Approximate Querying for the Property Graph Language Cypher

George Fletcher*, Alexandra Poulovassilis†, Petra Selmer‡ and Peter T. Wood†
* Eindhoven Univ. of Technology, The Netherlands, Email: g.h.l.fletcher@tue.nl
† Birkbeck Knowledge Lab, Birkbeck, Univ. of London, UK, Email: {ap,ptw}@dcs.bbk.ac.uk
‡ Neo4j, London, UK, Email: petra.selmer@neo4j.com

Abstract—**Graph databases are well-suited to managing large, complex, dynamically evolving datasets. However, for data that is irregular and heterogeneous, it may be difficult to formulate queries that precisely capture a user's information seeking requirements. This points to the need for approximate query processing capabilities that can automatically make changes to a query so as to aid in the incremental discovery of relevant information. In this paper we motivate and explore techniques for providing such capabilities for the Cypher query language. This is the first time that query approximation has been investigated in the context of the property graph data model, which is becoming increasingly prevalent in research and industry.**

I. Introduction

Graph databases are well-suited to managing large, complex, dynamically evolving datasets due to the inherent flexibility and extensibility of graph data models and the efficiency and scalability of native graph storage implementations. However, for data that is irregular and heterogeneous in nature, it may be difficult to formulate queries that precisely capture a user's or developer's information seeking requirements, hence motivating the need for approximate query processing capabilities that can automatically make changes to a query so as to aid in the incremental discovery of relevant information.

Early work on approximating graph queries focussed on approximation of *regular path queries* (RPQs) [10] and *conjunctive regular path queries* (CRPQs) [13] through edit operations (addition, deletion, substitution) on the edge labels appearing in the query, each with an associated cost. CRPQs assist in the querying of graph-structured data by finding conjunctions of paths through the data that match given regular expressions over edge labels [2]. CRPQs are supported fully or in restricted form in contemporary practical languages such as SPARQL 1.1 [11], G-CORE [1], and the Cypher query language of the Neo4j graph DBMS.[1]

The benefits of supporting *approximate* CRPQs include: correcting users' erroneous queries, finding additional relevant answers, and generating new queries which may return unexpected results and bring new insights. Other recent work has explored flexible querying of graph data through the use of similarity measures — either structural or ontology-based — for basic graph pattern queries; we review this work in Section V. Our own recent work has focussed on the theoretical and practical aspects of approximating CRPQs [19] and approximating the property paths of SPARQL 1.1 queries [9], both in the context of a simple graph data model.

In contrast, in this paper we explore the benefits of providing such capabilities for the Cypher query language of the Neo4j graph DBMS, which supports a more intricate *property graph* data model [2]. Such capabilities are particularly useful for Neo4j in that it is a schema-free DBMS. Hence the user does not have immediate access to information such as the sets of node and edge labels used in the data they are querying, or constraints such as which labels occur on edges between pairs of nodes with particular labels. We also propose here for the first time data-driven ranking of approximate query answers, whereas earlier work used only the edit distance of the approximated query from the original query in order to rank query answers.

II. Approximating Cypher

Cypher is a graph query language originally developed as part of the Neo4j graph DBMS and now supported by several other products (e.g. SAP HANA Graph, Redis Graph, AgensGraph, Memgraph). Cypher 9 is the first version of the language developed under the auspices of the openCypher Implementors Group[2] and the version that we assume for our purposes here [8]. Cypher adopts a *property graph data model* which, in brief, comprises nodes (representing entities) and relationships (synonymous with edges) between pairs of entities; moreover, any number of properties, in the form of a set of key-value pairs, may be associated with a node or a relationship. We refer readers to the work of Francis et al. for details of the formal semantics of Cypher queries, and for an overview of Cypher's many applications in industry [8].

