Distributed and Scalable OWL EL Reasoning

Raghava Mutharaju[1(✉)], Pascal Hitzler[1], Prabhaker Mateti[1], and Freddy Lécué[2]

[1] Wright State University, OH, USA
{mutharaju.2,pascal.hitzler,prabhaker.mateti}@wright.edu
[2] Smarter Cities Technology Centre, IBM Research, Dublin, Ireland
freddy.lecue@ie.ibm.com

Abstract. OWL 2 EL is one of the tractable profiles of the Web Ontology Language (OWL) which is a W3C-recommended standard. OWL 2 EL provides sufficient expressivity to model large biomedical ontologies as well as streaming data such as traffic, while at the same time allows for efficient reasoning services. Existing reasoners for OWL 2 EL, however, use only a single machine and are thus constrained by memory and computational power. At the same time, the automated generation of ontological information from streaming data and text can lead to very large ontologies which can exceed the capacities of these reasoners. We thus describe a distributed reasoning system that scales well using a cluster of commodity machines. We also apply our system to a use case on city traffic data and show that it can handle volumes which cannot be handled by current single machine reasoners.

1 Introduction

We predict that ontology-based knowledge bases will continue to grow to sizes beyond the capability of single machines to keep their representations in main memory. Manually constructed knowledge bases will most likely remain considerably smaller, but the automated generation of ABox and TBox axioms from e.g. data streams [10] or texts [12] will likely go beyond the capabilities of current single-machine systems in terms of memory and computational power required for deductive reasoning. Also, for some reasoning tasks the output is several times larger than the input. For such cases, distributed memory reasoning will be required.

In this paper, we consider knowledge bases (ontologies) which fall into the tractable OWL 2 EL profile [13]. In particular, our distributed reasoner, `DistEL`, supports almost all of $\mathcal{EL}^{++}$ which is the description logic underlying OWL 2 EL. The following are our main contributions.

1. We describe our distributed algorithms along with the data distribution and load balancing scheme. To the best of our knowledge, this is the first such work for the $\mathcal{EL}^{++}$ description logic.
2. We demonstrate that `DistEL` scales well and also achieves reasonable speedup through parallelization. It can handle ontologies much larger than what current other reasoners are capable of.

F. Gandon et al. (Eds.): ESWC 2015, LNCS 9088, pp. 88–103, 2015.
DOI: 10.1007/978-3-319-18818-8_6

3. DistEL is GPL open-sourced at https://github.com/raghavam/DistEL. Its usage and build are fully documented and it works on publicly available ontologies.

The paper is structured as follows. After recalling preliminaries on OWL EL (Sect. 2), we describe the algorithms for DistEL (Sect. 3) and discuss some specific optimizations we have used (Sect. 4). We close with a performance evaluation (Sect. 5), related work (Sect. 6), and a conclusion (Sect. 7).

2 Preliminaries

We will work with a large fragment of the description logic $\mathcal{EL}^{++}$ [2] which underlies OWL 2 EL. We briefly recall notation, terminology, and key definitions, primarily taken from [2] which serves as general reference. We define only the fragment which we use through this paper, and for convenience we call it *EL**.

The underlying language of our logic consists of three mutually disjoint sets of atomic concept names N_C, atomic role names N_R and individuals N_I. An (EL*-)axiom can have one of the following forms. (i) General concept inclusions of the form $C \sqsubseteq D$, where C and D are *classes* defined by the following grammar (with $A \in N_C$, $r \in N_R$, $a \in N_I$):

$$C : := A \mid \top \mid \bot \mid C \sqcap C \mid \exists r.C \mid \{a\}$$
$$D : := A \mid \top \mid \bot \mid D \sqcap D \mid \exists r.D \mid \exists r.\{a\}$$

(ii) Role inclusions of the form $r_1 \circ \cdots \circ r_n \sqsubseteq r$, where $r, r_i \in N_R$.

An (EL*-)ontology consists of a finite set of EL*-axioms. Axioms of the form $\{a\} \sqsubseteq A$ and $\{a\} \sqsubseteq \exists r.\{b\}$ are called *ABox* axioms, and they are sometimes written as $A(a)$ and $R(a,b)$ respectively.

The primary omissions from $\mathcal{EL}^{++}$ are concrete domains and that we limit the use of *nominals*, which are classes of the form $\{a\}$, to the inclusion of ABox axioms as described above.[1] In particular, DistEL does not support concept inclusions of the form $C \sqsubseteq \{a\}$.

The model-theoretic semantics for EL* follows the standard definition, which we will not repeat here. For this and other background see [7].

We recall from [2] that every EL* ontology can be normalized in such a way that all concept inclusions have one of the forms $A_1 \sqsubseteq B, A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B, A_1 \sqsubseteq \exists r.A_2, \exists r.A_1 \sqsubseteq B$ and that all role inclusions are in the form of either $r \sqsubseteq s$ or $r_1 \circ r_2 \sqsubseteq r_3$, where $A_i \in BC_{\mathcal{O}} = N_C \cup \{\top\}$ (for all i) and $B \in BC_{\mathcal{O}}^{\bot} = N_C \cup \{\bot\}$.

In rest of the paper, we assume that all ontologies are normalized.

The reasoning task that is of interest to us (and which is considered the main reasoning task for $\mathcal{EL}^{++}$) is that of *classification*, which is the computation of the complete subsumption hierarchy, i.e. of all logical consequences of the form $A \sqsubseteq B$ involving all concept names and nominals A and B. Other tasks such

[1] Domain axioms can be expressed directly, and allowed range axioms can be rewritten into EL* as shown in [3].

