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

brought to you by CORE

provided by CORE

Wright State University

CORE Scholar

Computer Science and Engineering Faculty Publications

Computer Science & Engineering

10-21-2013

DistEL: A Distributed EL^+ Ontology Classifier

Raghava Mutharaju

Pascal Hitzler
pascal.hitzler@wright.edu

Prabhaker Mateti
prabhaker.mateti@wright.edu

Follow this and additional works at: https://corescholar.libraries.wright.edu/cse

Part of the Computer Sciences Commons, and the Engineering Commons

Repository Citation

Mutharaju, R., Hitzler, P., & Mateti, P. (2013). DistEL: A Distributed EL^+ Ontology Classifier. *CEUR Workshop Proceedings, 1046*, 17-32.
https://corescholar.libraries.wright.edu/cse/214

This Conference Proceeding is brought to you for free and open access by Wright State University's CORE Scholar. It has been accepted for inclusion in Computer Science and Engineering Faculty Publications by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.

DistEL: A Distributed $\mathcal{EL}^+$ Ontology Classifier

Raghava Mutharaju, Pascal Hitzler, and Prabhaker Mateti

Kno.e.sis Center, Wright State University, Dayton, OH, USA

Abstract. OWL 2 EL ontologies are used to model and reason over data from diverse domains such as biomedicine, geography and road traffic. Data in these domains is increasing at a rate quicker than the increase in main memory and computation power of a single machine. Recent efforts in OWL reasoning algorithms lead to the decrease in classification time from several hours to a few seconds even for large ontologies like SNOMED CT. This is especially true for ontologies in the description logic $\mathcal{EL}^+$ (a fragment of the OWL 2 EL profile). Reasoners such as Pellet, Hermit, ELK etc. make an assumption that the ontology would fit in the main memory, which is unreasonable given projected increase in data volumes. Increase in the data volume also necessitates an increase in the computation power. This lead us to the use of a distributed system, so that memory and computation requirements can be spread across machines. We present a distributed system for the classification of $\mathcal{EL}^+$ ontologies along with some results on its scalability and performance.

1 Introduction

The OWL 2 EL profile [4] is used for modeling in several domains like biomedicine,[1] sensors and road traffic [7], and herein we work on a subset called $\mathcal{EL}^+$ [1]. Even though there are not yet any existing very large ontologies in the $\mathcal{EL}^+$ profile, we can very well imagine ontologies with large ABoxes in those domains.[2] Consequently, reasoners should be able to handle very large amounts of data. And although there are some very efficient reasoners available [3, 6], there is only so much a single machine can provide for.

In this paper, we describe a distributed approach to $\mathcal{EL}^+$ ontology classification. Similar to other distributed systems, the design decisions and the performance of our distributed system, DistEL[3] involve answering the following questions effectively.

Synchronization Is synchronization among the distributed processes required? If so, how is it achieved?

Termination What is the termination condition for the distributed processes and how is it detected?

[1] http://bioportal.bioontology.org

[2] $\mathcal{EL}^+$ extended with ABoxes can be handled with essentially the same algorithm.

[3] The source code is available at https://github.com/raghavam/DistEL.

Communication How do the distributed processes communicate and how can this be minimized?

Data Duplication Is data duplication required? How many copies are maintained?

Result Collection After all the processes terminate, will the results be spread across the cluster?

Note that several other characteristics of a distributed system such as fault tolerance, transparency, etc. have been excluded since they are not yet supported by our system. In the following sections, we describe how our system solves the issues mentioned above, and show that our system can handle large ontologies.

The plan of the paper is as follows. In Section 2 we recall preliminaries concerning $\mathcal{EL}^+$. In Section 3 we describe our distributed approach. In Section 4 we present and discuss our experimental evaluation. In Section 5 we discuss limitations of our approach and future work. In Section 6 we discuss related work, and in Section 7 we conclude.

2 Preliminaries

$\mathcal{EL}^+$ Profile We present a brief introduction to the $\mathcal{EL}^+$ profile. For further details, please refer to [1]. Concepts in the description logic $\mathcal{EL}^+$ are formed according to the grammar

$$C ::= A \mid \top \mid C \sqcap D \mid \exists r.C,$$

where A ranges over concept names, r over role names, and C, D over (possibly complex) concepts. An ontology in $\mathcal{EL}^+$ is a finite set of *general concept inclusions* $C \sqsubseteq D$ and *role inclusions* $r_1 \circ \cdots \circ r_n \sqsubseteq r$, where $r, r_1, \ldots, r_n$ are role names, $n \in \mathbb{Z}^+$. For a general introduction to description logics, and for the formal semantics of the constructors available in $\mathcal{EL}^+$, please refer to [5].

Classification Classification is one of the standard reasoning tasks. The classification of an ontology refers to the computation of the complete subsumption hierarchy involving all concept names occurring in the ontology.

A classification algorithm for $\mathcal{EL}^+$ using forward-chaining rules is given in [1], and Table 1 presents a slightly modified set of completion rules which is easily checked to be sound and complete as well [10]. We use these rules to compute the classification of input $\mathcal{EL}^+$ ontologies. For our modification, we divided the rule **R3** from [1] into **R3-1** and **R3-2**, as follows.

