Efficient Conjunctive Queries over Semi-Expressive Ontologies

Used on Triple Stores

Master Thesis

Submitted by

Christian Neuenstadt
Matriculation Number 20523185

Supervised by

Prof. Möller
Prof. Zimmermann

Technische Universität Hamburg-Harburg

TU Hamburg-Harburg

Abstract

The Semantic Web has become an important movement in the internet during the past years. A general issue in this context is reasoning on large ontologies. Traditional reasoning strategies rely on efficient main memory data structures. As the growing amount of assertional statements and data is more and more reaching the capacity of standard user computers, there is a need for new reasoning strategies. Sebastian Wandelt has shown in 2011 how to release this main memory burden from tableau based reasoning systems. His idea was to create smaller subsets of the original data for reduced instance checks. Therefore, he made use of several modularization strategies that are able to split assertional data into logical parts for reasoning on small main memory.

In this thesis we first of all want to extend the idea of Sebastian Wandelt to be able to use even grounded conjunctive queries efficiently on small main memory machines. For this purpose we use an architecture that is a combination of a new improved client and an AllegroGraph triplestore.

In a second step, we want to show the possibility of converting any existing ontology on an AllegroGraph store into small modules directly on the fly and without any preparation and splitten files as in the approach of Wandelt.

We first recap the modularization strategy of Sebastian Wandelt and further describe how we think conjunctive queries can be efficiently used on small ontology modules by implementing islands directly on the AllegroGraph server. Finally we improve our system with the use of specific new skeleton queries and show how we have implemented the client system.

Afterwards, we evaluate the architecture by using the LUBM Benchmark ontology, especially their own designed testqueries. Our conclusion is that we are able to use all kinds of grounded conjunctive queries, but we are reaching performance issues for some barely selective queries. Additionaly we can show that computing ABox modularization for an existing ontology on the fly is possible, but requires a lot of communication between our client and the AllegroGraph server, which reduces the performance. Therefore, we present several improvement possibilities for the future.

Christian Neuenstadt

Contents

List of Figures

1 Introduction

1.1 The Semantic Web and Reasoning

The so called Semantic Web can be seen as a collaborative Movement started more than ten years ago in a Scientific American article [Lee et al. 2001] by Tim Berners-Lee, the inventor of the World Wide Web. In this work he defined the Semantic Web as “a web of data that can be processed directly and indirectly by machines.” During the past years the Semantic Web movement led by the World Wide Web Consortium[1] (W3C) has proposed several standard data formats to improve the processing of machines on webdata. Some examples for standardized and promoted formats are: the Resource Description Framework (in Chapter 2.3.1), RDF Schema (in Chapter 2.3.2), the Web Ontology Language (in Chapter 2.3.3) and SPARQL (in Chapter 2.4).

The interest in Semantic Web application has increased, e.g. digital libraries [Kruk and McDaniel 2009][Goncalves et al. 2008], community management [Brickley and Miller 2000][Maicher and Park 2006] and health care systems [Doms and Schroeder 2005][Cornet and De Keizer 2008]. According to the increased number of applications the size of its data grows extremely fast. Since the Semantic Web is expected to grow even further in the next years, applications require an additional amount of flexibility, scalability and performance to overcome future challenges. Such systems make intelligent use of promoted data format standards like OWL.

Today exist several solutions for query answering on Semantic Web Data in different Description Logics. To receive knowledge from standardized data so called reasoners are used. Examples for such machines are Racer [Haarslev et al. 2004] and Pellet [Sirin et al. 2007]. However, those systems show bad performance in context with large ontologies, as the implementation of the tableau based algorithms relies on efficient and high performance main memory. If the ontology representation does not fit into a relatively small main memory, the system will fail because of memory error or expensive operating system activities. Implementations, that make successful and efficient use of external memory have yet to be shown.

[1] www.w3.org

1.2 Goal of the Thesis

As a result of the increasing number of Semantic Web applications a new kind of external memory-based retrieval systems has been developed during the past. These “triple stores”, originally designed to store RDFS information, are currently getting more and more in the focus, for instance OWLIM [Kiryakov et al. 2005] or Franz AllegroGraph [Allegrograph. 2011], which we will use in this thesis. Triplestores show impressive performance results, if it comes to retrieval systems on external memory. But there are two weaknesses in this context.

First, the exact kind of reasoning used is not clear from outside. It could be anything from simple lookup to complex description logic reasoning. And second, the bar that inventors have set for hardware usage is pretty high. Using four parallel computers with 48 GB of main memory each looks actually contradictory to the idea of triple stores, which is based on using external memory.

Several strategies to overcome the problem of small main memory usage try to approximate or summarize the given input to the main memory, which usually comes along with a reduction of expressivity or information [Cruz 2007][Dolby et al. 2007]. Other existing approaches make use of modularization techniques and try to extract independent modules out of the whole external data with respect to a specific reasoning problem. Most of these modularization techniques concentrate on TBox modularization as for instance shown in [Grau et al. 2006]. A first demonstration of ABox modularization was finally shown in [Guo et al. 2005].

However, our review shows that many current implemented reasoning technologies make use of main memory techniques. Therefore, these systems often have difficulties when it is needed to handle large ABox data which can not fit into main memory. Proposed solutions often use less expressive description logics.

This Thesis is based on an approach by Sebastian Wandelt in 2011 [Wandelt 2011]. In his work he proposed a way of using ABox modularizations to optimize query answering with tableau-based reasoning systems. Therefore, he focused on a class of description logics which we call semi-expressive. This description logic is also known as $\mathcal{SHI}$ (no nominals and no choose rule). He also showed the possibility for instance checks and instance retrieval on large ontologies efficiently and provided updatable index data structures for reasoning.

The main goal of the research presented in this thesis is not only to further extend the approach of Wandelt to make efficient use on grounded conjunctive queries with external storage on a triple store database. We want to show a fast and efficient method for grounded conjunctive queries, which is a lot better then the normal naive approach and makes use of specific skeleton queries and SPARQL language.

As Sebastian Wandelt used preconfigured files with an ontology that is on the one hand not implemented on amy server and on the other hand already split in parts before the actual modularization step, our second goal is to show a way to create ABox modularization by using an already existing ontology directly on the AllegroGraph store and convert this one on the fly into small modules.

1.3 Outline

In the following Chapter 2, an overview over basics needed to understand the work in this thesis is given. The focus lies on existing Semantic Web Standards like RDF and OWL. In the second part, important preliminaries and formal notions are defined for instance the handling of graphs and description logics. Furthermore, we have a look at ontologies and reasoning procedures.

Chapter 3 explaines the work of breaking down large ontologies to small parts from Sebastian Wandelt. His idea was it to rewrite new assertional parts into smaller chunks or modules. To solve decision problems on these smaller chunks only. His modularization techniques are further extended to break up existing assertions by so called intensional-based partitioning, which he showed first in $\mathcal{ALC}$ and further lifted up to $\mathcal{SHI}$.

Chapter 4 shows first, based on the work from Wandelt explained in Chapter 3, how to make efficient use of ABox modularizations for instance checking and retrieval. It proposes the strategie of individual islands. These islands are based on individuals containing assertional axioms which are necessary for specific instance checking. Furthermore, similarity measures are introduced for optimizing instance retrieval over islands. In the last part of this chapter we have a look on the use of conjunctive queries and their optimal usage on islands and triple stores.

Later we propose a prototypical implementation in Chapter 6.2, which uses the explained algorithms in Chapter 3 and 4 to show our improvements on conjunctive queries in practice. This will be evaluated over test ontologies and given test queries to proof the scalability on large ontologies.

In Chapter 7 we conclude our work and summarize the main issues to indicate interesting topics in the future.

2 Preliminaries

This chapter will introduce basic knowledge on Ontologies and Semantic Web standards. First, we define required mathematical knowledge and introduce the language and axioms for the description of ontologies in Section 2.1 and 2.2. Furthermore, in Section 2.2.3 we make use of these definitions to create an example ontology and discuss languages and frameworks that are used to describe ontologies in general. Finally, we explain ontology storage on external memory on AllegroGraph triple stores in Chapter 2.4.

2.1 Basic Mathematical Notions

In this section basic mathematical notions will be defined. We will start with the definition of sets.

Sets A set is a collection of well defined distinct objects. They are conventionally denoted with capital letters. The sets $\mathbf{A}$ and $\mathbf{B}$ are equal if and only if they have exactly the same elements. If every element of $\mathbf{A}$ is also element of $\mathbf{B}$ then $\mathbf{A}$ is called the subset of $\mathbf{B}$ denoted with $\mathbf{A} \subseteq \mathbf{B}$. The powerset $\wp(\mathbf{S})$ of set $\mathbf{S}$ is defined as the set of all subset of $\mathbf{S}$, including the empty set and $\mathbf{S}$ itself. The number of all elements in a set $\mathbf{S}$ is called the cardinality and denoted with $|\mathbf{S}|$.

Relations Given a collection of sets $\mathbf{X}_1, ..., \mathbf{X}_n$ the n-ary relation $\mathbf{R}$ over $\mathbf{X}_1, ..., \mathbf{X}_n$ is a subset of $\mathbf{X}_1 \times ... \times \mathbf{X}_n$. The type of a relation $\mathbf{R}$ is donated with $\mathbf{R} : \mathbf{X}_1 \times ... \times \mathbf{X}_n$. A binary relation $\mathbf{R}$ is given by the relation over two sets $\mathbf{X}$ and $\mathbf{Y}$. This can also be donated with x$\mathbf{R}$y. Several different kinds of binary relations are:

- *left-total* if $\forall x \in \mathbf{X} : \exists y \in \mathbf{Y}$ such that $\mathbf{R}(x, y)$,
- *surjective* if $\forall y \in \mathbf{Y} : \exists x \in \mathbf{X}$ such that $\mathbf{R}(x, y)$,
- *injective* if $\forall x_1, x_2 \in \mathbf{X} : \forall y \in \mathbf{Y} : \mathbf{R}(x_1, y) \wedge \mathbf{R}(x_2, y) \implies x_1 = x_2$,
- *bijective* if $\mathbf{R}$ is surjective and injective.

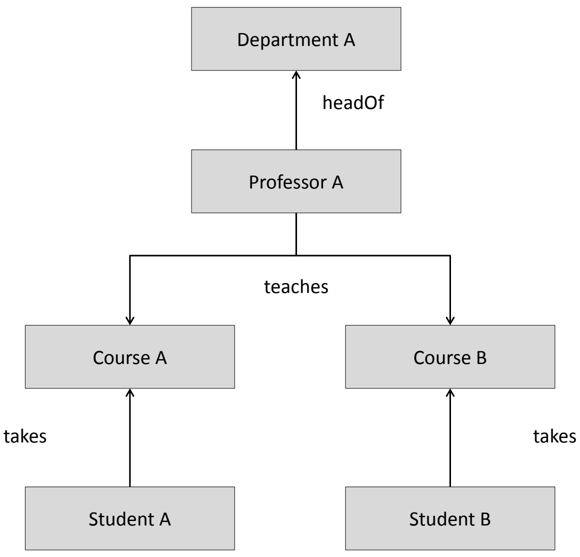


Figure 2.1: Example of an directed graph with successors, predecessors and neighbours

Given a binary relation $\mathbf{R} : \mathbf{X} \times \mathbf{X}$ we define

- *reflexive closure* of $\mathbf{R}$ as $\mathbf{R} \cup \{(x, x) \mid x \in \mathbf{X}\}$
- *symmetric closure* of $\mathbf{R}$ as $\mathbf{R} \cup \{(x_1, x_2) \mid (x_2, x_1) \in \mathbf{X}\}$
- *transitive closure* of $\mathbf{R}$ as $\mathbf{R}(x_1, x_2) \wedge \mathbf{R}(x_2, x_3) \Longrightarrow \mathbf{R}(x_1, x_3)$

Directed Graphs A tuple $\mathbb{G} = \langle \mathbf{N}, \mathbf{E} \rangle$ is called directed Graph, where $\mathbf{N}$ is a set of nodes and $\mathbf{E}$ is a set of edges. Given a node n $\in \mathbf{N}$, an edge $(n_2, n) \in \mathbf{E}$ is called *incoming edge* denoted $in_{\mathbb{G}}(n)$. An edge (n, n_2) is called *outgoing edge* denoted $out_{\mathbb{G}}(n)$. The *successor nodes* of n, denoted as $succes_{\mathbb{G}}(n)$ are nodes, that are connected by an outgoing edge from n. The *predecessor nodes* of n, denoted as $preds_{\mathbb{G}}(n)$ are connected by an incoming edge to n. Where *node neighbours*, denoted $neighbors_{\mathbb{G}}(n)$, are connected with an outgoing or incoming edge.

Trees A tuple $\mathbb{T} = \langle \mathbf{N}, root, children \rangle$ is called a *directed tree*, such that $\mathbf{N}$ is a set of nodes, $root \in \mathbf{N}$ is a distinguished root node, and children : $\mathbf{N} \rightarrow \wp(\mathbf{N})$ is a function which assigns each node to a number of child nodes, where every node except the rootnode is reachable from the root and has exactly one predecessor. Furthermore, a node n $\in \mathbf{N}$ is called *leaf node* if children(n) $= \emptyset$, otherwise it is called *inner node*.

2.2 Description Logics

In this section we will shortly introduce description logics, a family of logic-based knowledge representation languages that can be used to represent the terminological knowledge of an application domain in a structured way. Description logics are commonly used in topics of artificial intelligence for formal reasoning on concepts of an application domain. Fundamental for Description Logics is the modeling of concepts, roles and individuals as well as their relationships. The basic modeling concept are axioms, which are simple logical statements to create relations between roles and concepts.

