Graph Pattern Matching – Do We Have to Reinvent the Wheel?

Andrey Gubichev*
Technische Universität München
Germany
gubichev@in.tum.de

Manuel Then
Technische Universität München
Germany
then@in.tum.de

ABSTRACT

This paper presents an empirical study of how a wide spectrum of systems handle the graph pattern matching problem. Our approach is to take the well-known LUBM benchmark, model it across various domains (relational, RDF, property graph), and execute the benchmark queries on the corresponding systems. We evaluate the systems using a large data instance on a single machine (the largest dataset is LUBM-8000, which contains over 1 billion RDF triples). Additionally, we provide a brief analysis of how different cases of graph pattern matching problem are stressed by the benchmark queries. Our main finding is that, contrary to popular belief and various vendors' claims, modern native graph stores do not necessarily offer a competitive advantage over traditional relational and RDF stores, even for the graph-specific problem of pattern matching. To the best of our knowledge, this is the first independent empirical comparison of different approaches towards pattern matching performed on a large scale graph.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications

Keywords

Graph-Structured Data, Benchmarking, RDF

1. INTRODUCTION

As graph processing problems continue to attract the interest of both the research community and industry, the advent of seemingly novel data management approaches can be observed in the field of graph databases. Rapid development and adoption of new specialized graph databases may leave the impression that they substantially outperform existing solutions. However, the challenges of graph query processing often boil down to well-studied problems of general data management, such as query optimization and memory management.

In this paper we consider one of the most fundamental problems in graph processing — *pattern matching*. We show that this problem can be expressed in the different data models Resource Description Framework (RDF), property graphs (PG) and the relational model. Therefore, we also demonstrate the pattern matching problem's solvability in a wide range of systems. We provide an empirical study of five systems from these three domains, using the well-known semantic web benchmark LUBM. In order to do so, we model the LUBM dataset and a relevant subset of the benchmark's queries in these three domains, thus, giving every system the best chance to perform well.

Specifically, our contributions are as follows: *(i)* we demonstrate that LUBM queries provide a variety of challenging graph pattern matching problems, *(ii)* we describe how the graph-shaped data of LUBM can be stored and queried in a wide spectrum of data management domains, and *(iii)* we provide an extensive experimental study of LUBM on different systems. Although LUBM is a very popular benchmark in the Semantic Web area, so far there has been little studies of how non-RDF systems perform on it. We find particularly interesting the comparison between novel graph stores and classical relational database systems.

The rest of the paper is organized as follows. In Section 2 we briefly describe the LUBM benchmark which we will use for the empirical study. Then, in Section 3 we describe the general problem of graph pattern matching and consider its important subproblems that arise in LUBM queries. Section 4 reviews possible mappings of the LUBM benchmark into different data management domains (relational, RDF, property graph). This discussion is followed by a description of different systems that we used in our evaluation in Section 5. Section 7 references the related work. Finally, Section 6 presents the main contribution of this work, the empirical results of five different system on LUBM benchmark. Discussion of our findings and conclusions are given in Section 8.

2. LEHIGH UNIVERSITY BENCHMARK

LUBM [3] is a well-known and widely used synthetic RDF benchmark. It models a university scenario with universities, departments, research groups, professors and students. Further, LUBM provides an ontology (effectively a schema) for the data; it describes for example that *Full Professor*s are *Professor*s or the transitive relation between the universities' suborganizations. The benchmark data is produced by a data generator and can be of arbitrary size.

In our experiments we use scale factors 50 and 8000, i.e. the datasets contain 50 and 8000 universities, respectively. LUBM-50 contains ca. 6.8 million unique triples; LUBM-8000 has ca. 1065 million triples.

LUBM provides 14 queries. In this paper we will focus on 9 of them: query 2, 4, 5, 6, 7, 8, 9, 12 and 14. Doing so, we omit queries that are simple and/or have a sub-milliseconds runtime in

*supported by the LDBC EU FP7 project

GRADES'14, June 22 - 27 2014, Snowbird, UT, USA 2014
ACM 978-1-4503-2782-4/14/04 ...$15.00.
http://dx.doi.org/10.1145/2621934.2621944.

our benchmarks as such queries do not allow to distinguish between the systems under test.

