Partitioning OWL Knowledge Bases for Parallel Reasoning

Sambhawa Priya*, Yuanbo Guo†, Michael Spear‡ and Jeff Heflin§
*‡§ Department of Computer Science and Engineering, Lehigh University
19 Memorial Drive West, Bethlehem, PA 18015, USA.
†Microsoft Corporation, USA
Email: *sps210@lehigh.edu, ‡spear@cse.lehigh.edu, §heflin@cse.lehigh.edu, † Yuanbo.Guo@microsoft.com

***Abstract*—The ability to reason over large scale data and return responsive query results is widely seen as a critical step to achieving the Semantic Web vision. We describe an approach for partitioning OWL Lite datasets and then propose a strategy for parallel reasoning about concept instances and role instances on each partition. The partitions are designed such that each can be reasoned on independently to find answers to each query subgoal, and when the results are unioned together, a complete set of results are found for that subgoal. Our partitioning approach has a polynomial worst case time complexity in the size of the knowledge base. In our current implementation, we partition semantic web datasets and execute reasoning tasks on partitioned data in parallel on independent machines. We implement a master-slave architecture that distributes a given query to the slave processes on different machines. All slaves run in parallel, each performing sound and complete reasoning to execute each subgoal of its query on its own set of partitions. As a final step, master joins the results computed by the slaves. We study the impact of our parallel reasoning approach on query performance and show some promising results on LUBM data.**

***Index Terms*—parallel reasoning, OWL Lite, knowledge base, partition, empirical study**

I. Introduction

The Semantic Web is increasingly being used for representation and reasoning of data on the Web. However, there are several challenges to be addressed before the Semantic web vision can be practically realized. One of the challenges the Semantic Web community is facing today is the issue of scalable reasoning that can generate responsive results to complicated queries involving large datasets. Unfortunately, there is a significant gap between sound and complete description logic reasoners and scalable Semantic Web systems. One way to overcome this scalability issue is to split the workload across multiple nodes and perform reasoning in parallel, ideally with minimal communication between nodes.

Splitting the Semantic Web data into independent partitions is hard [1]. It is difficult to guarantee sound and complete reasoning on the split workload. If we split the knowledge base naively across multiple nodes, we would suffer significant communication penalties due to the need to draw inferences whose data is distributed across nodes. Another possibility is to split the knowledge base such that all the triples involving a particular term are sent to a single node so that all inferences involving those triples could be drawn. But this might result in poor load balancing since some terms are more popular than others in accordance to the power law [2], and nodes handling triples for such popular terms will have very high load. We address this issue in our algorithm for partitioning the knowledge base.

OWL, the W3C recommended Web Ontology Language, is one of the key standards for building the Semantic Web. OWL has three species, i.e. **OWL Lite**, **OWL DL** and **OWL Full**, with increasing expressivity and thus increased reasoning complexity. **OWL 2** is an extension and revision of OWL and has 2 major dialects - **OWL 2 DL** and **OWL 2 Full**, more expressive counterparts of OWL DL and OWL Full. OWL 2 specifies 3 profiles - OWL 2 EL, OWL 2 QL and OWL 2 RL, which are subsets of OWL 2 DL. OWL Lite is a subset of OWL DL, hence it is effectively a subset of OWL 2 DL as well. OWL Lite covers most of the features in OWL 2 EL, OWL 2 QL and OWL 2 RL as well as others in none of these languages. OWL Lite and OWL DL logically correspond to expressive Description Logics (DLs) - $\mathcal{SHIF}(\mathcal{D})$ and $\mathcal{SHOIN}(\mathcal{D})$, respectively [3]. An OWL Lite/DL knowledge base consists of a DL **TBox** $\mathcal{T}$ and a DL **ABox** $\mathcal{A}$. The TBox contains axioms about concepts (sets of objects) and roles (binary relations). ABox is a set of assertions about individuals (concept assertions) and their relationships (role assertions). Thus, a TBox is analogous to ontology and the associated ABox contains instance data over that ontology.

Reasoning with OWL is complex. Even OWL Lite has exponential worst-case complexity. Nonetheless, contemporary reasoners such as FaCT++ [4], Racer [5], Pellet [6] and Hermit [7] tend to work well with realistic TBoxes which usually have moderate sizes. All of the above systems are main memory-based systems that are based on optimized implementations of tableaux algorithms. As Horrocks et. al. [8] pointed out, a great challenge for systems to support ABox reasoning is due to the fact that ABoxes can be extremely large in a setting like the Semantic Web. We address this issue by adopting a divide-and-conquer approach of partitioning an OWL ABox into independent smaller pieces that can be processed in parallel while guaranteeing the completeness of reasoning in question when the results are combined.

In this work, we describe an approach for partitioning OWL Lite datasets and then propose a strategy for parallel reasoning about concept instances and role instances on each partition. In our current implementation, we partition semantic web

datasets and execute reasoning tasks on partitioned data in parallel on independent machines. We implement a master-slave architecture where we have a master process that distributes a given query to the slave processes on different machines. A typical query to the knowledge base may involve multiple subgoals of different selectivity, referred to as a conjunctive query. The selectivity of a query subgoal is determined by the size of its resultset; the smaller the resultset, the more selective a query subgoal is and vice versa. A retrieval conjunctive query is the most frequently used query in the Semantic Web [6], and searches the ABox for instances that satisfy a conjunction of concepts and roles. In a parallel reasoning environment, the results of the subgoals of a conjunctive query need to be joined optimally in order to generate responsive results. All slaves run in parallel, each performing sound and complete reasoning to execute each subgoal of its query on its own set of partitions. As a final step, the master joins the results computed by the slaves.

