Approximate Instance Retrieval on Ontologies*

Tuvshintur Tserendorj[1], Stephan Grimm[1], and Pascal Hitzler[2]

[1] FZI Research Center for Information Technology, Karlsruhe, Germany
[2] Kno.e.sis Center, Wright State University, Dayton, Ohio

Technical Report, Kno.e.sis Center, Wright State University, Dayton, Ohio, 2010.

Abstract. With the development of more expressive description logics (DLs) for the Web Ontology Language OWL the question arises how we can properly deal with the high computational complexity for efficient reasoning. In application cases that require scalable reasoning with expressive ontologies, non-standard reasoning solutions such as approximate reasoning are necessary to tackle the intractability of reasoning in expressive DLs. In this paper, we are concerned with the approximation of the reasoning task of instance retrieval on DL knowledge bases, trading correctness of retrieval results for gain of speed. We introduce our notion of an approximate concept extension and we provide implementations to compute an approximate answer for a concept query by a suitable mapping to efficient database operations. Furthermore, we report on experiments of our approach on instance retrieval with the Wine ontology and discuss first results in terms of error rate and speed-up.

1 Introduction

For description logics, there are two main approaches to reasoning. Tableaux-based methods [1] implemented in tools such as Pellet [2] and Racer [3] have been shown to be efficient for complex TBox reasoning tasks with expressive DLs. In contrast, the reasoning techniques based on reduction to disjunctive datalog as implemented in KAON2 [4] scale well for large ABoxes, with support for the DL $\mathcal{SHIQ}$. Besides these two directions, other approaches such as rule engines and database-based techniques scale very well for large ABoxes, but are in principle limited to lightweight language fragments [5].

Observing the application domain of these approaches, an issue which remains to be investigated is the problem of scalable reasoning over expressive ontologies with large ABoxes as well as complex or large TBoxes. From a theoretical point of view we know that it is impossible to find any tractable algorithm for reasoning over expressive ontologies due to the underlying high computational complexities [6]. Thus, non-standard reasoning solutions like approximate reasoning [7, 8] are helpful in time-critical applications when it is acceptable to sacrifice soundness or completeness for increased efficiency. Approximate reasoning algorithms can be tractable although the underlying language is not, in contrast to limiting attention only to inexpressive tractable fragments as e.g. [9].

* Research reported in this paper was supported by the the German Federal Ministry Economics (BMWi) under the Theseus project, `http://theseus-programm.de`.

Investigations into approximate reasoning usually start from a sound and complete algorithm and system and directly addresses performance bottlenecks in order to improve efficiency, i.e. the algorithms are altered, leading to approximate outputs, while improving speed and keeping the introduced error ratio as low as possible.

In previous work [10, 11] we have shown how to approximate instance retrieval for named classes within the KAON2 approach. In this paper we show how instance retrieval for complex classes can be approximated by reducing it to instance retrieval for named classes. For this, we compute what we call *approximate extensions* of complex classes by means of combining extensions of named classes, e.g. by using standard database operations. The approach leads to a speedup of about factor 10, while the number of introduced errors varies depending on the query, but is within reasonable bounds.

The present paper is structured as follows. After recalling necessary preliminaries on description logics, we present our definition of approximate query answering. Then we describe our approximate algorithms and report on corresponding evaluations. We conclude with some ideas for further work.

2 Preliminaries

In this section we introduce and recall some formal notions used throughout the paper. Besides basic notions from description logics we also cover relational algebra notation for the description of our database implementation.

Description Logics. Description logics (DLs) are a family of knowledge representation formalisms that provide the formal basis for the Web Ontology Language (OWL) and allow for sophisticated reasoning about ontologies in the Semantic Web. The basic constituents to represent knowledge in DLs are *concepts* C, *roles* r and individuals a. They are used to form *axioms* collected in a *knowledge base* KB to make statements about a domain of interest. We primarily consider concept and role assertion axioms of the form $C(a)$, $r(a, b)$ that assign an individual to a concept or relate two individuals via a role, and concept inclusion axioms of the form $C \sqsubseteq D$ that state subclass relationships. For a detailed presentation of DLs we refer to [12].

While there are many DL dialects, we recall the syntax of concept expressions in the description logic $\mathcal{SHIQ}$, where complex concepts C and roles r are produced from the following grammar that involves named concepts A and roles p, role inverses, conjunction, disjunction, complements, restricted existential and universal quantification as well as qualified cardinality restrictions.

$$\begin{aligned}&C \to A \mid \bot \mid \top \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \exists r.C \mid \forall r.C \mid \geq n\, r.C \mid \leq n\, r.C\\ &r \to p \mid p^-\end{aligned}$$

The *signature* of a knowledge base KB, denoted by $\sigma(\mathit{KB})$, is the set of all individual, concept and role names that occur in the axioms within KB. In particular, $\sigma(\mathit{KB})$ comprises all individuals occurring in KB.

Instance retrieval with DL knowledge bases builds on the standard reasoning task of instance checking. An individual $a \in \sigma(\mathit{KB})$ is an instance of a concept C with respect to a knowledge base KB if the axiom $C(a)$ is a logical consequence of KB, which is denoted by $\mathit{KB} \models C(a)$. Instance retrieval can be interpreted as the repeated application of instance checking for all known individuals of KB and a given concept. We call the result of retrieving all instances of concept C from KB the *(conventional) extension* of C with respect to KB, denoted by $|C|$, and define it as follows.[3]

$$|C| := \{x \in \sigma(\mathit{KB}) \mid \mathit{KB} \models C(x)\}$$

In the context of instance retrieval, the concept C is often also called the *query*.

