View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Birkbeck Institutional Research Online

Implementing Flexible Operators for Regular Path Queries

Petra Selmer
London Knowledge Lab
Birkbeck, University of London
lselm01@dcs.bbk.ac.uk

Alexandra Poulovassilis
London Knowledge Lab
Birkbeck, University of London
ap@dcs.bbk.ac.uk

Peter T. Wood
London Knowledge Lab
Birkbeck, University of London
ptw@dcs.bbk.ac.uk

ABSTRACT

Given the heterogeneity of complex graph data on the web, such as RDF linked data, a user wishing to query such data may lack full knowledge of its structure and irregularities. Hence, providing users with flexible querying capabilities can be beneficial. The query language we adopt comprises conjunctions of regular path queries, thus including extensions proposed for SPARQL 1.1 to allow for querying paths using regular expressions. To this language we add two operators: APPROX, supporting standard notions of approximation based on edit distance, and RELAX, which performs query relaxation based on RDFS inference rules. We describe our techniques for implementing the extended language and present a performance study undertaken on two real-world data sets. Our baseline implementation performs competitively with other automaton-based approaches, and we demonstrate empirically how various optimisations can decrease execution times of queries containing APPROX and RELAX, sometimes by orders of magnitude.

1. INTRODUCTION

The volume of graph-structured data on the web continues to grow, most recently in the form of RDF Linked Data. At the time of writing, there are 570 large datasets, spanning a variety of domains, such as the life sciences, geographical and government domains [2]. The prevalence of graph databases, such as Sparksee [21], Neo4j [14] and OrientDB [16], has also greatly increased over the past few years; they have been used in areas as diverse as social network analysis, recommendation services [20] and bioinformatics [1].

Graph-structured data in these domains may be complex, heterogeneous and evolving in terms of its structure, making it difficult for users to formulate queries that precisely match their information retrieval requirements. In this paper, we discuss the development of efficient algorithms for approximate matching and relaxation of conjunctive regular path (CRP) queries over such data, with the aim of assisting users in formulating queries and interactively retrieving results that are of relevance to them. Query results are returned incrementally to the user in order of their increasing edit or relaxation distance from the original query, with users being able to specify a limit on the number of results returned in each phase.

This paper extends earlier work in [9, 18], where the APPROX and RELAX operators were introduced, and in [17], where an initial prototype implementation was described, by describing our system implementation, called *Omega*, in detail. We also undertake here a performance study on real-world data sourced from adult further education [17] and from YAGO [10]. This study demonstrates that the performance of exact query evaluation is competitive with other automaton-based approaches, while a number of novel optimisations improve the performance of queries containing APPROX and RELAX, sometimes by orders of magnitude.

In Section 2 we give the necessary background and motivation, introducing our graph-based data model and query language. In Section 3 we discuss the implementation of *Omega*. We present our performance study in Section 4. In Section 5 we review related work in CRP query evaluation for graph-structured data. Section 6 summarises the contributions of the paper, gives our concluding remarks and directions for further work.

©2015, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2015 Joint Conference (March 27, 2015, Brussels, Belgium) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

2. BACKGROUND AND PRELIMINARIES

Omega uses a general graph-structured data model comprising a directed graph $G = (V_G, E_G, \Sigma)$ and a separate ontology $K = (V_K, E_K)$. The set V_G contains nodes each representing an entity instance or an entity class, while the set $E_G \subseteq V_G \times (\Sigma \cup \texttt{type}) \times V_G$ represents relationships between members of V_G. For an edge $e = (x, l, y) \in E_G$, l is called the *label* of e, x the *source* of e, and y the *target*. We assume that the *alphabet* Σ is finite. The label `type` is used to connect an entity instance to its class, and can represent the corresponding notion in RDF/S (see below).

The set V_K contains nodes each of which represents an entity class or a property. We call a node in V_G or V_K that represents an entity class a *class node* and a node in V_K that represents a property a *property node*. So $V_G \cap V_K$ consists of all the class nodes of V_G.

The edges in E_K capture subclass relationships between class nodes, subproperty relationships between property nodes, and domain and range relationships between property and class nodes. Hence, $E_K \subseteq V_K \times \{\texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\} \times V_K$. We assume that $\Sigma \cap \{\texttt{type}, \texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\} = \emptyset$. We also assume that the set of labels of property nodes in V_K does not contain the label `type`.

This general graph model encompasses RDF data, except that it does not allow for the representation of RDF's 'blank' nodes; however, blank nodes are discouraged for linked data [7]. Our graph model also encompasses a fragment of the RDFS vocabulary: `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, and `rdfs:range`, which we abbreviate by the symbols `type`, `sc`, `sp`, `dom`, and `range`.

The query language underlying *Omega* is that of *conjunctive regular path queries* [3]. A conjunctive regular path (CRP) query Q, consisting of n conjuncts, is of the form

$$(Z_1, \ldots, Z_m) \leftarrow (X_1, R_1, Y_1), \ldots, (X_n, R_n, Y_n)$$

where $m, n \geq 1$, each X_i and Y_i is a variable or a constant, each Z_i is a variable appearing in the right-hand-side of Q, and each R_i is a *regular expression* over the alphabet from which edge labels in the graph are drawn. In our context, a regular expression R is defined as follows:

$$R := \epsilon \mid a \mid a- \mid _ \mid (R1 \cdot R2) \mid (R1|R2) \mid R^* \mid R^+$$

where ϵ is the empty string, a is any label in $\Sigma \cup \{\texttt{type}\}$, $a-$ represents traversal of an edge in the reverse direction, "_" denotes the disjunction of all constants in $\Sigma \cup \{\texttt{type}\}$, and the operators have their usual meaning.

