Optimizing a Rete-based Inference Engine using a Hybrid Heuristic and Pyramid based Indexes on Ontological Data

Tuğba Özacar, Övünç Öztürk, Murat Osman Ünalır
Department of Computer Engineering,
Ege University
Bornova, 35100, Izmir, Turkey
Email: {tugba.ozacar,ovunc.ozturk,murat.osman.unalir}@ege.edu.tr

Abstract— **This paper describes indexing of ontological data to reduce the memory consumption of a Rete-based reasoner whose time performance is increased using a hybrid optimization heuristic. The aforementioned indexing mechanism is known as the Pyramid Technique. Our work organizes three dimensional ontological data in a way that works efficiently with this indexing mechanism and it constructs a subset of the querying scheme of the Pyramid Technique that supports querying ontological data. This work also implements an optimization on the Pyramid Technique. Finally, it represents the progress in the memory consumption of the reasoner.**

Index Terms— **scalability, reasoning, ontology, pyramid technique, optimization heuristic**

I. Introduction

Handling large and combined ontologies is essential for many semantic web tools including reasoners. Although there are a number of rule based reasoners, that can manage medium sized ontologies with fairly good performances, it is hard to say the same for large ontologies [1]. Since reasoning is a time and memory consuming process, many of the reasoners can not manage with large ontologies. Especially in the case of dynamically updated data, we use algorithms that are immune to changes, such as Rete.

Although Rete is an optimized forward chaining algorithm, it lacks memory while dealing with large ontologies. This work introduces a solution for reducing memory consumption of an optimized Rete reasoner by adapting an indexing method to ontological data. This method is known as the Pyramid Technique [2]. Next section represents the Rete algorithm in detail. Section three introduces the hybrid optimization that we implement on the reasoner. Section four describes mapping Semantic Web resources to numerical values in order to support the input format required by the Pyramid Technique. Section five represents adaptation of the Pyramid Technique for indexing three dimensional ontological data. This section also has two subsections. The first one represents a subset of the querying scheme of the Pyramid Technique that supports querying ontological data. The second one describes an optimization on the Pyramid Technique in order to get better query response times. Section six represents the effect of the hybrid optimization and compares the memory consumptions and query performances of our reasoner with and without implementation of the Pyramid Technique. Finally section seven concludes this paper with an outline of some potential future research.

II. Rete Algorithm

Rete [3] [4] is an optimized forward chaining algorithm. An inefficient forward chaining algorithm applies rules for finding new facts and whenever a new fact is added to or removed from ontology, algorithm starts again to produce facts that are mostly same as the facts produced in the previous cycle. Rete is an optimized algorithm that remembers the previously found results and does not compute them again. Rete only tests the newly added or deleted facts against rules and increases the performance dramatically. Rete algorithm is based on the reasoning process of Rete network. Following definitions give a formal representation of the concepts in a Rete network.

Let $\mathcal{O} = (\mathcal{W}, \mathcal{R})$ be an ontology where $\mathcal{W} = \{w \mid w = (s,p,o) \wedge s,p,o \in \mathcal{U}\}$ is the set of all facts in the ontology and $\mathcal{R}$ is the set of all rules related with the ontology, then every fact $w \in \mathcal{W}$ consists of a subject s, a predicate p and an object o and $\mathcal{U}$ denotes the set of constants. Given a $r \in \mathcal{R}$, $r = (lhs, rhs)$ where both lhs and rhs are lists of atoms. An atom $at = (a,i,v)$ consists of an attribute a, an identifier i and a variable v where a,i,$v \in \mathcal{T}$ and $\mathcal{T} = \mathcal{U} \cup \mathcal{V}$ where $\mathcal{V}$ denotes the set of variables. A lhs atom is called a condition. $\mathcal{C}$ is the set of all conditions of all rules $r \in \mathcal{R}$. Given a Rete network of the ontology $\Omega(\mathcal{O}) = (\alpha, \beta)$, we denote by α the alpha network and by β the beta network. $\alpha = \{\delta(c) \mid c \in \mathcal{C}\}$ and $\delta : \mathcal{C} \rightarrow \mathcal{D}$ is a function where $\mathcal{D} = \{x \mid x \subseteq \mathcal{W}\}$. $\delta(c)$ returns the set $\{w \mid w = (s,p,o) \wedge c = (a,i,v) \wedge ((s = a) \vee (a \in \mathcal{V})) \wedge ((p = i) \vee (i \in \mathcal{V})) \wedge ((o = v) \vee (v \in \mathcal{V}))\}$ denoting all matching facts with condition c. β network consists of beta memories and join nodes where beta memories store partial instantiations of rules, which are called tokens, and join nodes perform tests for consistency of variable bindings between conditions of rules. Figure