Table 1. Completion rules and key value pairs

Rn	Input	Action	Key: Value
R1	$A \sqsubseteq B$	$U[B] \cup= U[A]$	$A_{R1} : B$
R2	$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$	$U[B] \cup= U[A_1] \cap \cdots \cap U[A_n]$	$(A_1, \ldots, A_n)_{R2} : B$
R3	$A \sqsubseteq \exists r.B$	$R[r] \cup= \{(X, B) \mid X \in U[A]\}$	$A_{R3} : (B, r)$
R4	$\exists r.A \sqsubseteq B$	$Q[r] \cup= \{(Y, B) \mid Y \in U[A]\}$	$A_{R4} : (B, r)$
R5	$R[r], Q[r]$	$U[B] \cup= \{X \mid (X, Y) \in R[r]$ and $(Y, B) \in Q[r]\}$	⟨none⟩
R6	$R[r]$	$U[\bot] \cup= \{X \mid (X, Y) \in R[r]$ and $B \in U[\bot]\}$	⟨none⟩
R7	$r \sqsubseteq s$	$R[s] \cup= R[r]$	$r_{R7} : s$
R8	$r \circ s \sqsubseteq t$	$R[t] \cup= \{(X, Z) \mid (X, Y) \in R[r]$ and $(Y, Z) \in R[s]\}$	$r_{R8a} : (s, t)$
			$s_{R8b} : (r, t)$
		$U[X] = \{A, B, \ldots\}$	$X_U : \{A, B, \ldots\}$
		$R[r] = \{(X, Y), \ldots\}$	$(Y, r)_{RY} : X; \ldots$
			$(X, r)_{RX} : Y; \ldots$
		$Q[r] = \{(X, Y), \ldots\}$	$(Y, r)_Q : X; \ldots$

as concept satisfiability and consistency checking are reducible to classification. Note that ABox reasoning (also known as *instance retrieval*) can be reduced to classification in our logic.

To classify an ontology, we use the completion rules given in Table 1 (left of the vertical line). These rules make use of three mappings $U : BC_{\mathcal{O}}^{\bot} \rightarrow 2^{BC_{\mathcal{O}}^{\bot}}$, $R : N_R \rightarrow 2^{BC_{\mathcal{O}} \times BC_{\mathcal{O}}}$ and $Q : N_R \rightarrow 2^{BC_{\mathcal{O}} \times BC_{\mathcal{O}}^{\bot}}$ which encode certain derived consequences. More precisely, $X \in U[A]$ stands for $X \sqsubseteq A$, while $(A, B) \in R[r]$ stands for $A \sqsubseteq \exists r.B$ and $(A, B) \in Q[r]$ stands for $\exists r.A \sqsubseteq B$. For each concept $X \in BC_{\mathcal{O}}^{\bot}$, $U[X]$ is initialized to $\{X, \bot\}$, and for each role r, $R[r]$ and $Q[r]$ are initialized to $\emptyset$. The operator $\cup=$ adds elements of the set on the right-hand side to the set on the left-hand side.

The rules in Table 1 are applied as follows. Given a (normalized) input ontology, first initialize the $U[X]$, $R[r]$ and $Q[r]$ as indicated. Each axiom in the input knowledge base is of one of the forms given in the Table 1 *Input* column, and thus gives rise to the corresponding action given in the table. R5 and R6 are exceptions as they do not correspond to any input axiom types, but instead they take $Q[r]$, $R[r]$ as input and trigger the corresponding action.

To compute the completion, we non-deterministically and iteratively execute all actions corresponding to all of the rules. We do this to exhaustion, i.e., until none of the actions resulting from any of the axioms causes any change to any of the $U[X]$, $R[r]$ or $Q[r]$. Since there are only finitely many concept names, role names, and individuals occurring in the input knowledge base, the computation will indeed terminate at some stage.

The rules in Table 1 are from [2], except for rules R4 and R5, which is combined into one rule in [2]. Using two rules instead of one helps in the division and distribution of work in our reasoner; conceptually, we only have to store intermediate results (using Q, and this is the only use of Q we make), and otherwise there is no difference. We also use the function U instead of a function S which is used in [2], where $A \in S[X]$ is used to stand for $X \sqsubseteq A$. The difference is really notational only. Our rules (and corresponding algorithm) are really just a minor syntactic variation of the original rules, and the original correctness proofs carry over trivially. In Sect. 4 we will comment further on the reasons we have for using U instead of S: while it is only a notational variant, it is actually helpful for algorithm performance.

In `DistEL`, we use key:value pairs to encode both the input knowldge base and the output resulting from rule actions. In turn, these key:value pairs are also used to control the (then deterministic) parallel and sequential execution of rules, and we will discuss this in detail in the next section.

3 Algorithms of `DistEL`

In the algorithm descriptions in this section, we use a few CSP [1] inspired notations. The expression P **!** tag(e) **?** v, occurring in a process Q, denotes that the message tag(e) is sent to a process named P and the response received from P is assigned to v. If P is not ready to receive tag(e), Q blocks until P is ready. After this message is sent, Q waits for a response from P which it will save in v. P may take a while to compute this response. But when it sends this reply, Q is ready (since it has been waiting). So P does not block when replying. The corresponding expression Q **?** tag(u) occurring in process P denotes receiving a message tag(e) from process Q and the body of the message is assigned to variable u local to P. The expression P **!** tag(e) occurring in a process Q simply sends a message tag(e) to process P.

A process might receive many messages, and in order to distinguish between them and provide the right service to the requester, *tag* is used. These tags are descriptive names of the service that ought to be provided.