We focus here on Cypher's *path patterns* and their approximation. This is because path patterns are the core query construct for matching fragments of the data graph within a MATCH clause in Cypher. A Cypher path pattern, p, is of the form $\chi_1, \rho_1, \chi_2, ...\chi_{n-1}, \rho_{n-1}, \chi_n$ where the χ_i are *node patterns* and the ρ_i are *relationship patterns*. A node pattern matches a set of nodes in the data graph, while a relationship pattern matches a set of paths in the data graph.

A node pattern, χ, is a triple of the form (a, L, P) where a is an optional name for the node pattern in the query (i.e.

[1] https://neo4j.com

[2] http://www.opencypher.org

a variable), L a possibly empty set of node labels, and P a possibly empty map (set of $key \rightarrow value$ mappings).

A relationship pattern, ρ, is a quintuple of the form (d, a, T, P, I), where d specifies the directionality of the graph traversal ($\rightarrow$, $\leftarrow$ or $\leftrightarrow$), a is an optional name for this relationship pattern in the query, T a possibly empty set of relationship types (edge labels), P a possibly empty map, and I an optional interval indicating a lower and/or upper bound for the length of the paths matched in the data graph (the default being (1,1)). We note that the T and I components provide a limited form of RPQ capability, and hence a path pattern as a whole provides a limited form of CRPQ capability. On the other hand, the node patterns and the P components of the relationship patterns go beyond the syntax of CRPQs and are oriented towards Cypher's property graph data model.

To illustrate Cypher path patterns and the benefits of their approximation, consider a journalist looking into international financial crimes. For her investigations, she is using the excellent up-to-date Offshore Leaks financial social network data set, linking company officers and legal entities (i.e., companies) registered in the Bahamas.[3]

Suppose the investigator starts by looking for connections between intermediary companies and those companies which are in the same market as the parent company. She submits a Cypher query Q_1 containing the following path pattern in its MATCH clause:

```
(c1:Entity)-[:intermediary_of]->(c2:Entity)
           -[:related_company]->(c3:Entity)
```

In this query, c1, c2 and c3 are node names (variables), Entity is a node label, and intermediary_of and related_company are relationship types (edge labels). Unfortunately, there are no related_company relationships in the graph, and query Q_1 returns an empty result set. Such mis-specification of relationships is very common during exploratory analytics, and in such cases a **Substitution** edit operation is useful by allowing replacement of an erroneous edge label in the query by a different one. It is also useful for seeking additional, potentially relevant, answers through usage of different edge labels.

Following one substitution operation applied by the system to Q_1 (at a user-configurable cost), the investigator is presented with the results of query Q_1', with path pattern:

```
(c1:Entity)-[:intermediary_of]->(c2:Entity)
           -[:same_company_as]->(c3:Entity)
```

having over 15,000 matches in the graph, as well as the results of query Q_1'' having path pattern:

```
(c1:Entity)-[:intermediary_of]->(c2:Entity)
        -[:similar_company_as]->(c3:Entity)
```

with an additional 200 matches.

Previous investigations of a Substitution edit operation (and Insertion and Deletion) evaluate a set of approximated queries that have the same edit distance from the original query in an arbitrary order (see Section V). Here, we propose instead the application of *data-driven* heuristics to order equal-distance queries. For example, if the user elects to see the answers of the most selective queries first, then the answers of Q_1'' will be returned first and those of Q_1' will be returned last.

Suppose next that our investigator turns her attention to links between officers and companies, with special attention to the fact that companies often go by different names. She submits query Q_2 containing the following path pattern:

```
(o1:Officer)-[:officer_of]->(c1:Entity)
            -[same_company_as]->(c2:Entity)
```

returning around 1,200 results in the graph. However, she suspects that some officers are missing, and so requests approximation of the query. By **Insertion** of an additional edge into the start of the pattern, we are able to obtain additional officers by seeing also probable connections between officers:

```
(v)-[:probably_same_officer_as]-(o1:Officer)
        -[:officer_of]->(c1:Entity)
        -[:same_company_as]->(c2:Entity)
```