2.2.1 Naming convention

Description Logics can be divided into many different varieties. Therefore there exists a naming convention which roughly describes all allowed operators and their expressivity.

A common variety is the description base language $\mathcal{AL}$ or attributive language, which was first introduced in [Schmidt-Schauß and Smolka 1991].

$\mathcal{AL}$ Attributive language allows:

- Atomic negation
- Concept intersection
- Universal restrictions
- Limited existential quantification

Common extensions for description base languages are:

- $\mathcal{F}$ Functional properties,
- $\mathcal{U}$ Concept union,
- $\mathcal{C}$ Complex concept negation,
- $\mathcal{H}$ Role hierarchy (subproperties),
- $\mathcal{R}$ Limited complex role inclusion axioms, reflexivity and irreflexivity,
- $\mathcal{O}$ Nominals,
- $\mathcal{I}$ Inverse properties,
- $\mathcal{N}$ Cardinality restrictions,
- $\mathcal{Q}$ Qualified cardinality restrictions.

To create conventions for description logics any combination of the base language with extensions are now possible. For instance $\mathcal{ALC}$ is obtained from $\mathcal{AL}$ by adding the complement operator ($\neg$). Compared to $\mathcal{AL}$ in $\mathcal{ALC}$ complex negation of whole concept axioms is possible, where in $\mathcal{AL}$ only negation of simple atomic concepts is allowed. $\mathcal{S}$ stands for an abbreviation for the description logic $\mathcal{ALC}$ with added transitive roles. In this thesis we will describe concepts in $\mathcal{ALC}$ with role hierarchies, inverse properties and transitive roles. Thus, the following concepts are described in $\mathcal{SHI}$ a shorter form for $\mathcal{ALCHI}$ with transitive roles.

2.2.2 Modeling

Description languages use a given formal syntax. In the following we give a definition on syntax and semantics of the constructors we describe. We will use $\mathcal{ALC}$ as an example.

Concept Descriptions Let **CN** be a set of concept names and **RN** be a concept of role names. The expression C is a concept description of $\mathcal{ALC}$ if and only if:

- C = $\top$,
- C = $\bot$,
- C = A, such that A $\in$ **CN**,
- C = $C_1 \sqcup C_2$, such that C_1 and C_2 are concept descriptions (union),
- C = $C_1 \sqcap C_2$, such that C_1 and C_2 are concept descriptions (intersection),
- C = $\neg C_2$, such that C_2 is a concept descriptions (negation)
- C = $\exists R.C_2$, such that C_2 is concept description and R is a role description (existential restriction)
- C = $\forall R.C_2$, such that C_2 is concept description and R is a role description (value restriction)

Example for concept description Given a set of concept names **CN** = {GraduateStudent, UnderGraduateStudent, Course} and a set of role names **RN** = {teaches, takes}, possible examples for concept descriptions are:

- Atomic concepts: GraduateStudent, Course,
- Non atomic concepts: GraduateStudent $\sqcup$ UnderGraduateStudent, $\exists takes.Course$,
- Negated concept names: $\neg GraduateStudent$, $\neg Course$,

Interpretations An interpretation $I = (\Delta^I, \cdot^I)$ consist of a set of Δ^I and a function $\cdot^I$ that maps every concept to a subset of Δ^I and every rolename to a subset of $\Delta^I \times \Delta^I$ such that for all concept names **CN** and all role names **RN**:

- $(\top)^I = \Delta^I$,
- $(\bot)^I = \emptyset$,
- $(C_1 \sqcap C_2)^I = C_1^I \cap C_2^I$,
- $(C_1 \sqcup C_2)^I = C_1^I \cup C_2^I$,
- $(\neg C)^I = \Delta^I \setminus C^I$,
- $(\exists R.C)^I = \{x \in \Delta^I | \exists y. \langle x, y \rangle \in R^I \wedge y \in C^I\}$,
- $(\forall R.C)^I = \{x \in \Delta^I | \forall y. \langle x, y \rangle \in R^I \rightarrow y \in C^I\}$.

Example for interpretations Given a set of concept names **CN** = {GraduateStudent, UnderGraduateStudent, Course} and a set of role names **RN** = {teaches, takes} possible examples for interpretation descriptions are:

- $\Delta^I = \{\delta_a, \delta_b, \delta_c\}$,
- $Course^I = \{\delta_a\}$,
- $GraduateStudent^I = \{\delta_b, \delta_c\}$,
- $takes^I = \{(\delta_b, \delta_a), (\delta_c, \delta_a)\}$,

Closure of Concepts For syntactical analysis we introduce the closure of a concept description, which is used as notion of a concept description in negation normal form in order to further ease syntactical analysis.

Given a concept description C, the *concept closure* of C, denoted $clos(C)$, is defined as follows:

$$clos(C) \begin{cases} \{\top\} & \text{if C} = \top \\ \{\bot\} & \text{if C} = \bot \\ \{A\} & \text{if C} = A \\ \{\{a\}\} & \text{if C} = \{a\} \\ \{C\} \cup clos(C_1) \cup clos(C_2) & \text{if C} = C_1 \sqcup C_2 \\ \{C\} \cup clos(C_1) \cup clos(C_2) & \text{if C} = C_1 \sqcap C_2 \\ \{C\} \cup clos(C_1) & \text{if C} = \neg C_1 \\ \{C\} \cup clos(C_1) & \text{if C} = \forall R.C_1 \\ \{C\} \cup clos(C_1) & \text{if C} = \exists R.C_1 \end{cases}$$

Negation Normal Form The negation normal form defines that all negations have to occur in front of atomic concepts or concept names only. A concept description in negated normal form is denoted as *nnf*(C), where C $\in$ **CN**.

Every concept can be transformed into a concept description in negation normal form. For details see [POUR 2012].

2.2.3 Knowledge Base

A tuple K = $\langle T, A\rangle$ is called knowledge base, where T is a TBox and A is an ABox. The TBox stores a finite set of constraints stating general properties of concept and roles of the form $C \sqsubseteq D$ and $C = D$, where C and D are concept expressions. Compared to the TBox, the ABox comprises assertions on individual objects of the form C(a) and individual role assertions R(a, b) where a, b are names of individuals. A typical assertion in the ABox defines that an individual is an instance of a certain concept.

The TBox can be seen as a kind of an Entity-Relationship Model in databases that has a highly expressive semantic for description logics, defined in the terms of interpretation. Another important issue of description logic is the possibility of reasoning over T- and ABox which is associated with the knowledge base. Several reasoning tasks can be carried out. The simplest form of reasoning involves computing the subsumption relation between two concept expressions. Another way is to check whether a certain assertion is logically implied by a knowledge base, which can be a more complex reasoning task.

Example Ontology

In this section we will present an example of an ontology, consisting of T-, A- and RBox. This ontology is related to the one used in the LUBM project of Lehigh University [Guo et al. 2005]. Which we will use entirely during this thesis, especially benchmarking and evaluating of the proposed prototype. Subsets of this example will be used for further demonstrations in the following chapters. An example graph is shown in Figure 2.2.

Example The example ontology $\mathcal{O}_{Ex} = \langle \mathcal{T}_{Ex}, \mathcal{R}_{Ex}, \mathcal{A}_{Ex} \rangle$ is defined as follows:

$$
\begin{aligned}
\mathcal{T}_{Ex} = \quad \{ & \\
& Chair \equiv \exists headOf.Department, \\
& Student \equiv \exists takes.Course, \\
& Student \sqsubseteq person, Professor \sqsubseteq person, \\
& UndergraduateCourse \sqsubseteq Course, \\
& GraduateCourse \sqsubseteq Course, \\
& GraduateCourse \sqcap UndergraduateCourse \sqsubseteq \bot, \\
& \top \sqsubseteq \forall teaches.Course, \top \sqsubseteq \forall takes.Course, \\
& \exists memberOf.\top \sqsubseteq Person \\
& \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{R}_{Ex} = \quad \{ & \\
& headOf \sqsubseteq memberOf, \exists headOf.Department, \\
& teaches \equiv isTaughtBy^{-} \\
& \}
\end{aligned}
$$

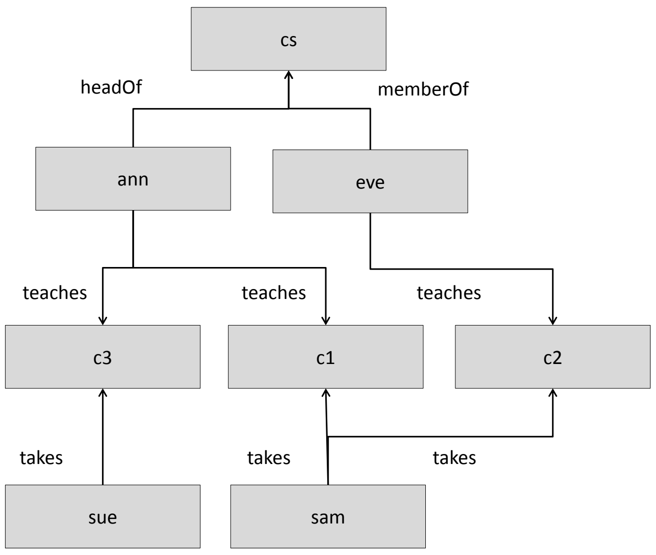


Figure 2.2: Graph of the example ontology

$$
\begin{aligned}
\mathcal{A}_{Ex} = \quad \{ & \\
& Department(cs) \\
& Professor(ann), Professor(eve) \\
& UndergraduateCourse(c1), UndergraduateCourse(c2), \\
& GraduateCourse(c3), \\
& Student(sam), Student(sue) \\
& headOf(ann, cs) \\
& memberOf(eve, cs) \\
& teaches(ann, c1), teaches(ann, c3), teaches(eve, c3), \\
& takes(sam, c1), takes(sam, c2), takes(sue, c3) \\
& \}
\end{aligned}
$$

2.2.4 Inferences

The basic idea of knowledge representation systems based on Description Logics is to perform specific kinds of reasoning. Their purpose goes beyond storing concept definitions and assertions. A knowledge base including TBox and ABox uses semantics that makes it equivalent to a set of axioms in first-order predicate logic. Thus, like any other set of axioms, it contains implicit knowledge that can be made explicit through inferences. Considering the university example it is clearly visible that the Professor who teaches *sues* course is named *ann*. Although it is not directly written down. Gaining these new informations in a machine based process is called *reasoning*. Reasoners are able to reason in several ways on ABox and TBox. In the following section, we have a look at these inferences for TBoxes and ABoxes.

For TBox reasoning there are five different reasoning tasks used:

Knowledge Base Consistency: A TBox **T** is consistent if its negation is not a tautology. I.e., **T** is inconsistent if there is no interpretation which entails **T**.

Satisfiability: A concept **C** is satisfiable with respect to **T** if there exist a interpretation **I** of **T** such that C^I is true.

Subsumption: A concept **C** is subsumed by a concept **D** with respect to **T** if $C^I \subseteq D^I$ for every model **I** of **T**.

Equivalence: Two concepts **C** and **D** are equivalent with respect to **T** if $C^I = D^I$ for every model **I** of **T**.

Disjointness: Two concepts C and D are disjoint with respect to **T** if $C^I \cap D^I = \emptyset$ for every model **I** of **T**.

Usually, the basic reasoning mechanism checks only the satisfiability of concepts. This is actually sufficient to implement as the other inferences can be converted into satisfiability due to the following reductions. If a system also allows negationform one can reduce the problem to the satisfiability problem. For a concept C and a concept D we define:

- C is unsatisfiable $\rightarrow C \sqsubseteq \bot \rightarrow C \sqcap \neg D$ is unsatisfiable
- C and D are equivalent $\rightarrow C \sqsubseteq D \wedge D \sqsubseteq C \rightarrow (C \sqcap \neg D)$ and $(\neg C \sqcap D)$ are unsatisfiable
- C and D are disjoint $\rightarrow C \sqcap D \sqsubseteq \bot \rightarrow C \sqcap D$ is unsatisfiable.

For reasoning on ABoxes one has to consider that there are only concept membership assertions of the form C(a) and role membership assertions of the form R(a,b). Therefore, the ABox can only be seen as a knowledge base attached with its TBox and ABox reasoning can also only be done with respect to its TBox. The basic reasoning services used on ABox are:

Instance Check: Determines whether an assertion is entailed by ABox A ($A \models C(a)$). An assertion is entailed if every interpretation that satisfies A also satisfies C(a).

Instance Retrieval: Given an ABox A and a concept C, find all individuals a such that $\{a \mid A \models C(a), a \in A\}$.

Instance Realization: Given an individual a and a set of concepts, find the most specific concepts C such that $\{a \mid A \models C(a), a \in A\}$.

ABox Consistency: An ABoxis consistent if it is consistent with respect to the TBox.

2.2.5 Conjunctive Queries

We have already briefly talked about rather weak forms of querying like instance retrieval and instance checks in the last section. With conjunctive queries (CQs) we add another well known topic in the database community.

Definition A conjunctive query or CQ can be represented as $q(\vec{x}) \leftarrow conj(\vec{x}, \vec{y})$, where $q(\vec{x})$ is a query of the conjunctive set $conj(\vec{x}, \vec{y})$. The two vectors $\vec{x}$ and $\vec{y}$ are vectors of variables. $\vec{x}$ is the vector of so called distinguished variables that are bound to individuals (single objects) of the knowledge base used to answer the query; $\vec{y}$ is the vector of non-distinguished variables (existentially quantified variables). $conj(\vec{x}, \vec{y})$ itself is a conjunction of terms of the form v_1 : C, $\langle v_2, v_3 \rangle$: R where C is a concept name, R is a role name and v_1, v_2, v_3 are variables from $\vec{x}$ or $\vec{y}$ (see [Dongilli 2008]).