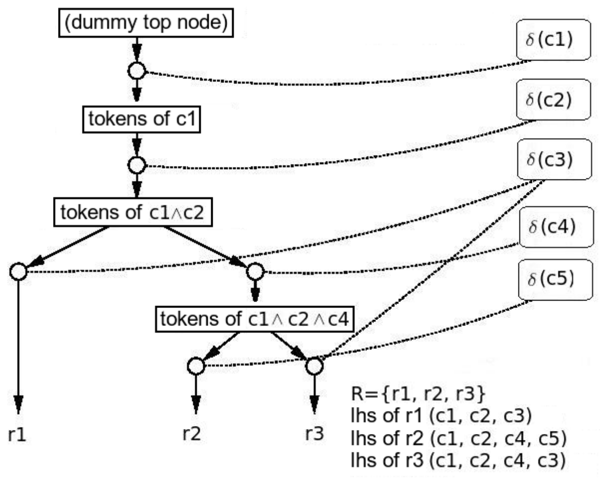

Figure 1. An example RETE Network.

1 explains the relationships among the formal definitions with the help of an illustration existing in [4]. In this figure rectangles show beta memory nodes, ovals show alpha memories and circles show join nodes.

The set of all beta memories is $\mathcal{PI} = \{\phi(s) \mid s \in \mathcal{S}\}$ where $\mathcal{S} = \{x \mid x = (c_1 \wedge ... \wedge c_n) \wedge r = (lhs, rhs) \wedge lhs = (c_1, ..., c_t) \wedge (1 \leq n \leq t) \wedge r \in \mathcal{R}\}$ and $\phi : \mathcal{S} \rightarrow \mathcal{I}$ is a function where $\mathcal{I} = \{x \mid x$ is a conjunctive set of $w \in \mathcal{W}\}$. $\phi(s)$ returns the conjunctive sets of facts that are matching with s. The instantiations at the end of the Rete network are handled as production nodes, abbreviated as p-nodes. $\mathcal{P} = \{\phi(s) \mid s = (c_1 \wedge ... \wedge c_t) \wedge r = (lhs, rhs) \wedge lhs = (c_1, ..., c_t) \wedge r \in \mathcal{R}\}$. Whenever a propagation reaches the end of the Rete network in other words a p-node gets activated, it indicates that a rule's conditions are completely matched and the right handside atoms of the rule produces a new fact that will be added to the ontology. Adding a new fact to the ontology triggers Rete network and new facts are inferred by these newly added facts without recomputing the previously found facts.

Our Rete based reasoner uses the syntax and semantics described in OWL Rules Language for rules, which is a special case of axiom-based approach [5]. The reasoner is optimized with a hybrid usage of some heuristics, which are introduced in the following section.

III. Heuristics and Their Hybrid Usage

Although Rete is an optimized forward chaining algorithm, we use a hybrid heuristic to gain extra performance. This hybrid heuristic [6] optimizes Rete algorithm by mixing and modifying some well known optimization heuristics [7]. This section introduces these heuristics and their hybrid usage.

Heuristic 1: Place Restrictive Conditions First

This optimization reduces the intermediate data by joining restrictive conditions first. Rete tests a rule against ontology triples finding all partial instantiations of the rule. Given an ordered list $\mathcal{L}$ $((c_1), (c_1 \wedge c_2), ..., (c_1 \wedge ... \wedge c_{n-1}), (c_1 \wedge ... c_n))$ where left hand side of the rule r is a set $\{c_1, ..., c_n\}$, all partial instantiations of r is a set $\mathcal{K} = \{k_1, ..., k_n\}$ and k_x is the set of matches for the x_{th} element of $\mathcal{L}$. Let k_n be the n_{th} element of $\mathcal{K}$ and $\mathcal{E}(k_n)$ be the size of k_n then;

$\mathcal{E}(k_0)$=1

$\mathcal{E}(k_n) \subseteq \mathcal{E}(k_{n-1})$ X $\delta(c_n)$

Thus ordering conditions with minimum alpha memory(restrictive) first will also decreases the size of the following instantiations. The following three methods are used in order to find the more restrictive conditions:

- *Method 1:* This method sorts the conditions ascending based on the number of edges matching with the condition. It finds the condition with minimum alpha memory m at time t, but it does not guarantee m will be the conditon with minimum alpha memory after a series of addition and deletion.
- *Method 2:* This heuristic assumes the conditions with more variables have alpha memories big in size so it sorts the conditions ascending based on the number of variables.
- *Method 3:* The usage of complex predicates is at minimum in semantic web ontologies [8]. Ontologies mainly consist of facts describing subsumption and assertion relations. Besides, the subsumption predicates in the ontologies give rise to cyclic repetitive[1] calculation and the number of facts with these predicates gets even more at the end of the inference [9]. This heuristic assumes that the number of alpha memories for conditions with complex predicates will be much smaller than the conditions with assertion or subsumption predicates. Thus the conditions with frequently used predicates are placed at the end of the conditions of the rule. This heuristic also improves the performance of additions and removals. The conditions with a frequently used predicate have more possibility to change and if one of these conditions is at the end of n conditions then whenever it matches with a newly created(or deleted) fact, only one join operation would be invoked. If this condition was the first then $n-1$ join operations would be invoked subsequently [7].

