)	0.22(5.31)
multiget link	0.5%	0.04(0.15)	0.44(1.27)	0.19(0.48)
get link list	50.7%	0.03(18.80)	0.58(369.30)	0.28(255.19)

Table 6: LinkBench operation distribution, and average performance in seconds for the 100M node graph, 10 requesters case. Maximum times are provided in parentheses.

As in the case of DBpedia, we measured the size of the largest 1B dataset for LinkBench, and found that SQLGraph took 850GB on disk, Neo4j took 1.1TB, and Titan took 1.7TB on disk. Similar to the DBPedia case, we failed to load largest LinkBench dataset to OrientDB. For the smaller datasets, OrientDB's footprint is between Neo4j and Titan.

6. FUTURE WORK AND CONCLUSIONS

We have shown that by leveraging both relational and non-relational storage in a relational database, along with very basic optimization techniques it is possible for relational systems to outperform graph stores built on key value stores or specialized data structures in the file system. More optimization opportunities exist, especially for Gremlin queries

Operation	SQLGraph	Neo4j
add node	0.002(0.160)	1.643(6.586)
update node	0.003(0.241)	2.429(490.562)
delete node	0.422(0.586)	4.527(2059.552)
get node	0.002(0.093)	1.453(7.259)
add link	0.405(0.723)	2.435(786.816)
delete link	0.003(0.152)	2.297(1273.229)
update link	0.307(0.704)	2.677(1132.209)
count link	0.002(0.266)	1.505(7.128)
multiget link	0.004(0.013)	2.387(1165.061)
get link list	0.002(1.297)	1.651(684.347)

Table 7: LinkBench operation distribution, and average performance in seconds for the 1B node graph, 100 requesters case. Maximum times are provided in parentheses.

without any side effects, if one builds a better compiler for the query language. This will be a focus for our future work.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev, and I. Toma. The linked data benchmark council: A graph and RDF industry benchmarking effort. *SIGMOD Rec.*, 43(1):27–31, May 2014.

[3] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, New York, NY, USA, 2013. ACM.

[4] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *International Journal On Semantic Web and Information Systems*, 2009.

[5] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2013.

[6] F. Bugiotti, F. Goasdoué, Z. Kaoudi, and I. Manolescu. RDF data management in the Amazon cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[7] A. Chebotko, S. Lu, and F. Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data and Knowledge Engineering*, 68(10):973 – 1000, 2009.

[8] M. Ciglan, A. Averbuch, and L. Hluchy. Benchmarking traversal operations over graph databases. In *28th International Conference on Data Engineering Workshops (ICDEW)*, pages 186–189. IEEE, 2012.

[9] R. Cyganiak. A relational algebra for SPARQL. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, page 35, 2005.

[10] F. Di Pinto, D. Lembo, M. Lenzerini, R. Mancini, A. Poggi, R. Rosati, M. Ruzzi, and D. F. Savo.

Optimizing query rewriting in ontology-based data access. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 561–572, New York, NY, USA, 2013. ACM.

[11] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J.-L. Larriba-Pey. Survey of graph database performance on the HPC scalable graph analysis benchmark. In *Web-Age Information Management*, pages 37–48. Springer, 2010.

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

[13] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. In *Web Information Systems Engineering–WISE 2005 Workshops*, pages 235–244. Springer, 2005.

[14] O. Hartig and B. Thompson. Foundations of an alternative approach to reification in RDF. *CoRR*, abs/1406.3399, 2014.

[15] F. Holzschuher and R. Peinl. Performance of graph query languages: Comparison of Cypher, Gremlin and native access in Neo4J. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204, New York, NY, USA, 2013. ACM.

[16] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[17] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *SocialCom*, pages 708–715. IEEE, 2013.

[18] Z. Kaoudi and I. Manolescu. Cloud-based RDF data management. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 725–729, New York, NY, USA, 2014. ACM.

