NOTE TO USERS

Large ABox Store (LAS):

Database Support for TBox Queries

Jiaoyue Wang

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September 2005

ABSTRACT

Large ABox Store (LAS): Database Support for TBox Queries

Jiaoyue Wang

Large ABox Store (LAS) extends the DL reasoner Racer with a database. LAS stores the given information about a TBox, a taxonomy, and ABox in a database, and answers most TBox and ABox queries by combining SQL queries with DL reasoning. The main feature of our system is that it can deal with ABox role assertions. Acting as a filter for Racer, LAS speeds up the TBox and ABox queries. The main techniques exploited in LAS are the pseudo model merging test technique, and the transitive closure algorithm implemented by Oracle 9i.

This thesis presents the design, theories and implementation of the LAS system, and it mainly addresses the database support for TBox queries in LAS. A user-friendly interface is designed to facilitate users to implement many kinds of queries. More complex querying functions can be integrated into this system in the future.

Acknowledgements

I would like to express my deepest sense of gratitude to my supervisor, Dr. Volker Haarslev. Throughout my study period, he provided me with good teaching, sound advice, encouragement, and lots of help. I really appreciate his patience, support and care for me.

I would like to thank my colleague Cuiming Chen in the LAS project for her cooperation and help.

Also, I wish to thank all the colleagues in our lab for providing me a good and friendly working environment to do my research.

Finally, I am forever indebted to my parents for their understanding, support and endless love.

Table of Contents

List of Tables and Figures

1. Introduction

Knowledge representation is one of the central issues in computer science, in particular in Artificial Intelligence. It provides descriptions of the world that can be effectively used to build intelligent applications, so that the systems can derive implicit consequences from the explicitly represented knowledge.

Description Logics (DLs), as one of the important formalism for knowledge representation, unify and give a logical basis to the well-known traditions of frame-based systems, semantic networks, and KL-ONE-like languages, object-oriented representations, semantic data models, and type systems. In more recent years, Description Logics have become popular after more attention moved towards the properties of the underlying logical systems [9]. They are introduced in Chapter 2 in more detail.

Similar to Description Logics, databases are usually used to maintain models of some domain of discourse [2] as well. However, in contrast to Description Logics which express relatively complex information, databases provide simpler but more effective management of data. Specifically, they are a collection of data in machine-readable form, which can be manipulated by software to appear in varying arrangements and subsets. In other words, the main difference between Description Logics and databases is that while the former provide more supports for inference, which means finding the implicit consequences from the model, the latter mainly manipulate large and persistent models of relatively simple data.

As we can see, the relationship between Description Logics and databases is rather strong. Therefore, there is research in the field of building systems which involve both areas together. Doing so can make good use of their respective advantages: while Description Logics can be used not only to represent some indeterminate information, such as disjunctions, existential quantifications and number restrictions, but also to do reasoning, such as: classification, satisfiability, subsumption and instance checking; databases can be utilized to store a large number of concepts and individuals in order to realize persistency, scalability, secure and concurrent transaction management, and some reasoning and optimization can be done through processing SQL queries.

Concerning the research on combining Description Logics and databases, several investigations have been carried out. In 1993, Borgida and Brachman considered two possible ways to couple Description Logics Management System (DLMS) and databases Management System (DBMS), namely loose coupling and tight coupling, and chose the loose coupling approach to load database facts [1]. For checking inconsistency and performing reasoning, they need to periodically insert the objects from a DB to a DL reasoner. This approach may be well suited to already populated databases. However, it might be too late to abort the insertion transaction when inconsistent information was the input. Moreover, the DL language supported by it is much less expressive, and the database schemas must be customized according to the given TBoxes. In 1995, Bresciani adopted the tight coupling approach [19]. The basic idea of his approach is to extend the traditional DL ABox with a DBox which is a connection between a DL system and a

2

database, and then one can make queries to this extended system directly. Its advantage is that answers are given on the basis of the current state of the Description Logics and the database without considering updates, but the disadvantage is that it suffers from lack of automated translation between Description Logics and database schemes. Based on the above two investigations, Mathieu Roger developed a set-oriented model with three kinds of classes: abstract, concrete, and virtual classes [18]. This new approach realizes the classification process with a more general constraint language by using Racer.

Now we narrow the problem domain to an application related to Racer, which is a DL reasoning system [27]. Besides Roger's research, a new DL application for performing efficient and scalable DL reasoning over individuals named instance Store (iS) was built by Daniele Turi [23]. iS relies on Racer to perform classification, subsumption, checking consistency, and it queries databases by using SQL and the programming language Java. They showed that iS performs much better than Racer when the number of individuals is large [10]. Whereas, it has its limitations as well: it can only query role-free ABox, which means the ABox must not include role assertions. iS is introduced in more detail in Chapter 3.

To solve above problems and make the reasoning more optimized, we designed and implemented a sound and complete DL application: the Large ABox Store (LAS) system [6]. Similar to iS, our LAS system is mainly based on Racer. It realizes DL querying (reasoning) by using a relational database: Oracle. LAS communicates with Racer to get the ontology information first, and then stores the information into a database. Then the

system updates the tables. When a user queries LAS, the system will decide whether to query the database, Racer or both. Finally, LAS includes complete TBox and ABox information, effectively saves time to repeat large complex computations, and results in a significant decrease of query processing time compared to Racer and iS. The evaluation of test results is shown in [6]. LAS is sound because it always relies on Racer to retrieve and test the result, and the Racer system is always sound. LAS is complete because it can execute the queries through connecting to Racer and databases. In conclusion, to the best of our knowledge, LAS is the first system that provides reasoning, and places no a-priori restriction on the size or structure of the ABox and TBox.

Based on above advantages and features, the LAS system can be applied in many fields. Firstly, since it can be viewed as a system extending a DL reasoner, LAS can inherit the main features of Racer, which means it can be used in, e.g., semantic web, electronic business, medicine, natural language processing, knowledge-based vision, process engineering, knowledge engineering, and software engineering. Secondly, as a Java application for performing efficient and complete DL reasoning over large numbers of individuals, LAS is crucial in applications of ontologies in areas such as bioinformatics (gene description) and web service discovery because these applications might require vast volumes of individuals exceeding the capabilities of existing reasoners.

This thesis provides a thorough introduction of the TBox part of the LAS system, covering all the related aspects, namely background, theories, and implementation. Consequently, the thesis is divided into 6 chapters.

4

Chapter 1 narrows knowledge representation to Description Logics, analyzes the relationship between Description Logics and databases, and addresses some of most recent developments in combining Description Logics with databases.

Chapter 2 introduces the corresponding background knowledge, especially some important terms related to Description Logics, TBox, ABox, $ALCH_{R^+}$ language, Ontology, OWL file and Racer file.

Chapter 3 includes the problem statement and current state of the art. It emphasizes on introducing Racer and instance Store.

Chapter 4 introduces our design decisions including the Precompletion and Pseudo Model merging technique, application of Oracle database and the relevant Racer commands.

Chapter 5 focuses on the implementation of LAS. It is the core part of the thesis. It covers the system architecture, the specific operations and the GUI part.

Chapter 6 summarizes the current work we have done and mentions some future research.

LAS is a joint work with Cuiming Chen. We jointly designed LAS and worked on the connection with the reasoner. This thesis reports on the TBox query part and the graphical user interface. Details about the ABox query part can be found in [5].

2. Background

In this chapter, we introduce some concepts and definitions to set the stage for our work.

2. 1 Description Logics

The reader should have a basic understanding after the introduction of Description Logics in Chapter 1, but we still need to explain some core terms and give some definitions which will be frequently mentioned in this report.