The (exact) *answer* to a CRP query Q on a graph G can be obtained in a standard way by finding all pairs of nodes in G satisfying each conjunct, joining the results, and projecting over the variables in the head of Q.

Example 1. *Suppose a user wishes to find people who graduated from an institution located in the UK and poses the following query, Q, over the YAGO graph [10]:*

```
(?X) <- (UK,isLocatedIn-.gradFrom,?X)
```

(Variables in a query have an initial question mark.) This query returns no results since it requires that there is some entity y, located in the UK, which has graduated from some institution. However, no such y exists, since only people can graduate from an institution and only events and places can be located in a country.

The work in [9] investigated *approximate* matching of CRP queries, allowing edit operations such as insertions, deletions and substitutions of edge labels to be applied to the regular expressions R_i of a CRP query, each with some edit cost configurable by the user.

Example 2. *In* Omega*, the user can submit a variant of Q in which the conjunct can be approximated:*

```
(?X) <- APPROX (UK,isLocatedIn-.gradFrom,?X)
```

`isLocatedIn−.gradFrom` *is approximated by* `isLocatedIn−.gradFrom−`*, at some edit distance α, by substituting* `gradFrom` *with* `gradFrom−`*. This query now returns results, matching the user's original intention by correcting the error in Q.*

The work in [18] also considered applying ontology-based *relaxation* to the regular expressions R_i. This allows query relaxations entailed using information from the ontology K, in particular: (i) replacing a class/property label by that of an immediate superclass/superproperty, at some cost β; (ii) replacing a property label by a `type` edge with target the property's domain or range class, at some cost γ.

Example 3. *In* Omega*, the user can submit a variant of Q in which the conjunct can be relaxed:*

```
(?X) <- RELAX (UK,isLocatedIn-.gradFrom,?X)
```

This query allows `gradFrom` *to be relaxed to its parent property* `relationLocatedByObject` *at cost β, which now allows properties such as* `happenedIn` *and* `participatedIn` *to be matched, and answers to be returned at 'distance' β.*

3. IMPLEMENTATION OF *OMEGA*

Figure 1 illustrates the architecture of the *Omega* system. Sparksee [21] (formerly DEX) is used as the data store. The development was undertaken using the Microsoft .NET framework. The system comprises four components: (i) the *console* layer, in which queries are submitted, and which displays the results; (ii) the *system* layer in which query plans are constructed and executed; (iii) the Sparksee API; and (iv) the data store itself.

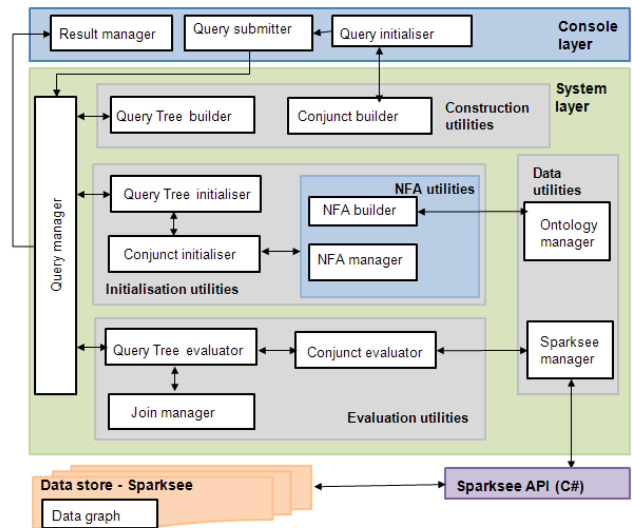

Figure 1: System architecture

The architecture of the *system* layer broadly follows that described in [17], with the major change being that the data store used in *Omega* is Sparksee [21] rather than XML. This layer is responsible for the construction of the automaton (NFA) corresponding to each query conjunct. Given a query conjunct (X, R, Y), a weighted NFA M_R is constructed to recognise the language denoted by the regular expression R. If the conjunct is prefixed by APPROX or RELAX, then M_R is augmented to produce an automaton A_R or M_R^K, respectively; we discuss this in Section 3.3. Further responsibilities of the *system* layer include the construction of the query tree, the incremental construction of a weighted product automaton H_R from each conjunct's automaton and the data graph G, and the evaluation of the overall query, including performing a ranked join for multi-conjunct queries. We make extensive use of data structures provided by the **C5 Generic Collection** library [15].

3.1 The Sparksee Data Model and API

The two main Sparksee structures used in our implementation are nodes and edges (which may be directed or undirected), each of which has a pre-created *type* (this is a label,

of *string* data type), and a unique object identifier (*oid*) of *long* data type. Associated with each node and edge are zero or more attributes, which are key-value pairs; values may be of any primitive data type. Further details regarding Sparksee may be found in [12, 13] and the User Manual[1]. The main Sparksee API functions used in *Omega* are as follows:

`Neighbors` takes as arguments a node n and edge type t, and returns the set of nodes connected to n via an edge of type t; the directionality of edges may also be specified.

`Heads` takes a set of edges E, and returns the set of nodes which are the target of an edge in E. `Tails` is analogous to `Heads`, except that nodes which are sources are returned. `TailsAndHeads` returns the union of `Heads` and `Tails`.

To store the data, Sparksee uses a combination of maps (inverted indexes) and associated bitmap vectors [13]. To improve the performance of the `Neighbors` function, an option may be set to index the neighbouring nodes when creating an edge type t. This means that an index entry is created when an edge of type t is created between any two nodes. Node- and edge-related attributes may also be configured to be indexed when they are created (the index stores all *oids* associated with each value of the attribute).

3.2 Omega data graphs

As it is mandatory in Sparksee for each node to have a type, and as our data model does not assume that nodes are typed, we create all of our nodes to be of the same type, 'node'. All of our nodes have one attribute, of *string* data type, representing the node label (which is unique in the data graph G). This attribute has indexing enabled.