[19] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete OWL ontology benchmark. In *Proceedings of the 3rd European Conference on The Semantic Web*, ESWC'06, pages 125–139, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] P. Macko, D. Margo, and M. Seltzer. Performance introspection of graph databases. In *Proceedings of the 6th International Systems and Storage Conference*, page 18. ACM, 2013.

[21] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey. DEX: High-performance exploration on large graphs for information retrieval. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 573–582, New York, NY, USA, 2007. ACM.

[22] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL benchmark–performance assessment with real queries on real data. In *The Semantic Web–ISWC 2011*, pages 454–469. Springer, 2011.

[23] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

[24] T. Neumann and G. Weikum. RDF-3X: A RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, Aug. 2008.

[25] T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB Endow.*, 3(1-2):256–263, Sept. 2010.

[26] K. Nitta and I. Savnik. Survey of RDF storage managers. In *DBKDA 2014, The Sixth International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 148–153, 2014.

[27] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2RDF: Adaptive query processing on RDF data in the cloud. In *Proceedings of the 21st International Conference Companion on World Wide Web*, WWW '12 Companion, pages 397–400, New York, NY, USA, 2012. ACM.

[28] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H2RDF+: an efficient data management system for big RDF graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2014, Snowbird, Utah, USA on June 22-27, 2014.* ACM, 2014.

[29] M. Rodríguez-Muro, R. Kontchakov, and M. Zakharyaschev. Ontology-based data access: Ontop of databases. In *International Semantic Web Conference, ISWC 2013*, pages 558–573. Springer, 2013.

[30] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, S. Auer, J. Sequeda, and A. Ezzat. A survey of current approaches for mapping of relational databases to RDF. *W3C RDB2RDF XG Incubator Report*, 2009.

[31] S. Sakr and G. Al-Naymat. Relational processing of RDF queries: a survey. *ACM SIGMOD Record*, 38(4):23–28, 2010.

[32] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: a SPARQL performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.

[33] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[34] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[35] Tinkerpop. Blueprints. Available: https://github.com/tinkerpop/blueprints/wiki, 2014.

[36] Tinkerpop. Gremlin pipes. Available: https://github.com/tinkerpop/pipes/wiki, 2014.

[37] Tinkerpop. Gremlin query language. Available: https://github.com/tinkerpop/gremlin/wiki, 2014.

[38] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Semantic Web and Databases Workshop*, pages 131–150, 2003.

[39] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: A fast and compact system for large scale RDF data. *Proc. VLDB Endow.*, 6(7):517–528, May 2013.

APPENDIX

A. GREMLIN TRANSLATION IN DETAIL

As described in section 4.3, Gremlin queries can be translated into SQL queries (CTEs) or stored procedure calls (SPs) based on a set of pre-defined templates. Table 8 gives a full list of Gremlin pipes that are currently supported by our query translator and the corresponding CTE templates. Here we only include the templates for the basic form of the pipes. Possible variations and combinations of the pipe translation have been discussed in section 4.

B. SPARQL TO GREMLIN QUERY CONVERSION

As described in section 5.1, our first DBPedia Gremlin query set was converted from the SPARQL queries used in [22]. Given the declarative nature of the SPARQL queries, we tried to implement each of the SPARQL query using graph traversals in Gremlin as efficient as possible. More specifically, we first identify the most selective URI as the Gremlin start pipe, and then use Gremlin transform pipes to implement the SPARQL triple patterns. Note that the traversal order of the transform pipes is also based on the selectivity of the different triple patterns. SPARQL filters are directly translated into Gremlin filter pipes, and SPARQL UNION operations are translated into Gremlin branch pipes if possible. In addition, the translated Gremlin queries will only return the size of the result set of the corresponding SPARQL queries, to minimize the result set composing and consumption differences of the different property graph stores.

