SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases

Kemafor Anyanwu
LSDIS Lab
University of Georgia
Athens, GA 30602
anyanwu@cs.uga.edu

Angela Maduko
LSDIS Lab
University of Georgia
Athens, GA 30602
maduko@cs.uga.edu

Amit Sheth
Kno.e.sis Center
Wright State University
Dayton, OH 45435
amit.sheth@wright.edu

ABSTRACT

Many applications in analytical domains often have the need to "connect the dots" i.e., query *about* the structure of data. In bioinformatics for example, it is typical to want to query about interactions between proteins. The aim of such queries is to "extract" relationships between entities i.e. paths from a data graph. Often, such queries will specify certain constraints that qualifying results must satisfy e.g. paths involving a set of mandatory nodes. Unfortunately, most present day Semantic Web query languages including the current draft of the anticipated recommendation SPARQL, lack the ability to express queries about arbitrary path structures in data. In addition, many systems that support some limited form of path queries rely on main memory graph algorithms limiting their applicability to very large scale graphs.

In this paper, we present an approach for supporting Path Extraction queries. Our proposal comprises (i) a query language SPARQ2L which extends SPARQL with path variables and path variable constraint expressions, and (ii) a novel query evaluation framework based on efficient algebraic techniques for solving path problems which allows for path queries to be efficiently evaluated on disk resident RDF graphs. The effectiveness of our proposal is demonstrated by a performance evaluation of our approach on both real world and synthetic datasets.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Algorithms, Languages.

Keywords

SPARQL Extensions, RDF, Querying Semantic Web Databases.

1. INTRODUCTION

As more and more organizations contemplate the value of adopting Semantic Web technologies, it is likely that one of the deciding factors will be the degree to which the needs of their applications are supported. Therefore, it is important for Semantic Web storage and querying facilities to provide a range of data exploration and querying paradigms in order to promote broad uptake of the Semantic Web technologies. While there is a measurable effort being made towards the development of storage and querying facilities for the Semantic Web, there is still a chasm between the support provided by querying technologies and the needs of certain classes of applications. In particular, investigative and analytical applications in domains such as national security, bioinformatics and business intelligence often have the need to identify relationships that exist in data. In such scenarios, it is crucial to be able find answers to questions like "How are A, B, C and D related?". Such queries referred to here as *Subgraph Extraction Queries*, specify some anchor points in a data graph with the semantics of "extracting" a data subgraph connecting the anchor points. Often, queries will include some constraints on the kinds of connections to be included in the result e.g., connecting paths must contain some mandatory nodes or edges. The following describes two real world analysis tasks involving the use of such queries.

Scenario Example 1. (*Flight and Airport Risk Assessment*) To assess a potential threat to the safety of flights to certain airports, security officials would like to investigate all high risk passengers scheduled for such flights.

Find any relationships between passengers on flights to New York or Washington DC, who either purchased their tickets less than 24hrs before departure time or purchased their tickets by cash, particularly links associated with flight training.

Scenario Example 2. (*Analysis of gene interactions involved in advanced ovarian cancer adapted from* [14]). Microarray analysis data of tissue from advanced ovarian cancer revealed 1191 differentially expressed genes when compared to normal samples. Researchers will like to analyze these genes with respect to the biological pathways that they participate in to help understand the mechanism of action of the disease.

Show the interaction network for all genes that are differentially regulated in advance stage papillary serous ovarian cancer with respect to the signaling pathways. Constrain the results to those genes expressed in epithelial cells.

Both queries seek to retrieve paths or networks connecting specific nodes and the qualifying paths are subject to some constraints e.g. *associated with flight training* or *genes expressed in epithelial cells*. Other example applications arise in the transportation and telecommunications domains where network analysis is used for planning; in financial applications such as anti-money laundering analysis [25] and detecting potential biomedical patent infringement [22]. Unfortunately, as far as we know, none of the existing Semantic Web query languages support the expression of such queries about constrained path relationships. The sentiment about the need to extend Semantic Web query languages has been echoed by some researchers [3] who have argued for querying RDF graphs from a graph perspective. This will enable useful query paradigms

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

in Semantic Web query languages by interpreting them as standard graph operations like path finding, neighborhood searches, reachability and closure queries.

1.1 Background and Related Work

There are now several RDF query languages including RDQL[24], TRIPLE[27] N3, RxPath[28], RQL[20] SeRQL[9], RDFQL, Versa and the current W3C standard RDF query language proposal SPARQL[2] which is a successor language to RDQL[24]. A query algebra RAL[15] has also been proposed as a vehicle for studying RDF query languages. These languages exhibit one of four styles: SQL-like languages SPARQL, RDQL, SeRQL; functional query languages such as RQL, SeRQL; rule-based languages such as TRIPLE, N3 and graph traversal languages like Versa and RxPath. CORESE[13] is a Semantic Web search engine based on conceptual graphs that offers an RDF query language that uses the triple syntax. Our discussion of these languages will be limited to their expressiveness with respect to path query classes. For a detailed comparative analyses of these languages see [3][18]. In general, these languages focus on supporting graph pattern matching queries via path expression constructs. A notable exception is RDFPath which does allow for the expression of certain path queries if the source node is fixed. However, queries such as shortest path queries or those involving more complex constraints are not supported. It is also possible to express fixed path length queries in CORESE[13]. However, no other types of path queries are supported. In the languages that offer only a pattern matching paradigm, one could get around their limited expressiveness if the need is to express fixed path length queries using a union of queries capturing each of the lengths. However, this is very cumbersome and optimizing such queries is non trivial. There is a new proposed extension to SPARQL called PSPARQL[2] which allows query graph patterns involving regular expressions. This provides a lot of flexibility in expressing graph patterns that can be matched. However, PSPARQL like most of the other languages supports only a pattern matching query paradigm. Another class of queries that is very relevant to many applications is closure queries. Certain closure queries are naturally expressible in the rule based languages such as TRIPLE and N3, although Versa provides means to express transitive closure queries.

Of the existing RDF storage and querying systems [1][7][9][12] [16][31], a few [16][31] support bounded length and shortest path finding operations via a programmatic interface but without a query language. A limitation of most of these systems is that they either rely on traversal algorithms on main-memory graph representations - disadvantageous for massive graphs, or relational database storage where these queries are evaluated using multi-way joins. What would be desirable is to investigate the possibility of newer storage and indexing approaches that will support efficient processing of path queries. An effort in this direction is [6] which proposes an index structure called ρ-index for optimizing ρ-queries [4] whose foundation is path queries. However, the ρ-index approach is based on a technique that transforms general graphs into trees blowing up the size of the resulting graph when there are many non-tree edges. This paper goes in the direction of enabling subgraph extraction queries however, the discussion is limited to only path extraction queries or path queries. It addresses both issues in query expression and query evaluation. Our specific contributions are as follows:

1.2 Contributions

- We propose an extended SPARQL query language called SPARQ2L (*Recursively Protocol And RDF Query* ***and Link*** *Language*) which introduces the concept of *path variables* along with *path constraint expressions* that allow for a wide variety of useful path and reachability queries to become expressible over RDF data.
- We present the formal syntax and semantics of SPARQ2L
- We propose a novel persistent storage and query evaluation framework that involves (i.) a preprocessing phase on RDF graphs that computes partial path fragments which are indexed and stored on disk based on a novel labeling scheme. (ii) a query processing phase that retrieves relevant path fragments from disk via the index and assembles them into complete path representations. The framework proposed adapts efficient algebraic techniques for solving path problems.
- We present a comprehensive evaluation of the fundamental part of our evaluation framework. The evaluation is performed on both real world and synthetic benchmark datasets and shows an average of a 60% improvement in query response time with our labeling scheme.