We create multiple edge types, all of which are defined to be *directed* edge types with indexing enabled. Specifically, for each edge in G having label $l \in \Sigma$, two Sparksee edges are created: (i) one having type l, and (ii) one having type 'edge' with an associated indexed *string*-valued attribute corresponding to l. We introduce the generic 'edge' type to counter a limitation of the `Neighbors` function which requires the type of the edge to be provided as an argument, in order to allow us easily to retrieve multiple types of edges simultaneously. For each edge in G labelled `type`, only one edge is created, whose type is `type`. In cases which require the retrieval of *all* types of edges of a node, we retrieve all 'edge' edges, followed by all `type` edges.

3.3 Query conjunct initialisation

The initialisation of a query conjunct (X, R, Y) comprises the construction of the associated automaton (one of M_R, A_R or M_R^K), and the initialisation of its data structures prior to the evaluation of the conjunct. We discuss each here.

In all cases, an automaton (NFA) M_R is first constructed from regular expression R using standard techniques. Then, if the conjunct is prefixed by APPROX or RELAX in the query, additional transitions and states are added (see [18]), along with the removal of ϵ-transitions, to form A_R or M_R^K respectively. As the automaton is weighted, the removal of ϵ-transitions may result in final states having an additional, positive weight [5]. For state s, we denote this weight by $weight(s)$. The NFA is represented as a set of transitions (s, a, c, t), where s is the 'from' state, t is the 'to' state, a is the label, and c is the cost.

If the conjunct is APPROXed, the *insertion* edit operation would result in many additional transitions in the NFA, one for each label in $\Sigma \cup \{\texttt{type}\}$ and their reversals. To make our automaton more compact, we represent these as a single transition labelled by the wildcard label $*$.

In all cases, if X (respectively, Y) is a constant c, we annotate the initial (resp. final) state with c; otherwise we annotate the initial (resp. final) state with the wildcard symbol matching any constant.

The pseudocode for the initialisation of a conjunct is given in the Open procedure below. After constructing the appropriate automaton, the procedure evaluates the conjunct by traversing the automaton and the data graph simultaneously. This traversal is represented by tuples of the form (v, n, s, d, f), where d is the distance associated with visiting node n in state s having started from node v, and f denotes whether the tuple is 'final' or 'non-final' (see below).

The tuples are added to and removed from a dictionary D_R whose key is an integer-boolean variable (where the integer portion represents a distance and the boolean portion represents the final or non-final tuples at that distance). The value associated with each key is a linked list of tuples. Tuples are always added to, and removed from, the head of a linked list (at cost $O(1)$). We introduced the notion of final and non-final tuples in order to prioritise the removal of 'final' tuples (rather than 'non-final' ones) at the minimum distance (if any), so that answers may be returned earlier. Including this refinement improved the performance of most of our queries, and also ensured that some queries, which had previously failed by running out of memory, completed.

```
Procedure Open
     Input: query conjunct (X, R, Y)
(1)  construct NFA M_R for R; initial state is s_0
(2)  transform M_R into A_R (APPROX) or M_R^K (RELAX) if
     necessary
(3)  visited_R ← ∅
(4)  d ← 0
(5)  if conjunct is of the form (C, R, ?X) then
(6)      //Let n be the node in G corresponding to C
(7)      if RELAX is being applied then
(8)          foreach node m ∈ GetAncestors(n) do
(9)              add(D_R, (m, m, s_0, d, false))
(10)     else
(11)         add(D_R, (n, n, s_0, d, false))
(12) else
(13)     //the conjunct is of the form (?X, R, ?Y)
(14)     if s_0 is final then
(15)         if weight(s_0) = 0 then
(16)             foreach node n in G do
(17)                 add(D_R, (n, n, s_0, d, true))
(18)         else
(19)             foreach n ∈ GetAllNodesByLabel(s_0) do
(20)                 add(D_R, (n, n, s_0, d, false))
(21)     else
(22)         foreach n ∈ GetAllStartNodesByLabel(s_0) do
(23)             add(D_R, (n, n, s_0, d, false))
```

We distinguish between 3 cases in the Open procedure: (*Case 1*) If the conjunct is of the form $(C, R, ?Y)$ where C

[1] www.sparsity-technologies.com/downloads/UserManual.pdf

is a constant, we begin the traversal at the node in G having the attribute value C.

(*Case 2*) A conjunct of the form $(?X, R, C)$ is transformed to $(C, R^-, ?X)$, where R^- is the *reversal* of R. This reversal can be accomplished in linear time starting from the NFA for R [23]. Thus, *Case 2* reverts to *Case 1*.

If the conjunct has the RELAX operator and C is a class node, we also add to D_R every node returned by $GetAncestors$ (line 8). This function returns all superclasses of C in order of increasing specificity so that they are added to the list in D_R in that order. We want to process more specific classes first, given that nodes representing more general classes will have larger degree (owing to transitive closure) and will lead to answers of greater cost.

(*Case 3*) For a conjunct of the form $(?X, R, ?Y)$, lines 14 to 23 are invoked. The function $GetAllNodesByLabel$ (line 19) takes as input a list of all labels on transitions whose 'from' state is the initial state s_0. Each label in the list is then processed as follows: (i) the directionality of the label is determined — i.e. whether it is an incoming or an outgoing edge, or whether both incoming *and* outgoing edges are required (as for the *-labelled transitions, introduced above); (ii) the set of object identifiers (oids) for the nodes having the relevant edge and directionality are retrieved using the Sparksee methods `Heads`, `Tails` and `TailsAndHeads`; (iii) Sparksee set operations are used to maintain a distinct set of nodes, so that the same node is not re-added to D_R at a higher cost (this can occur with the '*' label); and (iv) the remaining nodes in the graph G are returned. When adding to D_R, we iterate through the set of nodes in order of decreasing cost. The function $GetAllStartNodesByLabel$ (line 22) is identical to $GetAllNodesByLabel$, except that it does not include step (iv).

We have implemented the above two functions and that retrieving all nodes in G (line 16) as coroutines in conjunction with the $GetNext$ procedure (discussed in Section 3.4), incrementally obtaining nodes in batches (the default is 100 nodes at a time). We found that, as a result, the execution time of some queries was reduced by half, since nodes not required to answer the user's query are not added to D_R.

3.4 Query conjunct evaluation

The two algorithms concerned with the evaluation of a single query conjunct are *GetNext* and *Succ*, which have previously been presented in [9, 18]. We now describe our physical implementation of these algorithms.

GetNext returns the next query answer, in order of non-decreasing distance from the original query Q, by repeatedly removing the first tuple (v, n, s, d, f) from the distance d list of D_R until D_R is empty. If the removed tuple is final (f is *true*) and the answer (v, n, d') has not been been generated before for some d', the triple (v, n, d) is returned after being added to $\texttt{answers}_R$. If the tuple is not final, we add (v, n, s) to $\texttt{visited}_R$, and add $(v, m, s', d + d', false)$ to D_R for each transition $\xrightarrow{d'} (s', m)$ returned by $Succ(s, n)$ such that $(v, m, s') \notin \texttt{visited}_R$. If s is a final state, its annotation matches n, and the answer (v, n, d') has not been been generated before for some d', then we add the weight of s to d, mark the tuple as final, and add the tuple to D_R.

For $visited_R$, we use a hashed set which has $O(1)$ lookup time. Lines 8 and 9 in practice are executed as a single step, and the logic in lines 10 to 13 is only executed if the item was added. This means that we never re-process a previously-processed (v, n, s) triple; this situation may arise when (v, n, s) triples of monotonically-increasing distances are created and added at lines 11 and 13 (we therefore never process 'duplicate' tuples at a higher distance).

Procedure GetNext(X, R, Y)

Input: query conjunct (X, R, Y)
Output: triple (v, n, d), where v and n are instantiations of X and Y

```
(1)  while nonempty(D_R) do
(2)      (v, n, s, d, final) ← remove(D_R)
(3)      if final then
(4)          if ∄d'.(v, n, d') ∈ answers_R then
(5)              append (v, n, d) to answers_R
(6)              return (v, n, d)
(7)      else
(8)          if (v, n, s) ∉ visited_R then
(9)              add (v, n, s) to visited_R
(10)             foreach -d'-> (s', m) ∈ Succ(s, n) s.t.
                 (v, m, s') ∉ visited_R do
(11)                 add(D_R, (v, m, s', d + d', false))
(12)             if s is a final state and its annotation
                 matches n and ∄d'.(v, n, d') ∈ answers_R
                 then
(13)                 add(D_R, (v, n, s, d + weight(s), true))
(14)     //Incrementally add the next batch of initial nodes
(15)     if no distance 0 tuples in D_R and more initial nodes
         available then
(16)         foreach initial node n' do
(17)             add(D_R, (n', n', s_0, 0, false));
(18) return null
```

In lines 15 to 17, we utilise a coroutine for $(?X, R, ?Y)$ conjuncts. If D_R no longer contains any tuples at distance 0, we retrieve and add the next batch of initial nodes from the functions initially invoked in the *Open* procedure.

The *Succ* function takes as input a node (s, n) of the weighted product automaton H_R and returns a set of of transitions $\xrightarrow{d} (p, m)$, such that there is an edge in H_R from (s, n) to (p, m) with cost d. The function $NextStates(s)$ returns the set of states reachable from state s on reading some label, along with the associated costs. We only retrieve those edges for node n in G whose label corresponds to one of those returned by $NextStates(s)$, thereby using the transitions in the automaton to guide the selection of neighbouring nodes in G.

$NeighboursByEdge$ takes as input the *oid* of a node n from G and a *label*, and returns a list of neighbouring node *oids*. If the label is not '*', we invoke the Sparksee method `Neighbors` in order to retrieve all neighbouring nodes for n connected by an edge labelled with *label*, taking directionality into account. If the label is '*', we invoke `Neighbors` once for edges labelled 'edge' and once for edges labelled `type`. We do this for both directions in both cases.

In each case, we iterate over the neighbouring nodes, adding their *oid* to W (lines 7 and 8). As it is possible for $NextStates$ to return identical labels consecutively (at the same cost), we store the results of $NeighboursByEdge$ for new labels in a

Procedure Succ(s, n)

Input: state s of NFA and node n of G
Output: set of transitions from (s, n) in H_R

(1) $W \leftarrow \emptyset$; $U \leftarrow \emptyset$
(2) $currlabel \leftarrow$ null; $prevlabel \leftarrow$ null
(3) **foreach** $(label, successor, cost) \in NextStates(s)$ **do**
(4) $\quad currlabel \leftarrow label$
(5) $\quad$ **if** $currlabel \neq prevlabel$ **then**
(6) $\quad\quad U \leftarrow NeighboursByEdge(n, label)$
(7) $\quad$ **foreach** $node\ m \in U$ **do**
(8) $\quad\quad$ add the transition $\stackrel{cost}{\rightarrow} (successor, m)$ to W
(9) $\quad prevlabel \leftarrow currlabel$
(10) **return** W

Class hierarchy	Depth	Average fan-out
Episode	2	2.67
Subject	2	8
Occupation	4	4.08
Education Qualification Level	2	3.89
Industry Sector	1	21

Figure 2: Characteristics of the class hierarchies.

set U (line 6), avoiding identical calls to $NeighboursByEdge$.