As sketched in Chapter 1, Description Logics (DLs) is an important name for a family of knowledge representation formalisms that represent the knowledge of an application domain (the "world") by TBoxes and ABoxes. As the name indicates, one of the characteristics of these formalisms is that they are equipped with a formal, logic-based semantics. Another distinguished feature of them is the emphasis on reasoning as a central service. Figure 2.1 sketches the architecture of a DL system.
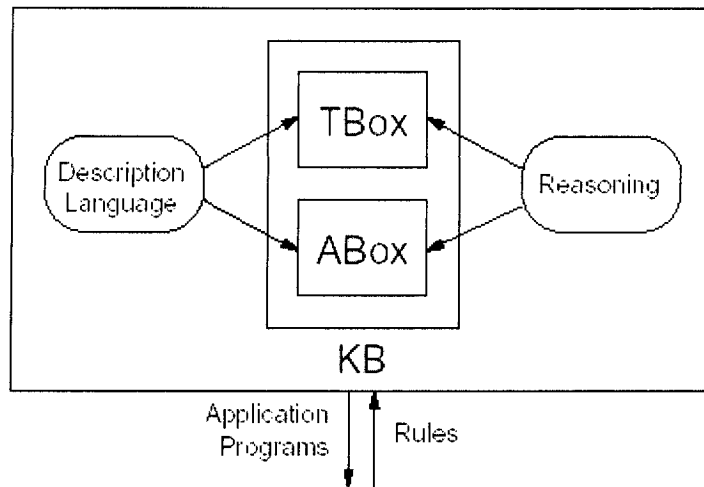
Figure 2.1. Architecture of a DL System [9]

In particular, a TBox is a collection of concept axioms, and composed of concepts which denote sets of individuals, and roles which denote binary relationships between individuals. An ABox is a collection of assertional axioms, and composed of concept and role assertions.

2.2 $ALCH_{R^+}$ language

Table 2.1 shows the semantics of DL constructor for the $ALCH_{R^+}$ language. Though the basic Description Language is ALC [22], the scope of our system is $ALCH_{R^+}$. Its definition is based on a standard Tarski-style semantics with an interpretation $I=(\Delta,\,^{I}\,)$ [28]. $ALCH_{R^+}$ extends ALC by adding role hierarchies and transitively closed roles.

7

Constructor	Syntax	Semantics	Example
concept	A	$A^I \subseteq \Delta^I$	woman
individual	i	$i^I \in \Delta^I$	charles
Top	$\top$	Δ^I	Top
Bottom	$\bot$	$\varnothing$	Bottom
negation	$\neg C$	$\Delta^I \setminus C^I$	$\neg$female
conjunction	$C \sqcap D$	$C^I \sqcap D^I$	person $\sqcap$female
disjunction	$C \sqcup D$	$C^I \sqcup D^I$	woman $\sqcup$man
value restriction	$\forall R.C$	$\{x \in \Delta^I \mid \forall y.(x,y) \in R^I \Rightarrow y \in C^I \}$	$\forall$has-child.parent
existential restriction	$\exists R.C$	$\{x \in \Delta^I \mid \exists y.(x,y) \in R^I \wedge y \in C^I \}$	$\exists$has-child.parent
role	R	$R^I \subseteq \Delta^I \times \Delta^I$	Has-child
role hierarchy	$R \subseteq S$	$R^I \subseteq S^I$	has_brother $\subseteq$has_sibling
transitive role	R_t	$\{ (x,z) \mid (x,y) \in R^I \wedge (y,z) \in R^I \}$	has_descendant

Table 2.1. Constructor Semantics and Examples

8

2.3 TBox

A collection of concept *axioms* is called a TBox (Terminological Box) [9]. It states how concepts and roles are related to each other.

Typical TBox axioms have the form: C $\subseteq$ D (R$\subseteq$ S) or C =D (R=S) where C and D are concepts (R and S are roles). Axioms of the first kind are called *inclusions* (or *subsumptions*), C is called *subsumee*, and D is called *subsumer*, while the second kind is called *equalities*.

An equality whose left-hand side is an atomic concept is called a *definition*. *Definitions* are defined as specific axioms because terminologies actually could be identified as sets of definitions. Definitions are used to introduce symbolic names for complex descriptions. For instance, the axiom father= man $\cap\exists$has-child.person is called a definition. The terms father, man, and person are called *atomic concepts*; the expression man $\cap\exists$has-child.person is called a *description* (or *complex concept*).

As mentioned before, Description Logics not only have logic-based semantics, but also offer powerful inference services. Since a DL knowledge base is divided into a TBox and an ABox, DL inference services can be divided into TBox Query Answering (or inference, reasoning) and ABox Query Answering.

Before introducing details of TBox Query Answering, we give an example that can help introduce some terms. Below is a Racer file format for the ontology Smith-family.

```
(in-knowledge-base family smith-family)
(signature :atomic-concepts (human person female male woman man
                                    parent mother father
                                    grandmother aunt uncle
                                    sister brother
                                    only-child)
           :roles ((has-descendant :transitive t)
                   (has-child :parent has-descendant
                              :domain parent
                              :range person)
                   (has-sibling :domain (or sister brother)
                                :range (or sister brother))
                   (has-sister :parent has-sibling
                               :range (some has-gender female))
                   (has-brother :parent has-sibling
                                :range (some has-gender male))
                   (has-gender :feature t))
           :individuals (alice betty charles doris eve))
(implies person (and human (some has-gender (or female male))))
(disjoint female male)
(implies woman (and person (some has-gender female)))
(implies man (and person (some has-gender male)))
(equivalent parent (and person (some has-child person)))
(equivalent mother (and woman parent))
(equivalent father (and man parent))
(equivalent grandmother
            (and mother
                 (some has-child
                       (some has-child person))))
(equivalent aunt (and woman (some has-sibling parent)))
(equivalent uncle (and man (some has-sibling parent)))
(equivalent brother (and man (some has-sibling person)))
(equivalent sister (and woman (some has-sibling person)))
(instance alice mother)
(related alice betty has-child)
(related alice charles has-child)
(instance betty mother)
(related betty doris has-child)
(related betty eve has-child)
(instance charles brother)
(related charles betty has-sibling)
```

Figure 2.2. Racer File Format for the Ontology Smith-family

Figure 2.3 and 2.4 show the concept hierarchy and role hierarchy for the family TBox,

