On the Performance of Analytical and Pattern Matching Graph Queries in Neo4j and a Relational Database

Jürgen Hölsch   Tobias Schmidt   Michael Grossniklaus

Department of Computer and Information Science, University of Konstanz
P.O. Box 188, 78457 Konstanz, Germany

{juergen.hoelsch, tobias.2.schmidt, michael.grossniklaus}@uni-konstanz.de

ABSTRACT

Graph databases with a custom non-relational backend promote themselves to outperform relational databases in answering queries on large graphs. Recent empirical studies show that this claim is not always true. However, these studies focus only on pattern matching queries and neglect analytical queries used in practice such as shortest path, diameter, degree centrality or closeness centrality. In addition, there is no distinction between different types of pattern matching queries. In this paper, we introduce a set of analytical and pattern matching queries, and evaluate them in Neo4j and a market-leading commercial relational database system. We show that the relational database system outperforms Neo4j for our analytical queries and that Neo4j is faster for queries that do not filter on specific edge types.

1. INTRODUCTION

Application domains such as social media, biology, and transportation planning produce large amounts of graph data. Therefore, the management and processing of graph data is gaining importance. As a result, there are several graph databases such as Neo4j, DEX/Sparksee, and OrientDB. In general, graph databases can be categorized into systems using an existing relational database as a backend or systems implementing a custom non-relational backend, which are often called "native graph databases". Although a relational backend offers numerous advantages such as transactions, reliable storage and access control, native graph databases attract customers with the promise of outperforming traditional relational databases. Recent empirical studies [4,9], however, demonstrate that this promise is not always kept. Gubichev and Then [4] show that several queries of their benchmark timed out in Neo4j and DEX/Sparksee. While these studies are a good starting point for comparing graph databases with a relational and a custom non-relational backend, they only focus on pattern matching queries and neglect analytical queries such as shortest path and centrality measures.

In this paper, we bridge this gap by defining a set of analytical queries (Section 2). In addition, compared to related work, we introduce a more fine-grained categorization of pattern matching queries including (i) paths that filter on specifc node labels without restricting edge types, (ii) paths that filter on specific edge types without restricting node labels, (iii) paths that filter on node labels and edge types, and (iv) paths containing cycles. This categorization provides the basis to systematically breaking down the reasons why one system is outperforming another system. Since Neo4j is the most widely used native graph database, we compare the execution times of Cypher queries in Neo4j to the execution times of corresponding SQL queries in a market-leading relational database system (Section 3). We show that the relational database system outperforms Neo4j for our analytical queries. We argue that analytical queries, which access most or all nodes of the graph, benefit from the more advanced disk and buffer management of the relational database system and that there is room for improvement in this respect in Neo4j. In contrast, we demonstrate that Neo4j is more efficient for path queries that do not filter on specific edge types. Section 4 summarizes related work and Section 5 gives an outlook on future work.

2. QUERIES

We begin by introducing the set of queries that we use to compare Neo4j to a relational database system. We categorize our queries in analytical and pattern matching queries. Analytical queries process large parts of the graph at a time, whereas pattern matching queries access only small parts of the graph in most cases. In order to formally define our queries, we first have to introduce the property graph data model on which Neo4j and its query language Cypher is based [6].

Definition 1. Assume a domain $\mathcal{V}$ of nodes, a domain $\mathcal{E}$ of directed edges, and a domain $\mathcal{A}$ of attributes. Additionally, suppose a domain $\mathcal{D}_A$ of atomic attribute values, a domain $\mathcal{D}_L$ of node labels, and a domain $\mathcal{D}_T$ of edge types. A *property graph* is a finite structure $G = (V, E, A, \lambda, \alpha, \beta, \gamma)$, where

- $V \subseteq \mathcal{V}$ is a finite set of nodes,

- $E \subseteq \mathcal{E}$ is a finite set of edges,

- $A \subseteq \mathcal{A}$ is a finite set of attributes,

- $\lambda : E \to V \times V$ is a function assigning nodes to edges,

- $\alpha : (V \cup E) \times A \to \mathcal{D}_A$ is a partial function assigning values to attributes of nodes and edges,

- $\beta : V \to \mathcal{P}(\mathcal{D}_L)$ is a partial function assigning labels to nodes, and

- $\gamma : E \to \mathcal{D}_T$ is a partial function assigning a type to edges.

Besides the property graph data model, we also define the notion of a path in a property graph.

Definition 2. Suppose the nodes $v_1, v_n \in V$. A *path* from v_1 to v_n, denoted as $v_1 \to^* v_n$, is a sequence of nodes and edges $\langle v_1, e_1, v_2, \ldots, v_{n-1}, e_{n-1}, v_n \rangle$, where $\forall 1 \leq i \leq n-1 : \lambda(e_i) = (v_i, v_{i+1})$.

2.1 Analytical Queries

In this subsection, we formally define the set of analytical queries that have been proposed in the graph benchmark of Grossniklaus *et al.* [3]. Since the focus of this paper is to compare Neo4j with a relational database system, we only use queries that can be expressed in Cypher. Therefore, we cannot cover all analytical graph queries that are used in real world applications. For each query of this subsection, we also derive the corresponding Cypher and SQL statements that are based on the dataset of the LUBM [5] benchmark (*cf.* Section 3). The tables and views that are used in the SQL statements are described in the Appendix. Some of the following queries are limited in the number of result tuples because Neo4j is not able to execute these queries without restricting the result size.