The **on** statements stand for an event processing mechanism that is ever ready but asleep until triggered by a request, and the corresponding response is shown on the rhs of the **do**.

Table 1 lists six unique axiom forms (excluding R5 and R6). R5 and R6 depend on the sets, $Q[r]$ and $R[r]$, for each role r. $Q[r]$ and $R[r]$ are set representations of axioms. For simplicity, we consider these two sets also as two separate axiom forms. This gets the total axiom forms to eight and now the input ontology $\mathcal{O}$ can be partitioned into eight mutually disjoint ontologies, $\mathcal{O} = \mathcal{O}_1 \cup \cdots \cup \mathcal{O}_8$, based on the axiom forms. Ontology $\mathcal{O}_i$ is assigned to a subcluster (subset of machines in the cluster) SC_i. Rule Ri, and no other, must be applied on $\mathcal{O}_i$. `DistEL` creates eight subclusters, one for each rule, from the available machines. For example (Fig. 1) axioms that belong to SC_4 are divided among its three nodes. Note that, axioms in $\mathcal{O}_i$ are further divided among the machines in SC_i and are not duplicated.

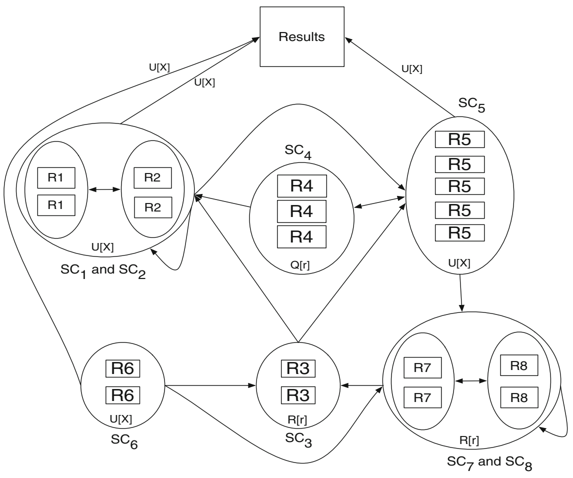

Fig. 1. Node assignment to rules and dependency among the completion rules. A rectangle represents a node in the cluster and inner ovals represent subclusters. Outer ovals enclosing SC_1/SC_2, and SC_7/SC_8 show their tighter input-output relationships. The set (U[X], R[r], or Q[r]) affected by the rule is shown within the enclosing oval. For simplicity, only one node is shown to hold results.

```
$K_1 := x := 0$;
forall the $A \sqsubseteq B \in \mathcal{O}_1$ do
    UN ! update($B_U, A$) ? $x$;
    $K_1 += x$;
```

Algorithm 1. R1: $A \sqsubseteq B \Rightarrow U[B] \sqsubseteq U[A]$

```
$K_2 := x := 0$;
forall the $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B \in \mathcal{O}_2$
do
    UN ! $\sqcap(B_U, \{A_1, \ldots, A_n\})$ ? $x$;
    $K_2 += x$;
```

Algorithm 2. R2: $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B \Rightarrow U[B] \cup= U[A_1] \cap \cdots \cap U[A_n]$

Ontology partitioning should be done in such a way, so as to reduce inter-node communication. By following the described partitioning strategy, this goal is achieved since most of the data required for the rule application is available locally on each node. Other partitioning strategies such as MapReduce based data partitioning where spatial locality is followed (data in contiguous locations are assigned to one mapper) and hash partitioning (axiom key is hashed) did not yield good results [15].

Rule Processes. This section presents the bodies of each of the rules of Table 1. These bodies are wrapped and repeatedly executed by the rule processes; this wrapper code is discussed in the termination section further below.

The service process UN is described as Algorithm 9, and RN as Algorithm 10, further below. Note that, there can be any number of processes of a particular type (R1, ..., R8, UN, RN). In all the algorithms, immediately following the **forall the** keywords is the retrieval of axioms, discussed further below. Given a key such as $(Y, r)_Q$, it is fairly easy to i) extract individual values from it

```
$K_3 := x := 0;$
forall the $A \sqsubseteq \exists r.B \in \mathcal{O}_3$ do
    $s :=$ timeOf($A_U$);
    UN ! queryTS($A_U, s$) ? $M$;
    forall the $X \in M$ do
        RN ! update($(B, r)_{RY}, X$) ? $x$;
        $K_3 += x;$
```

Algorithm 3. R3: $A \sqsubseteq \exists r.B \Rightarrow R[r] \cup= \{(X, B) \mid X \in U[A]\}$

```
$K_4 := x := 0;$
forall the $\exists r.A \sqsubseteq B \in \mathcal{O}_4$ do
    $s :=$ timeOf($A_U$);
    UN ! queryTS($A_U, s$) ? $M$;
    forall the $Y \in M$ do
        R5 ! new($(Y, r)_Q, B$) ? $x$;
        $K_4 += x;$
```

Algorithm 4. R4: $\exists r.A \sqsubseteq B \Rightarrow Q[r] \cup= \{(Y, B) \mid Y \in U[A]\}$

```
$K_5 := x := 0;$
on R4 ? new($(Y, r)_Q, B$) do
{ R4 ! ($Q[(Y, r)_Q] \cup= \{B\}$)#;
$s :=$ timeOf($(Y, r)_{RY}$);
RN ! queryTS($(Y, r)_{RY}, s$) ? $T$;
forall the $X \in T$ do
    UN ! update($B_U, X$) ? $x$;
    $K_5 += x;$
};
on RN ? rpair($(Y, r)_{RY}, X$) do
{ $s :=$ timeOf($(Y, r)_Q$);
$T :=$ range($Q[(Y, r)_Q], s, \infty$);
forall the $B \in T$ do
    UN ! update($B_U, X$) ? $x$;
    $K_5 += x;$
}
```