Heuristic 2: Order Conditions with Common Variables Sequentially

This optimization reduces the intermediate data by ordering conditions having common variables sequentially. If the n^{th} condition of the rule has a common variable x with $n-1^{th}$ condition then $\mathcal{E}(k_n)$ gets smaller because of the restrictions have been made on x at $n-1^{th}$ instantiation.

The Hybrid Heuristic

These two heuristics can be used in a hybrid way without conflicting each other while preserving the claim that the

[1]In this work only the subsumption relationship is handled, in addition to subsumption the other transitive properties giving rise to cyclic repetitive calculation can be handled as frequently used predicates.

performance of the hybrid usage will generally be better than the performance of using every heuristic separately. The heuristic works in the following order, where r is the rule to be optimized, $\mathcal{C}\ (r)$ is the ordered list of all conditions of r, r' is the result of the optimization and l is the last element of $\mathcal{C}\ (r')$:

```
Step 1:
  • $\mathcal{C}\ (r') \leftarrow$ null
  • Find the most restrictive condition[2] $x$ in $\mathcal{C}\ (r)$, remove $x$ from $\mathcal{C}\ (r)$ and append $x$ to $\mathcal{C}\ (r')$
Step 2:
  • if $\mathcal{C}\ (r) \neq \emptyset$
    – Find $x \in \mathcal{C}\ (r)$ and $x$ is the most restrictive condition having maximum number of common variables with $l$, remove $x$ from $\mathcal{C}\ (r)$ and append $x$ to $\mathcal{C}\ (r')$
    – Step 2
  • else
    – return $r'$
```

Determination of the most restrictive condition differs for the rules and the queries. Although *Method 1* guarantees to find the facts with minimum alpha memory in the ontology while optimizing queries, it is useless for optimizing rules. Because the Rete network has not been created, the number of matching edges with a condition can not be obtained during the optimization of the rules. To determine the most restrictive condition in the rules two methods are used according to the following priorities: *Method 2*, *Method 3*. This means that the first condition with a complex predicate among the conditions having minimum number of variables is the most restrictive one. *Method 2* has a higher priority than *Method 3* because there are a great number of conditions in rules having three variables and returning all of the edges in the ontology. These conditions have the biggest alpha memory and Method 2 guarantees to place them at the end of the conditions.

In addition to using the hybrid heuristic, we also used indexes on ontological data to improve the performance. Ontological data consists of facts where each fact is a triple containing a subject, a predicate and an object. Queries on ontological data are based on these three parts of the triple, in other words ontological data has three dimensions. Our analysis proved that the indexes, that we created on these dimensions, in order to reduce time consumption, make up a big percentage of the memory consumption. In order to reduce memory consumption by sacrificing a reasonable amount of performance, we decided to implement a new indexing mechanism that is known as the Pyramid Technique.

IV. Mapping Semantic Web Resources to Numerical Values

The Pyramid Technique indexes d-dimensional data using d-dimensional points as input. Ontological data has three dimensions (these dimensions are *subject*, *predicate* and *object*) where every dimension is a Semantic Web resource represented by an URI[3]. Thus, it is required to map the three dimensional ontological data to a three dimensional point in order to support the input format required by the Pyramid Technique. Originally the Pyramid Technique has numerical dimension values between 0-1 but we mapped every URI to a 16-digit long number because of the inadequacy when dealing with rational numbers. The cause of this inadequacy is that in computational environment rational numbers are represented using a floating point encoding with limited precision smaller than 16.

TABLE I.
Example URIs.

No	URI
0	http://www.w3.org/wine#Wine
1	http://www.w3.org/wine#Winery
2	http://www.w3.org/wine#madeFromGrape
3	http://www.w3.org/food#PotableLiquid
4	http://www.w3.org/food#Grape
5	http://www.w3.org/owl#Class

We slightly modified the mapping scheme in [10], in needs of semantic web resources. The mapping mechanism will assign a unique 16-digit long number for every semantic web resource residing in the ontology. The first six digits of this number represent the namespace and the remaining ten digits represent the reference name. In other words, mapping scheme supports an ontology having namespaces up to 10^6, and reference names up to 10^{10}.