2.1.1 QA_1: Node Degree

Query QA_1 returns for each node $v \in V$ its degree, which is the number of ingoing and outgoing edges and formally defined as follows:

$$\deg(v) := \big|\{e|e \in E \wedge \lambda(e) = (x,y) \wedge (v = x \vee v = y)\}\big|.$$

Cypher

```
MATCH (n)--(m)
RETURN n, COUNT(m)
```

SQL

```
SELECT n.id, COUNT(*) AS degree
FROM nodes n LEFT JOIN rel_all_bidirectional r ON n.id = r.sID
GROUP BY n.id
```

2.1.2 QA_2: Degree Centrality

Query QA_2 returns for each node $v \in V$ its degree centrality, which is the number of ingoing and outgoing edges normalized by the total number of nodes in the graph. Formally, the degree centrality of a node v is defined as follows:

$$C^D(v) := \frac{\deg(v)}{|V|}.$$

Cypher

```
MATCH (a)
WITH TOFLOAT(COUNT(*)) AS total
MATCH (n)-[e]-()
RETURN n, COUNT(e)/total AS centrality
```

SQL

```
SELECT n.id, CAST(COUNT(*) AS FLOAT)/
  (SELECT COUNT(*) FROM nodes) AS degree_centrality
FROM nodes n LEFT JOIN rel_all_bidirectional r ON n.id = r.sID
GROUP BY n.id
```

2.1.3 QA_3: Connectedness

Query QA_3 returns all pairs of nodes $u, v \in V$ which are connected by a path $u \to^* v$.

Cypher

```
MATCH (n), (m)
RETURN n, m, EXISTS((n)-[*]->(m)) AS is_connected
```

SQL

```
SELECT a.id1, a.id2,
  CASE WHEN tc.sID IS NULL THEN 0 ELSE 1 END AS connected
FROM (SELECT n.id AS id1, m.id AS id2 FROM nodes n
  CROSS JOIN nodes m) AS a
  LEFT JOIN transitive_closure tc ON a.id1 = tc.sID
  AND a.id2 = tc.dID
```

2.1.4 QA_4: Shortest Paths

Suppose $P(u, v)$ is a set containing all paths $u \to^* v$, where $u, v \in V$, and $\mathrm{length}(p)$ is a function returning the number of edges of a path p. A shortest path between two nodes u and v is a path p where $\forall p' \in P(u, v) : \mathrm{length}(p) \leq \mathrm{length}(p')$. The length of a shortest path between u and v is denoted as $\delta(u, v)$. Query QA_4 returns for all pairs of nodes $u, v \in V$ a shortest path between u and v.

Cypher

```
MATCH p = shortestPath((n)-[*]->(m))
RETURN n, m, p
```

SQL

```
SELECT *
FROM shortest_path
```

2.1.5 QA_5: Closeness Centrality

Query QA_5 returns for each node $v \in V$ its closeness centrality which is defined as follows:

$$C^C(v) := \frac{1}{\sum_{u \in V} \delta(v, u)}.$$

Cypher

```
MATCH p = shortestPath((n)-[*]-(m))
WHERE n <> m WITH n, p LIMIT 60000000
RETURN n, 1.0/SUM(LENGTH(p)) AS centrality
```

SQL

```
SELECT TOP 60000000 sID AS node,
  1.0 / SUM(length) AS centrality
FROM (SELECT sID, dID, MIN(length) AS length
  FROM shortest_connection GROUP BY sID, dID
  UNION ALL SELECT dID, sID, MIN(length) AS length
  FROM shortest_connection GROUP BY sID, dID) AS t
GROUP BY sID
```

2.1.6 QA_6: Diameter

Query QA_6 returns the diameter of the graph which is the length of the longest shortest path between any two nodes $u, v \in V$:

$$d := \max_{u,v \in V} \delta(u, v).$$

Cypher

```
MATCH p = shortestPath((n)-[*]-(m))
WITH p LIMIT 40000000
RETURN p, LENGTH(p)
ORDER BY LENGTH(p) DESC
LIMIT 1
```

SQL

```
SELECT MAX(length)
FROM (SELECT TOP 40000000 *
 FROM shortest_connection) AS t
```

2.1.7 QA_7: Grouping

Query QA_7 returns for each existing value x of an attribute $a \in A$ the number of nodes $v \in V$ where $\alpha(v, a) = x$:

$$\gamma(a) := \{(x, i) \mid v \in V \wedge a \in A \wedge x = \alpha(v, a)$$
$$\wedge\ i = |\{u \mid u \in V \wedge \alpha(u, a) = x\}|\}.$$

Cypher

```
MATCH (a:Course)
RETURN a.name, COUNT(a)
```

SQL

```
SELECT name.value, COUNT(*) AS count
FROM node_label node LEFT JOIN name ON name.id = node.id
WHERE node.value = 'Course'
GROUP BY name.value
ORDER BY count DESC
```

2.2 Pattern Matching Queries

In the following, we define the types of pattern matching queries used in our evaluation. We categorize our queries in paths that filter on specific node labels without restricting edge types (QP_1), paths that filter on specific edge types without restricting node labels (QP_3), paths that filter on node labels and edge types (QP_4), and paths containing cycles (QP_2). This categorization provides a way to determine how efficient Neo4j and the relational database system can filter on specific node labels or edge types. For each pattern matching query type, we derive Cypher and SQL statements. However, in this subsection we only present the actual code of queries with patterns of length 1 (denoted as $QP_{x,1}$ in Section 3, where x is the query type). The queries with longer patterns used in the evaluation are found in the appendix.