Algorithm 5. R5: $(X, Y) \in R[r] \wedge (Y, B) \in Q[r] \Rightarrow U[B] \cup= \{X\}$

```
$K_6 := x := 0;$
on RN ? yxpair($Y_{R6}, X$) do
{ UN ! isMember($\bot_U, Y_{R6}$) ? $b$;
  if $b$ then
    UN ! update($\bot_U, X$) ? $x$;
    $K_6 += x;$
}
```

Algorithm 6. R6: $X \sqsubseteq \exists r.Y \Rightarrow U[\bot] \cup= \{X \mid Y \in U[\bot]\}$

```
$K_7 := x := 0;$
on RN ? rpair($(Y, r)_{RY}, X$) do
 forall the $s$   (with $r \sqsubseteq s \in \mathcal{O}_7$)
do
    RN ! update($(Y, s)_{RY}, X$) ? $x$;
    $K_7 += x;$
```

Algorithm 7. R7: $r \sqsubseteq s \Rightarrow R[s] \cup= R[r]$

(such as Y and r) and ii) convert to key of different type but same values, such as $(Y, r)_{RY}$. This conversion, though not explicitly stated in all the algorithms listed here, is implicitly assumed.

Algorithms 1 and 2 follow directly from rules R1 and R2 respectively. Here (and in subsequently described algorithms), keys such as B_U correspond to those listed in Table 1; see also the discussion of axiom retrieval further below. K_1 (and more generally the K_i in subsequently described algorithms) are used for termination handling, as detailed towards the end of this section.

In Algorithms 3, 4 and 5, timeOf(X) returns the access timestamp up to which the values of the key A_U have been read previously. Only the subsequently added values are considered.

In the first **on** statement of Algorithm 5, the rule process R5 receives values for $(Y, r)_Q$ and B from R4. The expression R4 **!** $(Q[(Y, r)_Q] \cup= \{B\})\#$ shall mean that $\{B\}$ is added to $Q[(Y, r)_Q]$ and that either 1 or 0 (the latter if B was already in $Q[(Y, r)_Q]$) is returned to R4. In the second **on** statement, R5 receives values for $(Y, r)_{RY}$ and X from RN. R5 gets triggered either when an axiom $\exists r.Y \sqsubseteq B$ is newly generated by R4 or a new (X, Y) is added to $R[r]$,

```
$K_8 := x = 0$;
on RN ? rpair($(Y, r)_{RY}, X$) do {
  forall the $s, t$ (with $r \circ s \sqsubseteq t \in \mathcal{O}_8$) do
    RN ! queryX($(Y, s)_{RX}$) ? $T$;
    forall the $Z \in T$ do
      RN ! update($(Z, t)_{RY}, X$) ? $x$;
      $K_8$ += $x$;
  forall the $s, t$ (with $s \circ r \sqsubseteq t \in \mathcal{O}_8$) do
    RN ! queryY($(X, s)_{RY}$) ? $T$;
    forall the $Z \in T$ do
      RN ! update($(Y, t)_{RY}, Z$) ? $x$;
      $K_8$ += $x$;
}
on RN ? isOnLHS2($s_{R8b}$) do { $b$ := exists($s_{R8b}$); RN ! $b$ };
```

Algorithm 8. R8: $r \circ s \sqsubseteq t \Rightarrow R[t] \ \cup\!= \ \{(X, Z) \mid (X, Y) \in R[r], (Y, Z) \in R[s]\}$

```
on pid ? queryTS($X_U$, ts) do { $T$ := range($U[X]$, ts, $\infty$); pid ! $T$ };
on pid ? update($X_U, \{A_1, \dots, A_n\}$) do pid ! $(U[X] \ \cup\!= \ \{A_1, \dots, A_n\})\#$;
on R6 ? isMember($X_U, Y$) do R6 ! ($Y \in U[X]$);
on R2 ? $\sqcap(B_U, \{A_1, \dots, A_n\})$ do R2 ! $(U[B] \ \cup\!= \ U[A_1] \cap \dots \cap U[A_n])\#$;
```

Algorithm 9. Process UN maintains $U[X]$, for all X.

which is what these two **on** statements represent. range($Q[(Y, r)_Q], s, \infty$) is a range operation on the set $Q[(Y, r)_Q]$ in which elements starting at timestamp s and going up to the maximum available timestamp are returned. The $Q[r]$ sets, for all roles r, are maintained by rule R5 since it is the only rule process that uses them.

In Algorithm 6 for rule process R6, a set membership request is made to UN which returns a boolean value that is stored in b. Algorithm 7 straightforwardly follows from rule R7.

In Algorithm 8 for rule process R8, whenever a new role pair (X, Y) is added to $R[r]$, it is checked whether this particular role r is part of any role chain axiom, say $p \circ q \sqsubseteq t$. The two possible cases are i) r equals p or ii) r equals q. Based on the case, the corresponding matching role pair is retrieved from RN.

Service Processes UN and RN. Each $U[X]$ is a set and the process UN handles the operations over each of the $U[X]$, for any X. There can be several such UN processes which allows them to share the load. UN associates with the elements e of set $U[X]$ a timestamp indicating when e was added to that set.

