Approximate Querying Over Property Graphs

Alexandra Poulovassilis
www.dcs.bbk.ac.uk/~ap

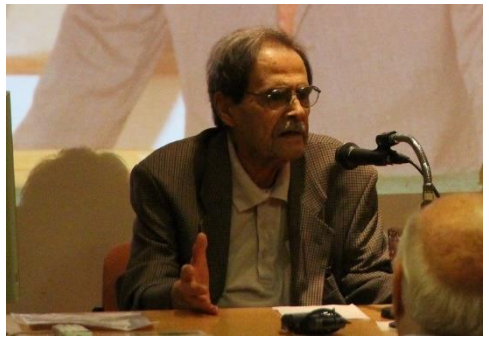
Joint work with George Fletcher (Eindhoven University of Technology), Petra Selmer (Neo4j) and Peter Wood (Birkbeck, University of London)

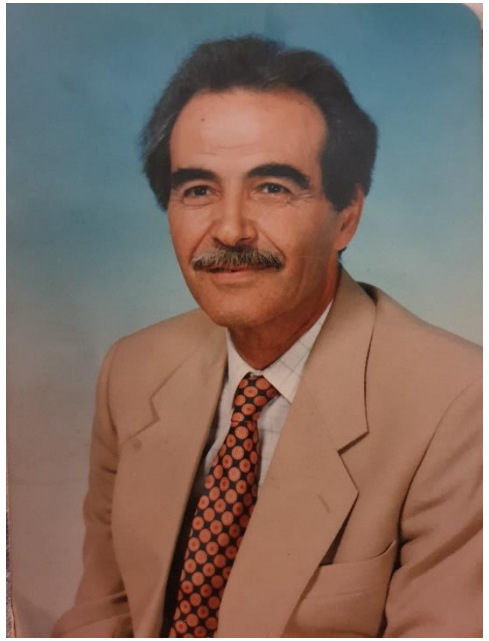
Athena Research Centre, Athens
20/02/2020

This talk is based on work reported in “Approximate querying for the property graph language Cypher”, G.Fletcher, A.Poulovassilis, P.Selmer, P.Wood. In: Proc. IEEE Big Data, Los Angeles, Dec. 2019

Knowledge Lab

Birkbeck
UNIVERSITY OF LONDON





In memoriam

This talk is dedicated to the memory of my father
Alexander Poulovassilis
Αλέξανδρος Πουλοβασίλης
1928 – 2020

Principal of the Agricultural University of Athens (AUA) 1982-1991
Professor of Agricultural Hydraulics at the AUA 1977-1996
Emeritus Professor 1996-2020
Fellow of the Agricultural Academy of Greece
President of the Agricultural Academy of Greece 2010-2012

Ας είναι ελαφρύ το χώμα που τον σκεπάζει
Let the soil that covers him be light

Talk Outline

1. Motivation
2. Previous work on graph query approximation
3. Approximating Cypher Path Patterns
4. Empirical evaluation
5. Conclusions and future work