The rest of the paper is organized as follows: in Section II we describe an approach to partition OWL Lite knowledge base into independent partitions. We present our approach for parallel reasoning on the partitioned knowledge base in Section III. In Section IV, we describe our implementation, and we present results of our evaluation on partitioning and reasoning with conjunctive query processing using the LUBM benchmark in Section V. In Section VI we discuss related work, and we conclude with a discussion of future work in section VII.

II. An Approach for Independent ABox Partitioning

In this section, we discuss our approach for determining independent ABox partitions. This section recapitulates the work presented in Guo and Heflin [9] and serves as a foundation for the novel work done in this paper. In our discussion, we ignore datatypes which can be easily supported without loss of generality.

In [10], Amir and McIlraith provide a definition for partitioning with respect to a logical theory. We have specialized the definition for an ABox, as follows:

Definition 1 (ABox Partitioning). *$\{\mathcal{A}_i\}_{i\leq n}$ is a partitioning of ABox $\mathcal{A}$ iff $\mathcal{A} = \cup_i \mathcal{A}_i$*

We focus mainly on inference of concept instances, in the form $a:C$, and the inference of role instances, in the form of $\langle a, b\rangle : R$. Our goal is to work with partitions independently while still guaranteeing complete reasoning for combined results. Following is our definition of independent partitioning:

Definition 2 (Independent ABox Partitioning). *Given an OWL knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, a partitioning of $\mathcal{A}, \{\mathcal{A}_i\}_{i\leq n}$ is an independent partitioning of $\mathcal{A}$ with respect to $\mathcal{T}$ iff for every concept assertion or role assertion φ such that $\mathcal{K} \models \varphi$, there exists $\mathcal{A}_i$ such that $(\mathcal{T}, \mathcal{A}_i) \models \varphi$.*

We assume that there are no assertions in ABox that contain complex concepts, i.e. the assertion φ in Definition 2 is a role or concept assertion whose concept is atomic.

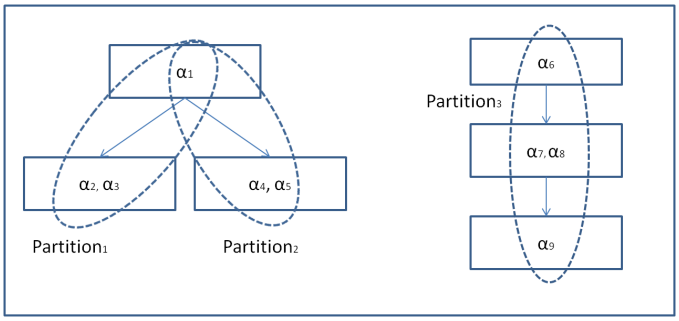

Fig. 1. An Example chunk graph

Guo et. al. [11] have shown that for two OWL Lite/DL knowledge bases, $\mathcal{K}_1 = (\mathcal{T}, \mathcal{A}_1)$ and $\mathcal{K}_2 = (\mathcal{T}, \mathcal{A}_2)$, iff $\mathcal{A}_1$ and $\mathcal{A}_2$ do not contain any individual names in common, then $\mathcal{K}_1$ and $\mathcal{K}_2$ are independent from each other with respect to the inference of concept or role assertions. Thus, we can say that $\mathcal{A}_1$ and $\mathcal{A}_2$ is an independent partitioning of $\mathcal{A}_1 \cup \mathcal{A}_2$.

A. Building Blocks - Chunk and Chunk Graph

In order to build the partitions, we build a chunk graph G for a given ABox A. Chunks are the basic unit of partitions and the structure of the chunk graph G determines partitioning. Each vertex of G is a chunk which is a set of assertions from A. G is a directed graph where two chunks sharing an edge must belong to the same partition. The chunk graph represents relevance between assertions in the ABox in terms of their interdependence on each other for deriving new inferences. The idea is that two assertions which are relevant to each other should be assigned to the same chunk or to two chunks that share an edge. In this way, both assertions will be guaranteed to appear in the same partition.

Thus, the chunk graph serves as the building block for partitions. Fig. 1 shows an example chunk graph and the circles depict its three partitions. For determining partitions, first a partition is created for each chunk that has no outgoing edge and then all the chunks from which this chunk is reachable are added to the partition. There can be duplication of assertions among partitions as depicted in Fig. 1 where $Partition_1$ and $Partition_2$ have assertion α_1 in common.

1) Determining Relevant Assertions: For determining relevant assertions in an ABox such that they depend on each other for deriving a new inference, we utilize the natural deduction style inference rules. Royer and Quantz [12], [13] have developed a general approach for deriving a sound and complete set of inference rules for any given DL. They first translate the DL into FOL according to the method described by Tsarkov et. al. [14] and then identify from the resultant FOL formula necessary and sufficient conditions of provability in the Sequent Calculus. This finally leads to derivation of inference rules for DL. In their work [13], the authors have applied this technique to a very expressive DL which includes intersection, negation, value restriction, existential quantity, role hierarchy, transitive role, inverse role, and number restriction (which subsumes $\mathcal{SHIF}$). The rules they have given

are complete if nominals are ignored which does not affect us since $\mathcal{SHIF}$ does not have such a feature.

Based on their results, we have identified the following set of inference rules for a DL $\mathcal{SHIF}$ knowledge base $(\mathcal{T}, \mathcal{A})$. In these rules, a, b and c are individuals, C, C_1 and C_2 are class (concept) expressions and R, R_1 and R_2 are roles.

TBox Rules

There are around 100 TBox rules for inferring concept and role subsumptions. We refer the readers to [13] for details. Below we use $\mathcal{T} \vdash C_1 \sqsubseteq C_2$ or $\mathcal{T} \vdash R_1 \sqsubseteq R_2$ to denote such inferences.

ABox Rules

For ABoxes rules, we use $\mathcal{A} \vdash \varphi$ as an abbreviation for $(\mathcal{T}, \mathcal{A}) \vdash \varphi$. Also for simplicity, we ignore the cardinality constructs - $\geq$0R, $\geq$ 1R and $\leq$ 0R since they can be translated into $\top$, $\exists R.\top$ and $\forall R.\bot$ respectively.