Besides SPARQL pattern matching, the LUBM queries also test the reasoning capabilities of RDF systems. However, these capabilities are orthogonal to the goals of this paper. We will, therefore, simulate reasoning by both (i) re-writing the queries to take into account the rules from the ontology, and (ii) saturating the datasets with the inferred data (parts of the graph that are implied by the ontology but not generated by the data generator). We report both cases separately in our evaluation section.

In the first case, queries that ask for entities of a general type (e.g., all `Professors`) are expanded to explicitly look for all specific subclasses of that type (e.g., `AssociateProfessor` and `FullProfessor`). This hierarchy of types is suggested by the ontology, but is missing in the data.

In the second case, the dataset itself is enriched with all the facts that follow from the ontology rules. We will, for instance, add the general type `Professor` to every node of type `AssociateProfessor`, so that these nodes will be returned by the query that searches for all professors. The inferred versions our datasets contain 8.3 million and ca. 1.3 billion triples, respectively.

Although the generated dataset in LUBM is very regular and does not have any particular RDF's distinguishing properties, we explain in the next section how its diverse set of queries test the graph pattern matching capabilities of the system and, thus, make LUBM a good candidate for benchmarking graph pattern matching for both relational, RDF and property graph stores.

3. GRAPH PATTERN MATCHING

In this section we consider different flavors of graph pattern matching and illustrate them with examples from the LUBM benchmark.

3.1 General Definition

The input of the subgraph pattern matching problem are two graphs:

1. a graph G with nodes V and edges E, where nodes and edges are labeled with strings. We denote the label of a node v and an edge e as $\mathsf{label}(v)$ and $\mathsf{label}(e)$, respectively.
2. a query pattern, which can be viewed as a graph $P = (V_p, E_p)$. Here, the nodes and edges describe conditions that a subgraph of G must satisfy in order to be a match. Frequently, the query pattern is a conjunction of smaller patterns that together impose requirements on nodes and their neighborhoods in the data graph.

Given a graph G, the pattern matching problem is to find all possible subgraphs of G that match a given pattern P. More precisely, the match is defined in terms of

1. structural match (or isomorphism) between P and a candidate solution, and
2. conditions on labels of certain nodes and edges in a candidate solution (e.g., equality of given node labels in P and a subgraph of G)

In the following we use the SPARQL Protocol and RDF Query Language to describe query patterns. We note that SPARQL is not the only way to express such queries; we merely use it because it is well-known and has a concise SQL-inspired syntax. In the scope of this paper it is sufficient to assume that a SPARQL query is a set of (s, p, o) condition triples where $s, o \in V_p$ and $p \in E_p$ for an edge between s and o. Two nodes and the edge between them match the triple when all of its conditions are fulfilled. Elements of the condition triple may be unbound variables that match any node or edge. Conceptually, a query is evaluated by first finding all matches for each pattern and then joining the respective matches on their common variables. In the following we illustrate important cases of pattern joins using queries from the LUBM benchmark.

3.2 Basic Patterns

Nodes and edges. The most basic graph pattern consists of a single triple pattern that satisfies given conditions. In this case no join is necessary. An example of this is LUBM's query 14 that requests all undergraduate students.

Neighborhoods and Stars. A second very important type of graph patterns are neighborhoods, i.e. matches on the nodes which are adjacent and edges that are incident to a given node. As an example, LUBM query 1 asks for all graduate students that attend a certain course. Here, all the nodes that are adjacent to the two nodes labeled `GraduateStudent` and `GraduateCourse0`, respectively, are answers to the query.

Neighborhood with multiple patterns[1] around a central node, e.g. as in LUBM query 4, are also referred to as star patterns.

Triangles. Triangle patterns look for three nodes adjacent to each other. An example is the following simplified version of LUBM query 2:

```
select ?x ?y ?z where {
   ?x undergraduateDegreeFrom ?y.
   ?x memberOf ?z.
   ?z subOrganizationOf ?y. }
```

Matching triangles is especially challenging from a query optimization point of view as potentially many intermediate results need to be processed. Worst case-optimal join algorithms exist that avoid such an explosion of intermediate results [7], however, to the best of our knowledge, none of the systems considered in this paper implement such an approach.