Example For a CQ example, consider a knowledge base containing an ABox assertion $(\exists hasDaughter.(\exists hasSon.\top))(Sue)$, this assertion informally states that the individual Sue has a daughter who has a son, which says nothing else than Sue is a grandmother. And hence, we additionally assume that both roles *hasSon* and *hasDaughter* have a transitive super-role named *hasDescendant*, which implies a direct relationship between her and her grandchild via that role. Alltogether, Sue is obviously an answer to the conjunctive query $hasDaughter(x, y) \cap hasSon(y, z) \cap hasDescendant(x, z)$, as far as we assume x as a free variable and *y,z* as quantified. We call free variables like x distinguished variables and quantified variables non-distinguished. If all variables are non-distinguished, all variables are set and the query answer can only be true or false. We call this a boolean query (see: [Abiteboul et al. 1995]).

In this thesis, we present a tableau-based reasoning solution on triplestores for answering grounded conjunctive queries over large semi-expressive ABoxes. Grounded conjunctive queries use distinguished atomic variables only, are more realistic in practice and can be answered efficiently for expressive Description Logics.

Certain Answers The notion of answers to a query we used in this section is not sufficient to capture the situation where a query is posed over an ontology, since in general an ontology will have many models, and we are not able to single out a unique interpretation (or database) over which to answer the query. Instead, the ontology determines a set of interpretations (see 2.2.2), i.e., the set of its models, which intuitively can be considered as the set of databases that are "compatible" with the information specified in the ontology. Given a query, we are interested in those answers to this query that depend only on the information in the ontology, i.e., that are obtained by evaluating the query over a database compatible with the ontology, but independently of which is the actually chosen database. In other words, we are interested in those answers to the query that are obtained for all possible databases (including infinite ones) that are models of the ontology. This corresponds to the fact that the ontology conveys only incomplete information about the domain of interest, and we want to guarantee that the answers to a query that we obtain are certain, independently of how we complete this incomplete information. This leads us to the following definition of certain answers to a query over an ontology (see [Calvanese et al. 2009]).

Definition Given an Ontology $\mathcal{O}$ and a conjunctive query q over $\mathcal{O}$. A tuple c of constants appearing in $\mathcal{O}$ is a *certain answer* to q over $\mathcal{O}$, written $c \in cert(q, O)$, if for every model $\mathcal{I}$ of $\mathcal{O}$, we have that $c^{\mathcal{I}} \in q^{\mathcal{I}}$.

Answering a query q posed to an ontology $\mathcal{O}$ means exactly to compute the set of certain answers to q over $\mathcal{O}$.

Example We consider again the previous Example 2.2.5 from the last section and find:

$$cert(q, \mathcal{O}) = \{(Sue^{\mathcal{I}})\}.$$

2.3 Semantic Web Data Formats

In this section we will introduce several Semantic Web standardizations which are promoted by the W3C [Consortium 2012] to make the web more readable by machines. The described standards are used further in this work and specifically for the prototype implementations. We describe briefly the resource description framework (see 2.3.1), the web ontology language (see 2.3.3) and SPARQL queries (see 2.4).

There currently is a lot of knowledge on the web, but its readability by machines is limited. Consider a webpage on Wikipedia, which offers a lot of information to the human reader. But for the computer information is opaque and stays like a simple presentation for human users.

What is meant by "semantic" is not that computers are going to understand the meaning of anything. But you can imagine it as a web of many databases. So application are to combine informations from several places. Consider a Website with a database about a single productline, another with product reviews and a third with retailer stocks and prizes. This becomes now easy to be meshed up together in a single application, if all informations are "semantic".

To guarantee machine readability the following standards are used.

2.3.1 The Resource Description Framework (RDF)

The Resource Description Framework as a family of the W3C specifications was originally designed as a metadata data model. Nowadays it has become to a general method for conceptual description of information that is implemented in web resources by a variety of syntax formats. The RDF data model itself is based on classic modeling approaches like class diagrams or entity-relationship models.

Triples of knowledge With RDF a general and flexible methode to decompose knowledge into small pieces is provided. Those pieces are also called triples. The foundation is represented in basically what we call a labeled, directed graph for a known terminology. Each edge in the graph is represented in RDF by one statement. If we consider our example from figure 2.2 for instance, one statement of this graph is "Sue takes c3" and another "eve teaches c2". Though, Semantic Web information is typically represented as a set of statements consisting of three elements called triples. Every statement can be divided into three parts *Subject*, *Predicate* and *Object*. All triples together make a an ontology graph, where subjects and objects are nodes and predicates are edges. Having this graph of statements, a data model is already given and a machine could now easily answer questions like "who teaches c2?" or "which courses are taught in department cs?". But a computer does not really need to know what "teaches" actually means, so it is left to the application writer to use the right predicates.

Usage of URIs As RDF is meant to be published on the web every used identifier has to be absolutely unique. Otherwise computers could identify for example many different *eves* as teacher of course *c2*. Therefore, we have to use global names, so called Uniform Resource Identifiers (URI). URIs can have the same syntax or format as website addresses, as those can be quite long we usually stick to the concept of namespaces as an abbreviation.

One example for using RDF as an external database are triple stores like *Allegro-Graph* which is further explained in 2.4.

2.3.2 RDF Schema

The presented Resource Description Framework (RDF) provides a way of modeling information, but does still not present a meaning of these information. For example it identifies *eve* as the one who is *teacher of* course *c2*. But there is absolutely no information who or what *eve* actually is.

RDF Schema now provides an extension to RDF and introduces the notion of classes. A class can simply be seen as the type of a thing. For instance in the case of *eve*, *eve* is a *Professor*. *Sam* is a *Student* and *c2* is of class *Course*. Classes allow expressive semantics for query answering with respect to ontologies. To add class information an application writer would just add another triple to the RDF files. In the case of *eve* this is simply:"lubm:eve rdf:type lubm:Professor". Where *lubm* and *rdf* are namespaces.

RDFS vocabularies also describes the classes of resources and properties in a model allowing to arrange classes and properties in generalization hierarchies and define domain as well as range expactations for properties. In RDFS one named class can be a subClass of a more generalized class. However, all classes are instances of the class rdfs:Class and all properties are instances of rdf:Property. All Properties have defined domains and ranges, which says that every resource, called domain, that has the given property, must be of that specific type. And every Range, which is the value of the property, must be instances of the property class. Though, in the given example the domain of the property *teacherOf* can be stated as class *Professor* and the range as class *Course*.

2.3.3 Web Ontology Language (OWL)

The Web Ontology Language is another web standard promoted by the W3C and written in XML. With the presented standard of RDFS it is now possible to define class hierarchies as well as domain and range restrictions for properties. But additional resources are needed for more complex ontologies. Therefore, a new standard for a richer language was proposed, the Web Ontology Language (OWL). OWL extends the RDFS vocabulary with additional resources that can be used to build more expressive ontologies for the web and adds several new possibilities to the Semantic Web standard. Some of the including features are:

Cardinality: restricts the number of elements to a minimum or maximum value. Can be used to set the minimum of students who take a course to one.

Equality: makes use of equivalent classes. The class of professors who are head of department is equal to the class *chair*.

Relationships between classes: explains the relation between different classes. The union of *UndergraduateStudents* and *GraduateStudents* is used to be the class *Student*.

Characteristics of properties: defines properties as transitive, symmetric, functional or inverse property.

The combination of a language with an expressivity as powerful as a combination of RDF Schema with full logic and efficient reasoning support seems to be incompatible and have lead to define three different sublanguages. Each of these fullfilling different aspects of the incompatible requirements. Those sublanguages are *OWL FULL*, *OWL DL* and *OWL Lite*. For more information on OWL and the three sublanguages we refer to [McGuinness et al. 2004].

In the recent update of OWL 2 the W3C introduced three different profiles: OWL EL, OWL RL, and OWL QL. They are lightweight sublanguages of OWL, which restrict the modelling features to simplify the reasoning procedure and has an large impact on performance and scalability in respect to OWL 2 (see [Motik et al. 2009]).

2.4 AllegroGraph

In the following section we will briefly explain the external database structures used in this thesis.

AllegroGraph is a closed source external graph database or triplestore. A triplestore is a purpose-built database for storage and retrieval of triples, where one is composed of subject, predicate and object. Related to relational databases, one can store information and retrieves it via query language. Triples together form a network or graph. In the case of AllegroGraph, triples are saved with special indexes as quadrupels or even as quintuples with context information.

As developed to meet the W3C standards for the Resource Description Framework (see 2.3.1) AllegroGraph is properly considered as a RDF Database. The commonly used query language is SPARQL. To test performance issues on AllegroGraphs ABox querying, usually the Lehigh University Benchmark Database (see [Guo et al. 2005]) is used. For more information specific on AllegroGraph, we refer to [Allegrograph. 2011].

SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a query language for DL databases, able to retrieve and manipulate data stored in RDF format. In SPARQL queries consist of triple patterns, conjunctions, disjunctions and optional patterns.

It specifies four different query variations for different purposes, that can be used within AllegroGraph:

SELECT query: Used to extract raw values from the triple store. The results are returned in table format.

CONSTRUCT query: Is able to extract information and transform it into valid RDF.

ASK query: Provides a boolean query format for queries on AllegroGraph.

DESCRIBE query: Is used to extract RDF graphs from data bases.

Example To give an example we model the question “Which teachers of undergraduate courses are also head of a department?” as a SELECT query:

Listing 2.1: Query example

```
PREFIX lubm: <http://example.com/lubmOntology#>
SELECT ?prof ?department
WHERE {
  ?prof lubm:teacherOf ?course;
        rdf:type lubm:Professor;
        lubm:headOf ?department .
  ?course rdf:type lubm:UndergraduateCourse .
  ?department rdf:type lubm:Department .
}
```

Variables are indicated by a "?" prefix. In this case the bindings for ?prof and ?department will be returned.

Different patterns can be connected here with dots to conjunctive queries. The SPARQL query engine basically searches sets of triples that matches these five patterns while binding the variables to the corresponding parts of each triple.

To make queries more concise, SPARQL allows the definition of prefixes and base URIs in a fashion similar to Turtle. In this query, the prefix *lubm* stands for *http://example.com/lubmOntology*, what is a placeholder for the Lehigh University Benchmark.

3 Modularization

As already mentioned, reasoning over large ontologies is difficult. The modularization techniques shown in the work of Sebastian Wandelt [Wandelt 2011] can help to overcome problems, where small main memory is not sufficient to store whole description logic ontologies. Here we will focus on the basic concepts of his ABox modularization, since it exceeds the size of the terminological part by orders of magnitude. Based on these preliminaries, we will show later how conjunctive queries can be used in context with ABox modularization on triple stores. We will start in this chapter by offering techniques proposed by Wandelt to break down ABoxes in smaller chunks for the description logic $\mathcal{ALC}$ (see 3.1) and then show his techniques for an implementation in $\mathcal{SHI}$ (see 3.2). We finally conclude this Chapter in 3.3.

3.1 Basic modularization techniques

In general, first of all we can define an ABox Modularization as a set of ABoxes $A_1, ..., A_n$ constructed in an ABox-Modularization process. We stick to this general notion, as ABox modules are not necessarily subsets of the original ABox. According to this, we can make a definition of ABox module entailment.

ABox Modularization Entailment Given a TBox $\mathcal{T}$, a RBox $\mathcal{R}$, a ABox $\mathcal{A}$ and an ABox modularization $\mathbf{M}$, we say that $\mathbf{M}$ entails a concept assertion C(a), denoted $\langle \mathcal{T}, \mathcal{R}, \mathbf{M} \rangle \models C(a)$, if $\exists \mathbf{A}_1 \in \mathbf{M}. \langle \mathcal{T}, \mathcal{R}, \mathbf{A} \rangle \models C(a)$. Further, we say that $\mathbf{M}$ entails a role assertion $R(a_1, a_2)$, denoted $\langle \mathcal{T}, \mathcal{R}, \mathbf{M} \rangle \models R(a_1, a_2)$, if $\exists \mathbf{A}_1 \in \mathbf{M}. \langle \mathcal{T}, \mathcal{R}, \mathbf{A} \rangle \models R(a_1, a_2)$ (see [Wandelt 2011]).

For a better understanding of ABox entailment, we will give an ontology with two different ABox modularizations as an example.

Example 3.1 The Ontology $\mathcal{O}_{Ex3.1} = \langle \mathcal{T}_{Ex3.1}, \mathcal{A}_{Ex3.1} \rangle$ is defined as follows

$$\mathcal{T}_{Ex3.1} = \{Chair \equiv \exists headOf.Department\}$$

$$\begin{aligned}\mathcal{A}_{Ex3.1} = \{ & Department(ee), Professor(mae),\\ & UndergraduateCourse(c1), headOf(mae, ee),\\ & Student(sam), Student(sue),\\ & teaches(mae, c1), takes(Sam, c1),\\ & takes(sue, c1)\}\end{aligned}$$

Example 3.2 One possible ABox modularization for ontology $\mathcal{O}_{Ex3.1}$ is $\mathbf{M}_{Ex3.2} = \{\mathbf{A}_{Ex3.2a}, \mathbf{A}_{Ex3.2a}\}$, such that

$$\begin{aligned}\mathbf{A}_{Ex3.2a} = \{ & Department(ee),\\ & headOf(mae, ee),\\ & Professor(mae)\}\end{aligned}$$

$$\begin{aligned}\mathbf{A}_{Ex3.2b} = \{ & UndergraduateCourse(c1),\\ & Student(sam), Student(sue),\\ & teaches(mae, c1), takes(sam, c1), takes(sue, c1)\}\end{aligned}$$

We can directly see from Ontology $\mathcal{O}_E x3.1$ that *mae* is an instance of the concept description *chair*, because of the *headOf*-Relationship between mae and ee. Therefore, the ABox modularization from Example 3.2 also entails that *mae* is an instance of *chair*, as all necessary axioms are kept in one ABox. Nevertheless, we could think of other possible ABox modularization. A second possible modularization is given in Example 3.1.