and Figure 2.5 shows the depiction of the family ABox.
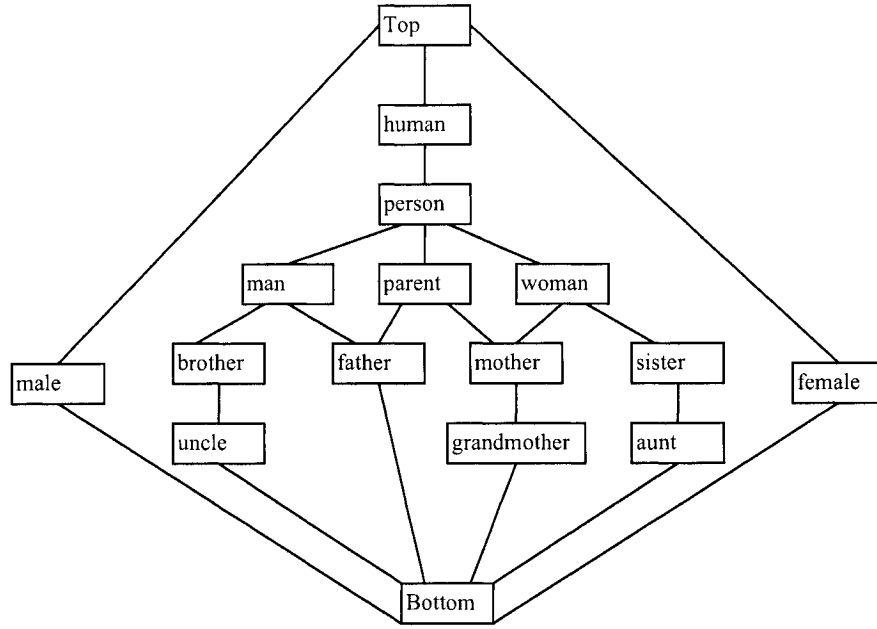
Figure 2.3. Concept Hierarchy for the Family TBox
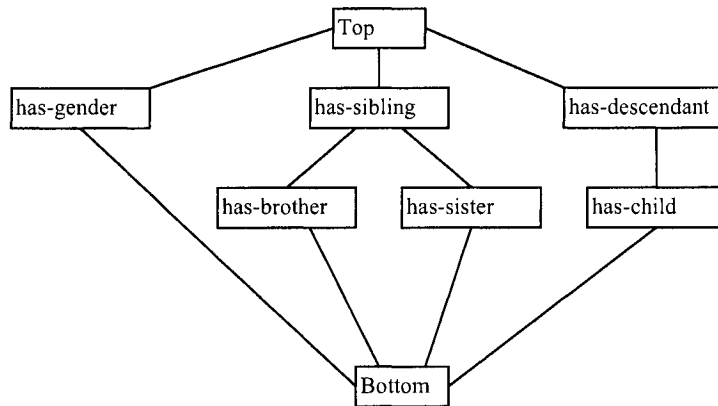
Figure 2.4. Role Hierarchy for the Family TBox
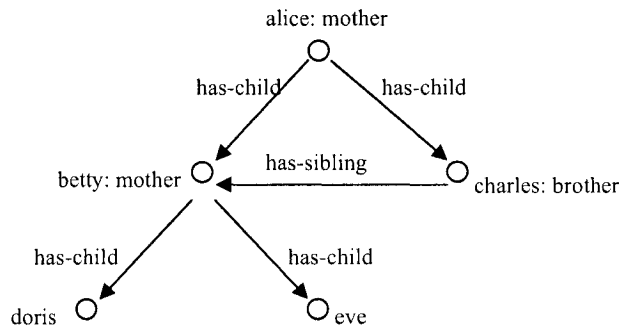
Figure 2.5. Depiction of the Family ABox

Considering Figures 2.3 and 2.4 for instance, TBox Query Answering includes:

Concept Satisfiability: Check whether the concept is non-contradictory, e.g., female $\sqcap$ male is not satisfiable.

Concept Parents: Get the direct atomic subsumers of the specified concept in the TBox, e.g., the Parent of aunt is sister.

Concept Ancestors: Get all atomic concepts of a TBox which are subsuming the specified concept, e.g., the Ancestors of aunt are sister, woman, person and human.

Concept Children: Get the direct atomic subsumees of the specified concept in the TBox, e.g., the Children of woman are sister and mother.

Concept Descendants: Get all atomic concepts of a TBox, which are subsumed by the specified concept, e.g., the Descendants of woman are sister, aunt, mother, and grandmother.

Concept Synonyms: Return equivalent concepts for the specified concept in the given TBox.

Role Parents: Get the roles from the TBox that directly subsume the given role in the role hierarchy, e.g., the Parent of has-brother is has-sibling.

Role Ancestors: Get all roles from the TBox, that subsume the given role in the role hierarchy, e.g., the Ancestors of has-brother is has-sibling.

Role Children: Get all roles from the TBox that are directly subsumed by the given role in the role hierarchy, e.g., the Children of has-sibling are has-brother and has-sister.

Role Descendants: Get all roles from the TBox, that the given role subsumes, e.g., the Descendant of has-descendant is has-child.

Role Synonyms: Get the synonyms of a role including the role itself.

Classification: the computation of the parents and children of every concept name is called classification of the TBox.

Actually, computing Concept Parents, Concept Ancestors, Concept Children, Concept Descendants, Role Parents, Role Ancestors, Role Children, and Role Descendants can be reduced to checking subsumption; computing Concept Synonyms and Role Synonyms can be reduced to checking equivalence. Meanwhile, we can see that both checking equivalence and subsumption can be reduced to checking concept satisfiability. Because $C=D \leftrightarrow C \sqsubseteq D \cap D \sqsubseteq C$; $D \sqsubseteq C \leftrightarrow (\neg C \cap D) = \bot$.

2.4 ABox

A collection of assertional axioms is called an *ABox (Assertional Box)*, which means knowledge about individuals is asserted in terms of concepts and roles from a TBox [9].

One can make the following two kinds of assertions in an ABox: i: C and (a,b): R , where i, a, b are individuals, C is a concept, and R is a role. The first kind is called a *concept assertion*, which means i belongs to C. The second one is called a *role assertion,* which means a and b are in the relationship R. Here, a is called the *predecessor*, b is called the *filler*, (a,b) is called the *related individuals* of the *filled role* R.

Considering Figure 2.5 for instance, ABox Query Answering includes:

Consistency: Check whether the set of assertions of an ABox is consistent with the TBox, that is, whether the ABox has a model.

Instance checking: Check whether the specified individual is an instance of the concept, e.g., check whether betty belongs to woman, and the answer is "T".

Retrieve: Get all individuals from an ABox that are instances of the specified concept, e.g., the individuals of woman are betty and alice.

Individual Types (realization): Get all atomic concepts of which the individual is an instance, e.g., Individual Types of betty are mother, woman, parent, person, and human.

Individual Direct Types: Get the most-specific atomic concepts of which an individual is an instance, e.g., Individual Direct Type of betty is mother.

Individual Direct Predecessors: Get all individuals that are predecessors of a role for a specified individual, e.g., alice is the predecessor of has-child for betty.

Individual Fillers: Get all individuals that are fillers of a role for a specified individual, e.g., betty and charles are the fillers of has-child for alice.

Individual Filled Roles: This function gets all roles that hold between the specified pair of individuals, e.g., has-child, has descendants are the filled roles that hold between alice and betty.

Related Individuals: Get all pairs of individuals that are related via the specified relation, e.g., (alice, betty), (alice, charles), (betty, doris) and (betty, eve) are related via has-child.

Here, Consistency, Instance checking, Retrieve, Individual Types, Individual Direct Types are called *concept assertion queries*. Similar to TBox inference services, all these services can be reduced to check ABox consistency. Individual Direct Predecessors, Individual Fillers, Individual Filled Roles, Related Individuals are called *role assertion queries*.

2.5 Ontology

Except DL inference services, the TBox and ABox are defined together by the term: *Ontology*. This term was borrowed from philosophy, where an ontology is a systematic account of Existence. For AI systems, "An ontology is an explicit specification of a conceptualization" [26]. A definition will be "it is a document or file that formally defines the relations among terms". The most typical kind of an ontology for the web has a taxonomy and a set of inference rules.

There are wide varieties of languages for describing ontologies, such as OIL, DAML, DAML+OIL[7], OWL[17], etc. In the LAS system, we only considered and implemented the methods for parsing OWL files and Racer files, so only OWL and Racer files are briefly introduced below.

2.6 Racer File

A Racer file is a kind of ontology description file based on the Racer system. An example of the Racer file format for Ontology "Smith-Family" is shown in Figure 2.2.

2.7 OWL DL file:

The Web Ontology Language (OWL) recently became a W3C recommendation [8]. A typical OWL ontology always begins with a namespace declaration, and it is always in the format of URLs, such as http://www.w3.org/2002/07/owl#. A concept, a role or an individual name is always composed of the namespace and the value that follows the

symbol #. Even though the real meaning is the value, the value has to be expressed with

the name space together. For instance, in the following ontology, if one wants to query

Disney_mouse, the input format must be http://www.w3.org/2002/07/#Disney_mouse.