2.2.1 QP_1: Paths filtering on node labels

The first type of pattern matching queries in our evaluation are paths filtering on specific node labels. This query type is illustrated below, where $l_1, \ldots, l_n \in \mathcal{D}_L$ are labels.
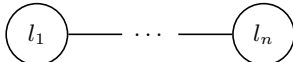
Note that we do not restrict the edge direction. Both outgoing and ingoing edges are matched by the pattern above.
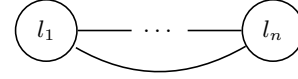
Cypher

```
MATCH (a:University)--(b:FullProfessor)
RETURN a, b
```

SQL

```
SELECT n1.id, n2.id
FROM rel_all_bidirectional r
  INNER JOIN node_label n1 ON n1.id = r.sID
  INNER JOIN node_label n2 ON n2.id = r.dID
WHERE n1.value = 'University' AND n2.value = 'FullProfessor'
```

2.2.2 QP_2: Cycles

Queries of type QP_2 extend queries of type QP_1 with cycles.
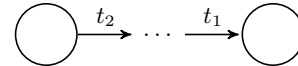

Cypher

```
MATCH (a:UndergraduateStudent)--(b:FullProfessor)
 --(c:Department)--(a)
RETURN a,b,c
```

SQL

```
SELECT n1.id, n2.id, n3.id
FROM rel_all_bidirectional r1
  INNER JOIN node_label n1 ON n1.id = r1.sID
  INNER JOIN node_label n2 ON n2.id = r1.dID
  INNER JOIN rel_all_bidirectional r2 ON r2.sID = n2.id
  INNER JOIN node_label n3 ON n3.id = r2.dID
  INNER JOIN rel_all_bidirectional r3 ON n3.id = r3.sID
WHERE n1.value = 'UndergraduateStudent'
  AND n2.value = 'FullProfessor'
  AND n3.value = 'Department'
  AND r3.dID = n1.id
```

2.2.3 QP_3: Paths filtering on edge types

Queries of type QP_3 return paths with specific edge types as it is shown in the figure below, where $t_1, \ldots, t_n \in \mathcal{D}_T$ are edge types.
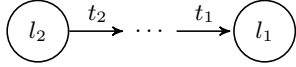

Cypher

```
MATCH (a)-[:teacherOf]->(b)<-[:takesCourse]-(c)
RETURN a, b, c
```

SQL

```
SELECT n1.id AS teacher, n2.id AS course, n3.id AS student
FROM nodes n1, rel_teacherOf rto, nodes n2,
 rel_takesCourse rtc, nodes n3
WHERE n1.id = rto.sID AND rto.dID = n2.id
  AND n2.id = rtc.dID AND n3.id = rtc.sID
```

2.2.4 QP_4: Paths filtering on node labels and edge types

Queries of type QP_4 are a combination of queries of type QP_1 and of type QP_3.

$$\left(l_2\right) \xrightarrow{t_2} \cdots \xrightarrow{t_1} \left(l_1\right)$$

Cypher

```
MATCH (a:FullProfessor)-[:teacherOf]->(b:Course)
  <-[:takesCourse]-(c:UndergraduateStudent)
RETURN a, b, c
```

SQL

```
SELECT a.id, b.id, c.id
FROM node_label a, rel_teacherOf r1, node_label b,
  rel_takesCourse r2, node_label c
WHERE a.value = 'FullProfessor' AND b.value = 'Course'
  AND c.value = 'UndergraduateStudent' AND a.id = r1.sID
  AND r1.dID = b.id AND b.id = r2.sID AND r2.dID = c.id
```

3. EVALUATION

Having presented the queries used in our performance evaluation, we now present the experimental setup and the obtained results.

3.1 Experimental Setup

All experiments presented in this paper were performed on a Mac Pro with a 3.5 GHz 6-Core Intel Xeon E5 processor with 64 GB main memory. In our evaluation, we compared Neo4j Version 3.0.3 with a market-leading commercial database system. Due to license terms, the commercial system is referred to as "RDBMS". Neo4j and RDBMS are installed on a 64-bit Windows 8.1 virtual machine with 32 GB of main memory. Both database systems run in their standard configuration. The data sets used in our evaluation were generated by the data generator of the LUBM benchmark [5]. We generated a data set with one university (LUBM-1) and a data set with two universities (LUBM-2). To store the graph in RDBMS, we used the decomposed storage model (DSM) described by Sakr *et al.* [10]. All runtime measurements were repeated five times in random order. The reported averages discard the smallest and the largest value.

Figure 1 shows the runtimes of our queries on the LUBM-1 data set, whereas Figure 2 shows the runtimes of our queries on the LUBM-2 data set. In our first experiments, we have noticed that the performance of specific query types in RDBMS heavily depends on how efficiently edges can be retrieved. Therefore, we created two different schemata in RDBMS that are both based on the decomposed storage model. Schema "A" has tables for each edge type. In addition, a view is computed containing edges of all types. In contrast, Schema "B" has a table containing edges of all types. In addition, there are views for each edge type. In the following, RDBMS with Schema A is denoted as "RDBMS A" and RDBMS with Schema B is denoted as "RDBMS B". We use the term "RDBMS" to refer to both RDBMS A and RDBMS B.
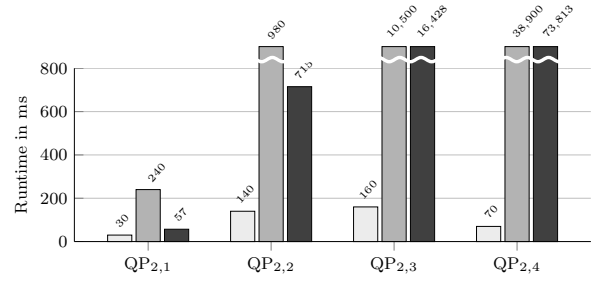
3.2 Results

