Diamond: A SPARQL Query Engine, for Linked Data Based on the Rete Match

Daniel P. Miranker, Rodolfo K. Depena, Hyunjoon Jung, Juan F. Sequeda and Carlos Reyna [1]

Abstract. This paper describes a system, Diamond, which uses the Rete Match algorithm to evaluate SPARQL queries on distributed RDF data in the Linked Data model. In the Linked Data model, as a query is being evaluated, additional linked data can be identified as additional data to be evaluated by the query; the process may repeat indefinitely. Casting Linked Data query evaluation as a cyclic behavior enables making a constructive analogy with the behavior of a forward-chaining AI production system. The Rete match algorithm is the most commonly used implementation technique for AI production systems. Where AI production systems are known to make a relatively consistent number of changes to working memory per cycle, dereferencing URIs in the linked data model is a potentially volatile process. The paper provides an overview of Diamonds architectural elements that concern integrating the Rete match with the crawling of Linked Data and provides an overview of a set of Rete operators needed to implement SPARQL.

1 INTRODUCTION

Linked Data defines a set of best practices in order to treat data as a distributed interconnected graph just as the Web, through hyperlinks, has enabled documents to be interconnected and distributed [9]. In the Linked Data model directed, labeled graph edges, known as triples, are defined using RDF, the Resource Description Framework; a triple is comprised of a subject, a predicate and an object. Each component of a triple may be represented by a URI. By definition, the URI will be associated with an Internet server. The Linked Data principles stipulate that when a URI is dereferenced, the server should return a set of triples [2]. Those triples, in turn, may contain URIs for different servers. Thus, there is a potential for a triple on one server to logically connect one to three graph-edges, such that additional graph structured data may be gathered from distributed servers.

SPARQL is a query language for RDF graphs, and is itself commonly described as a language for describing graph patterns [12, 13]. Hartig et al. present an approach to execute SPARQL queries over Linked Data called *Link Traversal Based Query Execution* [7, 8]. In this approach, if a triple satisfies just one clause in a SPARQL query, then the connected components of that triple, linked by URI, may satisfy other clauses. Thus, in the course of evaluating a SPARQL query, if a triple satisfies a clause, each of its embedded URIs must be dereferenced. In other words, for each such URI, it may be necessary to go to a server and collect an additional set of triples.

Observe that a consequence of evaluating a SPARQL query over Linked Data may result in additional data being collected from over the Internet, and further evaluating the SPARQL query in a context that includes this new data. This can be viewed as an iterative process that may continue indefinitely [5, 6, 7, 8].

We observe a similarity in the execution of SPARQL queries over Linked Data queries and forward-chaining rule systems in Artificial Intelligence. In the latter a set of rule antecedents are evaluated against a repository of working memory. A satisfied rule is selected and its consequent executed. The consequent may insert or delete additional working memory elements. The rule sets antecedents are reevaluated. The cycle may proceed indefinitely. Thus, in the Linked Data model, dereferencing a URI and the additional triples fetched can be seen, operationally, as the same as firing a rule that adds elements to working memory.

Forgy's Rete match is the defacto standard for implementing forward-chaining rule systems[3]. Rather than reevaluating the rule antecedents at each cycle, the Rete Match processes incremental changes to the working memory as incremental changes to the set of satisfied rules. This is accomplished by interposing state operators, or memory nodes, in between a network of filtering operators. Incremental changes to working memory are processed as cascading incremental changes to the memory nodes. See Section 3.

2 DIAMOND ARCHITECTURE

The Diamond architecture is illustrated in Figure 1. The Rete network is created, on demand, as queries are entered by a user. The URI dereferencing object is static and not at all complicated. The most critical architectural component is the pair of queues, one for triples, and one for URIs, that connect the Rete network object with the URI dereferencing object, and their manager. These queues are intended to form the basis of parallel, asynchronous, execution of query evaluation and URI dereferencing.