Below is an example for a "Cartoon Star" OWL file ontology.

```
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF
      xmlns="http://a.com/ontology#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs=http://www.w3.org/2000/01/rdf-schema#
      xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://a.com/ontology">
  <owl:Ontology rdf:about="" />
  <owl:Class rdf:ID="Disney_cat">
      <rdfs:subClassOf>
       <owl:Class rdf:about="#Cartoon_cat" />
      </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Cartoon_mouse">
      <rdfs:subClassOf>
          <owl:Restriction>
             <owl:onProperty>
                 <owl:SymmetricProperty rdf:about="#Is_cousin_of" />
             </owl:onProperty>
             <owl:allValuesFrom>
                 <owl:Class rdf:about="#Disney_mouse" />
             </owl:allValuesFrom>
          </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
          <owl:Class rdf:about="#Cartoon_star" />
      </rdfs:subClassOf>
      <owl:disjointWith>
          <owl:Class rdf:about="#Cartoon_cat" />
      </owl:disjointWith>
      <owl:disjointWith>
          <owl:Class rdf:about="#Cartoon_dog" />
      </owl:disjointWith>
  </owl:Class>
</rdf:RDF>
```

Figure 2.6. OWL File Format for Ontology "Cartoon_star"

3. Problem Statement and Current State of the Art

As mentioned in Chapter 1, the idea of combining Description Logics with databases is not new. But we will narrow our subject to a more particular problem related to the Racer system, namely that extending Racer with a relational database.

3.1 Racer

A brief introduction of the Racer system [29] is as follows. It is a first full-fledged DL system, which is capable of reasoning ontologies by implementing a highly optimized tableau calculus. *SHIQ* is a logic supported by Racer. It not only includes basic *ALC* features, but also role hierarchies and transitive roles as in $ALCH_R$. Moreover, Racer provides support for qualified number restrictions and inverse roles. This is the reason why we say Racer is full-fledged. Besides, Racer offers reasoning services for multiple TBoxes and ABoxes, and provides facilities for algebraic reasoning including concrete domains.

By processing commands, Racer can manage ontologies directly, through loading, deleting, mirroring, etc. An important function that Racer offers is its TBox and ABox query answering functions which reflect inference services.

Checking ABox consistency is the key for ABox inference services as mentioned in Chapter 2. Racer implements a consistency algorithm based on tableau methods, which

consist of a set of completion rules operating on constraint sets and tableau clash triggers [30]. In particular, a procedure for checking satisfiability of a concept transforms all concepts into negated normal form first, for instance, $\neg (C \sqcap D) \rightarrow \neg C \sqcup \neg D$, $\neg \exists R.C \rightarrow \forall R. \neg C$. Then it applies completion rules in an arbitrary order as long as possible. But applications of rules should terminate in case of a clash, or terminate if no completion rule is applicable anymore. At last, the result is satisfiable if and only if a clash-free tableau can be derived. This final result is called a completed ABox. Below are completion rules for ALC.

Clash trigger
$\{a:C, a:\neg C\} \subseteq \mathcal{A}$

Conjunction rule
if 1. $a:C \sqcap D \in \mathcal{A}$, and
2. $\{a:C, a:D\} \not\subseteq \mathcal{A}$
then $\mathcal{A}' = \mathcal{A} \cup \{a:C, a:D\}$

Disjunction rule
if 1. $a:C \sqcup D \in \mathcal{A}$, and
2. $\{a:C, a:D\} \cap \mathcal{A} = \varnothing$
then $\mathcal{A}' = \mathcal{A} \cup \{a:C\}$ or
$\mathcal{A}' = \mathcal{A} \cup \{a:D\}$

Role exists restriction rule
if 1. $a:\exists R.C \in \mathcal{A}$, and
2. $\neg \exists b \in O: \{(a,b):R, b:C\} \subseteq \mathcal{A}$
then $\mathcal{A}' = \mathcal{A} \cup \{(a,b):R, b:C\}$
with b fresh in $\mathcal{A}$

Role value restriction rule
if 1. $a:\forall R.C \in \mathcal{A}$, and
2. $\exists b \in O: (a,b):R \in \mathcal{A}$, and
3. $\{b:C\} \not\subseteq \mathcal{A}$
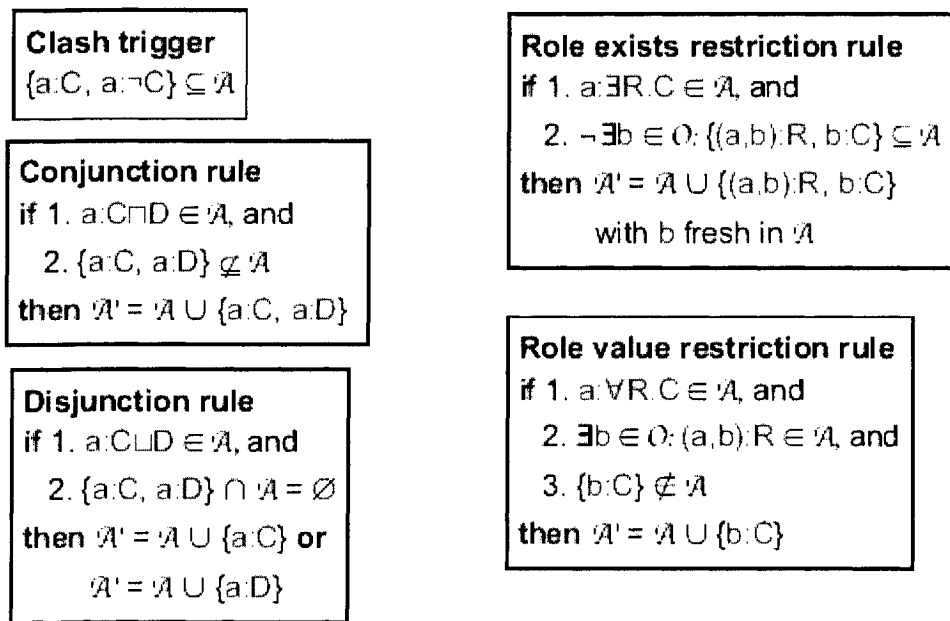then $\mathcal{A}' = \mathcal{A} \cup \{b:C\}$

Figure 3.1. Completion Rules for the Logic ALC [30]

Meanwhile, some optimized search techniques are employed in order to improve Racer's average-case performance. For checking satisfiabilty, approaches such as dependency-

directed backtracking [21], DPLL-style semantic branching [15], model caching [31], model merging [13], and processing qualified number restrictions with the simplex procedure were used. For TBox reasoning, approaches such as transformations of general axioms [14], classification order/ clustering of nodes, fast test for non-subsumption, marking, propagation, and lazy unfolding [11] were applied; For ABox reasoning, approaches such as graph transformation, fast test for non-subsumption, data-flow techniques for realization and dependency-driven divided-and-conquer [27] for instance checks were employed to improve Racer.

Though these optimization techniques effectively improved the speed of Racer, Racer is still not fast enough under some circumstances, especially when applications might require a large number of individuals which could exceed the capabilities of Racer. The main reason for the above problems is that Racer stores all the concepts, roles and individuals into the main memory, and the query answering speed depends on the tableau algorithm. Furthermore, while one can assume that changes in a TBox are relatively infrequent, changes of an ABox maybe more dynamic, frequent and possibly concurrent. To solve this problem, Sean Bechhofer, Ian Horrocks and Daniele Turi at University of Manchester developed a DL system called instance Store which combines TBox reasoning and a database to perform a role-free ABox reasoning [24].

3.2 instance Store (iS)

As a starting point, iS requires that no role assertion (no binary role between individuals) exists, i.e., the ABox is role-free. That means every individual is independent from others, so reasoning about individuals can be simply reduced to reasoning about the concepts they belong to. From an architectural point of view, the ontology of classes can then be treated as a static schema, and loaded into the TBox reasoner Racer, while the concept assertions are dynamically added to and retrieved from a database. This way, they can exploit databases offering persistency, scalability, and secure and concurrent transactions.

