GRIN: A Graph Based RDF Index

Octavian Udrea
University of Maryland
College Park, MD 20742

Andrea Pugliese
University of Calabria – DEIS department
Via P. Bucci, Rende, Italy

V.S. Subrahmanian
University of Maryland
College Park, MD 20742

Abstract

RDF ("Resource Description Framework") is now a widely used World Wide Web Consortium standard. However, methods to index large volumes of RDF data are still in their infancy. In this paper, we focus on providing a very lightweight indexing mechanism for certain kinds of RDF queries, namely graph-based queries where there is a need to traverse edges in the graph determined by an RDF database. Our approach uses the idea of drawing circles around selected "center" vertices in the graph where the circle would encompass those vertices in the graph that are within a given distance of the "center" vertex. We come up with methods of finding such "center" vertices and identifying the radius of the circles and then leverage this to build an index called GRIN. We compare GRIN with three existing RDF indexex: Jena, Sesame, and RDFBroker. We compared (i) the time to answer graph based queries, (ii) memory needed to store the index, and (iii) the time to build the index. GRIN outperforms Jena, Sesame and RDFBroker on all three measures for graph based queries (for other types of queries, it may be worth building one of these other indexes and using it), at the expense of using a larger amount of memory when answering queries.

Introduction

One of RDF's major strengths over the relational model of data is in *graphical queries*. For instance, users often want to propose a query pattern (think of this as a labeled graph with zero or more variables labeling either the vertices and/or the edges). The problem is to find "matches" for this pattern in the RDF graph. Past work on RDF indexing (Theoharis, Christophides, & Karvounarakis 2005) does not provide any index specialized to handle such queries. *This paper primarily focuses on providing a data/index structure that supports the efficient solution of such queries.* The principal contributions in this paper are the following.

1. The key problem in processing graphical queries is that we do not have any index supporting such graph matching. Clearly such an index structure must preserve the proximity of vertices from one another. We do this by proposing a way of identifying certain *center* vertices in a graph (think of these as vertices that occupy strategic positions in the graph) and by proposing a notion of *radius* from those center vertices. All vertices within the stated radius of a center vertex are associated with that vertex (each graph vertex must be associated with one or more center vertices). We propose methods of finding center and radius vertices.
2. We then define **GRIN** - an efficient tree data structure to store the regions defined by these center vertices (together with their associated radii).
3. Subsequently, we develop algorithms to answer graphical queries efficiently by using the **GRIN** data structure. The algorithms are proved correct, and their worst case computational complexity is stated.
4. Finally, we conduct a detailed series of experiments on the TAP[1] and the ChefMoz[2] data sets to determine the effectiveness of the **GRIN** index structure and the **GrinAnswer** algorithm. These are measured along three dimensions:
   - *How large is the size of the index?* We show that the **GRIN** index is smaller than that of Sesame (Broekstra, Kampman, & van Harmelen 2002), RDFBroker (Sintek & Kiesel 2006) and Jena. The use of circles provides a very compact representation of vertex neighborhoods.
   - *How long does it take to answer queries?* We show that **GrinAnswer** answers graph-based queries much faster than either Jena, Sesame or RDFBroker. We find experimentally that the determining factor in choosing GRIN over one of the other systems depends on the average degree of a node in the query graph.
   - *How long does it take to build the index?* We show that **GRIN** takes less time to build than Jena, Sesame and RDFBroker.

We use standard statistical tests (the t-test) to show that our experimental results are statistically significant.

RDF Graph Queries

This section provides a brief overview of RDF and graphical queries. Let $\mathcal{U}$ denote a set whose elements are called URI references, $\mathcal{L}$ denote a set whose elements are called literals, and $\mathcal{U}_p \subseteq \mathcal{U}$ denotes the set of *properties*. $\mathcal{R} = \mathcal{U} \cup \mathcal{L}$

[1] Available at `http://sp11.stanford.edu`.
[2] Available at `http://chefmoz.org/`.

denotes the set of *resources*. An *RDF triple* has the form (s, p, v)[3] where $s \in \mathcal{U}$, $p \in \mathcal{U}_p$, $v \in \mathcal{R}$. A set of RDF triples can be depicted in graphical form in a well-known way.