In this mapping algorithm, every namespace is numbered sequentially from 0 to 10^6 and every reference name is numbered sequentially from 0 to 10^{10}. Given an URI to be mapped u, let n be the number assigned to the namespace of u, r be the number assigned to the reference name of u and x be the 16-digit long number that is the result of the mapping procedure, then $x = (n \times 10^{10}) + r$.

The following example maps the example URIs[4] in Table I to numerical resources, where x_i is the number assigned to the i_{th} URI, n_i is the number assigned to the namespace of the URI and r_i is the number assigned to the reference name of the URI;

$$\begin{array}{ll} x_0 = (0 \times 10^{10}) + 0 & x_1 = (0 \times 10^{10}) + 1 \\ x_2 = (0 \times 10^{10}) + 2 & x_3 = (1 \times 10^{10}) + 3 \\ x_4 = (1 \times 10^{10}) + 4 & x_5 = (2 \times 10^{10}) + 5 \end{array}$$

Mapping semantic web resources to long values comes with an additional benefit beyond supporting the input format required by the Pyramid Technique. During the reasoning process we deal with long values instead of

[2]If the result contains more than one condition pick the first one

[3]Anonymous nodes and literals in ontologies are handled by assigning unique numbers to each of them programmatically.

[4]Fragment identifier (#) seperates the namespace and the reference name of a URI.

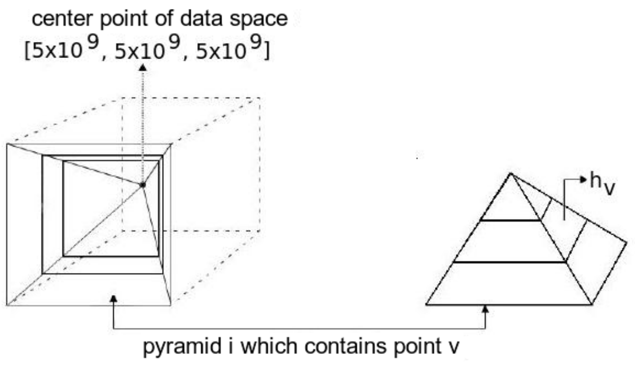

Figure 2. Indexing a 3-dimensional ontological data using The Pyramid Technique.

strings. Comparison and evaluation of long values are not as time and memory consuming as strings. Since there are huge numbers of comparison and evaluation of semantic web resources, representing these resources as long values makes a significant impact on time and memory performance of the reasoner.

V. Adapting the Pyramid Technique for Indexing Ontological Data

The basic idea of the Pyramid Technique is to transform the d-dimensional data points into one dimensional values and then store and access the values using an efficient index structure such as the $B^+ - tree$. In our case, this technique transforms a three dimensional point into a one dimensional value. The Pyramid Technique partitions the data space into 2d (in our case 6) pyramids having the center point of the data space as their top. Then each of six pyramids is divided into several partitions each corresponding to one data page of the $B^+ - tree$. The Pyramid Technique transforms a three dimensional ontological data into a one dimensional value $(i + h_v)$ where i is the index of the according pyramid p_i and h_v is the height of v within p_i (Figure 2).

Whenever we need to access ontological data, we first compute the pyramid value and query the $B^+ - tree$ using this value as a key. The resulting data pages of the $B^+ - tree$ contain points which constitute the answer of the query. Thus it is necessary to sequentially search these data pages in order to find the exact points corresponding to the query answer. The following formulas calculates the pyramid value and the height of a 3-dimensional point v:

Pyramid value(i) of v:

$$i = \begin{cases} j_{max} & \text{if } (v_{j_{max}} < 0.5) \\ (j_{max} + 3) & \text{if } (v_{j_{max}} \leq 0.5) \end{cases}$$

, where
$j_{max} = (j | (\forall k, 0 \leq (j,k) < 3, j \neq k : |0.5 - v_j| \geq |0.5 - v_k|))$

Height(h_v) of v:

$h_v = |0.5 - v_{iMOD3}|$

TABLE II.
All possible forms of range queries.