Relying on Racer, iS loads the ontology, meanwhile, it connects to a database and creates the tables. The second step is storing concept assertions (an individual is an instance of a concept) into the tables, together with additional information gathered through calls to Racer, such as equivalent atomic concepts, description's parents, children, ancestors, and descendants. Based on these requirements, the following DB schema was created [10]:

Description (id, description)

Assertions (individual, id)

Types (id, atomicConcept)

Equivalents (id, atomicConcept)

Parents(id, atomicConcept)

Children (id, atomicConcept)

A value for id is issued by iS, and the Types table stores all the ancestors of the related description. From the tables, we can see iS has to rely on Racer to classify the taxonomy whenever one asserts a concept assertion so that iS can store the information to the tables Types, Equivalents, Parents, and Children.

The above two steps can be viewed as the management of the ontology. About reasoning, iS only offers two ABox query answering: Retrieve and Individual types. As for retrieval, the first step is to check whether the query description is consistent. Then whether the description is stored in the DB is checked. If yes, then there is no need to classify this description. If no, then iS has to classify this description by using Racer, and store the corresponding information into the tables. After classification, if the description has an equivalent atomic concept, then iS simply returns the individuals of this atomic concept. Otherwise, it will invoke more complex operations: It has to check whether the conjunction of the parents of the query description is just equal to this description. If yes, it just returns the intersection of the individuals of its each parent, and union of the individuals of all its children. If no, it has to get the candidates which are the descriptions corresponding to individuals of parents which are not also individuals of children first, then, asks Racer to compute the subset of the candidates which are subsumed by the query description. At last, it returns the individuals of this subset [24].

From the above description, one can see iS relies on Racer not only for the ontology management, like classification, but also on the reasoning, like checking consistency, classification, and checking subsumption. The main point of iS is to use Racer but only

when necessary. Meanwhile, iS takes advantage of the features of databases, because databases are well suited to handling large amounts of data and optimized for operations such as joins and intersections.

However, though iS can be used in some applications, including web services discovery and genes classification, it has some limitations. The most severe and obvious restriction is that compared with a fully fledged DL ABox, iS can only deal with a role-free ABox. This means it can not be applied to ontologies that contain any axiom asserting role relationships between pairs of individuals. The reason why it can not contain role assertions is because, to a concept assertion a: C, where a is an individual and C is a concept, if C is consistent, the individual a can only cause a contradiction through a role assertion. So the iS system needs only to check TBox consistency by using a TBox reasoner, and check ABox consistency through checking TBox consistency, which means avoid using an ABox reasoner. Secondly, iS completely relies on Racer to classify the whole taxonomy, including the parents, children, ancestors and descendants. But the fact is databases can solve the problem of transitive closures directly, that means the system could only ask Racer to get the parent-child pairs, and then rely on databases to compute the ancestor-descendant pairs. This process can reduce Racer's computing time. Finally, iS stores all the description assertions into the table Assertions. Of course when the queries for retrieving these descriptions are processed, it does not need to classify the query descriptions again. But for example, as we know that $C \cap D = D \cap C$, in the iS system, both the descriptions need to be stored and classified, and one of them is

redundant. So, in some cases, it does not make sense to store complex description assertions.

Compared with iS, our system LAS is a fully-fledged DL system. It can store essential and sufficient ontology information into a database, and executes sound and efficient algorithms such as the merging test to query the ontology. It makes use of the Oracle database to compute the transitive closure and exploits some optimizations.

4. Design Decisions and Reasons

First, for eliminating the limitation of Racer that it currently does not efficiently deal with large ABoxes, we should find the reason. Because the ontology information must be stored and all the query algorithms must run in the main memory, it is obvious that the efficiency will be affected. So exploiting a database to store ontology will be a very suitable solution. Storing the necessary information of a TBox and ABox, and then querying the database by executing SQL, that's the basic idea of our system. As mentioned, classification is the core inference service, so LAS must rely on Racer to get it. However, since our task is to reduce complex computation of Racer as much as possible, LAS will only rely on Racer to retrieve basic classification which means only parent-child pairs, while it obtains all the ancestor-descendant pairs by exploiting a transitive closure algorithm which the Oracle DB offers. Thirdly, and most importantly, merging techniques for so-called pseudo models were applied in our system. Actually, "optimizing TBox and ABox reasoning with pseudo models" is known as an optimization technique and data structure for Racer [20]. This technique was applied to check consistency of the TBox and ABox, once Racer runs, and it was known to speed-up TBox and ABox reasoning for the description $ALCNH$. Based on the above theory, LAS just stores the pseudo models into the DB without recomputation, and exploits the merging test algorithm to check non-subsumption between descriptions and concepts, in order to answer TBox/ABox queries.

4.1 Precompletion

At the beginning, we tried to design a complete system extending iS by adding role assertions. This could be achieved by applying some form of precompletion to the ABox [4, 25].

The main idea behind a *Precompletion* is to eliminate ABox axioms specifying relationships between individuals by explicating the consequences of such relationships. Once these axioms have been eliminated, the assertions about a single individual can be independently verified using a standard TBox reasoner. For example, if (a, b): R, and a: $\forall$R.B, we can conclude that b: B. In this way, role assertions can be transformed to concept assertions. Then checking individual a's satisfiability can be transformed to checking description $\exists$R.B 's satisfiability by using TBox reasoning.

Below are the Precompletion rules [25] which can eliminate role assertions and make individuals become independent of one another, where A means an ABox, o, o' are individuals, C, C_1, C_2, D are concepts, and R, R', S, T are roles.

$$A \longrightarrow \{o{:}C\} \cup A$$
if o is in O, C is in T
and $o{:}C$ is not in A.

$$A \longrightarrow \{o{:}D_i\} \cup A$$
if $o{:}C_1 \sqcap C_2$ is in A,
and $D = C_1$ or $D = C_2$
and neither $o{:}C_1$ nor $o{:}C_2$ is in A.

$$A \longrightarrow \{o'{:}C\} \cup A$$
if $o{:}\forall R.C$ and $o,o'{:}S$ are in A,
$R \overset{*}{\sqsubseteq} S$ and $o'{:}C$ is not in A.

$$A \longrightarrow \{o'{:}\forall R.C\} \cup A$$
if $o{:}\forall T.C$ in A, $o,o'{:}S$ is in A,
and there is R, $T \in RN$ such that $S \sqsubseteq R \sqsubseteq T$,
and $o'{:}\forall R.C$ is not in A.

$$A \longrightarrow \{o{:}C_1, o{:}C_2\} \cup A$$
if $o{:}C_1 \sqcup C_2$ is in A,
and neither $o{:}C_1$ nor $o{:}C_2$ is in A.

$$A \longrightarrow \{o'{:}C\} \cup A$$
if $o{:}\forall R.C$ and $o,o'{:}S$ are in A,
there is $R' = R^{-}$, $R' \overset{*}{\sqsubseteq} S$
and $o'{:}C$ is not in A.

$$A \longrightarrow \{o'{:}C\} \cup A$$
if $o{:}\forall R.C$ is in A, and $o,o'{:}S$ is in A,
and $S \sqsubseteq R$, and $o'{:}C$ is not in A.

Figure 4.1. the Precompletion Rules for *SHF* [25]

As proven in [25] for the DL *SHF*, the precompletion algorithm will always terminate no matter which of the applicable rules is chosen first. Although different strategies for the priority of rules to be chosen can lead to different computing complexities, as long as disjunctions of concepts exist, the worst case of the computing complexities will be exponential. For instance, to a: $(C1 \cup D1) \cap (C2 \cup D2) \cap ... (Cn \cup Dn)$, we have to check the satisfiability of the description $(C1 \cup D1) \cap (C2 \cup D2) \cap ... (Cn \cup Dn)$. That means there are 2^n possibilities. Whenever there is a clash, we have to try another possibility. So the best case will be 1, and the worst case will be 2^n times.