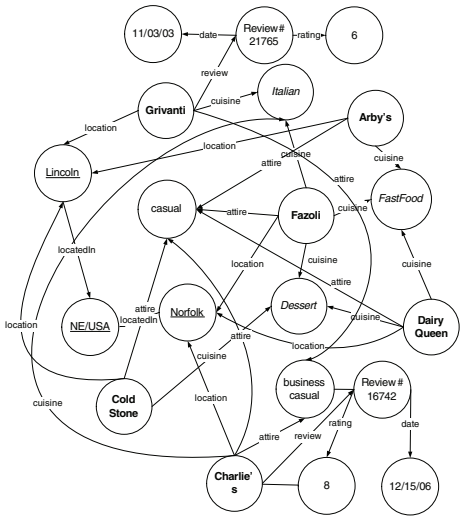

Figure 1: RDF graph example

Figure 1 contains an example graphical depiction of an RDF graph extracted from the ChefMoz dataset [4]. The RDF data contains six restaurants (**bold nodes**) in two locations in NE, USA (underlined nodes) for three different cuisine styles (*italicized nodes*); in addition, the data contains the type of attire required, as well as two restaurant reviews. We will use this RDF graph as a running example throughout the paper. We now define the concept of P-path.

Definition 1 (P-path) *Given an RDF graph D and a set $P \subseteq \mathcal{U}_p$, a* P-path *in D is a set $\{e_1, \ldots, e_q\}$, with $e_j = (s_j, p_j, v_j)$, such that*

- $\forall j \in [1, q]\ e_j \in D$;
- $\forall j \in [1, q-1]\ v_j = s_{j+1}$;
- $\forall j \in [1, q]\ p_j \in P$.

Intuitively, a P-path is a path in the RDF graph whose edge labels are all drawn from the set P.

Example 1 *Let P = {location, locatedIn}. The triples* (ColdStone, location, Lincoln) *and* (Lincoln, locatedIn, NE/USA) *constitute a P-path of length two in the graph in Figure 1.*

Our query language is an extension of the RDF query language SPARQL[5]. Specifically, we allow *path-at-most* predicates which can state that two resources are linked by a P-path with a given maximum length. For example, suppose we had an RDF-graph describing relationships and we want to find all people linked to John by a professional work relationship of length 3 or less. This is an example of a path-at-most relationship where P might be the set

[3]Special features of RDF such as blank nodes, reification and collections are not handled in this paper.

[4]Some URIs were shortened for readability.

[5]`http://www.w3.org/TR/rdf-sparql-query/`.

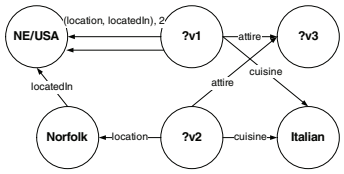

Figure 2: Graph query example

$\{boss, banker, lawyer\}$ saying that the relationships of interest are *boss, banker* and *lawyer*. We define an intuitive, graph-based representation for RDF queries.

Definition 2 (RDF graph query) *An* RDF graphical query *is a 4-tuple (N, V, E, λ_n) where:*

- *N is a set of vertices;*
- *V is a set of variables;*
- *$E = E_s \cup E_d$ is a set of edges, where $E_s \subseteq N \times N \times (V \cup \mathcal{U}_p)$ and $E_d \subseteq N \times N \times 2^{\mathcal{U}_p} \times \mathbb{N}$. We call E_s the set of* single *edges and E_d the set of* double *edges.*
- *$\lambda_n : N \rightarrow \mathcal{R} \cup V$ is a vertex labeling function.*

What this definition says is that in a graphical query, the structure of the graph is fixed. However, the vertices can be labeled either with variables or with values and likewise, the edges can be labeled either with variables or properties.