Relational Algebra. *Relational Algebra* is the formal underpinning of modern relational database systems and is used to formalise database operations on the relational model originally introduced by Codd [13]. The main construct for representing data in the relational model is a *relation*, denoted by $R(a_1, \ldots, a_n)$, that represents a database table with column attributes a_1 to a_n and rows that instantiate the columns as tuples of values. Relational algebra expressions are used to formulate queries on the thus represented database tables and result themselves in relations, such that expressions can be nested. Attributes in a relation can be referred to by means of path expressions of the form $R\,.\,a_i$, e.g. within conditions.

We briefly recall the relational operators that are used in this paper. A *projection* $\pi_{[a_1,\ldots,a_m]}(R(a_1, \ldots, a_n))$ restricts the columns of the resulting relation to the attributes $a_1, \ldots, a_m$ for $m < n$. A *selection* $\sigma_{[\texttt{condition}]}(R(a_1, \ldots, a_n))$ selects those rows for which `condition` holds. A *cross product* $R_1(a_1, \ldots, a_n) \times R_2(b_1, \ldots, b_m)$ generates a combined relation $R(a_1, \ldots, a_n, b_1, \ldots b_m)$ in the sense of the Cartesian product by multiplying rows, which is used for join operations.[4] Other set operations are used for relations as usual, namely *union* $R_1 \cup R_2$, *intersection* $R_1 \cap R_2$ and *difference* $R_1 \setminus R_2$, operating on relation tuples in the usual way. For a detailed description of relational algebra see e.g. [14].

3 Approximation of Instance Retrieval

Our approach for the approximation of instance retrieval queries is based on the notion of the *approximate extension* $\langle C \rangle$ of a concept C with respect to a knowledge base KB. Intuitively, $\langle C \rangle$ is the set of instances that are obtained through interpreting complex concepts in C as simple set operations on the individuals

[3] Notice that this notion of extension refers to a particular knowledge base and is different from the model-theoretic notion of extension defined for an interpretation, which we do not use in this paper.

[4] In relational algebra there is a special notation for database joins using the symbol $\bowtie$. However, for simplicity we present join operations by combinations of cross product with selection.

Table 1. Definition of an approximate extension. A stands for atomic classes while C and D stand for complex (non-atomic) classes. R stands for roles and n for a natural number.

Approximate Extensions		
$\langle \top \rangle$	$=$	$\|\top\|$
$\langle \bot \rangle$	$=$	$\emptyset$
$\langle A \rangle$	$=$	$\|A\|$
$\langle \neg A \rangle$	$=$	$\|\neg A\|$
$\langle R \rangle$	$=$	$\{(x,y) \mid KB \models r(x,y)\}$
$\langle R^- \rangle$	$=$	$\{(x,y) \mid KB \models r(y,x)\}$
$\langle C \sqcap D \rangle$	$=$	$\langle C \rangle \cap \langle D \rangle$
$\langle C \sqcup D \rangle$	$=$	$\langle C \rangle \cup \langle D \rangle$
$\langle \neg C \rangle$	$=$	$\langle \top \rangle \setminus \langle C \rangle$
$\langle \exists R.C \rangle$	$=$	$\{x \in \langle \top \rangle \mid \exists y : (x,y) \in \langle r \rangle \wedge y \in \langle C \rangle\}$
$\langle \forall R.C \rangle$	$=$	$\{x \in \langle \top \rangle \mid \forall y : (x,y) \in \langle r \rangle \rightarrow y \in \langle C \rangle\}$
$\langle \leq n\, R.C \rangle$	$=$	$\{x \in \langle \top \rangle \mid \#\{y \mid (x,y) \in \langle r \rangle \wedge y \in \langle C \rangle\} \leq n\}$
$\langle \geq n\, R.C \rangle$	$=$	$\{x \in \langle \top \rangle \mid \#\{y \mid (x,y) \in \langle r \rangle \wedge y \in \langle C \rangle\} \geq n\}$

known to KB, starting from the atomic extensions of concepts and roles that occur in C. In this way, the model-theoretic semantics of DLs is approximated by a straightforward combination of results for atomic queries that requires less effort to compute than the reasoning process for complex instance retrieval queries in DLs does. The exact definition of an approximate extension is given in Table 1 recursively for all language constructs. For an example, consider the knowledge base $KB = \{C \sqsubseteq A \sqcup B, A(a_1), C(a_2)\}$ and the instance retrieval query $A \sqcup B$. The conventional extension of the concept $A \sqcup B$ contains both individuals a_1 and a_2, i.e. $|A \sqcup B| = \{a_1, a_2\}$. However, the approximate extension of $A \sqcup B$ contains only a_1, i.e. $\langle A \sqcup B \rangle = \{a_1\}$.

The more complex the query concept C is, the more the approximate extension deviates from the conventional extension. For the simplest queries, such as atomic concepts, the two types of extensions coincide and no errors are made in instance retrieval. This characteristics is captured by the following proposition.