UN handles four kinds of requests, see Algorithm 9 – the first two from any arbitrary process (here named pid), the third one from R6 and the fourth from R2. The expression $(U[X] \ \cup\!= \ setS)\#$ stands for updating $U[X]$ and returning the number of new items added. The first type is a request from a process named pid asking for a range of elements newly added to $U[X]$ since its last such request made at time ts. It is the responsibility of the client to keep track of the previous

```
on R5 ? queryTS($(Y,r)_{RY}$, ts) do {$T :=$ range($R[(Y,r)_{RY}]$, ts, $\infty$); R5 ! $T$};
on R8 ? queryX($(X,r)_{RX}$) do R8 ! ($R[(X,r)_{RX}]$);
on R8 ? queryY($(Y,r)_{RY}$) do R8 ! ($R[(Y,r)_{RY}]$);
on pid ? update($(Y,r)_{RY}, X$) do {
    pid ! ($R[(Y,r)_{RY}] \cup= \{X\}$)#;
    R5 ! rpair($(Y,r)_{RY}, X$);
    R6 ! yxpair($Y_{R6}, X$);
    R7 ! rpair($(Y,r)_{RY}, X$);
    R8 ! rpair($(Y,r)_{RY}, X$);
    R8 ! isOnLHS2($r_{R8b}$) ? $b$;
    if $b$ then
        $R[(X,r)_{RX}] \cup= \{Y\}$;
}
```

Algorithm 10. Node RN maintains $R[r]$ sets

timestamp up to which it has read from a particular $U[X]$. The second one is a request of the form update(X_U, D) from pid. This updates U as in $U[X] \cup= D$. Elements of D are added to $U[X]$. The size increase of $U[X]$ is replied back. The third one is a membership request from R6 asking whether a particular element Y is in $U[X]$. A true or false value is given as a response. The fourth is a request from R2 to retrieve the intersection of a group of $U[A_1], \ldots, U[A_n]$.

Analogous to UN, there is an RN process that handles operations over each of the $R[r]$, for any role r, and there can be several such RN processes sharing the load. RN handles four kinds of requests, see Algorithm 10, with most of them similar to the requests handled by UN. The time stamp ts is sent in by the requester. Whenever RN receives an update message with a new role pair $((Y,r)_{RY}, X)$, it notifies the processes (R5, R6, R7, R8) that depend on $R[r]$ values. A new role pair is duplicated on the rule process R8 for further processing. This is done because it is more efficient than separate retrieval of the right role pair using a key. However, this duplication is not required in all cases: If, for a particular role r, this r does *not* appear in the second position of the chain, (e.g., in the position of q as in $p \circ q \sqsubseteq t$), then this particular $R[r]$ is *not* duplicated.

The expression $(R[(Y,r)_{RY}] \cup= \{X\})\#$ stands for updating $(R[(Y,r)_{RY}]$ and returning the number of new items, zero or one, added.

Retrieval of Axioms from the Key Value Store. We use key-value stores [5] to keep the eight parts of the ontology including the $U[X]$, the $R[r]$ and the $Q[r]$, for all concepts X and roles r. Each of these is maintained by separate service processes. The O_i processes are co-located with the R_i rule processes. We retrieve axioms from the O_i services in the **forall the ... do** statements.

Concepts and roles are mnemonic strings of the ontology and we encode them as integers. E.g., 032560 represents a concept (indicated by the last 0) whose ID is 256. The length of the ID is given in the first two positions (03 in this case).

Table 1 shows the keys and their corresponding values for axioms in the ontology. Axioms have a left hand side and a right hand side with respect to $\sqsubseteq$. In most cases, the left hand sides becomes the key and right hand side the

```
repeat
    $K_i$ := apply Ri on $\mathcal{O}_i$ once;
    broadcast($K_i$);
    nUpdates := barrier-sum-of $K_i$;
until nUpdates = 0;
```

Algorithm 11. Wrapper for Ri

```
repeat
    nUpdates := barrier-sum-of $K_i$;
until nUpdates = 0;
```

Algorithm 12. Wrapper for UN and RN

value, both encoded as unsigned 64-bit integers. The paired expressions yield an integer from which the paired items can be peeled off. The hash of the concepts is used in encoding them as keys.

The choice of key is not straightforward. For example, for axioms of type $A \sqsubseteq \exists r.B$ (R3), making r as the key would lead to load imbalance since there are generally only a few roles in an ontology and comparatively many axioms of type $A \sqsubseteq \exists r.B$. On the other hand, making A as key leads to better load distribution, thus allowing several machines to work on $R[r]$.

R8 gets triggered when there is a change to either $R[r]$ or $R[s]$. In order to retrieve the exact match, i.e., given (X, Y) of $R[r]$, get (Y, Z) of $R[s]$ or vice versa, the $R[r]$ sets, for any r, have two keys $(Y, r)_{RY}$ and $(X, r)_{RX}$. The $R[r]$ sets are selectively duplicated. For the same reason, there are two keys for the role chain axioms as well.

Termination. Algorithm 11 invokes the rule process Ri on the axioms in $\mathcal{O}_i$ once i.e., Ri is applied on the axioms one time and the updates made to the $U[X]$ and $R[r]$ sets are collected in K_i (this could be 0). Notice that a K_i is associated with each Ri in Algorithms 1–3. This value is broadcast to all the other rule processes. Then it waits for similar update messages to be received from other rule processes. Barrier synchronization [1] is used in waiting for K_i from all Ri (indicated by the barrier-sum statement). If no rule process made an update, they quit; otherwise, they continue with another iteration. The same termination condition is used for processes handling $U[X]$ and $R[r]$ sets (Algorithm 12). Algorithms 11 and 12 act as wrappers around the other processes Ri, UN, RN.