R1) If $\varphi \in \mathcal{A}$ then $\mathcal{A} \vdash \varphi$.
R2) $\mathcal{A} \vdash a : \top$
R3) If $\mathcal{T} \vdash C_1 \sqsubseteq C_2$ and $\mathcal{A} \vdash a : C_1$ then $\mathcal{A} \vdash a : C_2$.
R4) If $\mathcal{A} \vdash a : C_1, a : C_2$ then $\mathcal{A} \vdash a : C_1 \sqcap C_2$
R5) If $\mathcal{A} \vdash a : \forall R.C, \langle a, b\rangle : R$ then $\mathcal{A} \vdash b : C$
R6) If $\mathcal{A} \vdash \langle a, b\rangle : R, b : C$ then $\mathcal{A} \vdash a : \exists R.C$
R7) If $\mathcal{T} \vdash R_1 \sqsubseteq R_2$ and $\mathcal{A} \vdash \langle a, b\rangle : R_1$ then $\mathcal{A} \vdash \langle a, b\rangle : R_2$
R8) If $\mathcal{A} \vdash \langle a, b\rangle : R$ then $\mathcal{A} \vdash \langle b, a\rangle : R^-$
R9) If $R \in R_+$ (transitive role) and $\mathcal{A} \vdash \langle a, b\rangle : R, \langle b, c\rangle : R$ then $\mathcal{A} \vdash \langle a, c\rangle : R$
R10) If $\mathcal{A} \vdash a :\leq 1R, \langle a, b_1\rangle : R, \langle a, b_2\rangle : R$ then $\mathcal{A} \vdash b_1 = b_2$
R11) If $\mathcal{A} \vdash a = b$ and $a : C$ (resp. $\langle a, c\rangle : R, \langle c, a\rangle : R, a = c$) then $\mathcal{A} \vdash b : C$ (resp. $\langle b, c\rangle : R, \langle c, b\rangle : R, b = c$)

Given their forward chaining style, the above rules may not be suitable for implementing an effective reasoning system. However the advantage is that they provide us with an intuitive guidance for partitioning: *generally, assertions in the antecedent of an inference rule should be placed in the same partition.* However, it would require full backward chaining reasoning on the knowledge base to pinpoint the antecedents of implicitly inferred statements.

In order to avoid determining antecedent assertions for all implicitly inferred statements along with the explicit ones (i.e. reasoning on the entire knowledge base), Guo and Heflin [9] adapted some inference rules for OWL Lite to quickly determine which assertions might infer others. We refer to these new rules as "might-infer"($\vdash_?$) rules.

The might-infer ($\vdash_?$) rules can simplify the process of determining relevant assertions for constructing the chunk graph. We note that the adapted rules are complete but unsound, but this does not sacrifice the correctness of partitioning. The only consequences of unsoundness are that there might arise a case where two irrelevant assertions are required to be in the same chunk; or that arcs may be added between chunks that do not actually contain relevant assertions. We refer the readers to [9] for detailed discussion on the might infer ($\vdash_?$) rules.

2) Constructing the Chunk Graph: We use the might infer ($\vdash_?$) rules mentioned above as a guidance to develop the algorithm for building the chunk graph. We derive new rules called chunk rules (ChunkRule) by combining one or more of the might infer rules. The application of a chunk rule causes some alteration of the chunk graph - it may create a new chunk, merge existing chunks, or add arcs between some chunks. The partitions are determined based on the chunk graph.

When defining chunk rules, we keep in mind the following: 1) If based on $\vdash_?$, an assertion β might affect inference about concept membership of a, then the chunk containing concept assertions of a (denoted by $chunk(a)$) will be reachable from the chunk that either contains or infers β. 2) If a chunk ck depends on concept memberships of a to derive new inference, then ck will be reachable from $chunk(a)$. This is shown in Figure 2.

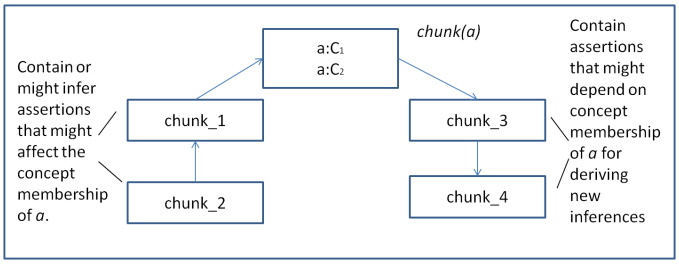

Fig. 2. Illustration of rules for determining reachability between chunks

Before we elaborate on the ChunkRules, we define three properties for roles which will be used in describing ChunkRules:

Definition 3 ($\forall$ - Possible($\mathcal{T}$,R))
1) If there exists a general concept inclusion (GCI) g in $\mathcal{T}$ such that $\forall$ R occurs on right hand side of g, then $\forall$ -Possible ($\mathcal{T}$,R) is true.
2) If $\mathcal{T} \models R \sqsubseteq S$ and $\forall$-Possible($\mathcal{T}$,S) is true, then $\forall$ - Possible($\mathcal{T}$,R) is also true.

Definition 4($\exists - Useful(\mathcal{T}, R)$**)**
1)If there exists a GCI g in $\mathcal{T}$ such that $\exists R$ occurs on the left hand side of g, then $\exists - Useful(\mathcal{T}, R)$ is true.
2) If $\forall$-Possible($\mathcal{T}, R$) is true, then $\exists - Useful(\mathcal{T}, R)$ is also true.
3) If $\mathcal{T} \models R \sqsubseteq S$ and $\exists - Useful(\mathcal{T}, S)$ is true, then $\exists - Useful(\mathcal{T}, R)$ is also true.