4.2 Pseudo Model Merging

Another solution to reasoning with ABoxes containing role assertions is called pseudo model merging technique [3]. For the initial TBox satisfiability test, when Racer checks a description, Racer only computes one completion and caches it as a pseudo model. Finally, all descriptions can be represented as the models only composed of conjunctions. For example, concerning description $(A \cup B \cap C)$, Racer will cache its model as $(A \cap C)$ or $(B \cap C)$. Such kinds of items that compose of a set of concepts representing a conjunction are called concept pseudo models [3]. The main goal of this strategy is to avoid a consistency test which relies on the "expensive" tableau technique. This idea was first introduced in [13] for the logic $ALCH_{R^+}$. A model merging test is designed to be a " cheap" test operating on cached concept pseudo models. It is a sound but incomplete non-subsumption test for a pair of concepts. The achievement of minimal computational overhead and the avoidance of any non-determinism are important characteristics of such a test.

A pseudo model M for a concept term C is defined as the tuple <MA, M¬A, M∃, M∀> of concept sets using the following definitions [3] where A' is a completed ABox as mentioned in Section 3.1, a is an individual, A and D are concepts, and R is a role:

MA = {A| a:A ∈ A'}

M¬A = {A| a: ¬A ∈ A'}

M∃ = {R | a: ∃R. D ∈ A'}

M∀ = { R | a: ∀R. D ∈ A'}

A simple example will be helpful to describe the definition of a concept pseudo model: concept Con=(A∪B)∩¬C∩∃R. D ∩∀S. E, so Racer will cache Mcon <{A},{C},{R},{S}> or Mcon <{B},{C},{R},{S}> .

Why and how can we apply the concept pseudo model technique in our system? In TBox reasoning, checking subsumption is one of the most important issues, such as querying concept ancestors and descendants. LAS relies on Racer to get the atomic concept taxonomy, and store it into the database, so looking up the relevant tables will help the system to retrieve the results directly. But concerning descriptions (complex concepts), it does not make sense to classify them into the taxonomy and store them into the database whenever a new description is encountered because the possibility for reusing the description is very low. Under these circumstances, using concept pseudo model merging techniques to check non-subsumption is a good solution. In fact, querying description ancestors means to find all the atomic concepts that subsume this specified description, and querying description descendants means to find all the atomic concepts that are

subsumed by this specified description. A subsumption test: whether C⊆D where C and D are descriptions (or atomic concepts), can be transformed to a satisfiability test: whether C∩ ¬D is not satisfiable. Then we begin to apply the pseudo model merging test. The procedure is that after computing the pseudo models of C and ¬D, namely M1 and M2, we compare their four sets of the model tuples separately. If M1A has a non-empty intersection with M2¬A , or M1¬A with M2A, or M1∃ with M2∀, or M1∀ with M2∃, we say the two models are interacting.

Now we have to analyze their interaction results: If they don't have any interaction after we check all these four sets, then these two pseudo models are mergable, i.e. their corresponding concepts are conjunctively combined, and C∩ ¬D is satisfiable, so we can draw the conclusion that C⊄D. For example, checking whether concept Con= (A∪B)∩¬C∩∃R.D∩∀S.E ⊆ atomic concept F, we transform it to checking the satisfiabilty of Con ∩ ¬F. By comparing the pseudo model of Con: <{A},{C},{R},{S}> and the pseudo model of ¬F: <{∅},{F },{∅},{∅}>, we can see there is no any interaction among the corresponding sets. So we can conclude, these two models are mergable and Con ⊄ F.

Conversely, if the two pseudo models have interactions, can we conclude that these two models are not mergable and C⊆D? The answer is no, because it is a sound but incomplete test for checking subsumption and satisifiablity as mentioned before. There are two reasons causing the incompleteness. One reason comes from the Racer's way of caching. As mentioned, Racer only caches one of the completions, so for instance,

checking whether A⊑A∩B, we compare pseudo model of A, M1: <{A },{∅},{∅},{∅}> and pseudo model of ¬(A∩B)= ¬A∪¬B, M2: could be <{∅},{ A },{∅},{∅}> or <{∅},{B },{∅},{∅}>. If Racer cache M2 as <{∅},{A},{∅},{∅}>, we can see there is an interaction between M1A and M2 ¬A; however, if Racer cache M2 as <{∅},{B},{∅},{∅}>, there is no interaction among M1 and M2. So we conclude these two models are mergable and A ⋢ A∩B. The other reason comes from the incompletion of the merging algorithm. In our algorithm, if M1∃ has the same factors as M2∀ has, or M1∀ has the same factors as M2∃ has, we say two models have an interaction, and they are not mergable. But this is due to the flat model merging technique. For example, if we check whether ∀R.C ⊑∀R.D, we compare pseudo model of ∀R.C, M1: <{∅},{∅},{∅},{R}> and pseudo model of ¬(∀R.D) = ∃R.¬D, M2: <{∅},{∅},{R},{∅}>. We can see there is an interaction between M1∀ and M2 ∃, but we can not say M1 and M2 are not mergable. Even though they have the same set R, we have to see the successors of R, which are C and ¬D. Actually C and ¬D are mergable, so ∀R.C and ∃¬D are mergable and ∀R.C not ⊑∀R.D. That comparing the successors is called deep model merging. However, the deep merging test is only correct for trees not for graphs, so it is only applicable for concept pseudo models, not individual pseudo models. Since we considered the efficiency of the system first; moreover, it is inconvenient to store the successors in the DB, we did not apply the deep merging test to LAS.

Though checking subsumption by applying the merging (flat model merging) test is incomplete, we can exploit it to implement part of the checking, which means since non-

subsumption checking by applying the merging test is sound, we can get the subsumption candidates by subtracting non-subsumption concepts from all named concepts in the given TBox. After that, we can call Racer to test the limited candidates and finally return the complete results. Note that the merging test is just implemented by a collection of SQL queries. By using this approach, it keeps the final results complete.

Because an individual can always be represented as the format of a: C, where C is a description, individuals can be described by pseudo models as well. These kinds of models are called individual pseudo models. Racer creates individual pseudo models from the initial ABox satisfiability test and exploits them for various ABox reasoning tasks. Similar to checking concept subsumption, checking whether a:C holds can be transformed into checking whether $A \cup \{a:\neg C\}$ is unsatisfiable where A is an ABox, and consequently, it is preceded by a check whether the pseudo model of a and the one of $\neg C$ are mergable. So the individual pseudo model technique can be used in retrieving individuals of the given description by removing concepts that do not instantiate the individual a.

In conclusion, both the precompletion and pseudo model merging technique can relax the limitations of the iS system. But the precompletion approach may involve an exponential number of computations which might be very expensive, and affect Racer's optimization algorithms. On the contrary, the pseudo model merging technique can be applied not only in ABox queries, such as retrieving individuals, but also in TBox queries, like querying

description ancestors, descendants, and synonyms. So in our system, we exploited the pseudo model merging technique instead of the precompletion technique.

4.3 Databases

Though there are many databases that can be exploited, we chose Oracle 9i for its performance, reliability and security. Moreover, we could use Oracle9i Database tools: CONNECT BY PRIOR operator and SYS_CONNECT_BY_PATH function for the computation of the transitive closure [16]. Transitive closure means to compute all descendants of a node in a directed graph. The detailed algorithm will be introduced in Chapter 5. However, the limitation of applying transitive closure by Oracle is that the top or bottom of the hierarchy must be known at the beginning. Fortunately, LAS can get all concept parent-child pairs, role parent-child pairs, top and bottom through Racer, so it can compute all ancestors-children pairs with the Oracle DB. On the other hand, LAS uses the transitive closure to implement role assertion queries as well. The detailed description of the implementation is discussed in [5].