Fixed and variable length paths. A special case of graph patterns are paths of fixed or variable length. The former occur for example when the names of students who attend a given course are queried. Variable length paths on the other hand are commonly used to match hierarchical relationships. In LUBM query 12, for example, all direct and transitive sub-organizations of a university must be matched.

Depending on the internal graph representation of a particular system, answering a path query requires a series of joins (in triple stores or RDBMS) or some form of breadth-first expansion on the graph (in native graph stores).

3.3 Patterns in LUBM queries

All LUBM queries use basic patterns. In addition, most queries contain further patterns. Table 1 provides an overview of the used patterns and, thus, of the queries' characteristics.

[1]To distinguish star patterns from fixed length paths we only use this term if three or more patterns are specified around the central node.

	Q2	Q4	Q5	Q6	Q7	Q8	Q9	Q12	Q14
Basic pattern	x	x	x	x	x	x	x	x	x
Neighborhood	x	x	x		x	x	x	x	
Star	x	x				x	x		
Triangle	x						x	x	
Path						x		x	

Table 1: Query pattern types used in the LUBM queries

4. DATA MODELS

In this paper we concentrate on how different systems solve the graph pattern matching problem. Doing pattern matching in database systems requires us to *(i)* model the graph in the corresponding domain (which is trivial for graph stores, but requires additional effort for relational systems), and *(ii)* express the graph pattern matching queries in the query language or API of the system. Additionally, since the benchmark in use – LUBM – requires RDF reasoning capabilities that non-RDF systems do not have, we essentially perform backward and forward chaining as described in Section 2.

Here we briefly describe the data models used for graph data representation, and techniques for to express graph pattern matching queries in them.

4.1 Resource Description Format

RDF and the SPARQL query language are the standard data representation and querying format in the Semantic Web domain. It views the data as a collection of *(subject, predicate, object)* triples, each describing a statement. An *object* can be either a literal value, or a URI which refers to a *subject* in another triple. This way, the RDF representation naturally describes a graph where *subject* and *object* are nodes, and the *predicate* is the label of an edge between them. Note that this matches the graph model described in Section 3.

4.2 Property Graph

The property graph model goes beyond RDF and allows nodes and edges in the graph to have an arbitrary number of properties (key/value pairs), in addition to labels. We use a natural mapping between the RDF and PG models: whenever a triple (S, P, O) has a literal value in O, we model (P, O) as a property-value pair of the node S. Otherwise (in case O is a URI), the triple describes an edge between two nodes S and O with the label on the edges P. This way, not all the triples in the RDF graph become edges in the PG model, making it a (conceptually) less verbose model.

4.2.1 Cypher Query Language

At the moment the only declarative query language developed for the property graph model is Neo4j's Cypher. Like SPARQL, Cypher allows to specify a subgraph in the `MATCH` clause with constants and variables in place of nodes and edges to be matched. Additional constraints on properties of nodes and edges can be expressed in the `WHERE` clause which is similar to SPARQL's `FILTER` clause. Consider LUBM query 7 that asks for all the students that attends classes taught by a certain professor:

```
match
(x)-[:TakesCourse]->(y)<-[:TeacherOf]-(z)
where z.id='AssociateProfessor0'
return x,y
```

Here, the relationship `TakesCourse` between nodes `x` (students) and nodes `y` (courses) is expressed with the ASCII-art pattern `(x:Student)-[:TakesCourse]->(y:Course)`, which is further extended by matching all edges `TeacherOf` between courses `y` and professors `z`. Unlike SPARQL, relationships between nodes can have multiple properties themselves. Moreover, it is possible to query the paths between nodes.

4.2.2 Query by API

When the graph store does not support any declarative language, as it is for example the case with the Sparksee system, one has to rely on a series of API calls. Specifically, all the cases of graph pattern matching (matching neighborhoods, triangles, paths) boil down to multiple calls of API functions that return nodes/edges of a given label and immediate neighbors of a given node. The following code snippet illustrates a part of LUBM query 2 that looks for all departments of all graduate students:

```
Objects gradstudents = graph.select(
        type,  Condition.Equal,
        v.setString("GraduateStudent"));

Objects departments = graph.neighbors(
        gradstudents, memberOf,
        EdgesDirection.Outgoing);
```

The first API call returns all graduate students (all the nodes whose `type` attribute is set to `GraduateStudent`), the second function gets all the neighbors of the students from the first function.