Table 9 gives an example of SPARQL query to Gremlin query translation (dq2 in section 5.1). It can be seen that the URI *'http://dbpedia.org/ontology/Person'* is used in the start pipe to lookup the start vertex. Then the property filter pipe applies on the most selective literal *"Montreal Carabins"@en*. The out pipe and back pipe are used to traverse the graph. Two table pipes are used to store the results of the main branch and the optional branch. Ideally, the outer left join of the two tables generates the desired results of the original SPARQL query. However, due to the lack of built-in table operation support in Gremlin language, we directly return the sizes of the both tables in the translated Gremlin query.

Language	Query

SPARQL	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
	 PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
	 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
	 PREFIX dbpedia-prop: <http://dbpedia.org/property/>
	 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	 SELECT ?var4 ?var8 ?var10
	 WHERE {
	 ?var5 dbpedia-owl:thumbnail ?var4 ;
	 rdf:type dbpedia-owl:Person ;
	 rdfs:label "Montreal Carabins"@en ;
	 dbpedia-prop:pageurl ?var8 .
	 OPTIONAL { ?var5 foaf:homepage ?var10 . }
	 }

Gremlin	t1=new Table();
	 t2=new Table();
	 var5=[];
	 g.V("__URI__", "http://dbpedia.org/ontology/ Person").in('http://www.w3.org/1999/02/ 22-rdf-syntax-ns#type').has('http://www.w3.org/ 2000/01/rdf-schema#label','"Montreal Carabins" @en').aggregate(var5).as('var5').out('http:// dbpedia.org/ontology/thumbnail').as('var4').back(1) .out('http://dbpedia.org/property/pageurl').as('var8') .table(t1).iterate();
	 var5._().as('var5').out('http://xmlns.com/foaf/0.1/ homepage').as('var10').table(t2).iterate();
	 [t1.size(),t2.size()];

Table 9: An example of SPARQL to Gremlin query conversion.

