

Scalable Web Reasoning using Logic Programming Techniques^{*}

Gergely Lukácsy¹, Péter Szeredi²

¹ Digital Enterprise Research Institute, Galway, Ireland

² Budapest University of Technology and Economics, Budapest, Hungary
`gergely.lukacsy@deri.org, szeredi@cs.bme.hu`

Abstract. One of the key issues for the uptake of the Semantic Web idea is the availability of reasoning techniques that are usable on a large scale and that offer rich modelling capabilities by providing comprehensive coverage of the OWL language. In this paper we present a scalable extension of our ABox reasoning framework called DLog.

DLog performs query-driven execution whereby the terminological part of the description logic knowledge base is converted into a Logic Program and the assertional facts are accessed dynamically from a database. The problem of instance retrieval is reduced to a series of instance checks over a set of individuals containing all solutions for the query. Such a superset is calculated by using static-code analysis on the generated program.

We identify two kinds of parallelism within DLog execution: (1) the instances in the superset can be independently checked in parallel and (2) a specific instance check can be executed in parallel by specialising well-established techniques from Logic Programming. Moreover, for efficiency reasons, we propose to use a specialised abstract machine rather than relying on the more generic WAM execution model. We describe the architecture of a distributed framework in which the above mentioned techniques are integrated. We compare our results to existing approaches.

KEYWORDS: Scalability, Parallelism, OWL, DL, Logic Programming

1 Introduction

In this paper we describe extensions of DLog, a *SHIQ* Description Logic (DL) ABox reasoner [1], using the unique name assumption. We are interested in scenarios that have large numbers of individuals and a relatively small terminology and where query answering is the most important reasoning task. This is in line with the aims of the upcoming OWL 2 QL profile. In this setup, DLog already proved to be very efficient thanks to its query-oriented top down execution model that ensures that only those parts of the ABox are accessed that are relevant to the given query. However, the DLog execution is sequential which turns out to be a bottleneck when working with really large datasets, as DLog is simply not able to feed the underlying database with queries fast enough.

^{*} This work has been funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2) and by the Irish Research Council for Science, Engineering and Technology (IRCSET). Earlier development work on the DLog system was supported by the Hungarian NKFP programme under Grant No. 2/052/2004.

The contributions of the present paper include a new abstract machine designed to efficiently execute logic programs generated by the DLog system as well as a new parallel architecture of DLog.

We acknowledge that having a scalable DL reasoner does not solve all the problems of the Semantic Web. Specifically, we still need solutions for handling the heterogeneity of web ontologies, to provide support for ontology evolution and time management and to create a solid foundation of trust. We also acknowledge that data complexity results on the *SHIQ* language suggest [2] that sound and complete query answering is unlikely to be efficient on really large amounts of data. However, we believe that by providing the basis of a scalable reasoning framework on an expressive but still decidable OWL fragment we make a step towards turning the Semantic Web idea into reality.

The paper is structured as follows. In Section 2 we introduce the DLog system in a nutshell and summarise those features that are relevant for the rest of the paper. Section 3 discusses the design of the abstract machine for the execution of simple logic programs generated from DL knowledge bases. Section 4 presents the workflow and the architecture of the parallel DLog system. Section 5 gives a brief overview of related work, while in Section 6 we conclude with the discussion of future work and the summary of our results. Throughout the paper we assume basic level knowledge on Description Logics [3] and Prolog [4].

2 Overview of the DLog approach

In this section we give a brief overview of the DLog reasoning process using an example (see [1] for formal details). The main idea is to transform the DL knowledge base to a Prolog program and use normal Prolog execution on it to answer instance retrieval queries. Let us consider the following knowledge base.

1	$\exists \text{hasFriend. Alcoholic} \sqsubseteq \neg \text{Alcoholic}$
2	$\exists \text{hasParent. } \neg \text{Alcoholic} \sqsubseteq \neg \text{Alcoholic}$
3	<code>hasParent(joe, bill). hasParent(joe, eva). hasFriend(bill, eva).</code>

This TBox states that if someone has a friend who is alcoholic then she is not alcoholic (line 1). Furthermore, if someone has a non-alcoholic parent then she is not alcoholic either (line 2). The ABox contains several role assertions, but nothing about someone being alcoholic or non-alcoholic. Thus, in the database world, looking for non-alcoholic people would yield no results. In DL however, we can conclude that `joe` is non-alcoholic as one of his parents is bound to be non-alcoholic (as at least one of two people who are friends has to be non-alcoholic).