R3-1: If $A \in S(Y)$ and $\exists r.A \sqsubseteq B \in \mathcal{O}$, then $\exists r.Y \sqsubseteq B$
R3-2: If $\exists r.Y \sqsubseteq B$ and $(X, Y) \in R(r)$, then $S(X) := S(X) \cup \{B\}$

This helps in the division and distribution of work that needs to be done by our reasoner. The axioms in the ontology are in one of the normal forms given on the left column of Table 1. $S(X)$ contains all the subsumers of X i.e., $A \in S(X)$ means $X \sqsubseteq A$. Likewise, $R(r)$ stands for $\{(X, Y) | X \sqsubseteq \exists r.Y\}$. Our

Normal Form		Completion Rule
$A \sqsubseteq B$	**R1-1**	If $A \in S(X)$, $A \sqsubseteq B \in \mathcal{O}$, and $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$
$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$	**R1-2**	If $A_1, \ldots, A_n \in S(X)$, $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B \in \mathcal{O}$, $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$
$A \sqsubseteq \exists r.B$	**R2**	If $A \in S(X)$, $A \sqsubseteq \exists r.B \in \mathcal{O}$, and $(X, B) \notin R(r)$ then $R(r) := R(r) \cup \{(X, B)\}$
$\exists r.A \sqsubseteq B$	**R3-1**	If $A \in S(Y)$, $\exists r.A \sqsubseteq B \in \mathcal{O}$ then $P = P \cup \{\exists r.Y \sqsubseteq B\}$
$\exists r.A \sqsubseteq B$	**R3-2**	If $(X, Y) \in R(r)$, $\exists r.Y \sqsubseteq B \in P$ and $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$
$r \sqsubseteq s$	**R4**	If $(X, Y) \in R(r)$, $r \sqsubseteq s \in \mathcal{O}$, and $(X, Y) \notin R(s)$ then $R(s) := R(s) \cup \{(X, Y)\}$
$r \circ s \sqsubseteq t$	**R5**	If $(X, Y) \in R(r)$, $(Y, Z) \in R(s)$, $r \circ s \sqsubseteq t \in \mathcal{O}$, $(X, Z) \notin R(t)$ then $R(t) := R(t) \cup \{(X, Z)\}$

Table 1. Axioms (in normal forms) and modified completion rules of CEL

Algorithm 1 Pseudocode for CEL classification fixpoint iteration

$S(X) \leftarrow \{X, \top\}$, for each concept X in the ontology.
$R(r) \leftarrow \{\}$, for each role r in the ontology.
$P \leftarrow \{\}$
repeat
  Used below, Old.$S(X)$ stands for $S(X)$ now, and Old.$R(r)$ stands for $R(r)$ and Old.P stands for P;
  $S(X) \leftarrow$ apply R1-1 using $S(X)$;
  $S(X) \leftarrow$ apply R1-2 using $S(X)$;
  $R(r) \leftarrow$ apply R2 using $S(X)$ and $R(r)$;
  $P \leftarrow$ apply R3-1 using $S(X)$;
  $S(X) \leftarrow$ apply R3-2 using $R(r)$ and P;
  $R(r) \leftarrow$ apply R4 using $R(r)$;
  $R(r) \leftarrow$ apply R5 using $R(r)$;
until ((Old.$S(X) = S(X)$) and (Old.$R(r) = R(r)$) and (Old.$P = P$))

goal is to compute S(X) for each concept X in the ontology $\mathcal{O}$. The $R(r)$ is used in deriving all such subclass relationships. P is a set which holds the axioms generated by rule R3-1. Instead of representing these axioms by P, the other option is to add these axioms back in the ontology $\mathcal{O}$, but we would like to keep the ontology read only.

For classifying $\mathcal{EL}^+$ ontologies, all the rules from Table 1 are processed iteratively until no new output is generated, as shown in Algorithm 1.

Notations We use $U(X)$ to refer to the "inverse" of $S(X)$ i.e., $U(X) = \{A \mid X \sqsupseteq A\}$. The advantage and performance benefit that is obtained by using $U(X)$ instead of $S(X)$ is explained later in Section 3. It is typical in computer science to think of $U(X)$ as applying the definition of a function U to the argument X and thus $U(X)$ does not yield different results on repeated use. In constrast with

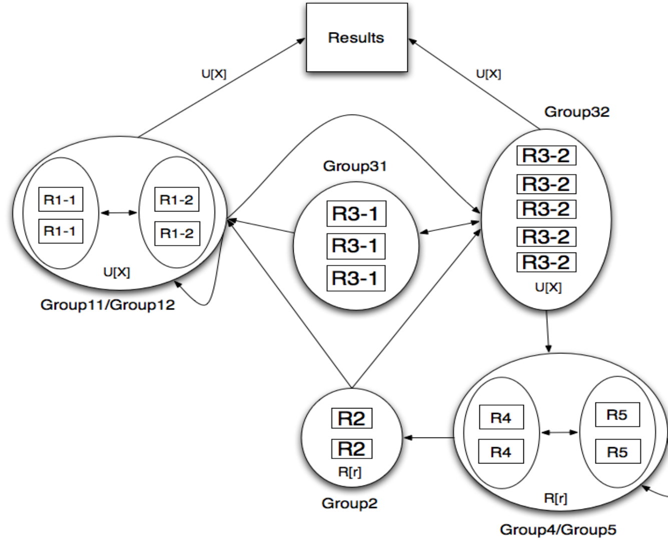

Fig. 1. Node assignment to rules and dependency among the completion rules. Each oval is a collection of nodes (rectangles).

this, we use $U[X]$ to stand for the value stored in an associative array U indexed by a possibly non-integer value X. The same applies to $R(r)$. Hence $U[X]$ and $R[r]$ are conceptually treated as associative arrays, but implementation details might vary. In the rest of the paper we use $U[X]$ and $R[r]$ instead of $S(X)$ and $R(r)$. Sometimes, we refer to all $R[r]$ collectively as *R-data*.