Ontological Query	Transformed Range Query
$*, *, *$	$[0, 10^{10}][0, 10^{10}][0, 10^{10}]$
$pv_s, *, *$	$[pv_s, pv_s][0, 10^{10}][0, 10^{10}]$
$*, pv_p, *$	$[0, 10^{10}][pv_p, pv_p][0, 10^{10}]$
$*, *, pv_o$	$[0, 10^{10}][0, 10^{10}][pv_o, pv_o]$
$pv_s, pv_p, *$	$[pv_s, pv_s][pv_p, pv_p][0, 10^{10}]$
$pv_s, *, pv_o$	$[pv_s, pv_s][0, 10^{10}][pv_o, pv_o]$
$*, pv_p, pv_o$	$[0, 10^{10}][pv_p, pv_p][pv_o, pv_o]$

The Pyramid Technique is an index structure for managing high-dimensional data. Our motivation to choose this index mechanism for indexing relatively low dimensional data (three dimensions) is to support potential index requirements. The Pyramid Technique allows to increment the dimension size in a flexible way and without sacrificing the performance.

A. Querying Indexed Ontological Data

Transformed ontological queries construct a subset of the queries described in the Pyramid Technique. The queries of the Pyramid Technique consist of point queries and range queries that are a set of d-dimensional intervals represented by r where $r = [q0_{min}, q0_{max}], \ldots, [q_{d-1_{min}}, q_{d-1_{max}}]$. Ontological queries consists of point queries and a subset of the range queries which is also a set of d-dimensional intervals represented by r' where $r' = [q0_{min}, q0_{max}], \ldots, [q_{d-1_{min}}, q_{d-1_{max}}]$ and $((q_{x_{min}} = q_{x_{max}})$ or $(q_{x_{min}} = 0$ and $q_{x_{max}} = 10^{10}))$ and $0 \leq x \leq d-1$.

The two types of the ontological queries are defined below and all possible cases of range queries on ontological data with their transformations are presented in Table II :

- *Point Queries:* The query answer is a simple yes/no. This type of queries is used in searching or editing ontological data. In this case, a point is given and the answer specifies whether the given point is in the data space. Every dimension of the point is specified in the query. This problem is solved by first computing the pyramid value of the point, then querying the $B^+ - tree$ using this value as a key.
- *Range Queries:* This kind of queries returns a group of ontological data. The answer set is a range in the data space rather than a point. In this case, a three dimensional interval is given and the result is a data region that includes the answer set. This data region is searched sequentially for obtaining the exact answer set.

B. Optimizing the Pyramid Technique

In [11], it is proven that the Pyramid Technique may be worse than sequential scan in some cases. Figure 3 shows the difference between querying a data range near the center of the pyramid and near the corner of the pyramid.

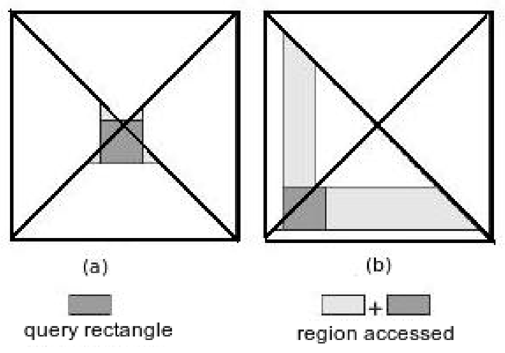

Figure 3. The difference between querying a data range near the center of the pyramid and near the corner of the pyramid.

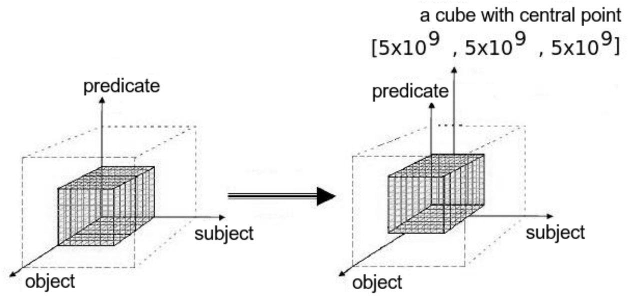

Figure 4. Data Distribution using the modified namespace numbering scheme.

When queries return a data range near the corner, the answer set make up a small percentage of this range. Consequently data accessed is much more than the answer set and sequential search for the answer set in this data makes the Pyramid Technique quite inefficient. Thus it makes sense to shift our data near the center point of the data space. This shifting can be done modifying the mapping scheme slightly. In the modified mapping algorithm,every reference name is numbered sequentially from 0 to 10^{10} and every namespace numbered proceeding through the following way :

Let s_i be the number assigned to the i_{th} namespace, s_0=5 $\times$ 10^5 and $s_{x+1} = \delta(s_x)$ then;

$$\delta(s_x) = \begin{cases} 5 \times 10^5 - \mid 5 \times 10^5 - s_i \mid & s_i < 5 \times 10^5 \\ 5 \times 10^5 + \mid 5 \times 10^5 - s_i \mid +1 & otherwise \end{cases}$$

Given a URI to be mapped u, let n be the number assigned to the namespace of u, r be the number assigned to the reference name of u and x be the 16-digit long number that is the result of mapping procedure, then $x = (n \times 10^{10}) + r$. Figure 4 depicts the effect of this new scheme on data distribution.