2. THE SPARQ2L LANGUAGE

2.1 Language Requirements

Based on our analysis of commonly expressed queries on different kinds of data networks such as biological networks and social networks, we developed some language requirements for SPARQ2L. First, we identify three classes of constraints that should be supported:

1) *Constraints on Nodes and Edges* - constrain paths based on the presence or absence of certain nodes or edges.
2) *Cost-based constraints* – for weighted graphs, constrain paths based on their "costs".
3) *Structure-based constraints* – constrain paths based on structural properties e.g. non-simple paths, presence of a pattern on path, etc.

To allow the expression of such queries in SPARQ2L, we introduce the notions of *path variables* and *path filter expressions* i.e. constraints on path variables into SPARQL. In addition, we will extend the query patterns supported to include *RDF path patterns* which generalize standard SPARQL *graph pattern expression* to include triple patterns with path variables in the predicate positions. In the sequel, we will give an algebraic formalization of RDF path patterns that builds on the formalization of SPARQL graph pattern queries in [17].

Let I, L and B be pairwise disjoint infinite sets of IRIs, literals and blank nodes respectively and are collectively referred to as *RDF Terms*. An *RDF triple* is a 3-tuple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ where s is the *subject*, p is the *predicate* and o is the *object*. A *directed edge-labeled graph* is a graph $G = (V, E, \lambda, \Sigma)$ such that $E \subseteq V \times V$, and λ is a function from E to a set of labels Σ i.e. $\lambda: E \rightarrow \Sigma$. An *RDF triple graph* for an RDF triple (s, p, o) is a directed edge labeled graph $G = (\{s, o\}, \{(s, o)\}, \lambda, \{p\})$. An *RDF database graph* for a set of triples $\{t_1, t_2, .. t_n\}$ is the directed edge labeled graph formed from the union of the triple graphs for t_1, t_2, and t_n. An *RDF path* from node x to node y in an RDF database G is a sequence of triples $<(x, p_1, o_1), (s_2, p_2, o_2), \ldots, (s_k, p_k, y)>$ such that $o_i = s_{i+1}$, $i = 1, 2, \ldots, k-1$. An RDF path is *simple* if for all i,j, $i \neq j$ implies $o_i \neq o_j$ i.e., no node is repeated on the path, otherwise it is *non-simple*.

2.2 A Formal Syntax for SPARQ2L Path Pattern Expressions

SPARQ2L's *path pattern expressions* resemble SPARQL triple patterns except that they could contain path variables in the predicate position. A *path pattern expression* could consist of a SPARQL graph pattern, a *triple pattern* [17] with a path variable in the predicate position (called a *path triple pattern*) and some *built-in path filter conditions*. To describe this formally, we need a few definitions. Let VN and VP (regular and path variables respectively) be two pairwise disjoint sets of variables that are also disjoint from $I \cup L \cup B$. A *triple pattern* is a tuple in $(I \cup L \cup VN) \times (I \cup VN) \times (I \cup L \cup VN)$ and we denote the set of all triple patterns as T. To be able to specify a desired pattern on a path we need to support regular expressions over triple patterns. Recall that a regular expression over a set X is an expression formed in the following way: if $x \in X$, then x, (x)*, (x)+, (x)? are regular expressions; if x and y are regular expressions, then $x \bullet y$, $x \mid y$ are also regular expressions. A *T-Regular Expression* is a regular expression over a triple pattern or an extended regular expression of the form ([s , ·], p, [· , o])+ where (s, p, o) a triple pattern. An extended regular expression matches a path such that the subject of the first triple in the path is s and object of last triple is o, · matches arbitrary intermediate nodes on the path and all the predicates on the path are p. We define $\mathcal{R}(T)$ as the set of regular and extended regular expressions over T.

Definition 1 (*Path Built-in Condition*) Given the following built-in path functions:

- containsAny : $(VP, 2^I) \rightarrow$ Boolean
- containsAll : $(VP, 2^I) \rightarrow$ Boolean
- containsPattern : $(VP, \mathcal{R}(T)) \rightarrow$ boolean
- isSimple : $VP \rightarrow$ Boolean
- cost : $VP \rightarrow \mathbb{R}$

A *path built-in condition* is an expression built from $I \cup VP \cup L \cup \mathcal{R}(T)$, the logical operators $(\neg, \wedge)$, the comparison operators $(=, \leq)$ and path builtin functions in the following manner - given a variable $??P \in VP$, a constant c, $C \subseteq I$ a set of IRIs, TP is a T-regular expression, the following are path built-in conditions:

1) cost(??P) = c, containsAny(??P, C), containsAll((??P, C), containsPattern(??P, TP), and isSimple(??P).
2) If BC_1 and BC_2 are path built-in conditions, then $(\neg BC_1)$ and $(BC_1 \wedge BC_2)$ are path built-in conditions.

We can now define the notion of a SPARQ2L *Path Pattern Expression*.

Definition 2 (*Path Pattern Expression*) A *Path Pattern Expression* PP is defined recursively as follows:

- a 3-tuple $q \in (I \cup VN \cup L) \times VP \times (I \cup VN \cup L)$ called a *path triple pattern* is a path pattern
- if GP is a SPARQL graph pattern and PP is a path pattern then (PP AND GP) is a path pattern
- if PP is an path pattern and F is a path built-in condition then (PP PATHFILTER F) is a path pattern

2.3 The Semantics of Path Queries in SPARQ2L

In [17], the semantics of a SPARQL graph pattern is defined in terms of a function $[[\cdot]]$ which takes a pattern expression and returns a set of *mappings* where a *mapping* μ is defined as a partial function from VN to RDFT, $RDFT = I \cup L \cup B$. The function $dom(\mu)$ is used to denote the subset of VN in which μ is defined. We extend this definition for SPARQ2L's path patterns.

Let 2^{RDFT} be the set of possible tuples from RDFT. We introduce the notion of a *pmapping* ω as a partial function from $(VP \cup VN)$ to $(2^{RDFT} \cup RDFT)$ such that $\omega(vp \in VP) = p \in 2^{RDFT}$ and $\omega(vn \in VN) = RDFT$. Then, for a path triple pattern tp, we denote by $\omega(tp)$, the tuple formed by substituting any variables $vn \in VN \cup vp \in VP$ in tp according to ω. The dom of ω is the subset of $VP \cup VN$ in which ω is defined and is denoted by $dom(\omega)$. We extend the notion of *compatibility* defined in [17] to include compability between a mapping μ and a pmapping ω. We say that a mapping μ is *compatible* with a pmapping ω if when $x \in dom(\mu) \cap dom(\omega)$, then $\mu(x) \in \omega(x)$. Next, we define the *join* of a set of mappings Ω and a set of pmappings Θ in the following way:

$$\Omega \bowtie \Theta = \{\mu \cup \omega \mid \mu \in \Omega, \omega \in \Theta \text{ are compatible}\}$$

Definition 3. (*Path Pattern Solution*) Let D be an RDF dataset over RDFT, tp a path triple pattern whose variables are defined by var(tp) and GP_1 a graph pattern. Then, the solution of a path pattern PP over D, denoted by $[[\cdot]]_D$ is defined as recursively as follows:

i. $[[tp]]_D = \{ \omega \mid dom(\omega) = var(tp)$ and $\omega(tp)$ forms a path in D $\}$
ii. $[[(PP \text{ AND } GP)]]_D = [[PP]]_D \bowtie [[GP]]_D$

For the path patterns with PATHFILTER expressions, we say that a pmapping ω satisfies a builtin condition F or $\omega \models F$ if given I' a subset of the set of IRIs and tr a tp-regular expression,

i. F is containsAny(??P, I') and $??P \in dom(\omega)$ and $I' \cap \omega(??P) \neq \emptyset$.
ii. F is containsAll(??P, I') and $??P \in dom(\omega)$ and $I' \subseteq \omega(??P)$.
iii. F is containsPattern(??P, tr) and $??P \in dom(\omega)$ and ground(tr) is a subpath of $\omega(??P)$.
iv. F is isSimple (??P') and $??P \in dom(\omega)$ and for x, y $\in \omega(??P)$, $x \neq y$.
v. F is $(\neg F_1)$, F_1 is a built-in condition, $\omega \not\models F_1$
vi. F is $(F_1\ F_2)$, F_1 and F_2 are built-in conditions, $\omega \models F_1$ and $\omega \models F_2$

2.4 SPARQ2L By Example

This section gives a feel of the SPARQ2L grammar by examples.