Example 3.3 Another possible ABox modularization for ontology $\mathcal{O}_{Ex3.1}$ is $\mathbf{M}_{Ex3.3} = \{\mathbf{A}_{Ex3.3a}, \mathbf{A}_{Ex3.3a}\}$, such that

$$\begin{aligned}\mathbf{A}_{Ex3.3a} = \{ & Department(ee),\\ & UndergraduateCourse(c1)\}\end{aligned}$$

$$\begin{aligned}\mathbf{A}_{Ex3.3b} = \{ \quad & Student(sam), Student(sue),\\ & teaches(mae, c1), takes(sam, c1), takes(sue, c1)\\ & Professor(mae), headOf(mae, ee)\}\end{aligned}$$

The result of the modularization in Example 3.1 is different. It can be seen that neither of the two ABoxes entails that *mae* is an instance of chair. Infact, they are both chosen quite arbitrarily and so the necessary information was split up into both boxes. Wandelt tries to keep all relevant information together in one ABox to avoid communication overhead and retain completeness.

Both Examples show that the choice of modularizaton is critical for completeness of instance retrieval. We will now briefly explain in the next part how Wandelt retains soundness and completeness for ABox modularization. For a detailed discussion and mathematical proofs see [Wandelt 2011].

In a component-based graphview of ABoxes, we can identify most individuals as connected by role assertions to many other individuals. To receive smaller modules, we have to split up as many of these role assertions as possible. After all of these splits are done in the modularization process, we have to make sure that the ontology retains soundness and completeness.

The idea to decide whether to split or not to split a role assertion is to analyze the terminological part of the ontology. We give an example of a split decision to show where information is propagated.

The Ontology $\mathcal{O}_{Ex3.4} = \langle \mathcal{T}_{Ex3.4}, \mathcal{A}_{Ex3.4} \rangle$ is defined as follows

$$\mathcal{T}_{Ex3.1} = \{ \quad \top \sqsubseteq \forall takes.Course\}$$

$$\begin{aligned}\mathcal{A}_{Ex3.1} = \{ \quad & Course(c1), Student(sue),\\ & teaches(mae, c1), takes(sue, c1)\}\end{aligned}$$

As we look into the ABox, we can see certain issues relating to the two role assertions *teaches* and *takes* in $\mathbf{A}_{Ex3.4}$:

- teaches(mae, c5): The role *teaches* is not used or mentioned anywhere in the TBox of ontology $\mathbf{O}_{Ex3.9}$. Thus, information can be propagated from *mae* to *c5* and vice versa by tableau algorithm (see [Wandelt 2011]), and it might be safe to split the role assertion to obtain smaller modules.

- takes(sue, c5): Although *takes* is mentioned in $\mathbf{T}_{Ex3.4}$, it is only used to propagate the concept description *Course*. Since *c1* is an instance of *Course* and that fact is directly in $\mathbf{A}_{Ex3.4}$, we might further split up this role assertion in some cases.

To decide, if a certain role assertion is splittable, we need necessary decision criteria to identify propagated concepts via role assertion. Since we make an assumption that only allows atomic concept descriptions, we can focus on a syntactical analysis of the TBox to obtain splittability information. To get over this, Sebastian Wandelt proposed a solution of the so called ∀-*info structure*, which he evolved from the TBox normal form. We will shortly define this structure and then show an example.

Definition A TBox $\mathcal{T}$ is in normal form if all concept inclusions have one of the following forms:

$$A_1 \sqsubset B,\ A_1 \sqcap A_2 \sqsubset B,\ A_1 \sqcap \exists r.A_2,\ or\ \exists r.A_1 \sqsubset B$$

where A_1, A_2 and B are concept names appearing in $\mathcal{T}$ or the top-concept $\top$.

Definition The $\forall - infostructure$ for a TBox $\mathcal{T}$ in normal form is a function $info^{\forall}_{\mathcal{T}}$: $\mathbf{Rol} \rightarrow \wp(\mathbf{Con})$, such that we have C $\in info^{\forall}_{\mathcal{T}}(R)$ if and only if $\forall R.C \in clos(\mathcal{T})$ (see 2.2.2 for closure of concepts).

An example for TBox with normal form and $\forall - infostructure$ is:

Example 3.5 Let

$$\begin{aligned}\mathcal{T}_{Ex3.5} = \{\ & \top \sqsubseteq \forall takes.Course, \\ & \exists takes.Course \sqsubseteq Student, \\ & \exists memberOf.\top \sqsubseteq Person, \\ & GraduateStudent \sqsubseteq Student\}.\end{aligned}$$

Then the TBox $\mathcal{T}_{Ex3.5}$ in normal form is

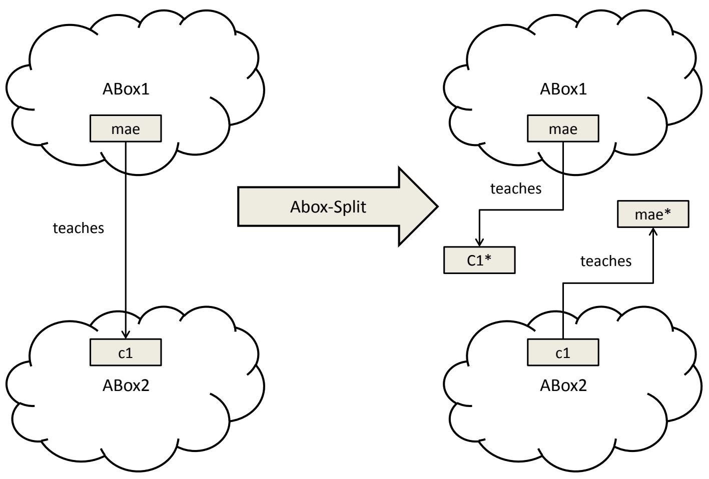


Figure 3.1: Intuition of an ABox split

$$
\begin{aligned}
\mathcal{T}_{norm} = \{ \quad & \top \sqsubseteq \forall takes.Course, \\
& \top \sqsubseteq \forall takes.\neg Course \sqcup Student, \\
& \top \sqsubseteq \forall memberOf.\bot \sqcup Person, \\
& \top \sqsubseteq \neg GraduateStudent \sqcup Student\},
\end{aligned}
$$

The $\forall$-info structure for $\mathcal{T}_{norm}$ is:

$$
info^{\forall}_{\mathcal{T}_{norm}}(R) \begin{cases} \{Course, \neg Course\} & \text{if R} = \textit{takes}, \\ \{\bot\} & \text{if R} = \textit{memberOf}, \\ \emptyset & \text{otherwise} \end{cases}
$$

The $\forall$-info structure shows which concepts descriptions are propagated over role assertions. Given the above $\forall - infostructure$, we follow up with an operation which allows us to split role assertions in a manner that we can apply graph component-based modularization techniques. This operation must retain soundness and completeness.

The idea is shown in figure 3.1. Both Clouds indicate a set of ABox assertions. We split a role assertion and keep the concept assertions for *mae* as a fresh and individual copy in ABox1 and ABox2.

Sebastian Wandelt presented three decision criteria to check for a possible role assertion splittability in the description logic $\mathcal{ALC}$ that have to be checked before a split can be made. He also prooved soundness and completeness of his criteria. For more information check [Wandelt 2011]. Those three criterias are:

Decision Criteria for ABox Splits in $\mathcal{ALC}$-ontologies Given an $\mathcal{ALC}$*-ontology* $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an ABox split is valid for $\mathcal{O}$ if for each C $\in$ $info^{\forall}_{\mathcal{T}}$(R)

- C = $\bot$ or
- there exists a concept description C_2, such that $C_2(b) \in \mathcal{A}$ and $\mathcal{T} \models C_2 \sqsubseteq C$ or
- there exists a concept description C_2, such that $C_2(b) \in \mathcal{A}$ and $\mathcal{T} \models C \sqcap C_2 \sqsubseteq \bot$.

3.2 Modularization in $\mathcal{SHI}$

Being able to use ABox modularization in $\mathcal{SHI}$ requires an extended algorithm. Structures for role hierarchies, inverse roles and transitiv roles have to be implemented. First, we start with $\mathcal{ALCH}$ and show how to add role hierarchies to $\mathcal{ALC}$.

According to Wandelt, we will introduce therefore an extended version of the $\forall$-info structure, that is able to handle role subsumptions, because propagations of concept descriptions can now occur over all super roles.

Extendend $\forall$-info structure Given a TBox $\mathcal{T}$ in normal form and a RBox $\mathcal{R}$, an extended $\forall$-info structure for $\mathcal{T}$ and $\mathcal{R}$ is a function $extinfo^{\forall}_{\mathcal{TR}}$: **Rol** $\rightarrow$ **Con**, such that we have C $\in$ $extinfo^{\forall}_{\mathcal{TR}}$(R) if and only if there exists a role $R_2 \in$ **Rol**, such that $\mathcal{O} \models R \sqsubseteq R_2$ and $\forall R_2.C \in clos(\mathcal{T})$.

Having the new extended $\forall$-info structure we can now also check which concept descriptions are additionally propagated via super roles. That extends also our definition of the splittability criteria (see Definition 3.1), which has to be used in $\mathcal{ALCH}$ with the extended structure.

As we try to lift up from $\mathcal{ALCH}$ to $\mathcal{ALCHI}$, we have to consider inverse roles. That means, now concept descriptions are not propagated in a single direction. They also can be propagated in the inverse direction. Thus, each decision criterion for splittability must be satisfied for each role assertion neighbour.

Role Assertion Neighbor in $\mathcal{ALCHI}$ Given an $\mathcal{ALCHI}$-ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, two individuals $a_1 \in$ Ind(A) and $a_2 \in$ Ind(A), and a role description R $\in$ **Rol**, a_2 is an R-neighbor of a_1 if and only if

- there exists a role description R_2, such that $\mathcal{O} \models R_2 \sqsubseteq R, and R_2(a1, a2) \in A$ or
- there exists a role description R_2, such that $\mathcal{O} \models R_2 \sqsubseteq R^-, and R_2(a2, a1) \in A$.

Therefore, the just extended decision criteria for splittabiliy in $\mathcal{ALCH}$ have to be checked twice in $\mathcal{ALCHI}$ once for role assertions in normal direction and once for the inverse direction (R^-).

To get one final step further from $\mathcal{ALCHI}$ to $\mathcal{SHI}$, we have to consider transitive roles. As transitive roles and especially transitive super roles can be propagated, we also have to remind this issue in the splittability criteria. For precise information on proofs and the tableau rule application in this context we refer once again to [Wandelt 2011].

To sum this part up, for extending the splittability criteria from $\mathcal{ALC}$ (see 3.1) to $\mathcal{SHI}$ we have proposed three additional steps. First, the extended $\forall$-info structure for hierarchical roles (see 3.2). Second, the check in both directions for inverse roles. And third, a check for transitive roles.

The final decision criteria for the $\mathcal{SHI}$-splittability is defined as follows:

$\mathcal{SHI}$-splittability of Role Assertions Given a $\mathcal{SHI}$-ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and a role assertion R(a,b), we say that R(a,b) is $\mathcal{SHI}$-splittable with respect to $\mathcal{O}$ if

1. there exists no transitive role R_2 with respect to $\mathcal{O}$, such that $\mathcal{O} \models R \sqsubseteq R_2$
2. for each C $in\ extinfo^{\forall}_{\mathcal{TR}}$(R)
   - C = $\perp$ or
   - there exists a concept description C_2, such that $C_2(b) \in \mathcal{A}$ and $\mathcal{T} \models C_2 \sqsubseteq C$ or
   - there exists a concept description C_2, such that $C_2(b) \in \mathcal{A}$ and $\mathcal{T} \models C \sqcap C_2 \sqsubseteq \perp$.
3. for each C $in\ extinfo^{\forall}_{\mathcal{TR}}(R^-)$

- C = $\bot$ or
- there exists a concept description C_2, such that $C_2(a) \in \mathcal{A}$ and $\mathcal{T} \models C_2 \sqsubseteq C$ or
- there exists a concept description C_2, such that $C_2(a) \in \mathcal{A}$ and $\mathcal{T} \models C \sqcap C_2 \sqsubseteq \bot$.

3.3 Concluding Modularization Steps

In this chapter we have shown how ABoxes can be split to create unique and small modules. First, we described graph-component-based modularization. To improve this, we examined the splittability criteria by Sebastian Wandelt to create ABox splits and finally defined all necessary criteria to ensure soundness and completeness of reasoning in $\mathcal{SHI}$.

The modularization process using the SHI-splittability is demonstrated in a final example.

For analyzing the concrete splittability criteria, we consider the example ontology from 2.2.3. The resulting extended $\forall$-infostructure (see 3.2) reads as follows:

$$extinfo^{\forall}_{\mathcal{T}_{Ex},\mathcal{R}_{Ex}}(R) \begin{cases} \{Course, \neg Course\} & \text{if R} = \textit{takes}, \\ \{\bot\} & \text{if R} = \textit{memberOf}, \\ \{Course\} & \text{if R} = isTaughtBy^{-}, \\ \{Course\} & \text{if R} = \textit{teaches}, \\ \{\neg \textit{Department}, \bot\} & \text{if R} = \textit{teaches}, \\ \emptyset & \text{otherwise} \end{cases}$$

Given the structure, we can decide on the splittabity of each role assertion. For example the *memberOf* role memberOf(eve, cs) is splittable as $extinfo^{\forall}_{\mathcal{T}_{Ex},\mathcal{R}_{Ex}}(memberOf) = \bot$ and $extinfo^{\forall}_{\mathcal{T}_{Ex},\mathcal{R}_{Ex}}(memberOf^{-}) = \emptyset$. But all role assertions of the kind *subOrgOf* are not $\mathcal{SHI}$-splittable as they are transitive roles, and this is against the first criterion of the $\mathcal{SHI}$-splittability check (see 3.2).

