# Implementation, Test and Evaluation of Load Balancing Strategies for Multi-User Inference Systems

**by Tobias Berger**

**Supervisor: Prof. Dr. Ralf Möller**
**Second Supervisor: Assoc. Prof. Dr. Volker Haarslev**
**Advisor: Atila Kaya, M. Sc.**

TUHH
*Technische Universität Hamburg-Harburg*

Submitted in partial fulfillment of the requirements for the degree
Master of Science in Information and Media Technologies

Hamburg, February 2007

# Declaration

I declare that:
this work has been prepared by me,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Montréal, February 2007
Tobias Berger

# Acknowledgements

I would like to thank Prof. Dr. Ralf Möller for providing me with a very interesting and challenging topic of research and for giving me the opportunity to perform this work at the Concordia University in Montreal, Canada.

I would also like to thank Associate Prof. Dr. Volker Haarslev for giving me the opportunity to perform this work at his department at Concordia University and providing me with excellent equipment for the testing.

Further I want to thank Atila Kaya for his patience, dedication and very many, long and valuable telephone conferences that provided me with inspiration and valuable guidance throughout this work.

I also would like to thank Irma Sofia Espinosa Peraldi and Michael Wessel for their valuable advices.

I would further like to thank the DAAD for providing me with the very important financial support that first gave me the possibility to stay in Montreal and perform my thesis abroad.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

> *They want to deliver vast amounts of information over the Internet. And again, the Internet is not something you just dump something on. It's not a big truck. It's a series of tubes.*
>
> *And if you don't understand those tubes can be filled and if they are filled, when you put your message in, it gets in line and it's going to be delayed by anyone that puts into that tube enormous amounts of material, enormous amounts of material.*
>
> - Senator Ted Stevens, member of the United States Senate, June 28th 2006

In its beginning the Internet was mostly the playground for people pursuing academic interests, connecting individual research networks with each other. At this time there was no to very little consumer use. Even when the personal use on the Internet increased in the middle of the 1990's web sites were not used much for commerce, it was more a new media for presentation. More or less none of the use was in any way critical to the profit of a company. At that time a single server was able to handle the traffic of even the most popular web sites, and when it needed to be maintained or went down due to a failure, it was not much of a big issue.

When companies started to recognize the potential of the Internet combined and triggered by a highly increased amount of users and potential customers, its importance for commerce and companies' profits started to rise. With the increase of its significance, many issues that could be neglected before became crucial and had to be addressed.

In the beginning companies and Internet Service Providers started to increase the capabilities of their servers, e.g., by increasing the memory or upgrading the processor. But this can only scale to a certain point. Thus a way had to be found to distribute the load over more than one server. The solution was the introduction of load balancing techniques that spread

in many varieties into various application areas to ensure and even increase scalability, availability and the quality of service.

#### Commercialization of the Semantic Web

A crucial requirement of the Semantic Web vision to come true is the efficiency of reasoning over web-wide distributed ontologies and the quality of service connected to that. In a way the Semantic Web undergoes the same development as the Internet did in its infancy.

State-of-the-art reasoners are able to deal with standard reasoning tasks and large ontologies in a quiet efficient way. The quality of service they offer suits their users, who mainly use it for special tasks that involve only a few users, respectively use it for academic purposes. Similar to the development of Internet technologies the Semantic Web offers a wide field for new business models and commercial activities that will change the demands for the systems.

The typical Semantic Web scenario where software agents accomplish complex tasks requires a scalable inference infrastructure that provides for efficient reasoning on ontologies with respect to implicit knowledge. The following example will illustrate that.

## 1.1 Multimedia Document Retrieval

The project *BOEMIE (Bootstrapping Ontology Evolution with Multimedia Information Extraction)*, funded by the European Commission, is currently involved in developing a system to automatically extract information from multimedia content. Low-level objects are extracted from from several modalities such as still images and based on these more abstract (highlevel) knowledge will be discovered with the help of reasoning about multimedia ontologies.

**Example.** Illustrators of newspapers, online content, books or other media enrich the textual content with pictures that fit the topic and underline the point the article tries to make. For this purpose they can access large databases of content brokers that offer photographs for all kinds of different topics and current events. In this case the illustrator may be working on an article about athletics and searches for an athlete jumping over a hurdle in a race to underline the strength, agility and technique needed in athletics. The service, provided with the keywords, will search the database for matches and return the found photographs. The information about the pictures in the database have been attached to the image (a.k.a. annotations) by a person previously.

In the BOEMIE scenario the attaching of information would be unnecessary. When a new photograph is put into the system, the service will

recognize low-level objects like a horizontal bar (the hurdle bar), two vertical bars, or a person with his legs spread (jumping) above the vertical bar. Given the low-level objects the reasoner can, based on the multimedia ontologies, infer knowledge about this photograph, using high-level semantics to put the low-level objects together to an athlete jumping over a hurdle. In combination with further information coming from a database, like where and when the picture was taken, the *multimedia document retrieval* service will find a match for the search criteria of the illustrator. Moreover endusers of the system, e.g., the illustrator in our example, can not only make a key word based search but also pose complex queries.

#### Scalability, Availability and Quality of Service

Like other services on the Internet, this service will attract many users who will access the service in parallel and wont be willing to wait for other users to finish their searches before using the service themselves. A single reasoner is not capable of answering multiple requests at the same time, i.e. not able to provide multi-user support that is essential for the Semantic Web scenario. Thus in order to address the upcoming, business critical issues of scalability, availability and quality of service the system has to increase its reasoning capabilities. More than one reasoner have to work in parallel to support the service and remove the bottleneck of the system, supported by an intelligent load balancing algorithm that distributes the incoming requests over the reasoners.

In the given example even more is needed. The service works on information of multiple sources, inferred by a reasoner and retrieved from a database. These information have to be combined to return the most suiting matches. Furthermore, as such a service will often involve critical business issues, more features may be needed to support it. Secure connections, electronic signatures, payment services, etc. Thus more than a plain and simple load balancing system is needed to support and integrate all demands. These features are not unique to the example, but will be needed in many other fields of application.

## 1.2 Towards A Middleware for Semantic Web Inference Systems

The analysis of the example showed that today's inference systems are neither well enough equipped to offer the same quality of service for multiple, parallel clients as for a single client. Nor do they offer the services and openness to be integrated into an existing service infrastructure and business environment.

To bridge this gap this thesis proposes the introduction of a Semantic

Web middleware that will be able to cope with the upcoming requirements for multi-user Semantic Web inference systems. While considering all requirements, the focus of this thesis is the issue of *load balancing*. Load balancing is of particular interest because introducing more reasoners for inference tasks are not enough to scale the system. It is the most essential requirement for such a middleware, as it builds the basis for multi-user systems, i.e. it builds as well the basis for the other requirements and further features.

In the following all the possibilities and constraints of load balancing for inference, or reasoning systems will be described and explained to come up with an intelligent load balancing algorithm. After the theoretic considerations, its effects on availability, scalability and quality of service will be evaluated on an empirical basis.

Further, the positive effects of intensive preprocessing as part of the load balancing activity will be analyzed and evaluated, theoretically and empirically as well.

The analysis of the topic will start with an overview on how load balancing techniques are applied in todays existing, well-known systems.

# Chapter 2

# Load Balancing in Well-known Systems

Load Balancing as such is not a new concept. It is well-known in many application areas, where load balancers of different kinds perform valuable work in order to enable large scale systems, networks and applications.

Load balancing systems exist in various kinds of implementations and are equipped with multiple extra features to serve the diverse requirements of the different systems. The main application areas are given in the following.

## 2.1 Application Areas for Load Balancing

### 2.1.1 Server Load Balancing

Servers are the publicly best-known, probably because most obvious, application area for load balancing. In situations when a single server is not capable to cope with the load because it exceeds its capabilities, more servers are put together in clusters. The load balancer serves as the central and only connection of the cluster to the external network. Redundancy and automatic failure handling and recovery can be easily integrated into the system, offering a high level of availability.

A common scenario for the use of *Server Load Balancing*[3] (SLB) is the use of load balancers in the webserver environment. Websites like Yahoo! offer their services to millions of users, and to tens of thousands in parallel. In order to have the service available and responding in an appropriate time, the load in form of requests must be distributed over many single servers that can cope with the requests in parallel. These servers include webservers as well as database servers.

As load balancing is best known from this application area and it is the area with the most significance in terms of the thesis, this chapter will focus mainly on server load balancing.

### 2.1.2 Global Server Load Balancing

The driving forces behind *Global Server Load Balancing*[3] (GSLB) are the need for *high availability* and *faster response times*.

While SLB addresses the problem of availability of a service by introducing server clusters, a certain redundancy with a high degree of automatic action in failure situations, GSLB tries to address the availability problem on the macro level and resolve the problem of a whole data center being disconnected. In the case of a natural disaster like an earthquake GSLB tries to ensure that the application, e.g. a website, can be operated from another data center probably located somewhere else in the world.

Besides the availability aspect, the response time is targeted as well. GSLB addresses the problem of network latency coming into play for long distance connections. If a website would be operated from a single data center, clients from the whole world would have to connect to this center in order to receive a response, which can be of serious concern. GSLB tries to distribute the requests based on information of the clients location. Thus the client will be connected to the nearest available data center to improve the response time.

### 2.1.3 Firewall Load Balancing

Firewalls have a limited throughput as they perform a costly task. If the network design has to be scaled in order to support higher throughput the introduction of one or more further firewalls may be the only solution. *Firewall load balancing*[3] addresses this problem and helps to distribute the load over systems with multiple firewalls.

A system with multiple firewalls is also preferable from an availability perspective. In a single firewalled system, this is the single point of failure. The whole system will loose its external connection when the firewall fails or has to be taken out of work for maintenance. The introduction of multiple firewalls improves the availability with one firewall being the backup for the other.

### 2.1.4 Cache Switching

While in the examples discussed so far the load balancer tries to distribute the load over the single units as evenly as possible, load balancing with caches is in some respect different from that. Here the load balancer has to pay attention to the information that are already contained in a certain cache, as it is more effective to direct a request to a cache that has the information already stored.

## 2.2 The Benefits of Load Balancing

### 2.2.1 Scalability

By distributing the load across many real servers in a cluster, the load balancer enables parallel processing of different requests. The collective processing capacity is thus much higher than the processing capacity of a single server. In an ideal scenario the processing capacity of the whole system would be the sum of the servers capacities that are part of the system. Although this is not necessarily achieved in real world applications, due to various inefficiency factors, it provides an excellent scalability.

### 2.2.2 Availability

The load balancer is able to continuously perform health checks on the connected servers by analyzing the traffic to and back from the server back. If a server fails to respond the load balancer can be automatically excluded this server and direct further work to one of the other, healthy servers. The load balancer makes this work totally transparent for the user. As it is done on the fly it minimizes downtime significantly. In a similar fashion, the excluded server can be included later, once it is healthy again.

### 2.2.3 Manageability

The use of a load balancer improves significantly the manageability of a system. When a server needs to be upgraded or maintained, it can be easily excluded from the system by instructing the load balancer to send further requests to one of the other servers in the cluster.

Besides the maintenance issue, the load balancer can also improve the manageability of different content and services. The load balancer can direct different kinds of services like HTTP and FTP to different servers, each of which being specialized to offer the particular service. Different demands of different services can be reflected by different amounts of servers, offering the particular service. A split of services is also performed in database systems. When a system-wide search task is offered, the work for this search is usually performed by a different part of the cluster, on a replication of the original database cluster. Being very performance critical, these processes can be separated from the other read and write operations.

The same is true for large amounts of contents. In case the amount of content is too big for one server, respectively is too big to guarantee a response in a reasonable amount of time, it can be split. One possibility might be the differentiation between dynamic and static content. In a cluster a lot more processing power is needed to generate the dynamic content so that more real servers can be assigned for this task while a smaller amount will work on delivering static content only.

### 2.2.4 Security

The load balancer is the only connection of the servers to the outer network. As all traffic runs through the load balancer, the servers can have private network IPs, which can not be reached from outside the network, thus protecting the servers from malicious queries . Many load balancers therefore come equipped with features to prevent attacks on the servers they make transparent.

### 2.2.5 Quality of Service

The quality of service (QoS) can be defined in different ways. A common way of optimizing a system is to define the response time as an important criteria for the QoS. Load balancers allow for real parallel processing of different requests and show an improvement over a single server (in an extreme scenario single CPU) system, by reducing the average response time.

Besides the response time, availability is also a common known QoS criteria.

## 2.3 Load Distribution

### 2.3.1 Stateless vs. Stateful Load Balancing

In server load balancing the system, respectively the servers in the system can be stateless or stateful.

In a stateless system the servers do not keep track of the conversations with the clients. Each of the requests that reach the server is handled in the same way, and independent from each other. In case there is some information that has to be carried from one request to the next, it has to be included in the request, as no information is retained on server side. This makes load balancing fairly easy, as each of the servers can be regarded as equal and the decision on where to forward the request can be based only on the current load level.

In a system with stateful servers load balancing is more complicated. The load balancer has to take into account that it cannot blindly distribute the load, but has to consider whether a request is part of an ongoing session. If this is the case, the session has the priority and the request needs to be forwarded to the server that owns the particular session.

For this purpose load balancers hold session tables. The tables map a client, e.g. via its source IP and port, to a server in the system that owns the session. Besides the advantage of being able to map sessions, these tables come long with the disadvantage of extra work for maintenance.

### 2.3.2 Strategies

The load balancing systems try to achieve the desired benefits by implementing a particular strategy to evenly distribute the load among the connected servers.

**Random** The load balancer picks randomly a server from a pool of idle servers.

**Round Robin** The load balancer holds a circular list of all connected servers. Circular in the sense that the first server in the list is the successor of the the last server. Consequently an endless list of servers is created. The load balancer always picks the next server in the list to send the request.