4. PERFORMANCE STUDY

In this section, we present performance results from two case studies. We also discuss two optimisations, showing how each results in improved performance for some of the APPROX/RELAX queries. All experiments were run on an Intel Core i7-950 (3.07-3.65GHz) with 6GB memory, running Windows 7 (64 bit).

4.1 L4All Case Study

Our first case study uses data from the L4All project [17]. Briefly, the L4All system aimed to support lifelong learners in exploring learning opportunities and in planning and reflecting on their learning. The system allows users to create and maintain a chronological record — a timeline — of their learning and work episodes. Each episode is (i) linked to an Episode category by an edge labelled `type`, (ii) linked to other episodes edges labelled 'next' or 'prereq' (indicating whether the earlier episode simply preceded, or was necessary in order to be able to proceed to, the later episode), and (iii) linked to either an occupational or an educational event, by means of an edge labelled 'job' or 'qualif', which in turn is classified in terms of Education Qualification Level or Industry Sector, respectively.

Figure 2 shows the class hierarchies used in the ontology accompanying the data; the *depth* is the length of the longest path from the root to the leaf nodes, and the *average fan-out* is the average number of children of each non-leaf class. There is only one property hierarchy: the super-property 'isEpisodeLink' has 'next' and 'prereq' as subproperties. These properties also have defined domains and ranges, but as these are not used in the our performance study, we do not discuss them further.

Our initial data comprised five detailed timelines from real users, to which we added 16 additional realistic timelines. Each of these timelines consisted of a mixture of educational and occupational episodes, and varied in terms of the number of episodes contained within them, as well as the classification of each episode.

	L1	L2	L3	L4
Nodes	2,691	15,188	68,544	240,519
Edges	19,856	118,088	558,972	1,861,959

Figure 3: Characteristics of the L4All data graphs.

Q1	`(Work Episode, type⁻, ?X)`
Q2	`(Information Systems, type⁻.qualif⁻, ?X)`
Q3	`(Software Professionals, type⁻.job⁻, ?X)`
Q4	`(?X, job.type, ?Y)`
Q5	`(?X, next+, ?Y)`
Q6	`(?X, prereq+, ?Y)`
Q7	`(?X ,next+|(prereq+.next), ?Y)`
Q8	`(Mathematical and Computer Sciences, type.prereq+, ?X)`
Q9	`(Alumni 4 Episode 1_1, prereq*.next+.prereq, ?X)`
Q10	`(Librarians, type⁻, ?X)`
Q11	`(Librarians, type⁻.job⁻.next, ?X)`
Q12	`(BTEC Introductory Diploma, level⁻.qualif⁻.prereq, ?X)`

Figure 4: The L4All query set.

We then scaled this data graph up by creating synthetic versions of the real timelines in order to obtain four data graphs of increasing size, called **L1** (143), **L2** (1,201), **L3** (5,221) and **L4** (11,416), where the number in brackets refers to the number of timelines. Figure 3 shows the characteristics of each data graph. The synthetic timelines were generated by duplicating a real timeline and using the ontology to alter the classification of each episode to be a 'sibling' class of its original class, for as many sibling classes as are present. Each duplicated timeline remained identical to the original in terms of the number of episodes, whether the type of the episode was educational or occupational, and the manner in which episodes were linked to each other. Thus, as the data graph increases in size, the degree of the class nodes (i.e. the nodes with incoming `type` edges) increases linearly. As the data graph size increases, the total number of edges also increases linearly with the number of nodes.

We execute a series of single-conjunct queries on this data in order to evaluate the performance of our APPROX and RELAX operators, shown in Figure 4. These 12 queries include actual queries used in the original L4All case study, as well as others designed to stress test our implementation. Each query is first run in 'exact' mode — i.e. neither APPROX nor RELAX is used — followed by versions of the same query containing either the APPROX or the RELAX operator. We therefore run 36 queries in total.

We used a cost of 1 for each approximation operation (insertion, substitution and deletion). For RELAX, we applied rules of type (i) (see Section 2), also at a cost of 1. We ran each query five times, discarding the first run as the cache-warm-up. After initialisation, each exact query was run to completion, in which *all* results are obtained. On the other hand, each APPROX and RELAX run comprises the following sequence: initialisation; obtain results 1–10 ('batch 1'); obtain results 11–20 ('batch 2'); ...; obtain results 91–100 (batch 10). For exact queries, the average time to return all answers was taken across runs 2 to 5. For APPROX and RELAX queries, we took the average of each of the 10 batches across runs 2 to 5 to obtain an average for each batch. We

	Q3	Q8	Q9	Q10	Q11	Q12
L1: Exact	58	0	1	1	2	0
L1: APPROX	100 **1** *(42)*	100 **2** *(100)*	100 **1** *(32)* **2** *(67)*	100 **1** *(7)* **2** *(92)*	100 **1** *(12)* **2** *(86)*	100 **1** *(100)*
L1: RELAX	100 **1** *(42)*	0	12 **1** *(11)*	100 **1** *(20)* **2** *(20)* **3** *(59)*	100 **1** *(40)* **2** *(40)* **3** *(18)*	59 **1** *(59)*
L2: Exact	1,090	0	1	1	2	0
L2: APPROX	100	100 **2** *(100)*	100 **1** *(32)* **2** *(67)*	100 **1** *(7)* **2** *(92)*	100 **1** *(12)* **2** *(86)*	100 **1** *(100)*
L2: RELAX	100	0	12 **1** *(11)*	100 **1** *(20)* **2** *(20)* **3** *(59)*	100 **1** *(40)* **2** *(40)* **3** *(18)*	59 **1** *(59)*
L3: Exact	3,104	0	1	1,024	2,048	0
L3: APPROX	100	100 **2** *(100)*	100 **1** *(32)* **2** *(67)*	100	100	100 **1** *(100)*
L3: RELAX	100	0	12 **1** *(11)*	100	100	59 **1** *(59)*
L4: Exact	3,104	0	1	1,024	2,048	0
L4: APPROX	100	100 **2** *(100)*	100 **1** *(32)* **2** *(67)*	100	100	100 **1** *(100)*
L4: RELAX	100	0	12 **1** *(11)*	100	100	100 **1** *(100)*

Figure 5: Results for each query and data graph.

then computed the average over all batches. Some of these queries yielded fewer than 100 results.