The common property of such problems is that solving them requires *case analysis* and therefore the trivial Prolog translation usually does not work. There are many other examples showing how incomplete knowledge is handled during DL reasoning, some of them do not even require a TBox [5].

The first step in the sound and complete DLog reasoning process is to convert a *SHIQ* TBox to a set of first order clauses containing no function symbols,

called *DL clauses* [6]. This allows us to break the reasoning into two parts: an ABox independent TBox transformation followed by the actual data reasoning.

The second step deals with the transformation of DL clauses to a Prolog program. This is based on the Prolog Technology Theorem Proving (PTTP) approach, which provides a generic first-order theorem prover on top of Prolog [7]. PTTP uses *contrapositives* to compensate for the simple literal selection rule of Prolog; *ancestor resolution* for implementing the factoring resolution rule; and *iterative deepening* to ensure termination. For efficiency reasons in DLog we specialised this approach for the case of DL clauses. Specifically, for the simple function-free Prolog code generated from DL clauses, normal Prolog execution extended with *loop elimination* can be used instead of iterative deepening.

DL clauses are transformed to a *DL predicate* format by generating certain contrapositives and grouping these into predicates according to the functor of the clause head. Negations are eliminated by introducing new predicate names. DL predicates can be executed by an interpreter or, alternatively, DL predicates can be compiled into directly executable Prolog code, by adding an argument for storing the list of ancestors and including loop elimination and ancestor resolution in the DL predicates themselves. As an example, the DL predicate format of the above alcoholic problem is shown below:

```

1 alcoholic(A) :- hasParent(B, A), alcoholic(B).           → contrapositive
2 not_alcoholic(A) :- hasParent(A, B), not_alcoholic(B). → original clause
3 not_alcoholic(A) :- hasFriend(A, B), alcoholic(B).     → original clause
4 not_alcoholic(A) :- hasFriend(B, A), alcoholic(B).     → contrapositive

```

We now present the Prolog code generated for the DL predicate `not_alcoholic`, as shown in lines 2-4 above (we use the abbreviation `not_alc` for compactness).

```

1 not_alc(A, L0) :- member(B, L0), B == not_alc(A), !, fail.
2 not_alc(A, L0) :- member(alcoholic(A), L0), !.
3 not_alc(A, L0) :- L1=[not_alc(A)|L0], hasParent(A, B), not_alc(B, L1).
4 not_alc(A, L0) :- L1=[not_alc(A)|L0], hasFriend(A, B), alcoholic(B, L1).
5 not_alc(A, L0) :- L1=[not_alc(A)|L0], hasFriend(B, A), alcoholic(B, L1).

```

Lines 1 and 2 implement loop elimination and ancestor resolution, respectively. Lines 3-5 are derived from the clauses of `not_alcoholic`, by extending the head and appropriate body calls with an additional argument, storing the ancestor list (variables `L0` and `L1`). Similar code is generated for predicate `alcoholic`. Although we proved that the translation exemplified above is complete it is not efficient. In DLog we use a series of optimisations that result in more efficient (and more complex) Prolog translation; these are described in detail in [1].

Here we only mention two optimisations, decomposition and superset. The goal of *decomposition* is to split a body into independent components and make sure that the truth value of each component is only calculated once. Decomposition is achieved by a recursive process that uncovers the dependencies between

the goals of the body. This optimisation results in clause bodies where independent components are separated using conditional Prolog structures. For example, the axiom that “someone is happy if she has a child having both a clever and a pretty child” results in the following translation (excluding the management of ancestor lists).

```

1 Happy(A) :-
2   hasChild(A, B),
3   (   hasChild(B, C),
4       Clever(C) -> true           → first component
5   ),
6   (   hasChild(B, D),
7       Pretty(D) -> true          → second component
8   ).

```

The idea of *superset* is to determine for each predicate P a set of instances S for which $I(P) \subseteq S$ holds, where $I(P)$ denotes the set of solutions of P . If the size of S is not significantly greater than the size of $I(P)$, then we can use S to efficiently reduce the initial instance retrieval problem to a finite number of *deterministic* instance checks. Technically this generic schema can be implemented by creating a “choice” predicate for each concept `Concept` that invokes the deterministic variant of `Concept` for each individual in the superset.

```

1 choice_Concept(A, AL) :-
2   (   nonvar(A) -> deterministic_Concept(A, AL)
3   ;   member_of_superset_Concept(A),
4       deterministic_Concept(A, AL)
5   ).
6 deterministic_Concept(A, AL) :- ..., !.   → A is a specific instance
7 ...