4.3 Relational Model

Finally, graphs and pattern matching problems can be formulated in the relational domain as well. In order to do so, we assume that every node belongs to one particular type (e.g., `Student`) with a fixed set of properties that we statically determine. These node types we translate into relations. Further, we store the relationships between nodes in mapping tables. For instance, the `TakesCourse` relationship stores connections between `Students` and `Courses`. Once the schema for the entire dataset is defined, we create indexes, e.g, on the create ID and URI as well as the `Person` name, to speed up lookups and joins.

Note that this assumption about a fixed schema for nodes holds in the majority of practical use cases. Obviously, this is especially the case for the synthetic LUBM data. However, this is true even for real-world RDF datasets [6].

In our experience SQL queries for graph pattern matching are very verbose: indeed, the fact that relationships are stored as separate tables means that a single hop lookup in the graph conceptually translates into two joins. As an example for this issue consider LUBM query 3 which finds all publications of a given professor. The query can be mapped to SQL as follows:

```
select P.URI
from publication P, faculty A,
     publicationAuthor PA
where A.URI='AssistantProfessor0'
  and PA.Id1=P.Id
  and PA.Id2=A.Id;
```

On the other hand, using SQL and the relational model for graph pattern matching allows us to leverage decades of development in transactional processing, query optimization and system tuning. Relational database management systems (RDBMS) can become a particularly attractive option for "hybrid" datasets, where only part of the data is a graph, while some information comes in tables.

5. SYSTEMS

After we have surveyed the ways to conceptually model graph pattern matching problems in different areas, in this section we describe the systems we use for our empirical evaluation.

5.1 RDF Databases

5.1.1 Virtuoso

We use open-source versions of Virtuoso 6 and 7 as representatives of RDF quadstores. Virtuoso is a de-facto relational store that models RDF graph as a single table and translates SPARQL queries into SQL. Virtuoso 6 is a row store that keeps two full B+-tree indexes on quadruples (PSOG and POGS). In addition, three partial indexes SP, OP, GS are stored. Virtuoso 7 is a column store that takes advantage of compression scheme and vectored execution suitable for relational column stores.

5.1.2 TripleRush

TripleRush is a research RDF database that is based on the Signal/Collect framework [9]. It represents the triple data as partially-evaluated read-optimized patterns, matches a given query's graph pattern against those in parallel and then builds the results from the matched parts. This can be compared to join indices. We include TripleRush in our evaluation because it represents a novel approach towards SPARQL query processing that offers competitive query evaluation performance.

5.2 Relational Databases: Virtuoso

Since Virtuoso is a relational store as well, it is a viable solution for our relational mapping of LUBM. Note that our mapping is hand-tuned. Thus, it greatly differs from the schema that Virtuoso natively uses for RDF data.

In order to simplify matching the variable length paths we rewrote the SQL queries using domain knowledge such that matching `suborganizationOf` becomes a two-way join. This is possible since the hierarchy of that edge is only up to two-hops high. In general, one would need to rely on recursive SQL features to avoid making such assumptions about the data.

5.3 Graph Databases

Most native graph databases implement the PG model directly or indirectly. In this paper we talk about two systems that do so.

5.3.1 Neo4j

Neo4j is an open-source native graph database. that offers functionality similar to traditional RDBMSs such as full transactional support, a declarative query language (Cypher), availability and scalability through a distributed version. The major benefit of Neo4j is its intuitive way of modeling and querying graph-shaped data. Internally, it stores edges as double linked lists. Properties are stored separately, referencing the nodes with corresponding properties.

We access the graph pattern matching functionality of Neo4j by means of its query language Cypher. We access the query endpoint via a Java API that comes with the Neo4j distribution.