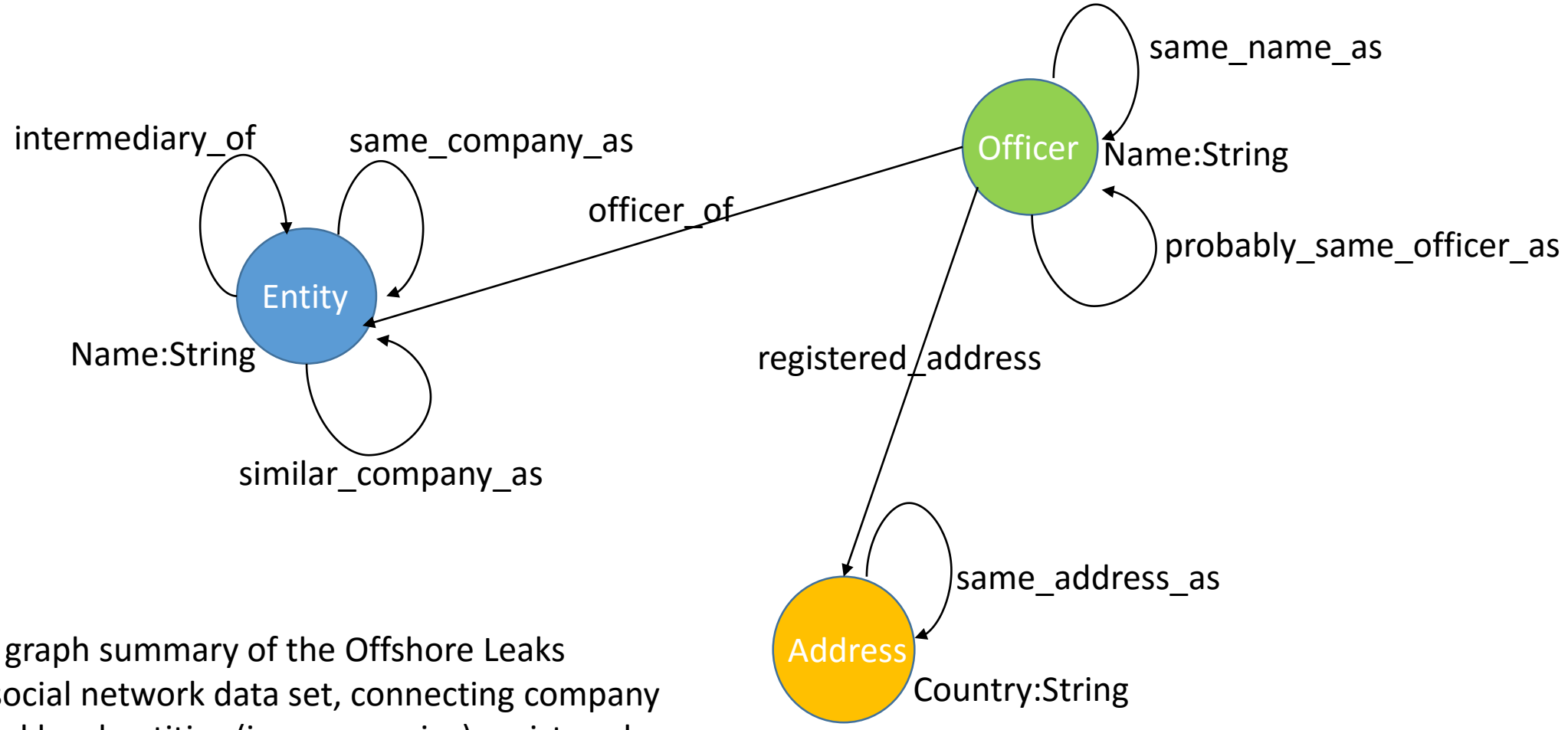
1 Motivation

- Graph databases are well-suited to managing large, complex, dynamically evolving data:
  - inherent flexibility/extensibility of graph data models
  - efficiency/scalability of graph storage implementations
- There are many contemporary graph DBMS and query languages
  - e.g. Neo4j and Cypher/openCypher
- For data that is irregular and heterogeneous, it may be difficult to formulate queries that precisely express a user's information seeking requirements
- Also, for exploratory or investigative querying, users may benefit from support in formulating “similar” queries
- One way of tackling this is through *query approximation* for graph query languages such as Cypher

Motivation

- Cypher was developed as part of Neo4j and is now supported by several other products (SAP HANA Graph, Redis Graph, AgensGraph, Memgraph)
- Cypher 9 is the first version developed under the auspices of the openCypher Implementors Group and is the version that we assume

- Cypher adopts a *property graph data model* which comprises
  - nodes, representing entities;
  - each node can have zero or more labels (similar to entity types);
  - relationships (synonymous with edges) between pairs of entities;
  - each relationship can have at most one type (i.e. edge label);
  - properties, in the form of key-value pairs, any number of which may be associated with a node or a relationship.



Fragment graph summary of the Offshore Leaks financial social network data set, connecting company officers and legal entities (i.e. companies) registered in the Bahamas. Compiled by the *International Consortium of Investigative Journalists,* see *https://offshoreleaks.icij.org/*

Example Cypher Query

```
MATCH (o1:Officer)-[:officer_of]->(c1:Entity)-[same_company_as]->(c2:Entity)
RETURN o1.name, c1.name, c2.name
```

o1, c1, c2 match triplets of nodes such that

- o1 has label Officer
- c1 and c2 have label Entity
- there is an edge from o1 to c1 labelled officer_of
- there is an edge from c1 to c2 labelled same_company_as
- returns values of the name properties of o1, c1, c2

We focus on approximating Cypher's *path patterns* since they are the core construct for matching fragments of the data graph within a MATCH clause

2 Previous work on graph query approximation

- Grahne & Thomo, Annals of Mathematics and Artificial Intelligence 2006, first proposed approximate matching of *regular path queries* (RPQs) using edit operations (addition, deletion, substitution) on the edge labels in the query, each with an associated cost
- Hurtado, Poulovassilis, Wood, ESWC 2009, extended this to *conjunctive regular path queries* (CRPQs)
- Poulovassilis & Wood, ISWC 2010, further extended the approach to encompass also *query relaxation* based on ontological information
- CRPQs are a key graph querying construct supported fully or in restricted form in contemporary languages such as SPARQL 1.1, G-CORE and Cypher

Previous work on graph query approximation

- More recently, we have investigated:
  - theoretical and practical aspects of approximating CRPQs (Poulovassilis et al, JWS 2016)
  - approximating the property paths of SPARQL 1.1 (Frosini et al, SWJ 2017)

  in the context of a simple graph data model
- Our BigData 2019 paper reviews other related work on, e.g.: graph query relaxation based on user preferences, approximate graph query answering based on similarity matching, keyword search, and answer ranking
- Ours is the first work to investigate *query approximation in the context of the property graph data model*
- We also propose 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 to rank query answers

3 Approximating Cypher Path Patterns

- A Cypher path pattern, p, is of the form

$$\chi_1 , \rho_1, \chi_2, \ldots \chi_{n-1}, \rho_{n-1}, \chi_n$$

the χ_i are *node patterns*, matching a set of nodes in the data graph

the ρ_i are *relationship patterns*, matching a set of paths in the graph

- A node pattern is a triple of the form (a,L,P) where:

a is an optional name for the pattern within the query (i.e. a variable)

L is a possibly empty set of node labels

P is a possibly empty map (i.e. set of key→value mappings)

Approximating Cypher Path Patterns

- A relationship pattern is a quintuple of the form (d,a,T,P,I) where:

d specifies the directionality of the edge traversal (→, ←, ←→)

a is an optional name for this pattern within the query

T a possibly empty set of relationship types (edge labels)

P a possibly empty map

I an optional interval indicating a lower and/or upper bound for the length of the paths to be matched in the data graph (with default (1,1))

- The T and I components provide *a limited form of RPQ capability*; hence a path pattern as a whole provides *a limited form of CRPQ capability*
- However, Cypher's *node patterns and its P components of relationship patterns go beyond the syntax of CRPQs*, addressing querying requirements over the property graph data model

Example 1

Referring to the Offshore Leaks dataset, the path pattern

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

has no matches as there are no related_company relationships in the graph

Approximations resulting from *one edge label substitution* operation – at a user configurable cost – include:

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

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

with 15,000 and 200 matches, respectively

Example 2

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

has around 1,200 matches. Approximations resulting from *insertion of one node pattern/relationship pattern pair* include:

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

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

with 234 and 3800 additional matches, respectively

Example 3

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

has only 1 match. Approximations resulting from *deletion of one node pattern/relationship pattern pair* include:

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

with 5 additional matches

Approximate Query Generation and Evaluation

- Our approximate Cypher query evaluation is based on *query rewriting*
- Given a path pattern p and a user-specified maximum approximation cost c, 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. 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 c'
- The approximated patterns p' are evaluated in this order, using normal Cypher evaluation
- Finer-grained ordering can be applied to approximated patterns with the same approximation cost, based on their *selectivity*, e.g. most or least selective first:
  - a user may prefer to view the results of the *most selective* queries first if a small number of answers are expected and each answer needs to be explored in detail
  - conversely, a user may prefer to view the results of the *least selective* queries first if a large number of answers are desired

Algorithm 1: **Path Pattern Rewriting**

```
Input: path pattern p, maximum edit cost c
Output: list of path pattern/cost pairs, ordered by non-decreasing cost
oldGen := {(p, 0)}
pairs := [(p, 0)]
while oldGen != {}  do
    newGen := {}
    foreach (p, cost) ϵ oldGen do
        foreach node pattern np ϵ p do
            foreach (p', cost') ϵ applyNPApprox(p,np) do
                totCost := cost + cost'
                if totCost  ≤ c then
                    newGen := newGen U {(p',totCost)}
                    pairs := addTo(pairs,(p',totCost))
        foreach relationship pattern rp ϵ p do
            foreach (p', cost') ϵ applyRPApprox(p,rp) do
                totCost := cost + cost'
                if totCost ≤ c then
                    newGen := newGen U {(p',totCost)}
                    pairs := addTo(pairs,(p',totCost))
        foreach (p', cost') ϵ removeRelPattern(p) U addRelPattern(p) do
            totCost := cost + cost'
            if totCost ≤ c then
                newGen := newGen U {(p',totCost)}
                pairs := addTo(pairs,(p',totCost))
    oldGen := newGen
return pairs
```

removeRelPattern(p)