giving an additional 234 officers to investigate (matching the new variable v); and by seeing also officers having the same name:

```
(v)-[:same_name_as]-(o1:Officer)
        -[:officer_of]->(c1:Entity)
        -[:same_company_as]->(c2:Entity)
```

returning over 3800 additional officers to investigate[4]. In general, when there is an under-specification of a path pattern, Insertion is useful by allowing the addition of an edge label. It is also useful for seeking additional, potentially relevant, answers, e.g., there might have been some answers returned by the original query but, due to the heterogeneity of the data, there may be additional relevant ones still to be found. As with Substitution, the selection of different data-driven heuristics by the user allows the results of equal-distance approximated queries to be returned in specific orders.

Finally, trying to collect as many officers as possible, our investigator poses query Q_3 to find officers going under multiple names and the addresses of their companies:

```
(o2:Officer)<-[:same_name_as]-(o1:Officer)
      -[:registered_address]->(c1:Address)
        -[:same_address_as]->(c2:Address)
```

Unfortunately, this search only returns 1 match in the graph. Requesting query approximation, she is presented with the results of the following pattern obtained by **Deletion** of an edge (once again at a user-configurable cost):

```
(o1:Officer)
      -[:registered_address]->(c1:Address)
        -[:same_address_as]->(c2:Address)
```

[3] https://offshoreleaks.icij.org/

[4] In the concrete syntax of Cypher as used in the above queries, traversing edges in either direction is specified using $--$ rather than $\leftrightarrow$ as in the abstract syntax.

which leads to 5 further matches in the graph to investigate. In general, if there is an over-specification of a path pattern, Deletion is useful by allowing the removal of an edge label. It is also useful for seeking additional, potentially relevant, answers, similarly to the case of Insertion. Again, selection of different data-driven heuristics by the user allows the results of equal-distance queries to be returned in specific orders.

III. Approximating Cypher Patterns

We now provide an algorithm for undertaking approximate Cypher query evaluation, as illustrated above. It is a query rewriting algorithm, similar in approach to that proposed by Frosini et al. for approximation of SPARQL 1.1 property paths [9]. In particular, given a Cypher path pattern p and a user-specified maximum approximation cost, we incrementally build a list of pairs (p', c') such that p' is an approximated version of p and c' is the cost of deriving p' from p (i.e. c' is the sum of the costs of the edit operations applied to p to obtain p'). This list of pairs is sorted in non-decreasing order of the approximation costs, c'. The approximated patterns p' are evaluated in this order (using the normal Cypher evaluation [8]) so that query answers are returned in order of non-decreasing approximation cost.

Finer-grained ordering can be applied to the evaluation of approximated patterns that have the same approximation cost, based on their selectivity, e.g. most or least selective first, as illustrated in the above examples. The user may elect to view the results of the most selective queries first if a small number of answers are expected and the user would prefer to explore each answer in detail. Conversely, the user may elect to view the results of the least selective queries first if a large number of answers are expected and desired.

Algorithm 1 rewrites a path pattern p of the form $\chi_1, \rho_1, \chi_2, ...\chi_{n-1}, \rho_{n-1}, \chi_n$ into a list of pairs (p', c'), where p' is an approximated version of p and c' is the cost of deriving this approximation from p. We first initialise the variables $oldGen$ and $pairs$ to contain just the input path pattern p. For each path pattern p in $oldGen$ we apply one step of approximation to each of its node patterns and to each of its relationship patterns and we assign the cost of applying that approximation to the resulting path pattern. Each of these path patterns is added to the set-valued variable $newGen$ and to the list-valued variable $pairs$ (ordered by non-decreasing cost) provided the maximum cost limit $maxCost$ is not exceeded. If the same path pattern is generated more than once, only the one with the lowest cost is retained within $pairs$.