Proposition 1 (soundness and completeness of simple approximate extensions). *For a knowledge base KB and a concept C of the form $C = A_1 \sqcap \cdots \sqcap A_m \sqcap \neg B_1 \sqcap \ldots \neg B_n$, with all A_i and B_j atomic, the approximate extension of C is equivalent to its conventional extension, i.e. $\langle C \rangle = |C|$.*

Proposition 1 states that, for queries that have the form of conjunctions of possibly negated named concepts, the approximate and conventional extensions have exactly the same instances. In other words, computing the approximate extension is *sound and complete* with respect to the conventional extension.

For more complex queries, however, the approximation might deviate significantly from the correct answer in both that it might miss instances as well as show improper instances. In particular the approximation of the complement constructor is supposed to cause significant deviation as it interprets negation in a closed-world sense, potentially including improper instances in an answer. Hence, we aim at eliminating general complements by means of normalisation, avoiding this source of error.

For standard reasoning in DLs a query concept can be expressed in various normal forms and semantics-preserving transformations do not affect the result of instance retrieval. For the calculation of approximate extensions, however, the result depends on the form of the concept, and different semantically equivalent concept expressions can have different approximate extensions. We can exploit this characteristics by choosing a normal form for query concepts that fits best the process of approximation in terms of both error rate and ease of computation. In this light, we consider the negation normal form [15] of concept expressions for queries, denoted by $\mathsf{NNF}(C)$ for a concept C, in which negation symbols are pushed inside to occur only in front of atomic concepts. This eliminates the case of considering the approximation for general complements with its rather drastic closed-world interpretation. Besides the lower expected error rate this also avoids the computationally costly handling of large sets of individuals in case of large ABoxes by an algorithm that computes approximate extensions. The positive effect that elimination of complement approximation has on the error rate in instance retrieval can be expressed by the following property, which ensures that approximation of concepts in negation formal form only gives up completeness but preserves soundness at least for a certain class of queries.

Proposition 2 (soundness of limited approximate instance retrieval). *Let KB be a knowledge base and C be a concept such that $\mathsf{NNF}(C)$ contains no $\forall$- and no $\leq$- and $\geq$-constructs. The approximate extension of $\mathsf{NNF}(C)$ only contains instances that are also contained in the conventional extension of C with respect to KB, i.e. $\langle \mathsf{NNF}(C)\rangle \subseteq |C|$.*

Proposition 2 states that, for queries that do not make use of the $\forall$, $\geq$ and $\leq$ constructs (after normalisation), the approach of approximating concepts in their negation normal form yields an extension that might miss some instances but has no improper instances in it. In other words, computing the approximate extension is *sound* with respect to the conventional extension.

4 Computing Approximate Extensions

In this section, we will design algorithms for computing the approximate extension of a query concept. We will lay out the architecture of a system for approximate instance retrieval and elaborate on two implementations of the algorithms, one in a database and one in memory.

4.1 System Architecture

Our system for approximate instance retrieval takes as input a $\mathcal{SHIQ}$[5] knowledge base KB and a complex query concept Q to compute the approximate

[5] We use $\mathcal{SHIQ}$ since we build on KAON2 for our experimental results. However, our approximation approach can easily be extended to nominals, the missing feature for handling OWL ontologies.

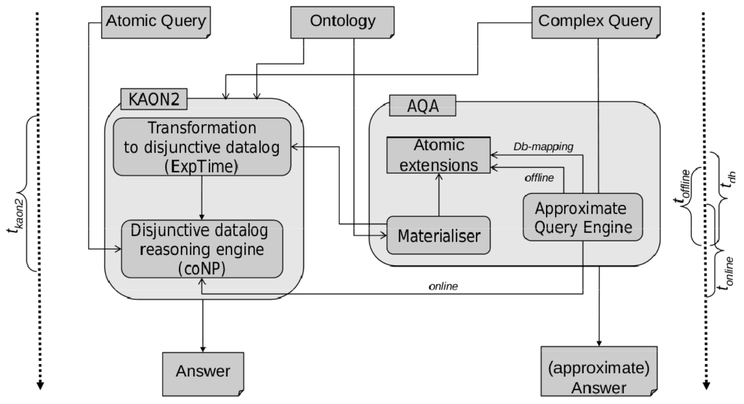

Fig. 1. An overview of the system architecture

extension of Q with respect to KB as a set of individuals. This is depicted on the right-hand side of Figure 1. The principle behind computing $\langle Q \rangle$ is always to start from the individuals in the conventional extensions of (possibly negated) atomic concepts and (possibly inverse) roles that occur in Q and to recursively combine these according to the structure of concepts in Q, reflecting the set operations from Table 1. According to Propositions 1 and 2, this results in an answer that is sound and complete for some cases, only sound for others, or neither sound nor complete, depending on the language constructs used in the query.

We have implemented the approximate instance retrieval method in two different ways and distinguish between *database* and *in-memory* computation: in the first case computation is delegated to underlying database operations, whereas in the second case it is performed in main memory. While for the database variant the atomic extensions are pre-computed prior to query-time and materialised in the database using a sound and complete reasoner, the in-memory variant allows for two possibilities to access the atomic extensions: in *online processing* a sound and complete DL reasoner is invoked at query-time to compute atomic extensions, while in *offline processing* they are again pre-computed and materialised either in a database or in memory if possible. Database computation and offline processing are very useful when dealing with large amounts of data in scenarios with frequent querying on rather static ontologies, for which materialisation can be done in advance. Online processing is intended to be used in such cases where a materialisation is hardly manageable as ontologies are subject to frequent changes.