- yields the set of path patterns that can be obtained from p by removing an adjacent node pattern/relationship pattern pair

addRelPattern(p)

- yields the set of all path patterns that can be obtained from p by adding into any position within p a new node pattern/relationship pattern pair

χ_{new} , ρ_{new}

where χ_{new} is the triple (v,{},{}), with v a new variable

and ρ_{new} is a quintuple (←→,v,{t},{},(1,1)), with v a new variable and t a relationship type appearing in the graph

Algorithm 2: **applyNPApprox**

Input: path pattern p, node pattern np within p

Output: set of path pattern/cost pairs

S := {}

foreach (np',cost) ϵ **approxNP**(np) do

    p' := replace np by np' in p

    S := S U {(p',cost)}

return S

Algorithm 3: **applyRPApprox**

Input: path pattern p, relationship pattern rp within p

Output: set of path pattern/cost pairs

S := {}

foreach (rp',cost) ϵ **approxRP**(rp) do

    p' := replace rp by rp' in p

    S := S U {(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) | (L',c1) ϵ **approxNodeLabelSet**(L) AND
(P',c2) ϵ **approxMap**(P)}

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) | (T',c1) ϵ **approxRelTypeSet**(T) AND
(P',c2) ϵ **approxMap**(P) AND
(I',c3) ϵ **approxInterval**(I)}

Algorithm 6: approxNodeLabelSet

Input: labelset L

Output: set of label/cost pairs

return {(L,0)} U {(L', ld) | L' is obtained from L by removing a label}

// where ld is the cost of removing one label from a set of labels L

Algorithm 7: approxMap

Input: map P

Output: set of map/cost pairs

return {(P,0)} U

{(P',md) | P' is obtained from P by removing a mapping k→v}

// where md is the cost of removing a mapping k→v from a map P

Algorithm 8: approxRelTypeSet

Input: reltypeset T

Output: set of relationship type/cost pairs

if T = {} then return {(T,0)}

return {(T,0)} U { ({t},ts) | t is a relationship type not in T}

// ts is the cost of replacing T by a single type different from each of the types in T