This termination condition is easy to check on a single machine. But in a distributed system, termination is no longer obvious. For example, just when the process working on rule R1 is done and quits, the next moment, a process working on rule R5 might add a new B to $U[X]$. Although barrier synchronization simplifies the termination detection, it also makes several nodes wait idly. This idleness is reduced in our system using a work stealing mechanism, which is detailed in Sect. 4.

4 Optimizations

We discuss some of the efficiency optimizations we have realized in our approach.

U[X] instead of S[X]. $S[X]$ defined as $A \in S[X]$ iff $X \sqsubseteq A$ is used in the original formulation of the algorithm in [2]. We recast this as $U[X]$ defined as $A \in U[X]$ iff $A \sqsubseteq X$. Use of $U[X]$ instead of $S[X]$ makes the check $A \in S[X]$, which is required in several rules, a single read call, and thus significantly more efficient.

For example, assume that there are five concepts in the ontology, K, L, M, N and P. Suppose $K \sqcap L \sqcap M \sqsubseteq N \in \mathcal{O}$. During some iteration of the classification assume $S(K) = \{K, L, N, \top\}$, $S(L) = \{L, P, M, \top\}$, $S(M) = \{M, N, K, \top\}$, $S(N) = \{N, \top\}$, and $S(P) = \{P, K, L, M, \top\}$. Now, according to rule R2 in [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, we 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 is represented as $U[N] \cup= \{P\}$. In large ontologies, the number of concepts could be in the millions, but the number of conjuncts in axioms like $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$ would be very low. So the performance is better by using $U[X]$ since set intersection needs to be performed only on a very small number of sets.

Rule Dependencies. Say rule R3 just finished processing axiom $\alpha = A \sqsubseteq \exists r.B$. If none of R1, R2, R5 or R6 make any changes to $U[A]$, R3 need not be triggered again to consider α. If and when R3 gets triggered again, it resumes from entries in $U[A]$ with a later timestamp. Thus, we reduce the number of axioms to work on in subsequent iterations.

Dynamic Load Balancing. Processing time for each of the rules, R1 to R8, varies due to the number and type of axioms. This can lead to improper load balancing where there are busy and idle nodes. We apply the well known work stealing mechanism [11], where idle nodes take (steal) work from busy nodes, thus reducing their load. Although this is a well known idea, to the best of our knowledge, there is no freely available distributed work stealing library. Although work stealing increases the communication cost, performance improvement outweighs it.

5 Evaluation

We believe that it is possible to distribute computation of the completion of OWL EL ontologies in such a way that the distributed approach ...

[**(Claim 1)**] scales to very large ontologies to finish the classification task and
[**(Claim 2)**] shows reasonable speedup in the number of nodes.

We verified these claims by implementing a prototype in Java, called `DistEL`, downloadable from http://github.com/raghavam/DistEL. We used Redis[2], a

[2] http://redis.io.

Table 2. Number of axioms, before and after classification, in ontologies.

	GO	SNOMED	SNOMEDx2	SNOMEDx3	SNOMEDx5	Traffic
Before	87,137	1,038,481	2,076,962	3,115,443	5,192,405	7,151,328
After	868,996	14,796,555	29,593,106	44,389,657	73,982,759	21,840,440

Table 3. Classification times in seconds.

Ontology	ELK	jCEL	Snorocket	Pellet	HermiT	FaCT++
GO	23.5	57.4	40.3	231.4	91.7	367.89
SNOMED	31.8	126.6	52.34	620.46	1273.7	1350.5
SNOMEDx2	77.3	OOM[a]	OOM[a]	OOM[a]	OOM[a]	OOM[a]
SNOMEDx3	OOM[a]	OOM[a]	OOM[a]	OOM[a]	OOM[a]	OOM[a]
SNOMEDx5	OOM[a]	OOM[a]	OOM[a]	OOM[a]	OOM[a]	OOM[a]
Traffic	OOM[b]	OOM[c]	OOM[c]	OOM[b]	OOM[b]	OOM[c]

OOM[a]: reasoner runs out of memory.
OOM[b]: reasoner runs out of memory during incremental classification.
OOM[c]: ontology too big for OWL API to load in memory.

key-value store, as our database. Redis was selected because it provides excellent read/write speed along with built-in support for set operations, database sharding, transactions and server-side scripting.

Since one of the use cases is *streaming* traffic data, `DistEL` also has support for incremental classification. It is *inherently supported*, since, in each iteration of the classification procedure, only the newly added axioms are considered and appropriate rules are applied.

We used Amazon's Elastic Cloud Compute (EC2) to run our experiments. Specifically, we used `m3.xlarge` instances which have 4 cores, 15 GB RAM and SSD hard disk. 5 GB was given to the JVM on each node, for all the experiments. These settings and the m3.xlarge instances were selected so as to evaluate our system on a cluster of machines with commodity hardware.

Our test data (see Table 2) comprises of biomedical ontologies GO,[3] SNOMED CT[4] and traffic data of the city of Dublin, Ireland.[5] We also duplicated 2x, 3x and 5x copies of SNOMED CT in order to test the scalability.

Traffic data reasoning is used in the diagnosis and prediction of road traffic congestions [9,10]. These tasks depend on (i) classifying any new individual from the ontology stream, and (ii) identifying their causal relationships and correlation with other streams such as city events. There is no bound on the

[3] http://code.google.com/p/elk-reasoner/wiki/TestOntologies.

[4] http://www.ihtsdo.org.

[5] Raw data of the traffic ontology is from http://dublinked.ie/datastore/datasets/dataset-215.php. This data is converted to $\mathcal{EL}^{++}$ ABox statements as described in [10]. The TBox statements (base ontology), along with two samples of ABox statements, are available from http://www.dropbox.com/sh/9jnutinqjl88heu/AAAi-5ot8A5fStz69Bd0VyGCa.