Query 1. (*Non-Simple Path Query*) Find any feedback loops (i.e. non simple paths) that involve the compound Methionine

```
SELECT ??p
WHERE  {  ?x  ??p  ?x .
          ?z  compound:name  "Methionine" .
          PathFilter(containsAny(??p, ?z) ) }
```

Query 2. (*Path Query with Terminal Node Constraints*) Is PassengerX connected in anyway to entities on the CIA watchlist.?

```
SELECT ??p
WHERE  {  ?x  ??p  ?y .
```

```
        ?x  foaf:name  "PassengerX" .
        ?y  rdf:type   sec:CIA_Watchlist_Entities .  }
```

Query 4. (*Path Query with Constraint on Intermediate Nodes*) Find the paths of influence of Mycobacterium Tuberculosis MTB organism on PI3K signaling pathways.

```
SELECT ??p
WHERE  {  ?x  ??p  ?y .
          ?x  bio:name  "MTB Surface Molecule" .
          ?y  rdf:type  bio:Cellular_Response_Event .
          ?z  rdf:type  bio:PI3K_Enzyme .
          PathFilter(containsAny(??p, ?z) ) }
```

Query 5. (*Path Query with Path Length Constraint*). Find all close connections (< 4 hops) between SalesPersonA & CIO-Y.

```
SELECT ??p
WHERE  {  ?x  ??p  ?y .
          ?x  foaf:name  "salesPersonA".
          ?y  company:is_CIO ?z.
          ?z  company:name "CompanyY" .
          PathFilter( cost(??p) < 4 )    }
```

Query 6. (*Path query with path pattern constraint*) Find social relationships between potential jurors and a defendant.

```
SELECT ??p
WHERE  {  ?x  ??p  ?y .
          ?x  foaf:name  "defendantX" .
          ?y  foaf:name  "jurorY" .
          PathFilter( containsPattern (??p, [?a, ·]
          foaf:knows [· , ?b] )+ )                  }
```

3. QUERY EVALUATION FRAMEWORK

Our query evaluation framework derives from an algebraic technique for solving path problems [29][30] which has a strong relationship to the Gaussian elimination technique for solving a system of linear equations by LU decomposition. Recall that to solve a system of equations using this technique, a matrix representing a system of linear equations $Mx = b$ is decomposed into two triangular matrices L and U. Then, the system is solved by first solving the system $Ly = b$ (frontsolving) then substituting y in the system $Ux = y$ and solving for the vector x (backsolving). These triangular systems L and U can be used to solve for different right hand sides i.e. different values of b, allowing for the computationally dominant phase (the LU decomposition phase) to be reused for different problem instances. In [29][30], the authors show how by interpreting the sum and product operations appropriately, we can solve a variety of path problems using this technique. In general, solving a path problem instance using the triangular matrices, we process each triangular matrix in a specific order. Our work focuses on indexing and storing the contents of these matrices so that we may "skip" processing submatrices that are irrelevant to a query.

3.1 System Architecture Overview

Figure 1 shows our system for *multi-paradigm* querying of RDF which includes support for *pattern matching queries*, *path queries* as well as *keyword queries*. The first step in our approach is to *load* RDF Schema and data documents into internal graph data structures. Then, different *preprocessing* steps are performed on the data which produces appropriate indexes on the data for each of the querying paradigms i.e. *Pattern Matching Indexes* stored in the *Pattern Match Store*, *Path Index* stored in the *Path Store* and statistical and structural summaries used for *Top-k queries* stored in the *System Catalog*.

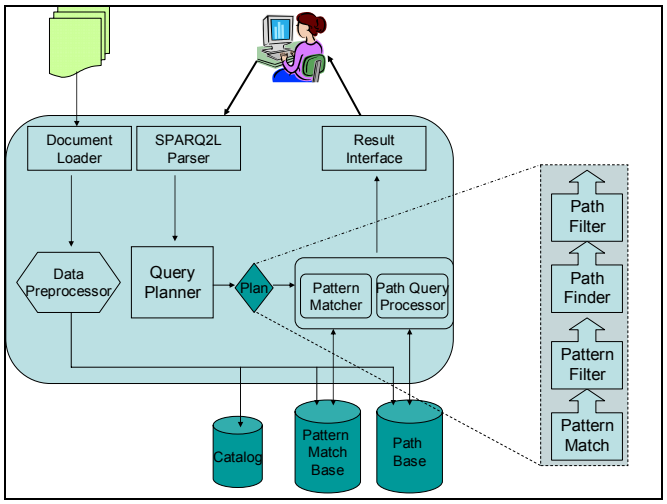

Figure 1: System Architecture

Our storage layer uses the BerkeleyDB data storage system because of its flexibility with respect to accommodating non-relational storage models and arbitrary data types. The *Query Processor Module* consists of three different kinds of query processors for processing each type of query.

However, a query may be processed by multiple processors. For example, Figure 1 shows an example query plan for a path query in which some constraints involve standard graph pattern matching. For brevity, we have omitted most of the components used to support keyword queries.

3.2 Data Preprocessing

Our discussion on preprocessing will focus only on what is relevant for path query processing which is the construction, labeling and indexing of a graph's path sequence. The LU decomposition phase of preprocessing can be seen as computing *partial path summaries.* This means that for certain pairs of nodes, *some* of the paths connecting the nodes are computed at this phase. We use the term "summaries" to imply a concise representation of path information as opposed to an enumerated listing of paths. A good analogy for path summarization is that of representing the set of strings in a regular language using a regular expression. To give an example, assume that we have the following triples (x, p_1, y), (x, p_2, y), (y, p_3, z) represented as labeled edges in an RDF graph. Then, we can summarize the paths from x to z as $(p_1 \cup p_2 \bullet p_3)$. We will refer to a triple of such a regular expression and the source and destination nodes as a *P-Expression* e.g. $((p_1 \cup p_2 \bullet p_3), x, z)$.While the discussion here will continue to refer to p-expressions as strings, our approach uses a more efficient implementation. P-expressions are represented using a binary encoding scheme that enables the path filtering step for path constraint evaluation to be performed efficiently using bit operations. However, a detailed discussion of the binary encoding scheme and path filtering algorithms is outside the scope of this paper.

The LU decomposition of phase of the preprocessing requires that an RDF graph G be *ordered* - $G_\alpha = (G, \alpha)$ where $\alpha : \{1, 2, \ldots N\} \rightarrow V(G)$ so that $\alpha(i)$ maps to some node v in G i.e. $v \in V(G)$. Conversely, $\alpha^{-1}(v)$ maps a node in G to an integer between 1 and N. (Our choice of α will be discussed in the next section). At the end of

the LU decomposition algorithm [29], the elements of M satisfy one of two conditions: For $u, v \in V(G)$:

- $M[\alpha^{-1}(u), \alpha^{-1}(v)]$ for $\alpha^{-1}(u) \geq \alpha^{-1}(v)$ contains a p-expression representing exactly the paths from u to v that do not contain any intermediate vertex w such that $\alpha^{-1}(w) > \alpha^{-1}(v)$.
- $M[\alpha^{-1}(u), \alpha^{-1}(v)]$ for $\alpha^{-1}(u) < \alpha^{-1}(v)$ contains a p-expression representing exactly the paths from u to v that do not contain any intermediate vertex w such that $\alpha^{-1}(w) < \alpha^{-1}(u)$.