The design of the database schema must support implementation of various TBox and ABox reasoning services. TBox reasoning includes Concept Parents, Concept Ancestors, Concept Children, Concept Descendants, Concept Synonyms, Role Parents, Role Ancestors, Role Children, Role Descendants, and Role Synonyms. Concepts have to be considered in two situations: If the concepts are atomic concepts, because their possibility for reuse is high, LAS stores their taxonomy into a database during the initialization.

From Racer LAS gets the parent-children pairs and synonym pairs which are then stored in tables *DesParents* and *DesSynonym*. Afterwards the table *DesAncesors* is computed as the transitive closure. It is the same situation for roles, so LAS needs tables *Rparents*, *RSynonyms* and *RAncestors*. If the concepts are complex descriptions, LAS does not store them in the database. Only when a query is posed, LAS parses the query description's pseudo model obtained from Racer. So LAS needs the table *TempDesPM* to store this temporary model. When answering the query Ancestors, Descendants or Synonyms, LAS gets the candidates by exploiting the merging test on the temporary model and all the atomic concept pseudo models or negated concept pseudo models. So LAS needs tables *Description* and *DescriptionPseudoModel*. However, for the query Parents or Children, even though LAS can execute the merging test to get the subsumption candidates, Racer does not offer a command such as *"(concept-parents +ConceptName+ +CandidatesConceptName+)"* or *"(concept-children +ConceptName+ +CandidatesConceptName+)"* to test only the specified candidates. Therefore LAS has to rely on the Racer commands *(concept-parents +ConceptName+)* or *(concept-children +ConceptName)* to get the query result directly without dealing with databases at all.

ABox reasoning includes Retrieve, Individual Types, Individual Direct Types, Individual Direct Predecessors, Individual Fillers, Individual Filled Roles, and Related Individuals. For the query Retrieve, LAS needs the table *InAssertion*. If the query concept is an atomic concept, LAS can get its negated pseudo model from the table *Description* and *DescriptionPseudoModel* directly; otherwise, LAS stores its temporary negated pseudo model obtained from Racer in the table *TempDesPM*. LAS gets the candidates by

exploiting the merging test on the above mentioned negated model and all the individual pseudo models. So it needs the tables *Individual* and *IndividualPseudoModel*. For the query Individual Types and Individual Direct Types, LAS relies on Racer to return the final results directly without a need to query the DB. For the other role assertions queries, LAS needs the table *RoleAssertion* to store role assertions obtained from Racer, and computes the final results by exploiting the tables *RoleType*, *RoleTransitive* and *RAncestors*.

4.4 Racer Commands

LAS can be considered as an extension of Racer, so most functions are implemented by relying on Racer. The Racer commands [16] that are used by LAS are listed in Appendix A.

5. System Implementation

This chapter introduces the implementation of our system LAS in detail. It includes the introduction of the system architecture which gives readers an intuitive impression of LAS, the system operation which is described by the implemented components, and the GUI (Graphical User Interface).

5.1 System Architecture

The LAS system consists of:

- an ontology

  ◆ such as an OWL file or RACER file.

- a reasoner

  ◆ Racer (accessed through JRacer).

- a database

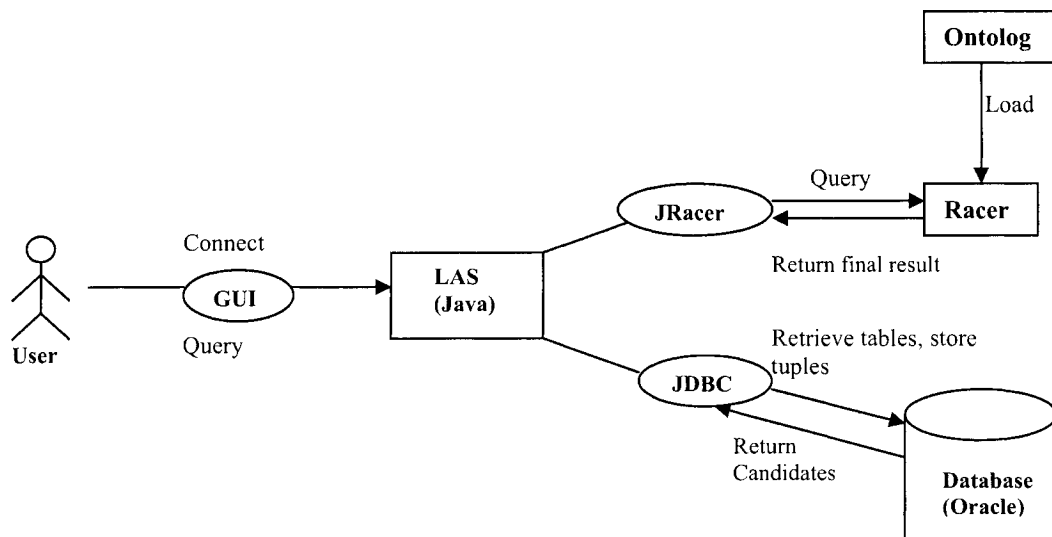 ◆ Oracle 9i (accessed through JDBC).

Figure 5.1. LAS Architecture

A user can access LAS through a GUI, which includes two main operations: Connect and Query. The Connect operation connects to Racer at first, and then reads the ontology file into the system. Finally, the user decides to initialize (create and store tables) or load the Oracle DB. The Query operation is the core part of LAS. By entering concept, role or individual names in the GUI, the user can query the Tbox and Abox.

5.2 Operation

5.2.1 Connect

Before a user starts to write a query, the system needs to connect to Racer and a database, load the ontology, and store the necessary information into the tables. This preparation process is called Connect operation.

5.2.1.1 Initialization

If it is the first time for a user to load a new ontology, the system will execute the initialization operation: firstly, LAS assigns a new DB name and password, and then creates 12 empty tables in the DB. After that, our system will load the OWL or Racer file into Racer, and then query Racer to get all the atomic concepts, individuals and their pseudo models, roles and transitive roles, concept parent-child and role parent-child pairs, concept synonyms and role synonyms, individual assertions and role assertions. After parsing Racer's output, LAS stores this data into the corresponding tables. Meanwhile, it generates two extended tables: DesAncestors and RAncestors by computing the transitive closure of the tables DesParents and RParents respectively. Therefore, in the initialization

35

step, LAS' task is to store the basic required information about the TBox, the ontology's taxonomy, and the ABox.

Connect to Oracle DB

As we know, JDBC is a set of classes and interfaces written in Java to allow other Java programs to send SQL statements to a relational database management system. We applied one of the categories of JDBC drivers which are provided by Oracle: JDBC Thin Driver [34] to connect our system to Oracle DB because this driver uses Java sockets to connect directly to Oracle and provides its own TCP/IP version of Oracle's Net8 (SQL*Net) protocol. This driver developed with pure Java is platform independent and can also run from a Web Browser (applets).

The command for connecting LAS to Oracle is shown below.

ConnectDB( ):

Connection c = DriverManager.getConnection(jdbc:oracle:thin:@

machineName:port:SID, userid, password);

Assign DB Name and Password

If a user wants to create a new database in Oracle for a new ontology, he must assign a DB name and password for this ontology first.

Below is the relevant algorithm:

Algorithm CreateDB( )

CREATE USER \""+DBname+"\" PROFILE \"DEFAULT\"   //assign DB name

IDENTIFIED BY \""+password+"\" DEFAULT TABLESPACE \"USERS\"

//assign password

ACCOUNT UNLOCK;   // assign users account unlocked.