```

Note that the deterministic translation of a DL concept has Prolog cuts (!) at the end of each of its clauses. This results in pruning the rest of the search space after a successful execution of any of the clauses.

A superset of predicate P is calculated by applying static program analysis as described in detail in [1]. For example, the superset of predicate `not_Alcoholic` includes all individuals having a parent or a friend, or being a friend of someone.

The optimisations applied in the DLog system guarantee that ancestor goals are always ground. This opens up the possibility to use hash tables rather than lists for managing ancestor resolution and loop elimination. For this purpose we have implemented in C a *backtrackable hash table* [8]. This, besides the obvious efficiency advantage, also makes DLog programs structure free.

3 The DLog Abstract Machine

The Warren Abstract Machine (WAM) [9] has become a de facto target platform for Prolog compilers. Most sequential and parallel implementations of

Prolog rely directly on WAM, or on a variant of it. For efficiency reasons we suggest to use a much simpler abstract machine, called the DLog Abstract Machine (DAM), to execute Prolog programs generated using the DLog approach. In the following we sketch the main design principles of DAM.

Compared to generic Prolog, DLog programs for instance checking are considerably simpler as: (1) predicates can only be unary or binary; (2) there are no compound data structures – unification is trivial; (3) predicate invocations are ground; (4) concept predicates are deterministic – no need for deep backtracking into concept predicates; (5) no need for the heap and the trail stack; (6) no need for cell tagging, as all constants are numeric.

3.1 Architecture of DAM

The DAM is a three-stack machine. It has a *control stack* containing *frames* of fixed size. A frame is created when entering a predicate and is used to store the local environment of the predicate and return address information. A predicate can be viewed as a function which receives its arguments implicitly in a frame and returns a Boolean value. The second stack, the *choice point stack*, is used to support deep backtracking in cases related to role predicates and also to ensure efficient communication with the database/triple store³. The third stack is used as a backtrackable hash table according to the principles discussed in [8].

Four pieces of information are stored in a frame of the DAM control stack:

1. the return address of the predicate (virtual register R);
2. the actual instance being checked, represented by a URI (virtual register A);
3. the ancestor list, represented as an index in the backtrackable hash table (virtual register H);
4. a pointer to the corresponding choice stack frame (virtual register P).

The fields of the current frame serve as (virtual) registers of the DAM. As the frames are of fixed size, accessing a field of e.g. the frame preceding or following the current one incurs no overhead.

The following information is stored in a frame of the choice point stack:

1. a counter used in implementing number restrictions (virtual register C);
2. a handle used for interfacing with the triple store;
3. a buffer for instances returned by the triple store.

Further to the virtual registers on the stacks, DAM has the following (global) registers: V – the Boolean return value of a procedure invocation; PC – the program counter variable; T – the current frame of the control stack.

DAM operates only with three control structures: conjunction, disjunction and loops (used for counting instances in a qualified number restriction, including existential restrictions as a special case). In discussing the DAM we assume that each predicate contains exactly one of the three control structures; this can be achieved by introducing auxiliary predicates. As an example we show below how the **Happy** example from Section 2 can be transformed to satisfy this assumption.

³ Triple stores are specialised databases for storing semantic web metadata. In the following we use the phrases “triple store” and “database” as synonyms.

1	Happy(A) :-	
2	aux_1(A).	→ a conjunction with a single member
3	...	→ possible other clauses of Happy
4	aux_1(A) :-	→ existential restriction
5	hasChild(A, B),	
6	aux_2(A).	
7	aux_2(A) :-	→ a conjunction with two members
8	aux_3(A),	
9	aux_4(A).	
10	aux_3(A) :-	→ existential restr.
11	hasChild(A, B),	
12	Clever(B), !.	
	aux_4(A) :-	→ existential restr.
	hasChild(A, B),	
	Pretty(B), !.	

3.2 The instruction set

The instruction set of the DAM is fairly limited. Each instruction consists of an operation code with typically zero, one, or two operands. For example, the instruction `call pred` invokes predicate `pred`, while `exit_on_failure` (with no arguments) exits the given predicate if register `V` contains the value `FAILURE`. The set of instructions of the DAM is summarised in Table 1.

In Figure 1 we give the operational semantics of the DAM instructions using pseudo-code with C syntax. Here we use capitals to refer to DAM registers, lower case names for parameters and local variables, while the keywords `previous` and `next` refer to the frames preceding and following the current one, respectively. A register reference can be used on its own (e.g. `A`), referring to the appropriate field of the current control frame; or it can be used together with the keyword `previous` or `next`, to refer to the appropriate field of a neighbouring frame. The instruction `exit_with` is invoked within other instructions: this is considered as a macro to be expanded, i.e. the invocation should be replaced by the definition.

