Developing a Distributed Reasoner for the Semantic Web

Raghava Mutharaju, Prabhaker Mateti, and Pascal Hitzler

Wright State University, OH, USA.
{mutharaju.2, prabhaker.mateti, pascal.hitzler}@wright.edu

Abstract. OWL 2 EL is one of the tractable profiles of the Web Ontology Language (OWL) which has been standardized by the W3C. OWL 2 EL provides sufficient expressivity to model large biomedical ontologies as well streaming traffic data. Automated generation of ontologies from streaming data and text can lead to very large ontologies. There is a need to develop scalable reasoning approaches which scale with the size of the ontologies. We briefly describe our distributed reasoner, DistEL along with our experience and lessons learned during its development.

1 Introduction

The Web Ontology Language (OWL) is a knowledge representation language which has been standardized by the World Wide Web Consortium (W3C). The current version, OWL 2, introduced three tractable profiles: OWL 2 EL, OWL 2 QL and OWL 2 RL [6]. They vary in the modeling features that they offer. Several large biomedical ontologies are in OWL 2 EL. Apart from biomedical domain, the expressivity offered by OWL 2 EL is sufficient to model streaming data traffic sensors [4] where automated generation of axioms is possible. Generating axioms automatically from sensors and text [2] can lead to very large ontologies. A reasoner infers logical consequences from the axioms of an ontology. As the ontologies grow in size, reasoners should also be able to scale correspondingly. This is not possible for existing reasoners since they are limited by the memory and computational resources on a single machine.

The software described in this paper, DistEL, is a distributed reasoner that can scale up to large ontologies. Source code of DistEL is available at `https://github.com/raghavam/DistEL`. DistEL in particular supports the reasoning task called *classification* which is the computation of complete subsumption hierarchy involving all the concepts in the ontology. It follows the peer-to-peer implementation model and works on ontologies in $\mathcal{EL}^+$ which is the underlying description logic fragment of OWL 2 EL. More details are in [8] and some of the other approaches we tried for distributed reasoning are mentioned in [7]. DistEL is built using Java and a key-value store called Redis[1]. Two processes run on each node of the cluster: a Java reasoning process and a Redis database server process. The Java process can read from and write to the database on other

[1] `http://redis.io`

nodes as well. The Java process on all the nodes work co-operatively to achieve the common goal of reasoning over the axioms which are distributed over the cluster.

2 Software Usage

DistEL can be used to classify any $\mathcal{EL}^+$ ontology, but is effective on very large ontologies where the number of axioms are in the millions and more. Detailed instructions are given at `https://github.com/raghavam/DistEL`. Linux shell scripts are provided and can be used to run the steps described briefly here.

1. Enable passwordless ssh to all the machines in the cluster.
2. Specify the cluster information in the configuration file.
3. Normalize the ontology if not already normalized and load the axioms into the database present on each node of the cluster.
4. Classify the loaded ontology. Depending on the ontology, it takes several iterations to classify it. Final results are collected in the node specified in the configuration file.

3 Highlights

Some of the highlights during the development of DistEL in terms of features, technologies, optimizations and the implementation effort required are described here.

3.1 Database

Classification algorithm consists of applying a set of rules to the axioms iteratively until no new inferences can be computed. Rules almost exclusively involve set operations. Based on this and the distributed nature of the implementation, following were the requirements from the database.

- *Very good read and write speed.* It is not the case here that there are very less number of reads compared to writes or vice-versa. So, both read and write speeds are important.
- *Built-in set operations.* If there is no built-in support, then client (Java reasoning process) should fetch the required data, perform the set operations and write it back. This is inefficient especially if the data has to be fetched from a non-local database.
- *Transaction support.* In a distributed setup, there is a possibility that several requests can be sent to the same database by different processes. Atomicity of some operations is also required.
- *Random reads*
- *Server-side scripting.* Operations on large data is efficient if done at the server-side rather than fetching the data to the client side.

- *Batch processing.* Network round-trip time on each message/request can be avoided if several operations can be batched together.
- *Scalability.* As the data grows, the reads and writes should scale accordingly.
- *Good documentation and community support.*
- *Support for Java.* Since rest of the application is in Java, a good Java interface to the database is needed.

An in-memory key-value store provides excellent read/write speed. Among the options available, Redis has support for all the mentioned requirements. It is a single thread server that supports set operations, sharding and Lua[2] scripts for server-side scripting.