For online processing we utilise the KAON2 reasoner, as illustrated in Figure 1. The reason for this choice is that KAON2 was designed to be an efficient ABox reasoner on knowledge bases with large ABoxes and simple TBoxes in comparison to other state-of-art DL reasoners, which typically perform better on knowledge bases with large (or complex) TBoxes and small ABoxes. As depicted

Concept Expression	*Relational Algebra Expression*
$\tau_{\mathsf{db}}(A)$	$\pi_{[ind]}(\sigma_{[class=A]}(\mathsf{Ext}^C))$
$\tau_{\mathsf{db}}(\neg A)$	$\pi_{[ind]}(\sigma_{[class=\neg A]}(\mathsf{Ext}^C))$
$\tau_{\mathsf{db}}(\exists r.C)$	$E_C := \tau_{\mathsf{db}}(C)$ $E_r := \sigma_{[role=r]}(\mathsf{Ext}^r)$ $\pi_{[ind_1]}(\sigma_{[ind=ind_2]}(E_C \times E_r))$
$\tau_{\mathsf{db}}(\forall r.C)$	$E_C := \tau_{\mathsf{db}}(C)$ $E_r := \sigma_{[role=r]}(\mathsf{Ext}^r)$ $E_{\_} := \pi_{[ind_1]}(\sigma_{[ind \neq ind2]}(E_C \times E_r))$ $\pi_{[ind_1]}(\mathsf{Ext}^C) \setminus E_{\_}$
$\tau_{\mathsf{db}}(\leq n\, R.C)$	$E_C := \tau_{\mathsf{db}}(C)$ $E_r := \sigma_{[role=r]}(\mathsf{Ext}^r)$ $\pi_{[ind]}(\sigma_{[count(ind_1) \leq n \ \wedge \ ind=ind2]}(E_C \times E_r))$
$\tau_{\mathsf{db}}(\geq n\, R.C)$	$E_C := \tau_{\mathsf{db}}(C)$ $E_r := \sigma_{[role=r]}(\mathsf{Ext}^r)$ $\pi_{[ind]}(\sigma_{[count(ind_1) \geq n \ \wedge \ ind=ind2]}(E_C \times E_r))$
$\tau_{\mathsf{db}}(C_0 \sqcap C_1 \sqcap \cdots \sqcap C_n)$	$\tau_{\mathsf{db}}(C_0) \cap \tau_{\mathsf{db}}(C_1) \cap \cdots \cap \tau_{\mathsf{db}}(C_n)$
$\tau_{\mathsf{db}}(C_0 \sqcup C_1 \sqcup \cdots \sqcup C_n)$	$\tau_{\mathsf{db}}(C_0) \cup \tau_{\mathsf{db}}(C_1) \cup \cdots \cup \tau_{\mathsf{db}}(C_n)$

Table 2. Mapping of DL concept expression to Relational Algebra Expression

on the left-hand side of Figure 1, KAON2 transforms the TBox together with complex queries into a disjunctive datalog program in a first step, to perform ABox reasoning in a second step based on the result of this transformation. Hence, for every complex ABox query KAON2 needs to repeatedly perform the TBox transformation, which is computationally costly. For ABox queries that have the form of atomic concepts, however, this transformation is not necessary and can be bypassed. For the variant with in-memory and online processing we can take advantage of this because for computing atomic extensions with KAON2 the costly TBox translation is saved.

In Figure 1, we introduce the running times for various steps in the query execution, which will be used in the remainder of this paper. We refer to the time required to compute conventional extensions with KAON2 as t_{kaon2}. Furthermore, we refer to the time taken for the computation with involving the DL reasoner as t_{online}, the time for that with the materialisation as $t_{offline}$ and the time for the database variant as t_{db}.

4.2 Delegation of Computation to Database

The variant that performs database computation is a presumably efficient implementation of approximate instance retrieval as the pre-computed atomic extensions are materialised and approximate extensions are computed by making use of highly optimised database operations. This variant is essential in practice for handling ontologies with large ABoxes that cannot be processed efficiently in memory. Here, the recursive combination of atomic extensions in terms of set operations as defined in Table 1 is completely delegated to the underlying database, which benefits performance. As a basis for this form of computation we use a database schema that consists of two Relations, namely $\mathsf{Ext}^C(ind, class)$ for storing concept extensions and $\mathsf{Ext}^r(ind_1, role, ind_2)$ for storing role extensions. In their schema, the attribute $ind_{(i)}$ stands for individual names, *class* for names of possibly negated concepts and *role* for names of possibly inverse roles. Starting from a knowledge base *KB*, these two relations are initialised as follows.

$$\mathsf{Ext}^C(ind, class) = \{(a, C) \mid KB \models C(a)\},\ \textit{for } C = A \mid \neg A \textit{ with } A \in \sigma(KB)$$
$$\mathsf{Ext}^r(ind_1, role, ind_2) = \{(a, r, b) \mid KB \models r(a, b)\},\ \textit{for } r = p \mid p^- \textit{ with } p \in \sigma(KB)$$

Notice that, for the purpose of approximate instance retrieval, Ext^C and Ext^r form a complete representation of the original knowledge base *KB*.