To simplify the presentation we do not deal with memory management issues, assuming that the stacks have enough memory allocated to perform the computation. Thus creating or removing a stack frame is simply done by incrementing or decrementing register `T` (which points to the current control frame).

We assume that the DAM-triple store interface works as follows. Once a query has been posed to the triple store it returns a handle which is stored in the actual choice-stack frame. Using this handle the DAM can ask for the first batch of solutions, which is then stored in the buffer part of the relevant choice-stack frame. The buffer involves a header specifying the buffer length and the number of solutions not yet processed. This setup makes it possible to return query solutions one by one to the DAM code which issued the given query. When a solution is requested and the buffer is empty, a request is sent to the triple store (using the query handle) to provide the next batch of solutions.

Table 1. The instruction set of the DAM

Instruction	Arguments	Description
<code>put_ancestor</code>	N	extend the ancestor list in the local frame by the term with name N and argument A
<code>check_ancestor</code>	N	succeeds if the ancestor list contains a term with name N and argument A
<code>fail_on_loop</code>	N	fails if a loop occurred, i.e. the term with name N and argument A is present on the ancestor list
<code>call</code>	P	invokes procedure P in a new control frame
<code>execute</code>	P	invokes procedure P in the existing control frame
<code>exit_with</code>	S	returns from a procedure with status S, continues execution according to register R
<code>exit_on_failure</code>	–	returns from procedure if V = FAILURE
<code>exit_on_success</code>	–	returns from procedure if V = SUCCESS
<code>jump</code>	L	jumps to label L
<code>has_n_successors</code>	R, n	checks if instance A has at least n R successors; creates a choice point; loads the first choice to A
<code>count_and_exit</code>	–	decreases counter C if the previous instruction was successful; returns with success if C is 0
<code>next_choice</code>	–	loads the next solution from the choice stack to A
<code>abox_query</code>	Q	returns success if A is a solution of query Q

We now describe the auxiliary procedures used in Figure 1. Procedures `add_to_hash` and `hash_search` perform the extension and search of the back-trackable hash table [8], representing the ancestor list. The procedure `cardinality_check(i, r, n)` returns true if the triple store contains at least n r -successors for the instance i . The procedure `create_choice(i, r)` issues a query to the triple store to find all r -successors of the individual i , and returns the first solution found. The procedure `has_choice` returns true if the current choice frame has more solutions, while `next_choice` returns the next solution.

Finally, the procedure `abox_query(i, q)` checks if instance i belongs to query predicate q according to the triple store. Query predicates are defined in terms of ABox predicates using conjunction and disjunction only; they can be thought of as complex database queries. In its most simple case a query predicate corresponds to a simple ABox concept predicate.

3.3 Transforming into DAM

Now we turn our attention to discuss how certain parts of the DLog programs are transformed into DAM code.

Role axioms are handled partly by the DLog framework (the transitivity axioms are eliminated) and partly by the underlying triple store. Namely, we assume that the database “understands” the notion of role hierarchy and is able to answer queries such as `hasSpouse(bill, Y)` (Y is the spouse of `bill`) based on hierarchical relation between `hasSpouse` and `hasWife`, for example.

```

1 put_ancestor n:      → inserts term n(A) into the hash table
2   H = add_to_hash(A, n, H);

3 check_ancestor n:   → checks if term n(A) is in the hash table
4   if (hash_search(A, n, H)) exit_with SUCCESS;

5 fail_on_loop n:     → checks if term n(A) is in the hash table
6   if (hash_search(A, n, H)) exit_with FAILURE;

7 call p:
8   T++; A = previous->A; H = previous->H; R = PC + 1;
9   PC = &p;          → invokes procedure in new frame

10 execute p:
11   PC = &p;          → invokes procedure in the current frame

12 exit_with s:
13   T--; V = s; PC = next->R; → drops frame; jumps to return address

14 exit_on_failure:
15   if (V == FAILURE) exit_with V;

16 exit_on_success:
17   if (V == SUCCESS) exit_with V;

18 jump l:             → jumps to label l
19   PC = l;

20 has_n_successors r n: → loads successors of A to the choice stack
21   if (!cardinality_check(A, r, n)) exit_with FAILURE;
22   A = create_choice(A, r);

23 count_and_exit:     → counts and exists if counter reaches zero
24   if (V == SUCCESS) P->C--;
25   if (P->C == 0) exit_with SUCCESS

26 next_choice:
27   if (!has_choice()) exit_with FAILURE;
28   A = next_choice(); → sets the next solution instance to A

29 abox_query q:
30   V = abox_query(A, q); → executes a (complex) database query