VI. Performance Analysis

We use Lehigh University Benchmark [1] in order to evaluate the performance of the inference engine and to see the effects of the Pyramid Technique on performance and scalability. Lehigh University Benchmark is developed to evaluate the querying performance of Semantic Web repositories over a large data set. The data set is generated according to one ontology, named univ-bench, using the synthetic data generation tool provided with the benchmark. The performance of the repositories is evaluated through a set of metrics including data loading time, repository size, query response time, and query completeness and soundness. Benchmark suite also includes 14 example queries that cover a range of types.

Our data set is generated using the synthetic data generation tool. We use LUBM(1,0), LUBM(2,0) and LUBM(3,0) data sets in our benchmark. The tests are done on a desktop computer with the following specifications:

- AMD Athlon 64 3500 2200 Ghz CPU; 2 GB of RAM;320 GB of hard disk
- Windows XP Professional OS, .NET Framework 1.1

The evaluated inference engine, Aegont Inference Engine, is a part of the Aegean Ontology Environment Project. The inference engine is developed to work in correspondence with the Aegont Ontology Editor. Ontology Editor is used to load and query ontologies, in other words, ontology editor can be seen as a graphical user interface to the ontology repository residing in memory. Aegont inference engine is a forward chaining reasoner like OWLJessKB [12], this means once it loads the ontology, the ontology is complete and sound according to the rules defined in the system and there is no need to make inference while answering the queries.

In our system all queries are answered with full completeness and soundness, this results in a high F-measure value, calculated according to the formula in [1]. The system's inference level is between OWL Lite and OWL DL. This inference level is satisfied by approximately 30 rules, which are written according to the OWL entailment tests [13]. In order to get more accurate query execution times, they are measured ten times and then their average is calculated.

Figure 5 represents the improvement in the query execution times of the reasoner after implementing the hybrid optimization heuristic.The optimizations suggested and evaluated in this paper are mainly about decreasing the size of partial instantiations, i.e. count of tokens, created during the execution of the query. When we have one condition, the answering process is trivial, all constructed tokens are in the answer set. When we have two conditions, we need to check the tokens in the second condition with the tokens in the first condition in order to see whether they are in the answers set or not. The order of the conditions doesn't make a difference in a query with two conditions. So we need at least three condition to see the effects of the optimization. Therefore, we inspect queries with number 2, 4, 7, 8, 9 and 12, since they have at least three or more conditions. During our benchmark for all of these queries query execution time is decreased. For queries 2, 7 and 9[5] the improvement is more significant as can be seen from Figure 5. The

[5]The unoptimized query execution time is much bigger than one second so it isn't shown in the figure.

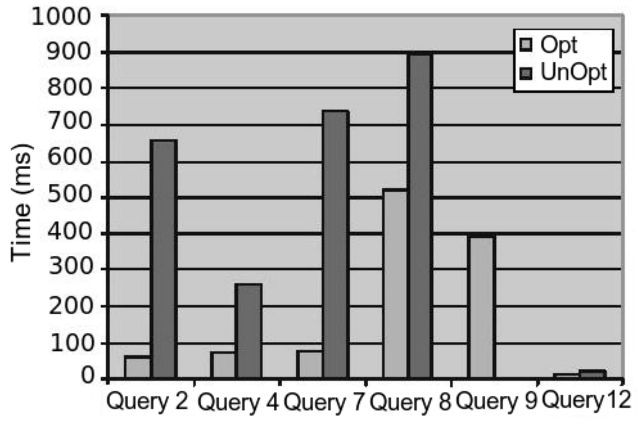

Figure 5. Performance improvements of the queries after implementing the hybrid heuristic.

TABLE III.
CONDITIONS AND SIZE OF THEIR CORRESPONDING ALPHA MEMORIES OF SECOND QUERY.

Number	Condition	Size
a	?x rdf:type ub:GraduateStudent	1874
b	?y rdf:type ub:University	979
c	?z rdf:type ub:Department	15
d	?x ub:memberOf ?z	8330
e	?z ub:subOrganizationOf ?y	463
f	?x ub:undergraduateDegreeFrom ?y	2414

TABLE IV.
BETA MEMORIES AND COUNT OF CONSTRUCTED TOKENS.