5.3.2 Sparksee

Sparksee is a proprietary native graph database. Internally it is a disk-based system that relies on B+-trees and compressed bitmap indexes to store nodes and edges with their properties. Sparksee provides access to data via custom API functions (we use the Java version of the API). The API contains a set of primitive operations on nodes and edges like adding/deleting nodes or extracting neighborhoods. In addition, the system provides native implementation of core graph algorithms such as connected component detection, shortest paths as well as different traversals. The reported use-cases for Sparksee include various types of graph analysis such as cluster and outlier detection.

	50	50 Inf	8000	8000 Inf
Neo4j	0:01:38	0:01:46	3:32:48	4:18:57
Sparksee	0:02:00	0:02:03	26:25:11	27:06:41
TripleRush	0:03:39	0:03:40	-	-
Virtuoso 6	0:05:56	0:07:31	timeout	timeout
Virtuoso 7	0:00:36	0:01:09	1:56:42	2:00:20
Virtuoso 7 Rel	0:00:32	-	1:50:07	-

Table 2: LUBM dataset loading times, in hours

6. EVALUATION

This section describes the setup and results of our experiments. We will distinguish between the two variants of LUBM benchmark: one will use *inferred* dataset, where all the facts implied by the ontology are added; another will operate on the original dataset but *re-write* the queries to account for ontology rules.

6.1 Experimental Setup

We ran our experiments on a server with two quad-core Intel Xeon X5570 CPUs @ 2.93 GHz and 64 GBs of main memory. The used operating system was Ubuntu Linux 14.04 with kernel 3.13.0-24. In our benchmarks we used the following versions of the database systems: Virtuoso 6.1.8, Virtuoso 7.1.0, Neo4j 2.0.1, Sparksee 5.0.0 and TripleRush received on March 26, 2014. All our test drivers were implemented in Java 6 and executed using the Oracle runtime version 1.7.0_25.

We configured all databases to use the entire system memory. Furthermore, Neo4J's interface allowed us to split the memory across caches (for nodes, edges, property, string stores).

Note that Sparksee, Neo4j and TripleRush were run in the same process as the test driver, whereas Virtuoso was run as a separate process. For the latter the communication between the database and the test driver was performed via JDBC.

6.2 Loading

Since the LUBM data generator outputs RDF XML files, loading the benchmark data into Virtuoso (both version 6 and 7) is straightforward. In order to convert the generated RDF file into the PG model, we sort and group RDF triples on subject. Then, the subject becomes a node, its literals create the properties of the node, while URI objects turn into nodes connected to the subject via an edge marked with the predicate. The resulting graph is serialized in the CSV format. It is then loaded into Neo4J using the built-in bulk loader; the Sparksee database is populated using the API for creating individual nodes and edges. In both databases we created all the relevant indexes on node IDs and node types. For the relational model, we create a CSV file for every relation, and then load them using Virtuoso's bulk loader. In both cases, the time to generate CSV files is not included in loading time.

Loading times are given in Table 1. Note that TripleRush runs out of memory for LUBM-8000. Virtuoso 6 was not able to load the large dataset within 30 hours.

6.3 Results

In this section we report the runtime results for the queries across all systems for both variants. Each query is run 10 times in warm cache, we report the average time. Due to a lack of space we do not report the cold cache runtimes.

6.3.1 Re-written queries

The runtime of all the queries is given in Figure 1 and Figure 2 for LUBM-50 and LUBM-8000, respectively. The runtimes of 10^6,