```

Fig. 1. Operational semantics of the DAM instructions

Conjunctions of concept predicates are transformed into a series of `call` and `exit_on_failure` instructions. That is, a conjunction consisting of goals...

```
1 g1(X), ..., g_{k-1}(X), g_k(X)
```

...is directly transformed into the following DAM form:

```
1 call g1
2 exit_on_failure
3 ...
4 call g_{k-1}
5 exit_on_failure
6 execute g_k
```

Note that the last goal of the conjunction is invoked using the `execute` instruction rather than the `call` instruction. This is an optimisation which allows us to use the current frame to invoke g_k rather than creating a new one. For recursive predicates this is known as the *tail-recursion optimisation* which basically transforms a recursive call into an iteration.

Analogously to conjunctions, disjunctions of concept predicates are transformed into a series of `call` and `exit_on_success` instructions, with `execute` for the last goal. Note that both transformation schemata use the fact that we work with deterministic predicates, e.g. a disjunction immediately succeeds if one of its members completes successfully. Alternatively, one could use an even more compact schema, where each pair of `call` and `exit_on_success` instructions is replaced by a single one, similar to the `try` instruction of the WAM [9]. However, this would require that two separate return addresses – one for success and one for failure – were stored on the control stack for each predicate.

Finally, a qualified number restriction ($\geq nRC$) is transformed into a *loop*, where we first check if the instance at hand has at least n R -successors, then we enumerate the successors until we find at-least n successors belonging to concept C . Specifically, ($\geq nRC$) is transformed into the following DAM program.

```
1 has_n_successors R n → fails if A has not enough successors, sets A, C
2 label1:
3 call C → returns with success or failure
4 count_and_exit → if success: C--, returns success if C = 0
5 next_choice → set A to next successor, return fail if no more
6 jump label1
```

Note that this technique handles the only case where deep backtracking is needed in the Prolog code. As an optimisation for the above schema we can use techniques that allow us to reuse a frame rather than repeatedly build it with `call C` in each iteration.

As an example for the DLog to DAM translation, let us show below parts of the DAM code for the predicate `not_alc` from Section 2.

```

1 predicate(not_alc):           → A contains the instance to check
2   fail_on_loop not_alc      → return fail if within not_alc(A)
3   check_ancestor alcoholic → return success if within alcoholic(A)
4   call aux_1
5   exit_on_success
6   call aux_3
7   exit_on_success
8   execute aux_4

9 predicate(aux_1):
10  put_ancestor not_alc      → uses A, sets H
11  has_n_successors has_parent 1 → return fail if A has no parent at all
12  label(1):
13    call not_alc,
14    count_and_exit          → returns if we found a not_alc parent
15    next_choice             → return fail if no parent belongs to not_alc
16    jump 1

```

4 The Parallel DLog Architecture

We now identify several parallelisation possibilities in executing DLog programs. First, we briefly summarise the main ideas behind how to turn the DAM into a parallel execution engine – this is a very high level discussion and its purpose is to give an insight to the possibilities. Next, we introduce in detail a new parallel architecture for the DLog system.

4.1 Kinds of Parallelism available in DLog

The DLog Abstract Machine, discussed in the previous section, can be viewed as a simplification of the WAM for the case of special Prolog programs, produced by the DLog transformation. Analogously, we can simplify well-studied parallelisation techniques for DLog programs. Combining these two ideas produces the *Parallel DLog Abstract Machine* (PADAM).

Logic programming offers an excellent ground for parallel execution. On one hand, the operational semantics of Logic Programming includes *non-determinism* which leads to a very natural parallelisation. On the other hand, logic programs use single assignments for variables, which makes it possible to avoid the problems of certain types of flow dependencies present in more traditional languages.

There are two main kinds of parallelism applicable for Logic Programs: AND- and OR- parallelism. The former consists of the simultaneous computation of several goals in a body, while the latter relies on executing the clauses of a predicate in parallel. Because of the decomposition DLog applies at compile time we already have the independent components of a clause body allowing us to apply independent AND-parallelism (IAP) techniques directly.

An interesting feature of DLog programs is that each clause contains a *cut* operation at the very end: thus once a clause succeeds we can terminate the

computation of all the other clauses being executed in parallel. This means that exploiting OR-parallelism within a single instance check involves *speculative work* [10], i.e. wasted computer efforts, which may be lost because of being on a branch of computation pruned by a cut operation.