Conditions	# tokens	Conditions	# tokens
a	1874	c	15
$a \wedge b$	1834646	$c \wedge e$	15
$a \wedge b \wedge c$	27519690	$c \wedge e \wedge b$	15
$a \wedge b \wedge c \wedge d$	0	$c \wedge e \wedge b \wedge f$	0
$a \wedge b \wedge c \wedge d \wedge e$	0	$c \wedge e \wedge b \wedge f \wedge a$	0
$a \wedge b \wedge c \wedge d \wedge e \wedge f$	0	$c \wedge e \wedge b \wedge f \wedge a \wedge d$	0

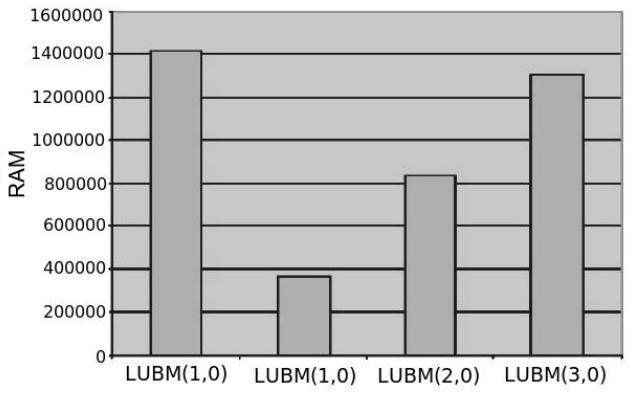

Figure 6. Memory consumption of the reasoner with the Pyramid Technique.

reason for this improvement is the type of conditions of the query. These queries have conditions with no common variables and big alpha memories corresponding to these conditions. When these conditions are computed in a consecutive manner the possibilities to check in successive nodes will increase dramatically. To give an example lets inspect Query 2. Conditions of Query 2 can be seen in Table III.

However, when we optimize the query we will change the order of the conditions. The optimized query order will be c, e, b, f, a, d. While optimizing the query we will start with the most restrictive condition. Most restrictive condition is the condition with the smallest alpha memory, namely c. Then we will find the conditions with common variables with c. These conditions are d and e. Condition e is more restrictive than d, therefore e will be the second condition. In third step, we will find the conditions with common variables with e which are b, d and f. Condition b is more restrictive than d and f therefore b will be the third condition. Condition f is the only condition remaining having common variables with b. f is the forth condition. a and d are remaining conditions and both of them have common variables with b, but a has a smaller alpha memory so it is the fifth condition. Finally, d is the last condition because there aren't any other remaining conditions except d. The number of tokens created during the execution of the query both in optimized and in unoptimized query pattern order is shown in Table IV.

When we compare the size of the partial instantiation of first and second execution of the same query on Table IV, we can see the reason of the difference in the execution time. The main time consuming task in adding a new query to the Rete network is constructing tokens in corresponding nodes. When the possibilities grow we will need to construct more tokens. When we execute the Query 2 in the given order we get 27519690 different possibilities to test with the alpha memory elements of fourth condition. But if we optimize it we will have only 15 different possibilities to test with the alpha memory elements of fourth condition. The cause of the difference in the execution time is the difference in the number of created tokens.

Another reason for the acceleration is the indexing mechanisms we have used in Rete network. By using these indexing mechanisms we eliminate the need for comparison tests between tokens of the previous nodes and alpha memory elements of the current node. But these indexes increased the memory consumption of the reasoner. So we decided to use a new indexing mechanism that is based on the Pyramid Technique. As a result, the memory consumption of the reasoner is reduced by seventy percent by implementing the Pyramid Technique (Figure 6). The first row in the figure shows the memory consumption of the reasoner without the Pyramid Technique. The second, third and fourth rows show the memory consumption of the reasoner with the Pyramid Technique. By using the Pyramid technique the reasoner can open a three times larger data set,LUBM(3,0), with the same amount of memory. And finally, Table V represents the query execution times of the reasoner after

TABLE V.
QUERY EXECUTION TIMES WITH THE PYRAMID TECHNIQUE.

Query	Metrics	LUBM(1,0)	LUBM(2,0)	LUBM(3,0)
1	Time(ms)	762.0	540.0	440.2
	Answers	4	4	4
	Completeness	100	100	100
2	Time(ms)	705.8	434.0	433.8
	Answers	0	0	2
	Completeness	100	100	100
3	Time(ms)	868.6	340.2	421.6
	Answers	6	6	6
	Completeness	100	100	100
4	Time(ms)	40.0	646.4	105.6
	Answers	34	34	34
	Completeness	100	100	100
5	Time(ms)	121.8	502.6	665.2
	Answers	719	719	719
	Completeness	100	100	100
6	Time(ms)	315.0	499.6	596.6
	Answers	7790	17878	26258
	Completeness	100	100	100
7	Time(ms)	874.8	577.6	577.6
	Answers	67	67	67
	Completeness	100	100	100
8	Time(ms)	90.2	718.0	674.4
	Answers	7790	7790	7790
	Completeness	100	100	100
9	Time(ms)	705.8	693.4	634.0
	Answers	208	449	692
	Completeness	100	100	100
10	Time(ms)	462.0	634.0	374.6
	Answers	4	4	4
	Completeness	100	100	100
11	Time(ms)	899.6	765.0	655.4
	Answers	224	224	224
	Completeness	100	100	100
12	Time(ms)	727.6	868.4	340.0
	Answers	15	15	15
	Completeness	100	100	100
13	Time(ms)	687.2	530.8	777.6
	Answers	1	4	8
	Completeness	100	100	100
14	Time(ms)	537.0	280.6	633.6
	Answers	5916	13559	19868
	Completeness	100	100	100