The process begins by initializing $M[i, j]$ for $1 \leq i, j \leq N$ with a p-expression representing a union of the set of edges between the nodes $\alpha(i)$, $\alpha(j)$. Then, this union p-expression is systematically updated to represent other paths that satisfy the above constraints. A naïve algorithm for the LU decomposition phase runs in $O(N^3)$. Its details and other optimizations are omitted for brevity but can be found in [29]. Then, the *path sequence* [29] for G is: the sequence of p-expressions (X_i, u_i, v_i) where $\alpha^{-1}(u_i) \leq \alpha^{-1}(v_i)$ in increasing order on $\alpha^{-1}(u_i)$ is followed by the sequence of p-expressions (X_i, u_i, v_i) for $\alpha^{-1}(u_i) > \alpha^{-1}(v_i)$ in decreasing order on $\alpha(u_i)$. This notion can also be defined in terms of the *strongly connected components* of G_α [29] which leads to a more efficient technique for computing path sequences.

Definition 4 (*Path Sequence by Strong Components*) Let $G_1, G_2, \ldots, G_k$ be a topologically ordered list of the strongly connected components of a graph G i.e., for $i > j$, there does not exist an edge from a component G_i to a component G_j. Further, let A_i be the path sequence for G_i and B_i be the elements: $\{ (X, source(e), target(e)) \mid source(e) \in V(G_i)$ and $target(e) \notin V(G_i) \}$ ordered arbitrarily. Then $A_1, B_1, A_2, B_2, \ldots, A_{k-1}, B_{k-1}, A_k$ is a *path sequence* of G.

3.2.1 Labeling and Indexing Path Sequences

Our goal is to support efficient evaluation of path queries on disk-based databases which means that we need to develop an effective disk storage model for graphs. Now, a path sequence has what we will call the *Single-Scan-Path-Preserving* property which means that for any given node u in G, we can compute complete path information for u by aggregating the partial path fragments during a single scan of the path sequence. This suggests that it should be possible to index this sequence using a B+tree and then process queries using modified range queries. However, we must endeavour to minimize the width of the range retrieved to process each query. To achieve this goal, we should cluster p-expressions on the path sequence based on their likelihood of being relevant or irrelevant for the same class of queries. This will ensure that we minimize the number of disk requests and disk-seek operations needed when evaluating queries. On the other hand, a more fragmented organization of relevant and irrelevant p-expressions will lead to queries requiring many small relevant clusters that are scattered across the sequence and consequently many more disk seek operations. This clustering is achieved logically by using a graph numbering or labelling scheme that assigns groups of related nodes and therefore associated p-expressions numbers in contiguous intervals. To enforce this clustering on disk, we exploit the fact that in BerkeleyDB, insertions are *appended* to a log file and are physically stored in the order that they are inserted. So we need to insert related p-expressions in path sequence order. In the following section, we will consider some relationships between nodes that allow us to consider them related or "*Prunable Equivalent*" for some classes of queries.

3.2.1.1 Prunable Equivalence

We will explain this concept intuitively. Figure 2 shows an example RDF data graph with information about faculty, students, research projects, etc., and the relationships between them. Each node is labelled with a set of symbols in shaded boxes indicating its types (see legend at the bottom left of the picture). The dotted circles are the strong components (maximal subgraph where there is a path connecting each pair of nodes) of the graph. It is clear that for any non-singleton strong component i.e. strong component with more than one node, if a path contains one of its constituent nodes as an intermediate node, then there is a path containing all of its nodes as intermediate nodes. This means that if, given a path query we can determine that any constituent node of a strong component is "irrelevant" to the query, we can conclude the irrelevance of all the other constituent nodes and edges and losslessly ignore their associated p-expressions. Consequently, we say that all the p-expressions associated with the nodes and edges of a non-singleton strong component are "*Prunable Equivalent*".

We can also consider similar equivalence relationships amongst sets of strong component nodes which form interesting subgraph structures. For example, consider the small subgraphs at the bottom right of the figure enclosed in the boxes numbered 3, 4, 5 – numbers in thick bordered boxes. We call these structures *dangling trees* because they form a tree structure (in these examples they are path structures) that "dangle" from a parent non-tree subgraph – the ancestor subgraph enclosed in the thick bordered box 2. Interesting properties of these structures is that (i.) no paths connect any pair of dangling trees, (ii.) there does not exist any paths from a dangling tree back to the parent non-tree subgraph. From the point of view of a path query from s to d, this implies that if we can determine that d is not a node of a dangling tree T, then we can conclude the irrelevance of all the nodes in T and we can losslessly ignore all p-expressions associated with the strong component nodes and edges in T.

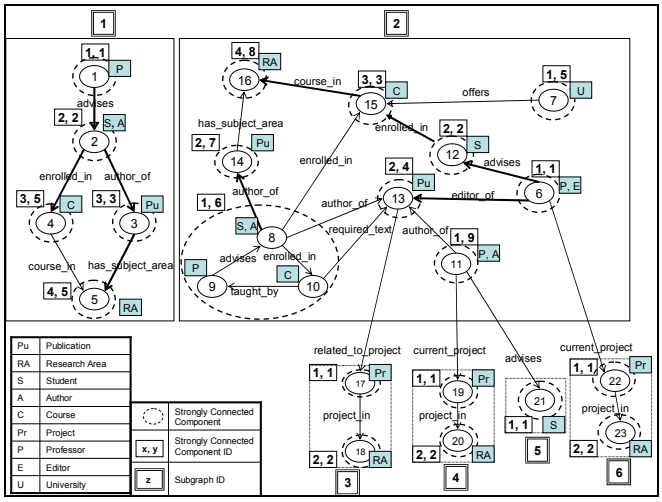

Figure 2: Hierarchical labelling of an RDF graph

Therefore, we say that the p-expressions of strong component nodes and edges in a dangling tree are *Destination-Induced Prunable Equivalent*. This relationship trivially applies to disconnected subgraphs such as two disconnected subgraphs (left, right) with double bordered boxes 1 and 2 respectively.

For the groups of strong components forming non-tree subgraphs, imposing a tree structure on them allows us to identify some interesting equivalence relationships. Recall that a *spanning tree* is a

tree subgraph of a graph that includes all the nodes in the graph. In the case of a graph with multiple source nodes, we introduce a root node as the parent of all source nodes. We call a spanning tree *optimal* if every edge (u, v) lies on the *longest path* from root of the tree to v. In Figure 2, the darkened edges are the edges of the optimal spanning tree (the introduced root node is omitted from the figure). This definition of the optimal spanning tree implies that for any node s at depth k in the tree, all of the nodes reachable from s are at level l > k while all nodes that can reach node s are at a level l < k. Consequently, we can say that the p-expressions for edges and strong component nodes at the same or lower depth in the optimal spanning tree are *Treedepth-Induced Prunable Equivalent*.

3.2.2 Framework for Graph Labeling

We would like to label the nodes and edges of a graph such that we can easily identify the groups of related nodes based on the relationships discussed in the previous section. To achieve this, we use a hierarchical labeling scheme that captures the above groupings at the three levels of abstraction. At the finest level, we have a *component identifier* which is a unique number assigned to each individual strong component which records the pre/post order visit time during a depth first traversal of the graph. In Figure 2, this is the second of the numbers in the rectangular boxes associated with strong components. The first number in the rectangle is the *level identifier* which is the depth of the strong component node in the optimal spanning tree. At the coarsest grouping we have what we call *subgraph identifiers* which identify disconnected non-tree subgraphs and the dangling tree subgraphs. The subgraph identifiers are the numbers in thick bordered boxes and the intervals of subgraphs identifiers for non-tree subgraphs and that of dangling tree subgraphs are non-overlapping. By default all the non-tree subgraphs are assigned subgraph identifiers lower than the tree subgraphs. The following statement states the properties of this labelling scheme that we exploit to determine relevance of strong component nodes and edges connecting them.