Definition 5 ($\leq -Possible(\mathcal{T}, R)$**)**
1) If there exists a GCI g in $\mathcal{T}$ such that $\leq$1R occurs on the right hand side of g, then $\leq -Possible(\mathcal{T}, R)$ is true.
2) If $\mathcal{T} \models \mathcal{R} \sqsubseteq \mathcal{S}$ and $\leq -Possible(\mathcal{T}, S)$ is true, then $\leq -Possible(\mathcal{T}, R)$ is also true.

We now describe the ChunkRules:

ChunkRule 1.(Based on TBox rules on concept and role subsumptions and R2-R4) *Create a chunk for each individual a in ABox $\mathcal{A}$ such that $a : C_{\in \mathcal{A}}$.* For any individual a, $chunk(a) \vdash_? a : C$ for any concept C.

Using this rule, we group all the concept assertions about

a in the ABox into $chunk(a)$ which infers any concept membership of a. If a is untyped, then $chunk(a)$ is initially empty (since OWL Lite requires that all instances be typed, this is only significant if the data is messy).

ChunkRule 2. Create a chunk for each role assertion φ in $\mathcal{A}$. Hereafter, we will use $chunk(\varphi)$ to denote the chunk containing φ. We note that $chunk(\varphi) \vdash_? \varphi$.

ChunkRule 3. (Based on R10) *If* $\leq 1 - Possible(\mathcal{T}, R)$ *and* $ck_1 \vdash_? < a, b >: R$, $ck_2 \vdash_? < a, c >: R$, *then:*
1) Merge ck_1 *and* ck_2 *to form* ck.
2) Draw an arc from $chunk(a)$ *to* ck.
3) From the above two steps, we record that $ck \vdash_? b = c$

ChunkRule 4.(based on R11) *If* $ck \vdash_? b = c$, *then :*
1) Record that $b \approx c$ *(abbreviation for* $\mathcal{A} \vdash_? b = c$*).*
2) Merge $chunk(b)$ *and* $chunk(c)$ *to form chunk* ck'.
3) Draw an arc from ck *to* ck'.
4) For every role assertion φ *involving* b *or* c, *draw an arc from* ck *to* $chunk(\varphi)$.

ChunkRule 5.(Based on R7-R9) *For each* $R \in R_+$,*conduct an individual names based merging on the following set of assertions:*
$\{< a, b > | R_1 \sqsubseteq R$ *or* $R_1 \sqsubseteq R^-\}$ *according to the following rule:*
For any two of the above assertions, $\varphi_1 =< a, b >: R_1$ *and* $\varphi_2 =< c, d >: R_2$, *where* $a \approx c$ *or* $a \approx d$ *or* $b \approx c$ *or* $a \approx d$, *merge* $chunk(\varphi_1)$ *and* $chunk(\varphi_2)$.

ChunkRule 6.(Based on R5) *If* $\forall - Possible(\mathcal{T}, R)$ *and* $ck \vdash_? < a, b >: R$ *then*
1) Draw an arc from $chunk(a)$ *to* $chunk(b)$.
2) Draw an arc from ck *to* $chunk(b)$.

ChunkRule 7.(Based on R6) *If* $\exists - Useful(\mathcal{T}, R)$ *and* $ck \vdash_? < a, b >: R$ *then*
1) Draw an arc from $chunk(b)$ *to* $chunk(a)$.
2) Draw an arc from ck *to* $chunk(a)$.

We apply the above ChunkRules to construct the chunk graph G. We first initialize the chunks using ChunkRule 1 and ChunkRule 2. Then, we repeatedly apply ChunkRules 3, 4 and 5 which lead to creation of new chunks that might infer something new. This may result in generating new antecedents for these ChunkRules and activate another round of application of the rules. Finally, we apply ChunkRules 6 and 7, which result in addition of new arcs on the graph. This algorithm to build the chunk graph always terminates and has a polynomial worst case time complexity in the size of ABox $\mathcal{A}$. We refer the readers to [9] for details on this algorithm.

B. Determining Partitions

After the chunk graph G is built, to determine the partitioning of G becomes fairly straightforward. For every chunk ck that has no outgoing links, we create a partition p = $\cup_i$ ck_i where ck is reachable from ck_i on G (including ck). Effectively, we create partitions by merging every set of chunks that form a connected component of G.

The most complex operation in determining partitions is computing reachability between vertices. In the worst case, if the number of chunks in chunk graph is n and the graph is dense, the algorithm will have a time complexity of O(n^3). However, the graph is typically sparse, and as shown in the later experiments, the run time grows linearly with the problem size.

The above algorithm keeps the resultant partitions as small as possible by allowing chunks to be put in the same partition only when they are required by the chunk graph. In practice, however, we may prefer a restricted number of partitions as long as their sizes do not go beyond a predefined limit. In that case, we perform merging of resultant partitions.

III. Parallel Reasoning on Partitioned ABox

Once the knowledge base has been divided into independent partitions using the strategy described above, we reason with the partitions in parallel on independent machines. We implement a master-slave architecture (see Figure 3) where we have a master process that distributes a given query to the slave processes on different machines. During the bootstrapping phase, the slave processes are assigned a disjoint set of partitions such that the load is balanced as much as possible between them in terms of number of triples. Each slave has a sound and complete DL reasoner. The slaves read their allotted partitions and load RDF models for their partitions in memory. Once a slave has loaded RDF models for its partitions, it signals the master that it is ready to receive queries. Each query is a conjunctive query $(x_1, x_2, ...) \leftarrow A_1$ AND A_2 AND ... where each A_i is a subgoal of the form $\langle x_1, x_2 \rangle : R$ or $x_1 : C$ where R is a role and C is a concept name. x_1, x_2, etc. are distinguished variables and an answer to the query provides bindings for all such variables. The master waits until it receives notifications from all the slaves and then sends the query across to them. Upon receiving the query, the query processing phase begins when the slaves start processing the query on each of their in-memory models. The slaves reason on their respective models simultaneously in order to build the corresponding inference models. They divide the query into its subgoals and each subgoal is independently processed on each inference model. The results for a particular subgoal are stored as a shared set of n-tuples over different models and slaves. Once a slave completes processing a given query on all of its models, it sends a signal to the master indicating its completion of task. The master waits to receive the completion signal from all the slaves and then performs a join on the merged results of each subgoal to obtain complete answers to a given query. In the meantime, the slaves wait to receive another query from their master. After computing the final result of a query, the master sends the next query to process across the slaves. Thus, the process continues while the master keeps receiving queries from the user.