4 Using Modules

We have introduced techniques to modularize the assertional part of an Ontology in the last chapter. Our goal is now to show how to use these modularized ABoxes for efficient reasoning. We will basically present the ideas of Sebastian Wandelt, who proposed his individual islands and a new structure called *one-step nodes* for this purpose.

In Section 4.1 we will formally define a subset of ABox assertions, called *individual island*. Which allows us to define an optimized strategy to perform instance checking for individuals on each *individual island* seperately.

In Section 4.2 we discuss the similarity of islands to introduce a new data structure called *one step node*. The criteria of similarity allows us to perform reasoning on *one step nodes* instead of complete *indiviual islands* and answer querys fast.

Finally, Section 4.3 shows how we use *one step nodes* an islands for individual query answering as well as instance checking and retrieving.

This chapter is concluded with Section 4.4.

4.1 Individual Islands

In this Section, we will show an optimized way to perform instance checking for a given named individual proposed by Sebastian Wandelt in [Wandelt 2011].

Our goal is to formally identify a subset of assertions that we call *individual island* and is sufficient to perform sound and complete instance checking for a given individual. In Chapter 3 we defined criteria that allow us to split up role assertions, while retaining soundness and completeness of instance checking algorithms. Based on these preliminaries, we can formally define an individual island candidate.

Individual Island Candidate Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and a named individual a ∈ Ind(A), an *individual island candidate*, is a tuple $ISL_a = \langle \mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}, a \rangle$ such that $A^{isl} \subseteq A$.

Individual Island Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and an *individual island candidate* $ISL_a = \langle \mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}, a \rangle$, ISL_a is called *individual island* for $\mathcal{O}$ if ISL_a is sound and complete for reasoning in $\mathcal{O}$.

Informally spoken, an individual island candidate becomes an individual island if it can be used for sound and complete reasoning. Soundness is easy to see for an island candidate, since it contains a subset of the original ABox. For more on soundness and an explanation on completeness we refer to [Wandelt 2011].

We will now focus on the generation of individual islands. In Figure 4.1 we define an algorithm for the computation of islands. The algorithm gets as input an ontology $\mathcal{O}$ and a single individual **a**. At the end it returns the individual island of individual **a**. The set **agenda** includes all individuals that have to be visited next, starting with individual **a**. Where the set **seen** has all individuals that are already visited. In the process we check any role assertion between a_1 and a_2 for the defined $\mathcal{SHI}$-splittability. If the role assertion is not splittable, we will add a_2 to the agenda and continue with the role assertions of a_2 until the *agenda* is empty. Any individual we pass will be added to the individual island.

With the computation of islands we are already able to reduce the required size in the main memory as we can use our islands instead of the whole ABox. But in order to find instances for individuals it might be useful to combine individuals in groups and start retrieval over one entire group. From the computation of islands it is clear that two different individuals might belong to the same island or ABox, if both are connected by a chain of $\mathcal{SHI}$-unsplittable role assertions. However, finding individuals belonging to a similar island is not a trivial task.

Therefore, Wandelt proposed a similarity measure for graphs that are "structually equivalent" and not identical, but entail the same set of atomic concept descriptions for the root node. His similarity measure is based on the definition of homomorphism over labeled graphs. He defines the three homorphism criteria for individual island graphs as follows:

Individual Island Graph Homomorphism Given an individual island graph, a homomorphism is mapped to this graph and called individual island graph exactly if:

- The concept of the root is equal to the mapped root.
- The set of all concept descriptions is equal for all nodes and their mapped nodes
- All nodes have exactly the same concepts of successors nodes as their mapped nodes.

```
Input: Ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, individual a $\in$ Ind(A)
Output: Individual Island $ISL_a = \langle \mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}, a \rangle$
Algorithm:
Let agenda $= \emptyset$
Add $a$ to agenda
Let seen $= \emptyset$
Let $\mathcal{A}^{isl} = \emptyset$
While agenda $\neq \emptyset$ do
    Remove $a_1$ from agenda
    Add $a_1$ to seen
    Let $\mathcal{A}^{isl} = \mathcal{A}^{isl} \cup \{C(a_1) | C(a_1) \in \mathcal{A}\}$
    For each R($a_1$,$a_2$) $\in \mathcal{A}$
        $\mathcal{A}^{isl} = \mathcal{A}^{isl} \cup \{R(a1, a2) \in \mathcal{A}\}$
        If R($a_1$,$a_2$) $\in \mathcal{A}$ is $\mathcal{SHI}$-splittable with respect to $\mathcal{O}$ then
            $\mathcal{A}^{isl} = \mathcal{A}^{isl} \cup \{C(a_2) | C(a_2) \in \mathcal{A}\}$
        else agenda = agenda $\cup$ ($\{a2\}$ without seen)
    For each R($a_2$,$a_1$) $\in \mathcal{A}$
        $\mathcal{A}^{isl} = \mathcal{A}^{isl} \cup \{R(a2, a1) \in \mathcal{A}\}$
        If R($a_2$,$a_1$) $\in \mathcal{A}$ is $\mathcal{SHI}$-splittable with respect to $\mathcal{O}$ then
            $\mathcal{A}^{isl} = \mathcal{A}^{isl} \cup \{C(a_2) | C(a_2) \in \mathcal{A}\}$
        else agenda = agenda $\cup$ ($\{a2\}$ without seen)
```

Figure 4.1: Algorithm for computing an individual island [Wandelt 2011]

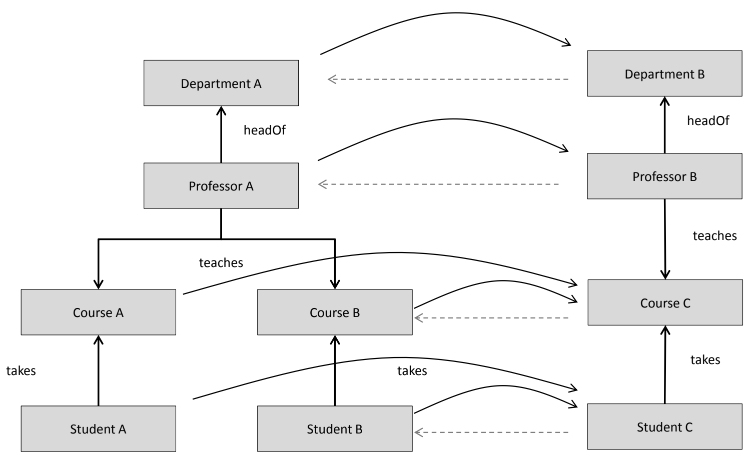


Figure 4.2: Example of two similar graphs

Example Homomorphism To show the homomorphism in an example, we consider once again our graph from Section 2.1. We say that the two graphs in Figure 4.2 are similar. We can see that a homomorphism exists by definition from the left to the right graph and vice versa, indicated by the dashed lines.

For a detailed formal description about similarity measures and entailment for individual island graphs we refer to [Wandelt 2011].

In this section we have shown sets of individuals as an individual island representation of ABoxes. Our individual island approach can be seen as an generalization of this approach. We have shown a formal foundation on why sets of individuals are similar and how it can be used during reasoning. However, the problem of deciding whether two graphs are similar is a hard problem. If the decision becomes too complex, the similarity measure does not have any performace gain. We will show in the next section how to further overcome these performance issues.

4.2 One Step Nodes

We discuss here a new data structure that allows us to quickly decide, whether two individual islands are similar or not. This structure can be used to detect several dissimilar islands immediately. Only if this fails, we apply further techniques to find a homomorphism.

The idea in general is to combine information on the original individual of the island with information about the so-called *pseudo node neighbors*, which represent the directly asserted successors of the root, to obtain the *one-step nodes*. Those nodes can be used in addition to answer instance checks directly without any need to use the specific island which is stored on an external server. Each one-step node stands for one group of islands that are similar, which allows us to check the simplyfied one-sted node instead of the complete island.

At first, we define a *pseudo node successor* with respect to an ABox.

Pseudo Node Successor Given an ABox $\mathcal{A}$, a *pseudo node successor* of an individual a $\in NInd(\mathcal{A})$ is a pair $pns^{a,A} = \langle \mathbf{rs}, \mathbf{cs} \rangle$, such that $\exists a_2 \in Ind(\mathcal{A})$ with

1. $\forall R \in \mathbf{rs}.(R(a, a_2) \in \mathcal{A} \vee R^-(a_2, a) \in \mathcal{A})$,
2. $\forall C \in \mathbf{cs}.C(a_2) \in \mathcal{A}$, and
3. $\mathbf{rs}$ and $\mathbf{cs}$ are maximal.

The third criterion is especially important as it says that for each pair of named individuals $\langle a, a_2 \rangle$ the node a_2 is exactly one *pseudo node successor* for individual a.

Example for Pseudo Node Successors For the *pseudo node successors* we consider our example ontology from section 2.2.3. By applying the *pns* criteria, we find the pseudo node successors as:

- $pns^{ann,A_{Ex}} = \langle \{headOf, Department\} \rangle$
- $pns^{ann,A_{Ex}} = \langle \{teaches, GraduateCourse\} \rangle$
- $pns^{sue,A_{Ex}} = \langle \{takes, GraduateCourse\} \rangle$

In the next step we will combine the set of pseudo node successors of a in ABox $\mathcal{A}$, the reflexive role assertions for a, and the directly asserted concepts of a, in order to create a summarization object, called *one-step node*. The *one-step node* structure can be defined as follows.

One-Step Node Given an ontology $\mathcal{O} = \langle T, R, A \rangle$ and an individual $a \in NInd(\mathcal{A})$, the *one-step node* of a for $\mathcal{A}$, denoted $osn^{a,\mathcal{A}}$, is a tuple $osn^{a,A} = \langle \textbf{rootconset}, \textbf{reflset}, \textbf{pnsset} \rangle$, such that $\textbf{rootconset} = \{C | C(a) \in \mathcal{A}\}$, $\textbf{reflset} = \{R | R(a,a) \in \mathcal{A} \vee R^{-}(a,a) \in \mathcal{A}\}$, and **pnsset** is the set of all *pseudonode successors* of individual a. The set of all possible *one-step nodes* is donated **OSN**.

Example for One-Step Nodes With the definition of *one-step nodes* we are now able to build the *osn* for *Professor A* in the previous homomorphism example (see 4.1). The corresponding *one-step node* lists as follows:

$$\begin{aligned} osn^{ProfessorA,A} &= \langle \{Professor\}, \emptyset, \{\langle \{headOf\}, \{Department\} \rangle, \langle \{teaches\}, \\ &\quad \{Course\} \rangle \} \rangle \end{aligned}$$

As *one-step nodes* are nothing else than a summarization object for individual islands, it is clear that not every *one-step node* is complete for instance checking. However, Wandelt shows in his work that they are actually complete in the case that a *one-step node* coincides with the individual island. For this case, he defines so-called *splittable one-step nodes*, for which each role assertion to a direct neighbor is $\mathcal{SHI}$-splittable.

Splittable One-Step Node Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an individual a $\in NInd(\mathcal{A})$ and a one-step node $osn^{a,A} = \langle \textbf{rootconset}, \textbf{reflset}, \textbf{pnsset} \rangle$, we say that $osn^{a,A}$ is **splittable** if for each $\langle \textbf{rs}, \textbf{cs} \rangle \in \textbf{pnsset}$, a fresh individual $a_2 \notin Ind(\mathcal{A})$ and for each $R \in \textbf{rs}$, the role assertion axiom $R(a, a_2)$ is $\mathcal{SHI}$-splittable with respect to Ontology $\mathcal{O}_{\in} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A}_{\in} \rangle$ with $\mathcal{A}_{\in} = \{C(a) | c \in \textbf{rootconset}\} \cup \{C(a_2) | C \in \textbf{cs}\} \cup \{R(a, a_2)\}$.

In this section we have defined a structure for improved integrity checks for individual island similarity. The one-step node allows to use instance on similar islands without the need to load the island itself in the local memory. Only if that check fails, we refer to an instance check on the specific individual island. In the following we will further explain how to use these techniques for optimized reasoning.

4.3 Optimized Reasoning

In the following we show instance checking and instance retrieval (see section 2.2.4) in several examples and how to use them in an optimized way with the techniques we have just introduced.

4.3.1 Instance Checking

Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an atomic description C and an individual $a \in NInd(\mathcal{A})$, we want to check for $\mathcal{O} \models C(a)$. The process of instance checking can be done in two steps. First, we just check the one-step node $osn^{a,A}$ of individual a for $osn^{a,A} \models C(a)$. If this is true, we are done. Since we know that one-step nodes are sound for instance checking with respect to $\mathcal{O}$ (compare [Wandelt 2011]). But if we find $osn^{a,A} \models C(a)$ not to be true, we have to distinguish two cases. At first, if $osn^{a,A}$ is splittable (see 4.2), we know that it already coincides with the original island and we actually have $osn^{a,A} \not\models C(a)$. But otherwise, if $osn^{a,A}$ is not splittable, then we have to load the individual island ISL_a for a and perform instance checking over the original island.