NonReachability Property (*PropertyNR***)**. Given a path query q with s and d as source and destination nodes, let i and j be the subgraph identifiers of s and d respectively, and x and y be their level identifiers. Then

- $i \neq j$ implies that q's result is empty.
- $x > y$ implies that q's result is empty.
- any node u with level identifier k such that $k < i$ or $k > j$ implies that u is not a member of q's result set.

3.2.2.1 2-Color Path Sequences

Based on the above labeling scheme we can develop an effective sequential representation for a graph which associates key values derived from the hierarchical labeling with elements of a path sequence. However, we need to take additional measures to ensure that the resulting sequence clusters "related" p-expressions. As a first step, we can co-index (assign same key value) all the p-expressions associated with a non-singleton strong component since they are prunable equivalent with respect to every query. This will allow for collectively retrieving or skipping over all the p-expressions associated with the strong component once its relevance is determined. For the same reason, we can co-index multiple edges connecting two strong components. For example, consider the components with rectangular boxes 1,6 and 2,4 which have two edges {"*author_of*", "*required_text*"} connecting them but involve different constituent nodes 8 and 10. By co-indexing both edges, we can collectively retrieve or skip their associated p-expressions once the relevance of the source and destination components have been determined. For similar reasons, we would also like to cluster on disk, destination-induced and treedepth-induced prunable equivalent p-expressions. To achieve this, we will exploit a property of the BerkeleyDB log file system where insertions are always appended to a log file, so that clustering can be achieved using consecutive insertions. We will now propose a sequential representation that we can index using a BerkeleyDB B+tree. The next definition shows how we construct the sequential representation called a *2-Color Code.*

Definition 5. (2-Color Code) Let a *label component* be a triple of one of the forms (s, l, t) or (s, t, l) where s is *subgraph identifier*, l is *level identifier* and t is *traversal identifier*. Then the 2-Color Code C for a graph G = (V, E) is a lexographically ordered sequence of 2-tuples of label components – denoted as SLT() and STL() for the two label component forms, such that:

- *for* $v \in V$, *if* v *is in a dangling tree then the tuple* $\langle STL(v), STL(v)\rangle \in C$ *otherwise* $\langle SLT(v), SLT(v)\rangle \in C$
- *for* $e = (v_i, v_j) \in E$, *if* v_j *is in a dangling tree of* G *then* $\langle STL(v_j),$ label component of $v_i\rangle \in C$ *otherwise* $\langle STL(v_i), STL(v_j)\rangle \in C$

We call each 2-tuple of label components a 2-Color Label denoted as 2CL(). The two different forms of label components and the reversing of source and destination label components in the 2-Color code of edges connecting strong components is done to achieve an ordering that simulates the most appropriate way to process each type of subgraph - *depth-first* ordering for tree subgraphs and a *breadth first* ordering for non-tree subgraphs. The ordering induced by a 2-Color Code defined as above can be summarized the following way.

Order Property (*PropertyOP*): For D and T, non-tree and tree subgraphs respectively of a graph G,

- $u \in V(D)$ and $v \in V(T)$ implies $2CL(u) < 2CL(v)$. Similarly, $e \in E(D)$ and $f \in E(T)$ implies $2CL(e) < 2CL(f)$.
- $u \in V(D)$ and $e = (u, v) \in E(D)$ implies $2CL(u) < 2CL(e)$ while u in V(T) and $u = (v, u) \in E(T)$ implies $2CL(u) > 2CL(e)$.

This says that the ordering induced by the 2-Color code is such that all p-expressions associated with non-tree subgraphs appear before the p-expressions of tree subgraphs. Also, within each disconnected subgraph the p-expressions are in level order for non-tree subgraphs and depth-first order for tree subgraphs. Finally, for non-tree subgraphs p-expressions associated with an edge appears after the p-expression for the edge's source node whereas the reverse is the case for tree subgraphs. These properties are exploited during query processing as will be seen in the next section. Figure 3 shows a labelled path sequence of 43 key-value pairs. Singleton strong components have a key representing the node and an empty value set e.g. element 1. Non-singleton strong components have a key and a value which is the set of p-expressions of its constituent nodes and edges e.g. element 16. All other keys identify two connected components that have at least one edge connecting them. The key represents the pair of nodes and the value is the set of p-expressions for all the connecting edges e.g. element 17.

1: [(1,1,1), (1,1,1), { }], **2:** [(1,1,1), (1,2,2), { (advises, 1, 2) }], **3:** [(1,2,2), (1,2,2), { }],
4: [(1,2,2), (1,3,3), { (author_of, 2, 3) }], **5:** [(1,2,2), (1,3,5), { (enrolled_in, 2, 4) }],
6: [(1,3,3), (1,3,3), { }], **7:** [(1,3,3), (1,4,5), { (has_subject_area, 3, 5) }],
8: [(1,3,5), (1,3,5), { }], **9:** [(1,3,5),(1,4,5), { (course_in, 3, 4) }], **10:** [(1,4,5), (1,4,5), { }],
11: [(2,1,1), (2,1,1), { }], **12:** [(2,1,1), (2,2,2), { (advises, 6, 12) }],
13: [(2,1,1), (2,2,4), { (editor_of, 6, 13) }], **14:** [(2,1,5), (2,1,5), { }],
15: [(2,1,5), (2,3,3), { (offers, 7, 15) }], **16:** [(2,1,6), (2,1,6), { (enrolled_in, 8, 10), (advises•enrolled_in, 9, 10), ((taught_by•advises•enrolled_in)*, 10, 10), (taught_by, 10, 9), (advises, 9, 8) }],
17: [(2,1,6), (2,2,4), { (author_of, 8, 13), (required_text, 10, 13)}],
18: [(2,1,6), (2,2,7), { (author_of, 8, 14) }], **19:** [(2,1,6), (2,3,3), { (enrolled_in, 8, 15) }],
20: [(2,1,9), (2,1,9), { }], **21:** [(2,1,9), (2,2,4), { (author_of, 11, 13) }],
22: [(2,2,2), (2,2,2), { }], **23:** [(2,2,2), (2,3,3), { (enrolled_in, 12, 15) }],
24: [(2,2,4), (2,2,4), { }], **25:** [(2,2,7), (2,2,7), { }],
26: [(2,2,7), (2,4,8), { (has_subject_area, 14, 16) }], **27:** [(2,3,3), (2,3,3), { }],
28: [(2,3,3), (2,4,8), { (course_in, 15, 16) }], **29:** [(2,4,8), (2,4,8), { }],
30: [(3,1,1), (2,4,2), { (related_to_project, 13, 17) }], * * * **31:** [(3,1,1), (3,1,1), { }],
32: [(3,2,1), (3,1,1), { (project_in, 17, 18) }], **33:** [(3,2,2), (3,2,2), { }],
34: [(4,1,1), (2,9,1), { (current_project, 11, 19) }], **35:** [(4,1,1), (4,1,1), {}],
36: [(4,2,1), (4,1,1), { (project_in, 19, 20) }], **37:** [(4,2,2), (4,2,2), { }],
38: [(5,1,1), (2,9,1), { (advises, 11, 21) }], **39:** [(5,1,1), (5,1,1), { }],
40: [(6,1,1), (2,1,1), { (current_project, 6, 22) }], **41:** [(6,1,1), (6,1,1), { }],
42: [(6,2,1), (6,1,1), { (project_in, 22, 23) }], **43:** [(6,2,2), (6,2,2), { }]

Figure 3: 2-Color Code for Example Graph

For a graph of n nodes, m edges and k strong components, the overall time for preprocessing includes the time to find strong components $O(m + n)$, the optimal spanning tree $O(k + m')$ where $m' < m$ is the number of edges connecting strong components, and the time to run the LU decomposition algorithm for all strong components: $O\sum_{i=1 to\ k} {n_i'}^3$ where n_i' is the number of nodes in strong component i.