So far, these approximations have not changed the length of the path pattern p; they have only modified one of its components (a node pattern or a relationship pattern), generalising the Substitution edit operation illustrated earlier. We also generate a set of path patterns that are derived from p by removing a relationship pattern and adjacent node patterns (cf. the Deletion operation illustrated above). These removals of relationship patterns from p are captured by the function $removeRP$ in Algorithm 1.

Algorithm 1: Path Pattern Rewriting

```
Input: path pattern $p$, maximum edit cost $maxCost$
Output: list of path pattern/cost pairs, ordered by
        non-decreasing cost
$oldGen$ := $\{(p, 0)\}$
$pairs$ := $[(p, 0)]$
while $oldGen \neq \emptyset$ do
    $newGen$ := $\emptyset$
    foreach $(p, c) \in oldGen$ do
        foreach node pattern $np \in p$ do
            foreach $(p', c') \in applyNPApprox(p, np)$ do
                update $newGen$ and $pairs$
        foreach relationship pattern $rp \in p$ do
            foreach $(p', c') \in applyRPApprox(p, rp)$ do
                update $newGen$ and $pairs$
        foreach $(p', c') \in removeRP(p) \cup addRP(p)$ do
            update $newGen$ and $pairs$
    $oldGen$ := $newGen$
return pairs
```

Algorithm 2: applyNPApprox

```
Input: path pattern $p$, node pattern $np$ within $p$
Output: set of path pattern/cost pairs
S := $\emptyset$
foreach $(np', cost) \in approxNP(np)$ do
    $p'$ := replace $np$ by $np'$ in $p$
    $S$ := $S \cup \{(p', cost)\}$
return $S$
```

Also generated is a set of path patterns that are derived from p by the addition of a new relationship pattern and adjacent node pattern (cf. the Insertion operation illustrated above). The value of each new node pattern is $(v, \emptyset, \emptyset)$, where v is a new variable not appearing anywhere else in the query. For each relationship type t in the graph, the new relationship pattern is $(\leftrightarrow, v, \{t\}, \emptyset, (1, 1))$, where v is again a new variable not appearing anywhere else in the query. These insertions of relationship and node patterns into p are captured by the function $addRP$ in Algorithm 1. This process of producing a new generation of path patterns from the current set continues until no new-generation path patterns can be produced.

Algorithm 1 calls functions $applyNPApprox$ (Algorithm 2) and $applyRPApprox$ (Algorithm 3) to apply one step of approximation to a node pattern or a relationship pattern of p, respectively. These two algorithms themselves call Algorithm 4 and Algorithm 5, respectively, to generate an approximated node pattern or an approximated relationship pattern, respectively.

In Algorithm 4, given the label set L of a node pattern, the function $approxLabelSet$ returns a set of pairs of label

```
Algorithm 3: applyRPApprox
  Input: path pattern $p$, relationship pattern $rp$ within $p$
  Output: set of path pattern/cost pairs
  S := $\emptyset$
  foreach $(rp', cost) \in approxRP(rp)$ do
      $p'$ := replace $rp$ by $rp'$ in $p$
      $S$ := $S \cup \{(p', cost)\}$
  return $S$
```

```
Algorithm 4: approxNP
  Input: node pattern $np = (a, L, P)$
  Output: set of node pattern/cost pairs
  return $\{((a, L', P'), c1 + c2) \mid (L', c1) \in$
      $approxLabelSet(L) \wedge (P', c2) \in approxMap(P)\}$
```

sets and costs: the original label set L with cost 0, and each set obtained from L by removing a label, with an associated cost. Insertion and substitution are not applied to L for the following reasons. Because a graph node will match a node pattern only if its label(s) match *all* of the labels in L, inserting a new label will not provide the user with additional answers. Furthermore, substituting a node label l by a different label will be subsumed by removing l from L (i.e., the latter will match at least as many nodes as the former). Similarly, given map P, the function $approxMap$ returns a set of pairs of maps and costs: the original map P with cost 0, and each map obtained from P by removing a mapping, with an associated cost. As for the function $approxLabelSet$, it is not necessary to support either insert or substitution edit operations on P.