3 Distributed Approach

Architecture Each group of nodes in the cluster (see Figure 1) is dedicated to work on only one particular completion rule R_i, i.e., on axioms belonging to R_i's normal form. Axioms of the ontology are split into disjoint collections based on their normal form. Each disjoint set of axioms are assigned to a particular group, G_i, responsible for processing rule R_i. Within each group, axioms are again split among the nodes of the group. Nodes of Group11, Group12, Group32 produce results which are collected by a single node. We use a set of key-value pairs to represent the axioms and the sets $U[X]$, $R[r]$ that are distributed over the cluster. Figure 1 also shows the dependency among the completion rules i.e., the axioms that are to be processed in the next iteration are determined by the updates done by other nodes. For example, the axioms that will be considered in rule R2 in the next iteration will depend on the output from rules R1-1, R1-2 and R3-2. This is explained further in the optimization section.

Algorithm 2 Pseudocode for rule R1-1

```
K ← 0
for all axioms of the form A ⊑ B do
    K ← K + (U[B] ∪= U[A])    //add U[A] to U[B]
end for
return K
```

Algorithm 3 Pseudocode for rule R1-2

```
K ← 0
for all axioms of the form A1 ⊓ ··· ⊓ An ⊑ B  do
    K ← K + (U[B] ∪= U[A1] ∩ ··· ∩ U[An])
end for
return K
```

Algorithm 4 RolePairHelper({r, X}, B)

```
K ← Send triplet ({r, B}, X) to Group32    //send to D1
if there exists s such that r ⊑ s then
    K ← Send ({r, X}, B) to Group4    //send to D1
end if
if  there exist s and t such that r ∘ s ⊑ t then
    K ← Send ({r, B}, X) to Group5    //send to D0
end if
if  there exist s and t such that s ∘ r ⊑ t then
    K ← Send ({s, X}, B) to Group5    //send to D1
end if
return K
```

Key-Value Store Redis[4] is an open source, high performance key-value store implementation. It provides several data structures like sets, sorted sets, hash, lists. It also supports atomic operations and server-side Lua[5] scripting along with client-side sharding. All these features are used in our implementation. Redis runs on each node of the cluster holding the axioms, $R[r]$ and $U[X]$ values.

We use a Java client named Jedis[6] to interact with Redis.

Pseudocode for completion rules Pseudocode for some rules use a notation such as D_0, D_1. They denote databases of Redis. Each Redis instance can have several databases associated with it. If not mentioned specifically then all the data goes into database-0 (or D_0). Pseudocode for all the rules captures the total number of updates made and returns this number.

The pseudocode of R1-1 is given in Algorithm 2. The operator $\cup=$ performs the set union and returns the number of elements that were added to the destination set – the latter is needed for termination checking, discussed

[4] `http://redis.io`
[5] `http://www.lua.org`
[6] `https://github.com/xetorthio/jedis`

Algorithm 5 Pseudocode for rule R2

```
K ← 0
for all axioms of the form A ⊑ ∃r.B do
    for all X ∈ U[A] do
        K ← K + RolePairHelper({r, X}, B)
    end for
end for
return K
```

Algorithm 6 Pseudocode for rule R3-1

```
K ← 0
for all r, A, B in axioms of the form ∃r.A ⊑ B do
    for all Y ∈ U[A] do
        K ← K + (Send axiom ∃r.Y ⊑ B to Group32)
    end for
end for
return K
```

Algorithm 7 Pseudocode for rule R3-2

```
K ← 0
for all axioms of the form ∃r.Y ⊑ B received from R3-1 do
    T_r := {X | ({r, Y}, X) ∈ D_1};    //received from RolePairHelper
    K ← K + (U[B] ∪= T_r)
end for
return K
```

Algorithm 8 Pseudocode for rule R4

```
K ← 0
T_3 := set of triplets received from RolePairHelper();
T_r := {r | ({r, X}, Y) ∈ T_3};
for all r ∈ T_r do
    for all roles s such that r ⊑ s is an axiom do
        K ← K + RolePairHelper({s, X}, Y)
    end for
end for
return K
```

below. All $U[X]$ are stored on the result node. So $\cup=$ also implicitly involves contacting the result node for read ($U[A]$) and write ($U[X]$) operations.

The pseudocode of R1-2 is given in Algorithm 3. As mentioned below, it suffices to find the intersection of all $U[A]$ involved in the conjuncts, i.e., $A_1 \ldots A_n$.

Expanding on these two rules, let us briefly come back to an issue mentioned earlier, namely why we chose to use $U[X]$ instead of $S[X]$ for our implementation.