In this approach, partitioning needs to be done only once for every dataset. Once partitioning is done, the slaves load the models for their partitions which also occurs once per dataset. Thus, the time-cost of partitioning and loading can be amortized over several queries. Depending upon the number of slaves available, the total time taken to reason and answer

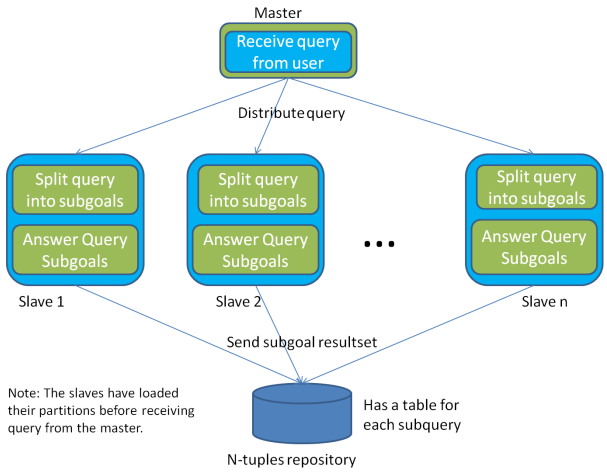

Fig. 3. Master Slave Architecture for Parallel Reasoning on Partitions

a query over a partitioned dataset, which has been loaded in-memory across slaves, can be given by the sum of time taken for reasoning on the partitions in parallel and the time to join the results from different slaves.

IV. Implementation

A. Partitioning

Our algorithm allows for an implementation based on secondary storage so as to accommodate large datasets. We use a MySQL instance to store the input data and the intermediate results - chunks, chunk graph, partitions, $\approx$ and $\vdash_?$ information. One important feature of the partitioning algorithm is that it works on a small subset of input data at a time, for example, concept assertions for the same individual, role assertions for the same role, a single chunk, and so on. This locality is important for the scalability of the system.

As we mentioned before, our algorithm keeps resultant partitions as small as possible. Since our purpose is not to evaluate granularity of partitioning, but to reason on them to evaluate conjunctive queries, we do not generate the smallest partitions. Instead, we let the partitioner merge the partitions by grouping every 1000 small partitions into one merged partition. We found such a grouping to be most optimal for balancing the I/O overhead for the slaves during loading the partitions for parallel reasoning.

B. Parallel Reasoning on Partitions

We use Jena, an open source Semantic Web toolkit for processing the OWL ontologies to create models for partitions. These are loaded by each slave into the main memory. Reasoning on these raw models to generate inference models is done using an open source reasoner, Pellet [6]. The communication between master and slaves is implemented using Java TCP sockets. SPARQL is used as the query language for processing conjunctive queries.

For storing the results of query subgoals computed by the slaves, we use a separate MySQL instance on the master node. We create a table for each subgoal of the conjunctive query where the columns of a table correspond to the variables in the respective subgoal. For a given conjunctive query: $(x, y) \leftarrow \langle x, y \rangle : R \wedge y : C$, we create the tables: **table1:(x,y)** and **table2:(y)** to hold results of the subgoals of the query. In order to populate the database, we use DELAYED inserts, a MySQL extension to standard SQL, that takes inserts from many clients (in our case, slave processes) and clubs them together to write in one block, which is much faster than doing separate individual inserts. The MySQL tables are indexed to generate B-tree indexes on the fields corresponding to join variables (In the conjunctive query example above, we create an index on column y in table1 and table2). This speeds up join processing to compute the result of the conjunctive query.

V. Evaluation

To evaluate the performance benefit of our parallel reasoning system, we performed experiments on a cluster with 17 machines. Each machine has one 4-core (8-thread) 2.93 GHz Xeon processor and 6 GB of RAM. Each machine ran CentOS 6.4 and the Oracle JDK, version 1.7.

A. Evaluation of Partitioning

Partitioning was performed serially on one of the machines. We conducted the experiments on LUBM data to evaluate performance of our partitioner. LUBM is a benchmark for evaluating OWL knowledge base systems focusing mainly on ABox processing. Using the LUBM data generator, we have created test sets with between 6M and 27M triples. The benchmark data includes $someValuesFrom$, $transitiveProperty$, $inverseOf$, and other properties. The benchmark is intended to be a realistic ontology of the university domain.

Table I, II and Fig.4 show the test results. As can be seen, the partitioning system scales well. The triples are found to be evenly distributed across partitions. Prior to merging, about 55% of partitions have only 1 triple (these are triples that do not influence inference) and thus we get the median size of 1000 triples for the merged partitions in Table II. The maximum partition size only grows slightly as the dataset doubles. The minimum partition size is because the last group of partitions that were merged may have fewer than 1000 partitions, and thus may have much fewer triples than most partitions. Fig. 4 shows that the partitioning time is roughly linear with the size of the dataset. As shown in Fig. 1, there can be duplicate assertions among partitions. About 1.5% of assertions in the original dataset were duplicated at least once which resulted in 18% total duplicated triples in the partitioned dataset. Partitioning needs to only occur once per dataset, thus the partition time can be amortized over many queries.

TABLE I
Partitioning statistics I (For merged partitions)

Dataset	Size(# triples)	Time(hh:mm:ss)	# partitions
LUBM(50)	6.8M	01:23:25	5178
LUBM(100)	13.8M	02:51:00	10421
LUBM(200)	27.6M	03:47:22	20760

TABLE II
PARTITIONING STATISTICS II (FOR MERGED PARTITIONS)

Dataset	Average Partition Size (# triples)	Min Partition Size (# triples)	Max Partition Size (# triples)	Median Partition Size (# triples)
LUBM(50)	1593	67	6721	1000
LUBM(100)	1594	159	6747	1000
LUBM(200)	1595	279	6755	1000

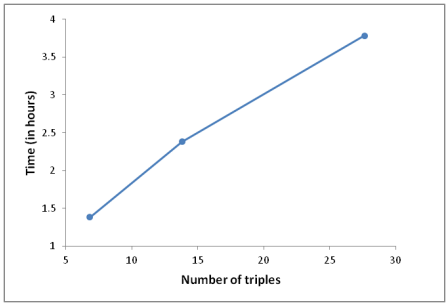

Fig. 4. Partition time for LUBM data

B. Evaluation of Parallel Reasoning on Partitions

Out of the 17 machines in our cluster, one serves as the master and the remaining 16 machines host the slave processes. For maximum utilization of our resources, we decided to run 2 processes per machine (PPM) to experiment with 8, 16 and 32 slave processes. However, due to memory limitations, we were unable to achieve the above configuration for our largest dataset, LUBM-200, with 8 processes (2PPM * 4 machines). Thus, for fair comparison between experiments on each dataset with respect to 8 processes, we resort to the configuration of 1PPM * 8 machines[1]. For experiments with respect to 16 and 32 processes, we keep the configuration of 2 PPM for each dataset. Note that, in theory, we do not need to run more than 1 PPM but we resorted to the above configuration for maximum utilization of resources available to us. For LUBM-100 and LUBM-50, the query processing time with the configuration of 2PPM * 4 machines was only 10% higher than that with 1PPM * 8 machines. We restricted our experiment to 32 processes to avoid too much resource contention between processes on the same node. Experiments were conducted when the load on these machines was very low. Results are an average of 10 trials.

We evaluated our system's performance using LUBM-50, LUBM-100 and LUBM-200 benchmark datasets. Fig. 5 shows the speedup for these datasets for 8, 16 and 32 processes. It shows the arithmetic average of speedup over all LUBM queries where 8 processes is our baseline.[2] The speedups fall

[1]For LUBM-200, the maximum memory usage for the in-memory DL models is 4GB per machine (for 1PPM *8 machines). For larger datasets, we can rely on the swap memory (6GB) for caching the models. Ideally, we would pre-compute and serialize the DL models for the partitions and load them only at query processing time, instead of holding them in-memory throughout.

[2]A single process cannot load LUBM-100 or LUBM-200 without partitioning. Hence we did not use 1 process as our baseline.

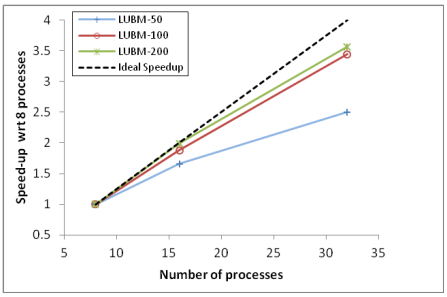

Fig. 5. Speed up for LUBM-50, LUBM-100 and LUBM-200 on 8, 16 and 32 processes.

short of embarrassing parallelism, mostly due to the setup cost (time spent distributing the query to the slaves) and performing join to get the final answers. We observe that we get better speedup with larger datasets. This can be attributed to the fact that the parallel framework overcomes the overhead of setup and performing joins. LUBM-100 and LUBM-200 result in a 3.5x speed-up which is 12.5% short of the perfect speedup. [3]

In Fig. 6, we compare the query execution times (wall clock time) for different LUBM datasets over different benchmark queries using 32 processes. For LUBM-50, response time for queries 1, 6, 10, 11, 12 and 14 are observed to be within 15 seconds. We find that queries 7 and 9 are worst performing queries and, in general, degrade rapidly as the size of dataset increases. We have found that the queries with high query processing time have very large intermediate resultsets for query subgoals. These large intermediate resultsets took longer to insert into database tables, to index and to join to obtain final results.

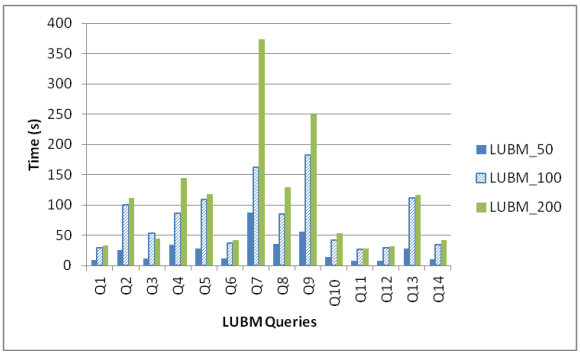

Fig. 6. Time taken to answer LUBM Queries on LUBM-50, LUBM-100 and LUBM-200 data using 32 processes

In order to explore this issue further, we analyze the overheads of different steps involved in query processing. In Fig. 7, *exec* denotes the time taken by the slaves to execute the query subgoals on their set of partitions and hold resultsets temporarily in memory. After this, we iterate over the resultsets and prepare to insert them into the database tables

[3]The configuration with respect to processes per machine, that we described earlier in this section, does not affect the speedup analysis very much and the shape of the curve remains intact.