As OR-parallelism is easier to exploit, our initial efforts go in this direction. Because of the speculative nature of fine grained OR-parallelism, we decided to first address the issue of coarse grained parallelism, by executing multiple instance check problems in parallel.

4.2 An initial coarse-grained model of parallelism

We now focus on exploiting coarse-grained parallelism, which does not require the modification of the sequential DAM engine, and still seems to promise good scalability. The proposed architecture of the parallel DLog system is presented in Figure 2. We explain the workflow of the system, which starts with the receipt of the input and completes with producing the answer to the instance retrieval query. We refer to certain parts of the architecture by using the numbers and letters in Figure 2.

The content of the ABox is stored either in a triple store or in a relational database (arrow 1). As a general consideration, we assume that the ABox is extensionally reduced, i.e. beside roles, it contains only atomic concepts and their negations. When using a database we need to create appropriate tables for the concept and role assertions, taking care that tables should also be created for negated concepts (to properly model incomplete knowledge). If we use the DLog system over an existing database we need to create links between the concept and role names in the TBox and the database tables [11].

The content of the TBox is first transformed into DL clauses (arrow 2). These clauses are then further transformed into DAM byte code (arrow A) using the specialised PTTP techniques and optimisations outlined in Section 2. Specifically, the DAM code contains the component clause bodies and the superset expressions (i.e. an expression whose *evaluation* gives the superset) for each predicate. The DAM programs are stored in a repository (arrow B) allowing us to re-use them without performing the transformation steps again.

The conjunctive query, i.e. a conjunction of possibly negated atomic concepts and unnegated (positive) binary roles, first undergoes an optimisation phase (arrow 3) inspired by database join optimisations and [12]. Here, using heuristics, the conjunctions can be re-ordered and grouped in order to avoid cross product computations and ensure efficient execution.

As the last step of the ABox independent transformations, the generated DAM program is simplified/partially evaluated with respect to the given query (arrow C). Practically this means that we leave out those parts of the DAM program that do not play any role when executing the specific query, thus reducing the size of the byte code that needs to be transferred between workers in a later stage of the execution. We also “pack” the query itself into the DAM code together with a newly constructed superset expression belonging to it.

In the next step, the Superset Builder receives the simplified byte code (arrow 4) and calculates the superset for the given query. One of the key ideas here is

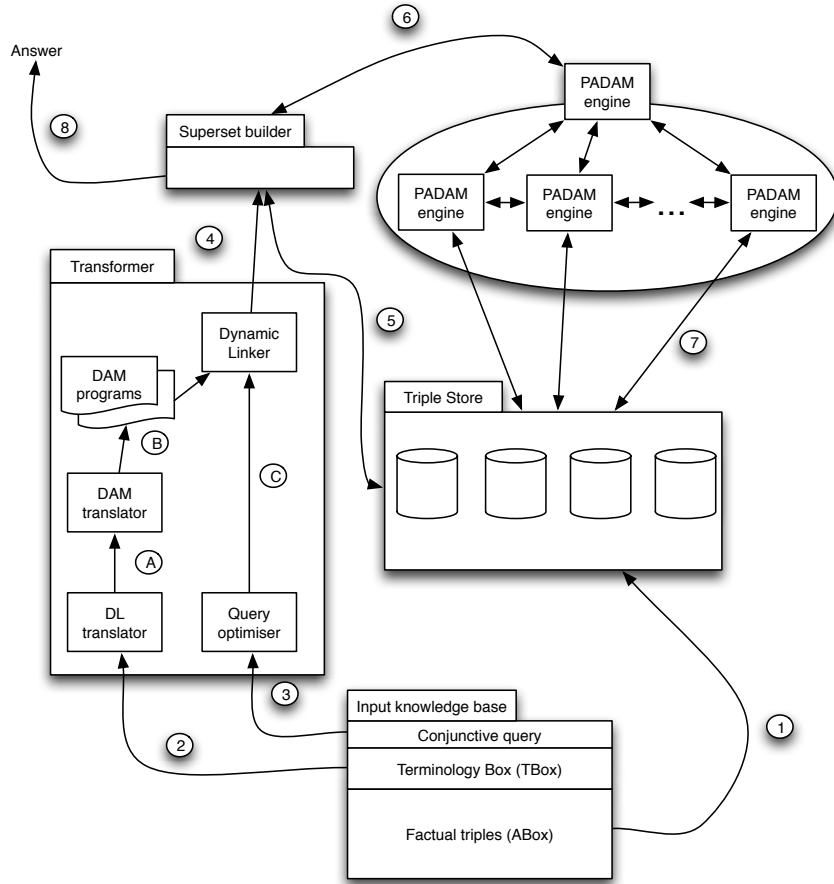


Fig. 2. The architecture of the Parallel DLog system.

that (i) the superset expression is evaluated in parallel and (ii) the instances in the superset are checked in parallel. Note that this scheme can be used for supersets which do not fit into memory: the Superset Builder basically acts as a mediator in the producer-consumer setup between the database and the PADAM execution engines (arrows 5 and 6). The parallel DAM engines also use the database when checking particular individuals (arrow 7).