Correctness of the Rete match algorithm requires a change initiated at the top of the network to be processed, depth-first to completion before beginning to process another change. Early in the execution of a query, little data will have been consumed, the network will be nearly empty, and processing will be fast. As data accumulates, one can anticipate query processing to slow, just as a query on a large database will take longer than the same query on a smaller database. Dereferencing a URI may yield an arbitrary number of additional triples. Some of those triples may not match any basic triple patterns in the query and the Rete network will dispatch them quickly. Other triples may propagate through the entire Rete network, and yield new solutions to the query. Even the number of solutions so produced is unpredictable. Thus, the queues act both as buffers, and the synchronization mechanism between two processes whose execution behavior is anticipated to be volatile.

[1] Department of Computer Science, University of Texas at Austin, email: {miranker, jsequeda}@cs.utexas.edu, email: {rudy.depena, polaris79, creynam89}@gmail.com

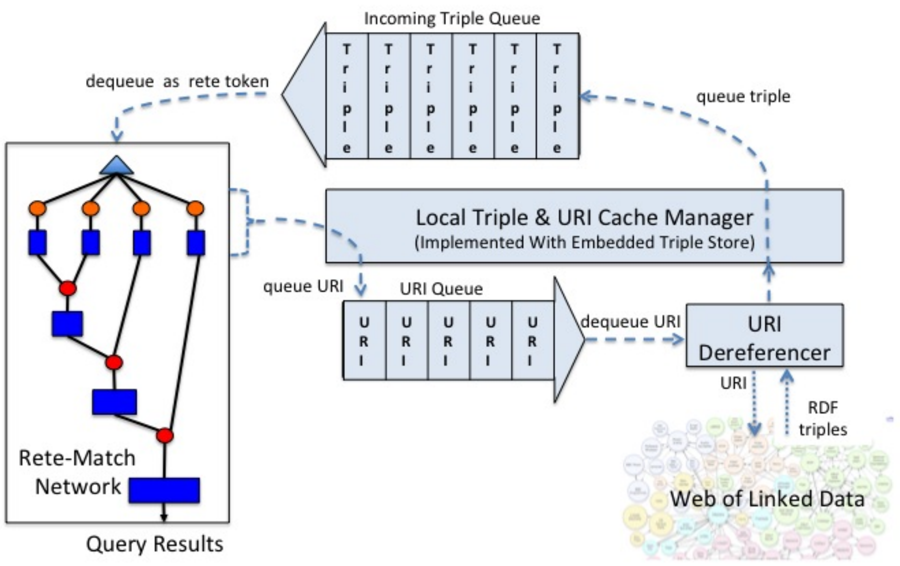
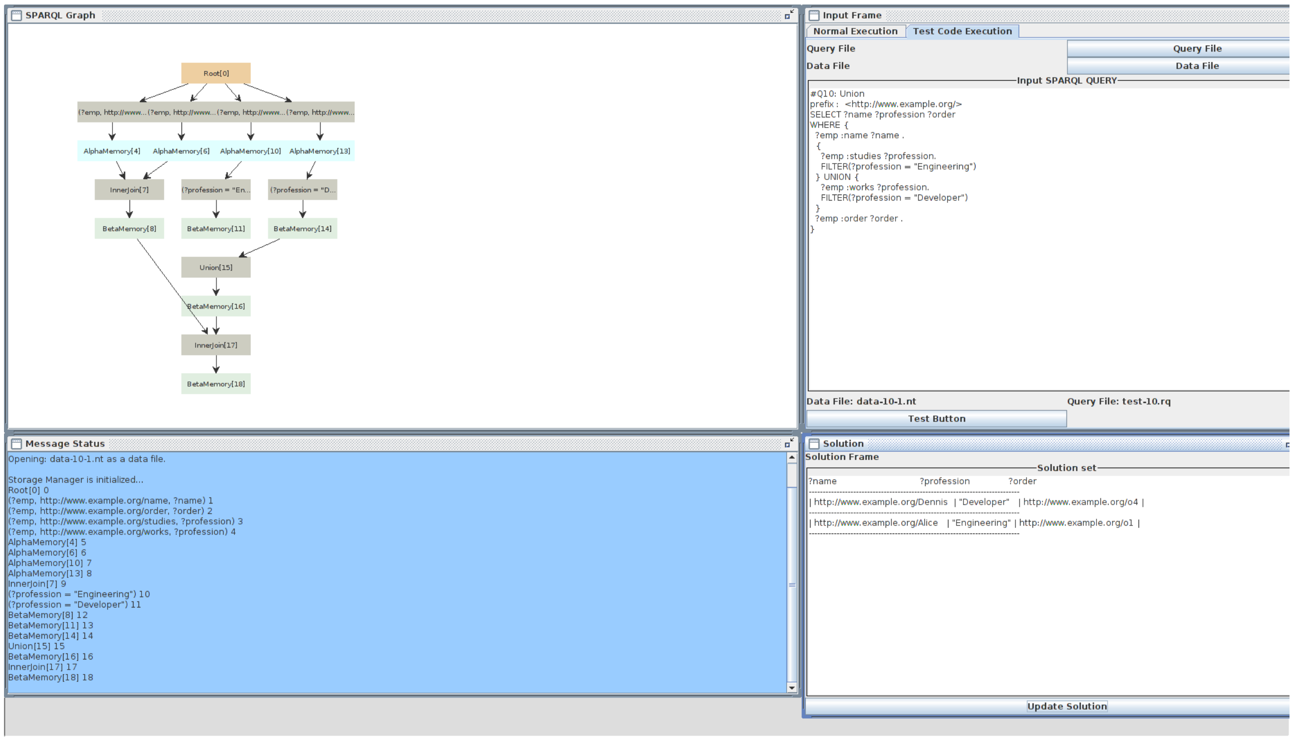