GRANT UNLIMITED TABLESPACE TO \""+DBname+"\"   // assign sort space

After creating a new DB, required privileges are assigned:

GRANT \"CONNECT\" TO \""+DBname+"\"

GRANT \"DBA\" TO \""+DBname+"\"   //make a user a DB Administrator

Create Tables

For system requirements, LAS needs to create the main tables listed as following:

1. Description (DescriptionName, DesPseudoModelID, NegationDesPseudoModelID, DesComplete): This table is used to store all atomic concepts in the ontology. DescriptionName: Store the atomic concept names.

   DesPseudoModelID: Store the atomic concepts' pseudo model IDs, which are assigned by the system.

   NegationDesPseudoModelID: Store the pseudo model IDs of the negated atomic concepts.

   DesComplete: Indicate whether all the individuals belonging to the atomic concepts are stored in the table.

2. DescriptionPseudoModel (DesPseudoModelID, DesA, DesNotA, DesExistence, DesUniversal, Unique): This table is used to store the atomic concepts' pseudo models and the pseudo models of the negated atomic concepts.

DesPseudoModelID: Store the atomic concepts' pseudo model IDs, which are assigned by the system.

DesA: Store MA sets.

DesNotA: Store M¬A sets.

DesExistence: Store M∃ sets.

DesUniversal: Store M∀ sets.

Unique: Indicate whether this is the only possible pseudo model.

3. Individual (IndividualName, InPseudoModelID, IndComplete): This table is used to store all the individuals in the ontology.

IndividualName: Store the individual name.

InPseudoModelID: Store this individual pseudo model ID, which is assigned by the system.

IndComplete: Indicate whether all the atomic concepts that this individual belongs to are stored in the table.

4. IndividualPseudoModel (InPseudoModelID, InA, InNotA, InExistence, InUniversal, Unique): This table is used to store the individual pseudo models.

InPseudoModelID: Store the individual pseudo model ID.

InA: Store MA sets.

InNotA: Store M¬A sets.

InExistence: Store M∃ sets.

InUniversal: Store M∀ sets.

Unique: Indicate whether this is its only pseudo model.

5. RoleType (RName): This table is used to store all the role names in the ontology.

   RName: Store the role name.

6. RoleTransitive (RName): This table is used to store the transitive role names.

   RName: Store all the transitive role name.

7. DesParents (DesParents, DesChildren): This table is used to store the atomic concept parent-child pairs.

   DesParents: Store the atomic concept names.

   DesChildren: Store the children concept names.

8. RParents (RParents, RChildren): This table is used to store the roles parent-child pairs.

   RParent: Store the role names.

   RChildren: Store the children role names.

9. DesSynonyms (Description1, Description2): store the synonym concepts pairs.

   Descripiton1: Store the atomic concept names.

Descripiton2: Store the synonym concept names.

10. Rsynonyms (RName1, RName2): This table is used to store the synonym roles pairs.

RName1: Store the role names.

RName2: Store the synonym role names.

11. InAssertion (IndividualName, DescriptionName, MostSpecific): This table is used to store concept assertions in the ontology.

IndividualName: Store the individual names.

DescriptionName: Store the atomic concept name which the individual belongs to.

MostSpecific: Note whether this concept is the most specific concept which the individual belongs to.

12. RoleAssertion (Individual1Name, Individual2Name, RoleName, Complete1, Complete2, RoleComplete): This table is used to store role assertions in the ontology.

Individual1Name: Store the predecessors.

Individual2Name: Store the fillers.

RoleName: Store the role names.

Complete1: Note whether all the predecessors of individual2 and role are stored in the table.

Complete2: Note whether all the fillers of individual1 and role are stored in the table.

RoleComplete: Note whether all the filled roles of individual1 and individual2 are stored in the table.

Besides these main tables, LAS needs to create some supplementary tables to assist in executing the propagation operation and merging test.

13. Tmp_DesParents (DesParents, DesChildren): This table is used to store the temporary parent-child concept pairs.

    DesParents: Store the atomic concepts which replace '/' in the table DesParents with '*'.

    DesChildren: Store the children concepts which replace '/' in the table DesParents with '*'.

14. Tmp_DesAncestors (DesAncestors, Descendants): This table is used to store the temporary ancestor-descendant concept pairs.

    DesAncestors: Store the atomic concepts propagated from the table Tmp_DesParents.

    Descendants: Store the descendants propagated from the table Tmp_DesParents.

15. Tmp_Rparents (Rparents, Rchildren): store the temporary parent-child role pairs.

    Rparents: Store the roles which replace '/' in the table RParents with '*'.

    Rchildren: Store the children roles which replace '/' in the table RParents with '*'.

16. Tmp_RAncestors (RAncestors, RDescendants): This table is used to store the temporary ancestor-descendant role pairs.

    RAncestors: Store the roles propagated from the table Tmp_Rparents.

    RDescendants: Store the descendant roles propagated from the table Tmp_Rparents.

17. TempDesPM (InPseudoModelID, InA, InNotA, InExistence, InUniversal, Unique): This table is used to store the pseudo models of a temporary description or an individual.

InPseudoModelID: Store the description or individual pseudo model IDs, which are assigned by the system.

InA: Store MA sets.

InNotA: Store M¬A sets.

InExistence: Store M∃ sets.

InUniversal: Store M∀ sets.

Unique: Note whether it is the only possible pseudo model.

There are two tables created for storing the propagation.

18. DesAncesors (DesAncestors, Descendants): This table is used to store the final ancestor-descendant atomic concept pairs.

DesAncestors: Store the atomic concepts which replace '*' in the table Tem_DesAncestors with '/'.

Descendants: Store the descendant concepts which replace '*' in the table Tem_DesAncestors with '/'.

19. RAncestors (RAncestors, RDescendants): This table is used to store the final ancestor-descendant role pairs.

RAncestors: Store the roles which replace '*' in the table Tem_RAncestors with '/'.

RDescendants: Store the descendant roles which replace '*' in the table Tem_RAncestors with '/'.

Connect to Racer

The Racer Server can be executed both under the Linux and Windows operating systems. A user can simply run it from a shell or double click the program icon in a graphics-based environment. If the user supplies an ontology and a query file, he can directly get the query results by typing commands in the shell. But for our system requirement, we need to connect to Racer with our Java-based system. The good thing is, Racer also offers an extensible Java client interface: JRacer [16], making it straightforward to implement interactions between the reasoning engine and the Java-based user interface. The main idea of JRacer is to execute the Racer commands using Java codes, and then return the query results with string formats. The following algorithm expresses how to send a query "(all-individuals)"running at ip: 127.0.0.1 under port: 8088 to the Racer Server, and get the query result: racerResult.

```
Algorithm Get_racer_queryResult()
RacerSocketClient client=null;
client = new RacerClient("127.0.0.1", 8088);
client.openConnection();
racerCommand="(all-individuals)";
```

racerResult = client.synchronousSend(racerCommand);

return racerResult;

Parse Racer's Output

In general, a Jracer's output is a simple string, and it can not be stored in the database directly. Therefore it must be translated into a vector, which is a stack filled with elements. Each element can be an atomic concept, a role or an individual. After getting the vector, LAS can store its elements into the related tables. Hence, parsing Racer's result is actually a process of applying the Java class String. There are some crucial methods we applied in our program:

- charAt (int index) to return the character at the specified index of the string.

- indexOf (char ch) to return the index location of the first occurrence of the specified character.

- substring (int beginIndex, int endIndex) to return a new string that is a substring of the string.

- length () to return the number of characters in the string.

The other important approach we used in our program is the Pattern class which is a compiled representation of a regular expression [12]. In order to apply it a regular expression, specified as a string, it must be compiled into an instance of this Pattern class at first. The resulting pattern can then be used to create a Matcher object that matches arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern. A typical invocation sequence is: