MapResolve

Anne Schlicht, Heiner Stuckenschmidt

University of Mannheim
{anne, heiner}@informatik.uni-mannheim.de

Abstract. We propose an approach to scalable reasoning on description logic ontologies that is based on MapReduce. Our work is inspired by previous work that provided fast materialization of RDFS ontologies and proposed MapReduce for more expressive logics. We explain challenges imposed by higher expressivity that were not addressed before and describe how they can be solved.

1 Introduction

The MapReduce framework [2] was developed by Google labs for facilitating distributed processing of large datasets. An open source implementation of the framework is available[1] and is attractive for many resource intensive computations. With a quite small effort for adapting a given application to the MapReduce interface, the user benefits from multiplied resource availability and built-in fault tolerance. However, the simple interface comes by the cost of reduced flexibility in process interaction, which is a drawback in complex applications. We investigate the application of MapReduce for reasoning on ontologies. In particular, we review previous approaches of MapReduce for RDFS and OWL Horst materialization and $\mathcal{EL}+$ classification. Moreover, we propose some extensions that extend the applicability to other OWL fragments up to first order logic.

2 Previous Work

Recently, the MapReduce framework was proposed for materialization and classification of OWL fragments. We first review these approaches before we discuss the MapReduce application for checking satisfiability of expressive ontologies.

2.1 MapReduce

The MapReduce framework [2] provides a simple interface for cluster computation. Two functions have to be implemented by the user to access the automatic distribution, both functions are executed by a set of *workers* (i.e. machines). First, the *map* function assigns a key to each input value (in this work the values are axioms) and outputs $(key, value)$ pairs. Then, the *reduce* function is called once for each key. It processes all corresponding values and outputs a list of results. A partition function assigns the keys of the map output to reduce workers.

[1] e.g. Apache Hadoop http://hadoop.apache.org

2.2 RDF Schema Materialization

One of the first applications of MapReduce in ontology reasoning is the computation of the closure of a large RDF(S) graph described in [8]. RDF Schema rules are implemented by MapReduce jobs. For example, the RDFS subclass rule

$$s \text{ rdf:type } x \quad \& \quad x \text{ rdf:subClassOf } y \quad \Rightarrow \quad s \text{ rdf:type } y$$

is implemented by a map function that maps potential premises to the shared element x. I.e., the key for triples with predicate “rdf:type” is the object, the key for triples with predicate “rdfs:subClassOf” is is the subject of the triple. The whole triple is returned as value of the map output pair. The reduce function is called once for each key and derives new axioms according to the subclass rule from all triples that share this key. Note that a single call to this job performs all derivations of this rule. The work for deriving all implied triples of type $(s, \text{rfd:type}, o)$ is partitioned among the reduce workers based on the objects o that are the keys in the input to the reduce function.
The other RDFS rules are implemented by MapReduce jobs in a similar way. The complete materialization consists of a sequence of MapReduce jobs, where the output of one job is the input of the next job. As shown in [8], this method is quite efficient when the number of schema triples is small enough to be stored in memory of each reducer node. With clever ordering of the RDFS rules, the materialization is usually[2] complete after calling each job once. Hence, only a handful of MapReduce jobs is necessary for materialization of the deductive closure.

2.3 OWL Horst Materialization

The RDFS materialization was extended to OWL Horst in [7]. OWL Horst [6] is a fragment of the Web Ontology language OWL that can be materialized using a set of rules that is an extension of the set of RDF schema rules. The fragment is popular for triple stores that are focused on scalability because of the relatively high expressivity and feasible reasoning methods. The additional rules add semantics for the OWL constructs “owl:someValuesFrom”, “owl:allValuesFrom” and “owl:TransitiveProperty”. The higher expressivity of OWL Horst compared to RDFS requires a couple of optimizations to keep tractability. While for RDFS it is possible to have a single ’stream’ of instance triples for each reduce worker, OWL Horst requires joins over more than one instance triple. The number of necessary expensive joins is reduced by storing the “owl:sameAs” triples only implicitly and other optimizations for transitive properties and property restrictions. With these optimizations, the authors were able to compute the closure of 100 billion triples. However, some inefficiencies were detected: For OWL Horst rules, there is no order that can avoid the need for iterating repeatedly over all

[2] For certain cases (e.g. if subproperties of ’rdf:SubpropertyOf’ are defined) that are very rare in real world ontologies, repeated application of the rule sequence is necessary for completeness.

rules. As the authors report, this is problematic because the same conclusions are derived again and again in every iteration.

2.4 $\mathcal{EL}+$ Classification

$\mathcal{EL}+$ [1] is a fragment of OWL that does not contain union operators or forall restrictions. Concepts in $\mathcal{EL}+$ are built according to the grammar

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

where A is a concept name, r is a role name and C, D are concept names or complex concepts. In addition to general concept inclusions $C \sqsubseteq D$ and assertions, an $\mathcal{EL}+$ ontology may contain role inclusions $r_1 \circ ... \circ r_n \sqsubseteq r$ where $r, r_1, ..., r_n$ are role names. The essential property of $\mathcal{EL}+$ is the existence of a simple set of derivation rules that allows classification of $\mathcal{EL}+$ ontologies in polynomial time. For example, the rule

$$X \sqsubseteq A \quad \& \quad A \sqsubseteq \exists r.B \quad \Rightarrow \quad X \sqsubseteq \exists r.B$$

propagates a restriction on a class A to the subclass X of A. Motivated by the materialization approaches mentioned before, [3] proposes a MapReduce variant of the $\mathcal{EL}+$ classification algorithm CEL. The derivation rules of CEL are translated to MapReduce jobs. Before the translation, the rules are slightly adapted, such that for every rule all premises share at least one class or property name. The shared terms are used as key in the input of the reduce function (output of the map function) similar to the RDFS materialization. For the above rule, axioms $A \sqsubseteq B$ are assigned the key B and restrictions $A \sqsubseteq \exists r.B$ are assigned the key A. The reduce workers derive new axioms from sets of axioms that share the same key. In contrast to the previous approaches, only the input to the reduce function is considered as premises and this set of potential premises is not changed while the reduce worker runs. Recall that in the RDFS materialization, *all* applications of a certain rule are executed in a single MapReduce job. In $\mathcal{EL}+$ classification, an axiom derived by a reduce worker can only be considered as premise in the next job. Hence, the number of required MapReduce jobs is at least the depth of the derivation graph. Another difference to previous approaches is the maintenance of the axiom set. The authors propose to store the axioms in a database instead of the files that are used by, e.g., the Hadoop implementation of MapReduce.
The approach suffers from an unsolved efficiency issue: Rules of the underlying CEL algorithm are only applied, if the conclusion is not already contained in the current axiom set. But, in the MapReduce variant of the algorithm, the authors do not report how this preconditions are checked and the preconditions are not mentioned in the adapted rules set. We assume, that the database that is used for storing intermediate results deletes duplicate axioms. But anyway, if already derived axioms are repeatedly derived in every iteration, the method is inefficient, especially because the number of iterations is very high as mentioned before.

3 Description Logic Satisfiability

After analyzing the challenges of previous approaches we apply the MapReduce framework for checking satisfiability of expressive ontologies. We will face similar problems as the $\mathcal{EL}+$ classification and propose a different solution that is also relevant for other approaches.
In previous work [4, 5], we developed a distributed resolution method for checking satisfiability of a given set of axioms translated to first order clauses. Different variants of the algorithm are used depending on the expressivity of the ontology. In theory, the method can be used for first order theories, but due to limited space we focus on the basic variant for $\mathcal{ALCHI}$ in this paper. The reasoning method is based on ordered resolution. For clauses C and D and literals A and $\neg B$, standard resolution is defined by the rule

$$C \vee A \quad \& \quad D \vee \neg B \quad \Rightarrow \quad C\sigma \vee D\sigma$$

where σ is the most general unifier of A and B. For ordered resolution, the literals of each clause are ordered based on a precedence of predicate and function symbols. Ordered resolution inferences are than restricted to premises where the literals A and $\neg B$ that are unified are the maximal literals of the premises. A second rule (factoring) is necessary to guarantee completeness for first order logic. We skip the definition because it has only one premise and hence we do not need to take it into account for distribution of the reasoning method. For clauses obtained from $\mathcal{ALCHI}$ ontologies, ordered resolution terminates and derives an empty clause if and only if the input ontology is inconsistent.
All literals that occur in clauses obtained from an $\mathcal{ALCHI}$ ontology are of the form $P(t)$ or $P(t_1, t_2)$ where P is a unary or binary literal and t is a constant or variable or a term of form $f(x)$ where f is function symbol and x is a constant or variable. Literals A and B are only unifiable, if the predicates are the same, i.e. $A = P(...)$ and $B = P(...)$ with a predicate P.

3.1 Naive MapReduce for Distributed Resolution

The key to applying MapReduce to description logic resolution is the shared predicate of unified literals. Similar to the previous approaches, the map function reads all clauses and outputs a $(key, value)$ pair for every clause. The value is the clause, the key is the predicate of the maximal literal of the clause. Every clause has a unique key, because clauses obtained from $\mathcal{ALCHI}$ ontologies have a unique maximal literal [5]. For more expressive ontologies, literal types and unification are more complicated and multiple $(key, value)$ pairs may be generated for a clause. The partitioning function of the MapReduce job allocates keys (i.e. predicates) to reduce workers. In the reduce function, the derivations are performed on clauses that have the same predicate in their maximal literals. This can be implemented using a standard reasoner that returns the local saturated clause set as output. The output is then merged before the next call to the MapReduce job. In the next map phase keys are recomputed. In contrast to the previous approaches we can use a single map and reduce function for the whole saturation. But, like for OWL Horst and $\mathcal{EL}+$ we have to repeat the job until no new clause is derived.