implementing the Pyramid Technique.

VII. CONCLUSION AND FUTURE WORK

In order to reduce the time and memory consumptions of the reasoner, we implement a hybrid optimization technique that reorders rule patterns and an indexing mechanism that is based on the Pyramid Technique. The hybrid optimization increases the query performances as expected. The Pyramid Technique reduces the memory consumption of the reasoner by 70% and makes the reasoner to be able to open larger data sets. The system is still open to develop. The performance can be increased by further optimizations. Since the Pyramid Technique is not affected by the index count, more indexes can be created on ontological data in order to increase query performances. Although there are reasoners that can open larger data sets with better performances, our reasoner differs from them in that it is immune to changes in the data set.

REFERENCES

[1] Y. Guo, Z. Pan, and J. Heflin, "An evaluation of knowledge base systems for large owl datasets." in *International Semantic Web Conference*, 2004, pp. 274–288.

[2] S. Berchtold, C. Böhm, and H.-P. Kriegel, "The pyramid-technique: Towards breaking the curse of dimensionality," in *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, L. M. Haas and A. Tiwary, Eds. ACM Press, 1998, pp. 142–153.

[3] C. Forgy, "Rete: A fast algorithm for the many patterns/many objects match problem." *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.

[4] R. B. Doorenbos, "Production matching for large learning systems," Pittsburgh, PA, USA, Tech. Rep., 2001.

[5] E. Franconi and S. Tessaris, "Rules and queries with ontologies: A unified logical framework." in *PPSWR*, 2004, pp. 50–60.

[6] M. Ünalir, T. Özacar, and Ö. Öztürk, "Reordering query and rule patterns for query answering in a rete-based inference engine." in *WISE Workshops*, 2005, pp. 255–265.

[7] T. Ishida, "Optimizing rules in production system programs," in *National Conference on Artificial Intelligence*, 1988, pp. 699–704. [Online]. Available: citeseer.ist.psu.edu/ishida88optimizing.html

[8] S. Staab, "Ontologies' kisses in standardization," *IEEE Intelligent Systems*, vol. 17, no. 2, pp. 70–79, 2002.

[9] L. Zhang, Y. Yu, J. Lu, C. Lin, K. Tu, M. Guo, Z. Zhang, G. Xie, Z. Su, and Y. Pan, "Orient: Integrate ontology engineering into industry tooling environment." in *International Semantic Web Conference*, 2004, pp. 823–838.

[10] H. V. Jagadish, N. Koudas, and D. Srivastava, "On effective multi-dimensional indexing for strings," in *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2000, pp. 403–414.

[11] R. Zhang, B. C. Ooi, and K.-L. Tan, "Making the pyramid technique robust to query types and workloads." in *ICDE*, 2004, pp. 313–324.

[12] J. Kopena and W. C. Regli, "Damljesskb: A tool for reasoning with the semantic web." in *International Semantic Web Conference*, 2003, pp. 628–643.

[13] J. J. Carroll and J. D. Roo, "Owl web ontology language test cases," 2004.

Tuğba Özacar was born in Izmir, Turkey, on February 03, 1980. She received the M.Sc. degree in Computer Engineering from Ege University in 2004. She is currently working toward the Ph.D. degree in computer engineering at the same institution. Her main research interests include (temporal) reasoning, scalability of reasoning, temporal and knowledge engineering.

Övünç Öztürk was born in Izmir, Turkey, on November 13, 1978. He received the M.Sc. degree in Computer Engineering from Ege University in 2004. He is currently working toward the Ph.D. degree in computer and system engineering at the same institution. His main research interests include reasoning, scalability of reasoning and integration of semantic web inference engines and databases.

Murat Osman Ünalır was born in Izmir, Turkey, on April 15, 1971. He received the M.Sc. and the Ph.D. degree in Computer Engineering from Ege University in 1995 and 2001 respectively. He is currently giving lectures on metadata management and the semantic Web. His main research interests include databases, semantic web and knowledge management.