The advantage of the method lies in its simplicity. In situations where the computing of other distribution algorithms can consume significant amounts of processing time - e.g. when the least connections have to be found for 1000 real servers - round robin is a lot more effective [3].

On the other hand its simplicity is its biggest disadvantage. As this method acts blindly it does not take the actual situation on the servers into account, which could have very different amounts of work waiting to be done due to different complexities of requests. This may lead to a very unbalanced situation, at least for a certain period of time.

**Least Connections** The next request is sent to the server with the fewest open connections at that time. As the load balancer is servers' only connection to the network, it can keep track of each of the servers' connections. This method is one of the most popular and effective methods in load balancing for many applications [3].

**Response Time** Similar to the least connections strategy, the load distribution can also be based on the response times. As in many application areas, e.g. for many websites this is the case, the quality of service is measured in response time, where it is absolutely crucial to deliver a response as fast as possible. The server with the fastest response time gets the next request. As this request, respectively a set of following requests, will slow this server down again, another server will have the fastest response times and therefore become the preferred one.

**Server-Side Agents** The use of server-side agents is another way of including data about the current load status of a server into the the decision process. This may be the most accurate process as the agents will deliver all kinds of data, e.g. the CPU usage and information about the system's

health. However using agents for this work also means a lot higher degree of complexity. This increases the need for maintenance as well as the complexity for maintenance.

**Weighted Distribution** The introduction of weights in the load distribution gives the load balancer the possibility to reflect different processing capabilities of the connected servers. Servers with the double amount of processing capabilities than a "standard" server in a system, could have assigned twice the standard workload. Consequently, the existing equipment can be used much more efficient.

## 2.4 Health Checking

Besides the distribution of load, the load balancer fulfills also the purpose of health checking the system's components. The level to which the health checking is done is thereby arbitrarily high.

The basic health checking can be performed as so called in-band check. This basically means that the load balancer keeps track, if it received a response to a particular request it send. If so, the concerned server can be considered to be healthy. A load balancer at OSI Layer 4 could, e.g., attempt to access a specific TCP or UDP port. For that the load balancer sends a TCP SYN request and waits for the TCP SYN ACK response from the particular server in return. If the TCP SYN fails and no acknowledgment is received, the particular port at the server is down.

In some applications the load balancer can also perform more sophisticated health checking at the application layer. E.g., a load balancer could be instructed to send out HTTP requests for a specific URL. The load balancer can be configured to analyze the response code to detect 404 (Object not found) codes.

The load balancer thereby performs a very valuable task and makes efficiently use of its resources.

## 2.5 Implementation

Todays load balancers for server load balancing are mostly implemented in two ways: as Layer 2 switch or Layer 4 Switch.

**Layer 2 Switching** got their name from and perform their work at the OSI Model[1] Layer 2, the data link layer. They make use of the information at this level, to make a decision on where to forward the request. As they perform their work on a very basic level, it does not involve a lot of work for the analysis of a packet and therefore can be performed very fast. The flip

side of this medal is that thus they can not profit from these information either. This is where Layer 4 Switching starts to work.

**Layer 4 Switching** tries to make use of the benefits that Layer 2 Switching neglects. It uses information from the headers of layer 4 (and sometimes beyond this) to make more intelligent load balancing decisions. Besides an improved load distribution, these load balancers also offer advanced health checking by recognizing traffic for different protocols such as HTTP, SSL, FTP etc.

## 2.6 High Availability Design

A very important feature with regard to the availability of a system is the possibility of a load balancer to detect server failures and dynamically route the traffic load to other available servers. However, a problem that is not addressed by this mechanism is the failure of the load balancer itself. Being the central in and out of a subnetwork the load balancer is a potential single point of failure.

The solution in *high availability design* is again a backup - this time of the load balancer- , respectively added redundancy to make the system more tolerant to failures in load balancers. Load balancers can work in pairs in two different modes: *active-standby* or *active-active*.

### 2.6.1 Active-Standby Configuration

The active-standby configuration involves two load balancers. As indicated by the name, one of the load balancers is active while the other remains in standby mode, monitoring the active load balancer. The standby unit basically performs health checking through a private link on the active unit to detect a failure, where it will take over the active part. Some systems are able to do this in less than a second.

In order to take up full performance as fast as possible, the standby unit has to keep track of the health status of the system. Thus, although being in standby mode, it has to perform health checking to be better prepared for a failure of the active unit.

Another issue is the session management. In order not to drop all current connections, the system has to perform a *stateful failover*, which basically means that the standby unit keeps a current copy of the session table of the active one. This is only possible, if the active unit provides the standby unit with an update whenever a new session is established. Due to the heavy communication between the load balancers this feature can affect the performance of the load balancers significantly. It is preferable to have the units physically as close as possible in order to reduce network latency to a minimum.

### 2.6.2 Active-Active Configuration

The active-active configuration also involves the use of two load balancers, but in this case both are active. While the active-standby configuration provides good support for high availability, the load balancers in this configuration work simultaneously while backing each other up. Besides improved availability this system offers higher load balancing performance as well. In the case that one load balancer fails, the other has to do the whole work.

In order to be able to take over the other load balancers work the two units have to share their sessions in the same way as it was described in the stateful failover scenario of the active-standby configuration. Besides that this allows for both load balancers to deal with any session that reaches the system. Thus the system does not need to take care about routing a session always to the same load balancer.

# Chapter 3

# Multi-User Inference Systems

Today's reasoning systems are developed to a point where they are highly optimized for standard reasoning tasks and deal with large ontologies. In an experimental, academic environment where the reasoning systems sufficiently meet the demands and expectations of their users.

For an application of the Semantic Web technologies in a business environment where multiple users access the systems concurrently and the applications will be embedded in an existing service infrastructure, new and more requirements will be demanded from the systems that have to be addressed.

## 3.1 Description Logics Reasoners

### 3.1.1 Description Logics

Description Logics (DL)[4] are a family of formal languages for representing knowledge and reasoning. The family of knowledge representation (KR) systems provides the basis to represent knowledge of a domain by providing the formalisms and to define the concepts of a domain as well as for escribing the individuals and their attributes that occur in the domain.

Besides the representation of knowledge, description logics provide a way to reason about this knowledge. *Reasoning allows one to infer implicitly represented knowledge from the knowledge that is explicitly contained in the knowledge base.*[4]. All this is supported by inference patterns. which are also used by humans. This provides the systems with a clear distinction and advantage over database systems.

Figure 3.1 shows the basic architecture of a DL based KR system. A knowledge base (KB) consists of two components, the TBox, which describes the *terminology* and the ABox, which gives *assertions*, the specifications of

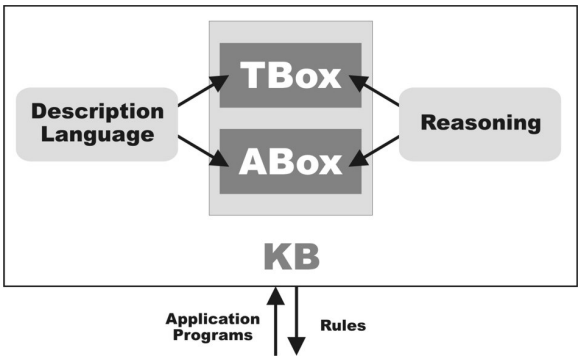


Figure 3.1: Architecture of a knowledge representation system based on Description Logics.[4]

the individuals with reference to a TBox.

### 3.1.2 TBox

The TBox, the terminological Box[4] provides a knowledge base with the a set of axioms that describes a domain. This vocabulary consists of *concepts* and *roles* to describe relationships among the concepts.

Elementary descriptions are *atomic concepts* and *atomic roles*. All other more complex description are constructed from them.

#### Description Language *AL*

The language $AL$[4] (attribute language) was introduced as the very basic way to describe knowledge that is applicable in praxis. There are several. $AL$ provides the basis syntax for taxonomies to describe their concepts and roles.

#### Terminologies

The DL uses axioms to make statements about concepts and their relations such as inclusion or equivalance with other concepts in the TBox. The finite set of definitions is called the terminology or TBox. In general these axioms have the form

$$C \sqsubseteq D \quad or \quad C \equiv D$$

where $C$ and $D$ are concepts. The same definitions also apply to roles as well. Figure 3.2 shows an example TBox defining the concepts for the domain family, assuming that the concepts *Person* and *Female* as well as the roles *hasChild* and *hasWife* are atomic.

$$
\begin{array}{rcl}
\text{Woman} & \equiv & \text{Person} \sqcap \text{Female} \\
\text{Man} & \equiv & \text{Person} \sqcap \neg\ \text{Female} \\
\text{Mother} & \equiv & \text{Woman} \sqcap \exists\ \text{hasChild.Person} \\
\text{Father} & \equiv & \text{Man} \sqcap \exists\ \text{hasChild.Person} \\
\text{Parent} & \equiv & \text{Mother} \sqcup \text{Father} \\
\text{Grandmother} & \equiv & \text{Mother} \sqcap \exists\ \text{hasChild.Parent} \\
\text{MotherWithoutSon} & \equiv & \text{Mother} \sqcap \forall\ \text{hasChild.}\neg\text{Man} \\
\text{Husband} & \equiv & \text{Man} \sqcap \exists\ \text{hasWife.Woman} \\
\text{FatherWithoutSon} & \sqsubseteq & \text{Father} \\
\text{Father} & \sqsubseteq & \text{Parent}
\end{array}
$$

Figure 3.2: TBox defining concepts of the domain family.

In a domain there may be concepts that can not be defined entirely. For those concepts inclusions can be used to at least state some necessary conditions.

### TBox Inference Services

As mentioned before, a knowledge representation system delivers more than a storage unit for concepts and their assertions. Besides this explicit knowledge, a KR system that is based on DL contains also implicit knowledge that can be accessed through inference. A very basic example for inference is that although it was not stated anywhere explicitly, it can be concluded that a *Grandmother* is a *Woman*. The basic reasoning tasks for TBoxes[4] are listed below, where checking a concept for *satisfiability* is a key inference. Many other inference tasks can be reduced to a problem of satisfiability. For the following list let $\tau$ be a TBox.

**Satisfiability.** A concept $C$ is satisfiable with respect to $\tau$ , if there exists a model $\iota$ of $\tau$ such that $C^\iota$ is not empty.

**Subsumption** checks if one concept is more general than another. A concept $C$ subsumes a concept $D$, if in every model of $\tau$ the set defined by $D$ is a subset of the set defined by $C$.

**Equivalence** checks if two concepts $C$ and $D$ are equal, meaning if for every model $\iota$ of $\tau$ $C^\iota \equiv D^\iota$.

**Disjointness.** Two concepts $C$ and $D$ are disjoint, if for every model $\iota$ of $\tau$ the interception of $C^\iota$ and $D^\iota$ is the empty set.

**Example.** With respect to the TBox of figure 3.2 it can be stated that e.g. *Man* subsumes *Father*, *Mother* subsumes *Grandmother* as well as that *Mother* and *Father* are disjoint.

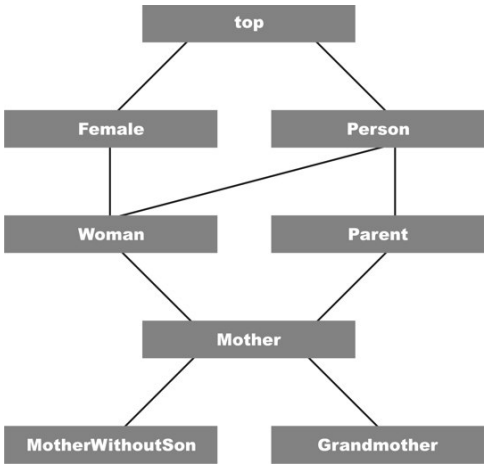


Figure 3.3: Taxonomy based on the family TBox.

#### Taxonomy

DL reasoners often compute a hierarchy based on the subsumption relationships of the concepts of a TBox. This is on the one hand a very demonstrative and intuitive way for a human to explore the concepts and their relationships of a TBox, on the other hand it offers a way to improve the reasoning of a reasoner. An example taxonomy, based on the TBox given in figure 3.2 is drawn in figure 3.3.

### 3.1.3 ABox

The second part of a knowledge base is called ABox[4], assertional Box or world description. The ABox defines and describes the individuals of a domain by asserting them particular concepts. The individuals of a concept are similar to instances of a class.

$$homer : father,$$

with Homer being the individual and father the concept. In the same way roles can be defined

$$(homer, bart) : hasChild$$

For sure the ABox has to be as consistent as the TBox, otherwise arbitrary conclusions can be drawn from it.

#### ABox Inference Services

In the following the inference services[4] for ABoxes are given:

**ABox satisfiability.** The assertions in the ABox are checked for satisfiability with respect to the referenced TBox. E.g., an ABox with the

assertions $homer : father$ and $homer : mother$ is not satisfiable because the concepts father and mother are defined as disjoint concepts in the TBox. (Remember figure 3.2 $Man \equiv Person \sqcap \neg Female$)

**Instance checking.** The system checks if the individual $a$ an instance of the concept $C$ or if $C$ subsumes the individual. $a$: $instance?(a, C, A)$

**ABox realization.** Compute for all individuals in $A$ the most specific concept they are an instance of (with respect to the TBox $\tau$).

Similar to the TBox inference, the key inference is the *satisfiability* to which all other inference services can be reduced.

#### Taxonomies

In the same way as a TBox taxonomy can be inferred from the concept definitions, a taxonomy for the individuals can be created.