During the execution the Superset Builder receives notifications from the PADAM engines about whether particular individuals are solutions or not. This information is forwarded to the output of the reasoning process (arrow 8) where another consumer can pick it up and use it for various purposes.

5 Related Work

Because of the separation of the TBox and ABox reasoning and the usage of a database to store ABox facts, the suggested parallel DLog architecture

can be considered as a Deductive Database system. Here we heavily rely on the scalability capabilities of the underlying relational databases/triple stores. We argue that this is the right way to do as there are decades long expertise and proved implementations for databases and triple stores that scale up to several billions of records [13]. Note that this setup is actually suggested by the database literature as well. For example, [14] discusses the problem of handling massive amounts of data in a relational database with support for recursive queries. The suggested solution exploits optimisation techniques both from the relational database as well as from the deductive database theory – much like the parallel DLog architecture.

From the DL point of view, several techniques have emerged for dealing with large scale ABox-reasoning. We can divide these into two main groups, based on whether they support some full-featured DL subset or whether they pose restrictions on the terminology axioms in order to retain even better scalability.

Full reasoners To make traditional tableau-based reasoning more efficient on large data sets, several techniques have been developed in recent years [12]. These are used by the state-of-the-art DL reasoners, such as RacerPro or Pellet.

In [15], a resolution-based inference algorithm is described, which is not as sensitive to the increase of the ABox size as the tableau-based methods. However, this approach still requires the input of the *whole content* of the ABox before attempting to answer any queries. The KAON2 system implements this method and provides reasoning services over the description logic language *SHIQ* by transforming the knowledge base into a disjunctive datalog program.

Article [16] introduces the notion of distributed ordered resolution which is a parallelised variant of ordered resolution usable for reasoning over already distributed *ALC* knowledge bases.

DLog belongs to this first group: it provides *SHIQ* support while retains as much scalability as possible. The original DLog system was evaluated in depth in [1]. There we compared the performance of DLog with that of the RacerPro, Pellet and KAON2 systems on publicly available benchmark ontologies. The test results showed that DLog is significantly faster than traditional tableau-based reasoners and it also outperforms KAON2 in most of the test cases.

Restricted reasoners Extreme cases involve serious restrictions on the knowledge base to ensure efficient execution with large amounts of instances. For example, [17] suggests a solution called the *instance store*, where the ABox is stored externally, and is accessed in a very efficient way. The drawback is that the ABox may contain only axioms of form $C(a)$, i.e. we cannot make role assertions.

The YARS2 framework [13] aims to provide efficient support for the upcoming OWL2 QL profile that is based on a DL-Lite variant. DL-Lite [18] is a family of Description Logics that allows the separation of TBox and ABox during reasoning and provides polynomial-time data complexity for query answering. On the other hand, DL-Lite poses restrictions on the terminology axioms, e.g. cardinality constraints are not allowed.

Article [19] introduces the term Description Logic Programming (DLP). This idea uses a direct transformation of *ALC* description logic concepts into def-

inite Horn-clauses, and poses some restrictions on the form of the knowledge base, which disallow axioms requiring disjunctive reasoning. As an extension, [20] introduces a fragment of the *SHIQ* language that can be transformed into Horn-clauses where queries can be answered with polynomial data complexity.

Work in [21] presents the DLDB2 system based on the DLP idea that delegates certain reasoning tasks to an external TBox reasoner. Similarly to our approach, DLDB2 takes advantage of the scalability of the underlying relational database. However it poses serious restrictions on the supported language (e.g. universal restrictions are not allowed).

6 Conclusion and Future Work

In this paper we have presented the design of the scalable extension of the description logic *SHIQ* reasoning system DLog. Unlike the traditional tableau-based approach, we answer conjunctive queries by transforming the knowledge base into a logic program that is executed in a distributed fashion. This technique allows us to use top-down query execution and to store the content of the ABox externally in a scalable/distributed database.