Let $\mathcal{O}$ be an ontology and let $K \sqcap L \sqcap M \sqsubseteq N \in \mathcal{O}$. Furthermore, assume that there are five concepts in the ontology, K, L, M, N and P. During some iteration of the classification assume $S(K) = \{K, L, N, \top\}$, $S(L) = \{L, P, M, \top\}$,

Algorithm 9 Pseudocode for rule R5

```
K ← 0
T_0 := set of triplets received from RolePairHelper();    //in D_0
for all ({r, Y}, X) ∈ T_0 do
    T_r := {r | ({r, Y}, Z) ∈ D_1};
    for all t ∈ axioms r ∘ s ⊑ t do
        K ← K + RolePairHelper({t, X}, Z)
    end for
end for
return K
```

$S(M) = \{M, N, K, \top\}$, $S(N) = \{N, \top\}$, and $S(P) = \{P, K, L, M, \top\}$. Now, according to rule R1-2, we have to check for the presence of K, L and M in each of the five $S(X)$, where $X = K, L, M, N, P$. Since only $S(P)$ has K, L, M, we have to add N to $S(P)$.

On the other hand, use instead $U[K] = \{K, M, P\}$, $U[L] = \{L, K, P\}$, $U[M] = \{M, L, P\}$, $U[N] = \{N, K, M, P\}$, $U[P] = \{P, L\}$. In this case, instead of checking all $U[X]$, we can compute the intersection of $U[K]$, $U[L]$, $U[M]$, which is P. So, $P \sqsubseteq N$ which means $U[N] \cup= \{P\}$. In large ontologies, the number of concepts would be in the millions or more, but the number of conjuncts in axioms like $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$ would be very less in number. So the performance is better by using $U[X]$ since set intersection needs to be performed only on a very small number of sets in this case.

Rules R2, R4 and R5 deal with $R[r]$ values. RolePairHelper, with pseudocode given in Algorithm 4, provides functionality that is common to these three rules. The R-data, R[.], is an associative array indexed by roles. RolePairHelper({role r, concept X}, concept B) is invoked in rules R2, R4 and R5. RolePairHelper() informs all nodes (namely Group32, Group4 and Group5) of these updates. Group4 does not care to know the updates to $R[r]$ unless role r has a super role s, that is for some s, $r \sqsubseteq s$. Similarly, Group5 does not care to know of these updates unless there exist roles s and t such that $r \circ s \sqsubseteq t$. Note that the R[.] across all the nodes will, in general, not be the same because of these selective updates. Note also that replicating R[.] across all nodes causes no semantic harm. This is done to facilitate local reads of R-data on nodes dealing with role axioms, i.e., Group32, Group4 and Group5. The Send primitive in Algorithms 4 and 6, returns 0 if the message sent is duplicate, or 1 otherwise.

Nodes handling rule R2, i.e., Group2, do not make use of $R[r]$ values. So they do not need to be stored locally on Group2 nodes as shown in Algorithm 5. Rules R2, R4 and R5 potentially add new entries to the R-data, and every such update implies a triggering of rules R3-2, R4, and R5. RolePairHelper({r, X}, B) broadcasts such updates.

The pseudocode for rule R3-1 is given in Algorithm 6. Here, newly formed axioms do not need to be sent to all the nodes in Group32 but can be sent to only a specific node. For the sake of clarity, this is not shown in the pseudocode and is explained in the section on optimizations.

Algorithm 10 Modeling of One Iteration, $OIP_i(D)$

```
K ← apply rule R_i using (D)
if K == 0 then
    return true
else
    return false
end if
```

Algorithm 11 Modeling of a Process, P_i

```
repeat
    isNew ← OIP_i(D)
    broadcast isNew to all P_j
    receive t_j from all P_j
    t ← t_1 ∨ t_2 ∨ ... ∨ t_j
until ¬t
```

The pseudocode for rule R3-2 is given in Algorithm 7. Here, database-1 (D_1) is queried with key $\{r, Y\}$ and T_r holds all such X.

Regarding R4, in Algorithm 8, T_3 receives only such triples whose r participates in an axiom of the form $r \sqsubseteq s$. T_r is formed for each r found in T_3.

Concerning R5, in Algorithm 9, using the key $\{r, Y\}$, the database D_1 is queried and the results are referenced by T_r. All axioms of the form $r \circ s \sqsubseteq t$ in which r participates in, are retrieved. Note, that the value of s does not matter, since it is already taken care of in RolePairHelper. The following example illustrates how Algorithms 4 and 9 are connected. Let k, m and n be roles, where $k \circ m \sqsubseteq n$, $(X, Y) \in R(k), (Y, Z) \in R(m)$. RolePairHelper({k, X}, Y) sends ({k, Y}, X) to D_0 of Group5. RolePairHelper({m, Y}, Z) sends ({k, Y}, Z) to D_1 of Group5. In Algorithm 9, T_k contains ({k, Y}, X). D_1 is queried with {k, Y} as the key and ({n, X}, Z) is produced. n is obtained from the Rolechain axiom.

Termination One iteration of the process, OIP, is modeled by the pseudocode shown in Algorithm 10. Each OIP reads and writes to a Redis instance D (database).

The appropriate pseudocode for rule R_i is processed and the return value is collected in K, which holds the number of updates made. Depending on the value of K, either true or false is returned.

Each process P_i performs the computations in Algorithm 11. The receive() is a blocking operation; i.e., until a message is received, the calling process (P_i) does not proceed to the next state. However, even though the pseudocode does not show it, we assume that the messages from P_j can be received in any order.