Figure 1. Diamond Architecture Diagram



```
#Q10: Union
prefix :  <http://www.example.org/>
SELECT ?name ?profession ?order
WHERE {
 ?emp :name ?name .
 {
  ?emp :studies ?profession.
  FILTER(?profession = "Engineering")
 } UNION {
  ?emp :works ?profession.
  FILTER(?profession = "Developer")
 }
 ?emp :order ?order .
}
```

Data File: data-10-1.nt   Query File: test-10.rq

```
Opening: data-10-1.nt as a data file.

Storage Manager is initialized...
Root[0] 0
(?emp, http://www.example.org/name, ?name) 1
(?emp, http://www.example.org/order, ?order) 2
(?emp, http://www.example.org/studies, ?profession) 3
(?emp, http://www.example.org/works, ?profession) 4
AlphaMemory[4] 5
AlphaMemory[6] 6
AlphaMemory[10] 7
AlphaMemory[13] 8
InnerJoin[7] 9
(?profession = "Engineering") 10
(?profession = "Developer") 11
BetaMemory[8] 12
BetaMemory[11] 13
BetaMemory[14] 14
Union[15] 15
BetaMemory[16] 16
InnerJoin[17] 17
BetaMemory[18] 18
```

```
?name                          ?profession          ?order
--------------------------------------------------------------
| http://www.example.org/Dennis  | "Developer"   | http://www.example.org/o4 |
--------------------------------------------------------------
| http://www.example.org/Alice   | "Engineering" | http://www.example.org/o1 |
--------------------------------------------------------------
```

Figure 2. SPARQL rule graphical debugger

The function, initially, of the triple & URI manager is to avoid redundant dereferencing of a URI and, similarly, redundant processing of triples through the Rete network. The manager is currently implemented using the Sesame[2] triplestore. The intrinsic depth-first processing requirement of the Rete match coupled with the intuition behind the Link Traversal model suggests treating the dereferencing queue as a stack (LIFO) is a natural, good fit. Or, the triple queue may be treated as a stack. If priority is given to both the most recent URI, and the ensuing triples, it is plausible that processing will focus strictly on the first few triple patterns of the query. It is easy to construct adversarial graphs such that no progress can be made in completing the full pattern of the query until traversals satisfying the first few triple patterns are exhausted.