### 3.1.4 OWL Query Language

The OWL Query Language[13] (OWL-QL) is a candidate standard language and protocol submitted to the W3C Consortium for query-answering dialogs among Semantic Web agents. It is designed for a broad range of Semantic Web applications, where servers derive answers for client queries. It is intended for knowledge bases represented in the Web Ontology Language.

Examples for OWL-QL queries can be found in appendix D.

#### Web Ontology Language

The Web Ontology Language[14] (OWL) offers a formal way to describe, publish and distribute ontologies. It is based on web languages as XML, RDF and RDF-S, but exceeds their capabilities in terms of semantically rich, machine-interpretable knowledge representation.

There are three species of OWL: OWL Lite, OWL DL and OWL Full. Each of these sub-languages provides a different degree of expressiveness with OWL Lite being the least expressive language.

OWL describes the terminology of a domain as *classes* and *properties* and the assertions as *instances*. Its representation is XML.

#### OWL-QL Features

The OWL Query Language supports the following features, based on assumptions about query-answering dialogs in the Semantic Web.

**Variety Support.** OWL-QL supports query-answering dialogs of different kinds. Scenarios where agents perform automated reasoning, scenarios with multiple as well as scenarios with unspecified knowledge bases are

supported. Thus OWL-QL tries to be as universal as possible and live up to the expectations of a multi-purpose Semantic Web environment.

**Adaptable Query Answering Protocol.** Not only the purposes of querying, but also the queries as such are assumed to be very different. Thus OWL-QL offers a means to handle partial information, performance limitations and unpredictable settings in terms of answer sizes and processing time by offering a protocol that on the one hand allows the client to specify the maximum amount of answers and on the other hand allows the server to respond with a partial set of answers.

**Server-side KB Selection.** Similar to the functions of todays's search engines, in the Semantic Web scenario there will be needs for a client to send a query without specifying the knowledge base that is to be queried. It will be expected from the server to select a reliable source from which it will produce the answers. OWL-QL offers a means for this feature, providing clients with the possibility to instruct servers with the selection.

**Representation Independence.** On the Web of today there are a lot of syntactically different notations for information. This can be assumed to be the same in the Semantic Web scenario. OWL-QL supports this variety of notations by defining only an abstract, structural level, leaving the concrete implementation open.

**Semantic Description.** The promise of the Semantic Web is that knowledge will be represented with formally defined semantics and a theory of logic entailment. Thus this premise has to be also applied and supported by a query language. OWL-QL supports a semantic description of the relationships among the queries and the knowledge bases they use to generate answers.

### 3.1.5 Current Reasoner Technology

Description logic systems allow for formal domain modeling and support decidable reasoning problems [4]. Nowadays, DL-based reasoners offer a variety of useful inference services which solve reasoning problems.

However, when building practical applications, the efficient interaction between clients and DL-based systems is still a big challenge for modern reasoners, especially in case of ontologies with a large number of individuals. In the Semantic Web context, the amount of time a reasoning server needs to answer a query and the answer size may both become unpredictable. Although research on optimization techniques for single reasoning systems has led to substantial performance improvements recently, optimization techniques for front-end systems managing multiple reasoners have not been developed.

In the following requirements for a load balancing solution in connection with the introduction of a *semantic middleware* will be discussed.

## 3.2 Semantic Middleware

Load balancing is a common technique in various systems, as it was explained in the previous chapter. Referring to paragraph 2.5 about the *implementation of load balancing in known systems*, the first load balancing was implemented as Layer 2 application, performing the load balancing on the OSI Model Data Layer. Although this method still suites various current applications very well, Layer 4 load balancing was introduced later to perform its work on the OSI Model[1] Network Layer. This was done so to profit from information that were contained within this layer to improve the load balancing performance and enable other kinds of applications.

The same kind of evolution has now to take place in order to enable efficient load balancing for Semantic Web applications. While, e.g. for server load balancing, the Layer 4 systems achieved significant performance improvements, the limited features of these kinds of systems are not suitable for Semantic Web reasoning systems. Those information that will enable the load balancer to make an efficient decision on where to direct the incoming requests are contained in Layer 7. Therefore this information offers chances for further improvements. Based on this and the following arguments it is evident that the only suitable place for an efficient load balancing system is at the application layer, Layer 7 of the OSI Model.

Moreover the requirements of many Semantic Web scenarios, as well as the possibility for further optimization of the load balancing process due to the special nature of reasoning systems, show clearly that it is required to introduce a *semantic middleware*. This middleware has to offer possibilities and address issues that go far beyond the capacities of a plain and simple load balancer. The following arguments will support this claim of implementing the load balancer as part of a Semantic Web middleware.

## 3.3 Requirements for a Semantic Web Middleware

### 3.3.1 Integration into Existing Architecture

In order to make Semantic Web applications a success and suitable for many clients, businesses and environments, applications and the load balancers will have to be integrated in existing infrastructures and architectures. Thus the systems have to implement widely accepted and used standard languages, interfaces and protocols in order to achieve a broad client acceptance and

to support a wide range of applications. Consequently service oriented architectures (SOA) and web services are a first class choice.

This requirement alone exceeds the possibilities of known load balancer implementations for server or firewalls. Only the extension of a load balancer to a middleware system on the Application Layer is able to handle the integration of a load balanced reasoning system into an existing service oriented architecture.

### 3.3.2 Improved Load Balancing

As mentioned earlier, although significant improvements have been made to reasoners, reasoning still remains an expensive task. Therefore preprocessing of requests and various steps of optimization, based on the information that is contained in the requests, i.e. in the queries, offer a very promising way for a significant performance improvement.

#### History Sensitivity

Reasoners are *history sensitive*, i.e. their performance for a particular query is directly influenced by the queries they answered in the past.

The most significant factor about this issue is the fact if a reasoner has already loaded the knowledge base that is referenced by the current query or not. A reasoner that has, due to an earlier query to the same knowledge base, already loaded the knowledge base for the current query, is able to answer the query faster than a reasoner that has to load the knowledge base prior to answer the current query. In the case of a large knowledge base this advantage can be very significant.

But not only the time for the loading of a knowledge base is of significance. In the Semantic Web context, the amount of time a reasoning server needs to answer a query and the answer size may both become unpredictable. In order improve the response times, most reasoners implement a cache that will store the inferred knowledge. Especially in the case of very expensive queries this can make a big difference, if a query has been answered by the same reasoner before.

#### Sophisticated Strategy and Optimization

The history sensitivity of the reasoners makes it crucial for an effective load balancer to implement a sophisticated load balancing algorithm that takes into account as many information as possible to make the best decision for the distribution of load.

Moreover a semantic middleware offers the possibility for optimization, which gives the chance to avoid as much unnecessary reasoning as possible.

### 3.3.3 Integration with other Components

In various Semantic Web scenarios features that exceed the basic load balancing are needed to enable different kinds of applications. A semantic middleware is needed to integrate further services, like the following, into the system.

**Accounting Modules** In some scenarios it is possible that clients pay for the service that various reasoning services offer. Thus it must be possible to integrate a accounting module into the middleware to allow the clients to identify, use a service and bill them for it. Instead of implementing a system for each reasoner, the central middleware system is the most efficient place for the integration of such a service.

**Verification Service** There will be clients that due to the seriousness of there issues, will demand verification of the service provider and the sources that were used. Especially in those scenarios that will include payments for services this will be an important issue. For the same reasons as in the previous module, the middleware system is the optimal place for the integration of such a service.

**Security Features** Business applications for the Semantic Web will have a need for security features such as encryption and firewalling in the same way as current Web applications have it. Again a middleware system is here the first choice as it offers a natural hiding and shielding mechanism for the server architecture behind it. The integration with a semantic middleware offers here a good chance for the use and transfer of technologies that are known from other SOA applications.

### 3.3.4 Scalability

In the same way as scalability was and still is an issue in Webserver systems, firewalls and other systems described in the previous chapter, it is an issue in the Semantic Web scenario. Different applications will cause high query traffic which will cause a need for the systems to scale. For the same reasons as in known scenarios, the best scalability is provided by multiple reasoners used in parallel an managed by a central load balancing unit, which as argued in the previous paragraph, is ideally part of a semantic middleware. Moreover, the fact that single reasoners are not able to handle concurrent client requests makes it impossible for them to scale. Thus, in order to scale further reasoner have to be added and managed in parallel.

### 3.3.5 Availability

Systems in the Semantic Web scenario will have to support as well a high degree of availability.

#### Handling of Concurrent Client Requests

Although highly optimized reasoners have achieved a significant performance increase over the last years, in the application scenarios with multiple parallel client requests this effect decreases significantly. Due to reasoners being only able to handle one client request at a time, thus do not support concurrency of client requests, a single query can block the whole reasoner, i.e. the whole system.

In order to provide a basic and further on a high degree of availability, the system has to provide capabilities for scenarios with multiple parallel client requests. This leads to the same requirements as in the previous paragraph: multiple, parallel reasoner instances that are managed by a central semantic middleware.

#### Health Checking

Besides the basic support for parallel client requests, the issue of health checking can provide a significant improvement of the availability of the system. This is very similar to issues in server load balancing.

**Failure Detection.** In a scenario with a client and a single reasoner, the client may not be able to detect that the reasoner failed and is no longer - at least for a certain amount of time - able to provide reasoning capabilities. A system with a central unit like a load balancer, i.e. semantic middleware, is able to detect such failures. It is then able to (i) send the current client's request to another server in the system and (ii) exclude the failed reasoner from further instructing with queries. This allows the system to maintain a high degree of availability, even more in combination with a mechanism for *automatic recovery*.

**Automatic Recovery** Besides the exclusion of failing reasoners, thus ensuring the correct answering of client queries, the system can further improve its availability by monitoring or even initiating automatic recovery functions in the reasoners. Once a reasoner failed, the semantic middleware can, e.g. cause it to flush its memory or restart. When the server is back and ready to perform, it can be dynamically reintegrated in the list of available reasoners.

### 3.3.6 Support for Standard Query Languages and Protocols

As argued before the integration of the semantic middleware into existing service oriented architectures is the preferred choice. In this scenario the OWL Query Language, for the reasons that were given in paragraph 3.1.4, offers a promising suitability as query language and protocol.

The OWL-QL standard, as mentioned earlier, does not specify anything about server-side implementation details such as caching, number of reasoners used as back-end etc. However, the OWL-QL protocol for query-answering dialogs and the heterogeneous nature of the Semantic Web itself makes it obvious that the middleware which claims to serve as an OWL-QL server has to manage multiple reasoners in the background.

With regard to the near future of modern DL systems, the existing standard interface for accessing DL reasoners (DIG) will become more important not only as a communication protocol but also as a query language. The upcoming version, namely DIG 2.0 [19], offers many essential features such as iterative query answering and query management. Thus it is realistic to expect that DIG 2.0 will replace OWL-QL as a standard query language.

### 3.3.7 Iterative Query Answering

As explained in the previous paragraphs a semantic middleware has to support various application scenarios. The support of OWL-QL in the middleware is one feature that helps to support this.

Modern reasoners as well as OWL-QL (being the interface to the client) support iterative query answering, where clients may specify the maximum number of answers they want to get from the server, and thus servers return partial answer sets. By using available configuration options, such as incomplete modes, that are powerful enough for semi-structured ontologies, the reasoner achieves significant performance improvements for handling large ontologies. However, this task still demands reasonable hardware resources and is notably memory-intensive.

Thus the semantic middleware not only has to support this feature, but it is also required to exploit its functionalities and configuration options to further improve the quality of service.

### 3.3.8 Server-side Knowledge Base Selection

The issue of server-side knowledge base selection was already mentioned in the paragraph about OWL-QL. In some of the Semantic Web scenarios the client may not be able to specify the knowledge base that is to be used to answers the query. This may be due to missing information at the client side, or due to improvement in the quality of service of the system. However, the semantic middleware has to support this feature in order to be used in these scenarios.

Firstly the semantic middleware has to support a query language and protocol that considers this feature. As earlier mentioned the OWL Query Language offers support for this.

Secondly the semantic middleware itself has to implement components that allow for an efficient discovery and selection of an appropriate knowl-

edge base. Even if the reasoners of the system are equipped to support the feature, the more efficient way will be the central discovery and selection of a knowledge base. All queries will profit from earlier discovered and already loaded knowledge bases, independent of the particular reasoner that will be instructed to deliver the answers.

### 3.3.9 Quality of Service

Besides the basic implementation of the given requirements, a central concern for a semantic middleware will always be - as it was already in load balancing systems for servers, firewalls, etc. - the improvement of the quality of service, respectively the offering of a quality of service above a certain level. The two basic concerns in this respect are *high availability* and *fast responses*.

**High Availability.** The specification, respectively the implementation of a semantic middleware can put its focus on high availability. For applications that require a system that they can absolutely rely on, redundancies have to be implemented. The servers and the middleware itself can be implemented redundantly, backup systems can be installed and all in all the whole system can be focused more on the requirement of health checking than others.

**Fast Responses.** A common requirement for any kind of Web application are fast responses. As it was the case in many Web business applications, it will be the case for business scenarios on the Semantic Web as well. Clients are very sensitive in terms of the time they are willing to spend waiting for a certain service to respond. Especially nowadays when the network becomes less and less of an issue, users are used to fast responding applications and will demand this as well from applications of the Semantic Web. Thus the success of many Semantic Web applications will depend on the time users have to spend waiting for a response.

# Chapter 4

# Optimization Criteria

Reasoning is an expensive task. This makes it very probable for intensive preprocessing to pay off. Thus various possibilities for optimization will be analyzed and explained in order to improve the system's availability and quality of service by reducing the response times of the queries.

## 4.1 Query Types

Although the probability is very high that any kind of preprocessing of a query pays off by reducing the amount of reasoning it has to be avoided anywhere, where a profit from the preprocessing can be excluded beforehand.

Given the characteristics of reasoning systems, the queries can be classified as one of four query types, each of which offers a particular possibility for optimization. The classification can be assumed to be a minor cost, especially when compared with the unnecessary preprocessing that can be avoided by it.

In the following the four query types with their characteristics will be explained.

### 4.1.1 Query to an Unknown Knowledge Base

An unknown knowledgebase is a knowledgebase, which has not yet been loaded by any of the reasoners that are managed by the semantic middleware. As the only external connection of the reasoners goes through the middleware an unknown knowledgebase is also always unknown to the middleware, which makes it more easily to detect.

Instances of this query class reference an unknown knowledgebase, which means it is the first query to a knowledgebase. Thus the system cannot have any information, neither about the knowledgebase nor about the query. This query class therefore does not offer any possibility for optimization. Every attempt for optimization would be a waste of resources and thus has to be

avoided. In this case the classification will only help to avoid a decrease in the response time by unnecessary optimization attempts.

### 4.1.2 New Query to a Known Knowledge Base

Instances of this query class are as well new to the system. The difference is that the knowledgebase they reference is already known to the system, i.e. at least one reasoner has already loaded the knowledgebase. Having a reasoner, which has already loaded the knowledgebase and the implied fact the already information of the particular knowledgebase has been retrieved due to the earlier, at least one query offers a lot of possibilities for optimization.

Knowing which reasoner already has loaded the knowledgebase offers the advantage to exclude the possibility that the reasoner which answers the query will have to load the knowledgebase in order to be able to answer the query. As the loading of a knowledgebase always adds on the response time of a query (depending on the query and the size of the knowledgebase the time to load a knowledgebase may exceed the answering time for a query many times) it is always preferable from the perspective of the current query to chose the reasoner that has already loaded the knowledgebase to answer the current query. From the perspective of the whole system it might not always be favorable.

Moreover the fact that already one ore more queries with reference to the same knowledgebase as the current query have been answered offers the possibility for optimization. Instances of this query class offer to profit from the already reasoned information by exploiting subsumption relationships of the current query with previous queries to the same knowledgebase.

### 4.1.3 Known Query to a Known Knowledge Base

In addition to the previously explained query class, queries that are instances of this class are already known to the system, i.e. they have already been answered completely or in parts for another client. This offers even more effective possibilities for optimization.

To allow a current query to profit from an earlier query most effectively, the semantic middleware has to implement a cache. Answers that have been reasoned for earlier queries can be used to answer the current query as another equal query to the reasoner will have to result in equal answers. In the case that the requested number of answers of the current query does not exceed the size of the answer bundle in the cache the whole query can be answered without involving any reasoning.

### 4.1.4 Continuation Query

Instances of this query class are special cases of the previously described class of known queries. These queries are as well known to system as the previous

class of queries but they come with information about an existing session which they want to pick up. This feature was described in the previous chapter as *iterative query answering*.

Besides a session register to support the feature, the system should implement, as it was already argued in the previous paragraph, a cache to let as well continuation queries profit from previously gathered information of other queries. However this will add more complexity to the cache.

The important advantage of instance of this class over instances of the known query class is that no expensive search for the right entry in the cache has to be made. The right cache entry can be found by letting the session have a reference to it. This is a further step of optimization.

## 4.2 Cache Usage

As it was argued in the previous paragraphs, reasoning can be considered to be an expensive task, requiring considerable system resources and time. The usage of a caching mechanism for inferred knowledge offers for some query classes the way to avoid reasoning in parts or even completely, but still being answered correctly.

Most modern reasoners already implement efficient caching mechanisms for inferred knowledge, which can speed up the answering of queries that have been answered before by the particular reasoner. However the introduction of a complete reasoning infrastructure with a central middlelayer that mediates between the clients and the reasoners offers a much more efficient way to cache the inferred knowledge.

**Faster Response Times for the Current Query** The usage of a cache allows a query to be answered completely without the involvement of a reasoner. In the case the query can be classified as a *known query* or a *continuation query* the query was answered by the system before and in consequence the answers are already contained in the cache.

Due to the support of *iterative query answering* the set of answers in the cache is not necessarily complete and thus may not contain all the answers that are requested by the current query. In this case it might be possible to get the requested answer partly from the cache and instruct a reasoner to deliver the missing number of tuples.

However the more popular a particular query is the higher is the probability that all answers will be contained in the cache. The cache definitely ensures that all answers to a query will have to be inferred only once (in case the cache does not expire).

By placing the cache in the middlelayer it can be ensured that all queries that reach the system can profit from the inferred knowledge without being

dependent on the decision of the load balancer and thus dependent on the cache of the particular reasoner that is instructed to answer the query.

**Freeing Resources** Another advantage of caching the inferred knowledge in the middleware is the possibility of total exclusion of a reasoner in the response process. This lets other queries benefit from free resources that would have been otherwise blocked by a *known query* or *continuation query*.

### 4.2.1 Safeguarded the Order of Answers

An important prerequisite for the successful usage of a cache in the middle-layer is that the cache ensures the genuine order of the answers. In praxis this boils down to the fact that the cache makes sure that the answers it returns to to the client are in the same order as if they came directly from the reasoner. This prerequisite is important for two reasons.

Firstly most of the reasoners are non-repeating servers. These reasoners ensure that in a response collection there are no duplicate answers. Moreover a reasoner can be called terse when it ensures that none of its answers is redundant. An answer can be called redundant, if it subsumes another answer in the response collection. It is expected that most applications will require the servers to be serially terse [14], thus the semantic middleware has to support this feature as well, to be able to provide the same services as a single reasoner.

Secondly a requirement for the semantic middleware was to support *iterative query answering*. This requirement can only be supported if the cache safeguards the order of the answers as explained before. Otherwise it might be possible that some answers come up more than once, when the next answer bundle is requested from the system. Besides the right order the cache has to integrate a kind of session register in order to know which query and was referenced by a particular session and to know how many answers already have been returned from the cache.

## 4.3 Exploitation of Subsumption Relationships

In the previous chapter the basics about DL reasoners and their way of knowledge representation were given. The reasoner are able to compute taxonomies for the TBoxes and ABoxes of their loaded knowlege bases, ordering the concepts and assertions in a subsumption hierachy.

DL reasoners like RacerPro[5] can also classify queries in a subsumption hierarchy. This offers instances of the query class "New query to a known knowledgebase" two mutually exclusive ways to profit from previously answered queries and thereby reduce their response times.

It should be noticed that although this way of optimization involves a step of reasoning, the time spent on the computation of the query subsumption is ignorable short compared with the time needed for query answering, especially for complex queries and large knowledge bases.

### 4.3.1 QBox

A reasoner like RacerPro can be instructed to keep track of the queries to its loaded knowledge bases. For this task the reasoner holds a query storage unit, the QBox.

Based on the taxonomy of the TBox, the QBox sorts the queries in a subsumption hierarchy. The query that asks for assertions of a concept $C$, with $D \sqsubseteq C$, subsumes the query that asks for the assertions of the concept $D$.

**Example.** If query $Q_{student}$ asks for all individuals that belong to the concept *student* and query $Q_{graduatestudent}$ asks for all individuals of the concept *graduatestudent* then it applies

$$Q_{graduatestudent} \sqsubseteq Q_{student}$$

given that

$$graduatestudent \sqsubseteq student$$

### 4.3.2 Earlier Query Subsumes Current Query

The first possible way of subsumption is that the current query is subsumed by one or more previous queries. The reasoning process can profit from this relationship by reducing the search space for the current query to the answer set of the so called *parent query*. However this implies that the same reasoner answers the current query, which answered the parent query. A smaller search space can result in a much faster reasoning process, especially when the search space is significantly smaller than the complete TBox.

In the case that the current query is subsumed by more than one of the previous queries, the one with the smallest answer set, i.e. with the smallest search space for the current query is the preferred candidate. Thus that query's reasoner is to answer the current query.

### 4.3.3 Current Query Subsumes Earlier Query

Keeping the concept of the QBox in mind it is obvious that not only a previous query can subsume the current one, but also the opposite relationship can exist between the queries. Finding the current query subsuming an earlier query has an even bigger potential for optimization than the previously described one.

The answer bundle of a subsumed query is included in the answer bundle of the subsuming query. So if it is discovered that the current query subsumes an earlier query it can be stated that parts of the answer bundle of the current query have already been retrieved during the answering of the so called *child query*, or in other words, all answers of the child query are answers of the current query. In order to profit from these answers the usage of a cache is essential.

As the answers of the subsumed query are included in the answer bundle of the current query the idea is to let the current query profit from the cached answers in the middleware by first returning those. This offers a way to reduce the amount of needed reasoning, in case the current query is an incremental query it even offers the chance of answering the query without any reasoning at all. If the requested number of answers does not exceed the number of cached answers of the subsumed query, the current query can be answered completely from the cache. If it exceeds the number of cached answers, only the missing answers have to be requested from the reasoner.

In case more than one previous query is subsumed by the current one, the one with biggest answer bundle is to be preferred.The lesser the difference in the two search spaces, the more the current query can profit from the already cached answers as well as the whole system can profit from queries equal to the current and the subsumed one.

### 4.3.4 Subsumption Relationship in Both Directions

In the case that the current query both subsumes a query and is subsumed by another one the relationship of subsuming an earlier query should be exploited. Considering the previously made argumentation the relationship in this direction offers the possibility of profiting from the cache and even avoiding reasoning completely, which is the most desired outcome of all the preprocessing and optimization.

### 4.3.5 No Subsumption Relationship

If the current query neither subsumes any previous query nor is subsumed by any previous query, there is only a small possibility for optimization.

As the query references a known knowledge base at least one of the reasoners has already loaded this knowledge base. Considering the arguments that were given before it is preferable to let a reasoner answer a query that already has loaded the referenced knowledge base. Thus this information can and has to be exploited in order to avoid a waste of resources and shorten the current query's response time.

## 4.4 Forced Knowledge Base Distribution

Considering the requirements for reasoning systems that were given in an earlier chapter a basic concern of the system is to minimize the response times. Short response times have a positive influence on the scalability, the availability as well as on the quality of service. The distribution of the knowledge bases that are known to the system over the system's reasoners has an important influence on the response times. Thus it is important to have a closer look at this characteristic.

### 4.4.1 The Ideal Knowledge Base Distribution

The question about the ideal distribution of knowledge bases is a question of perspective. To illustrate the problem consider the following example.

**Example.** The given system includes 2 reasoners ($R_1$ and $R_2$) that are fully working, idle and prepared to answer queries. 2 knowledge bases ($KB_1$ and $KB_2$) are known to the system, which means the system only accepts queries to these knowledge bases. Besides that 2 clients ($C_1$ and $C_2$) are querying the system. Both are at first interested in information about $KB_1$ and then in information about $KB_2$.

In a first scenario each reasoner specializes in one knowledge base, i.e. $R_1$ loaded $KB_1$ and $R_2$ loaded $KB_2$. Both clients query both kbs. In the first step they both send a query about $KB_1$ to the system. In the system only $R_1$ has loaded $KB_1$. So the queries are put in sequence in the way they arrived and answered one after the other by $R_1$. In step 2 both received the answer and now request information about $KB_2$ at the same time. Again only one reasoner has loaded the knowledge base. This time its $R_2$. Both queries are again answered sequentially.

In this scenario there is no gain for the system having more than one reasoner working. Although 2 reasoners are working the whole system acts as if it would be only one, which answered the queries. The response time for $C_1$ can be considered to be $t_1$, while the response time for $C_2$ is $t_1+t_2$ - the time for waiting for $C_1$'s query being answered plus the time to answer the own query.

The whole outcome would have been different in the case that both reasoners had loaded both knowledge bases, which is the second scenario. In this case the two queries about the same knowledge base arrive again at the same time at the system. But this time both reasoners are able to answer the query. Thus $C_1$'s queries are answered by $R_1$ and $C_2$'s queries are answered by $R_2$, which means not sequentially but concurrently.

With regard to the response times this means that $C_1$'s response time still is $t_1$ per query. But $C_2$'s response time was reduced by $t_1$ to be $t_2$.

### Complete Knowledge Base Distribution

The example shows clearly that in situations when the majority of queries that reach the system refer to the same knowledge base, it is preferable to have more than one reasoner equipped to answer the query, i.e. have the particular knowledge base loaded. Otherwise the system will perform below its capabilities.

In a worst case scenario all queries at a particular moment that reach a system refer to the same knowledge base. In this scenario all reasoners have to have the referenced knowledge base loaded in order to use all capabilities of the system. It will ensure that no resource will be unused as many queries as possible will be answered concurrently.

If the situation changes and queries to other knowledge bases reach the system, the complete distribution of knowledge bases will ensure that the new queries will be answered as well in the fastest possible way, allowing the load balancer to chose from all reasoners in the system.

Thus from this perspective the ideal system makes every reasoner load every knowledge base to be able to answer every query at any time and giving the load balancer the optimal setting to chose from and distribute the load.

### Minimum Knowledge Base Distribution

An important fact that neither the example nor the previous perspective consider is the cost of loading a knowledge base. In many scenarios in which load balancing techniques will be applied, the referenced knowledge bases are initially unknown to the system and will be loaded by the reasoners on the fly. This loading consumes time which adds down the overall response time of the system, i.e. slows down the processes. The problem can be viewed on from two sides.

From the view of a single query that reaches the system it is preferable to be answered by a reasoner which already has loaded the knowledge base. Otherwise the loading will add an additional waiting time to the response time of the query, thus will extend the response time of this query. If already one reasoner has loaded the knowledge base it is therefore preferable to let this reasoner answer the query instead of letting another reasoner load the knowledge base and answer the query.