A complex query concept Q is answered by transforming its negation normal form $\mathsf{NNF}(Q)$ into a relational algebra expression according to a mapping τ_{db} posed as a query to the underlying database system. The complete mapping definition for τ_{db} is given in Table 2. The left-hand side shows the concept constructors that can occur in $\mathsf{NNF}(Q)$ and the right-hand side shows their respective relational algebra expression. Recursive application of τ_{db} ultimately produces a single database query $\tau_{\mathsf{db}}(\mathsf{NNF}(Q))$ that is used for computing $\langle Q \rangle$.

For an example consider the query $Q = A \sqcap \exists r.\neg B$. The mapping τ_{db} produces the following nested relational algebra expression.

$$\begin{aligned}\tau_{\mathsf{db}}(Q) = {} & \pi_{[ind]}(\sigma_{[class=A]}(\mathsf{Ext}^C)) \cap \\ & \pi_{[ind_1]}(\sigma_{[ind=ind_2]}(\sigma_{[role=r]}(\mathsf{Ext}^r) \times \pi_{[ind]}(\sigma_{[class=\neg B]}(\mathsf{Ext}^C)))) \ .\end{aligned}$$

When posed to the underlying database, this rather large expression is subject to efficient internal query optimisation strategies as they are typically employed by database systems.

4.3 In-memory Computation

Both the variants with online and offline processing share the same implementation of the approximate algorithm of which the pseudocode is described in Algorithm 1. The difference is the handling of atomic extensions which is presented by the function $\mathsf{Compute\_Ext}$. This function takes as parameters a knowledge base and an atomic concept or atomic role for which the atomic extension is to be computed while the algorithm ψ accepts the knowledge base and a complex concept query for which the approximate extension is to be computed.

For the computation of the atomic extension, depending on the chosen variant, $\mathsf{Compute\_Ext}$ invokes either a complete and sound reasoner or retrieves the atomic extension from the database. In the following, we exemplarily describe the most common part of the algorithm. For concepts being of the form $\exists r.C$, it first computes the atomic extension of the role r, represented by $\mathcal{R}$. Next, it recursively calculates the approximate extension of the concept C, which is assigned to a temporary set $\mathcal{C}$. Then, it determines such role assertions in $\mathcal{R}$ whose second component belongs to $\mathcal{C}$ and returns the set of their first components. A similar procedure is used in the calculation of $\forall r.C$ and cardinality restrictions.

However, for cardinality restrictions the number of explicit role fillers is taken into account and for $\forall r.C$ the set difference of $\langle\top\rangle$ and $\mathcal{B}$.

Algorithm 1: $\psi(KB, Q)$

```
Data: $\mathcal{SHIQ}$ knowledge base $KB$ and a complex query concept $Q$ in NNF
Result: The approximate extension of $Q$
if $Q$ is of the form $A$ or $\neg A$ then
    return Compute_Ext($KB$, $Q$)
else if $Q$ is of the form $\exists r.C$ then
    $\mathcal{T} := \emptyset$; $\mathcal{R} :=$ Compute_Ext($KB, r$); $\mathcal{C} := \psi(KB, C)$;
    for $(x, y) \in \mathcal{R}$ do
        if $y \in \mathcal{C}$ then
            $\mathcal{T} := \mathcal{T} \cup \{x\}$
    return $\mathcal{T}$
else if $Q$ is of the form $\forall r.C$ then
    $\mathcal{T} := |\top|$; $\mathcal{B} := \emptyset$; $\mathcal{R} :=$ Compute_Ext($KB, r$); $\mathcal{C} := \psi(KB, C)$;
    for $(x, y) \in \mathcal{R}$ do
        if $y \notin \mathcal{C}$ then
            $\mathcal{B} := \mathcal{B} \cup \{x\}$
    return $\mathcal{T} \backslash \mathcal{B}$;
else if $Q$ is of the form $\geq n\, r.C$ or $\leq n\, r.C$ then
    $\mathcal{T} := \emptyset$; $\mathcal{K} := \emptyset$; $\mathcal{R} :=$ Compute_Ext($KB, r$); $\mathcal{C} := \psi(KB, C)$;
    for $(x, y) \in \mathcal{R}$ do
        if $y \in \mathcal{C}$ then
            $\mathcal{K} := \mathcal{K} \cup \{(x, y)\}$
    if $\geq n\, r.C$ then
        we determine such $x$ holding the property
        $\{x \in |\top| \mid \#\{y \mid (x, y) \in \mathcal{K}\} \geq n\}$ and put it into $\mathcal{T}$
    else if $\leq n\, r.C$ then
        we determine such $x$ holding the property
        $\{x \in |\top| \mid \#\{y \mid (x, y) \in \mathcal{K}\} \leq n\}$ and put it into $\mathcal{T}$
    return $\mathcal{T}$
else if $Q$ is of the form $C_0 \sqcap C_1 \cdots \sqcap C_n$ then
    return $\psi(KB, C_0) \cap \psi(KB, C_1) \cap \cdots \cap \psi(KB, C_n)$;
else if $C$ is of the form $C_0 \sqcup C_1 \cdots \sqcup C_n$ then
    return $\psi(KB, C_0) \cup \psi(KB, C_1) \cup \cdots \cup \psi(KB, C_n)$;
return $\emptyset$;
```

5 Experimental results

We have conducted experiments to determine how effective our approach is at instance retrieval for concept queries. In this section, we present the experimental results obtained from the execution of online and offline processing in the in-memory variant as well as the database variant. The primary metrics we have considered are response time and correctness of the computed extensions in terms of precision and recall.