Example 2 *The query graph in Figure 2 informally says: find restaurants $?v1$, $?v2$ with the same attire $?v3$, such that both restaurants serve Italian food, $?v2$ is in Norfolk, NE, USA and $?v1$ has a {location, locatedIn}-path of length at most 2 to NE, USA.*

Note that, with the exception of P-path edges, the RDF graph queries are standard SPARQL queries[6]. Consider the query in Figure 2, without the double edge between $?v1$ and NE/USA. The query can be expressed in SPARQL as:

$$
\begin{array}{ll}
SELECT & ?v1\ ?v2\ ?v3 \\
WHERE & \{\{(?v1\ attire\ ?v3)\ .\ (?v1\ cuisine\ Italian)\} \\
 & \{(?v2\ attire\ ?v3)\ .\ (?v2\ cuisine\ Italian)\ . \\
 & (?v2\ location\ Norfolk)\} \\
 & \{(Norfolk\ locatedIn\ NE/USA)\}\}
\end{array}
$$

An answer to a query is a set of instantiations (or substitutions) for the variables in the query. Each element in the answer is a set of triples that forms a subgraph of the RDF graph D and matches the query graph q. This is formally defined below.

Definition 3 (Graph query answer) *The answer to an RDF graph query $q = (N, V, E, \lambda_n)$ w.r.t. a graph D, denoted $q(D)$, is a set of variable substitutions $\{\theta_1, \ldots, \theta_k\}$, with $\theta_i : V \rightarrow \mathcal{R}$, such that*

- *$\forall e = (n, n', x) \in E_s$, $\forall i \in [1, k]$, $(\lambda_n(n)\theta_i, x\theta_i, \lambda_n(n')\theta_i) \in D$;*
- *$\forall e = (n, n', P, l) \in E_d$, $\forall i \in [1, k]$, $\exists$ a P-path $\{e_1, \ldots, e_m\}$ in D, with $e_j = (s_j, p_j, v_j)$, such that $m \leq l$ and $\lambda_n(n)\theta_i = s_1$, $\lambda_n(n')\theta_i = v_m$.*

[6]SPARQL is the standard query language for RDF. A description can be found at `http://www.w3.org/TR/rdf-sparql-query/`.

The following example shows the answer to the query expressed in Example 2.

Example 3 *Consider the query in Example 2 w.r.t. the RDF graph in Figure 1. The possible substitutions are: ($?v1 \leftarrow$ Grivanti , $?v2 \leftarrow$ Charlie's, $?v3 \leftarrow$ businessCasual) and ($?v1 \leftarrow$ Fazoli, $?v2 \leftarrow$ Charlie's, $?v3 \leftarrow$ casual).*

The GRIN Index

For the rest of the paper, we assume that all inferences on *rdfs:subClassOf* and *rdfs:subPropertyOf* are made apriori in the RDF graph. Let $d : \mathcal{R} \times \mathcal{R} \rightarrow I\!N$ be a metric defined on the set of resources in the RDF graph. There are many such metrics (e.g., the minimum or maximum cycle-free path length between two resources in the RDF graph). For instance, the minimum distance between *Fazoli* and *NE/USA* is 2 in Figure 1. For simplicity, we assume in the rest of the paper that d is the minimum path length between two resources. We will now define the formal properties of the GRIN index structure.

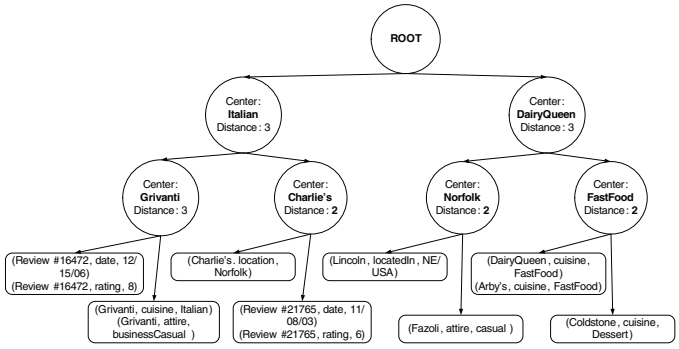

Figure 3: GRIN index example

Definition 4 (GRIN index) *A* GRIN index *is a* balanced *binary tree such that:*

- *Each leaf node ℓ contains a set $N_\ell \subseteq \mathcal{R}$ of nodes s.t. for all leaf nodes $\ell' \neq \ell$, $N_\ell \cap N_{\ell'} = \emptyset$, and $\cup_{\ell \in L} N_\ell = \mathcal{R}$;*
- *Each non-leaf node t contains a pair (c, r), with $c \in \mathcal{R}$ and $r \in I\!N$. Intuitively, this is a very succinct representation of the set of resources in the graph at distance at most r of the resource c according to the metric d. We write this set as $N_t = \{c' \in \mathcal{R} | d(c, c') \leq r\}$.*
- *For any nodes x, y in the tree such that x is a parent of y, $N_x \supseteq N_y$.*

The GRIN index is a binary tree. The set of leaf nodes in the tree form a partition of the set of triples in the RDF graph D. Interior nodes are constructed by finding a "center" triple, denoted c, and a radius value, denoted r. An interior node in the binary tree *implicitly* represents the set of all vertices in the RDF graph that are within r units of distance (i.e. less than or equal to r links) from the center.