In the following we have an example for instance checking on one-step nodes. We check, whether the individual *ann* is an instance of concept description *chair*. Let the one-step node $osn^{ann,A}$ be defined as follows:

$$\begin{aligned} osn^{ann,A} \quad = \quad & \langle \{Professor\}, \emptyset, \{\langle \{headOf\}, \{Department\} \rangle, \langle \{teaches\}, \\ & \{UnderGraduateCourse\} \rangle \} \rangle \end{aligned}$$

Then a possible realisation of $osn^{ann,A}$ is

$$\begin{aligned} ABox(osn^{ann,A}) \quad = \quad & \{Professor(ann), headOf(ann, a_1), teaches(ann, a_2), \\ & Department(a_1), UnderGraduateCourse(a_2)\}. \end{aligned}$$

We can directly see from the notion that we have $ABox(osn^{ann,A}) \models Chair(ann)$. Thus, by soundness of one-step node reasoning $\mathcal{O} \models Chair(ann)$.

In a second example we want to check for $c1 \models Chair$ with respect to $\mathcal{O}$. The one-step node $osn^{c1,A}$ looks like the following:

$$\begin{aligned} osn^{c1,A} \quad = \quad & \langle \{UnderGraduateCourse\}, \emptyset, \{\langle \{teaches^{-}\}, \{Professor\}\rangle, \\ & \langle \{takes^{-}\}, \{Student\}\rangle\}\rangle \end{aligned}$$

A possible realization of the one-step node would be

$$\begin{aligned} ABox(osn^{c1,A}) \quad = \quad & \{UnderGraduateCourse(c1), teaches(a_1, c1), takes(a_2, c1), \\ & Student(a_2), Professor(a_1)\}. \end{aligned}$$

In this case it is easy to see that we have $ABox(osn^{c1,A}) \not\models Chair(c1)$. Thus, the one-step node does not indicate entailment, we finally have to refer to the individual island for checking soundness and completeness as $osn^{c1,A}$ is not splittable.

4.3.2 Instance Retrieval

Instance retrieval optimization is a direct extension of instance checking optimization, such that we use one-step node similarity in addition. The first idea to solve this problem is that you just apply instance checking on every individual in your ABox. However, as there exist a one-step node similarity for similar islands, we say that similar one-step nodes entail the same set of concept descriptions for named root individual. Thus, given all one-step nodes, we can reduce the number of instance checks.

We continue with an example of instance retrieval for the concept description *Chair* with respect to ontology $\mathcal{O}$. At first, we retrieve the set of all one-step nodes and for each individual in $\mathcal{A}$. The concluding one-step nodes are as follows:

$$\begin{aligned} osn^{ann,A} \quad = \quad & osn^{mae,A} = \langle Professor, \emptyset, \{\langle \{headOf\}, \{Department\}\rangle, \\ & \langle \{teaches\}, \{UnderGraduateCourse\}\rangle\}\rangle \end{aligned}$$

$$\begin{aligned} osn^{c1,A} \quad = \quad & osn^{c2,A} = osn^{c3,A} = \langle \{GraduateCourse\}, \emptyset, \\ & \{\langle \{teaches^{-}\}, \{Professor\}\rangle, \langle \{takes^{-}\}, \{Student\}\rangle\}\rangle \end{aligned}$$

$$osn^{ee,A} = \langle\{Department\}, \emptyset, \{\langle\{headOf^{-}\}, \{Professor\}\rangle\}\rangle$$

We are left with three instance checks as we have only three one-step nodes instead of the six instance checks for each individual. For big ABoxes it can be shown that instance checks are usually reduced by orders of magnitude.

We can directly see by instance check that *ann* and *mae* are instances of concept *Chair*. Another instance check for ¬Chair shows that *c1*, *c2* and *c3* are individuals of this concept description, if the input ontology is consistent. Thus, only *ee* remains. Usually we would have to use an instance check over the individual island, but as we know that $osn^{c1,A} \not\models chair$ and that $osn^{c1,A}$ and $osn^{ee,A}$ are indeed splittable, *c1* and *ee* can not be an instance of *chair* and we are done.

4.4 Concluding Remarks

In Chapter 3 basic ABox modularizations were defined. With our additional structures in this chapter we are able to perform instance checking in an optimized way for a given individual and atomic concept description. We have also discussed optimizations of instance retrieval. Therefore, we have introduced a similarity measure, in order to reduce the number of necessary instance checks to perform instance retrieval.

To determine the island similarity, we have introduced a structure called *one-step nodes*. These are used as a kind of proxy to answer queries faster.

5 Using Conjunctive Queries

In this Chapter we present our ideas for using conjunctive queries efficiently on small ABox modules stored in an AllegroGraph triplestore. We will first propose a theoretical solution for grounded conjunctive queries that we have introduced in Section 2.2.5. Furthermore, we propose the new idea of skeleton queries, as we believe these kind of structures are able to improve the performance of query answering for grounded conjunctive queries with a huge benefit. Finally we will show how we are able to easily rebind the use of SPARQL from queries on graph pattern to skeleton queries for grounded conjunctive queries.

5.1 Grounded Conjunctive Query

For solving grounded conjunctive queries we can directly think of an naive approach, which is based on instance checks over all individuals and combining these with checks on each role assertion and for each individual in the query. This approach is not efficient for huge ontologies and a lot of individuals. We will now shortly describe this approach and show how we can improve it with skeleton queries.

A naive tableau-based algorithm for grounded conjunctive queries is split up first into single components of concept assertion atoms and relation assertion atoms, to solve each of these atomic elements for itself and join all elements afterwards. Consider a conjunctive query for example, we take $A(x) \cap R(x,y) \cap B(y)$ where x and y are distinguished variables, A and B are concepts and R is a role. Naively we would now solve $A(x)$, $B(y)$ and $R(x,y)$ as three seperate queries and then join all result bindings. But without any further optimization, we have to check and test every pair of individuals in the ABox for solving each membership atom. For the membership atom $A(x)$ we can naively realize an instance check for each individual a by adding the assertion a : ¬C to the ABox. The new ABox is then checked for consistency. For the relationship atom $R(x,y)$, for each individual pair a and b, we can add $a : \neg\exists R N_b$ and $b : N_b$, to the ABox, where N_b stands for a new concept. The ABox is then tested once again for consistency. Although there are optimizations that reduce the number of tests that need to be performed, this approach remains fundamentally impractical for large ABoxes. With the combination of an AllegroGraph triplestore and a modulebased ABox strategy we will try to achieve much better results.

5.2 Skeleton Query

For reducing individual checks we propose the usage of skeleton queries. Skeleton queries are commonly used to suppress unwanted results in database queries. We split up the conjunctive querying in two general steps. First, we want to reduce the number of individuals as much as possible. Therefore, we use a combination of all relationship axioms in a set query.

For simplicity, we can write a skeleton query as a set instead of as a conjunction of atoms. If we stick to our previous example, we can write the introductory example for conjunctive queries from Section 2.2.5 only consisting of roles.

$$\{hasSon(x,y), hasDaughter(y,z), hasDescendant(x,z)\}$$

Additionally to the roles, we want to save any resulting set binding to variables for query answering. Those variables are given in the head of the query.

$$(x_1,x_2,x_3) \leftarrow \{hasSon(x_1,x_2), hasDaughter(x_2,x_3), hasDescendant(x_1,x_3)\}$$

Query answers are tuples (a_1,a_2,a_3) of atomic individual names, that substitute the variables of (x_1,x_2,x_3).

As a result of the skeleton query, in which we can check for all role assertions at once, we received a number of tuples. These tuples consist of a reduced number of individuals. Thus, in a second step we only have to use instance checking per tuple.

The AllegroGraph triplestore (see 2.4) allows to build simple structures of *skeleton queries* by using the implemented query language SPARQL (see 2.4). With SPARQL we can directly build all necessary query constructions by combining distinguished variables within a set of role assertions. The resulting query binding can be used as a tuple in a second step and specific instance checks for concept assertions on individual names.

We basically use all role assertions and ignore information about the concept assertions in the first step. AllegroGraph automatically responds in a binding set that solves the used variables for each role assertion.

As example for a skeleton query we can direcly refer the example from above and ask “who is a grandmother who has a grandchild that is a student”. The skeleton query in SPARQL is:

Listing 5.1: Example of a skeleton query

```
SELECT ?x ?z
WHERE {
  ?x hasChild ?y.
  ?y hasChild ?z.
  ?x hasDescendant ?z
}
```

We only ask for grandparents and there grandchildren here as we only use role assertions. The skeleton query reduces the amount of individuals to the group of individuals who are grandparents and grandchildren. Thus, in a second step we check the concept assertion of *GrandMother* on x and the concept assertion of *Student* on far less individuals.

For instance tests we can rely on the reduced number of individuals in our small modularization structures, shown in the last chapter. Therefore, we check first on similar one step nodes. If the specific one step node does not model our concept assertion, this is true for the case that we have chosen an splittable one step node or have to make a second instance check on the corresponding island (see 4.3.1).

With the combination of reduced tuples and reduced ABox sizes we are able to show an implementation of a new query system for grounded conjunctive queries.

6 Implementation and Evaluation

In this chapter we want to show two things. First, we are able to use grounded conjunctive queries on small individual island in a fast and efficient way with our new idea of skeleton queries. Second, we don't need to store our ABox in seperate files before the preprocessing. We can, just as the usual workflow would be, make use of the existing ontology on the triplestore and generate individual islands as well as one step nodes directly on the fly by using simple SPARQL queries.

To demonstrate these issues we will first explain our implemented prototype and give detailed evaluation results afterwards.

6.1 Implementation

In this section we will describe an implementation of a prototype consisting of two parts. First, the introduced techniques and strategies of Chapter 4 are implemented as a preprocessing step to create the necessary island and one-step node structures. Additionally we explain internal data structures and show a server and client architecture that is capable of perfoming efficient query answering for grounded conjunctive queries by relying on instance checks as well as instance retrieval and describe how query answering is performed.

We will now describe the architecture of the client and server system. Basically, the implemented system consists of two parts, a number of clients and a server system, which is represented by the AllegroGraph triplestore (see 2.4). The servermodule is completely used for storage of the relatively large assertional part of the database in the external memory, the ABox. But due to the limited reasoning abilities of AllegroGraph servers, the reasoning over data and the storage of the terminological part has to take place on the client system. Therefore, we have to transfer any needed assertional data to the client for reasoning. At the beginning, we start with the complete ontology on the AllegroGraph store in one piece. During the preprocessing process we will split up the ontology in more and more small modules directly on the fly.

Although we can easily load assertional data into the AllegroGraph triplestore via web interface, every input to the client itself via file or query systen has to be

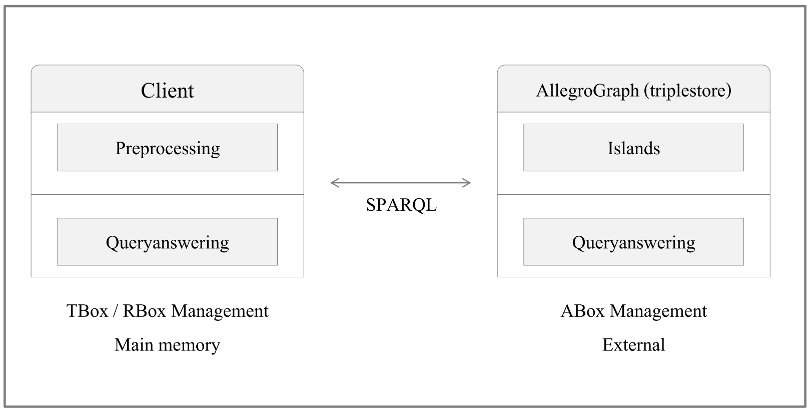


Figure 6.1: Basic Client- and Serverarchitecture

parsed and organized by a special Java Api. For this case we use the OWLAPI (see [Horridge and Bechhofer 2009]), which is able to create and manipulate various OWL Ontologies. It is available as an open source version under LGPL or Apache.

The communication between client and server consists basically of SPARQL queries for query answering (see 2.4) and AllegroGraph statements for adding additional triples directly to the ABox stored on the AllegroGraph server. Adding statements on the triplestore is necessary as we save each island as a subgraph directly on the AllegroGraph server. SPARQL queries are then used for the query process in the second step (see 6.1).

6.1.1 The Preprocessing Step

In a first step the whole data system has to be set up before we can start query answering. The necessary steps are described in Chapter 3 and 4. The client determines the extended ∀-info structure (see: 3.2) to generate splittability criteria (see: 3.2) for the description logic $\mathcal{SHI}$.

Islands Based on these criteria, islands are built by the client on the fly via the algorithm from Chapter 4 (see: 4.1). After creating islands all new generated triples are added to the triplestore as an islandstorage. For simplicity we use for each island a seperate subgraph. This is straightforward as we just add another value to the triple for creating a subgraph or link the triple to an already existing one.