On each node, N_i, the process P_i processes all the axioms local to it and keeps track of whether this resulted in any changes (*isNew*) to either its local Redis database or to the one on other nodes. This boolean value is broadcast to all the processes. When all the *isNew* messages are received, each process on all the nodes knows whether any of the other process made some updates or

not. If at least one process makes an update then all the processes continue with the next iteration. If none of the processes makes an update then all processes terminate.

All the nodes in the cluster keep processing the axioms over several iterations until no new output is generated by any of the nodes. In sequential computation, this is fairly easy to check, but in a distributed system, all the nodes should be coordinated in order to check whether any of them have produced a new output.

The coordination among the processes on all nodes is achieved by message passing. Process P_i on each node is associated with a channel C_i. At the end of each iteration, P_i broadcasts its status message to channels on all the nodes. It then does a blocking wait until it receives messages from all the processes on its channel. This is generally known as barrier synchronization.[7] If all the messages that P_i receives are false, i.e., none of the processes made an update to any of the key-value pairs, then P_i terminates.

Optimizations The following optimizations were put in place to speed up the processing of rules.

1. All the concepts and roles in the ontology are assigned numerical identifiers. This saves space and is easier to process.
2. If $X \sqsubseteq A$, normally this would be stored in a set whose key would be X and value would be A. But we reverse it, and make A the key and X its value. This makes the check $A \in S(X)$, a single read call. This check is required in rules R1-1, R1-2, R2 and R3-1.
3. As shown in Figure 1, the output of a rule can affect the processing of another rule. For example, rule R2 works on axioms of the form $A \sqsubseteq \exists r.B$. R2 then depends on the rules which affect A, which are R1-1, R1-2 and R3-2. If R1-1, R1-2 and R3-2 do not make any changes to $U[A]$, then the axiom $A \sqsubseteq \exists r.B$ need not be considered in the next iteration. We keep track of these dependencies and thereby reduce the number of axioms to work on in subsequent iterations.
4. Extending on the optimization just mentioned, if there is a change in $U[A]$, then not all elements of $U[A]$ need to be considered again. In fact, we need to consider only the newly added elements. This can be achieved by assigning scores to each element in the set $U[A]$. A node working on rule R2 and axiom $A \sqsubseteq \exists r.B$ keeps track of the scores of elements in $U[A]$, i.e., up to what it has read in the previous iteration, and only considers elements whose scores are greater than that.
5. In the pseudocode of Algorithm 6 and Algorithm 4, it is shown that the newly formed axiom and the triple is sent to all the nodes of a particular group. Instead, they can only be sent to a particular node in the group, and this node is selected based on the key. For the same key within the same group, however, we can ensure that always the same node gets selected. This reduces duplication of data.

[7] `http://en.wikipedia.org/wiki/Barrier_(computer_science)`

Ontology	#Logical Axioms	#Concepts	#Roles
Not-Galen	8,015	4,242	413
GO	28,897	20,465	1
NCI	46,870	27,653	70
SNOMED	1,038,481	433,106	62
SNOMED-DUP-2	2,076,962	866,212	124
SNOMED-DUP-3	3,115,443	1,299,318	186
SNOMED-GALEN-GO	1,075,393	456,319	476

Table 2. Sizes of (normalized) ontologies we used

4 Evaluation

To evaluate our implementation, we made use of the seven ontologies that are listed in Table 2. The numbers in Table 2 are obtained after normalizing the ontologies. The first three ontologies have been obtained from `http://lat.inf.tu-dresden.de/~meng/toyont.html`. SNOMED is from `http://www.ihtsdo.org/snomed-ct`. SNOMED-DUP-2 and SNOMED-DUP-3 are ontologies with axioms from SNOMED, but each axiom replicated twice and thrice, respectively, while concept and role names are systematically renamed for each copy. SNOMED-GALEN-GO is a merge of the three ontologies, SNOMED, Not-Galen and GO, which was obtained synthetically as follows: Upon normalization of each of these ontologies, new class names and role names were created which were assigned to a local namespace. However, class names and role names introduced in the normalization are *shared* between the ontologies. We thus obtain a merged ontology which, albeit the merge is synthetic, retains some of the real-life character of each of these ontologies.

DistEL is implemented in Java and makes use of Redis for storage. Our cluster consists of 13 Linux nodes, but our implementation scales to larger clusters. Each node has two quad-core AMD Opteron 2300MHz processors with 16GB RAM. DistEL treats a Redis instance as a node, so in order to show the scalability aspect of our implementation, we ran 2-3 Redis instances on a single node. This allowed us to effectively run tests for more than 13 nodes on the cluster. Since each node has 8 cores, and data on an instance of Redis generally doesn't go beyond 5GB (for our experiments), running 2-3 Redis instances does not adversely affect the evaluation.