Another view on the issue is the overall system performance. Having another reasoner first load a knowledge base and then answer a query does not only block the current query from being answered but also blocks the reasoner from answering other queries. In a life-like scenario the reasoner has to be considered to have already loaded other knowledge bases and therefore is likely to receive queries to them. If it is now being scheduled to load the new knowledge base its resources are blocked to answer queries to other

knowledge bases. Having the reasoner load the new knowledge base will result in extended response times for other queries that reach the system at the same moment.

Besides the pure costs for loading a knowledge base the previous perspective and example may be considered not to be very likely, which would reduce the benefits of distributing the knowledge bases. In a real world system a situation where all queries at a certain point of time refer to the same knowledge base may not occur at all, and if it occurs it may occur very rarely. So the costs of loading a knowledge base may weigh more than the benefits in many cases.

Therefore the second perspective on the knowledge base distribution argues for a minimal distribution, i.e. each knowledge base is loaded by only one reasoner. Thus the costs of loading a new knowledge base will be reduced to a minimum.

### 4.4.2 The Optimal Knowledge Base Distribution

The optimal solution has to consider both perspectives and try to find a good balance in order to improve the system's performance and average response time.

The crux of this issue lies in the undeterminable pay-off - or from the other perspective the undeterminable costs - of loading a knowledge base. The pay-off is higher, the bigger the amount of concurrent queries to the particular knowledge base is in the future. The costs are lower, the less queries to other knowledge bases arrive at the same time at the system.

Thus the proposed way is to make use of idle times at reasoners to load knowledge bases and thus reduce the costs of doing so to a minimum.

#### Loading Knowledge Bases during Reasoner Idle Times

The idea of this optimization criteria is to recognize idle times at reasoners. If for a certain amount of time the queue of a reasoner has been empty, the estimation is that as well in the coming near future there wont be any queries for this reasoner due to a general unpopularity of the loaded knowledge bases, which means the reasoner is and will be idle. Letting this idle reasoner load a new knowledge base - preferably one that is heavily queried to create the biggest relieve for the whole system - resolves the two basic problems that were given during the description of the minimum knowledge base distribution.

The first problem of extending the response time of the current query which references the new knowledge base is resolved by letting the reasoner load a new knowledge base without having a query waiting for it. No query to the knowledge base that is to be loaded will recognize any difference during the loading process.

The second problem was the interference with queries to other knowledge bases. As it is only estimated that the idle reasoner will not receive any query and thus will be also idle in the near future the costs of having a query to another knowledge base wait cannot be ruled out completely. But they are reduced to a minimum, which can be considered insignificant in contrast to the benefit that will originate from it. This benefit also originates from the choice of the right knowledge base, which will be discussed in the following paragraph.

### 4.4.3 Choosing the Optimal Knowledge Base

Many scenarios in which load balancers for reasoning systems are applied include the querying of many different knowledge bases, or at least the possibility for that. It is therefore very likely that in the situation when a reasoner is idle, many knowledge bases are known to the system but only a few have been loaded by the reasoner. As it is the aim of distributing the knowledge bases to shift some load to the idle reasoner, it is unlikely and desired that the reasoner is not idle anymore after loading the knowlegebase. Thus it can not load all knowledge bases it is missing at the same time and has has to be make a decision for the optimal knowledge base to load.

This decision for the optimal knowledge base to load has to be based on the various constraints that will be explained in the following.

#### Knowledge Base with the Biggest Impact

As mentioned in the beginning of the paragraph the aim is to have all reasoners load all knowledge bases. But this is not an end in itself. It is more a means to achieve an equally as possible distribution of load over the reasoners in the system. Having an equal distribution the system is prepared for every coming query in the same way and thus no query has to take the burden of a previously made false decision. In consequence the optimal choice of a knowledge base to load is the knowledge base which will have the biggest positive impact on the equality of the load distribution. The important question to answer is, which are the factors that determine if a knowledge base has a bigger positive impact than another one.

**Load generated by a knowledge base.** Important to consider is the load that has been generated by a particular knowledge base. The generated load consists of the number of queries that referenced the knowledge base, weighted with the actual processing time the query needed to be answered. Consequently a forced distribution of a particular knowledge base affects more load if the knowledge base causes a lot of load on the system. Thus the knowledge base causing the highest amount of load is preferable.

**Distribution of a knowledge base.** Besides the actual load that is caused by the queries, the distribution of the knowledge bases is important as well. The waiting time caused by a knowledge base's queries to other queries as well the waiting times caused by those queries' to the queries of the knowledge base in question have to be considered. Consider the following example.

In a system with three reasoners $R_3$ goes idle. It now has to be decided which knowledge base that $R_3$ has not yet loaded but $R_1$ or $R_2$ have is to be loaded by $R_3$. $R_1$ has loaded $KB_a$ and $KB_b$, while $R_2$ has loaded $KB_c$. The analysis of the load caused by the different knowledge bases resulted in 40% of the load for $KB_a$, 20% of the load for $KB_b$ and 40% of the load for $KB_c$. As the load caused by $KB_a$ and $KB_b$ is equal they are both equally preferable, regarding the arguments of the previous paragraph.

But this is not true. The difference is made by the second knowledge base loaded on to the reasoner $R_1$. Due to the other queries that are answered by $R_1$ the average waiting time for a query to $KB_a$ is higher then those to $KB_c$, as queries to $KB_a$ have to wait as well for the queries to $KB_b$ to be answered. This makes the average response time for a query to $KB_a$ longer than to $KB_c$. When $KB_a$ is distributed to $R_3$ it will not only cause parts of its load to move to $R_3$ and thereby speeding up the answering of queries to $KB_a$, but it will also cause the queue of $R_1$ to shrink and thereby have a positive effect on the average response time of queries to $KB_b$ as well by reducing their waiting times.

Thus the distribution of the knowledge bases also has to be taken into consideration in order to make the optimal decision.

**Load distribution after knowledge base is distributed.** The important third factor influences the impact of both of the previous factors and is at the same time virtually impossible to determine. The decision for the optimal knowledge base to load by the idle reasoner is based on scenarios about how the system, respectively the load distribution over the system, will look like after the knowledge base has been loaded. This prediction of the future depends on two variables.

Firstly it is unknown if the load caused by knowledge bases will be the same is in the very moment when a knowledge base is chosen. The best guess that can be made is that it is the same. Thus this will be anticipated.

Secondly it is unknown how much of the load will move from the reasoners that have already loaded the knowledge base to the idle reasoner. As the reasoner is idle and thus is not occupied by other queries it is very probable that a lot of the load will move to it. On the other hand, if a lot of the load is caused by cached or subsumed queries the loading of the knowledge base will cause less load to move to the new reasoner. How much this is exactly is therefore not possible to tell. Thus to be able to calculate scenarios a

percentage has to be anticipated.

#### Knowledge Base with Lowest Costs

While the previous considerations only focused on the benefit that the forced distribution of a particular knowledge base has, it is important not to forget about the costs that come with it. A knowledge base that takes less time to be loaded is to be preferred over a knowledge base that takes a longer time to load.

Firstly a faster loaded knowledge base makes the idle reasoner faster functional again and thus makes it have an earlier impact on the system. The earlier the system can profit from the reasoner having loaded the new knowledge base the bigger the benefit.

Secondly a the faster a knowledge base can be loaded, the less likely it is for the situation of the load distribution to change, the better the forecast will be.

Thus a less costly, faster to load knowledge base has to be preferred.

### 4.4.4 Systems with Initially Known Knowledge Bases

Systems that only allow queries to a particular set of knowledge bases that are known before hand are a special case. These reasoners in the system can be prepared before they start working by letting them load all the knowledge bases that will be referenced by the queries.

This way of handling the knowledge base distribution lets the system benefit from the advantages of having all reasoners load all knowledge bases without having to bear the costs of sacrificing processing time.

In the case that the total amount of knowledge bases exceeds the maximum number of knowledge bases a reasoner can have loaded at the same time a decision of how to distribute the knowledge bases has to be made upfront. This decision totally depends on the particular setting and needs of the system.

# Chapter 5

# Load Balancing Strategy

In paragraph 3.2 the arguments and requirements for a semantic middleware were listed and explained. It was argued that reasoning systems need a load balancer that implements a sophisticated strategy, basing the distribution decision on as much information as possible, due to complex scenarios and the high costs of reasoning.

This need for a sophisticated strategy becomes even more obvious and crucial considering all of the preprocessing for optimization that has been discussed in the previous chapter. The optimization is already part of the load distribution process. Each optimizer decides whether a particular query can be answered without reasoning or partial reasoning, whether there is a way for speeding up the reasoning process by choosing a particular reasoner or if there is nothing that can be optimized. Whenever the decision involves reasoning, the optimization module makes the decision for the reasoner that is considered to be optimal to answer the current query. But, this decision is made without further consideration of the current load situation.

Facing this issue the load balancer must implement a strategy that allows it to adapt to an always changing load distribution so that the load will be balanced over all connected reasoners. Static load balancing strategies like Round-Robin that do not take any information regarding the current load situation into account must therefore be regarded as not suitable for this kind of system.

Instead the Ant Colony Optimization algorithm provides an optimal basis for the development of a suiting strategy. Based on research on the behavior of ants navigating the landscape, this algorithm already serves as *AntNet* to route traffic through a network in a very efficient way. With adoptions to the algorithm it will provide this system with an effective means for load balancing.

## 5.1 Requirements

The sophisticated load balancing strategy that was demanded throughout the thesis has to fulfill the following requirements.

**Balancing Blind Load Distribution** The strategy must be able to cope with the blind load distribution of the optimization modules. The optimizers that were discussed in the previous chapter may chose a preferred reasoner based on their analysis. This is done under the premise that an optimizer's choice is always to prefer, disregarding the current load situation. But this premise does only hold if the load balancer is able to

1. minimize the unbalancing effect of the optimizer's blind choices, by building a basis for an even as possible load distribution, e.g. by having different reasoners load different knowledge bases.
2. cope with an always changing, dynamic load distribution, by considering even big inequalities between the single reasoners loads and recognizing such situations when they happen.

**Load Forecasting** With regard to the previous requirement and the history sensitivity, which was explained in paragraph 3.3.2, the load balancer needs not only to react to current load situation, but also has to keep in mind the future development of the load situation. In load balancing for reasoning systems the current decision always has an impact on the current and the future load distribution. Thus, to a certain degree, the decision has to be based on forecasting.

## 5.2 Ant Colony Optimization

### 5.2.1 The Ant's Way of Navigation

Early research on the behavior of ants showed that ants communicate among each other, or between individuals and the environment with the help of chemicals that they produce, so called *pheromones*. This is particularly important for some ant species to mark paths, e.g. from their nest to the food, so that other ants can sense these pheromones and follow the discovered path. This behavior was the basis for the research and one special experiment that was conducted by Deneubourg, Aron, Goss and Pasteels in 1990 [15] which built the basis for the Ant Colony Optimization (ACO).

#### Double Bridge Experiment

While walking from their nest to the food and back, ants place pheromones on the ground, forming a pheromone trail. Other ants can sense these

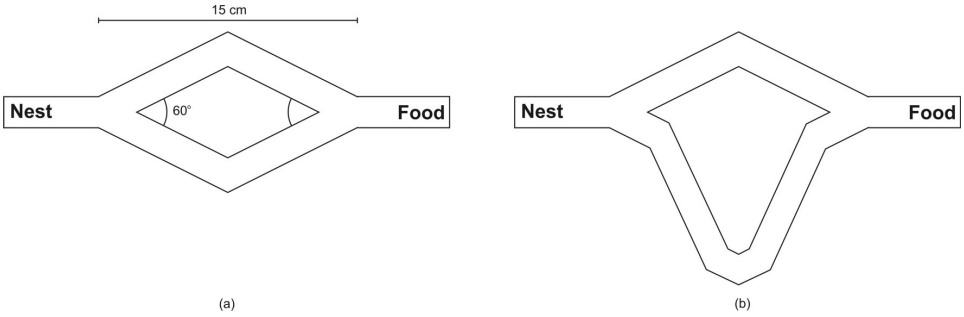


Figure 5.1: The two experimental setups for the double bridge experiment by Deneubourg and colleagues, 1990. [15]

pheromones and use them to navigate from to the food and back. It was found that the ants tend to choose, probabilistically, path with a high pheromone level over those with a lower one.

In order to investigate this behavior Deneubourg and colleagues conducted the an experiment using two bridges to connect the nest with the food. They ran the experiment with different settings, where in one setting the two bridges had exactly the same size, while in the other setting one bridge was significantly longer as the other (figure 5.1).

In the experiments with the first setup the outcome was that the ants had no preference and they selected with the same probability any of the branches. In one experiment one branch, in the other experiment the other branch, but in almost all cases one was preferred. This shows that the ants had no preference for one of the branches (in average over all experiments) but due to random fluctuations a few more ants selected one branch over the other. This made more ants follow this path and thus the they concentrated on a single branch.

In the second setting of the experiments the length of the second branch was doubled. In this case in almost all experiments the ants chose the shorter branch over the longer one. As both paths appeared to the ants identical the ants chose randomly so that half of the ants took the first branch and the other half the second branch. However the difference here could be seen on the way back. While the ants that chose the longer branch were still on the way to the food, the ants on the short branch were already on their way back, thus doubling the pheromone level on their path. Already the second time when an ant had to decide which way to take to the food the pheromone level was significantly higher on the shorter path, those increasing the probability that the ant choses this way over the longer. Still, some ants decided for the longer way. This can be interpreted as a type of "path exploration" [15].

For the further consideration it is important to notice that the actual length of a path is not the factor that directly influences the decision for one

or another branch. In fact it is the throughput that determines whether a path is to prefer or not. So it is the shortest way in terms of time needed to get from its beginning to its end.