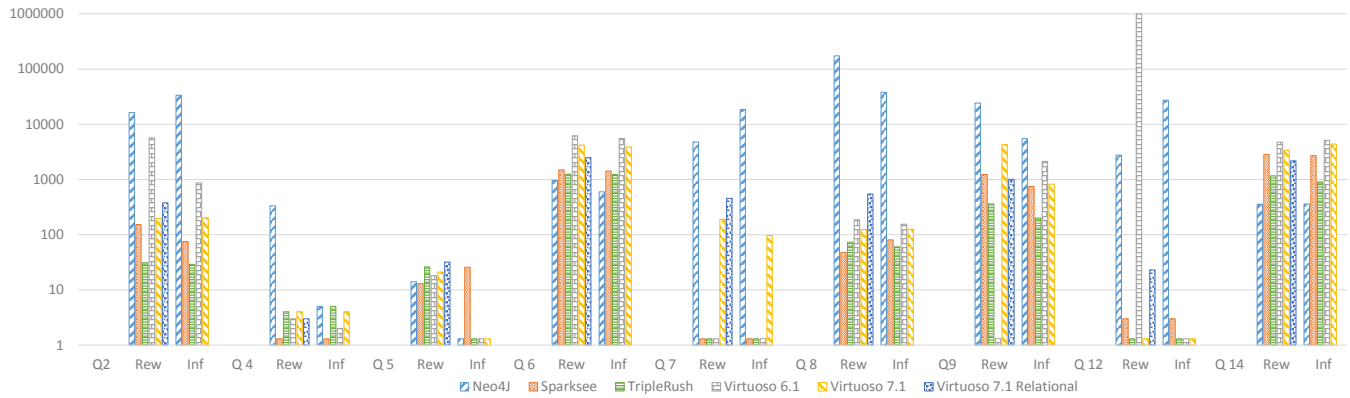

Figure 1: Benchmark results for LUBM 50, query runtime in ms

i.e. the bar reaches the top of the graph, indicate that the query timed out (took more than 10 minutes). For almost all the queries, the fastest system is Virtuoso. In Query 6 ("find all students"), however, Neo4J is faster than Virtuoso: we believe this is caused by the fact that Neo4J runs in the same process as test driver, so it avoids communication costs. Sparksee is fast for all the queries on the LUBM-50 dataset where the starting point of pattern matching is given and the amount of intermediate results is small. However, for LUBM-8000 dataset it is significantly slower than Virtuoso. Moreover, for queries like LUBM 6 and 9 it times out on the large dataset, since these queries require non-trivial query optimization effort that should avoid materialization of intermediate results (essentially those queries match triangles). We observe that the performance of Neo4j is even worse: on the larger dataset it times out for the majority of queries on LUBM-8000.

The row store Virtuoso is faster that column store only for Query 14 that returns all undergraduate students, since it essentially a look up of all rows with the same subject.

TripleRush seems like an competitive option for parallelized pattern matching in small-to-medium datasets. However, even for LUBM-50 it consumes more than 14 GB of RAM, which seems like a prohibitive cost for larger datasets: we were not able to load the LUBM-8000 dataset on our test machine.

Virtuoso Relational is faster than competitors for queries that extract multiple attributes of nodes or unselective queries. The performance deteriorates as the query pattern becomes more complex.

In additional measurements we showed that the re-written queries on Virtuoso are faster than the Virtuoso's built-in inferencing. For a lack of space we omit these numbers in our figures.

6.3.2 *Inferenced dataset*

Figure 1 and Figure 2 also provide the runtimes for queries over the inferenced datasets (LUBM-50 and 8000, respectively). Since all the facts implied by the ontology are in the database, the queries become simpler (see Appendix for full text of queries). This influences the results of all systems (except Neo4j), they become slightly faster across all the queries. As in previous case, Neo4j times out for majority of the queries.

6.4 Analysis

6.4.1 *Declarative vs Imperative Query Languages*

Our systems under test use either a declarative language (Cypher, SPARQL, SQL) or API (Sparksee). For the former, the system's optimizer has to figure out the optimal execution plan, while for the later the execution plan is the responsibility of the developer. The latter approach has several disadvantages. First, in Sparksee the intermediate results of API calls are immediately materialized and returned to the application, hurting the overall performance. Second, the application that uses this approach fixes the order of the operations to be performed, thus, providing an execution plan of the query. This way the application developer has to perform the work of a database query optimizer. Additionally, since query parameters may greatly affect the query plan, one would need to provide several implementations of the same query to account for changes in the appropriate execution strategy as it may change depending on the cardinalities of the input data or query parameters. Finally, this approach is the most labor-intensive for the application developer: in our experience, a single line of Cypher translates to a hundred lines of Java code using the Sparksee API.

The results show that Sparksee, provided with the optimal plan, achieves performance comparable with Virtuoso on the LUBM-50 dataset. However, for complex queries, such as Queries 9 and 12, it is penalized for materializing intermediate results. We also note that it does not scale well: for LUBM-8000 it performs much worse than Virtuoso across most of the queries.