Table 3 has the classification time of ontologies when run on Pellet (version 2.3.0), jCEL (0.18.2) and ELK (version 0.3.2). Heap space given to run all the ontologies is 12GB. Timeout limit given was 2 hours. All the reasoners are invoked through the OWL API. Time taken by the OWL API to load the ontology is not taken into consideration. Note that SNOMED-GALEN-GO could not be processed by any of the state-of-the-art systems. This is remarkable because the

Ontology	Pellet	jCEL	ELK
Not-Galen	12.0	3.0	1.0
GO	5.0	5.0	2.0
NCI	6.0	7.0	3.0
SNOMED	1,845.0	327.0	24.0
SNOMED-DUP-2	OutOfMemory	687.0	64.0
SNOMED-DUP-3	OutOfMemory	1149.0	93.0
SNOMED-GALEN-GO	OutOfMemory	TIME OUT	TIME OUT

Table 3. Classification time of ontologies using Pellet, jCEL and ELK

Ontology	7 nodes	9 nodes	12 nodes	15 nodes	18 nodes
Not-Galen	6.76	6.44	6.67	6.67	7.09
GO	11.58	11.65	11.74	12.59	12.51
NCI	21.13	21.57	21.53	22.15	22.80
SNOMED	382.77	385.09	392.09	398.07	393.57
SNOMED-DUP-2	774.34	767.85	787.10	798.58	826.50
SNOMED-DUP-3	2,160.00	2,160.00	2,113.57	2,194.80	2,233.12
SNOMED-GALEN-GO	—	—	—	—	411.72

Ontology	21 nodes	25 nodes	28 nodes	32 nodes
Not-Galen	6.78	6.83	6.74	6.77
GO	12.30	12.46	12.87	12.93
NCI	22.53	22.63	22.66	22.15
SNOMED	396.66	405.94	410.07	412.39
SNOMED-DUP-2	803.43	805.81	828.55	828.78
SNOMED-DUP-3	2,177.13	2315.19	2163.17	2257.94
SNOMED-GALEN-GO	416.99	418.99	419.58	428.43

Table 4. Load times (in seconds) of DistEL

ontology is hardly larger than SNOMED itself.[8] In fact, it shows that realistic handcrafted ontologies of a size which cannot be handled by state-of-the-art reasoners are not far out of reach.

Pre-processing times (in seconds) for the ontologies on DistEL are given in Table 4. Pre-processing includes the time taken by the OWL API[9] to load the ontology in-memory and the time taken to insert the axioms into the nodes of the cluster, and it is approximately constant with respect to the number of nodes. Time taken by SNOMED-GALEN-GO is not mentioned for 7, 9, 12 and 15 nodes because the classification time on these nodes times out.

Table 5 shows the classification times of DistEL with varying numbers of nodes, and a corresponding visualization is given in Figure 2. For the larger ontologies, i.e., SNOMED and larger, we see a steady decrease of classification time with increasing number of nodes used. Note that, for the larger ontologies,

[8] We are not entirely certain yet what causes this explosion. Our guess is that it is caused by the large number of role chains in Galen, together with the sheer size of SNOMED.

[9] `http://owlapi.sourceforge.net`

Ontology	7 nodes	9 nodes	12 nodes	15 nodes	18 nodes
Not-Galen	43	42.27	41.06	39.12	36.70
GO	46.20	49.39	51.83	52.44	53.62
NCI	275	168.96	157.36	156.45	156.82
SNOMED	1,610.00	1,355.81	865.89	886.44	613.53
SNOMED-DUP-2	3,238.19	2,687.75	1,699.73	1,765.31	1,255.87
SNOMED-DUP-3	4,880.78	4,052.00	2,570.29	2,644.40	1,825.51
SNOMED-GALEN-GO	TIME OUT	TIME OUT	TIME OUT	TIME OUT	1,336.28

Ontology	21 nodes	25 nodes	28 nodes	32 nodes
Not-Galen	37.51	36.69	36.11	35.09
GO	51.89	56.76	39.70	50.80
NCI	155.19	154.75	161.41	160.12
SNOMED	529.30	441.74	442.81	383.01
SNOMED-DUP-2	1,064.44	887.19	893.96	755.38
SNOMED-DUP-3	1,571.43	1278.62	1286.50	1146.71
SNOMED-GALEN-GO	1,241.96	702.02	693.51	618.18

Table 5. Classification time (in seconds) of DistEL

the effect of parallelization is very good indeed. E.g., using twice the number of nodes almost halves the runtime in most cases – so the effect of the parallelization is indeed near optimal.

As expected, for the smaller ontologies, the parallelization does not have much effect as soon as a certain threshold is reached. We see, however, that even when using many more nodes than necessary to reach optimal runtime, we do not get a significant amount of additional time lost due to communication overhead.

After having retrieved the runtime figures just discussed, we noticed that some of the measured times for 15 nodes were in fact higher than those of 12 nodes, and a similar effect showed for 28 versus 25 nodes. When additional nodes are added to the cluster, they should ideally be assigned to the slowest processing nodes. But if this is not the case, then we do not see any noticeable improvement in performance. On the contrary, there is a possibility of reduction in performance because of additional communication overhead. Since we are currently assigning nodes by intuition and rule-of-thumb, we have to expect to get such performance drops sometimes.

To further check on this effect, we repeated the run of SNOMED on 15 nodes using a different assignment of nodes to rules, and it resulted in a classification time of 606.05 seconds, which is a significant improvement compared to the timing obtained by the node assignment in Table 5. In fact, it is better than using 18 nodes in the previous assignment. This shows that our manual rule-of-thumb assignments are likely not optimal, and that, in fact, a signifcantly better performance should be achievable by automating the node assignment, or by using methods for dynamic load balancing. However, in this paper we only want to show that significant parallelization can be achieved, and do not yet focus on a most efficient implementation. This is left for future work.