### 5.2.2 Navigating Networks with Artificial Ants

Packet-switched networks use routing tables in the network nodes to guide data packets from their source to their destination. These routing tables hold information connected nodes which represent the shortest path to other nodes. One so called *link-state* algorithm used in network routing to create these tables is *Dijkstra's* algorithm. Whenever a path to another node is found it will be placed in the table, in case a shorter path is found the according value will replace the old one. Dijkstra's algorithm allows a packet to be routed on the shortest way through the network. However it does not consider traffic or load balancing on the net.

#### AntNet

Just like ants move along a path from their nest to the food, packets move along a network from their source to the destination. The idea of AntNet is to let the packets, or a special form of packets (artificial ants), reinforce shorter paths in a network with the use of artificial pheromones. Pheromone tables, replacing the existing routing tables in the network nodes, store information about the pheromone level of their network connections. The artificial ants update the pheromone tables when moving along their path through the network, like real ants drop their pheromones. The quicker the ants move along a path, the more throughput can be achieved. Thus the pheromone level for this path will be higher. The pheromone levels of each node will bias the probability for a packet to take the a particular path.

#### Pheromone Trail Evaporation

In experiments with the *Simple ACO*, an offspring of the ACO, it was found that the algorithm performs much better when the artificial pheromones *evaporate*. In fact a higher evaporation level is needed than that of real ant pheromones.

Evaporation here means, that earlier placed pheromones loose their impact over time, which could be done, e.g. by reducing the current pheromone level of all paths by 10% prior to the update with the new pheromone.

In the following explanation of the AntNet algorithm the evaporation will be done by the normalization step.

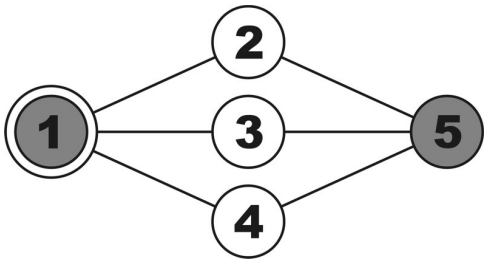


Figure 5.2: Example network graph for the AntNet algorithm

| Next Node | % chance |
|---|---|
| 2 | 33.33% |
| 3 | 33.33% |
| 4 | 33.33% |

Table 5.1: Pheromone Table for node 1, before first packet.

### 5.2.3 The AntNet Algorithm

To explain the algorithm, respectively the implementation in every node of the network, the problem will be reduced to a network with 5 nodes, shown in figure 5.2. Node 1 is the source and node 5 is the destination.

The pheromone routing table (table 5.1 in node 1 shows in the beginning equal levels for each of the paths. As the choice for a path is based on probabilities that are biased by the pheromone levels of each path, the table shows that each path has the same probability for being selected.

In this case node 2 is selected and the packet will travel along that path. The decision for node 2 leads to an update of the pheromone table in node 1, which consists of 2 steps.

1. Add 100% to the probability of node 2. The value will rise to 133.33%.

2. Normalize all values in the table, i.e. bringing the sum back to 100% - which means, with added 100% a division by 2.

Thus the updated pheromone table, table 5.2, reflects the last packet taking the path to node 2.

| Next Node | % chance |
|---|---|
| 2 | 66.66% |
| 3 | 16.66% |
| 4 | 16.66% |

Table 5.2: Pheromone Table for node 1, after first packet, before second.

This table will be then the basis for further packets that arrive at node 1. The probabilities with which a packet will move on from node 1 to node 2,3, or 4 to reach node 5 will now be biased by the weights in the table. The chance that node 2 will be selected by the next packet is now 4 times higher than for the other nodes.

## 5.3 Load Balancing with Artificial Pheromones

The ACO offers a promising basis for a load balancing strategy that suites the needs of the semantic middleware. While the problem of navigating through a whole network was reduced to a single node with its adjacent nodes in the explanation of the AntNet algorithm above, this situation basically reflects the problem of load balancing different reasoners.

Node 1 represents the load balancer. The queries arrive here and then have to be distributed to one of the reasoners (node 2,3,4). From there on the reasoner's response is forwarded to a feedback module or the front-end that creates the response to the client (node 5).

Before explaining the reasons for this algorithm being very well suited for the problem at hand, the basic difference has to be discussed. The ACO bases its routing decisions on the assumption that those paths that are preferred now, because they are the fastest ones, are as well the better choice for the next ant, packet or query. This is due to the fact that a single ant or packet is not able to block the whole traffic on that path, instead a path that offers a certain throughput can be assumed to do that as well for the next ones to come. This is totally different to the load balancing situation in reasoning systems.

Reasoners are, as explained earlier, only able to work on one query at a time. Thus once a query is send along a path it will block the following node (the reasoner) for a certain time, making this path less attractive for the following queries. This is a fundamental difference to the ACO. In fact it is kind of the reverse decision mechanism that has to be implemented here.

### 5.3.1 Balancing the Load with Reverse ACO

Like the AntNet algorithm, this load balancing algorithm is based on a pheromone table that is situated at the load balancer which keeps track of the load on each reasoner by updating the pheromone level for every query that a reasoner is instructed with.

#### Going for the Lowest Pheromone Level

As mentioned before, the basic difference for this strategy is to go for the minimal level in the pheromone table as this offers the biggest chance for a fast response.

Thus in this strategy there is no longer a need´to base the decision on probabilities, in order to have the queries do some kind of "path exploration" (to avoid being stuck due to a too high difference in pheromone levels). As the choice of the smallest level always comes in connection with an increase of this particular level. This system itself ensures that every path will be taken, thus the pheromone levels wont be used as biases for a probabilistic decision, but as basis for a clear, unambiguous choice for one particular reasoner.

#### Forecasting with Pheromones

The forecasting model of this strategy assumes that recently popular knowledge bases have a higher probability to become popular in the near future than those that have been unpopular in the past. The load balancer will involve this anticipation of the development of the load to improve its the overall load balancing performance.

**Reflecting All Scheduled Queries** All queries that are scheduled for a reasoner can be reflected by an increase of the corresponding pheromone level in the central pheromone table. Thus the load balancer can keep track of the load distribution and consider the impact of every query, including those that come directly from one of the optimizers, by considering only the situation reflected in a single, central table.

**Compensate Current Fluctuation** To understand the importance of the consideration of earlier queries for current load balancing decisions it is necessary to remember the *history sensitivity* of the reasoners which was explained earlier. Reasoners that have been very busy in the past, due to a very popular knowledge base they have loaded, are likely to become busy again when this knowledge base becomes popular again - especially due to direct scheduling of queries by the cache and subsumption optimizers (see 4.2 and 4.3). If the load balancer would base its decision exclusively on the current situation of the load distribution, there would be the danger of reacting to strong to an one time fluctuation, like a very steep increase in the load for a particular knowledge base that will never be queried again after this. Moreover it is very likely that information about recently heavily queried knowledge bases will be neglected and thus lead to misinterpretations of the current situation and wrong decisions.

**Compensate Legacy** It is also important to ensure that information about former load situations on the reasoners do not have a too big impact on the decision. A step-wise decrease in impact from young to old information would lead to the right balance for the forecast. The further

a query lies in the past, the lesser should its impact be. The normalization that was also performed in AntNet offers this effect. Due to the constant division by $1 + x$, with $x$ being the increase of the pheromone level for the current query, earlier pheromones loose their impact over time, yet very recent pheromones are still represented in the current pheromone level.

The very important factor in this forecasting model is the amount $x$ by which the pheromone level of a reasoner will be increased for a single query, i.e. how much more influence a current query should have on the decision of the load balancer than an older query. The higher the amount $x$ is, the higher the impact of the current query, and the faster older pheromones will evaporate, due to a likewise increase of the divisor $1 + x$.

The decision on the amount $x$ can not be made universally for all possible scenarios in which the semantic middleware might be applied. The use of a slow or fast evaporation will depend on the particular applications, and the situation in terms of query types that the system has to deal with preferably.

#### Different Increases for Different Queries

The pheromone table offers a way of reflecting even more information.

While one query involves only the retrieval of further answers to a query that had been partially answered before, others make a reasoner load a new knowledge base prior to answer the query. A a query referencing a new knowledge base might cause 4 times the load a continuation query that comes from the cache causes, thus this should be accounted for by increasing the pheromone level 4 times higher then the continuation query does.

The load, a particular query will cause on a reasoner, will vary significantly from query to query. In order to give the load balancer a accurate picture of the load situation this has to be considered. As it is impossible to determine for each single query in advance how much load it will cause, a convenient way to handle the issue is to classify the queries. Each class of queries will have assigned a reasonable value, reflecting the assumed load compared to the other query classes.

Such a classification was already introduced in the beginning of chapter 4. The further use of this classification would be suitable and convenient.

#### Dynamic Query Load Update

Introducing different pheromone level increases alone does not solve the issue. The problem still is how to define the ratios of the different query types to each other.

One way to introduce the ratios is to make the measurements and set the ratios upfront. The problem with these values is that (i) they may not

reflect the situation in the actual application scenario and (ii) it does not prepare for a possibility that these ratios may change over time.

Thus the best solution will be to dynamically change and adapt the ratios to the real average amounts of load the queries cause. As the load a query causes can be measured in the time it needs to be answered, each query will be measured so that an average time can be calculated for every query type. These average values will than be compared and the corresponding ratios will be determined.

The proposed algorithm will in the following be called *ACO-LB* (Ant Colony Optimization - Load Balancing).

# Chapter 6

# System Implementation

In order to support the many assumptions and theories that have been developed throughout the thesis - above all that the introduction of a semantic middleware creates a benefit -, a system has been implemented to provide the basis for empirical testing of the assumptions and modules.

In the following the the system architecture and the implementation of each mayor module and paradigm will be described and explained.

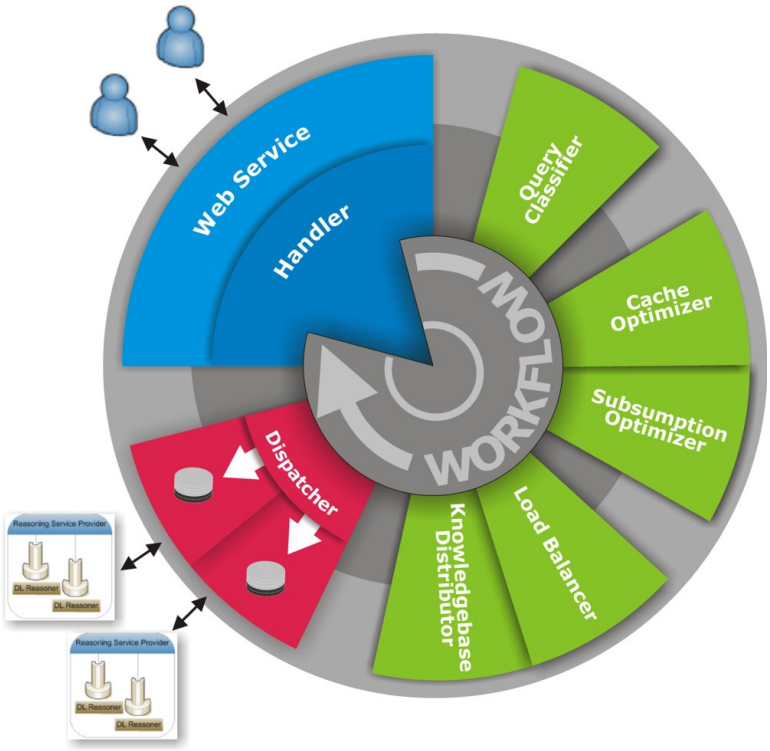


Figure 6.1: Enhanced Architecture for Semantic Web Middleware

## 6.1 RacerManager

The system is based on the open source system RacerManager[1] which has been developed by a workgroup of the STS department, University of Technology Hamburg. RacerManager acts as an OWL-QL server (henceforce called OWL-QL server) and forwards the queries to a local or remote DL reasoner. The used RacerManager implemented components to connect to the RacerPro reasoner, which was used as DL reasoner. Therefore it provided as well a translation module *OWL-QL to NRQL* to correnspond with the RacerPro instances.

This OWL-QL server is implemented in Java, intended to run as application in the widely used application server Apache Tomcat[8]. It further implements well known components such as the Apache Axis Web Services Framework[7], XMLBeans Framework[11] and Jena Semantic Web Framework[10].

The given RacerManager is implemented in a commonly used n-tier architecture, which makes it fairly easy to modify modules and offers a clear separation of responsibilities.

Racer Manager already includes a basic cache and load balancing strategy, which is kind-of similar to Round Robin. On this strategy the dispatching to the particular reasoner instances is based. Both of these modules have been altered during the implementation to fit to the changed requirements, thus they will be explained more detailed in the following.

Besides the feature for incremental query answering, this OWL-QL server also support premise queries. These queries can be seen as a special case of a "knew query", and thus wont be discussed further.

## 6.2 Workflow Assignment

A mayor change that had to be made to the system was the introduction of *workflows*. In chapter 4 various query classes were introduced. Part of that was the idea to minimize the preprocessing for each query as far as possible by determining as exact as possible its chances for optimization.

In consequence every query gets a workflow attached, which exactly states the used modules and the order in which they are applied to the query.

In the following the *classifier* module, which classifies a query based on certain parameters, followed by a description of the four workflows.

[1] RacerManager can be found at: http://racerproject.sourceforge.net

### 6.2.1 Classifier

After being received and processed by the Web service interface, the classifier is the first module in the OWL-QL server that works on the queries. Based on the proposal given in 4.1 the module categorizes the queries into one of the four disjoint classes (i) first query to an unknown KB; (ii) new query to a known KB; (iii) known query to a known KB; (iv) continuation query.