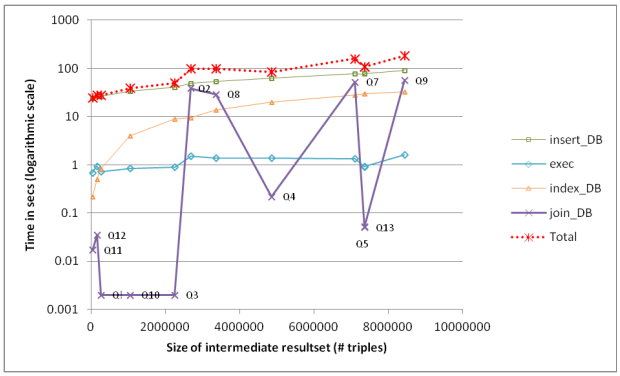

Fig. 7. Analysis of steps for query processing on LUBM-100 using 32 processes for the benchmark queries (except Q6 and 14).

for respective subgoals and we denote this entire operation by $insert\_DB$. Once all the slaves have finished processing the query on all its partitions, the master indexes the database tables, indicated by $index\_DB$. Finally, the master joins the tables corresponding to the query subgoals to compute the result of the query, indicated by $join\_DB$. Statistics are taken for query processing on LUBM-100 using 32 processes for all the benchmark queries, except queries 6 and 14 (since they have single subgoal and hence don't need indexing or joining). We can observe from Fig. 7 that, for most of the queries, the majority of the query processing time is spent in populating, indexing and joining the database tables compared to the time taken in executing the query subgoals (For queries shown in Fig. 7, the query execution time is less than 2 seconds for each query (denoted by $exec$) which is insignificant when compared to the database processing time). With respect to the resultset size, the query execution time grows slowly. For example, query Q10 has a resultset of size 1.04 million triples and an $exec$ time of 0.85s, while query Q7 has a resultset that is 7x larger and an $exec$ time that is only 1.5x longer. The database insertion and indexing times grow polynomially. The join time fluctuates from one query to another as it depends on multiple factors like number of subgoals, resultset size, number of join variables, query optimization and rewriting performed by the MySQL engine, etc. For the slowest queries (Q7 and Q9), the intermediate result sizes are at least 27 times larger than those for the fastest ones (Q1 and Q11). Also, the faster queries have fewer subgoal tables to join. Thus, for queries 7 and 9, a long time is spent in populating, indexing and joining the database tables compared to queries 1 and 11. Also, as the resultset size grows, the total insert, index and join time grows more quickly than the execution time. We can observe that queries 4, 5 and 13 have surprisingly smaller join time than queries 2, 7, 8 and 9, while they all have comparable intermediate resultset sizes. This is attributed to the fact that the latter queries have more intermediate resultsets and involve more join variables. These factors lead us to the conclusion that the database is essentially a bottleneck when dealing with queries involving multi-way joins with large intermediate resultsets and multiple join variables. We plan to address this issue in the future by devising intelligent and efficient join processing algorithms that take into account data distributed across nodes (e.g. parallel hash-join algorithm).

VI. Related works

Amir and McIlraith [10] have researched automatic partitioning of first-order and propositional theories and designed a message passing framework for reasoning with the partitioned knowledge base. The basic difference between their approach and our approach lies in the fact that their strategy, due to different requirements and goals, does not guarantee independent partitioning as we noted in Definition 2. Their approach generates partitions with potential links and correct reasoning may require communication between partitions.

Stuckenschmidt and Klien [15] have proposed an approach to partition OWL TBoxes based on the class hierarchy for modeling purposes. Unlike us, they do not take into account reasoning on the resulting partitions. Also, our focus is on the ABox. Fokoue et. al. [16] employ static analysis of knowledge representation and summarization techniques to extract a reduced proxy ABox, which, from the instantiation query point of view, depends on the given query; our partitioning is independent of the query.

Some previous work has been done in the area of parallel and distributed reasoning on RDF and OWL datasets. Some of these works [1, 17, 18] are based on RDFS reasoning in forward chaining style (materialization) which differ from our system which is based on reasoning in OWL Lite (a more expressive language than RDFS) using backward chaining. Oren et al. [1] combine parallel hardware with distributed algorithms to implement a system called MARVIN for scalable RDFS reasoning. The triples are randomly distributed between nodes based on a hash value of the triple and the node's rank in the network. Each node reasons on its input data and swaps some part of the computed data with another node for further inferencing. The technique guarantees anytime behavior and eventual completeness of inference process. In our system, there is no swapping of intermediate results between the nodes since we partition the dataset in such a way that each partition can be reasoned on independently.

Weaver et al. [17] derive an embarrassingly parallel algorithm for materializing complete RDFS closure using C/MPI. They divide the assertional triples evenly among the processes and give each process a complete set of ontological triples and RDFS rules. In their system, each of the processes perform fixpoint iteration on all finite RDFS rules. Unlike our system, they don't show query evaluation.

Heino et al. [18] perform parallel RDFS entailment on a massively parallel hardware in a shared memory setting where as our parallel reasoning system is deployed on cluster of compute nodes that don't share memory. DynamiTE [19] uses a parallel infrastructure to perform incremental materialization in the domain of stream reasoning. This system performs rule-based reasoning on ρdf RDFS fragment.

Urbani et al. [20] proposed a system called WebPIE which performs distributed computation of closure of an RDF graph

under the OWL Horst semantics using the MapReduce framework. OWL Horst is an extension of RDFS with ρD rules. It is a less complex dialect of OWL compared to OWL Lite. The entailment rules are encoded as Map and Reduce functions which are chained and applied on the data to compute the full closure. The compute nodes are partitioned into mappers and reducers according to their respective allocated functions. WebPIE might not scale that well in computing more complicated rules that use union and intersection. Also, this system has no query endpoint.

In [21], Urbani et. al. present an approach where they use a combination of backward-chaining and materialization of terminological closure to execute queries on OWL Horst knowledge bases. They introduce optimizations to reduce the search space of backward-chaining reasoning. However, they show query evaluation for only single pattern queries, not conjunctive queries.