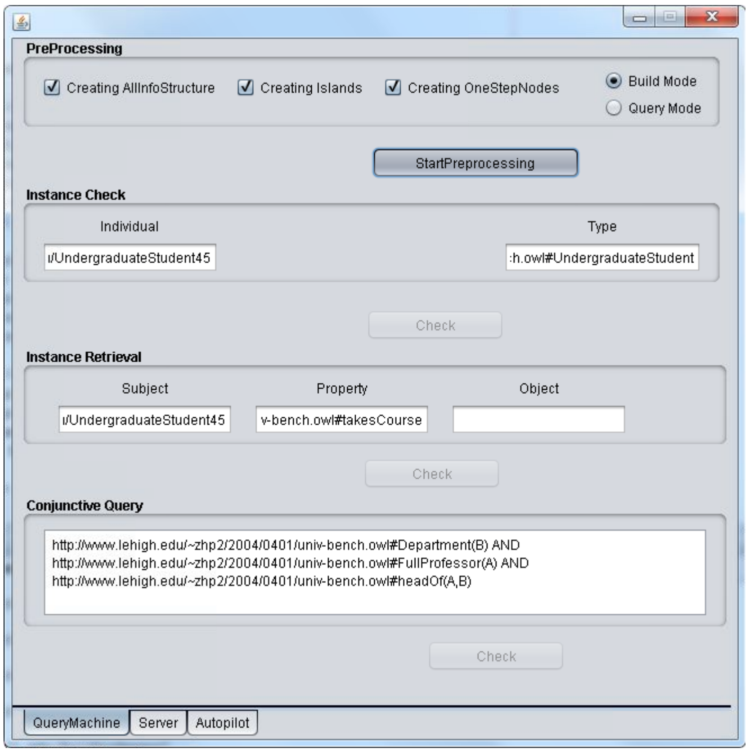
One-Step Node Building homomorphisms is solved by the usage of hashvalues with a sufficient bitlength. We first compute a set of root, pseudo node successors and reflexive role assertions from each island to build the specific one-step nodes. For each node, we immediately build a hashvalue and store it together with the specific island on the triplestore. Identical hash values refer to identical one-step nodes or corresponding similar individual islands. Thus, we only need to store a single one-step node for each hashvalue and find homomorphisms automatically. As the one-step nodes are each very small, we can keep those cached on the client system. However, the used hash algorithm has to be appropriate. Thus, we do not want any collisions, according to the number of individuals the bitlength of the hash key must be chosen. For the evaluation purpose in this thesis a MD5 hash value was found as sufficient.

6.1.2 Query Answering

The query process was implemented as described in 2.2.5. Conjunctive queries consist of a conjunctive combination of concepts and role assertions, which are bound to variables. To eliminate as many candidates for concept variables as possible in a preprocessing step, we make use of skeleton queries (see 5.2). Therefore, the client computes a SPARQL query consisting of all role assertions and bound variables. In a second step we only have to use instance checks on the received tuples. Given the concept description C from the query and the named individual a from the tuple, we load the specific one-step node for a from the cache and determine whether osn_a entails $C(a)$. Depending on the outcome, three states are possible:

- Osn_a entails $C(a)$, then a is actually a concept description of C.
- Osn_a entails $\neg C(a)$ or (does not entail $C(a)$ and is splittable), then a is actually not a concept description of C.
- Osn_a is not splittable, then the client has to load and check the entire island according to a to find out whether a actually is a concept description of C.

By eliminating all tuples that include individuals, which do not belong to concept assertions used in the query, finally all remaining tuples are correct answers to the original conjunctive query.



PreProcessing: [✓ Creating AllInfoStructure] [✓ Creating Islands] [✓ Creating OneStepNodes] [✓ Build Mode] Query Mode

Instance Check — Individual: i/UndergraduateStudent45 — Type: :h.owl#UndergraduateStudent

Instance Retrieval — Subject: i/UndergraduateStudent45 — Property: v-bench.owl#takesCourse — Object:

Conjunctive Query:

```
http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Department(B) AND
http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#FullProfessor(A) AND
http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#headOf(A,B)
```

Figure 6.2: User interface of a client node in preprocessing mode

6.1.3 GUI

The client comes along with an interface, that allows direct input for all necessary user data. As well as the implementation, the user interface itself is also split into two parts. The user is able to control the whole preprocessing process in the top panel. He is able to generate islands and one-step nodes or one-step nodes only, if islands do already exist. All one-step nodes are saved in a XML file during the process. Thus, the creating step can also be skipped and loaded directly from file for the next use of the database.

The second part of the user interface is about query answering. There is one panel for instance checks, one panel for instance retrieval and one panel for conjunctive queries. All panels can be used manually with a specific simplified syntax.

Additionally, there is a special tab available for automatic query answering. Queries can be saved before program start in a XML library (in our example we use the 14 official LUBM test queries) and chosen to be automatically send to a predefined number of servers and repositories. All query results together with runtimes and more information are saved in another XML file. This service called *Autopilot* was implemented to simplify most of the test cases (see 6.2).

During the past section we presented the implementation of a query answering client architecture, that uses an AllegroGraph triplestore as assertional database. The client makes use of the explained optimizations for memory management and conjunctive queries, we will evaluate all queries and tests in the following section.

6.2 The Evaluation

In this section we will present the evaluation result of the query answering prototype.

The used testset in this thesis is based on the *Lehigh University Benchmark* or LUBM. This benchmark is an ontology system designed to test large knowledge bases with respect to OWL applications. With the Description Logic of $\mathcal{SHI}$ we are able to express all of the concepts used in the lubm benchmark. The data can be generated automatically by a small Java tool the *LUBMgenerator*. Its terminological part is rather small, it consists of 43 classes and 32 properties on that basis the TBox is fixed, but the assertional part is very scalable and dynamic in size. Thus, it can easily be generated in sizes necessary for the user. The whole setting in this database is about the organisation of a university. A university consists of departments. Several professors are members of each department. Professors teach courses. Students take courses and so on. With help of the LUBMgenerator, the user can, given a number n, generate n Universities each consisting of a random number of departments and individuals.

As the number of individuals and the number of assertions increases nearly linear with the number of universities, LUBM is a perfect instrument to test the performance for query answering machines, especially for grounded conjunctive queries in a growing ABox environment.

Sebastian Wandelt has already investigated the efficiency of ABox modularization techniques for an SQL database server. He has shown modules with respect to $\mathcal{SHI}$-splittability and instance checks performed on them. In his method he used prepared OWL files for loading the ontology into the client, work on each file seperately and then load each block of modules iteratively onto the SQL server. In the following evaluation we will show that we are able to do the whole process without filepreparation. We work directly on the server, convert the large ontology step by step into small chunks and compare the generated modules on AllegroGraph store with the

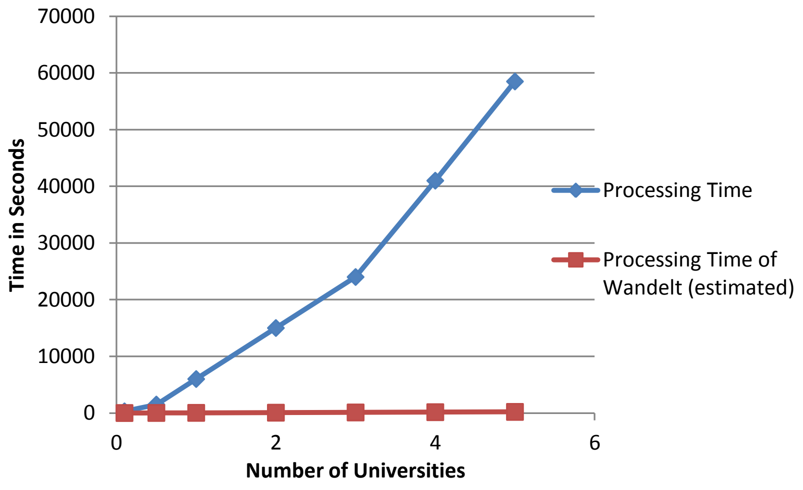


Figure 6.3: Preprocessing time for LUBM

modules of Sebastian Wandelt on the SQL server. For details on the results of Sebastian Wandelt see [Wandelt 2011].

6.2.1 Evaluating Modularization Techniques on AllegroGraph

In this section we will evaluate the preprocessing step, which are all the modularization steps together for creating individual islands and one-step nodes. First of all, many of the results for modularization techniques by the implementation in this thesis are identical to the results of Sebastian Wandelt modules. Nearly every role assertion in LUBM is actually splittable by $\mathcal{SHI}$-splittability. The number of unsplittable role assertions stays under one percent, despite an increasing number of universities. The number of created ABox assertion modules increases also lineary. For one university it is 18143, for two universities 40452 and for three universities 58947, which corresponds, as we expected, to the number of individuals in the assertional part.

But next, if it comes to the load time of the preprocessing process, we can see big differences in comparison to Wandelt in the time needed to build up the modularization directly on the server. The modularization steps on the AllegroGraph server takes far more time, than the modularization step in combination with the SQL-Server of Wandelt. As you can see in Figure 6.3 the processing time for one university is about 5000 seconds on AllegroGraph server, where it is like one minute on the implementation of Sebastian Wandelt on a SQL-Server (estimated from his diagrams as we have no real data). Wandelt achieves this speed as he is not modularizing the actual ontology on an external server, but an ontology saved in preconfigured OWL files on the local host and uploading these modules in groups to his SQL server.

A runtime analysis of the used implementation shows that most of the time is used on SPARQL queries to the allegrograph server. If we look at the queries used per university, it shows that this is a serious issue. From Figure 6.4 we see that, for instance, the modularization of one university takes nearly 200,000 (172,760) Queries. Even at the slow rate of the preprocessing we are still at roughly 50 SPARQL queries per second. This issue on query time has also been shown as we compared the needed time between a connection to the AllegroGraph server on an external fast multiprocessor system and as a small virtual box implemented directly on the local host. A Virtual Box is a simulated server with an simulated and seperate operation system, where we implemented a version of the AllegroGraph triple store. Although these simulated virtual servers have a slow performance in general, the virtual box is up to three or four times faster compared to the external multiprocessor system (see Figure 6.5).

Therefore, we see the decrease in performance is based on the increase of many SPARQL mini queries between server and client. What is exactly as expected, because Wandelt uses a single query, where we need thousands of SPARQL queries to be able to solve the modularization step on the fly. Nevertheless, this is sufficient. To show that a direct implementation on existing ontologies is possible without any need of preconfiguration in small files on the client was one of our main goals

We can even think of improvements to the server and client communication. The implemented prototype uses at least one query per splittability check, so one way of solving would be the implementation of an intelligent streaming module that streams and prunes queries at the right time for reducing the number of queries. But however, the problem on a huge amount queries between server and client has not been solved yet.

Furthermore, the increase of our processing time is even not linear. It is close to linear, but increases even more with an increasing number of universities. There are additional two reasons for the non-linearity in the preprocessing time. The data structure of the AllegroGraph server can not guarantee a complete linearity. It uses optimization and indexing technologies with the increasing size of assertional data. This works only well if the system does not read data again during the update phase. But, as we use regular updates, our system works directly on the database. The second reason can be found on the client side within unsufficient caching strategies. As for module creation every role assertion has to be checked several times for splittability. For reasons of memory efficiency with the java garbage collector not

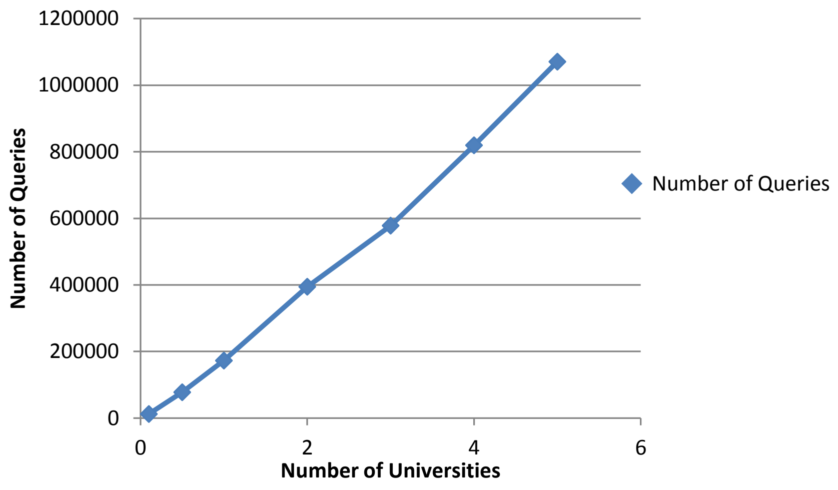


Figure 6.4: Number of SPARQL for LUBM modularization per university

every data is cached in the prototype. So an increasing number of information has to be received from the server several times with an increasing number of universities. This problem could also be solved with an intelligent streaming architecture that caches data of past splittability checks in an efficient way and does not have any need for rechecks.

6.2.2 Evaluating Conjunctive Queries

In this section we will evaluate the use of grounded conjunctive queries on the AllegroGraph triplestore. For evaluating grounded conjunctive queries, LUBM provides 14 predefined test queries, which check several criteria of the database.

Correctness of Query Answers The designer of LUBM (see 6.2) provides 14 query answers for an ontolgy of the size of a single university to the 14 test queries. By comparing our results with those from the provided answers, we can check our client and algorithm on correctness. The 14 answers of LUBM and the 14 answers of our clientsystem are identical. Therefore, we believe our algorithm on modularization and query answering works correct and the evaluation results are reliable.