The procedure that is implementing by the classifier is shown in algorithm 1.

**Algorithm 1** *classify(query)*:

```
dialog := null
worklflow := null

if continuation_query(query) then
  worklflow := createContinuationQueryWorkflow()
else if ¬referenced_KB_loaded(query) then
  worklflow := createFirstQueryToUnknownKBWorkflow()
else
  dialog := searchDialog(query)
  if dialog ≠ null then
    worklflow := createKnownQueryToKnownKBWorkflow()
  else
    worklflow := createNewQueryToKnownKBWorkflow()
  end if
end if

assignWorkflowToQuery(query, worklflow
```

### 6.2.2 Workflows

#### Continuation Query

A continuation query comes along with a *process handle* which identifies the session that contains all valuable information to enable the incremental query answering. Thus the workflow first retrieves the session.

The session contains a reference to the dialog which contains the already retrieved data. Besides that the session keeps track of the number of already send answers. This way it is possible to not send the client any of the answers twice, and to decide when the client has received the complete set of answers.

In order to retrieve the requested number of answers, the cache optimizer will be invoked and provided with the information it needs to retrieve the next set of anwers.

After receiving the response from the cache optimizer this workflow will place the answers in the answer queue of the particular client where it will be picked up and processed to be returned to the client.

The corresponding Algorithm 6 can be found in appendix A.

### Known Query to a Known Knowledge base

The workflow of a known query is very similar to that of a continuation query. However, this query does not provide a process handle. Thus the query has to be compared to the queries that are referenced by the dialogs in order to find the matching dialog that contains the information to answer the current query.

When the dialog is found, the same procedure as in for previously described workflow is implemented, i.e. the subsumption optimizer is invoked and the received response is placed in the clients answer queue.

The corresponding Algorithm 7 can be found in appendix A.

### New Query to a Known Knowledge base

The workflow for a new query to a known knowledge base consists of three steps. In the first step the subsumption optimizer is instructed to find an earlier query that is subsumed by the current one. In the case one is found a new *subsumption dialog* is created in the cache, which holds a reference to the dialog of the subsumed query. Further information about the implementation can be found at 6.2.3 and 6.2.5. In the case no earlier query was subsumed the second step follows.

In the second step the subsumption optimizer is instructed to find earlier queries that subsume the current query. If one is found, the query gets immediately assigned the same reasoner as the subsuming query in order to let it profit from a smaller search space. In the case that also this relationship could not be found, the third step will be executed.

In the third step the query will be handed over to the load balancer that is implemented in the system, which will take over from their on.

The corresponding Algorithm 8 can be found in appendix A.

### Query to an Unknown Knowledge base

The workflow for a new query to a new knowledgebase is very straight forward as it does not offer any potential for optimization.

In order to keep the subsumtion optimizer updated, the first step in the workflow is the instruction of the subsumption reasoner to load the new knowledge base.

In the next step it just has to create a session and a dialog in the cache, before it then is handed over to the implemented load balancer.

The corresponding Algorithm 9 can be found in appendix A.

### 6.2.3 Cache

The cache implementation of the new system is based on the cache of RacerManager. The OWL-QL server implemented the cache in the way that all results that were gained through reasoning were cached. This is as well the case for the new implementation.

The cache consists of four different registers, that keep track of the information. They will be explained in the following. In parts these registers did already exist in the previous system. In order to have them more efficiently work together they were integrated into one cache component

**Knowledge Base Register** The knowlegde base register contains information about all the knowledge bases that are known to the system, i.e. they have been referenced by at least one query that was handled by the system.

**Dialog Register** The dialog register contains the inferred knowledge. The first time a particular query reaches the system, a dialog is created where information about the query, the results and further information about the communication are stored.

The further information includes a flag that indicates if more information can possibly retrieved from the reasoner or if definitely all answers have been received. In order to be able to retrieve further results for a query, the dialog holds as well a query identifier, which unambiguously identifies the particular query on a particular reasoner. Consequently the dialog holds as well the information about which reasoner answered the query.

The expression dialog is taken from the OWL-QL specifications[13]. Here a dialog contains the information of a conversation between a particular client and a reasoner. By introducing the cache as a kind of new layer for the queries, the cache became the client for all reasoners, thus a dialog always happens between the cache and a reasoner. Thus this implementation lets more than one client profit from the inferred knowledge that is contained in a dialog.

**Session Register** As the system supports incremental query answering, it must be able to identify a particular client, when the client asks for more answers. Thus the OWL-QL server holds a session register, which keeps track of a list of session identifiers with attached information about the dialog that contains the information and the number of already sent answers. This session identifier, the so called *process handle* is returned with every query that offers more answers than having been retrieved yet.

**Query Register** The query register had to be introduced to support the subsumption optimizer. It brings together query identifiers and dialogs.

More information on this will follow in paragraph 6.2.5

#### Completed Answer Set

In order to improve incremental query answering, a little trick was implemented into the querying procedure. The cache always requests one answer more from the reasoner than it is needed to provide the client with the amount of answers he actually asked for.

When the cache was not differently told by the reasoner, it knows that there are still more answers to get from the reasoner. Thus the system sends a *process handle* along with the answers to allow the client to request further answers. If the cache receives less answers than the requested, the system sends a so called *termination token* along with the answers. This tells the client that all answers have been delivered. However, there is a problem with the case that the cache asks for the exact number of answers that can still be inferred. The cache will then conclude that there will be more answers, as it was not told differently, and will tell the same to the client. When the client then sends another query, to receive further answers, only an empty answer set will be returned.

In order ensure the client that it will only receive a process handle, when there are more information to retrieve, the cache asks for one more answer than needed, and thus ensuring that at least one more answer can be returned.

#### Cache Maintenance

Caching can result in extraordinary resource consumption. Thus intelligent cache management strategies are required. However, in the test scenarios the used resources are very small and therefore any cache expiration method was turned off or excluded in order not to interfere with the test results.

### 6.2.4 Cache Optimizer

Due to the growing functionality the cache optimizer (CO) was introduced to provide a good separation between the data - stored and managed by the cache - and the business logic that worked on the data, which was mainly placed in the CO, parts of it as well in the sumbsumption optimizer.

The CO basically implements the one central function of answering a query that is known to the system. Algorithm 2 describes in detail how the method is implemented.

The CO first tries to answer the current query from the cache by extracting the number of requested answers from the corresponding dialog. In case the dialog did not contain enough answers and it is not terminated - which would mean that the reasoner does not have further answers either - more answer are requested from the reasoner. For this information are taken from

**Algorithm 2** *getResponsesFromCache*(*dialog*, *answerSize*, *retrievedAnswers*):

```
answers := cache.getAnswers(dialog, answerSize, retrievedAnswers)

if answers.size() < answerSize then
  if ¬isTerminated(dialog) then
    missingNumberOfAnswers := answerSize − answers.size()

    query := dialog.getQuery()
    query.setAnswerQueue(thisCacheOptimzerQueue)
    reasonerQueue := getReasonerQueue(dialog)
    reasonerQueue.put(dialog.getQuery())

    // wait until reasoner places answer in the answerQueue
    reasonedAnswers := thisCacheOptimizerQueue.take()

    answers.add(reasonedAnswers)
    cache.addToDialog(dialog, reasonedAnswers)
  end if
end if

return answers
```

the dialog about the which reasoner answered the query and which identifier was provided by this reasoner to retrieve further answers.

In a second step the answers from the cache and the answers from the reasoner are combined and returned to the workflow. The initiation of an update of the cache is made by the *feedbacker module*. It is an extra thread working concurrently to the system, taking the information that were returned from the reasoner and caring for updates of all kinds.

### 6.2.5 Subsumption Optimizer

The subsumption optimizer implements the theoretical description and discussion of paragraph 4.3. The two ways of benefiting from a subsumption relationship introduced: either by a child or by a parent relationship.

As it was argued as well in the theoretical description, the discovery of a child relationship offers a bigger potential for improvement than a parent, i.e. finding a query that is subsumed by the current one is better than the other way around.

This argument is reflected in the workflow for a "new query to a known knowledge base", where the subsumption optimizer is invoked. The workflow puts the two methods in the order of first checking for a *subsumee* (a child) and then for a *subsumer*(a parent).

**Algorithm 3** *getSubsumee(query)*:

```
subsumee := null
children :=

children := subsumptionReasoner.getChildren(query)
if children.size > 0 then
  if children.size > 1 then
    subsumee := getReasonerWithLowestPheromoneLevel(child)
  else
    subsumee := children0
  end if
end if

return subsumee
```

### The Subsumption Reasoner

The system uses the capabilities of a local reasoner instance. Like the other reasoners that are managed by the system, this one is a RacerPro[5] reasoner. The reasoner offers the possibility of maintaining a query repository, the so called *QBox*. Here, based on the TBox and the ABox of the knowledge base, all queries are reflected in a subsumption hierarchy. THe QBox can be accessed to determine the successor or ancestors of a query, in particular its children and parents.

In order to offer the subsumption optimzation to every new query, the subsumption reasoner has to know about every query that was received and answered by the system. Thus the reasoner has to load every knowledge base that is known by the system, and has to add each of the queries to its QBox.

### Finding the Relative

The algorithms to find a child or a parent of a query are virtually equal. In the implementation they only differ from each other by one query to the reasoner. Thus, only the algorithm to find the subsumee will be described, exemplarily for both. It is described as well in algorithm 3.

The first step is to instruct the subsumption reasoner to retrieve all child-queries from the QBox for the current query. The reasoner will return a list of child elements which is either the bottom-query, contains one child or contains more than one child.

In case of the bottom-query it is clear that there exists no child-query for the current one. The situation is as well very clear in case that only one query was found to be the direct child of the current query. In the case that more queries are the children of the current one, a decision for one of the

queries has to be made.

**Most existing answers** One way of making the decision is to consider the answers that are already contained in the cache. As the current query is intended to be answered by the cached answers of its child query, this might be the best decision from the current query's perspective.

**Smallest Load on Server** Having a mature system, probably implementing a knowledge base distributor, it is likely that the found children have been answered by different reasoners. As it must be assumed that the number of answers that can be provided by a child will not be sufficient to satisfy the current client, there is a high probability that it will involve reasoning as well to answer the current query. Thus it makes sense to consider the load situation on the reasoners by which the particular children were answered. Choosing the one with the currently lowest load must be preferred.

Besides speeding up the answering process for the current query, such a decision will also help to balance the load.

The implemented system takes the second option and considers the load situation as more important. It is combined with the implemented load balancing strategy: ACO-LB. Therefore the decision is based on the existing pheromone level of the reasoners. The child-query which references the reasoner with the currently smallest level will be chosen.

### Cache Item Chains

In order to have the queries with a child-subsumption further profit from this relationship, the cache was extended to reflect it. When a child-subsumption is discovered a new subsumption-dialog is created which contains a reference to the answers of the child query. When the current query, respectively later, equal queries, are answered, the cache will first take the answers from the child-dialog, and then access the own answer tuples in the own dialog.

This creates two important advantages.

**Cache Efficiency** By referencing other answer tuples, these information do not have to be copied into the answer set of the new dialog. Thus the redundancy of stored data in the cache is reduced to a minimum, the cache becomes more efficient and does not waste resources.

**Answer Set Fill-up** Due to the referencing the current query can profit from the answers in the subsumed dialog. Moreover it also benefits from later queries to the subsumed dialog, which may further fill up its answer set. Thus, the next time a clone of the current query is received, the inferred knowledge for its own dialog grew with it. The same is true for the other way around.

> In order to ensure the right order the system was implemented so that always first the subsumed dialog has to be filled up, before answers that are specific to the own dialog will be retrieved from the reasoner.

But the whole implementation with references comes as well with a drawback. Theoretically dialog-chains of arbitrary size can develop. The worst case scenario would be that a client wants to have all answers to the broadest query (which subsumes all the others) and none of the subsumed dialogs have been completed. One after another the whole chain would have to be filled up, starting with the most specific. Each of these fill-ups would involve an interaction with the reasoner, thus causing an arbitrary amount of load on this machine and blocking the system probably totally.

To avoid that situation a safeguard had been implemented which ensures that always when a dialog is subsumed by another, the subsumed dialog checks (in case it subsumes as well another dialog) if this dialog if filled up completely. If it is not, it triggers a query which will cause the dialog to be terminated.

#### Query Modification

The idea of the subsumption optimizer is to let queries profit from already existing answers in the cache. If it now happens that further answers have to be retrieved from the reasoner, the dialog that asks for more answers does not want to receive as well those that are already contained in the answer set that it subsumes. Thus the query has to be modified, excluding the answers of the subsumed query. The implementation has shown to be very straightforward.

It should be mentioned that in case of a dialog chain still only the answer set of the immediately subsumed dialog has to be excluded as this answer set contains as well those answers of the dialog itself subsumes. This automatically prohibits the queries from becoming arbitrarily complex.

### 6.2.6 Knowledge Base Distributor

The *knowledge base distributor* is implemented in the system as an own thread, acting autonomously from the rest of the system. It is triggered by one of reasoners placing a request in its input queue.

Following the argumentations of the theoretical analysis and considerations of section 4.4 the most important part in the set of tasks of the knowledge base distributor is the identification of the knowledge base that is best suited to be loaded by the idle reasoner.

An important consideration in this context is the definition of load that builds the basis for the decision.