6.4.2 *Query Optimization Issues*

The declarative query language allows a system to employ several optimization techniques, as discussed previously. However, this requires having a sophisticated query optimizer; otherwise, the system will be highly sensitive to how the query is formulated. We observe, for example, that in Query 10 (and similar) changing the order of predicates in WHERE clause of Cypher query influences the query runtime up to 1.6 times. This indicates that the current Neo4J's query engine does not perform a cost-based query optimization, and in fact executes the query as it is written.

6.4.3 *Matching Path Traversals and Triangles*

Although graph-specific operations are considered to be a selling point of native graph stores, in reality these stores provide suboptimal performance for pattern matching operations. We see that fixed length path traversals and triangle matching can be efficiently executed in a relational store. For example, in Query 8 and 12, Virtuoso relational outperformes Neo4j by several orders of magnitude. For the LUBM 8000 dataset, none of the path traversal queries finished within 10 minutes on Neo4j.

Same observation holds for triangle matching: use of a declarative query languange and the query optimizer of Virtuoso yield an unmatched performance for Queries 9 and 12. We note that, to the best of our knowledge, Virtuoso does not employ the specialized leapfrog-trie join for triangle matching, so its result could be significantly improved.

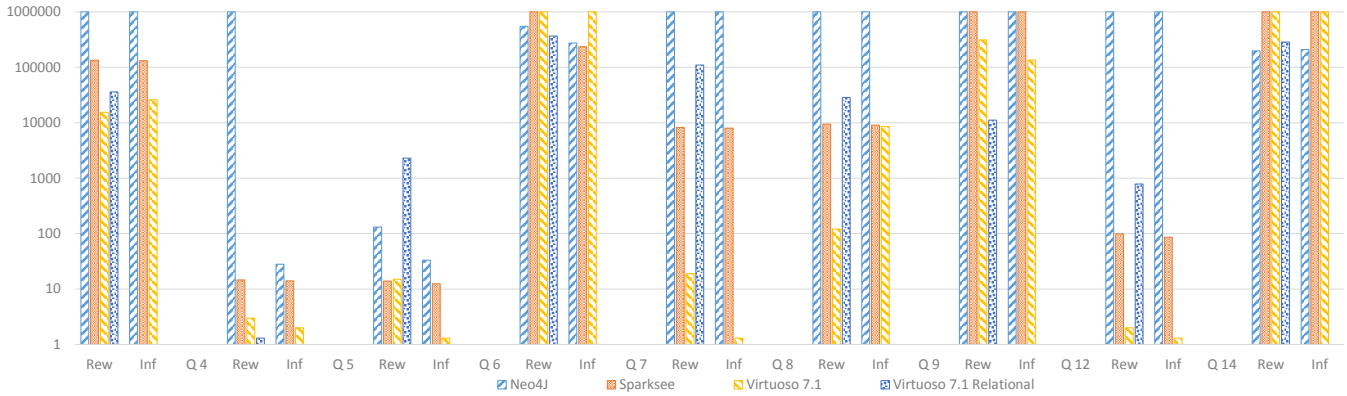

Figure 2: Benchmark results for LUBM 8000, query runtime in ms

6.4.4 Result Materialization

Some of the queries, e.g. query 6 and 14, yield a lot of results. We see that transferring these results to the application can significantly impact a query's runtime; for simple queries we observed runtime degradations of more than three orders of magnitude. This poses a challenge for all systems under test. For Sparksee, however, not only queries with many final results but also those with a lot of intermediate ones cause performance problems as all intermediate nodes along with their properties (e.g. names and email addresses) are transferred to the application.

7. RELATED WORK

Benchmarking graph and noSQL database systems has recently attracted a lot of attention in academia. Usually specific benchmark are limited to testing specific micro-operations (like getting neighborhoods of individual nodes or finding shortest paths between pairs of nodes) [2, 4], or only consider medium-sized graphs (up to few million edges) [1, 2, 5]. The LUBM benchmark, on the other hand, has been used as an instrument of comparison of RDF stores, although, as we have seen, it offers a convenient way of comparing graph pattern matching capabilities of very different systems.

This work can be thought of as a continuation of [10], where the shortest path algorithms were considered on a variety of systems, with an conclusion similar to ours.