Example 4 *Figure 3 shows an example GRIN index structure for the RDF graph in Figure 1. Note that the leaf nodes consist of clusters of resources – for this example, there is more than one possible GRIN structure, since membership of a resource to a cluster or another is often tied. The intermediate node* (Grivanti, 2) *signifies the set of resources in the graph with a minimum path less than or equal to two from the vertex* Grivanti.

Building the Index

Suppose M is the maximum number of RDF graph vertices per page. Since we want to build a balanced binary tree, the number of leaf nodes has to be a power of two. If we denote the number of leaf nodes by C, then we write $\frac{|\mathcal{R}|}{C} \leq M \Rightarrow C \geq \frac{|\mathcal{R}|}{M}$. The smallest C which is a power of 2 is $C = 2^{\lceil log_2 \frac{|\mathcal{R}|}{M} \rceil}$.

We use $d_c : 2^{\mathcal{R}} \times 2^{\mathcal{R}} \rightarrow I\!N$ to denote an arbitrary, but fixed, inter-cluster distance function based on the metric d. d_c takes two sets of resources and returns a numeric value. Three well-known inter-cluster distances are often used in clustering algorithms: (i) **Single link** defines $d_c(S, S') = \min_{x \in S, y \in S'}(d(x, y))$; (ii) **Complete link** defines $d_c(S, S') = \max_{x \in S, y \in S'}(d(x, y))$ and (iii) **Average link** defines $d_c(S, S') = \frac{\sum_{x \in S, y \in S'}(d(x,y))}{|S| \times |S'|}$.

We do not commit to any specific definition of d_c at this stage - however, we have found experimentally that average link provides the best query response time.

An algorithm to build the GRIN index is shown in Figure 4. The algorithm builds the index structure bottom up. Initially, we cluster the vertices in the graph into C disjoint sets using a modification of the *partitioning around medoids* (PAM) clustering algorithm (Kaufman & Rousseeuw 1987) (line 1). PAM starts by choosing C random vertices from the graph as cluster centroids. It then assigns all vertices in the graph to one cluster, based on their distances to the chosen cluster centroids. After C clusters have been formed, the centroids are re-computed and the process is repeated until an equilibrium is reached. We modified the original algorithm to ensure no cluster contains more than M vertices. Ties (cases when a vertex could be assigned to more than one cluster) are broken uniformly at random.

For each intermediate level in the tree, *GRINBuild* chooses a random node u from the available nodes and computes the "closest" node (denoted v) to u in terms of $d_c(N_u, N_v)$ (lines 7–8). u and v are then assigned a new parent node (c, r). The values of the center c and radius r are computed based on the set of vertices $N_u \cup N_v$ (lines 9–10). The process is repeated until we build the root of the tree, which corresponds to a set of resources encompassing the entire graph (loop condition on line 3).

We point out that the GRIN structure only *points* to the original RDF data, but it does not store it. Instead, we will represent D physically as a hash map in which each pair of resources c, c' is a key in the hash map associated with a list of properties $\{p_1, \ldots, p_k\}$. This means the RDF graph D contains the triples $(c, p_1, c'), \ldots, (c, p_k, c')$.

Complexity of building the GRIN Index. The set of vertices represented by a GRIN node is at most $|\mathcal{R}|$. For a level of the GRIN tree containing k nodes, the most time-

Algorithm GRINBuild(C, M, D)

Input: C is the number of leaf nodes, M is the maximum number of vertices on a page, D is the RDF theory.
Output: The GRIN index structure G.