3.3 Path Query Processing

This presents the approach for evaluating unconstrained path queries which is also fundamental for evaluating constrained queries. The discussion of constrained queries is outside the scope of this paper.

3.3.1 Evaluation of Unconstrained Path Queries

The Path Finder evaluates a query by successively retrieving the relevant p-expressions from disk and composing them into larger p-expressions that comprise the solution. Path Finder achieves this using the Path-Solve algorithm shown in Listing 1 below. The algorithm begins by initializing a matrix (*Result*) which keeps track of the composed p-expressions. To retrieve p-expressions from disk, the *openDBCursor* sub-routine returns a database *non-treeCursor*, *treeCursor* or *joinCursor*, depending on the subgraph in which the source and destination of the query is located.

A non-treeCursor (treeCursor) is always set to the 2-Color label for the strong component of the source (destination) node. The p-expressions needed to process the query are obtained using the *next* cursor function, until the end of the cursor (i.e. the 2-Color label for the strong component of the destination (source)) is reached. To illustrate this, consider a query for paths from node 8 to node 16 in **Figure 2**. These nodes are in the same non-tree sub-graph thus a non-tree cursor set to 16th element in Figure 3 is returned by *openDBCursor*. In the *processNon-Tree()* sub-routine, the *next* cursor function returns the 17th to the 29th elements which are processed by calls to *processPE* sub-routine.

```
Algorithm 2 Path-Solve(Node s, Node d, int Result[])
01 Result[id(s)] = ε        //id(x) = α-1(x) ie the id of node x
02 for each v ∈ V – {s} set Result[id(v)] = ∅
03 cursor ← openDBCursor(s, d)
04 if (cursor is treeCursor)
05      Result ← processTree(cursor, Result, α-1(d))
06 else if (cursor is non-treeCursor)
07      Result ← processNon-Tree(cursor, Result)
08 else
09      Result ← processJoin(cursor, Result)
10 return Result
openDBCursor(Node s, Node d)
01 if ((s & d) ∈ same non-tree sub-graph)
02      return non-treeCursor(s, d)
03 else if ((s & d) ∈ same tree sub-graph)
04      return treeCursor(s, d)
05 else if (s ∈ sub-graph g & d ∈ a dangling tree t of g)
06      return joinCursor(s, c, d) //c is the cut vertex of g ∪ t
07 else return null.
processNon-Tree(cursor, int Result[])
01 while ((Xi, vi, wi) ← cursor→next()) <> null)
02      Result = processPE((Xi, vi, wi))
03 return Result
processTree(cursor, int Result[], int dest)
01 Result[dest] = ε
02 while ((Xi, vi, wi) ← cursor→next()) <> null)
03      Result[dest] = processPE((Xi, vi, wi))• Result[dest]
04      (Xi, vi, wi) ← cursor→prev()
05      Result[dest] ← Xi • Result[dest]
06      cursor→set(vi)
07 return Result
processPE(int Result[], p-expression (Xi, vi, wi))
01 if (vi = wi ) then Result[wi] ← Result[vi] • Xi
02 else Result[wi] ←Result[wi] ∪ Result[vi] • Xi
03 return Result
```

Listing 1: Path-Solve Algorithm

On the other hand, *processTree* proceeds in a bottom-up manner, successively using the *next* and *prev* cursor functions to obtain the relevant p-expressions. Since p-expressions obtained from a treeCursor begin with the destination up to the source, p-expressions of tree edges (obtained with the *prev* cursor function) are prepended to the p-expression in *Result[*α^{-1}*(d)]*.

Given *Result* and a p-expression (X_i, v_i, w_i), *ProcessPE* computes a larger p-expression (X_i, u_i, w_i) by appending (X_i, v_i, w_i) to an existing p-expression in *Result* of type (X_j, u_i, v_j) (line 01). Further, a larger p-expression is computed as the union of any p-expression (X, u_i, w_i) which already exists in *Result*, but only if v_i is different from w_i. As an example, consider again the query for all paths from node 8 to node 16th. *Result[8]* initially contains ε, as shown in Figure 4a. To process the first p-expression *(enrolled_in, 8, 10)* of the 16th element, *Result[8]* is concatenated to *enrolled_in* and stored in *Result[10]*, as shown in Figure 4b. Processing *(advises•enrolled_in, 9, 10)* implies concatenating *Result[9]* to *advises•enrolled_in* and storing to *Result[10]*. However *Result[9]* is null, thus *Result[10]* remains unchanged. Figure 4c-j, show the changes made to *Result* as the rest of the p-expressions are processed.

A joinCursor retrieves p-expressions like a treeCursor until it meets p-expression for the cut vertex, after which it switches to a non-treeCursor behavior. To illustrate, consider a query for paths from node 11 to node 20. Node 11 is in a non-tree sub-graph which has a dangling tree that contains node 20. Thus, *openDBCursor* returns a joinCursor set to the 37th element in Figure 3. Its associated p-expression is empty, so that *Result[20]* is unchanged. A call to the *prev* cursor function returns the p-expression *(project_in, 19, 20)*, which is prepended to ε, so that *project_in* is stored to *Result[20]*.

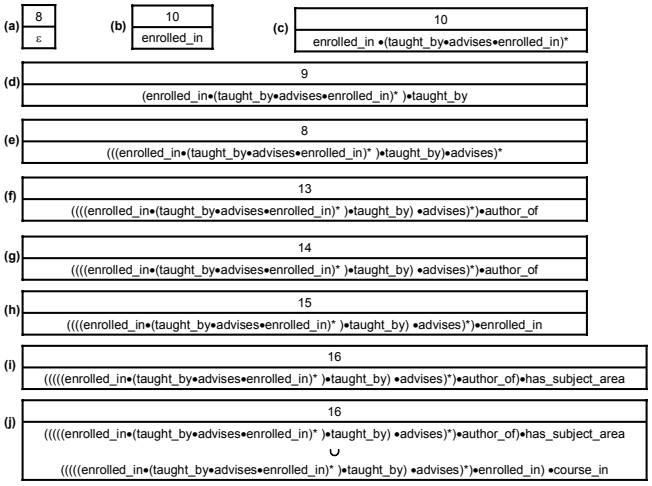

Figure 4:An illustration of the Path-Solve

The cursor is then set to the 35th element, which also contains an empty p-expression. The next call to *prev* returns the 34th element with the p-expression *(*current_project, *11, 19)*, which is prepended to *project_in* and stored back to *Result[20]*. Since this element represents the bridge edge, the cut vertex is noted and *set* sets the cursor to the 2-Color label of the source of the query, which is the 20th element. Since its associated p-expression is empty, no change is made to *Result[11]*. Since this element is the cut vertex, the result of the query is obtained with a final concatenation of *Result[11]* and *Result[20]*.

At the end of the Path-Solve algorithm, *Result[i]* contains a p-expression of type $(X_i, \alpha^{-1}(s), w_i)$, representing all paths from node s to node i.

4. PERFORMANCE EVALUATION

In this section, we describe an empirical evaluation of our query processing approach by comparing the performance when using our 2-Color Code (2CC) vs. five other several other randomly chosen topological orderings.

4.1 Experimental Setup

Implementation. We implemented our algorithms using Java 1.5, on a 1.8GHz Dual AMD Opteron processor with 10GB available RAM. We used Berkeley DB Java Edition for storage and indexing and performed all matrix implementations using the sparse matrix implementation of the Colt distribution. We used Brahms [16], an efficient main-memory storage to obtain a temporary graph representation of the RDF graphs in memory.