As the research moves forward we may consider the use of priority queues ordered by means of heuristics concerning provenance or develop optimization strategies driven by statistical models based on historical behavior. In other words we may bias the system to fetch data from higher quality and/or historically faster data sources. The formal semantics of Linked Data queries is still being determined [6]. It is possible that open issues may be resolved in ways that limit operational behaviors. For example, if fairness properties are stipulated, deterministic methods that favor certain servers may induce starvation of other servers.

Implementing the triple & URI manager by using an embedded triple store adds flexibility per two additional open issues, one concerns speed, the other semantics. Motivated, in part, by *Big Data* computing, some linked data systems compute queries against a large local cache of precrawled data [11]. By committing, at the onset, to embedding a triple store in the cache manager, the evaluation of this approach can be compared to a strict crawling of live data while minimizing additional engineering and thereby maximizing the control of the experiments. The semantic issue concerns the Link Traversal Based Query Execution Model. Consistent with thath model, our first implementation only dereferences URIs of triples that have passed the initial Rete-Match filters and are recorded in a memory node. However, we promptly observed the following. If query solutions are subgraphs of disconnected graphs, following an initial set of links may yield only a subset of the possible solutions.

The system includes a SPARQL rule debugger typical of graphical debuggers in Rete-based rule execution systems shown in Figure 2.

3 RETE NETWORK

A Rete network is comprised of filter nodes, interleaved with memory nodes. To implement SPARQL using a Rete network is to map the pattern testing primitives of SPARQL to Rete operators. Beginning with the TREAT algorithm the connection between Rete operators and database operators has been exploited by many [10, 4] and we will do so here. First we illustrate, through an example, that pattern matching in SPARQL differs little from early, Rete-based AI rule languages, and then detail the corresponding Rete network and its behavior with respect to incremental addition of data; See Figures 3, 4, 5.

It is convenient to think of memory nodes as containing the solution to a subset of the query (rule antecedent). Memory nodes that store the output of a unary operator are called alpha memories. Memory nodes that store the output of binary operators are called beta memories. The last beta memory contains solutions to the query.

We explain the contents and a change of state for the example Rete network. Initialization for a particular query starts with dereferencing the URIs in the query. The <http://cs.utexas.edu/miranker> URI is dereferenced and a set of RDF triples are returned. One of those triples includes (<http://cs.utexas.edu/miranker> :knows <http://web.ing.puc.cl/arenas>). As illustrated in Figure 4, a Rete network is found in a state having processed some sample triples. The existence of the "Arenas URI" in the first alpha-memory suggests additional information may be found at http://web.ing.puc.cl server. For simplicity assume the "Arenas URI" is dereferenced and the return result includes the triple (<http://web.ing.puc.cl/arenas> :age "28"). This triple now satisfies the SPARQL graph pattern. Figure 5 illustrates the state of the network after processing this new triple that matches the one-input node on the right side and satisfies the condition for the two-input, or *Join* node. A quanta of information that moves through Rete network is commonly called a token. Figure 5. Although our implementation is in Java, one can think of an initial token being a pointer to the new triple, entering the root of the network, distributed across the two filter operators, passing the "age" filter. This success is recorded in the alpha memory. The token proceed to a join node which may identify a consistent binding for variable ?x. Each binding is emitted from the join node as a new token. A copy of the token recorded in a beta memory.

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

```
SELECT ?age
WHERE {
  <http://cs.utexas.edu/miranker> :knows ?x .
  ?x :age ?age.
}

(P example-rule
  (http://cs.utexas.edu/miranker :knows <x>)
      (<x> :age <age>)
-->
     (make <age>)
```

Figure 3. A SPARQL Query and Its Equivalent in OPS5 Syntax

A Rete operator set for SPARQL follows from work by Angles and Guitierrez concerning the semantics of SPARQL [1]. Angles and Gutierrez proved that SPARQL is equivalent in expressive power to non-recursive Datalog with negation. Given the relationship between SPARQL, Datalog and relational algebra, to form a Rete operator set for SPARQL one needs to identify a mapping, by transitivity, from SPARQL syntax and its native logic, to relational operators. Table 1 summarizes the preceding construction including the relationship between SPARQL constructs, SPARQL algebra, formal semantics, and Rete operators.