Following an overview of the DLog system we presented the design considerations of an abstract machine aimed to execute DLog programs. Based on this we then proposed a parallel architecture that introduces parallelism at several stages of the execution process. The key ideas were to calculate the superset of a query in parallel and to check the individuals in the superset using instances of our proposed abstract machine communicating in a peer-to-peer fashion.

Future work involves designing the details of the communication between DLog and the triple store, the implementation and the performance evaluation of our initial parallel DLog model. Building on these results the model should be further refined, including the exploration of parallelism within a single instance check reasoning task (cf. Section 4.1).

As an overall conclusion, we argue that resolution-based techniques are very promising in practical applications, with relatively small TBox, but large ABox. Specifically, we believe that translating to Logic Programs and using the parallel DLog architecture provides a viable framework for scalable DL reasoning.

References

1. Lukácsy, G., Szeredi, P.: Efficient description logic reasoning in Prolog: the DLog system. *Theory and Practice of Logic Programming* **09**(03) (May 2009) 343–414
2. Hustadt, U., Motik, B., Sattler, U.: Data Complexity of Reasoning in Very Expressive Description Logics. In Kaelbling, L.P., Saffiotti, A., eds.: *Proc. of the 19th Int. Joint Conference on Artificial Intelligence (IJCAI 2005)*, Edinburgh, UK, Morgan Kaufmann Publishers (July 30–August 5 2005) 466–471
3. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F., eds.: *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press (2004)
4. Nilsson, U., Maluszynski, J., eds.: *Logic, Programming and Prolog*. John Wiley and Sons Ltd. (1990)

5. Nagy, Zs., Lukácsy, G., Szeredi, P.: Translating description logic queries to Prolog. In Hentenryck, P.V., ed.: *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL 2006)*, Charleston, South Carolina, USA. Volume 3819 of *Lecture Notes in Computer Science.*, Springer Verlag (January 9-10 2006) 168–182
6. Zombori, Zs.: Efficient two-phase data reasoning for description logics. In Bramer, M., ed.: *IFIP AI. Volume 276 of IFIP.*, Springer (2008) 393–402
7. Stickel, M.E.: A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science* **104**(1) (1992) 109–128
8. Kádár, B.: Architectural extensions of the dlog description logic reasoning system MSc Thesis. <http://sintagma.szit.bme.hu/lukacsy/docs/kadarMSc.pdf>.
9. Warren, D.H.D.: An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA (October 1983)
10. Gupta, G., Pontelli, E., Ali, K.A., Carlsson, M., Hermenegildo, M.V.: Parallel execution of Prolog programs: a survey. *ACM Trans. Program. Lang. Syst.* **23**(4) (2001) 472–602
11. Kádár, B., Lukácsy, G., Szeredi, P.: Large scale semantic web reasoning. In: *Proceedings of the 3rd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS2008)*, Udine, Italy. (December 2008) 57–70
12. Haarslev, V., Möller, R.: On the scalability of description logic instance retrieval. *Journal of Automated Reasoning* **41**(2) (August 2008) 99–142
13. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A federated repository for querying graph structured data from the web. In et al., K.A., ed.: *6th International Semantic Web Conference, 2nd Asian Semantic Web Conference. Volume 4825 of Lecture Notes in Computer Science.*, Berlin, Germany, Springer-Verlag (October 2007) 211–224
14. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. *Theory Practice of Logic Programming* **8**(2) (2008) 129–165
15. Hustadt, U., Motik, B., Sattler, U.: Reasoning for Description Logics around SHIQ in a resolution framework. Technical report, FZI, Karlsruhe (2004)
16. Schlicht, A., Stuckenschmidt, H.: Towards distributed ontology reasoning for the web. In: *International Conference on In Web Intelligence and Intelligent Agent Technology (WI-IAT '08)*. Volume 1. (December 2008) 536–539
17. Horrocks, I., Li, L., Turi, D., Bechhofer, S.: The Instance Store: DL reasoning with large numbers of individuals. In: *Proceedings of DL2004*, British Columbia, Canada. (2004)
18. Calvanese, D., Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The dl-lite family. *J. Autom. Reason.* **39**(3) (2007) 385–429
19. Grosz, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logic. In: *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, ACM (2003) 48–57
20. Hustadt, U., Motik, B., Sattler, U.: Data complexity of reasoning in very expressive description logics. In: *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, International Joint Conferences on Artificial Intelligence (2005) 466–471
21. Pan, Z., Zhang, X., Heflin, J.: DLDB2: A scalable multi-perspective semantic web repository. In: *ACM International Conference on Web Intelligence*, IEEE (December 2008) 489–495