We first present the results obtained on data set LUBM-1. Figure 1(a) shows that for most of the analytical queries
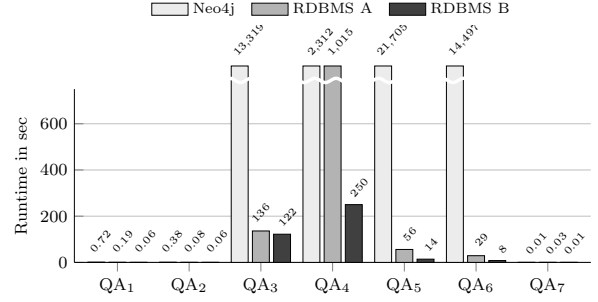
(a) Analytical queries
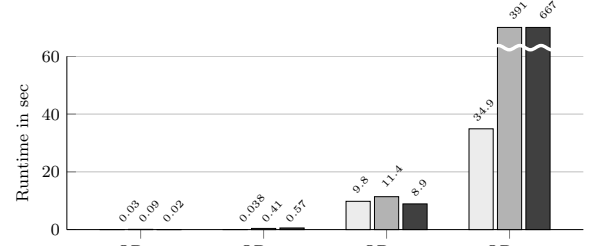
(b) Paths filtering on node labels

(c) Cycles

(d) Paths filtering on edge types

(e) Paths filtering node labels and edge types

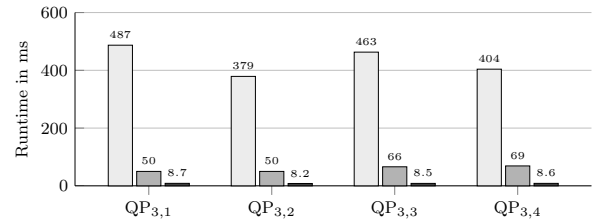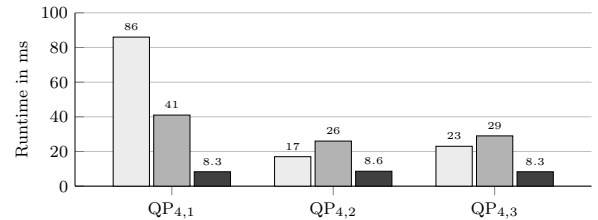**Figure 1: Runtimes of queries on LUBM-1.**

(a) Analytical queries

(b) Paths filtering on node labels

(c) Cycles

(d) Paths filtering on edge types

(e) Paths filtering node labels and edge types

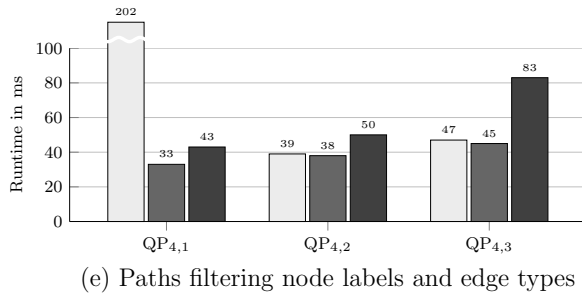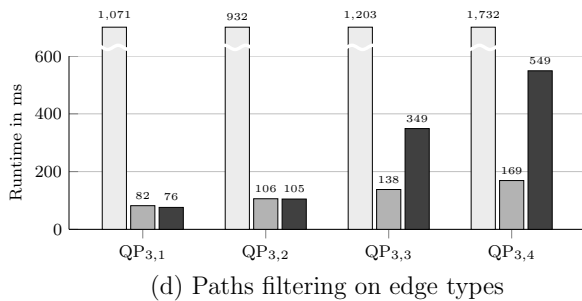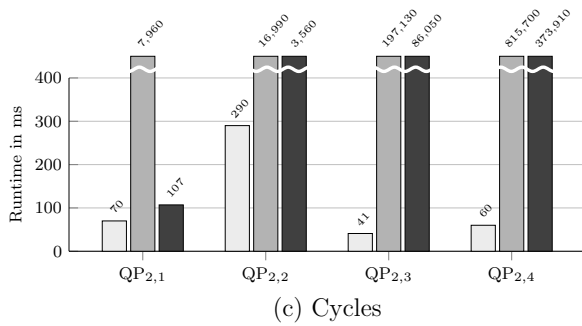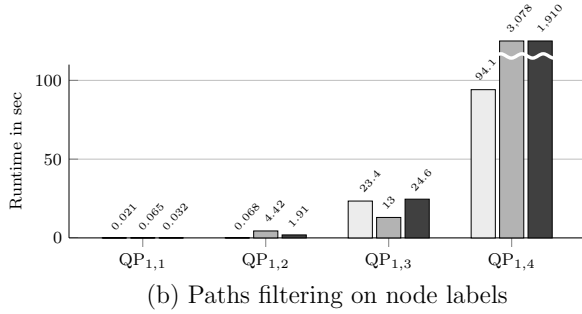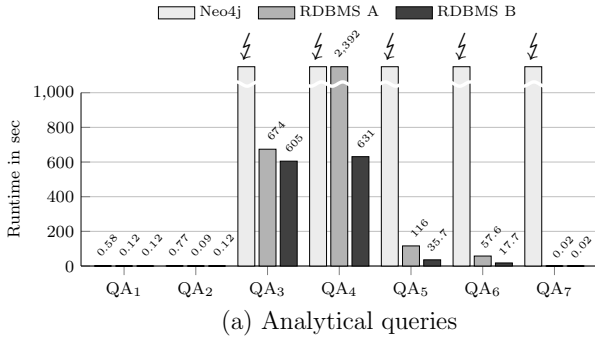**Figure 2: Runtimes of queries on LUBM-2.**