Table 4. Classification time (in seconds) of `DistEL`

Ontology	8 nodes	16 nodes	24 nodes	32 nodes	64 nodes
GO	134.49	114.66	109.46	156.04	137.31
SNOMED	544.38	435.79	407.38	386.00	444.19
SNOMEDx2	954.17	750.81	717.41	673.08	799.07
SNOMEDx3	1362.88	1007.16	960.46	928.41	1051.80
SNOMEDx5	2182.16	1537.63	1489.34	1445.30	1799.13
Traffic	60004.54	41729.54	39719.84	38696.48	34200.17

number of axioms since it is a continuous stream of traffic data. In this scenario, existing reasoners were not able to cope with the increasing velocity and volume of data. Here, we considered traffic data of *only one single day*. Data is collected every 20 seconds and we have 1441 such bursts.

Results. Table 3 has the classification times for ELK 0.4.1, jCEL 0.19.1, Snorocket 2.4.3, Pellet 2.3.0, HermiT 1.3.8 and FaCT++ 1.6.2. All the reasoners are invoked through the OWL API and ontology loading time is excluded wherever applicable.

All the reasoners ran out of memory on the SNOMEDx3, SNOMEDx5 and Traffic. On traffic data, incremental classification has been used by the reasoners that support it (ELK, Pellet, HermiT). This experiment with single machine reasoners demonstrates that a scalable solution is required to handle large ontologies.

Table 4 shows the classification times of our system as we added nodes. The cluster size need not be in multiples of 8. `DistEL` is able to classify all the ontologies including the largest one having close to 74 million axioms. This validates Claim 1 of our hypothesis.

Table 6 shows the speedup achieved by `DistEL` on SNOMED CT with increasing number of nodes. As can be seen, there is a steady increase in the speedup with increase in the number of nodes. This validates Claim 2 of our hypothesis.

Excluding GO (a small ontology), for all the other large ontologies, classification time decreases as we increase the number of nodes. On 64 nodes, we notice an increase in the runtime for all but the largest of the ontologies. This indicates that beyond a point, the advantages of the distributed approach are overshadowed by the distribution and communication overhead. However, this is not the case for largest ontology, traffic data. We believe this is also due to the axiom composition in traffic data. 75 % of traffic data axioms are in the form of $A \sqsubseteq \exists r.B$ (R3). The output of R3 serves as input to R5, R6, R7 and R8 i.e., 63 % of nodes are always busy, i.e. there are more busy nodes than idle nodes. This is not the case as such for the other ontologies.

Table 5 shows the memory (RAM) taken by Redis in MB on each of the 8 nodes for traffic data. In this case, only one node is used to collect the results

Table 5. Memory taken by `redis` on each node for traffic data

Node	MB
R1	186.72
R2	0.81
R3	257.47
R4	0.79
R5	1970
R6	380.61
R7	0.79
R8	1470.00
Result	654.53
Total	4921.72

Table 6. Speedup achieved by `DistEL` on SNOMED CT

Nodes	Runtime	Speedup
8	544.38	1.00
16	435.79	1.24
24	407.38	1.33
32	386.00	1.41
64	444.19	1.22

Table 7. Speed (in seconds) for simple read, write operations of 1,000,000 items using RAM and `redis`

Operation	RAM	`redis`
Read	0.0861	3.719
Write	0.1833	4.688

Table 8. Speedup achieved by ELK, with all the threads on one 8-core machine, on SNOMED CT

Threads	Runtime	Speedup
1	31.80	1.00
2	19.37	1.64
3	16.29	1.95
4	14.91	2.13
5	13.99	2.27
6	14.16	2.24
7	13.17	2.41
8	13.36	2.38

($U[X]$ sets). $R[r]$ sets are spread across other nodes. As can be seen, each node takes very little memory. But on single machine reasoners, this quickly adds up for large ontologies and current reasoners hit their limit in terms of memory (see Table 3) and computational power.

Discussion. We believe `DistEL` is the first distributed reasoner for EL ontologies and so we cannot do a like-for-like comparison. At the risk of being skewed, the following are our observations in comparison to ELK, which is the fastest reasoner among the ones we tested on (see Table 3).

Table 8 shows the speedup of ELK on SNOMED on an 8 core machine. For `DistEL`, 8 nodes was the starting point. Considering that ELK is a shared memory system with all the threads on one machine, the speedup achieved by `DistEL` (Table 6) is very reasonable in comparison. On this basis, we can say that our design and optimization decisions (Sects. 3 and 4) are justified.

`DistEL` on 8 nodes for SNOMED takes 544 seconds whereas ELK takes 32 seconds. Classification is not "embarrassingly parallel", so linear speedup cannot be achieved. Since axioms are distributed across many nodes, communication is necessary. Another contributing factor is the mismatch in the speed of in-memory and Redis operations (Table 7). This is a simple experiment where 1 million integers are read and written to a Java HashMap. Similar operations were performed on a Redis hash data structure.[6] Although this is a rather simple experiment, the difference in read/write speeds in the case of RAM and Redis is quite obvious.

[6] The code used for this experiment is available at https://gist.github.com/raghavam/2be48a98cae31c418678.

These experiments suggest that a distributed approach should be used only on very large ontologies where the size/complexity of ontologies simply overwhelms current reasoners. Thus a distributed approach has potential benefits which are quite complementary to single machine reasoners.