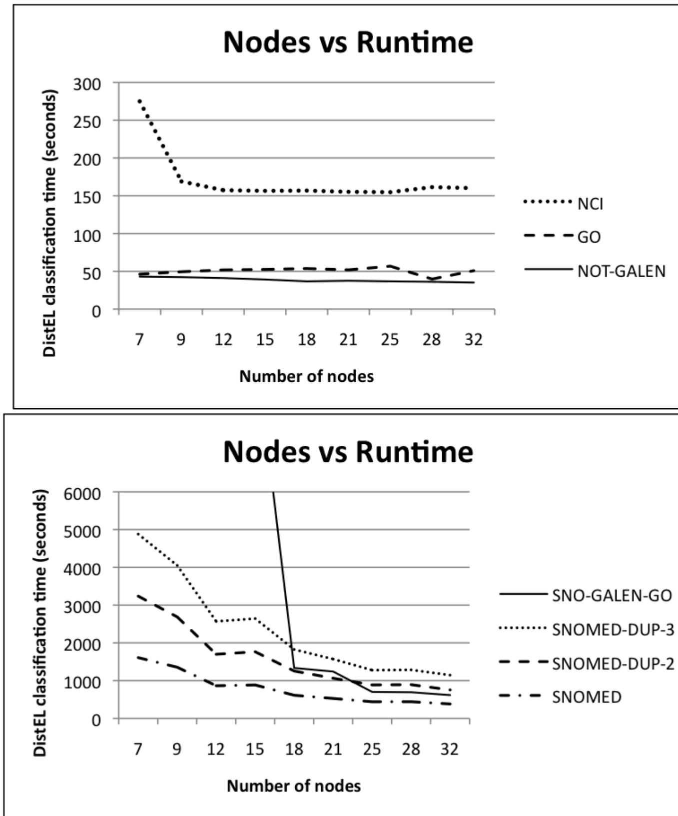

Fig. 2. Visualization of number of nodes vs runtimes

Correctness of the results produced by DistEL is verified by comparing the output with that of ELK, in the cases where ELK does not time out.

There is a significant difference in performance between our distributed implementation and ELK. One of the primary reasons for this difference is that our implementation requires cross-node communication (key-value pairs are sent across the nodes) and among the nodes. On the other hand, our architecture can be extended by adding more nodes, and thus can scale up to datasets which ELK cannot handle, such as SNOMED-GALEN-GO.

We list some further insights which we gained from our implementation and system.

1. For our manual assignment of nodes to rules, it is very important to figure out the slowest processing nodes in the cluster, so that, if additional nodes are available, it would be easy to determine, to which group these new nodes should be assigned to.

2. The majority of the time is in fact spent on reading and writing to local as well as remote databases. Design choices and architecture should be formulated in such a way so as to reduce cross-node communication.
3. We cannot estimate the runtime or node assignment to rules just by counting the number of axioms. Some axioms are harder to process than others although their number might be less. A case in point is SNOMED-GALEN-GO compared with SNOMED-DUP-3.

5 Limitations and Future Work

Some of the limitations and planned next steps are presented below.

1. Compared to other popular distributed frameworks like Hadoop,[10] our architecture does not currently provide support for fault tolerance.
2. Axioms are distributed across the cluster by type rather than load. This leads to improper load balancing. To improve load balancing we plan to implement work stealing so that the idle nodes can work on axioms from other nodes.
3. Completion rules are assigned to nodes manually, for now. This could be done automatically by considering ontology statistics such as the number of role axioms and subclass axioms, or other measures.
4. The OWL API is used to read the axioms from an ontology, and by design it loads the entire ontology into memory. With the size of ontologies that we hope to deal with using our work, this becomes a bottleneck. Going forward, we plan to use a streaming API for XML[11] to read the axioms.
5. We plan to extend our work to include ABox reasoning [11] where there is a greater scope of getting large ontologies and our distributed system could be put to test.
6. We also intend to use multicore threading to take advantage of the number of cores in modern machines.
7. Another possible line of future work is to apply our distributed approach to other $\mathcal{EL}^+$classification algorithms such as the materialization procedure used by ELK.

6 Related Work

Most of the distributed reasoning approaches in the literature are focussed on RDFS inference but there are a few that deal with a fragment of OWL, namely OWL Horst. In [17], Urbani et al., use MapReduce for reasoning over OWL Horst. They look to carry over their work from distributed reasoning over RDFS to OWL Horst, which was possible only to a certain extent. Soma et al., [15] investigate partitioning approaches for parallel inferencing in OWL Horst. Although not distributed, a backward chaining approach [16] is used to scale up

[10] `http://hadoop.apache.org`
[11] `http://en.wikipedia.org/wiki/StAX`

to a billion triples in the OWL Horst fragment. A distributed approach to fuzzy OWL Horst reasoning has also been investigated in [8].

Distributed resolution techniques were used by Stuckenschmidt et al., to achieve scalability of various OWL fragments such as $\mathcal{ALC}$ [12] and $\mathcal{ALCHIQ}$ [13]. There have been attempts at achieving distributed reasoning on the $\mathcal{EL}^+$profile in [10] and [14], but they do not provide any evaluation results. Distribution of OWL EL ontologies over a peer-to-peer network and algorithms based on distributed hash table have been attempted in [2], but again, no evaluation results are provided. Reasoning over Fuzzy-EL+ ontologies using MapReduce [18] has also been attempted but implementation and experiment details are not provided.