Algorithm 9: approxInterval

Input: path length interval I

Output: set of interval/cost pairs

S := {(I,0)}

if I is (1,∞) then return S

if I is (min, ∞) then S := S U {((min-1, ∞), c)}

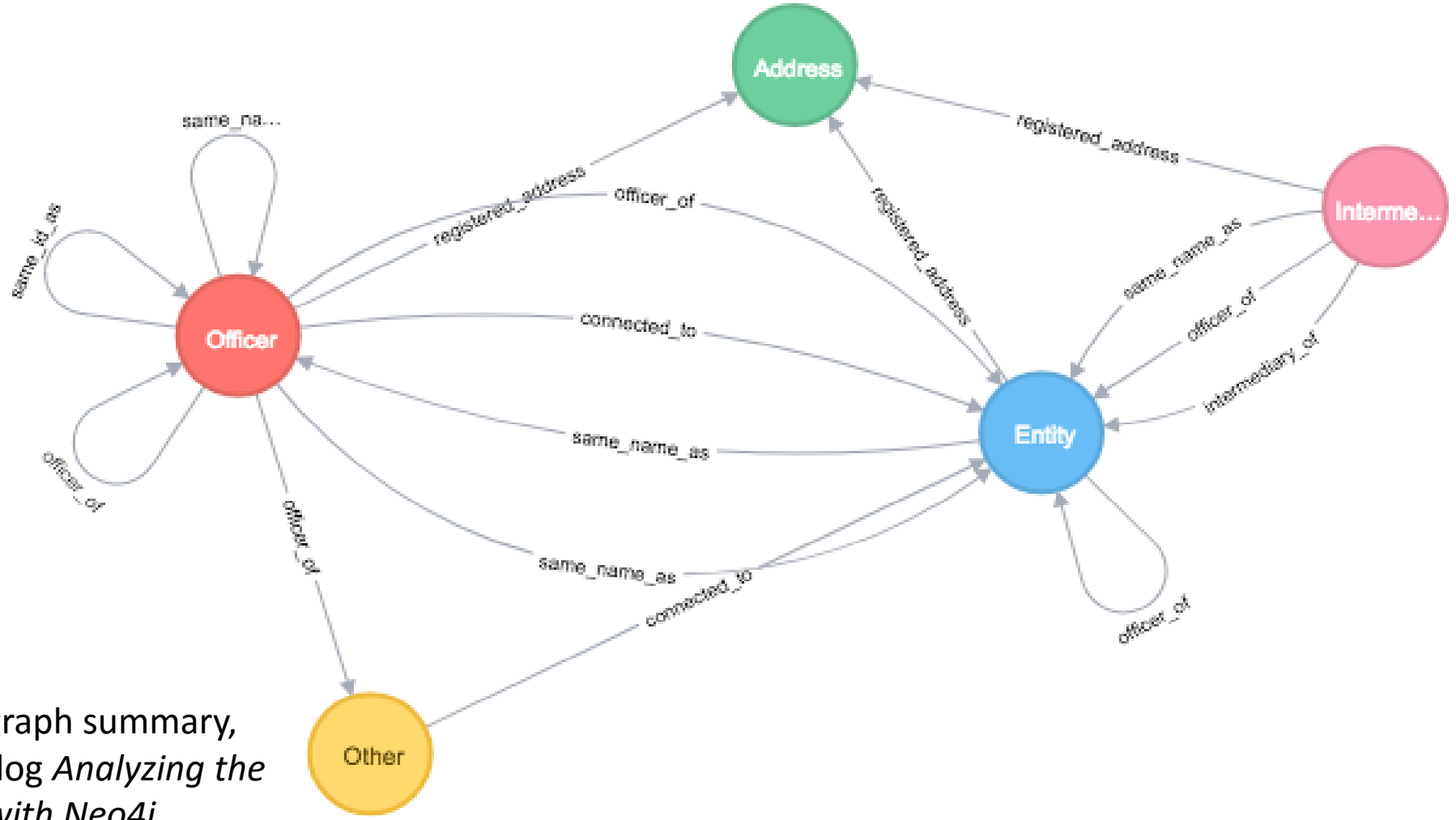
else if I is (1,max) then S := S U {((1,max + 1),c)}

else if I is (min,max) then S := S U {((min-1,max),c),((min,max+1),c)}

return S // c is the cost of expanding a path length interval I by 1

4 Empirical Evaluation

- We have conducted a preliminary performance evaluation of our query approximation algorithms using the Paradise Papers dataset https://offshoreleaks.icij.org/pages/database
- This has 867,931 nodes and 1,657,838 edges
- 5 node labels: Officer, Entity, Intermediary, Address, Other
- 6 edge labels: officer_of, registered_address, connected_to, intermediary_of, same_name_as, same_id_as
- All nodes have a name property for which indexes are created in the Neo4j database



Paradise papers graph summary, from the Neo4j blog *Analyzing the Paradise Papers with Neo4j https://neo4j.com/blog/analyzing-paradise-papers-neo4j/*

Test Queries

- Inspired by the example queries at several Neo4j blogs, we defined 10 test queries, listed in Table I
- For each query, we generated the set of approximated queries resulting from one step of approximation, i.e.

  we set the cost of all approximations to be 1

  we set the maximum edit cost input to Algorithm 1 to also be 1
- For the timings we recorded time elapsed from query submission to the server (on the same machine as the client) till all results were available to the client
  - Queries were initially run using a 2-minute timeout. Any queries that timed out were then modified to just *count* the number of results, rather than returning them all. Any queries that still timed out at 2 minutes were finally run with a timeout of 10 minutes, again attempting to count the results
- 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 I: THE 10 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

Query timings

- Table II shows, for each of the 10 exact queries:
  - 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
  - the execution time of the corresponding query
- Queries showing a number of results but ? for execution time timed out at 2 minutes but could return a count of the number of answers within that time
- Queries showing ? for both result count and execution time timed out at 10 minutes without being able to return a result size
- We see from Table II that
  - the number of one-step approximated queries ranges from 24 to 65
  - the number of these generating non-empty answers ranges from 1 to 20
  - The number of results they return varies from just a handful to millions
  - of the 84 queries returning non-empty answers, 61 ran in under 1s, a further 8 in under 5s, 7 up to 118s, and 8 timed out at two minutes