Each Rete-Match operator is implemented as a unique node in the network, with its object-oriented structure represented in Figure 5.

3.1 Root

The root node is the gateway for all data that enters the network. When a working memory change enters the network in the form of a tuple, it is wrapped in a structure called a token which includes a tag indicating whether the element is to be added (+ tag) or deleted (- tag) from memory. The root node then propagates the token to its direct children, all of them TriplePatternTest nodes.

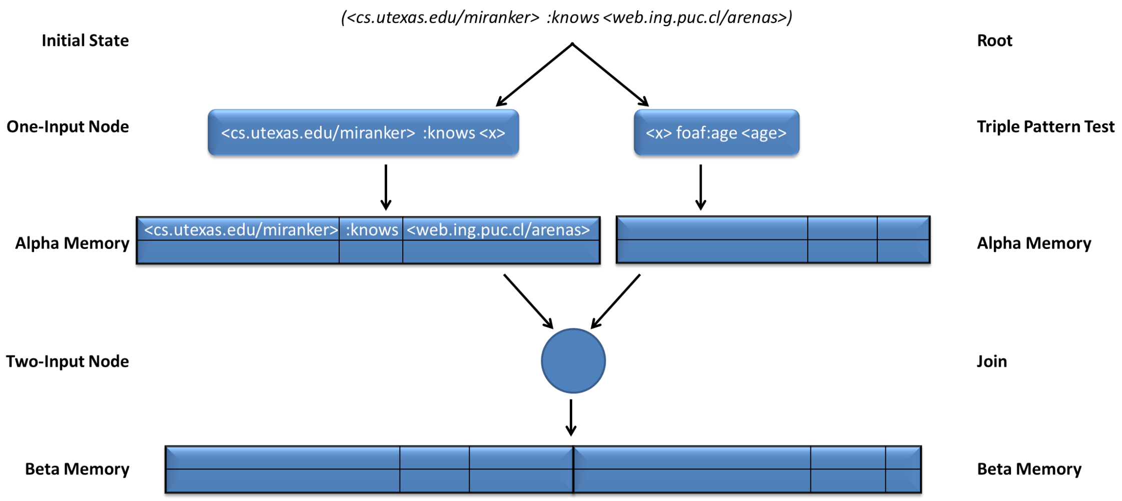


Figure 4. Rete network designed by Diamond for the SPARQL query from Figure 3; the memories have been previously filled with sample data.