We show the number of results obtained for queries 3, 8, 9, 10, 11 and 12 per data graph in Figure 5. Queries 1 and 2 showed similar performance to Query 3, while queries 4–7 all returned well over 100 exact results on all the data graphs, thus negating the need to apply APPROX and RELAX to them for the purposes of this performance study. For APPROX and RELAX queries yielding non-exact answers, we also show in Figure 5 the distances of the non-exact answers, as well as the number of the answers at each non-zero distance in brackets (with the number of exact answers comprising the difference). For example, query Q9/APPROX on data graph **L2** returns 1 exact answer (100-(32+67)), 32 answers at distance 1 and 67 answers at distance 2.

Figures 6, 7 and 8 show the average execution times for the exact, APPROX and RELAX versions, respectively, of queries 3, 8, 9, 10, 11 and 12 over the data graphs **L1**–**L4**.

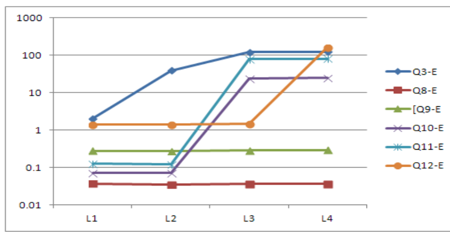

Figure 6: Execution time (ms) – exact queries.

For the exact queries, we see that queries 8 and 9 take constant time for all the data graphs since at most a single answer is returned. The jump in execution time from **L2** to **L3** for queries 10 and 11 is caused by the large increase in the number of answers; similarly for query 3. Query 12 shows a steep increase owing to the manner in which the synthetic timelines were generated, giving rise to the processing of nodes of ever-increasing degree. We note that the performance of all the queries is competitive with the expected behaviour of native NFA-based approaches to regular path query evaluation [11].

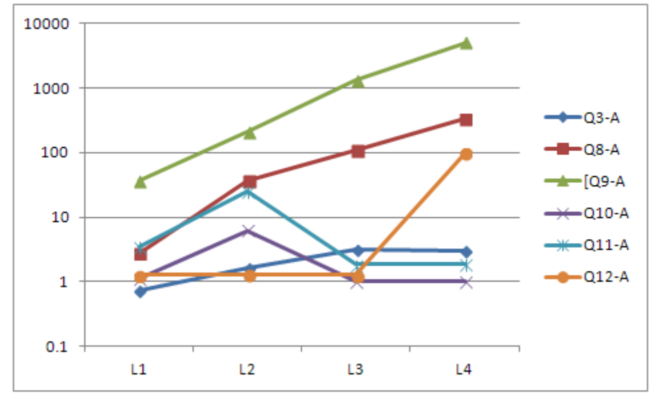

Figure 7: Execution time (ms) – APPROX queries.

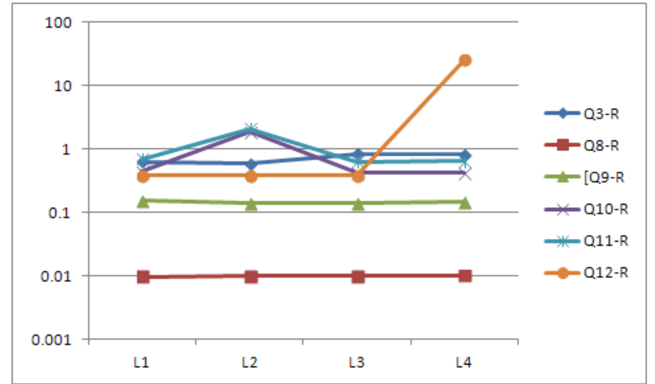

Figure 8: Execution time (ms) – RELAX queries.

For the APPROX queries, queries 10 and 11 show a decrease in the time taken for **L3** and **L4** compared with **L2** which is caused by the fast processing of sufficient exact results for the larger two data graphs; similarly for query 3. However, the APPROX versions of queries 8, 9 and 12 exhibit an exponential increase in time taken to retrieve the top 100 results. This is caused by a large number of intermediate results being generated (due to the *Succ* function returning a large number of transitions which are then converted into tuples in $GetNext$ and added to D_R). We discuss optimisation of query 9 in Section 4.3. Regarding queries 8 and 12, the time for query 8 decreased from 332ms to 272ms by applying the first optimisation of Section 4.3. Query 12 was not aided by the optimisations of Section 4.3.

The RELAX queries 3, 8, 9, 10 and 11 all exhibit a fairly constant execution time across the data graphs. Query 12 shows an increase from **L3** to **L4** for a reason similar to that for its APPROX version.

4.2 YAGO Case Study

For our second case study, we used data from YAGO [10]. The connectivity patterns in YAGO differ from the rather 'linear' timelines comprising the L4All data, so provide a contrasting basis on which to evaluate query performance. Additionally, the YAGO ontology differs from the L4All one in terms of its breadth and depth.