TABLE II: RESULTS AND TIMINGS OF THE 1-STEP APPROXIMATE QUERIES, IN ASCENDING ORDER OF RESULT SIZE

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

Numbers of results

- Table III shows, for each of the 10 exact queries:
  - the number of results returned by its exact form
  - the aggregated results of each subset of its approximated queries according to the type of approximation that has been applied
- for each exact query, some types of approximation are not applicable, indicated by - in the table
- a + after a number of results indicates that one or more approximated queries timed out at 2 minutes; for these queries the numbers of results are not included in the summation
- the numbers in parentheses in the table indicate the number of approximated queries of the given type which returned non-empty results

TABLE III: RESULTS OF THE EXACT QUERY, AND OF EACH OF ITS 1-STEP APPROXIMATE FORMS

Query	exact form	approx NodeLabelSet	approx Map	approx RelTypeSet	approx Interval	remove RelPattern	add RelPattern
Q1	0	-	-	90624 (5)	-	99695 (1)	0
Q2	0	68539 (1)	-	13695+ (3)	-	1032044 (1)	0
Q3	1	1285443 (3)	-	5443 (2)	-	227090 (1)	42064+ (3)
Q4	7	17 (2)	0+ (1)	-	83 (1)	350008 (1)	230 (5)
Q5	0	50 (1)	-	0	-	0	0
Q6	0	0	35 (1)	-	-	38 (1)	0
Q7	16	190 (4)	-	0	-	1791 (2)	27477 (5)
Q8	4	12 (3)	-	0	-	11 (1)	24 (6)
Q9	83	314+ (2)	0+ (1)	-	2894 (1)	350008 (1)	3947 (5)
Q10	35	111 (3)	-	-	-	74 (2)	203+ (15)

Discussion

- Table II shows that there can be a large difference in the number of results returned by the least and most selective approximated queries

  points to the potential usefulness of our proposed data-driven heuristics for ordering the results of equal-cost queries

- Table III shows that all of the different types of approximations in Algorithms 1-9 are potentially useful in that, over our 10 test queries, all of them yield approximated queries that return non-empty results
- Query timings are encouraging from a performance perspective – users would not need to wait a long time for results from most of the approximated queries to be returned

Discussion

- A qualitative discussion of the meaning of each exact query and of its 1-step approximated forms can be found in our BigData 2019 paper
- Also discussed in that paper are additional useful answers returned for each query due to the 1-step approximate evaluation
- In summary:
  - Q1, Q2, Q5, Q6 return no results in their exact form, due to not matching the graph structure, and approximation is able to correct this
  - Q3, Q4, Q8 return few results in their exact form, and approximation is able to generate additional relevant results
  - Q7, Q9, Q10 do return substantial results, but approximation is able to uncover additional relevant answers

5 Conclusions

- We have explored the benefits of supporting approximation of path patterns in Cypher queries:
  - correcting users' erroneous queries
  - finding additional relevant answers
  - generating new queries which may return unexpected results and bring new insights
- Our algorithms leverage existing Cypher query evaluation mechanisms
- A preliminary performance study shows the promise of our approach, both for returning useful answers for the user and in terms of query performance

Future work

- More extensive performance study to investigate, inter alia:
  - approximations at larger distances
  - wider variety of queries on graph datasets with varying topological characteristics
- Optimisation of approximate query evaluation e.g.
  - using knowledge of the graph structure to avoid executing approximate queries that can return no answers
  - rewriting queries into equivalent forms that execute faster, using knowledge of the system's query optimiser internals, database statistics, cost model
- Design of graphical user interface to support users' interaction with query approximation facilities
- Approximation for a wider subset of Cypher, beyond path patterns

Appendix: other ongoing research see www.dcs.bbk.ac.uk/~ap and DBLP

Fundamental research:

- Conceptual modelling approaches to data visualisation
- Graph query optimisation, indexing

Interdisciplinary research:

- Design of knowledge bases to support humanities research ("Weaving communities of practice", "Mapping Museums" projects): semantic technologies, data visualisation
- Learning analytics and awareness tools for teachers in Exploratory Learning settings; personalised learning environments