In Algorithm 5, given the relationship types component T of a relationship pattern, the function $approxRelTypeSet$ returns a set of pairs of type sets and costs: the original type set T with cost 0, and each set comprising a single type different from each of the types in T, with an associated cost. Since T is interpreted as a disjunction, it is not necessary to support a delete edit operation on T. Also, adding a type t to T will return the same set of new matches as replacing T by $\{t\}$. So the insertion edit operation is not needed. Given an interval I, the function $approxInterval$ returns the original interval I with cost 0, as well as all intervals (each with a cost) obtained by expanding a path length by 1; it is not necessary to support the contraction of a path length, as this will not generate any new matches against the graph.

IV. Evaluation

We have conducted a preliminary performance evaluation of our query approximation algorithms, using the Paradise Papers dataset[5] (which, at around 1.5GB, is much larger than the Offshore Leaks database we referred to in our earlier examples). The Paradise Papers data comprises 867,931 nodes and 1,657,838 edges. It

```
Algorithm 5: approxRP
  Input: relationship pattern $rp = (d, a, T, P, I)$
  Output: set of relationship pattern/cost pairs
  return $\{((d, a, T', P', I'), c1 + c2 + c3) \mid$
             $(T', c1) \in approxRelTypeSet(T) \wedge$
             $(P', c2) \in approxMap(P) \wedge$
             $(I', c3) \in approxInterval(I)\}$
```

uses the following five node labels: `Officer`, `Entity`, `Intermediary`, `Address` and `Other`, and the following six edge labels: `officer_of`, `registered_address`, `connected_to`, `intermediary_of`, `same_name_as` and `same_id_as`. Each node and edge has a number of properties. All nodes have a `name` property which is used in most of our queries and for which indexes are created in the Neo4j database. Inspired by the example queries over this dataset described in several Neo4j blogposts[6,7,8], we defined the 10 queries listed in Table I for use in our evaluation.

For each query, we generated the set of approximated queries resulting from one step of approximation (i.e. we set the cost of each approximation to 1, and the maximum edit cost input to Algorithm 1 to 1). We ran the original query and each of its approximations six times on an otherwise idle machine, discarding the first timing and averaging the next five. The queries were executed on a MacBook Pro (2016) running MacOS 10.14.6, at 2.9 GHz, with 16 GB RAM and using Version 3.5.8 of Neo4j (Enterprise Edition).

Table II shows, for each query: the number of approximated queries; the number of these that return non-empty results; the number of results returned by these queries, in ascending order of result size; and the execution time of the corresponding query. All queries were initially run using a 2-minute timeout. Any queries that timed out were then modified to just count the number of results (entries in Table II with a '?' for their execution time only) Those that still timed out, were run with a timeout of 10 minutes. Queries that have a '?' for both their result count and their execution time are ones that timed out at 10 minutes.

We make the following observations about each of the 10 queries:

Query Q1 returns no results because `related_to` is not a valid edge label. The edge label `connected_to` is the semantically closest valid label. When this label is substituted for `related_to` in Q1, the query returns 2114 results.

Query Q2 returns no results because edges labelled `intermediary_of` go from nodes with label `Other`, rather than `Officer`, to nodes with label `Entity`. When the node label `Officer` is removed, the query returns 68539 results.

Query Q3 returns only one result because most `officer_of` edges go from `Officer` nodes to `Entity` nodes, not other `Officer` nodes. Removing the `Officer`

[5] https://offshoreleaks.icij.org/pages/database

[6] https://neo4j.com/blog/depth-graph-analysis-paradise-papers/

[7] https://neo4j.com/blog/analyzing-paradise-papers-neo4j/

[8] https://offshoreleaks-data.icij.org/offshoreleaks/neo4j/guide/datashape.html

TABLE I
THE TEN QUERIES USED IN THE EVALUATION.