We downloaded the simpler taxonomy and core data facts from the YAGO website (the SIMPLETAX and CORE por-

Q1	(Halle_Saxony-Anhalt, bornIn$^-$.marriedTo.hasChild, ?X)
Q2	(Li_Peng, hasChild.gradFrom.gradFrom$^-$.hasWonPrize, ?X)
Q3	(wordnet_ziggurat, type$^-$.locatedIn$^-$, ?X)
Q4	(?X, directed.married.married+.playsFor, ?Y)
Q5	(?X, isConnectedTo.wasBornIn, ?Y)
Q6	(?X, imports.exports$^-$, ?Y)
Q7	(wordnet_city, type$^-$.happenedIn$^-$.participatedIn$^-$, ?X)
Q8	(Annie Haslam, type.type$^-$.actedIn, ?X)
Q9	(UK, (livesIn$^-$.hasCurrency)|(locatedIn$^-$.gradFrom), ?X)

Figure 9: The YAGO query set.

	Q2	Q3	Q4	Q5	Q9
Exact	2	0	0	0	0
APPROX	100 **1** *(98)*	100 **1** *(5)* **2** *(95)*	?	?	100 **1** *(100)*
RELAX	2	100 **1** *(100)*	0	100 **1** *(100)*	100 **1** *(100)*

Figure 10: Query results for the YAGO data graph.

tions) and imported these into our system[2]. The resulting data graph consists of 3,110,056 nodes and 17,043,938 edges. There is only one classification hierarchy in YAGO; its *depth* is 2 and *average fan-out* is 933.43.

Including the `type` property, YAGO uses 38 properties. There are two property hierarchies, containing 2 and 6 sub-properties respectively. The properties also have domains and ranges defined, not used in our performance study.

The queries we ran on the YAGO data are listed in Figure 9. The exact, APPROX and RELAX versions therefore give rise to 27 queries, for which we calculated the timings as described in Section 4.1, with the edit and relaxation costs the same as those used for the L4All case study.

The number of results obtained for queries 2, 3, 4, 5 and 9 for the YAGO data graph are shown in Figure 10. For each query, the exact version was run to completion, and the APPROX and RELAX versions were run until the top 100 answers were retrieved. The '?' indicates instances where the system ran out of memory and hence failed without returning any answers. Query 1 showed a similar performance to query 2; query 6 is similar to queries 4 and 5 in terms of query structure, but it terminated, unlike these; and queries 7 and 8 returned well over 100 exact results, therefore negating the need for APPROX and RELAX.

Figure 11 shows the average execution times for queries 2, 3, 4, 5 and 9. For the exact queries, queries 2 and 3 execute quickly. Queries 4 and 5 take longer to execute because their conjuncts are of the form $(?X, R, ?Y)$. Hence processing is initiated from a large number of nodes (41,811 and 33,834 respectively), and further traversal leads to large numbers of intermediate results; query 9 behaves similarly.

APPROX queries 2 and 3 exhibit poor performance due to a large number of intermediate results, while query 9 takes the same time as the exact version; we discuss these further in Section 4.3. Queries 4 and 5 failed to terminate as the system ran out of memory; this, too, is due to a large number of intermediate results.

RELAX queries 2, 3 and 9 performed competitively, returning more results for the latter two than their exact counterparts. Query 4's time was the same as for the exact version (with no extra results). Query 5 returned results and executed faster than the exact version; this is due to 100 results being found (by the application of rules of type i) and execution terminating sooner.

[2]http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/downloads/

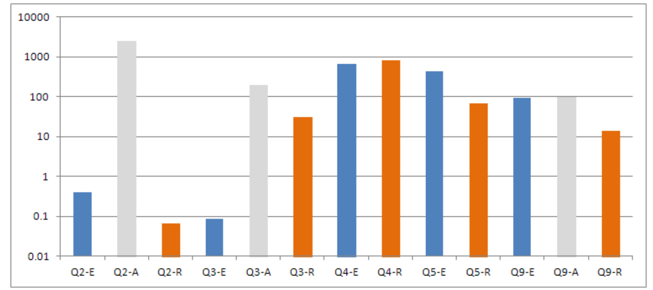

Figure 11: Execution times (ms), YAGO data graph.

4.3 Query execution optimisations

In this section we outline two optimisations which may improve the performance of APPROX and RELAX queries.

Retrieving answers by distance: We have implemented a distance-aware mode of query execution for APPROX and RELAX queries in order to prevent the unnecessary processing of data which yields answers at a cost higher than that required by the user. For example, if the user requests the top 100 answers in cases where there are over 100 answers at cost 0, using transitions of greater cost to traverse G and add tuples to D_R results in a slower query execution time overall, owing to the processing of redundant data.

We set a *current maximum cost*, ψ, to be 0 initially. No tuple having a cost greater than zero is processed (i.e. added or removed from D_R), and all answers of cost 0 are returned. Should more answers be required, we then increment ψ by the smallest cost, ϕ, of the edit or relaxation operations being applied. For each successive value of ψ, query evaluation commences from the beginning, as all tuples having a cost $c \leq \psi$ need to be considered (so this method is not suitable in cases where answers at high cost are required) but no tuple having a cost greater than ψ is processed.

Using distance-aware retrieval substantially improves the performance of some APPROX queries. For example, L4All queries 3 and 9 run three to four times faster with this optimisation. YAGO query 3 executes twice as fast, while query 2 takes 0.6ms instead of 2560ms, a dramatic improvement.

Replacing alternation by disjunction: Another optimisation for APPROX queries we have implemented is to decompose the NFA for a regular expression $R = R_1|R_2|...$ into sub-automata NFA$_i$ for each R_i, providing the NFA has a single start state. These are processed in default order (NFA$_1$, NFA$_2$, ...) for the distance 0 answers, and we store the number of answers returned in each case, $n_{0,i}$. To compute the answers at distance ϕ, we evaluate the sub-automata by increasing $n_{0,i}$ value. In general, to compute the answers at distance $k\phi$, we evaluate the sub-automata by increasing $n_{(k-1)\phi,i}$ value.