5.1 Test Data

There are already several well-known benchmarks for evaluating DL reasoning systems. However, objectively comparing the performance of approximate reasoning systems with that of complete and sound DL reasoners is different, and in fact it is not a priori clear how this should best be done. So currently, there exist no generally accepted benchmarks. The point of difficulty is that we do not only want to measure execution time, but we also need to determine empirically to what extent the evaluated algorithms are sound and complete – in terms of precision and recall. In order to do this, we need to test a suitable and large enough sample of queries whilst for comparing complete and sound reasoning systems usually a few complex queries suffice.

This poses the question, however, which of the potentially infinitely many queries should be used for the testing. Obviously, we want to restrict our attention to a finite set, but two general problems arise: it is not possible to do unbiased random selections from an infinite set, and many randomly generated queries would simply result in no answer at all. For our experiments, we thus confine our attention to relatively simple queries such that we have only a finite set to choose from, and we furthermore restrict our attention to those queries that actually produce non-empty answers using a sound and complete DL reasoner. The queries that we use for testing are of the forms $A \sqcap B$ (we call them $\sqcap$-*queries*), $A \sqcup B$ ($\sqcup$-*queries*), $\exists r.A$($\exists$-*queries*), $\forall r.A$ ($\forall$-*queries*) and $\geq nr.A$ ($\geq$-*queries*), where A and B are named classes. In addition to these basic queries, we performed some experiments with complex queries to investigate the runtime behaviour also for a combination of basic constructs as well as effects of error compensation.

Another issue is to choose appropriate test ontologies. Not all well-known benchmarking ontologies are suitable for highlighting the performance of our approximate algorithms. A particular difficulty is to find realistic ontologies that are of sufficient expressivity in terms of TBox constructors, and at the same time have a reasonably sized ABox. Since we consider scalable reasoning over expressive ontologies with large TBoxes and ABoxes, we decided to use the WINE ontology for our evaluation. WINE has originally been designed as a showcase for the expressivity of OWL, and thus is a hard enough task to tackle using reasoning. WINE has 140 named classes, 10 object properties, 123 subconcept relations, 61 concept equivalences, 10127 concept assertions and 10086 role assertions. As queries, we considered about 9876 queries of the forms $A \sqcap B$ and $A \sqcup B$ and 100 queries of the forms $\exists r.A$.

The approximate algorithms are implemented in Java 6.0 using the KAON2 API. Computation times are reported in milliseconds. All tests were performed on a Lenovo laptop with dual 2.40 GHz Intel(R) Core(TM)2 Duo processors, 2GB of RAM (with 1024M heap space allocated to JVM), Ubuntu 8.04, Kernel Linux 2.6.24- 21-generic.

5.2 Results

In our experiments, we compared our algorithms with KAON2 as a sound and complete DL reasoner. The results are summarised in Table 3.

For ∃-queries, we first had to identify a meaningful set of such queries, as randomly generated ones very often had empty extensions. We thus considered only such queries for which KAON2 computed non-empty extensions. This way, we identified 107 ∃-queries for testing. Running the approximation algorithm in the database variant, we obtained a significant performance improvement for each ∃-query, about 90%. The mean time consumed to answer the queries was 274 ms while it was 2789 ms for KAON2. Running the algorithm in offline processing where the approximation is computated in memory, we obtained another significant performance gain, indeed about 99% compared to KAON2. Considering this rather drastic speed-up by one respectively two orders of magnitude, the introduced error appears to be rather mild, namely with a recall of 0.65 and a precision of 1.0 (due to Proposition 2), resulting in an f-measure of 0.79. Note also that we considered only queries that do have a non-empty extension: obviously, queries with no answers are also sometimes used. If we would add such queries to our sample set, then recall would be even better, while precision would be unaffected.

Table 3. Summary of the offline and database variants, summarized over all considered queries. *miss* indicates the elements of the conventional extensions that were *not* found by the approximation, *corr* indicates those that were correctly found, and *more* indicates those that were incorrectly computed to be part of the extension. $\langle C\rangle$ indicates the sum of the sizes of the approximate extensions while $|C|$ indicates the sum of the sizes of the conventional extensions. t_{db} gives the runtime for the database variant, t_{offline} gives the runtime for the offline variant while t_{kaon2} gives the runtime of KAON2 – these times are in ms and are the sums over all considered queries.

query	miss	corr	more	$\langle C\rangle$	$\|C\|$	recall	prec	f-meas	t_{offline}	t_{db}	t_{kaon2}	$\frac{t_{\mathit{offline}}}{t_{kaon2}}$	$\frac{t_{db}}{t_{kaon2}}$
$\exists.r.A$	872	1667	0	1667	2539	0.65	1	0.79	3187	29331	298472	0.01	0.098
$\geq n\, r.C$	872	1667	0	1667	2539	0.65	1	0.79	404	4524	71031	0.005	0.063
$\forall.r.A$	0	3730	5109	8839	3730	1	0.42	0.59	545	13847	173088	0.003	0.080
$A \sqcup B$	0	43050	0	43050	43050	1	1	1	552	11681	312677	0.001	0.037
$A \sqcap B$	0	1476	0	1476	1476	1	1	1	2801	5432	278805	0.01	0.019

We again obtained a significant performance gain for 107 ∀-queries while it shows a precision of 0.42 due to unsoundness. Surprisingly, KAON2 took much time to answer the ∀-queries. For $\geq n\, r.C$-queries (n=1), the performance gain was also significant while recall is the same for ∃-queries due to the number restriction.