Operation	CTE Template for Query Translation
out pipe	(t_0, SELECT t.val FROM $\underline{t_{in}}$ v,OPA p, TABLES(VALUES($p.val_0$), ... ,($p.val_n$))AS t(val) WHERE v.val=p.vid AND t.val is not null), (t_1,SELECT COALESCE(s.val, p.val) AS val FROM t_0 p LEFT OUTER JOIN OSA s on p.val=s.valid)
in pipe	(t_0, SELECT t.val FROM $\underline{t_{in}}$ v,IPA p, TABLES(VALUES($p.val_0$), ... ,($p.val_m$))AS t(val) WHERE v.val=p.vid AND t.val is not null), (t_1,SELECT COALESCE(s.val, p.val) AS val FROM t_0 p LEFT OUTER JOIN ISA s on p.val=s.valid)
both pipe	(t_0, SELECT t.val FROM $\underline{t_{in}}$ v,OPA p, TABLES(VALUES($p.val_0$), ... ,($p.val_n$))AS t(val) WHERE v.val=p.vid AND t.val is not null), (t_1, SELECT COALESCE(s.val, p.val) AS val FROM t_0 p LEFT OUTER JOIN OSA s on p.val=s.valid), (t_2, SELECT t.val FROM $\underline{t_{in}}$ v,IPA p, TABLES(VALUES($p.val_0$), ... ,($p.val_m$))AS t(val) WHERE v.val=p.vid AND t.val is not null), (t_3,SELECT COALESCE(s.val, p.val) AS val FROM t_2 p LEFT OUTER JOIN ISA s on p.val=s.valid), (t_4,SELECT * FROM t_1 UNION ALL SELECT * FROM t_3)
out vertex pipe	(t_0, SELECT p.outv AS val FROM $\underline{t_{in}}$ v, EA p WHERE v.val=p.eid)
in vertex pipe	(t_0, SELECT p.inv AS val FROM $\underline{t_{in}}$ v, EA p WHERE v.val=p.eid)
both vertex pipe	(t_0, SELECT t.val FROM $\underline{t_{in}}$ v, EA p, TABLES(VALUES($p.outv$), ($p.inv$)) AS t(val) WHERE v.val=p.eid)
out edges pipe	(t_0, SELECT p.eid AS val FROM $\underline{t_{in}}$ v, EA p WHERE v.val=p.outv)
in edges pipe	(t_0, SELECT p.eid AS val FROM $\underline{t_{in}}$ v, EA p WHERE v.val=p.inv)
both edges pipe	(t_0, SELECT p.eid AS val FROM $\underline{t_{in}}$ v, EA p WHERE v.val=p.outv OR v.val=p.inv)
range filter pipe	(t_0, SELECT * FROM $\underline{t_{in}}$ v LIMIT ? OFFSET ?)
duplicate filter pipe	(t_0, SELECT DISTINCT v.val FROM $\underline{t_{in}}$ v)
id filter pipe	(t_0, SELECT * FROM $\underline{t_{in}}$ v WHERE v.val == ?)
property filter pipe	(t_0, SELECT v.* FROM $\underline{t_{in}}$ v, VA p WHERE v.val=p.vid AND JSON_VAL(p.data, ?) == ?)
interval filter pipe	(t_0, SELECT v.* FROM $\underline{t_{in}}$ v, VA p WHERE v.val=p.vid AND JSON_VAL(p.data, ?) $>$? AND JSON_VAL(p.data, ?) $<$?)
label filter pipe	(t_0, SELECT v.* FROM $\underline{t_{in}}$ v, EA p WHERE v.val=p.eid AND p.lbl == ?)
except filter pipe	(t_0, SELECT * FROM $\underline{t_{in}}$ v WHERE v.val NOT IN (SELECT p.val FROM t_k p))
retain filter pipe	(t_0, SELECT * FROM $\underline{t_{in}}$ v WHERE v.val IN (SELECT p.val FROM t_k p))
cyclic path filter pipe	(t_0, SELECT * FROM $\underline{t_{in}}$ v WHERE isSimplePath(v.path) == 1)
back filter pipe	$meta.cte \cup$ (t_0, SELECT * FROM $\underline{t_{in}}$ v WHERE v.val IN (SELECT p.path[0] FROM $meta.t_{out}$ p))
and filter pipe	$meta1.cte \cup meta2.cte \cup$ (t_0, SELECT * FROM $\underline{t_{in}}$ v WHERE v.val IN (SELECT p.path[0] FROM $meta1.t_{out}$ p1 INTERSECT SELECT p.path[0] FROM $meta2.t_{out}$ p2))
or filter pipe	$meta1.cte \cup meta2.cte \cup$ (t_0, SELECT * FROM $\underline{t_{in}}$ v WHERE v.val IN (SELECT p.path[0] FROM $meta1.t_{out}$ p1 UNION SELECT p.path[0] FROM $meta2.t_{out}$ p2))
if-then-else pipe	$test.cte \cup \{thencte_{in}, elsecte_{in}\} \cup then.cte \cup else.cte \cup$ (t_{out}, SELECT * FROM then.out UNION ALL SELECT * FROM else.out)
split-merge pipe	$meta1.cte \cup meta2.cte \cup$ (t_0, SELECT * FROM $meta1.t_{out}$ p1 UNION ALL SELECT * FROM $meta2.t_{out}$ p2)
loop pipe	expand to fixed-length ctes or call stored procedure
as pipe	$\emptyset$, record the mapping between the as pipe and the (t_{out}) of the current translated ctes
aggregate pipe	$\emptyset$, record the mapping between the aggregate pipe and the (t_{out}) of the current translated ctes
add vertex	$\emptyset$, call stored procedure
add edge	$\emptyset$, call stored procedure
delete vertex	$\emptyset$, call stored procedure
delete edge	$\emptyset$, call stored procedure

Table 8: Currently supported Gremlin pipes and the translation method. "?" denotes the parameters of the pipes.