Soma and Prasanna [22] propose data-partitioning and rule partitioning approaches. Data partitioning approach splits data among processes which have complete rulesets while rule partitioning partitions rules over processes which have complete dataset. Our work is somewhat similar to their domain specific data partitioning idea as they aim to put all related entities on same partition . They report good speedup for the LUBM benchmark for different numbers of processors.

VII. Conclusion and Future Work

We address the issue of scalable and parallel reasoning in this work. This work builds on the foundation of the work by Guo and Heflin [9] who proposed a polynomial time approach to partition OWL Lite knowledge base with respect to its TBox. Each resulting partition can be reasoned on independently while still guaranteeing sound and complete reasoning. We utilized this feature to design a framework to reason on these partitions in parallel on independent machines, which is our novel contribution in this work. We implement a master-slave architecture to carry out reasoning in parallel. The slave processes are assigned disjoint set of partitions such that the load is evenly balanced between them. Once the slaves have loaded the models for these partitions, the master distributes a conjunctive query across them. The slaves run in parallel, each performing reasoning to execute the subgoals of the query on its partition. Finally, the master joins the results computed by the slaves. We evaluated our system using LUBM-50, LUBM-100 and LUBM-200 datasets and we observed promising speedups for our approach using multiple compute nodes across different queries of the benchmark.

In the future, we plan to experiment with large scale real world datasets with expressive ontologies in OWL 2. We plan to parallelize the partitioning strategy which is currently serial. In order to address the bottleneck imposed by the relational database for queries involving large intermediate resultsets, we plan to introduce optimizations for efficiently computing joins on our parallel infrastructure. We also plan to look into join order processing by prioritizing the execution of selective subgoals whose resultset will be utilized by other subgoals. This may require sharing of intermediate results across the different nodes.

References

[1] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronald Siebes, Annette Teije, and Frank van Harmelen. MARVIN: A platform for large scale analysis of Semantic Web data, In Proc. of the WebSci'09: Society On-Line, 18-20 March 2009, Athens, Greece.

[2] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, et al. Sindice.com: A document-oriented lookup index for open linked data. International Journal of Metadata, Semantics and Ontologies,3(1):3752, 2008.

[3] I.Horrocks and P.Patel-Schneider. Reducing OWL entailment to description logic satisfiability. Journal of Web Semantics 1(4), pp. 345-357, 2004.

[4] D. Tsarkov and I. Horrocks, FaCT++ Description Logic Reasoner: System Description, In Proc.of IJCAR 2006,Seattle, USA, 2006.

[5] V.Haarslev and R. Moller. Racer: A Core Inference Engine for the Semantic Web. In the 2nd International Workshop on Evaluation on Ontology-based Tools (EON2003).

[6] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet:A practical OWL-DL reasoner. Journal of Web Semantics 5(2):51-53 (2007)

[7] Boris Motik, Rob Shearer, and Ian Horrocks. Optimized Reasoning in Description Logics using Hypertableaux. In Proc. of the 21st Conference on Automated Deduction (CADE-21), vol. 4603 of LNAI, pp. 67-83, Bremen, Germany, July, 2007. Springer.

[8] I. Horrocks, L. Li, D. Turi and S. Bechhofer. The instance store: DL reasoning with large numbers of individulas. In Proc. of 2004 International Workshop on Description Logics(DL2004).

[9] Guo, Yuanbo and Heflin, Jeff . A Scalable Approach for Partitioning OWL Knowledge Bases. In Proc. of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2006). Athens, Georgia. 2006.

[10] E. Amir and S. McIlraith. Partition-Based Logical Reasoning. In Proc. of the 7th International Conference on Principles of Knowledge representation and Reasoning (KR2000).

[11] Y. Guo and J. Heflin. On Logical Consequences for Collections of OWL Documents. In Proc. of the 4th International Semantic Web Conference (ISWC2005).

[12] V. Royer and J.J. Quantz. Deriving Inference Rules for Terminological Logics. In Proc. of the European Workshop on Logics in Artificial Intelligence (JELIA1992).

[13] V. Royer and J.J. Quantz. Deriving Inference Rules for Description Logics: a Rewriting Approach into Sequent Calculi. Technical Report: TUB-FB13-KIT-111, Dec. 1993. http://dl.acm.org/citation.cfm?id=894980

[14] D. Tsarkov and I. Horrocks. DL Reasoner vs. first-order provers. In Proc. of 2003 International Workshop on Description Logics (DL2003).

[15] H. Stuckenschmidt and M. Klein. Structure-based partitioning of large class hierarchies. In Proc. of the 3rd International Semantic Web Conference(ISWC2004).

[16] A. Fokoue, A. Kershenbaum, L. Ma, E. Schonberg and K.Srinivas. The Summary ABox: Cutting Ontologies Down to Size. In Proc. of 5th International Semantic Web Conference (ISWC2006).

[17] J. Weaver and J. Hendler. Parallel materialization of the finite rdfs closure for hundreds of millions of triples, In Proceedings of the ISWC, 2009.

[18] Norman Heino and Jeff Z. Pan, RDFS Reasoning on Massively Parallel Hardware, In Proc. of the International Semantic Web Conference ISWC 2012, Boston, 2012.

[19] Jacopo Urbani, Alessandro Margara, Ceriel J. H. Jacobs, Frank van Harmelen, and Henri E. Bal. DynamiTE: Parallel Materialization of Dynamic RDF Data, 12th Int. Semantic Web Conference (ISWC 2013), Sydney, Australia.

[20] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen and Henri Bal. OWL reasoning with WebPIE: calculating the closure of 100 billion triples, Journal of Web Semantics, Vol 10, 2012.

[21] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, Henri E. Bal: QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases. International Semantic Web Conference (1) 2011: 730-745

[22] Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In Proc. of the 37th International Conference on Parallel Processing, Washington DC, USA. 2008.