Datasets. We used a real world SwetoDBLP-Jan2006 [32] dataset and a synthetic dataset generated using the Lehigh University Benchmark with 6 Universities (UBA6). Table 1 below shows the properties of the datasets. SwetoDBLP has 9,921 non-tree sub-graphs with a total of about 300,000 scc nodes and 760,000 scc edges and 340,000 tree sub-graphs with a total of about 410,000 scc nodes. One of these non-tree subgraphs is very large (about 250,000 nodes and 660,000 edges) and the smallest sub-graph contains 2 scc nodes and 2 scc edges. It also contains about. The smallest and largest tree sub-graphs have a single scc node and 25 scc nodes respectively, with a maximum depth of 1. UBA6 however is more connected, containing a single non-tree subgraph with 118,195 strongly connected component (scc) nodes and 357,578 scc edges. Although it contains 61 tree subgraphs, each tree contains just a single scc node. Literal nodes and incident edges are ignored.

Table 1: Properties of the Datasets

	UBA6	SWETO_DBLP
Number of nodes	118,566	724,874
Number of edges	357,950	836,555
# of strong components	118,256	723,669
#of p-expressions	476,448	1,561,008

Performance Metrics. We evaluate the performance of the techniques by observing 1) the size of the reduced path sequence i.e. the number of p-expressions brought into memory from disk. This metric measures the goodness of our approach for identifying irrelevant p-expressions – those that were not retrieved from disk, 2) the query processing time including disk access time.

Query Workload. Our query workload consists of six different query types. First we have as positive (path exists) and negative (paths do not exist) and we denote positive or connected queries as (C-Queries) and negative or disconnected queries as (D-Queries). Then we identify queries based on whether their source/destination nodes are in tree or non-tree subgraphs. Queries with source and destination nodes in non-tree sub-graphs are denoted (NT-NT) queries, (NT-T) denotes source node is in a non-tree sub-graph and the destination node in a tree sub-graph and (T-T) denotes queries with both source and destination nodes in tree sub-graphs. We randomly selected 40 distinct source-destination pairs for each of the six categories and measured the average running time of all queries where the running time of a query is also an average over several executions of the query.

4.2 Experimental Results

Figure 5a – Figure 5l show the result of our experiments on the SwetoDBLP dataset. 2CC had the best performance for all the query workloads. The best performance of 2CC for C-Queries is observed in the T-T queries where only 4 p-expressions were brought into memory, with a total of 0.4 milliseconds query processing time. This is natural since the T-T queries for this dataset are single edge paths. For the D-Queries, 2CC performed very well using at most 0.025 milliseconds to determine that the result set is null. For these queries, no p-expression was brought into memory. As is expected the performance of the labeling schemes varied with the queries. This can be observed in Figure 5a – Figure 5b and Figure 5e – Figure 5f.

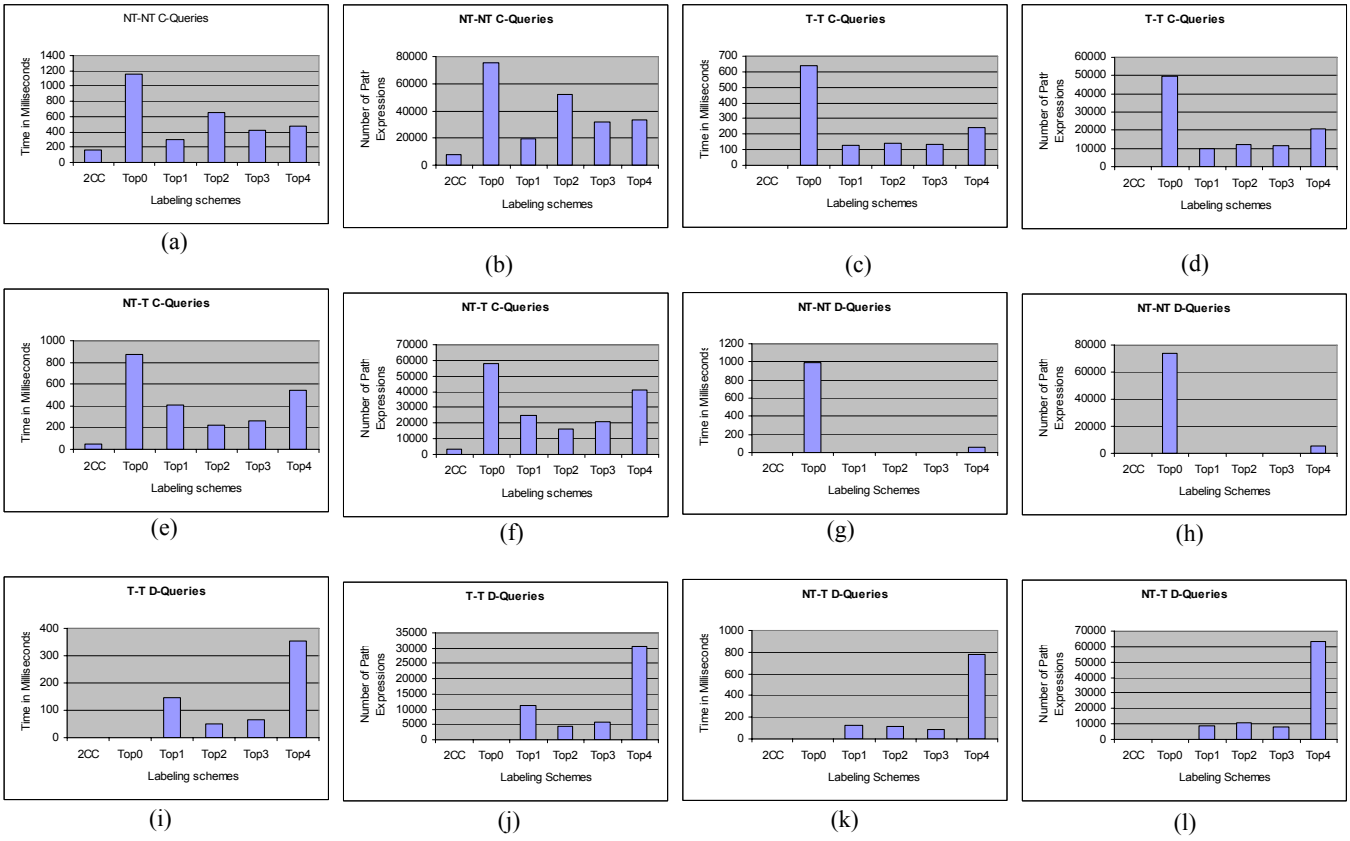

Figure 5: SwetoDBLP DataSet Figures

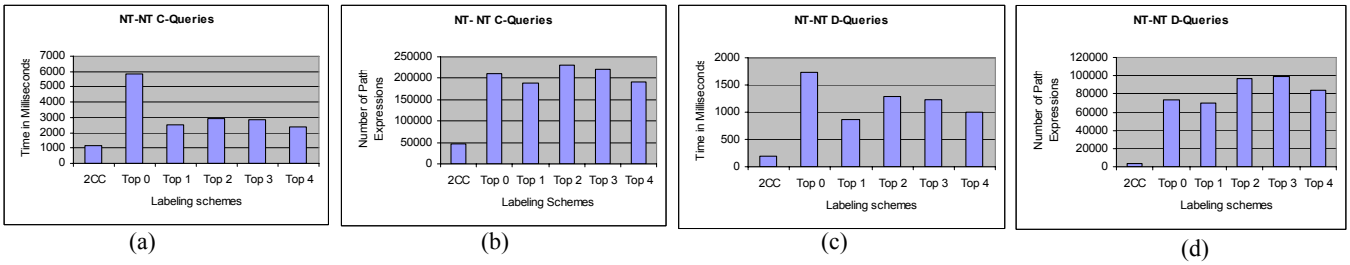

Figure 6: LUBM Dataset, Figures

While Top1 had the best performance amongst the topological labeling schemes for the NT-NT C-Queries, Top2 performed best for the NT-T C-Queries. Top0 performed worst for most of the query workloads but had the best performance in the NT-T D-Queries and T-T D-Queries (Figure 5i – Figure 5l). We note that although performance of 2CC surpasses that of all the other topological labeling schemes, the ratio of the time performance is always larger than the ratio of the size performance. This is as a result of the additional time the 2CC spends in pruning the p-expressions from the reduced path sequence based on properties PropertyNR and PropertyOP.