3.2 Barrier Synchronization

Computing classification is an iterative process and at the end of each iteration, a check is made to determine whether any new inferences have been derived in this iteration. If there are no new additions, process is terminated. But in a distributed setup, checking for termination condition is not straightforward because reasoning process on each node of the cluster needs to know whether any of the reasoning processes on other nodes have derived any new inference. If a new inference has been derived, only in that case will a reasoning process go to the next iteration. Note that, this requires each process to wait for all the other processes to complete their current iteration. This is achieved through barrier synchronization [1] where a software barrier is placed at a certain point. A process that reached this barrier stops and cannot proceed until all the other processes reach the barrier.

Barrier synchronization in DistEL is implemented using a combination of status message broadcast by each reasoning process to all others and the blocking wait feature of Redis. A status message is a simple UPDATE/NO-UPDATE indicating whether a new inference has been derived or not by a specific process. Each process can be made to block until it receives status messages from the reasoning process on all the other nodes.

3.3 Work Stealing

When a process reaches the barrier, instead of waiting, it can help other busy processes. The idle process *steals* a fixed set of axioms from the busy process [5]. This leads to better (dynamic) load balancing and utilization of resources. Axioms on each node are divided into fixed number of pieces called *chunks*. The reasoning process on each node works on one chunk at a time. The idle processes steal one chunk from the most busy process and works on it. After a chunk has been processed, it looks for another chunk that can be stolen from a busy process.

To the best of our knowledge, there is no freely usable distributed work stealing Java library. So we implemented work stealing in DistEL using Lua

[2] `http://www.lua.org`

scripting and Java. The idle process runs a script against the database of the busy process. The script first checks if there are any chunks available for stealing, and if they are, it adjusts the chunk counter on the busy process and retrieves the chunk for local processing. Note that this script should be run atomically and Redis ensures the atomicity of Lua scripts. After the idle process gets the chunk, it should temporarily implement the same functionality as that of the busy process since the data obtained corresponds to the busy process. After processing this chunk, the idle process looks for another chunk and steals it. This continues until no chunks are available on any of the database processes.

4 Lessons Learned

We briefly describe our experience and lessons learned during the development of DistEL.

- In a distributed system, debugging synchronization and timing issues (race conditions) is hard since it makes the results non-deterministic. Each run would give a different result. While debugging, it is best to consider the smallest available dataset as input. This should be small enough to hand trace the output and possible steps taken during execution. In this case, timing issues can be traced since we already know the order.
- We could not find any unit testing frameworks that can simulate a distributed setup involving a key-value store. Due to this, more time was spent on debugging than on coding. Reasoning related errors can be debugged using the explanation (justification) feature [3] of reasoners such as Pellet[3]. When the expected subsumption relation is given to Pellet and asked for an explanation, it retrieves a set of axioms from the given ontology using which the expected subsumption relation can be inferred. These axioms would be very small in number compared to the total axioms in the ontology. An ontology can be made out of these axioms and should be given as input to our reasoner implementation. This is the smallest sample that produces the reasoning error.
- Standard development practices such as commenting the code, using an IDE (Eclipse), version control system (Github), logging and a build tool (Ant) are extremely useful.
- Java is verbose. Lot of biolerplate code needs to be written to implement even a small functionality. Excluding comments and blank lines, our codebase on Github currently has 11,312 lines of code.
- Tools such as pssh[4] are very useful to run jobs in a distributed environment. It not only deploys the job on each node but also collects the output from each node.

[3] `http://clarkparsia.com/pellet`
[4] `https://code.google.com/p/parallel-ssh`

Acknowledgements: 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. Andrews, G.R.: Concurrent programming: Principles and Practice. Benjamin/Cummings Publishing Company (1991)
2. Cimiano, P.: Ontology Learning and Population from Text: Algorithms, Evaluation and Applications. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
3. Horridge, M., Parsia, B., Sattler, U.: Laconic and Precise Justifications in OWL. In: 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany. pp. 323–338. Springer (2008)
4. Lécué, F., Tucker, R., Bicer, V., Tommasi, P., Tallevi-Diotallevi, S., Sbodio, M.L.: Predicting Severity of Road Traffic Congestion using Semantic Web Technologies. In: Proceedings of the 11th Extended Semantic Web Conference (ESWC2014), Anissaras, Crete, Greece, May 25–May 29, 2014. Springer (2014)
5. 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'12, Delft, Netherlands. pp. 137–148. ACM (2012)
6. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language Profiles. W3C Recommendation (11 December 2012), available at `http://www.w3.org/TR/owl2-profiles/`
7. 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)
8. 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. CEUR-WS.org (2013)