RDBMS is several orders of magnitude faster than Neo4j. The largest difference can be seen for query QA_5 which computes the closeness centrality where RDBMS B is over $1500\times$ faster than Neo4j. The computation of the shortest path is almost $10\times$ faster in RDBMS B than in Neo4j. At a first glance, this is surprising since computing the shortest path is a core functionality of Neo4j and is also provided as a construct in the Cypher query language. Figure 1(b) gives the runtimes of our pattern matching queries that filter specific node labels without restricting edge types. The largest difference in the runtime of Neo4j and RDBMS can be observed for query $QP_{1,4}$ which contains the largest number of edges of all queries in (b). We can also see that the queries $QP_{1,1}$ and $QP_{1,3}$ are faster in RDBMS B than in Neo4j. The runtimes of cyclic pattern matching queries that filter specific node labels without restricting edge types are shown in Figure 1(c). It can be seen that Neo4j outperforms RDBMS. For all queries of Figure 1(c), Neo4j is almost an order of magnitude faster than RDBMS. The runtimes of pattern matching queries on filtering specific edge types without restricting node labels are given in Figure 1(d). In contrast to the previous pattern matching query types, RDBMS answers queries of this type more efficiently than Neo4j. This is due to the fact that each relationship type has its own table (or view) in RDBMS. Finally, Figure 1(e) plots the runtimes of pattern matching queries that filter on node labels and edge types. As before, RDBMS B outperforms Neo4j and RDBMS A.

Moving on to data set LUBM-2, Figure 2(a) shows the runtime of our analytical queries. In Neo4j, only query QA_1 and QA_2 could be executed in reasonable time. The other queries were aborted after 24 hours or ran out of memory (denoted by $\mathnot{\xi}$ in Figure 2(a)). The runtimes of our pattern matching queries that filter specific node labels without restricting edge types are given in Figure 2(b). For query $QP_{1,3}$ RDBMS A is faster than Neo4j, otherwise Neo4j outperforms RDBMS. Figure 2(c) plots the runtimes of cyclic pattern matching queries that filter specific node labels without restricting edge types. As for the results obtained on the LUBM-1 data set (Figure 1(c)), Neo4j is consistently faster than RDBMS for this type of pattern matching query. The runtimes of pattern matching queries filtering specific edge types without restricting node labels that are plotted in Figure 2(d) show that RDBMS consistently outperforms Neo4j for this type of query. Finally, the runtimes of pattern matching queries that filter on node labels and edge types are shown in Figure 2(e). For these queries, RDBMS A consistently outperforms Neo4j.

3.3 Discussion and Limitations

To conclude this section, we summarize our results and discuss limitations of our evaluation. We have shown that RDBMS outperforms Neo4j for our analytical queries. For most of the analytical queries, RDBMS is several orders of magnitude faster than Neo4j. A possible reason for this result is the fact that the analytical queries have to access most or all nodes of the graph. Therefore, they profit from the advanced disk and memory management of relational database systems. We also showed that RDBMS A and RDBMS B both perform badly for queries that need to join the whole edge table (or view) multiple times for longer patterns. For queries with cycles, the performance is even worse. However, if we only query specific edge types, RDBMS outperforms Neo4j as can be seen in Figure 1(d) and Figure 2(d). Comparing

RDBMS A with RDBMS B on LUBM-1, we can observe that for queries that filter on specific edge types RDBMS B is more efficient than RDBMS A. However, for queries with longer patterns that do not filter on specific edge types RDBMS A outperforms RDBMS B. As for LUBM-1, there is again no clear winner on LUBM-2.

Since the work presented in this paper is only the first step towards a more extensive empirical study, it has some shortcomings and open issues. First, we only use the decomposed storage model [10] to map a property graph to relations. Since the choice of the graph-to-relation mapping strongly impacts the performance of queries, we need to evaluate alternative mappings such as the hybrid schema described by Sun *et al.* [11]. Second, the performance of a query also depends on the characteristics of the graph, *e.g.*, the number of distinct labels or average node degree. Hence, further data sets with different graph characteristics need to be evaluated. Finally, pattern matching queries that filter on specific attribute values should be included as our current queries filter on node labels and edge types only.

4. RELATED WORK

There are several empirical studies on the query performance of graph database systems. To the best of our knowledge, none of these studies cover analytical queries formulated in a declarative graph query language. The work of Vicknair *et al.* [12] benchmarks Neo4j and MySQL. The authors define three simple traversal queries and queries counting the number of nodes with a specific value. Ciglan *et al.* [2] implement graph traversal operations in Neo4j, DEX/Sparksee, OrientDB, NativeSail, and SGDB, and compare their performance. Grossniklaus *et al.* [3] benchmark Neo4j and a relational database system by implementing a set of analytical queries using the Neo4j API and SQL. Welc *et al.* [13] evaluate the performance of computing the shortest path in Neo4j, Oracle and the Green-Marl language infrastructure. Angles *et al.* [1] benchmark graph databases in the context of social networks. They define a set of pattern matching and reachability queries that are often used to analyse social networks. Jouili and Vansteenberghe [7] use traversal and shortest path queries to benchmark Neo4j, DEX/Sparksee, OrientDB and Titan. McColl *et al.* [8] evaluate open-source graph databases using implementations of the graph algorithms shortest path, connected components and PageRank. Pobiedina *et al.* [9] evaluate pattern matching queries in Neo4j (Cypher), PostgreSQL (SQL), Jena TDB (SPARQL) and Clingo (ASP).

5. CONCLUSION AND FUTURE WORK

In this paper, we defined a set of analytical and pattern matching queries in Cypher and SQL, and evaluated them in Neo4j and a market-leading commercial relational database system. We showed that the relational database system outperforms Neo4j for our analytical queries and that Neo4j is faster for queries that do not filter on specific edge types.

As future work we plan to build on this initial study in order to better understand the implications of different backends on graph query performance. First, we will study the influence of different graph-to-relation mappings on the performance of relational backends. Additionally, we will include further data sets with different graph characteristics and pattern matching queries that filter on attribute values.

6. REFERENCES

[1] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. Benchmarking Database Systems for Social Network Applications. In *Proc. Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 1–7, 2013.