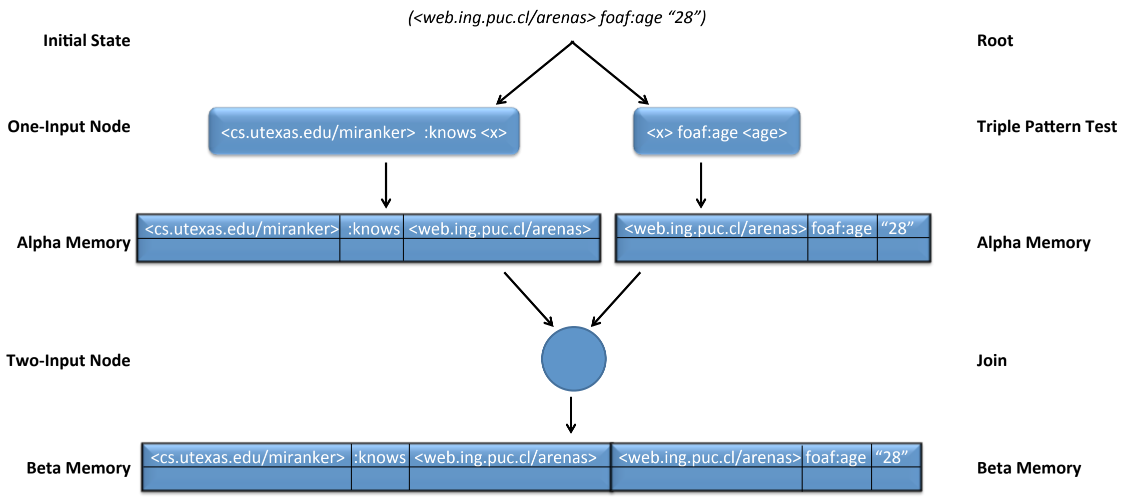


Figure 5. State of the Rete network after introducing the new triple. Labels on the left correspond to the terminology used in the original Rete match algorithm while labels on the right correspond to the names used on Diamond to convey the semantics of SPARQL.

Table 1. Summary of the Derivation of SPARQL Rete Operators

SPARQL Language Construct	SPARQL Algebra Operators	SPARQL Operators in [1]	Rete Operators
BGP	eval(D(G), BGP)	T(GroupGP)	TriplePatternTest(tp, R)
(.)	JOIN(Ω_1, Ω_2)	P1 AND P2	R1 InnerJoin R2
OPTIONAL	LEFTJOIN(Ω_1, Ω_2, C)	P1 OPT P2	R1 LeftJoin R2
UNION	UNION(Ω_1, Ω_2)	P1 UNION P2	R1 Union R2
FILTER	FILTER(C, Ω)	P1 FILTER C	Select(C,R)
SELECT	PROJECT(Ψ, PV)	SELECT	Project(s,R)

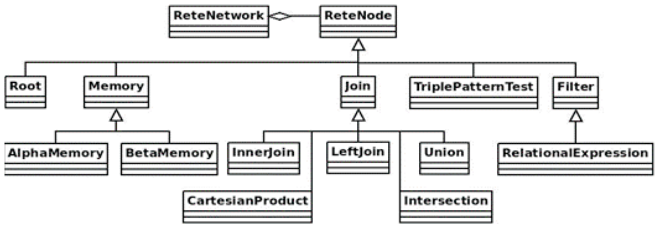


Figure 6. UML Class Diagram for the Rete-Mach implementation

3.2 TriplePatternTest

A unary test node that propagates only those tokens whose RDF tuple matches the constants in a single triple pattern; matching tokens are sent to the subsequent child alpha memory.

3.3 Memory

Alpha and Beta memories store the contents of + tokens they receive and delete tuples matching the contents of the - tokens.

3.4 Inner Join

All Join operators have two memories as their parents. The inner join operator is activated by receiving a token from either the left or right side.

The token will be compared with every token contained within the opposing memory, and two tokens are joined together if their variable bindings are consistent or no shared variables exist in either token. The resulting joined token will then be propagated to the succeeding Beta memory for storage.

3.5 Left Join

Behaves like an Inner Join, but with a couple of special cases:

If a token from the left parent does not match a token from the right hand side, then the left hand side token will propagate to the child Beta memory without being joined; else the tokens are joined as usual.

If a token comes from the right hand side parent, it is compared with the left side, and for each match a token with the joined tuples is propagated alongside with a negated token containing the matching left side tuple.

3.6 Union

A binary operator that computes the set union between the two memories, it propagates all + tokens that are not already contained in the succeeding child memory and only those - tokens that match a tuple in that same child memory.

3.7 Intersection

A binary operator that computes the set intersection between the two memories. It propagates only those + tokens that match a tuple in the opposing side memory and are not already contained in the succeeding child memory. It also propagates only those - tokens that match a tuple in the succeeding child memory.

3.8 Cartesian Product

Whenever a token reaches one side of this binary node, a new token with the same tag is created and propagated to the subsequent child Beta memory for each element of the other side memory containing the joined tuples.

3.9 Filter

Compares the contents of each + and - tokens to a local constraint, if there is a match the token will propagate to the subsequent Beta memory, else it stops.

4 STATUS and PRELIMINARY RESULTS

To minimize the learning curve of future project participants Diamond is constructed using canonical Java compiler tools. The lexer and parser are build using a BNF grammar description of SPARQL and JavaCC4. Java Tree Builder6 (JTB) is the basis of internal syntax tree. At runtime, a SPARQL query is parsed into internal form, and the Rete network created by traversing the syntax tree and calling the Rete network object constructors.