Figure 6a – Figure 6d show the results of our experiments on UBA6. As we mentioned earlier, the tree sub-graphs in UBA consist of only single nodes, so that T-T queries are meaningless for this dataset. Furthermore, NT-T queries translate to either (a) finding the bridge edges or (b) finding paths through the bridge edges. Our experiments on the SwetoDBLP dataset showed that the performance of 2CC is very good when finding single edge paths. We observed a similar performance for this dataset and omit the results on the NT-T queries which can be inferred from the performance of (b) the NT-NT queries.

Figure 6a and Figure 6b show the performance of the labelling schemes for the NT-NT C-Queries. Again, 2CC performs best amongst all the labeling schemes. However, its performance on UBA6 is worse than on SwetoDBLP because UBA6 is very connected, having only 1 non-tree sub-graph as opposed to SwetoDBLP which is fragmented into 9, 921 non-tree sub-graphs. Thus, queries in this workload have many more and longer paths. This is also reflected in the results of the NT-NT D-Queries shown in Figure 6c and 5d. Although 2CC also performed best, there were some queries for which determining non-reachability required more than a constant time check using labels leading the an average performance of 188 milliseconds for processing 3927 p-expressions. The disparity in the ratios of the time and size performances of 2CC to the other labelling schemes is also evident in the results. In spite of this, the time performance of 2CC is at least half of the time

performance of the best topological labelling scheme (Top1) for both the C-Queries and the D-Queries.

5. CONCLUSION

This paper addresses the issue of providing support for path extraction queries in RDF databases. This feature while crucial for many applications has limited support from most RDF querying systems. We address both the issues of how such queries can be expressed using SPARQ2L – an extended SPARQL language, and how to efficiently evaluate queries on disk based stores. In the future, we will address the issue of efficiently evaluating path extraction queries with complex filtering conditions.

6. ACKNOWLEDGMENTS

This work is funded by NSF-ITR-IDM Award#0325464 and #0714441 titled 'SemDIS: Discovering Complex Relationships in the Semantic Web.'

7. REFERENCES

[1] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In SemWeb 2001.

[2] Alkhateeb, F., Baget JF., Euzenat, J. *Complex path queries for RDF. Poster paper in* ISWC2005 , 6th - 10th Nov. 2005, Galway, (Ireland).

[3] Angles, R., Gutierrez, C. Querying RDF Data from a Graph Database Perspective , ESWC2005, May 2005, Heraklion, Greece.

[4] Anyanwu, K., Sheth, A. ρ-Queries: enabling querying for Semantic Associations on the Semantic Web. WWW 2003.

[5] Bailey, J. Bry, F., Furche, T., Schaffert, S. Web and Semantic Web Query Languages: A Survey.

[6] Barton, S: Designing Indexing Structure for Discovering Relationships in RDF Graphs. DATESO 2004

[7] D. Beckett. The design and implementation of the Redland RDF application framework. Computer Networks, 39(5):577--588, 2002

[8] Bonstrom, V., Hinze, A., Schweppe, H. "Storing RDF as a Graph," *la-web*, p. 27, (LA-WEB'03), 2003.

[9] J. Broekstra. *SeRQL:* Sesame RDF query language. In M. Ehrig et al., editors, SWAP Deliverable 3.2 Method Design, pages 55--68. 2003

[10] J. Broekstra, A. Kampman, F. v. H. 2001. Sesame: An architecture for storing and querying rdf data and schema information. In D. Fensel, J. Hendler, H. L., and Wahlster, W., eds., Semantics for the WWW. MIT Press.

[11] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, Kevin Wilkinson: Jena: implementing the semantic web recommendations. WWW (ATPP) 2004: 74-83

[12] Corby, O., Dieng-Kuntz, R., Faron-Zucker, C., Gandon. F. "Searching the Semantic Web: Approximate Query Processing Based on Ontologies," *IEEE Intelligent Systems*, vol. 21, no. 1, pp. 20-27, Jan/Feb, 2006.

[13] Corby, O., Dieng-Kuntz, R., Faron-Zucker, C.. *Querying the Semantic Web with the corese search engine*. ECAI/PAIS2004, Valencia (ES), August 2004. IOS Press.

[14] Donninger, H., Bonome, T., Radonovich, M., Pise-Masison, C. A., Brady, J., Shih, J. H., Barrett, J., Birrer, M. J. Whole genome expression profiling of advance stage papillary serous ovarian cancer reveals activated pathways. *Oncogene* (2004) **23,** 8065–8077

[15] Flavius Frasincar, Geert-Jan Houben, Richard Vdovjak, Peter Barna: RAL: An Algebra for Querying RDF. World Wide Web 7(1): 83-109 (2004)

[16] Janik, M., Kochut, K.: BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. ISWC2005: 431-445

[17] Jorge Perez, Marcelo Arenas and Claudio Gutierrez. Semantics and Complexity of SPARQL**.** ISWC'06, Athens, GA, USA, 2006.

[18] Peter Haase, Jeen Broekstra, Andreas Eberhart, Raphael Volz: A Comparison of RDF Query Languages. ISWC'04: 502-517

[19] Harth, A., Decker. S. Optimized index structures for querying RDFfrom the web. In LAWEB 2005.

[20] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Schol. RQL: A Declarative Query Language for RDF. WWW'02, Honolulu, Hawaii, USA, May7-11 2002.

[21] Lassila, O., Swick, R. RDF Model and Syntax Specification. W3C Recommendation

[22] Mukherjea, S., Bamba, B. BioPatentMiner: An Information Retrieval System for BioMedical Patents. VLDB 2004: 1066-1077

[23] Prud'hommeaux E., Seaborne, A. SPARQL Query Language for RDF. W3C Candidate Rec. 6 April 2006.

[24] **Seaborne**, A. *RDQL — A Query Language for RDF*, WWWConsortium, Member Submission SUBM-RDQL-20040109, January 2004.

[25] A. Sheth, B. Aleman-Meza, I. B. Arpinar, . Ramakrishnan, C. Halaschek, C. Bertram, Y. Warke, C David Avant, F. S. Arpinar, K. Anyanwu, K. Kochut, Semantic Association Identification and Knowledge Discovery for National Security Applications, JDM,16 (1), Jan-March 2005.

[26] Wolf Siberski, Jeff Z. Pan, Uwe Thaden: Querying the Semantic Web with Preferences. ISWC 2006: 612-624

[27] M. Sintek and S. Decker. TRIPLE - an RDF query, inference and transformation language. In DDLP, 2001.

[28] A. Souzis. RxPath specification proposal. http://rx4rdf.liminalzone.org/RxPathSpec

[29] Tarjan, R. E. "Fast Algorithms for Solving Path Problems". JACM, Vol. 28, No. 3, July 1981, pp. 594-614

[30] Tarjan. R.E**.** A Unified Approach to Path Problems**.** JACM, 28:3:577--593, 1981.

[31] Oracle® Spatial Resource Description Framework (RDF) 10*g* Release 2 (10.2) Manual

[32] SWETO-DBLP http://lsdis.cs.uga.edu/projects/semdis/swetodblp/