[2] M. Ciglan, A. Averbuch, and L. Hluchy. Benchmarking Traversal Operations over Graph Databases. In *Proc. ICDE Workshops (ICDEW)*, pages 186–189, 2012.

[3] M. Grossniklaus, S. Leone, and T. Zäschke. Towards a Benchmark for Graph Data Management and Processing. Technical report, University of Konstanz, 2013. http://nbn-resolving.de/urn:nbn:de:bsz:352-242536.

[4] A. Gubichev and M. Then. Graph Pattern Matching: Do We Have to Reinvent the Wheel? In *Proc. Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 1–7, 2014.

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

[6] J. Hölsch and M. Grossniklaus. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In *Proc. Intl. Workshop on Querying Graph Structured Data (GraphQ)*, 2016.

[7] S. Jouili and V. Vansteenberghe. An Empirical Comparison of Graph Databases. In *Proc. Intl. Conf. on Social Computing (SOCIALCOM)*, pages 708–715, 2013.

[8] R. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A Performance Evaluation of Open Source Graph Databases. In *Proc. Workshop on Parallel Programming for Analytics Applications (PPAA)*, pages 11–18, 2014.

[9] N. Pobiedina, S. Rümmele, S. Skritek, and H. Werthner. Benchmarking Database Systems for Graph Pattern Matching. In *Proc. Intl. Conf. on Database and Expert Systems Applications (DEXA)*, pages 226–241, 2014.

[10] S. Sakr, S. Elnikety, and Y. He. G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 335–344, 2012.

[11] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 1887–1901, 2015.

[12] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proc. ACM Southeast Regional Conference*, page 42, 2010.

[13] A. Welc, R. Raman, Z. Wu, S. Hong, H. Chafi, and J. Banerjee. Graph Analysis: Do We Have to Reinvent the Wheel? In *Proc. Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 1–6, 2013.

APPENDIX

A. TABLES AND VIEWS

Below, we introduce the tables and views that were created for Schema A and Schema B in order to store a property graph in a relational database system. A node has a unique id and is stored in the `node_label` relation with its corresponding labels. Therefore, for each label of a node, there is an entry in the `node_label` table. For each attribute, there is a relation containing entries of the node id and the attribute value. The `node_label` table and each attribute table have a clustered index on the node id. In Schema A, there is a table for each edge type containing the edge id and the ids of the source and target nodes. The edge tables have a clustered index on the id of the source and target node. In Schema B, there is a table containing edges of all types. This table has a clustered index on the id of the source and target node as well as an unclustered index on the column indicating the edge type. In addition, we create the following views, where the prefix "rel_" denotes an edge table (or view). However, in Schema B we have a table `rel_all` instead of a view.

```
CREATE VIEW nodes (id) AS
SELECT DISTINCT id FROM node_label

CREATE VIEW rel_all (sID, dID) AS
SELECT sID, dID FROM rel_advisor
  UNION ALL
SELECT sID, dID FROM rel_doctoralDegreeFrom
  UNION AL
SELECT sID, dID FROM rel_headOf
  UNION ALL
SELECT sID, dID FROM rel_mastersDegreeFrom
  UNION ALL
SELECT sID, dID FROM rel_memberOf
  UNION ALL
SELECT sID, dID FROM rel_publicationAuthor
  UNION ALL
SELECT sID, dID FROM rel_subOrganizationOf
  UNION ALL
SELECT sID, dID FROM rel_takesCourse
  UNION ALL
SELECT sID, dID FROM rel_teacherOf
  UNION ALL
SELECT sID, dID FROM rel_teachingAssistantOf
  UNION ALL
SELECT sID, dID FROM rel_undergraduateDegreeFrom
  UNION ALL
SELECT sID, dID FROM rel_worksFor

CREATE VIEW rel_all_bidirectional (sID, dID) AS
SELECT sID, dID FROM rel_all
  UNION ALL
SELECT dID AS sID, sID AS dID FROM rel_all

CREATE VIEW transitive_closure AS
WITH BacktraceCTE(sID, parent, dID, length) AS
  (SELECT sID, NULL, dID, 1
   FROM dbo.rel_all
     UNION ALL
   SELECT b.sID, b.dID, r.dID, b.length + 1
   FROM BacktraceCTE AS b
   INNER JOIN dbo.rel_all AS r ON b.dID = r.sID)
SELECT sID, parent, dID, length FROM BacktraceCTE

CREATE VIEW shortest_connection AS
SELECT DISTINCT t.* FROM dbo.transitive_closure t
INNER JOIN (SELECT sID, dID, MIN(length) as minLength
FROM dbo.transitive_closure GROUP BY sID, dID) m
ON m.sID = t.sID AND m.dID = t.dID AND m.minLength = t.length

CREATE VIEW shortest_path AS WITH
  ShortestPathCTE (sID, dID, step_from, step_to) AS (
   SELECT sID, dID, parent, dID FROM dbo.shortest_connection
     UNION ALL
   SELECT t.sID, t.dID, coalesce(s.parent, t.sID) AS step_from,
```

```
   t.step_from AS step_to
  FROM ShortestPathCTE t
  INNER JOIN shortest_connection s
   ON s.dID = t.step_from AND s.sID = t.sID
) SELECT sID, dID,
   coalesce(step_from, sID) AS step_from, step_to
 FROM ShortestPathCTE
```

B. PATTERN MATCHING QUERIES

B.1 Paths restricted on node labels

$QP_{1,2}$ Cypher

```
MATCH (a:University)--(b:FullProfessor)--(c:Department)
RETURN a, b, c
```