1: $L_0 \leftarrow PAM(D, C, M)$
2: Create leaf nodes in G from clusters in L_0
3: **for** $i \in [1, \log_2 C - 1]$ **do**
4: $F \leftarrow L_i$
5: $L_{i+1} \leftarrow \emptyset$
6: **while** $F \neq \emptyset$ **do**
7: Pick a random node $u \in F$
8: Find $v \in F$ that minimizes $d_c(N_u, N_v)$
9: Compute centroid c and radius r for $N_u \cup N_v$
10: Create node $p = (c, r)$ in G as a parent of u and v
11: $L_{i+1} \leftarrow L_{i+1} \cup \{p\}$
12: $F \leftarrow F - \{u, v\}$
13: **end while**
14: **end for**
15: **return** G

Figure 4: An algorithm to build the GRIN index

consuming operation is the computation of inter-cluster distance, which can be done in parallel for the entire level in time $\mathcal{O}(|\mathcal{R}|^2 \cdot k^2)$. The number of leaf nodes C is $\mathcal{O}(|\mathcal{R}|)$ and the height of the tree is $\mathcal{O}(\log_2(|\mathcal{R}|))$. This leads to a worst case complexity for building the index of $\mathcal{O}(|\mathcal{R}|^4 \log_2(|\mathcal{R}|))$. However, we will show experimentally that building the index is generally much faster than the worst case.

Query Evaluation

In this section, we show how to evaluate a graphical query $q = (N, V, E, \lambda_n)$ against the GRIN structure. We start by showing how to derive a set of inequality constraints $cons(q)$ from the query. The constraints will be evaluated against the nodes of the GRIN index. This is done to identify the smallest subgraph that contains answers to q. We derive the following constraints:

- For any path of length l on edges from E_s connecting a resource c and a variable v, we write $d(c, v) \leq l$. The rule applies similarly for paths from v to c.
- For any edge $(c, v, P, l) \in E_d$, we write $d(\lambda_n(c), \lambda_n(v)) \leq l$. The rule applies similarly for edges from v to c.

Example 5 *Consider the example query in Figure 2. The query leads to the following set of constraints:* $d(?v1,$ *NE/USA*$) \leq 2$, $d(?v2,$ *NE/USA*$) \leq 2$, $d(?v2,$ *Norfolk*$) \leq 1)$, $d(?v1,$ *Norfolk*$) \leq 3$, $d(?v1,$ *Italian*$) \leq 1$, $d(?v2,$ *Italian*$) \leq 1$, $d(?v3,$ *NE/USA*$) \leq 3$, $d(?v3,$ *Norfolk*$) \leq 2$, $d(?v3,$ *Italian*$) \leq 2$.

We want to use the constraints generated from the query to identify nodes in the **GRIN** structure that could contain answers to the query. On any **GRIN** node, we have the option of *accepting* the node (which means it may contain answers to the query) or *rejecting* the node (which means it is guaranteed **not** to contain answers to the query). Consider a **GRIN** node corresponding to the circle (c, r). We will define two rules to decide whether (c, r) should be rejected.

The first rule is straightforward: *for any constant (resource) x in q,* ***reject*** (c, r) *if* $d(c, x) > z$ (R1). Intuitively, we are rejecting the circle represented by the **GRIN** node if any constant factors in the query are outside it.

Let's consider the case of a constraint $d(x, v) \leq l$ involving variable v and resource x. Since d is a metric, $d(c, v) \leq d(c, x) + d(x, v) \leq d(c, x) + l$. Note that $d(c, x)$ is a constant. If $d(c, x) + l \leq z$, we are sure that v is inside the circle (c, r). If this case, we say (c, r) *definitely satisfies* v. Also from the fact that d is a metric we can write $d(c, x) - l \leq d(c, x) - d(x, v) \leq d(c, v)$. If $z \leq d(r, c) - l$ then we are sure v is outside the circle (c, r). In this case, we say (c, r) *does not satisfy* v. If any variable is **not satisfied**, then we cannot find an answer to the query within (c, r).

We may also have situations in which neither of the two cases hold – we do not know for certain whether v is inside or outside the circle solely on the constraints derived from the query. We will take a conservative approach and only look for solutions in nodes that *definitely satisfy* **all** variables. This has the potential downside of stopping at subgraphs larger than necessary, but we have found experimentally that this policy is very effective. The second rule is: *if there exists $v \in V$ such that (c, r) does not* ***definitely satisfy*** *v, then* ***reject*** (c, r) (R2). Intuitively, we want to find the smallest sets of vertices that definitely satisfy all variables – or the equivalent, nodes lowest in the **GRIN** tree that satisfy all variables. Note that only one constraint per variable needs to be satisfied in order for the variable to be satisfied.