3.2 Avoiding Repetition

The problem of the straightforward application of MapReduce to resolution are repeated inferences. Without recording the work that is already done, we will repeat every derivation that is performed in every subsequent call to the job. To solve the problem, we remember that repeated inferences are avoided by standard reasoners using a very simple strategy. The current clause set of the saturation process is partitioned into two sets: A set of clauses that is already completely interresolved, this is the *worked-off* (WO) set. All other clauses are in the *usable* (US) set. We start with all input clauses in the US set and an empty WO set. Now, we iteratively pick a clause from the US set and resolve it with any possible clause from the US set and then move the picked clause to the WO set. Derived clauses are always added to the US set. This method makes sure that any combination of premises is only tried once. The simple but effective method can be applied to the file-based MapReduce resolution: Every reducer works on two files that serve as US and WO set. When calling the reduce function, the reducer first reads the WO clauses and then starts resolving with the first usable clause. Derived clauses are appended to the local US set if they have a local key according to map and partition function. Other derived clauses are stored in a separate set *DE*. When the reducer is finished with the US set, the WO and DE sets are written to disk. In the next iteration, DE clauses are allocated by the map function to obtain the new US sets. Then the reducers are started on the new WO and US sets. Efficient resolution reasoning requires deletion of duplicate clauses and clauses that are subsumed by other clauses. This redundancy check is performed in the reduce function. For every clause picked from the US set, we first check if it is redundant or subsumes a clause of the WO set and delete the redundant clauses.

3.3 Work Load Balance

The time required for local saturation can be very different among the reduce workers depending on the size of the WO and US sets. To optimize balance of work load, we modify the reduce function to saturate only a given number (*usChunk*) of clauses and not the whole US set. At the beginning, we define the amount of time $t_{intended}$ an iteration should take and set *usChunk* to the same number for each key. After each run of the job, we increase or decrease the number of US clauses that have to be resolved depending on the difference between intended and actual runtime: $usChunk = usChunk \cdot t_{intended}/t_{actual}$. With chunked US sets work load balance would be improved considerably.

4 Conclusion

We investigated MapReduce approaches to ontology reasoning and found the main challenge is avoiding repetition of inferences. For the limited expressivity of RDFS, the problem can be avoided because every MapReduce job is executed only once. For more expressive ontologies fixpoint iteration is necessary and causes many repeated inferences in previous approaches. Applying MapReduce for distributed resolution requires an approach that efficiently avoids repetition. We propose solving the problem by adapting the standard method for avoiding repetition of resolution inferences. The solution is also applicable to other MapReduce approaches, $\mathcal{EL}+$ classification would be tractable, the runtimes of OWL Horst materialization would benefit from this optimization. Considering a reasoning application that faces scalability problems on large input, MapReduce implementations probably provide the easiest access to massive computation and memory resources.
However, there are problems caused by separating the saturation process into a sequence of jobs. With each iteration, the clause sets are parsed and written to disc, generating needless overhead. Furthermore, derived clauses are not passed on to the next worker instantly but only after the current job finishes. These disadvantages are inherent to the MapReduce framework, they are the price for usability and fault tolerance. For applications that focus on optimal performance, frameworks that allow interaction between workers are preferable.

References

1. F. Baader, C. Lutz, and B. Suntisrivaraporn. Efficient reasoning in $\mathcal{EL}^+$. In *Proceedings of the 2006 International Workshop on Description Logics (DL2006)*, CEUR-WS, 2006.
2. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
3. Raghava Mutharaju, Frederick Maier, and Pascal Hitzler. A MapReduce Algorithm for EL +. In *Workshop on Description Logics (DL2010)*, pages 464–474, 2010.
4. Anne Schlicht and Heiner Stuckenschmidt. Distributed resolution for expressive ontology networks. In *Web reasoning and rule systems : Third International Conference, RR 2009*, 2009.
5. Anne Schlicht and Heiner Stuckenschmidt. Peer-to-peer reasoning for interlinked ontologies. *International Journal of Semantic Computing, Special Issue on Web Scale Reasoning*, 4(1), March 2010.
6. H.J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):79–115, 2005.
7. Jacopo Urbani, Spyros Kotoulas, and Jason Maassen. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. *The Semantic Web:*, pages 213–227, 2010.
8. Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and F. van Harmelen. Scalable distributed reasoning using mapreduce. *The Semantic Web-ISWC 2009*, pages 634–649, 2009.