We have tried several distributed approaches for $\mathcal{EL}^+$ontology classification, some of which have been unsuccessful. Here, we presented an approach which gave encouraging results due to increase in the number of axioms that are processed locally and decrease in the cross-node communication, when compared to our previous approaches [9]. Our earlier approaches include use of MapReduce (involved many redundant computations) and distributed queue (distribution of CEL's queue approach). The latter approach involves a lot of cross-node communication.

7 Conclusions

With ever increasing data generation rates, large ontologies challenge reasoners from the perspective of memory and computation power. In such scenarios, distributed reasoners offer a viable solution. We presented our distributed approach to $\mathcal{EL}^+$ ontology classification, called DistEL, where we show that our classifier can handle large ontologies and the classification time decreases with the increase in nodes. The results are encouraging, and we plan to go ahead with adding ABox reasoning support to our work as well as explore other possible distributed classification approaches.

Acknowledgements. We would like to thank members of the Redis mailing list `redis-db@googlegroups.com` for their helpful suggestions. This work was supported by the National Science Foundation under award 1017225 "III: Small: TROn – Tractable Reasoning with Ontologies." Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Baader, F., Lutz, C., Suntisrivaraporn, B.: Is Tractable Reasoning in Extensions of the Description Logic EL Useful in Practice? In: Proceedings of the 2005 International Workshop on Methods for Modalities (M4M-05) (2005)
2. Battista, A.D.L., Dumontier, M.: A Platform for Reasoning with OWL-EL Knowledge Bases in a Peer-to-Peer Environment. In: Proceedings of the 5th International Workshop on OWL: Experiences and Directions, Chantilly, VA, United States, October 23-24 2009. CEUR Workshop Proceedings, vol. 529. CEUR-WS.org (2009)

3. Dentler, K., Cornet, R., ten Teije, A., de Keizer, N.: Comparison of Reasoners for large Ontologies in the OWL 2 EL Profile. Semantic Web 2(2), 71–87 (2011)
4. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S. (eds.): OWL 2 Web Ontology Language: Primer. W3C Recommendation (27 October 2009), available from http://www.w3.org/TR/owl2-primer/
5. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
6. Kazakov, Y., Krötzsch, M., Simancik, F.: Concurrent Classification of EL Ontologies. In: 10th International Semantic Web Conference, Bonn, Germany, October 23-27. Lecture Notes in Computer Science, vol. 7031, pp. 305–320. Springer (2011)
7. Lécué, F., Schumann, A., Sbodio, M.L.: Applying Semantic Web Technologies for Diagnosing Road Traffic Congestions. In: International Semantic Web Conference (2). Lecture Notes in Computer Science, vol. 7650, pp. 114–130. Springer (2012)
8. Liu, C., Qi, G., Wang, H., Yu, Y.: Large Scale Fuzzy pD* Reasoning Using MapReduce. In: 10th International Semantic Web Conference, Bonn, Germany, October 23-27. Lecture Notes in Computer Science, vol. 7031, pp. 405–420. Springer (2011)
9. Mutharaju, R.: Very Large Scale OWL Reasoning through Distributed Computation. In: International Semantic Web Conference (2). Lecture Notes in Computer Science, vol. 7650, pp. 407–414. Springer (2012)
10. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce Algorithm for EL+. In: Proceedings of the 23rd International Workshop on Description Logics (DL 2010), Waterloo, Ontario, Canada, May 4-7, 2010. CEUR Workshop Proceedings, vol. 573. CEUR-WS.org (2010)
11. Ren, Y., Pan, J.Z., Lee, K.: Parallel ABox reasoning of EL ontologies. In: Proceedings of the 2011 Joint International Conference on the Semantic Web. pp. 17–32. JIST'11, Springer, Heidelberg (2012)
12. Schlicht, A., Stuckenschmidt, H.: Distributed Resolution for ALC. In: Proceedings of the 21st International Workshop on Description Logics (DL2008), Dresden, Germany, May 13-16. CEUR Workshop Proceedings, vol. 353. CEUR-WS.org (2008)
13. Schlicht, A., Stuckenschmidt, H.: Distributed Resolution for Expressive Ontology Networks. In: Proceedings of the Third International Conference on Web Reasoning and Rule Systems, RR 2009, Chantilly, VA, USA, October 25-26, 2009. Lecture Notes in Computer Science, vol. 5837, pp. 87–101. Springer (2009)
14. Schlicht, A., Stuckenschmidt, H.: MapResolve. In: Web Reasoning and Rule Systems – 5th International Conference, RR 2011, Galway, Ireland, August 29-30, 2011. Lecture Notes in Computer Science, vol. 6902, pp. 294–299. Springer (2011)
15. Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In: 2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA. pp. 75–82. IEEE Computer Society (2008)
16. Urbani, J., van Harmelen, F., Schlobach, S., Bal, H.E.: QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases. In: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011. Lecture Notes in Computer Science, vol. 7031, pp. 730–745. Springer (2011)
17. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In: Proceedings of the 8th Extended Semantic Web Conference (ESWC2010), Heraklion, Greece, May 30–June 3, 2010. Springer (2010)
18. Zhou, Z., Qi, G., Liu, C., Hitzler, P., Mutharaju, R.: Reasoning with Fuzzy-EL+ Ontologies Using MapReduce. In: Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012). Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 933–934. IOS Press (2012)