Q1:	MATCH (e1)-[:intermediary_of]->(e2)-[:related_to]-(e3) RETURN e1.name, e2.name, e3.name
Q2:	MATCH (o:Officer)-[:intermediary_of]->(e1:Entity)-[:officer_of]-(e2) RETURN o.name, e1.name , e2.name
Q3:	MATCH (o1:Officer)-[:officer_of]->(o2:Officer)-[:registered_address]->(a:Address) RETURN o1.name, o2.name, a.name
Q4:	MATCH (o:Officer {name: ’The Duchy of Lancaster’})-[*1..2]-(e:Officer) RETURN o.name, e.name
Q5:	MATCH (i:Intermediary)-[:connected_to]->(e:Entity:Intermediary) WHERE i.name CONTAINS ’Appleby’ RETURN i.name, e.name
Q6:	MATCH (a:Address {country: ’US’})--(o:Officer)--(e:Entity) RETURN e.jurisdiction_description AS jurisdiction, COUNT(*) AS num ORDER BY num DESC
Q7:	MATCH (o1:Officer)-[:officer_of]->(e1:Entity)-[:registered_address]->(a:Address) <-[:registered_address]-(e2:Entity)<-[:officer_of]-(o1) WHERE a.name CONTAINS "Canon’s Court; PO Box" RETURN o1.name, e1.name, a.name, e2.name
Q8:	MATCH (o1:Officer)-[:registered_address]->(a:Address)<-[:registered_address]-(e2:Entity) WHERE a.name CONTAINS "Canon’s Court; PO Box" AND o1.name=e2.name RETURN o1.name, a.name, e2.name
Q9:	MATCH (o:Officer {name: ’The Duchy of Lancaster’})-[*1..3]-(e:Officer) RETURN o.name, e.name
Q10:	MATCH (a:Address)--(o:Officer)--(e:Entity) RETURN e.jurisdiction_description AS jurisdiction, COUNT(*) AS num ORDER BY num DESC

TABLE II
NUMBER OF ONE-STEP APPROXIMATIONS OF EACH QUERY; NUMBER OF THOSE RETURNING NON-EMPTY RESULTS; NUMBER OF RESULTS EACH RETURNS (IN ASCENDING ORDER OF RESULT SIZE); AND CORRESPONDING EXECUTION TIMES

Query	no. of approx. queries	no. with non-empty results	no. of query results, and query execution times (s)										
Q1	40	6	193	2114	2316	17462	68539	99695					
			0.465	0.457	0.466	0.516	0.664	0.609					
Q2	39	5	3893	9802	68539	1030945	12886966						
			0.226	0.241	0.442	2.128	?						
Q3	39	9	1	1	866	4577	16734	25330	227090	1285441	22944525		
			0.143	0.151	0.181	0.157	0.209	0.280	0.547	4.332	?		
Q4	24	10	7	7	7	10	12	76	83	128	350008	28876444	
			0.002	0.004	77.3	0.001	0.002	0.002	0.006	0.002	0.323	?	
Q5	27	1	50										
			0.005										
Q6	30	2	35	38									
			3.75	1.3									
Q7	65	11	1	16	16	16	40	40	142	1693	1790	12852	12852
			0.038	0.042	0.047	0.486	0.043	0.044	0.044	0.318	0.065	0.106	0.110
Q8	39	10	1	1	2	3	4	4	4	8	9	11	
			0.039	0.042	0.037	0.154	0.037	0.041	0.538	0.039	0.039	0.038	
Q9	24	10	63	83	314	546	907	2348	2894	350008	?	?	
			0.002	0.002	0.002	0.004	0.004	0.012	0.010	0.301	?	?	
Q10	29	20	1	1	1	5	13	16	16	17	30	33	
			0.302	0.337	0.348	13.9	0.632	0.468	2.85	0.251	7.06	6.98	
			35	35	36	36	37	38	38	?	?	?	
			22.2	118	0.495	3.48	4.56	1.24	65.7	?	?	?	

label from o1 or the Address label from a yield the same single answer as the original query (the first two entries with 1 result for Q3 in Table II). Possible alternative intentions for this query include replacing officer_of by same_name_as (yielding 866 results) or same_id_as (yielding 4577 results), or by removing the Officer label from o2 (yielding 1285441 results).