The system is not yet multithreaded. In other words there is no parallelism among the query processing and the URI dereferencing objects. The two queues are both implemented as standard FIFO queues.

The system successfully passes all 126 SELECT queries in the RDF API for PHP test suite (RAP). The suite contains another 25 test cases consisting of SPARQL, ASK, DESCRIBE, CONSTRUCT and solution modifier queries, which are currently not supported in Diamond.

For our preliminary experiments, we used the Berlin SPARQL Benchmark. We execute every experiment ten times (excluding three warm-up runs) and calculate the average query execution speed. See Table 2. Although the test data source was hosted on a machine on the local network, without multithreading, the execution times include the total network latency for dereference every URI in the query.

Table 2. Timing Results from Linked Data queries

Query	Query 1	Query 2	Query 3
# of Results	10	1	4
# of Tokens Processed	10,246	10,106	64
Execution Time	5 min, 47 sec	3 min, 43 sec	2.12 sec

5 FUTURE WORK

This paper presents the architecture of Diamond, a SPARQL query engine for Linked Data based on the Rete Match. The system has only just recently become functional. The system architecture anticipates future development and evaluation of optimization with respect to prioritizing dereferencing URIs, the processing of the resulting triples, and local caching of linked data. Future evaluation will necessarily include comparison with other Linked Data query systems, most notably SQUIN [3]. We made observation in several places in this paper that there are open implementation choices still to be researched that may be influenced or even disallowed, depending on how the semantics of these systems are resolved.

ACKNOWLEDGEMENTS

This research was supported by NSF grant IIS-1018554. Juan F. Sequeda was supported by the NSF Graduate Research Fellowship.

REFERENCES

[1] Renzo Angles and Claudio Gutierrez, 'The expressive power of sparql', in *International Semantic Web Conference*, pp. 114–129, (2008).

[2] Tim Berners-Lee. Linked data - design issues. `http://www.w3.org/DesignIssues/LinkedData.html`.

[3] Charles Forgy, 'Rete: A fast algorithm for the many patterns/many objects match problem', *Artif. Intell.*, **19**(1), 17–37, (1982).

[4] Eric N. Hanson, 'The design and implementation of the ariel active database rule system', *IEEE Trans. Knowl. Data Eng.*, **8**(1), 157–172, (1996).

[5] Olaf Hartig, 'Zero-knowledge query planning for an iterator implementation of link traversal based query execution', in *ESWC (1)*, pp. 154–169, (2011).

[6] Olaf Hartig, 'Sparql for a web of linked data: Semantics and computability', in *ESWC*, pp. 8–23, (2012).

[7] Olaf Hartig, Christian Bizer, and Johann Christoph Freytag, 'Executing sparql queries over the web of linked data', in *International Semantic Web Conference*, pp. 293–309, (2009).

[8] Olaf Hartig and Johann-Christoph Freytag, 'Foundations of traversal based query execution over linked data', in *HT*, pp. 43–52, (2012).

[9] Tom Heath and Christian Bizer, *Linked Data: Evolving the Web into a Global Data Space*, Synthesis Lectures on the Semantic Web, Morgan & Claypool Publishers, 2011.

[10] Daniel P. Miranker, 'Treat: A better match algorithm for ai production system matching', in *AAAI*, pp. 42–47, (1987).

[11] Eyal Oren, Renaud Delbru, Michele Catasta, Richard Cyganiak, Holger Stenzhorn, and Giovanni Tummarello, 'Sindice.com: a document-oriented lookup index for open linked data', *IJMSO*, **3**(1), 37–52, (2008).

[12] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez, 'Semantics and complexity of SPARQL', *ACM Trans. Database Syst.*, **34**(3), (2009).

[13] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Recommendation 15 January 2008, `http://www.w3.org/TR/rdf-sparql-query/`.

[3] http://squin.org