For example, applying this to YAGO query 9, results in the sub-automata NFA$_1$ for (UK, livesIn$^-$.hasCurrency, ?X) and NFA$_2$ for (UK, locatedIn$^-$.gradFrom, ?X). NFA$_1$ the returns the least answers at distance 0, so this is processed first for the distance 1 answers. This reduces the query execution time to 12.65ms compared with 101.23ms.

5. RELATED WORK

In this section we briefly review previous work on the implementation of regular path queries.

[11] presents a technique for the evaluation of exact queries which takes advantage of *rare* labels in a graph. A query containing one or more rare labels is broken down into a set of sub-queries such that each sub-query begins or ends with a rare label. Their method, using a bi-directional search utilising graph indexes, is shown to be faster than other automaton-based implementations. Our exact queries perform favourably compared with the results in [11].

[22] presents RPL, a regular path language for RDF data, whose implementation, like ours, uses an automaton-based approach. However, RPL is only able to process very small graphs efficiently [11].

[4] describes a framework allowing weighted RDF data to be queried in a cost-aware manner, and returning results ranked according to cost. This is accomplished by an extension to SPARQL, SPARankQL, encompassing the provision of novel predicates for expressing flexible paths between nodes and the capacity to define ranked queries (in which the weights are used). Our work allows the path to be expressed by a regular expression which may be mutated by edit operations, whereas SPARankQL can only be used to express either no restrictions on paths from a node or restrictions on specified labels of the path. The data graphs used in their performance study have, respectively, 9K nodes (24K edges) and 10K nodes (25K edges), and are both smaller and more sparse than our **L2** graph.

[6] discusses a SPARQL query graph model using transformation rules to rewrite queries. Experiments are run on RDF graphs of increasing size, with the largest comprising 1,272K triples. The rewritten queries run approximately twice as fast as the original ones. We do not yet make use of query rewriting, which is an area of future work.

6. CONCLUSIONS

Building on previous work on combining approximation and relaxation for regular path queries [18, 17], we have provided a detailed description of our implementation, *Omega*, focussing on low-level data structures and physical optimisations, both in terms of the interaction with the graph store, *Sparksee*, and our query processing layer within *Omega*.

We have presented a comprehensive performance study, using large graphs consisting of real-world data, in which we show that our baseline implementation performs competitively in terms of exact regular path queries. The benefits of our APPROX and RELAX operators have been shown in terms of additional answers being returned for queries returning few or no answers for the exact version. Many of the APPROX and RELAX queries executed quickly, but some either failed to terminate or did not complete within a reasonable amount of time. We discussed the reasons for this in each case, and showed how further optimisations, such as retrieval by distance and replacing alternation by disjunction, enabled several queries to execute faster.

For future work, we will consider the use of disk-based data structures to guarantee the termination of APPROX queries with large intermediate results (such as YAGO queries 4 and 5). We will also investigate using characteristics of the data graph and heuristics to reduce the amount of unnecessary processing. Other promising directions are query rewriting, and leveraging rare labels as in [11]. Distributed approaches [8, 19] are also relevant for flexible querying of larger-scale graphs than we have considered in our centralised approach so far.

7. REFERENCES

[1] Bio4j. http://bio4j.com/.

[2] C. Bizer, A. Jentzsch, and R. Cyganiak. http://lod-cloud.net/state/.

[3] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, pages 176–185, 2000.

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

[5] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer, 2009.

[6] O. Hartig and R. Heese. The SPARQL query graph model for query optimization. In *Proc. ESWC*, pages 564–578, 2007.

[7] T. Heath, M. Hausenblas, C. Bizer, and R. Cyganiak. How to publish linked data on the web (tutorial). In *Proc. ISWC*, 2008.

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

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

[10] G. Kasneci, M. Ramanath, F. Suchanek, and G. Weikum. The YAGO-NAGA approach to knowledge discovery. *SIGMOD Rec.*, 37(4):41–47, Mar. 2009.

[11] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Proc. SSDBM*, pages 177–194, 2012.

[12] N. Martinez-Bazan and D. Dominguez-Sal. Using semijoin programs to solve traversal queries in graph databases. In *Proc. GRADES*, pages 6:1–6:6, 2014.

[13] N. Martínez-Bazan et al. Efficient graph management based on bitmap indices. In *Proc. IDEAS*, pages 110–119, 2012.

[14] Neo4j. http://www.neo4j.com/.

[15] I. U. of Copenhagen. http://www.itu.dk/research/c5/.

[16] OrientDB. http://www.orientdb.org/.

[17] A. Poulovassilis, P. Selmer, and P. T. Wood. Flexible querying of lifelong learner metadata. *IEEE Trans. on Learning Technologies*, 5(2):117–129, 2012.

[18] A. Poulovassilis and P. T. Wood. Combining approximation and relaxation in semantic web path queries. In *Proc. ISWC*, pages 631–646, 2010.

[19] M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, and G. Lausen. RDFPath: Path query processing on large RDF graphs with MapReduce. In *ESWC Workshops*, 2011.

[20] Reco4j. http://www.reco4j.org/.

[21] Sparksee. http://www.sparsity-technologies.com/.

[22] H. Zauner, B. Linse, T. Furche, and F. Bry. A RPL through RDF: expressive navigation in RDF graphs. In *Proc. RR*, pages 251–257, 2010.

[23] D. D. Zhu and K. I. Ko. *Problem Solving in Automata, Languages, and Complexity*. Wiley, Newark, NJ, 2004.