Query Q4 returns 7 officers who are connected by one or two relationships to "The Duchy of Lancaster". Removing the node label Officer from e uncovers additional funds (since they are entities not officers), among the 10 answers reported in Table II. Inserting officer_of before o or before e returns the same 7 answers as the original query. Applying interval approximation increases the number of hops specified in the interval from 2 to 3 and yields 76 new results not returned by the original query.

Query Q9 is the approximated version of Q4 that has 3 hops specified in the interval. Applying interval approximation to Q9 increases the number of hops to 4, yielding 2894 results. Removing the Officer node label from o causes the query to time out, as does removing the property restriction on o.

Query Q5 returns no results because there are no nodes labelled with both Entity and Intermediary. Removal of the Intermediary label produces 50 answers.

Query Q6 returns no results because country is an incorrect property name. Removing this mapping yields 35 jurisdictions, while removing the entire first relationship pattern yields a further 3 jurisdictions (within the 38 returned). It turns out that one of these "new" jurisdictions is simply a modified spelling of one of the original ones.

Query Q10 is the approximated version of Q6 that yielded 35 results. By removing the node `a` and its following relationship or the `Address` label from `a`, the same 38 jurisdictions as above are returned. Removal of the relationship pattern containing `Officer` yields 36 results, including two new jurisdictions. Removal of the node label `Officer` from `o` yields 37 results, including the final undiscovered jurisdiction among the 41 in the data.

Query Q7 returns 16 results. Removing the relationship pattern `(e2:Entity)<-[:officer_of]` produces a query which finds officers registered at the same address as entities of which they are officers, yielding one new answer.

Query Q8 returns only 4 answers. By adding `(v)-[:officer_of]` before `(e2:Entity)`, the approximated query finds entities whose officers are registered at the address, yielding 3 new results.

We see from Table II that there can be a large difference in the number of results returned by the least and most selective approximated queries, pointing to the potential usefulness of our proposed data-driven heuristics for ordering the results of equal-cost queries. Also, the query timings are encouraging from a performance perspective, in showing that the user would not need to wait a long time for results from most approximated queries to be returned. Finally, all of the different types of approximations described in SectionIII are potentially useful in the sense that, over our 10 test queries, all of them yield approximated queries that return non-empty results.

V. Related Work

Query approximation and relaxation, sometimes referred to as *flexible* querying, has been studied in the context of many different data models and query languages.

Early work on flexible querying for semi-structured data was undertaken in [14], where flexible answers corresponded to paths whose set of edge labels contained those appearing in the query. More generally, [10] explored approximate matching of single-conjunct regular path queries (RPQs) using edit operations (addition, deletion, substitution) on the edge labels appearing in the query.

Several proposals for flexible RDF querying use similarity measures between resources, constants or structures to retrieve additional answers [5], [12], [15]. In [17] knowledge of the semantic relationships between nodes is used for approximate query matching, while [4] describes a framework for cost-aware querying of weighted RDF data through predicates that express flexible paths between nodes. Extending SPARQL with keyword search capabilities, together with IR-style ranking of query answers, is proposed in [7]. Relaxing RDF queries based on user preferences is investigated in [3], [6].

A topic related to flexible querying is that of so-called "why-not queries", where users wish to find out why particular answers are not returned by an original query [20]. However, in order for the system to answer a why-not query, the user must provide *specific* answer tuples (or mappings) that do not appear among the original results.