8. DISCUSSION

As we have seen, in terms of performance, modern graph stores have little to offer for pattern matching-intensive applications: a significant amount of queries timed out for LUBM-8000 on both Sparksee and Neo4j. We note that in some cases Sparksee had an advantage of executing the optimal query plan, since we have written queries as Java programs that calls API functions of the system. While TripleRush seems like a competitive option, one has to keep in mind that it is still a research prototype; moreover, it does not scale for large graphs due to its prohibitively large memory usage. This disadvantage seems to be an inherent property of the approach itself, and not merely of the current implementation.

However, this comparison is intentionally one-sided (for the purpose of this paper). In reality, users opt for native graph databases for their intuitive modelling of graph-shaped datasets and "query by example" capabilities that come with it. Moreover, modelling graph-data as purely relational, while possible for synthetic data of LUBM, quickly becomes hard for real-world "messy and noisy" datasets, which frequently can be split into more regular and purely graph parts. Another downside of modelling graph data in relational model and then querying it with SQL is its inflexibility and error-proneness as a result of mismatch between the graph model and the query language.

It is our belief, backed by the performance comparisons of this paper, that hybrid systems that extend relational stores with graph-specific operations and graph query languages, will be the best fit for pattern-matching and general graph analysis problems. An example of such a bridge between relational and RDF world is given in [8].

References

[1] D. D. Abreu, A. Flores, G. Palma, V. Pestana, J. Piñero, J. Queipo, J. Sánchez, and M.-E. Vidal. Choosing between graph databases and rdf engines for consuming and mining linked data. In *COLD*, 2013.

[2] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. Benchmarking database systems for social network applications. In *GRADES*, page 15, 2013.

[3] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semant.*, 3(2-3):158–182, Oct. 2005.

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

[5] P. Macko, D. W. Margo, and M. I. Seltzer. Performance introspection of graph databases. In *SYSTOR*, page 18, 2013.

[6] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *ICDE*, pages 984–994, 2011.

[7] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[8] M.-D. Pham. Self-organizing structured rdf in monetdb. In *ICDE Workshops*, pages 310–313, 2013.

[9] P. Stutz, M. Verman, L. Fischer, and A. Bernstein. Triplerush: A fast and scalable triple store. In *SSWS@ISWC*, pages 50–65, 2013.

[10] A. Welc, R. Raman, Z. Wu, S. Hong, H. Chafi, and J. Banerjee. Graph analysis: do we have to reinvent the wheel? In *GRADES*, page 7, 2013.

APPENDIX

A. CYPHER QUERIES

2.

```
 match (x:GraduateStudent)-
[:MemberOf]-(z:Department),
(x:GraduateStudent)-
[:UndergraduateDegreeFrom]-(y:University)
<-[:SubOrganizationOf]-(z:Department)
return x, y, z
```

4.

```
match (x:Professor)-[:WorksFor]
 -(y:Department)
where y.id='University0'
return x.name, x.emailAddress,
  x.telephone
```

5.

```
match (x:Professor)-[:MemberOf]-
 (y:Department)
where y.id='University0'
return x
```

6.

```
match (x:Student) return x
```

7.

```
match (x:Student)-[:TakesCourse]
-(y:Course)<-[:TeacherOf]-(z)
where z.id='AssociateProfessor0'
return x,y
```

8.

```
match (x:Student)-[:MemberOf]
-(y:Department)-
[:SubOrganizationOf*1..2]-(z)
where z.id='University0'
return x, y, x.emailAddress
```

9.

```
match (x:Student)-[:Advisor]
-(y:Professor)-[:TeacherOf]-(z),
 (x)-[:TakesCourse]-(z:Course)
return x, y ,z
```

12.

```
match (x)-[:WorksFor]-(y:Department),
(y:Department)-[:SubOrganizationOf*1..2]-
(uni),(x)-[:HeadOf]-(z:Department)
where (x:FullProfessor) and
(uni.id='University0')
return x, y
```

13.

```
match (x)-[:DoctoralDegreeFrom|
MastersDegreeFrom|
UndergraduateDegreeFrom]-(y)
where y.id='University0'
return x
```

14.

```
match (x:UndergraduateStudent)
return x
```