$QP_{1,2}$ SQL

```
SELECT n1.id, n2.id, n3.id FROM rel_all_bidirectional r1
  INNER JOIN node_label n1 ON n1.id = r1.sID
  INNER JOIN node_label n2 ON n2.id = r1.dID
  INNER JOIN rel_all_bidirectional r2 ON r2.sID = n2.id
  INNER JOIN node_label n3 ON n3.id = r2.dID
WHERE n1.value = 'University' AND n2.value = 'FullProfessor'
  AND n3.value = 'Department'
```

$QP_{1,3}$ Cypher

```
MATCH (a:University)--(b:FullProfessor)--(c:Department)
 --(d:UndergraduateStudent)
RETURN a, b, c, d
```

$QP_{1,3}$ SQL

```
SELECT n1.id, n2.id, n3.id, n4.id FROM rel_all_bidirectional r1
  INNER JOIN node_label n1 ON n1.id = r1.sID
  INNER JOIN node_label n2 ON n2.id = r1.dID
  INNER JOIN rel_all_bidirectional r2 ON r2.sID = n2.id
  INNER JOIN node_label n3 ON n3.id = r2.dID
  INNER JOIN rel_all_bidirectional r3 ON n3.id = r3.sID
  INNER JOIN node_label n4 ON n4.id = r3.dID
WHERE n1.value = 'University' AND n2.value = 'FullProfessor'
  AND n3.value = 'Department'
  AND n4.value = 'UndergraduateStudent'
```

$QP_{1,4}$ Cypher

```
MATCH (a:University)--(b:FullProfessor)--(c:Department)
 --(d:UndergraduateStudent)--(e:Course)
RETURN a, b, c, d, e
```

$QP_{1,4}$ SQL

```
SELECT n1.id, n2.id, n3.id, n4.id, n5.id
FROM rel_all_bidirectional r1
  INNER JOIN node_label n1 ON n1.id = r1.sID
  INNER JOIN node_label n2 ON n2.id = r1.dID
  INNER JOIN rel_all_bidirectional r2 ON r2.sID = n2.id
  INNER JOIN node_label n3 ON n3.id = r2.dID
  INNER JOIN rel_all_bidirectional r3 ON n3.id = r3.sID
  INNER JOIN node_label n4 ON n4.id = r3.dID
  INNER JOIN rel_all_bidirectional r4 ON n4.id = r4.sID
  INNER JOIN node_label n5 ON n5.id = r4.dID
WHERE n1.value = 'University' AND n2.value = 'FullProfessor'
  AND n3.value = 'Department'
  AND n4.value = 'UndergraduateStudent' AND n5.value = 'Course'
```

B.2 Cycles

$QP_{2,2}$ Cypher

```
MATCH (a:UndergraduateStudent)--(b:AssociateProfessor)
 --(c:GraduateStudent)--(d:Department)--(a)
RETURN a,b,c,d
```

$QP_{2,2}$ SQL

```
SELECT n1.id, n2.id, n3.id, n4.id FROM rel_all_bidirectional r1
  INNER JOIN node_label n1 ON n1.id = r1.sID
  INNER JOIN node_label n2 ON n2.id = r1.dID
  INNER JOIN rel_all_bidirectional r2 ON r2.sID = n2.id
```

```
  INNER JOIN node_label n3 ON n3.id = r2.dID
  INNER JOIN rel_all_bidirectional r3 ON n3.id = r3.sID
  INNER JOIN node_label n4 ON n4.id = r3.sID
  INNER JOIN rel_all_bidirectional r4 ON n4.id = r4.sID
WHERE n1.value = 'UndergraduateStudent'
  AND n2.value = 'AssociateProfessor'
  AND n3.value = 'GraduateStudent'
  AND n4.value = 'Department' AND r4.dID = n1.id
```

$QP_{2,3}$ Cypher

```
MATCH (a:UndergraduateStudent)--(b:Course)
  --(c:AssociateProfessor)--(d:UndergraduateStudent)
  --(e:Course)--(a) WHERE b <> e AND a<>d
RETURN a,b,c,d,e
```

$QP_{2,3}$ SQL

```
SELECT n1.id, n2.id, n3.id, n4.id, n5.id
FROM rel_all_bidirectional r1
  INNER JOIN node_label n1 ON n1.id = r1.sID
  INNER JOIN node_label n2 ON n2.id = r1.dID
  INNER JOIN rel_all_bidirectional r2 ON r2.sID = n2.id
  INNER JOIN node_label n3 ON n3.id = r2.dID
  INNER JOIN rel_all_bidirectional r3 ON n3.id = r3.sID
  INNER JOIN node_label n4 ON n4.id = r3.dID
  INNER JOIN rel_all_bidirectional r4 ON n4.id = r4.sID
  INNER JOIN node_label n5 ON n5.id = r4.dID
  INNER JOIN rel_all_bidirectional r5 ON n5.id = r5.sID
WHERE n1.value = 'UndergraduateStudent' AND n2.value = 'Course'
  AND n3.value = 'AssociateProfessor'
  AND n4.value = 'UndergraduateStudent' AND n5.value = 'Course'
  AND r5.dID = n1.id AND n1.id != n4.id AND n2.id != n5.id
```

$QP_{2,4}$ Cypher

```
MATCH (a:UndergraduateStudent)--(b:Course)
  --(c:AssociateProfessor)--(d:UndergraduateStudent)
  --(e:Course) --(f:AssistantProfessor)--(a)
WHERE b <> e AND a<>d
RETURN a,b,c,d,e,f
```