VI. Concluding Remarks

In this paper we have motivated, specified, and illustrated approximate querying for the path patterns of the Cypher property graph query language. We have described a preliminary performance study showing the promise of our approach, both in terms of returning potentially useful answers for the user and in terms of query performance.

Clearly a much more extensive performance study needs to be undertaken. This could identify where further research may be required in terms of optimisation techniques for evaluating approximate Cypher queries, e.g. using a graph summary [16] to avoid generating queries that are unsatisfiable.

Also important is the design of graphical interfaces to support users' interaction with such query approximation facilities — some preliminary ideas are given in [18]. Finally, extending approximation to a wider subset of the Cypher language, beyond just its path patterns, is also an open area of research.

References

[1] R. Angles et al. G-CORE: A core for future graph query languages. In *SIGMOD*, pages 1421–1432, 2018.

[2] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. *Querying graphs*. Morgan & Claypool, 2018.

[3] P. Buche, J. Dibie-Barthélemy, and H. Chebil. Flexible SPARQL querying of web data tables driven by an ontology. In *FQAS*, pages 345–357, 2009.

[4] J. P. Cedeño and K. S. Candan. R2DF framework for ranked path queries over weighted RDF graphs. In *Web Int., Mining and Semantics*, pages 40:1–40:12, 2011.

[5] R. De Virgilio, A. Maccioni, and R. Torlone. A similarity measure for approximate querying over RDF data. In *EDBT*, pages 205–213, 2013.

[6] P. Dolog, H. Stuckenschmidt, H. Wache, and J. Diederich. Relaxing RDF queries based on user and domain preferences. *J. Intell. Inf. Syst.*, 33(3):239–260, 2009.

[7] S. Elbassuoni, M. Ramanath, and G. Weikum. Query relaxation for entity-relationship search. In *ESWC*, pages 62–76, 2011.

[8] N. Francis et al. Cypher: An evolving query language for property graphs. In *SIGMOD*, pages 1433–1445, 2018.

[9] R. Frosini, A. Calì, A. Poulovassilis, and P. T. Wood. Flexible query processing for SPARQL. *Semantic Web*, 8(4):533–563, 2017.

[10] G. Grahne and A. Thomo. Regular path queries under approximate semantics. *Annals of Math. and Art. Int.*, 46(1-2):165–190, 2006.

[11] S. Harris and A. Seaborne, editors. *SPARQL 1.1 Query Language*, W3C Recom., 21 March 2013.

[12] A. Hogan, M. Mellotte, G. Powell, and D. Stampouli. Towards fuzzy query-relaxation for RDF. In *ESWC*, pages 687–702, 2012.

[13] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Ranking approximate answers to semantic web queries. In *ESWC*, pages 263–277, 2009.

[14] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *PODS*, pages 40–51, 2001.

[15] C. Kiefer, A. Bernstein, and M. Stocker. The fundamentals of iSPARQL: a virtual triple approach for similarity-based semantic web tasks. In *ISWC*, pages 295–309, 2007.

[16] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: A survey. *ACM Comput. Surv.*, 51(3):62:1–62:34, June 2018.

[17] F. Mandreoli, R. Martoglia, G. Villani, and W. Penzo. Flexible query answering on graph-modeled data. In *EDBT*, pages 216–227, 2009.

[18] A. Poulovassilis. Applications of flexible querying to graph data. In *Graph Data Management: Fundamental Issues and Recent Developments*, pages 97–142. Springer, 2018.

[19] A. Poulovassilis, P. Selmer, and P. T. Wood. Approximation and relaxation of semantic web path queries. *J. Web Semantics*, 40:1–21, 2016.

[20] M. Wang, J. Liu, B. Wei, S. Yao, H. Zeng, and L. Shi. Answering why-not questions on SPARQL queries. *Knowledge and Information Systems*, 58(1):169–208, Jan 2019.