For the ⊓-queries we obtained even more favourable results. The performance gain for each query was above 95% while recall, precision and f-measure are 100% as expected by Proposion 1. For the database variant the overall time gain was 98 %, while it was 99% for the offline variant.

For the $\sqcup$-queries we find it remarkable that recall turns out to be 100%, which is coincidental.[6] At the same time, we obtained reasonable speed-up for the database variant, namely 97%, and remarkable 99.9%, i.e. three orders of magnitude, for the offline variant. Turning to the online variant, we obviously obtain the same values for precision and recall as for the offline and database variants.

As for computation time, we expect an improvement over KAON2 if the TBox translation is time-consuming (since the approximate algorithm does not need it) compared to the ABox reasoning part performed with the datalog reasoner. For WINE, the effect is rather small, so we would need an ontology with a TBox translation that is very expensive compared to the datalog reasoning part. However, we did not find any real ontology with this property. In order to show that for some ontologies we would get the desired effect, we thus modified WINE by multiplying the TBox, i.e. we used n renamed copies of the original TBox together with the original ontology (including the ABox). Table 4 shows the results for the original WINE, for WINE with additional 4 copies of the TBox, and for WINE with additional 9 copies of the TBox. Indeed the TBox translation took 2629 ms for WINE, 6674 ms for WINE with 4 TBox copies, and 18223 ms for WINE with 9 TBox copies, showing – as expected – a worse than linear development. When taking WINE with 10 TBox copies, KAON2 in fact was no longer able to translate the TBox.

Table 4. Measured performance for querying 100 $\sqcup$-queries over wine ontology with different TBoxes

ontology	t_{online}	t_{kaon2}	$\frac{t_{online}}{t_{kaon2}}$
WINE	243808	277714	0.88
WINE + 4 TBoxes	551101	922626	0.60
WINE + 9 TBoxes	1055881	2058829	0.51

The results in Table 4 show the desired effect, namely that our approach brings an improvement for ontologies where the TBox translation is very expensive. However, the effect is not as strong as we originally hoped it to be. It should be noted that the online variant could be combined with further methods that we are investigating. In particular it can be combined with the Screech approach discussed in [10, 11], which results in further speed-up. It remains to be investigated, though, how the methods perform when combined. Another option for further improvement of the online variant is to use logic programming systems instead of a datalog reasoner following [16]; this is also under investigation.

Let us briefly discuss the matter of complex queries, i.e. of queries involving more than one class constructor. Generally speaking, answering complex queries with KAON2 is not more expensive than answering simple queries of the form we have discussed so far. For our approximate approach, however, any additional class constructor in the query causes additional computation, namely the retrieval of one or several approximate extensions, and the combination of these

[6] The experiments performed in [11] show that disjunction cannot in general be ignored in WINE without loss of precision or recall.

with the result of the remaining query. The effect of this can be expected to be roughly linear in the number of class constructors. In the end, this means that our approach yields less speed-up for more complex queries, but we also see from the figures in Table 3 that the approximation can still be worthwhile, in particular for the in-memory offline variant.

Table 5. Approximate extensions of complex queries

query (C_i)	miss	corr	more	$\langle C_i \rangle$	$\lvert C_i \rvert$	$t_{offline}$	t_{db}	t_{kaon2}
$C_1 = \exists \mathsf{locatedIn}.(\mathsf{ItalianRegion} \sqcup \mathsf{USRegion})$	1	36	0	36	37	11	123	1900
$C_2 = C_1 \sqcap \mathsf{WhiteWine}$	0	7	0	7	7	15	137	1583
$C_3 = \forall \mathsf{hasSugar}.\mathsf{Dry}$	0	47	109	156	47	5	104	1219
$C_4 = \forall \mathsf{hasSugar}.\mathsf{Dry} \sqcap \mathsf{WhiteWine}$	0	18	2	20	18	9	129	1500
$C_5 = \forall \mathsf{hasSugar}.\mathsf{Dry} \sqcap C_1 \sqcap \mathsf{WhiteWine}$	0	6	0	6	6	23	161	1564

In order to investigate the effects of error compensation for complex queries, we consider some complex concepts presented in Table 5. Consider first the complex concept C_1. The conventional extension of C_1 contains 37 individuals. In contrast, the approximate extension $\langle C_1 \rangle$ only contains 36 individuals; one individual is not found by the algorithm due to its incompleteness for $\sqcup$ and $\exists$-queries. If we combine this query with a more specific concept $WhiteWine$ as in C_2, the missing individual disappears, i.e. the error is being compensated for and we have that $\langle C_2 \rangle = |C_2|$.

As for unsoundness, we consider the query C_3 and its approximate extension. There are many (109) individuals that are computed incorrectly. Note that we have had relatively low precision for $\forall$-queries, in Table 3. If we further constrain this query by a conjunction with the named concept $WhiteWine$ as in C_4, then the amount of the individuals computed incorrectly for the query C_3 drastically reduces from 109 to 2. This shows that also an error due to unsoundness can be compensated for by a combination of constructs in complex queries.