6 Related Work

There is very little work implemented, evaluated and published on distributed approaches to OWL 2 EL reasoning. Three approaches to distributed reasoning were tried in [15]. Among them, two approaches – MapReduce [16] and a distributed queue version of the corresponding sequential algorithm from [4] turned out to be inefficient. In the MapReduce approach, axioms are reassigned to the machines in the cluster in each iteration. Communication between mappers and reducers cannot be finely controlled and the sort phase is not required here. In the distributed queue approach, distribution of axioms in the cluster happens randomly and hence batch processing of reads/writes from/to the database cannot be done unlike in the approach presented here. The work discussed here is an extension of the most promising one among the three approaches. Initial results of this approach were presented in [14]. Our current work expands on this in several ways, such as, support for nominals, incremental reasoning, static and dynamic load balancing, its application and evaluation over traffic data.

A distributed resolution technique for $\mathcal{EL}^+$ classification is presented in [18] without evaluation. Though not distributed, parallelization of OWL 2 EL classification has been studied in [8,17]. Classifying EL ontologies on a single machine using a database has been tried in [6].

7 Conclusion

We described `DistEL`, an open source distributed reasoner and presented a traffic data application where ontologies are generated from streaming data. We show that existing reasoners were not able to classify traffic data and other large ontologies. Our system on the other hand handles these large ontologies and shows good speedup with increase in the number of machines in the cluster.

Ontologies continue to grow and to hope to keep their representations in the main memory of single machines, no matter how powerful and expensive, is hardly realistic. Large farms of commodity inexpensive machines will push the field of ontology reasoning.

Next, we plan to further explore approaches to efficiently manage communication overhead, including other ontology partitioning strategies as well as alternate classification approaches and rule sets such as the one from ELK. We also plan to do performance modeling and fine-grained analysis on larger datasets, with higher number of nodes in the cluster. Alternatives to the usage of Redis including developing custom storage and data structure solutions can also be looked into.

Acknowledgement. The first two authors acknowledge support by the National Science Foundation under award 1017225 "III: Small: TROn – Tractable Reasoning with Ontologies."

References

1. Andrews, G.R.: Concurrent Programming: Principles and Practice. Benjamin/Cummings Publishing Company (1991)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005, pp. 364–369. AAAI (2005)
3. Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope further. In: Proceedings of the OWLED DC Workshop on OWL (2008)
4. 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)
5. Cattell, R.: Scalable SQL and NoSQL data stores. ACM SIGMOD Record **39**(4), 12–27 (2011)
6. Delaitre, V., Kazakov, Y.: Classifying ELH ontologies in SQL databases. In: Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23–24, 2009. CEUR Workshop Proceedings, vol. 529 (2009). CEUR-WS.org
7. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of semantic web technologies. Chapman & Hall/CRC (2010)
8. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of EL ontologies. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 305–320. Springer, Heidelberg (2011)
9. Lécué, F., Schumann, A., Sbodio, M.L.: Applying semantic web technologies for diagnosing road traffic congestions. In: Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A., Blomqvist, E. (eds.) ISWC 2012, Part II. LNCS, vol. 7650, pp. 114–130. Springer, Heidelberg (2012)
10. Lécué, F., Tucker, R., Bicer, V., Tommasi, P., Tallevi-Diotallevi, S., Sbodio, M.: Predicting severity of road traffic congestion using semantic web technologies. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8465, pp. 611–627. Springer, Heidelberg (2014)
11. Lifflander, J., Krishnamoorthy, S., Kalé, L.V.: Work stealing and persistence-based load balancers for iterative overdecomposed applications. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2012, pp. 137–148. ACM, Delft, Netherlands (2012)
12. Ma, Y., Syamsiyah, A.: A hybrid approach to learn description logic ontology from texts. In: Posters & Demonstrations Track of the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014. CEUR Workshop Proceedings, vol. 1272, pp. 421–424 (2014)
13. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web ontology language profiles. In: W3C Recommendation (2012). Available at http://www.w3.org/TR/owl2-profiles/

14. Mutharaju, R., Hitzler, P., Mateti, P.: DistEL: A distributed EL+ ontology classifier. In: Liebig, T., Fokoue, A. (eds.) Proceedings of the 9th International Workshop on Scalable Semantic Web Knowledge Base Systems, Sydney, Australia. CEUR Workshop Proceedings, vol. 1046, pp. 17–32 (2013). CEUR-WS.org
15. Mutharaju, R., Hitzler, P., Mateti, P.: Distributed OWL EL reasoning: the story so far. In: Proceedings of the 10th International Workshop on Scalable Semantic Web Knowledge Base Systems, Riva Del Garda, Italy. CEUR Workshop Proceedings, vol. 1261, pp. 61–76 (2014). CEUR-WS.org
16. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce algorithm for EL+. In: Haarslev, V., Toman, D., Weddell, G. (eds.) Proceedings of the 23rd International Workshop on Description Logics (DL2010), Waterloo, Canada. CEUR Workshop Proceedings, vol. 573, pp. 464–485 (2010). CEUR-WS.org
17. Ren, Y., Pan, J.Z., Lee, K.: Parallel ABox reasoning of EL ontologies. In: Pan, J.Z., Chen, H., Kim, H.-G., Li, J., Wu, Z., Horrocks, I., Mizoguchi, R., Wu, Z. (eds.) JIST 2011. LNCS, vol. 7185, pp. 17–32. Springer, Heidelberg (2012)
18. Schlicht, A., Stuckenschmidt, H.: MapResolve. In: Rudolph, S., Gutierrez, C. (eds.) RR 2011. LNCS, vol. 6902, pp. 294–299. Springer, Heidelberg (2011)