$QP_{2,4}$ SQL

```
SELECT n1.id, n2.id, n3.id, n4.id, n5.id, n6.id
FROM rel_all_bidirectional r1
  INNER JOIN node_label n1 ON n1.id = r1.sID
  INNER JOIN node_label n2 ON n2.id = r1.dID
  INNER JOIN rel_all_bidirectional r2 ON r2.sID = n2.id
  INNER JOIN node_label n3 ON n3.id = r2.dID
  INNER JOIN rel_all_bidirectional r3 ON n3.id = r3.sID
  INNER JOIN node_label n4 ON n4.id = r3.dID
  INNER JOIN rel_all_bidirectional r4 ON n4.id = r4.sID
  INNER JOIN node_label n5 ON n5.id = r4.dID
  INNER JOIN rel_all_bidirectional r5 ON n5.id = r5.sID
  INNER JOIN node_label n6 ON n6.id = r5.dID
  INNER JOIN rel_all_bidirectional r6 ON n6.id = r6.sID
WHERE n1.value = 'UndergraduateStudent'
  AND n2.value = 'Course' AND n3.value = 'AssociateProfessor'
  AND n4.value = 'UndergraduateStudent' AND n5.value = 'Course'
  AND n6.value = 'AssistantProfessor' AND r6.dID = n1.id
  AND n1.id != n4.id AND n2.id != n5.id
```

B.3 Paths restricted on edge types

$QP_{3,2}$ Cypher

```
MATCH (a)-[:teacherOf]->(b)<-[:takesCourse]-(c)
  -[:advisor]->(d)
RETURN a, b, c, d
```

$QP_{3,2}$ SQL

```
SELECT n1.id AS teacher, n2.id AS course, n3.id AS student,
  n4.id AS advisor
FROM nodes n1, rel_teacherOf rto, nodes n2,
  rel_takesCourse rtc, nodes n3, rel_advisor ra, nodes n4
WHERE n1.id = rto.sID AND rto.dID = n2.id
  AND n2.id = rtc.dID AND n3.id = rtc.sID
  AND n3.id = ra.sID AND n4.id = ra.dID
```

$QP_{3,3}$ Cypher

```
MATCH (a)-[:teacherOf]->(b)<-[:takesCourse]-(c)
  -[:advisor]->(d)-[:mastersDegreeFrom]->(e)
RETURN a, b, c, d, e
```

$QP_{3,3}$ SQL

```
SELECT n1.id AS teacher, n2.id AS course, n3.id AS student,
  n4.id AS advisor
FROM nodes n1, rel_teacherOf rto, nodes n2, rel_takesCourse rtc,
  nodes n3, rel_advisor ra, nodes n4,
  rel_mastersDegreeFrom rmdf, nodes n5
WHERE n1.id = rto.sID AND rto.dID = n2.id
  AND n2.id = rtc.dID AND n3.id = rtc.sID
  AND n3.id = ra.sID AND n4.id = ra.dID
  AND n4.id = rmdf.sID AND n5.id = rmdf.dID
```

$QP_{3,4}$ Cypher

```
MATCH (a)-[:teacherOf]->(b)<-[:takesCourse]-(c)
-[:advisor]->(d)-[:mastersDegreeFrom]->(e)
<-[:doctoralDegreeFrom]-(f)
RETURN a, b, c, d, e, f
```

$QP_{3,4}$ SQL

```
SELECT n1.id AS teacher, n2.id AS course, n3.id AS student,
  n4.id AS advisor
FROM nodes n1, rel_teacherOf rto, nodes n2, rel_takesCourse rtc,
  nodes n3, rel_advisor ra, nodes n4, rel_mastersDegreeFrom rmdf,
  nodes n5, rel_doctoralDegreeFrom rddf, nodes n6
WHERE n1.id = rto.sID AND rto.dID = n2.id
  AND n2.id = rtc.dID AND n3.id = rtc.sID
  AND n3.id = ra.sID AND n4.id = ra.dID
  AND n4.id = rmdf.sID AND n5.id = rmdf.dID
  AND n5.id = rddf.dID AND n6.id = rddf.sID
```

B.4 Paths restricted on node labels and edge types

$QP_{4,2}$ Cypher

```
MATCH (a:FullProfessor)-[:teacherOf]->(b:Course)
  <-[:takesCourse]-(c:UndergraduateStudent)
  -[:advisor]->(d:AssociateProfessor)
RETURN a, b, c, d
```

$QP_{4,2}$ SQL

```
SELECT a.id, b.id, c.id, d.id
FROM node_label a, rel_teacherOf r1, node_label b,
  rel_takesCourse r2, node_label c, rel_advisor r3, node_label d
WHERE a.value = 'FullProfessor' AND b.value = 'Course'
  AND c.value = 'UndergraduateStudent'
  AND d.value = 'AssociateProfessor'
  AND a.id = r1.sID AND r1.dID = b.id AND b.id = r2.dID
  AND r2.sID = c.id AND c.id = r3.sID AND r3.dID = d.id
```

$QP_{4,3}$ Cypher

```
MATCH (a:FullProfessor)-[:teacherOf]->(b:Course)
  <-[:takesCourse]-(c:UndergraduateStudent)
  -[:advisor]->(d:AssociateProfessor)
  -[:mastersDegreeFrom]->(e:University)
RETURN a, b, c, d, e
```

$QP_{4,3}$ SQL

```
SELECT a.id, b.id, c.id, d.id, e.id
FROM node_label a, rel_teacherOf r1, node_label b,
  rel_takesCourse r2, node_label c, rel_advisor r3, node_label d,
  rel_mastersDegreeFrom r4, node_label e
WHERE a.value = 'FullProfessor' AND b.value = 'Course'
  AND c.value = 'UndergraduateStudent'
  AND d.value = 'AssociateProfessor' AND e.value = 'University'
  AND a.id = r1.sID AND r1.dID = b.id AND b.id = r2.dID
  AND r2.sID = c.id AND c.id = r3.sID AND r3.dID = d.id
  AND d.id = r4.sID AND r4.dID = e.id
```