As a final example, we consider the intersection of all the query constructs used above in the query C_5. Here we have full compensation of both kinds of errors as its approximate extension is equal to its conventional one, i.e. $\langle C_5 \rangle = |C_5|$. This suggests that for many practical complex queries our approximation approach yields less errors than indicated for the simple queries in Table 3 due to the effect of compensation, while speed-up in computation (see the columns $t_{offline}$ and t_{db}) is still significant.

6 Related Work

The awareness that approximate reasoning approaches are needed for the Semantic Web is rising within the community. This is witnessed by an increasing number of research results which are explicitly or implicitly in the spirit of approximate reasoning. They range from general or visionary articles [7, 8] to concrete proposals of approximate systems, like the MARVIN and the SOAR systems presented at the Billion Triple Challenge at the 2008 International Semantic Web Conference.[7] We refrain from even attempting a general listing of current approaches.

Concerning work more closely related to the approach presented herein, we would like to remark that the general idea we have presented here is hardly very original. Using materialisations of extensions of named classes into databases for avoiding the expensive use of online reasoning algorithms is indeed a straightforward idea and has been used without further ado when practical cicrumstances seemed to make it feasible. Systematically, the approach has e.g. been used in the Instance Store system [17], which is a sound and complete system based on computing increments on the approximations obtained from the materialisations.

The new perspective we want to take, however, is that approximate extensions can indeed be used in the straightforward way we have laid out to realise approximate instance retrieval. But we also believe that approximate reasoning methods should be evaluated properly before being used to understand their advantages and their limitations, and to this end we have presented the first systematic study of this approach which indeed shows that significant speed-up can be obtained while introducing only few errors in the reasoning process.

7 Conclusion

We have presented an approach to approximate instance retrieval based on approximate extensions. Compared with a complete and sound DL reasoner, our approach can significantly improve the performance of reasoning over expressive ontologies with large ABoxes and TBoxes. We presented several instantiations of our approach resulting online and offline in-memory and database variants. We evaluated the approaches and showed that a significant speed-up of around 90% can be obtained while the number of introduced errors remains relatively small.

Future work includes improvements on the online variant using logic programming engines, further experiments for complex queries, combinations with other approximate reasoning methods, extension to more expressive language features and applications of our approach in suitable use case scenarios.

References

1. Schmidt-Schauß, M., Smolka, G.: Attributive Concept Descriptions with Complements. Artificial Intelligence **48**(1) (1991) 1–26
2. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web (2007) 51–53
3. Haarslev, V., Möller, R.: Racer System Description. In: IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning, London, UK, Springer-Verlag (2001) 701–706

[7] See `http://challenge.semanticweb.org/`.

4. Motik, B.: Reasoning in Description Logics using Resolution and Deductive Databases. PhD thesis, Universität Karlsruhe (2006)
5. Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM – A Pragmatic Semantic Repository for OWL. In Gil, Y., et al., eds.: Web Information Systems Engineering WISE 2005 Workshops, October 2005. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2005) 182–192
6. Tobies, S.: Complexity Results and Practical Algorithms for Logics in Knowledge Representation. PhD thesis, RWTH Aachen, Germany (2001)
7. Fensel, D., van Harmelen, F.: Unifying reasoning and search to web scale. IEEE Internet Computing **11**(2) (2007) 96, 94–95
8. Rudolph, S., Tserendorj, T., Hitzler, P.: What is Approximate Reasoning? In Calvanese, D., Lausen, G., eds.: Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems (RR2008). Lecture Notes in Computer Science, Springer (2008) 150–164
9. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ Envelope. In Kaelbling, L.P., Saffiotti, A., eds.: IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, Professional Book Center (2005) 364–369
10. Hitzler, P., Vrandecic, D.: Resolution-Based Approximate Reasoning for OWL DL. In Gil, Y., et al., eds.: Proceedings of the 4th International Semantic Web Conference, Galway, Ireland, November 2005. Lecture Notes in Computer Science, Springer, Berlin (2005) 383–397
11. Tserendorj, T., Rudolph, S., Krötzsch, M., Hitzler, P.: Approximate OWL-Reasoning with Screech. In Calvanese, D., Lausen, G., eds.: Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems (RR2008). Lecture Notes in Computer Science, Springer (2008) 165–180
12. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2007)
13. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM **26**(1) (1983) 64–69
14. Ramakrishnan, R., Gehrke, J.: Database Management Systems. Osborne/McGraw-Hill, Berkeley, CA, USA (2000)
15. Schmidt-Schauß;, M., Smolka, G.: Attributive Concept Descriptions with Complements. Journal of Artificial Intelligence **48**(1) (1991) 1–26
16. Krötzsch, M., Hitzler, P., Vrandecic, D., Sintek, M.: How to reason with OWL in a logic programming system. In Eiter, T., Franconi, E., Hodgson, R., Stephens, S., eds.: Proceedings of the Second International Conference on Rules and Rule Markup Languages for the Semantic Web, RuleML2006, Athens, Georgia, IEEE Computer Society (2006) 17–26
17. Bechhofer, S., Horrocks, I., Turi, D.: The OWL Instance Store: System Description. In: Proceedings of the 20th International Conference on Automated Deduction (CADE-20). Lecture Notes in Artificial Intelligence, Springer (2005) 177–181