High and Low Selectivity for Queries The provided queries differ in the amount of input, selectivity and reasoning behaviour for example by using role hierarchy or transitivity. Selectivity basically means that the grounded conjuctive queries we use

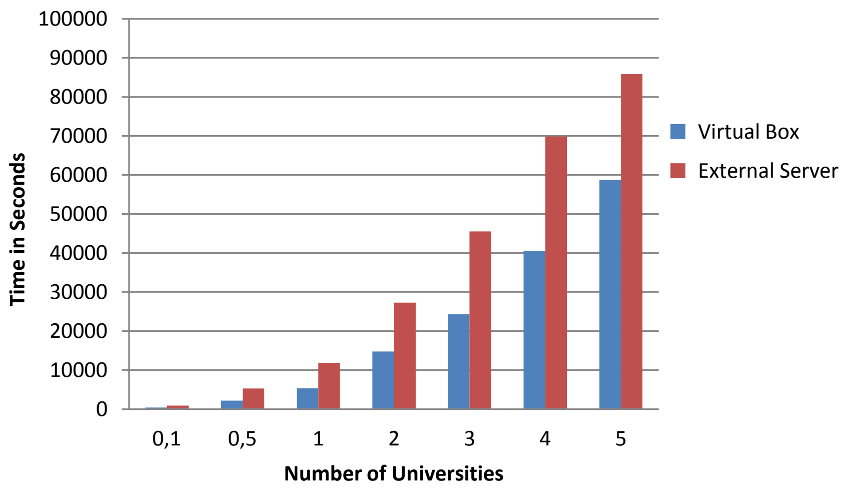
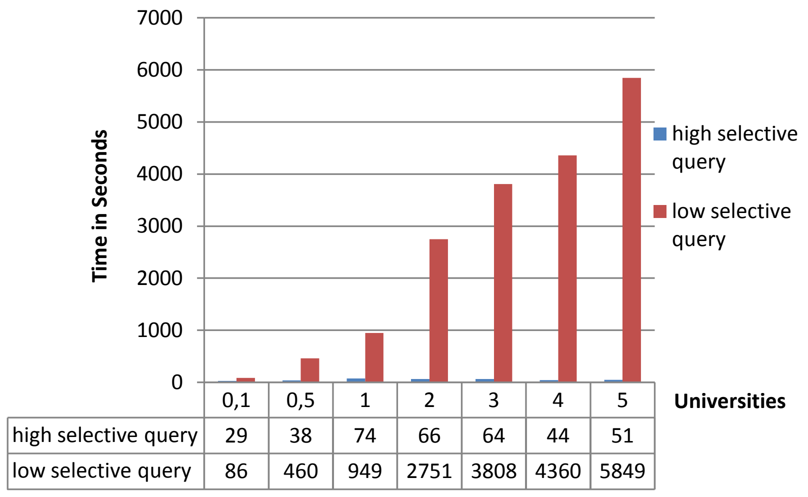


Figure 6.5: Comparison between load on an external server and a local virtual box

have roles that automatically includes many individuals, which we call barely sensitive, or automatically excludes a lot of individuals, what we call a highly selective query.

For example we use the role assertion *takesCourse*. *TakesCourse* is a connection between the concept descriptions *student* and *course*. So if we use the construct *A takesCourse B* in a conjunctive query, we already excluded any non Student for A and any non Course for B. However, as there are a lot of individual students in the ABox we still see this query as low selective. In the second step of query answering we have to consider the variables A and B. Let's say A are only *undergraduateStudents* and B *undergraduateCourses* only, so for every A and B we need additional n instance checks, where n is the cardinality of A and B, to make sure we only have *undergraduateStudents* and *undergraduateCourses*. The result is, that the more less selective our query is, the more instance checks we do need afterwards and the more time consuming those checks are. In Figure 6.6 we see a direct comparison between low selective and high selective queries. In the highly selective query we ask for all *chairs*, as there is only one chair per Department, we call this query highly selective. The figure shows the relatively high amount of consumed time for the barely selective clearly. This gap even increases with an increasing number of universities. The reason for not increasing for the highly selective is because of the fast and small number of instance checks compared to the single skeleton query, which takes most of the time in this case.



	0,1	0,5	1	2	3	4	5
high selective query	29	38	74	66	64	44	51
low selective query	86	460	949	2751	3808	4360	5849

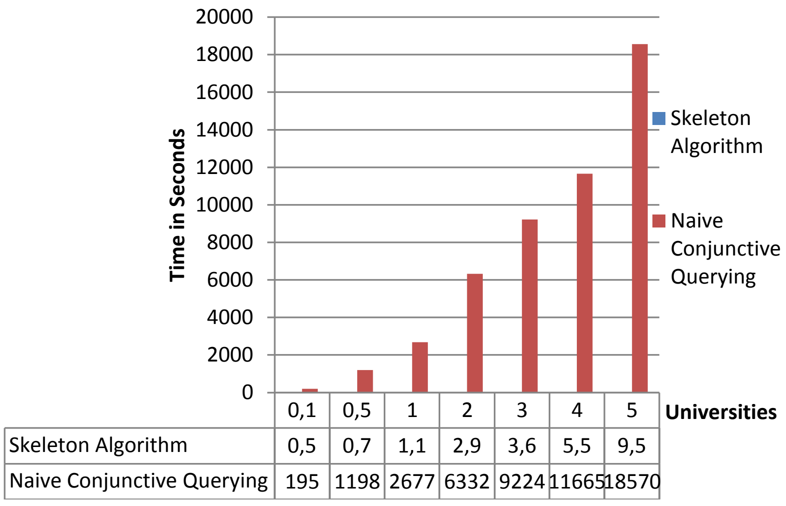
Figure 6.6: High and low selective queries compared in time

For the given reasons low selective queries always have a huge extra load on instance checks over the network. In these cases the implemented skeleton query is not able to exclude enough individuals for later instance checking. As so many extra queries are needed, the bottleneck stays in the network connection and server management of the AllegroGraph triplestore.

Skeleton Query To demonstrate that our skeleton query is able to significantly improve the results for queries with high selectivity, we compare the approach of skeleton queries with the naive approach without skeleton queries (see 5.1) in Figure 6.7. One can directly see the huge performance gain of the skeleton query. For this Example we have chosen a highly selective query, which shows really good results. We avoid a lot of instance checks and can therefore decrease the answering time by a factor of about 1000, depending on the number of individuals in the ontology.

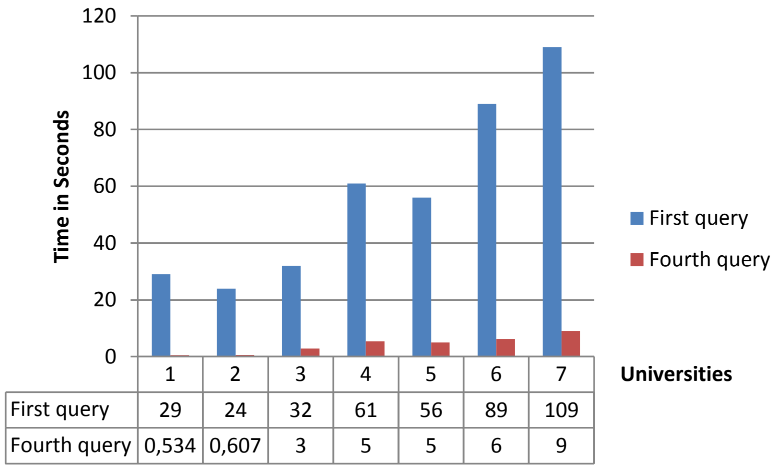
Server Caching Another aspect that the allegrograph server does is somehow the caching of query answers. You can see in Figure 6.8 how the querying process behaves after several rounds of query answering.

Here you see again how the runtime is effected by the server querytime. It shrinks to a tenth if the answer is in the cache of an allegrograph server.



	0,1	0,5	1	2	3	4	5
Skeleton Algorithm	0,5	0,7	1,1	2,9	3,6	5,5	9,5
Naive Conjunctive Querying	195	1198	2677	6332	9224	11665	18570

Figure 6.7: Comparison between skeleton query and naive approach.



	1	2	3	4	5	6	7
First query	29	24	32	61	56	89	109
Fourth query	0,534	0,607	3	5	5	6	9

Figure 6.8: Time of a single query and time after three queries

7 Conclusion

In this thesis we extended the ABox modularization strategies of Sebastian Wandelt to the efficient use of grounded conjunctive queries on AllegroGraph servers. We had two main goals. First, we showed that we don't need prepared files in which we store the ontology. We can compute all modules on the client by directly use of the AllegroGraph store. Second, we came up with an efficient way to use grounded conjunctive queries with modules, which is much faster than an naive approach would be.

Results The evaluation of the implemented prototype showed how grounded conjuctive queries on AllegroGraph servers are possible by using only a small size of main memory. The main strategy is to use a skeleton query and try to keep the necessary amount of instance checks in the second step as small as possible. But as the number of results for barely selective queries can be quite large, the number of instance check queries on AllegroGraph servers increases rapidly. The strategy makes profit of highly selective queries. In these cases performance is very good, as the number of queries to the server stays small. More queries are reducing the performance.

This mass of queries is also an issue that we have witnessed for the preprocessing step on the AllegroGraph Server. For every rolesplit and check for $\mathcal{SHI}$-splittability queries to the server are needed. Thus, if we have to check millions of statements, we also need millions of queries. This also creates a huge load on the network connection as we have to wait for each query answer. Our studies showed that even a small local virtual box server is way faster compared to an external multiprocessor and multicore machine.

Outlook These results lead to several ways of possible future improvements. The usage of query answering could be improved on the client or in a improved new system architecture. Someone could think of a streaming implementation when it comes to querying, where we stream more data at once and prune the current stream where possible. If this is used in an intelligent behaviour and the client could for instance check the splittability of more than one role assertion in one query, we would decrease the number of needed queries for the system to a huge amount.

Another possible improvement is the use of parallel working threads for instance checks or splittability criteria that are distributed on several cores of the processor. If instance checks take place on the client only, as in the case of one-step nodes, the usage of threads could be a serious issue for the large number of instance checks that are needed.

The next step from one client is to a system architecture that uses several clients or even several servers. If we share the one-step nodes on all clients we can distribute instance checks to specific clients and use only their results as a boolean function. One idea is to use all one-step nodes on all clients and a server node, that distributes all checks in a optimized way. In another version we could only distribute a specific unique set of one-step nodes to the clients to keep the used memorysize even smaller. Furthermore, we are able to reduce the networkload not only by usage of distributed client nodes, but also by several servers. The AllegroGraph server implements a feature that is able to make use of several servers, which are combined to one serversystem. Thus, the same possible strategy for one-step nodes could be used for the islands itself on AllegroGraph servers.

Apart from performance improvements we can also think of improvements in the direction of more expressive query languages. In this thesis we used grounded conjunctive queries only, but the next logical step would be towards the usage of non-grounded conjunctive queries. Grounded conjunctive queries only have solutions that exists as named individuals in the ABox. The extension to standard conjunctive queries is harder, but could be realised by an engine that automatically updates the ABox of the client and server by adding new concept descriptions.

Finally, we could think of more comprehensive studies on the present work. We believe that our results carry over to other ontologies. But we can think of ontologies, which use role assertions that are very rarely splittable and makes the modularization more difficult. Growing ABox modules can also have an impact on each instance check and thus the whole query answering.

Bibliography

S. Wandelt. *Efficient instance retrieval over semi-expressive ontologies.* Universitätsbibliothek, 2011. (document), 1.2, 3, 3.1, 3.1, 3.1, 3.2, 4.1, 4.1, 4.1, 4.1, 4.3.1, 6.2

T.B. Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific American*, 284 (5):34–43, 2001. 1.1

S.R. Kruk and B. McDaniel. *Semantic digital libraries.* Springer Verlag, 2009. 1.1

M.A. Goncalves, E.A. Fox, and L.T. Watson. Towards a digital library theory: a formal digital library ontology. *International Journal on Digital Libraries*, 8(2): 91–114, 2008. 1.1

D. Brickley and L. Miller. Foaf vocabulary specification 0.91. Technical report, Tech. rep. ILRT Bristol, Nov. 2007. ur l: http://xmlns. com/foaf/spec/20071002. html, 2000. 1.1

L. Maicher and J. Park. *Charting the topic maps research and applications landscape: First International Workshop on Topic Maps Research and Applications, TMRA 2005, Leipzig, Germany, October 6-7, 2005: revised selected papers*, volume 3873. Springer-Verlag New York Inc, 2006. 1.1

A. Doms and M. Schroeder. Gopubmed: exploring pubmed with the gene ontology. *Nucleic acids research*, 33(suppl 2):W783–W786, 2005. 1.1

R. Cornet and N. De Keizer. Forty years of snomed: a literature review. *BMC medical informatics and decision making*, 8(Suppl 1):S2, 2008. 1.1

V. Haarslev, R. Möller, and M. Wessel. Querying the semantic web with racer+ nrql. In *Proc. of the KI-2004 Intl. Workshop on Applications of Description Logics (ADL04)*, 2004. 1.1

E. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007. 1.1

A. Kiryakov, D. Ognyanov, and D. Manov. Owlim a pragmatic semantic repository for owl. In *Web Information Systems Engineering WISE 2005 Workshops*, pages 182–192. Springer, 2005. 1.2

Franz Inc. Allegrograph. http://www.franz.com/agraph/, 2011. 1.2, 2.4

I.F. Cruz. *The Semantic Web: ISWC 2006: 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006: Proceedings*, volume 4273. Springer-Verlag New York Inc, 2007. 1.2

J. Dolby, A. Fokoue, A. Kalyanpur, A. Kershenbaum, E. Schonberg, K. Srinivas, and L. Ma. Scalable semantic retrieval through summarization and refinement. In *Proceedings of the National Conference on Artificial Intelligence*, page 299. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007. 1.2

B.C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Modularity and web ontologies. *proc. KR*, 2006, 2006. 1.2

Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2): 158–182, 2005. 1.2, 2.2.3, 2.4

M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial intelligence*, 48(1):1–26, 1991. 2.2.1

L.R. POUR. A hybrid abox calculus using algebraic reasoning for the description logic shiq, 2012. 2.2.2

P. Dongilli. Natural language rendering of a conjunctive query. *KRDB Research Centre Technical Report No. KRDB08-3). Bozen, IT: Free University of Bozen-Bolzano*, 2008. 2.2.5

S. Abiteboul, R. Hull, and V. Vianu. Foundations of databases. 1995, 1995. 2.2.5

D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, and R. Rosati. Ontologies and databases: The dl-lite approach. *Reasoning Web. Semantic Technologies for Information Systems*, pages 255–356, 2009. 2.2.5

World Wide Web Consortium. http://www.w3.org, 2012. 2.3

D.L. McGuinness, F. Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10:2004–03, 2004. 2.3.3

B. Motik, B.C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. Owl 2 web ontology language: Profiles. *W3C Recommendation*, 27:61, 2009. 2.3.3

M. Horridge and S. Bechhofer. The owl api: a java api for working with owl 2 ontologies. *Proc. of OWL Experiences and Directions*, 2009, 2009. 6.1