Example 6 *Consider the node* (*Grivanti,* 2) *in the index in Figure 3 and the constraint that says* $d(?v2,$ *Norfolk*$) \leq 1$. *This constraint is trivially* not satisfied*, since Norfolk is not in the circle specified by the node. However, the variable* $?v1$ *is satisfied from* $d($*Grivanti,*$?v1) \leq d($*Grivanti, Italian*$) + d(?v1,$ *Italian*$) \leq 2$*,*

Figure 5 contains the query evaluation algorithm. Given a query q and a node n_I of a GRIN index I, evaluates q over the subtree rooted in n_I. Answering a query over the theory D is equivalent to calling $GRINAnswer(q, root(I))$.

The algorithm uses the subgraph matching algorithm by Cordella et al. (Cordella *et al.* 2004). We chose this algorithm empirically because it generally yielded the best query answer times. If the invocation of *GRINAnswer* is currently at a leaf node, we will simply match the query graph with a subgraph of D containing the resources represented in n_I (lines 2–4). Otherwise, – if n_I is a potential candidate (line 5 checks (R1), (R2)), we will attempt a recursive call on the left-hand and right-hand children of n_I (line 6). Given the strategy of computing answers when all variables are *definitely satisfied*, we are guaranteed one of two outcomes.

(i) One of the recursive calls will return a non-empty answer. This implies that there exists a descendant (c, r) of n_I that passes both rules (R1) and (R2). In turn, this implies that all answers to the query are guaranteed to be inside the (c, r). All we need to do is return the answer found by subgraph matching while analyzing (c, r) (line 11).

(ii) We have not found an answer for any descendant, in which case we will attempt to run the subgraph matching on n_I itself and return the results (line 8–9).

Algorithm GRINAnswer(D, G, q, n_I)

Input: RDF theory D, GRIN index G and query $q = (N, V, E, \lambda_n)$, GRIN node n_I. $subgraphMatch$ is a subgraph matching method that finds an isomorphism between the query graph q and a graph H and returns a set of substitutions Θ for the variables in q.

Output: A set of answers Θ.

1: $\Theta \leftarrow 0$
2: **if** n_I is a leaf node **then**
3: $H \leftarrow$ the subgraph of D containing the resources in N_{n_I}
4: **return** $subgraphMatch(q,H)$
5: **else if** n_i is not rejected by checking rules (R1), (R2) against $cons(q)$ **then**
6: $\Theta \leftarrow GRINAnswer(D, G, q, n_I.left) \cup GRINAnswer(D, G, q, n_I.right)$
7: **if** $\Theta = \emptyset$ **then**
8: $H \leftarrow$ the subgraph of D containing the resources in N_{n_I}
9: **return** $subgraphMatch(q,H)$
10: **else**
11: **return** Θ
12: **end if**
13: **else**
14: **return** $\emptyset$
15: **end if**

Figure 5: An algorithm to answer queries over the GRIN index

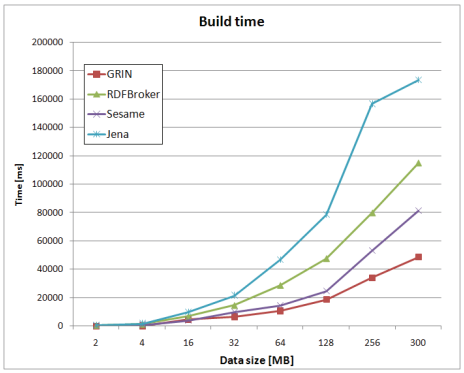

Figure 6: GRIN Index build time

Cordella et al. (Cordella *et al.* 2004) show the memory complexity of the subgraph matching algorithm to be $\Theta(N)$ (with a small constant factor), where N is the total number of vertices in the graphs to be matched, whereas time complexity ranges from $\mathcal{O}(N)$ in the best case to $\mathcal{O}(N!)$ in the worst case. The *GRINAnswer* algorithm therefore has a worst-time complexity of $\mathcal{O}(|\mathcal{R}|!)$. However, we have discovered in practice that *GRINAnswer* is able to identify very small subgraphs on which to match very efficiently, which greatly reduces the value of N. Our experimental results show that *GRINAnswer* is often faster than Jena, Sesame and RDFBroker for certain types of graph-based queries.

Example 7 *To better understand how query evaluation works, consider the query in Figure 2 without the node NE/USA. We start off at the root of the tree. We recursively call the evaluation for the nodes (Italian, 3) and (DairyQueen, 3). (DairyQueen, 3) can be quickly eliminated since we cannot determine whether* $?v1$ *and* $?v2$ *are definitely satisfied; this follows from* $d(DairyQueen, Italian) = 3$, $d(?v1, Italian) \leq 1$ *and the fact that the radius of the current circle is 3. On the other hand, (Italian, 3) definitely satisfies all variables, but its children do not. This means (Italian, 3) is the* ***GRIN*** *node the algorithm is looking for, namely the node that* definitely satisfies *all variables and is at the lowest possible level in the tree. When we match the query graph against the circle (Italian, 3), we obtain the two substitutions in Example 3, namely* $(?v1, ?v2, ?v3)$ = *(Grivanti, Charlie's, businessCasual) and* $(?v1, ?v2, ?v3)$ = *(Fazoli, Charlie's, casual).*

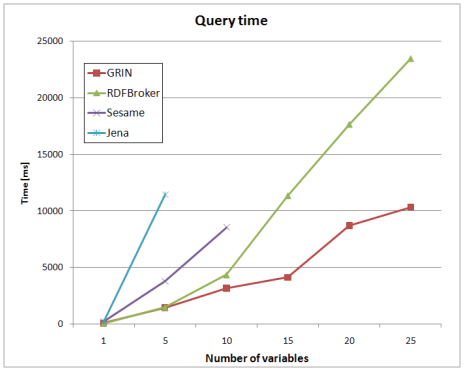

Figure 7: GRIN Query time

Experimental evaluation

We implemented **GRIN** in approximately 2450 lines of Java. We used the Jena API to parse the input RDF files, but did **not** use any other part of the Jena system. We evaluated GRIN on the set of RDF files from TAP and ChefMoz datasets. The TAP files range from approximately 1.5 MB up to 300 MB, with a number of triples from 17,086 to approximately 3.5 million. The ChefMoz data consists of three RDF files (about 220MB total) holding information about restaurant and dining guides from all over the world. We compare the implementation of GRIN to three other systems: RDFBroker (Sintek & Kiesel 2006), Sesame (Broekstra, Kampman, & van Harmelen 2002), and Jena (Wilkinson *et al.* 2003). All results are averaged over 5 independent runs.

Index creation time. We measured the index creation times on different file sizes. Figure 6 (note that the x-axis is in log-scale) shows that **GRIN** outperforms the other systems and builds the index for 300MB of RDF data in under 50 seconds. Two factors account for this performance: (i) **GRIN** does not store the data in the index, but points to it; the data is stored in a hash table representation and (ii) There is only one computationally intensive operation in building **GRIN**, namely the clustering at the leaf-node level. In each step, the rest of the algorithm finds the centroid vertices for a given subgraph and merges the circles in pairs based on the inter-circle distance.

Index size. **GRIN** is very efficient – it uses less than 395 MB for the 300MB data file. The index is stored in approximately 75 MB, the remaining 320MB is used for the hash

table representation of the actual data. RDFBroker, Sesame and Jena use 950 MB, 825 MB and 403 MB respectively.

Query time. We developed a set of 35 base queries for the TAP dataset, of which 25 were standard SPARQL queries[7] and 25 base queries for the ChefMoz dataset, of which 15 were standard SPARQL queries. The queries are typically very similar to the one in Figure 2, but larger. We then modified each base query by varying the number of variables as this is a good measure of the difficulty of a query. Figure 7 (which shows average time taken over these queries) shows that Jena and Sesame are unable to answer queries with over 10 variables (we stopped the run after 15 minutes). RDFBroker and **GRIN** tie as long as the number of variables is 5 or less, but **GRIN** beats RDFBroker when more than 5 variables are present. At this stage, we also evaluated the three inter-cluster distance metrics w.r.t. the query evaluation time. The index built with *average link* performed approximately 5% better than single link or complete link, which were comparable.

Related Work

Many RDF storage and querying systems rely on relational or object-relational DBMSs (Theoharis, Christophides, & Karvounarakis 2005). The majority of these systems associate relational tables with RDF triples, properties or class instances. Sesame (Broekstra, Kampman, & van Harmelen 2002) is designed to be independent from the specific back-end storage; however, the current implementation supports the creation of schema-specific tables if a schema is available, otherwise it employs class-specific tables. RDFSuite (Alexaki *et al.* 2001) also adopts a "schema-driven" storage technique; both systems rely on the underlying DBMS for query optimization. Coupling the table layout to the schema of RDF data allows to take into account the specific characteristics of employed classes and properties and to exploit the explicit schema relationships. On the other hand, limiting the number of created tables, especially when facing dynamic schema restructuring, becomes a crucial issue. A machine-learning based technique applied to data and queries to compute suitable relational table layouts is proposed in (Ding *et al.* 2003). Jena (Wilkinson *et al.* 2003) adopts instead a denormalized relational schema where long literals and URIs are compressed and stored in specific tables; moreover, it uses *property tables* that contain all subject-value pairs connected by a certain property. RDFBroker (Sintek & Kiesel 2006) computes resource *signatures*, i.e., the specific sets of properties used on each resource, and stores RDF data into lattice-organized tables corresponding to resource signatures. The Redland (Beckett 2002) system uses a hash-based technique that optimizes the search of RDF triples when two of their components are given. BRAHMS (Janik & Kochut 2005) specializes in searching for semantic association paths between resources.

[7]The graph patterns for the query did not contain any P-path double edges.

Comparison with Related Work

All the related work in RDF indexing mentioned in this section uses some form of relational representation for RDF triples that yields faster query answers than the standard relational representation of RDF in which all triples are stored in a three-column table. As a consequence, graph-based queries are translated into a series of relational join operations, which tend to be very time consuming for large queries. GRIN on the other hand introduces a radically new format for indexing triples that takes advantage of the layout of the RDF graph by grouping information around selected "center" nodes. We prove experimentally that the new structure has two major advantages over three of the best relational representations: (i) the index build time is considerably faster, since we do not have to build separate indexes for the subject, property and object part of a triple and (ii) query answers are faster as GRIN is able to quickly identify a small part of the RDF graph that contains the answer, thus avoiding time-consuming join operations.

Acknowledgements

Researchers funded in part by grant N6133906C0149, ARO grant DAAD190310202, AFOSR grants FA95500610405 and FA95500510298, and NSF grant 0540216.

References

Alexaki, S.; Christophides, V.; Karvounarakis, G.; Plexousakis, D.; and Tolle, K. 2001. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *International Workshop on the Semantic Web*.

Beckett, D. 2002. The Design and Implementation of the Redland RDF Application Framework. *Computer Networks* 39(5):577–588.

Broekstra, J.; Kampman, A.; and van Harmelen, F. 2002. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC*, 54–68.

Cordella, L. P.; Foggia, P.; Sansone, C.; and Vento, M. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. PAMI* 26(10):1367–1372.

Ding, L.; Wilkinson, K.; Sayers, C.; and Kuno, H. A. 2003. Application-Specific Schema Design for Storing Large RDF Datasets. In *International Workshop on Practical and Scalable Semantic Systems*.

Janik, M., and Kochut, K. 2005. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *ISWC*, 431–445.

Kaufman, L., and Rousseeuw, P. 1987. Clustering by Means of Medoids. In Dodge, Y., ed., *Statistical Data Analysis based on the L1 norm*. North Holland/Elsevier. 405–416.

Sintek, M., and Kiesel, M. 2006. RDFBroker: A Signature–Based High–Performance RDF Store. In *ESWC*, 363–377.

Theoharis, Y.; Christophides, V.; and Karvounarakis, G. 2005. Benchmarking Database Representations of RDF/S Stores. In *ISWC*, 685–701.

Wilkinson, K.; Sayers, C.; Kuno, H. A.; and Reynolds, D. 2003. Efficient RDF Storage and Retrieval in Jena2. In *Semantic Web and Databases Workshop*, 131–150.