**Algorithm 4** *getKBToLoad(reasoner)*:

```
kbCandidates := null
scenarios :=
kbCandidates := getAllKBsNotLoadedByReasoner(reasoner)

if kbCandidates ≠ null then

  for each_kbCandidates do
    newScenario := createScenario(pheromoneTable, reasoner)
    newStandardDeviation := getStandardDeviationOfReasonerLoad(newScenario)

    scenarios := scenarios∪(newStandardDeviation, current_kbCandidate)
  end for

  return getKBWithLowestStDeviation(scenarios)
else
  return null
end if
```

### Definition of Load

**The common view on load** on a computing system is that the load corresponds to the work that the system has to perform. The work can generally be seen as a series of calculations, each of which causes a fixed amount of work *w0*. The work, or load that a single request causes therefore must be a multiple of the basic work unit *w0*.

$$w_i = n_i \times w_0 \tag{6.1}$$

The abstract term *work* can also be expressed in a better graspable manner. Due to the systems fixed processing frequency, the performed work can be expressed in time as they are in a linear relationship to each other. The work *w0* for a fixed amount of work corresponds to a particular time that is needed to perform it.

$$t_0 = w_0 \times 1 \div f_r \tag{6.2}$$

With the frequency being constant it can be stated that

$$t_0 \equiv w_0 \tag{6.3}$$

This means that the load on the systems corresponds to the time that is needed to perform the work that caused the load. Thus the load is equivalent to the time.

#### Finding the Optimal KB to Load

In order to determine the optimal KB the implementation of the knowledge base distributor follows the ideas and argumentations of section 4.4 very closely. The basic algorithm is shown in Algorithm 4.

The decision for the optimal KB has to be based on the load situation that will exist after the KB has been loaded. Thus for every knowledge base that is a candidate for this reasoner, as it has not yet been loaded by it, a scenario will be created, simulating the load situation after the particular knowledge base would be chosen. At this point it should be mentioned that the whole implementation of the knowledge base distributor is very much related to the ACO-LB load balancing algorithm, e.g. the pheromones are used to calculate the load scenarios.

An essential assumption that has to be mentioned in this context as well is that the system assumes that 50% of the all over existing load for the particular knowledge base will shift in the near future to the now idle reasoner, due to its low load situation.

Once the scenarios are created, the standard deviations (see next paragraph) will be calculated and compared. The one with the smallest standard deviation will be chosen to be loaded.

#### Standard Deviation of Reasoner Waiting Times

As explained at the end of section 4.4 the decision for the optimal KB is all but trivial. In order to account for the load that a particular knowledge base creates, on the other hand for the positive impact that the distribution of another knowledge base has on the waiting times of the other queries and its own, the following idea was developed.

The so called waiting time for each reasoner is calculated as the sum of all waiting times experienced by the queries of each of the loaded knowledge bases. Then, in the next step the standard deviation of all reasoner waiting times will be calculated.

The reason to chose the one scenario with the smallest standard deviation is the fact that a small standard deviation means very small differences in the waiting times of the reasoners. Very small differences again mean that the load on each of the servers is fairly equally distributed, i.e. balanced, which is a primary goal of the load balancer in order to be able to provide every next query, independent of how it looks like with the same quality of service.

### 6.2.7 Load Balancer

The load balancer module implements the proposed ACO-LB algorithm of chapter 5. Based on artificial pheromones - which are represented by double values - the load situation will be reflected in an internal pheromone table.

This table keeps track on the load situation of all of the reasoners as also of the load that was caused by a particular knowledge base.

### Distribution Decision

The load balancer makes its decision on where to forward the current query by comparing the pheromone levels of all reasoners. The pheromones are implemented in the way they were proposed in 5. The reasoner with the lowest pheromone level can be considered to be the least busy one, thus the query will be scheduled with that reasoner.

The load balancing strategy also offers a way not to consider all but just a given set of reasoners. If a new query to a known knowledge base does not profit from any of the subsumption relationships, the load balancer has to decide where to place it. As the load balancer still implements the assumption that a query will be faster answered when the KB has already been loaded by the reasoner, it will perform a comparison just over the subset of servers that have loaded the KB.

### Pheromone Update

An important part of the strategy is the update of the pheromone table for every query that is queued for a reasoner. The algorithm that is implemented to perform this task is Algorithm 5. As it can be seen the algorithm first updates the level of the reasoner that was just instructed to answer the query. In a second step the algorithm normalized the whole table back to 1. This is the implementation of the pheromone evaporation that was discussed in 5.2.2.

**Algorithm 5** $pheromoneUpdate(currentReasoner, newPheromone)$:

$oldPheromoneLevel := getLevel(currentReasoner)$
$newPheromoneLevel := oldPheromoneLevel + newPheromone$
$setPhermoneLevel(currentReasoner, newPheromoneLevel)$

$all\_Pheromones :=$
$all\_Pheromones := getAllPheromones()$
**for** $all\_Pheromones$ **do**
  $currentPheromone := pheromone\_i/(1 + newPheromone)$
**end for**

### Pheromone Adjustment

The load balancing system tries to optimize itself during the operation. In order to perform the pheromone increases for each type of query in the correct way, the processing time for each query is measured and in a feedback

method compared to the times of the other query types. Doing so the system will be able to provide the right ratios for each of the query types after a certain time of operation. Besides that, this implementation offers a possibility for the system to react on certain changes.

## 6.3 Queuing

The existing system was altered by implementing more concurrently executed threads that communicate with each other by the means of queues. For example, each of the reasoner implements its on queue and thus allows more than one query waiting for the reasoner become idle.

## 6.4 Test Framework

The test framework, which will be explained in detail in chapter 7, is based on the opensource load and performance measurement tool *Apache jMeter*[9]. JMeter performs the load tests on the specified systems based on the instructions that are contained in so called test plans. These test plans are XML documents that are structured in a certain way so that it can be interpreted by jMeter.

The normal way to create such test plans is by the use of the jMeter GUI application that allows to assemble parallel clients in combination with different standard components (like a HTTP request sampler) quiet comfortable. However, during it turned out pretty fast that assembly of test plans for OWL-QL queries was much less straight forward than assumed. Therefore it was decided to implement an own GUI and query assembling application that was tailored toward the convenient assembly of OWL-QL queries into multi-user test plans that could be executed by jMeter.

### 6.4.1 jMeter Query Assembler

#### The View

The *jMeter Query Assembler (jMeterQA)* was implemented as a full Java application, based on the model view controller *Apache Struts 2 Framework*[17]. jMeterQA runs on the application server Apache Tomcat[8].

#### The Model

As mentioned before the test plan that is loaded by jMeter is an XML document. The document contains all information about each of the used components: general information about the test, the number of clients and the queries that each of the clients should execute, as well as information

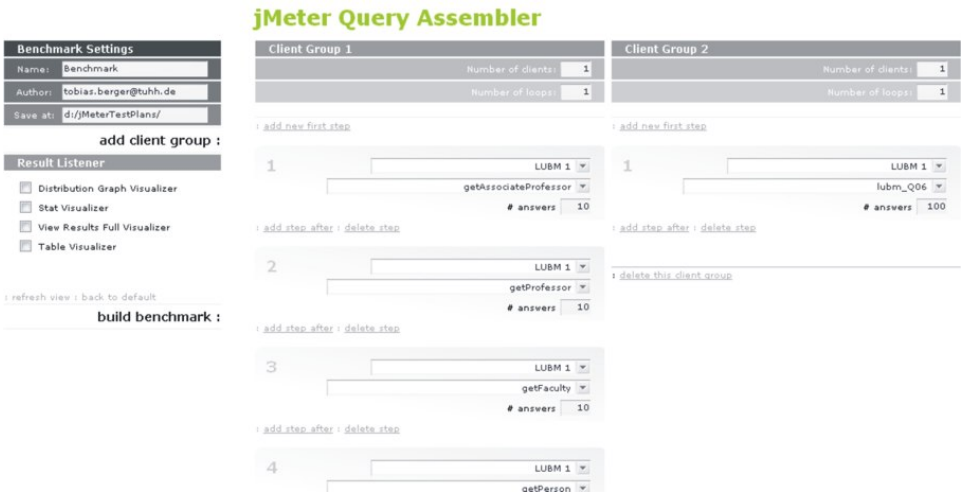


Figure 6.2: jMeter Query Assembler - Example GUI view with 2 clients and the control panel. The first client has a list of 4 queries to execute, the second client a list of just 1 query.

about listener modules that can be attached to each test in order to record and analyze the measured data.

In order to be able to model all of the components they were converted into classes, which provided the basis for the assembly of the test plans.

**The OWL QL Query Module** Due to the implemented OWL-QL server offering a Web service interface the test framework had to adapt to that. The SOAP module that came with jMeter offered the basic functionality needed to establish a connection with RacerManager. But in order to send an OWL-QL query to the server, the module had to be further adapted.

**Query Pool** Part of the idea to create a tailored assembly application was to increase the convenience for this task. Thus the application implements a query pool from where particular, predefined queries can be chosen. The application sorts those queries by their referenced knowledge bases, allowing the user to first select a KB and then the according query to the KB.

**Continuation Query** Due to the fact that jMeter is able to process the answers it receives, it can be instructed to filter out the process handle that is contained in the answer of one of the queries and then also automatically place this piece of information in the next query.

jMeterQA uses this functionality to support continuation queries. The GUI identifies on the fly when one of the queries in the list of a particular

client has the same combination of KB and query as one of the previous queries. It application offers than the user the possibility to declare this query to be a continuation query.

**Assembly** Once the user is finished with the composition of the test plan, the representation of it that exists within the system will be translated into XML and stored at a specified location. From there one it will be handled by jMeter itself.

In the same way as the rest of the system, the user interface (or the "view") was especially tailored toward this particular application. Figure 6.2 shows an example of the building of a test plan.

The user can add new client groups and queries as well as delete those with just one press of a button. Clients and queries are displayed in a way that reflects their relationship to each other as well as their order of execution. That provides the user with a very good overview over the whole test plan.

# Chapter 7

# Evaluation

After the theoretical approach to the topic in chapter 4 and 5 the system was implemented as described in chapter 6 This chapter is concerned with the evaluation of the various theories and assumptions that were proposed during the previous chapters based on empirical testing.

After describing the environment in which the tests were carried out, the different steps, different modules and features of the system will be looked at and evaluated based on the results of the test.

## 7.1 Environment for Empirical Evaluation

The choice of the right test environment is significantly important to assure genuine and reproducible test results. Thus components were chosen that are able to support this.

### 7.1.1 Test Plans

Each test follows a previously compiled *test plan* which describes

- the number of clients that concurrently send queries to the system
- each particular query that is used in the plan: the query pattern, the knowledge base and the number of answers
- the order in which each client sends the queries to the service.

When a test plan instructs a client to send more than one query, these queries are always executed sequentially, i.e. when the response for query 1 is received, query 2 will instantly be sent.

#### Knowledge Base Initialization

In some benchmarks the test plans will claim that previous to the test run the used knowledge bases were loaded and the index structures have been

built. This was done by a initialization query to each involved knowledge base causing a reasoner to load the KB and built the index structures in order to answer the query. It was assured that the used queries did not interfere with any of the queries in the benchmark, e.g. by making it possible for a query to profit from the cache or subsumption relationship.

### 7.1.2 Knowledge Bases

The knowledge base that was used for the testing was based on the *Lehigh University Benchmark Ontology (LUBM)*[18] and created with the *Data Generator(UBA)* that is provided by the *SWAT* team of the Lehigh University. The original knowledge base that was used in the test can be downloaded at
`http://tobiasberger.eu/racermanager/ont/univ-bench.zip`.

In order to create meaningful benchmarks more than one knowledge base was needed. To still assure that the results were not biased by queries referencing a different knowledge base than others, the knowledge bases were simply copied and differently named. Thus an equal basis for all queries were given, while still providing - from a system and test perspective - different knowledge bases

#### Queries

The queries are translations into OWL-QL of the 14 LUBM queries, which are provided by along with the benchmark ontology. All translations of those of the 14 LUBM queries that were used in the testing can be found in appendix D and can also be found at
`http://tobiasberger.eu/racermanager/ont/lubm-owlql.zip`.

The use of these widely used queries make the results reproducible and comparable to later versions of the system that may have a different interface as well as to other systems that already exist.

### 7.1.3 Framework

The test plans that were previously described were compiled by the *jMeter Query Assembler*[1] *(jMeterQA)*. The output of the assembler application is an xML document which is structured in a way that it can be read and executed by the load and performance measurement tool *Apache jMeter[9] (jMeter)*.

[1]For more information about the application please see the chapter about the implementation, paragraph 6.4.1. The application implementation and operation are described there in detail.

#### Apache jMeter

Originally designed for load and performance tests on standard Web applications and servers jMeter became a widely used tool for performance and load testing *both on static and dynamic resources (files, Servlets, Perl scripts, Java Objects, Data Bases and Queries, FTP Servers and more)*[9]. It is a 100% Java application which makes it suitable for more many environments.

jMeter comes equipped with a module for SOAP over HTTP. This allows it to send messages to the SOAP service that is implemented by the Semantic Web middleware that is to evaluate.

Although jMeter provides a GUI, the application was executed in a non-GUI mode due to the system and in order to minimize the impact of the application on the measured results.

**Concurrency** jMeter is full multi-threading application which allows to simulate the action of many concurrently acting clients, each which can perform different functions - i.e. in this case that each client can query the system in a different way, with different queries.

**Response Time Measurement** The time that jMeter measures is the difference of the time when the query is sent to the system and the time when the response arrives at the system. The data that is send with the response is neglected in order not to slow down further processing by making jMeter busy with handling the response.

All the times and describing data, like the query name, are cached while a test is executed. After the test each data tuple - consisting of the measured time and describing data - is stored in a text file. This way the measurements are as less as possible influenced.

**Data Handling and Analysis** jMeter also provides means for the display and analysis of the data that was measured. These functions are only available in the GUI mode of jMeter and fairly limited. Thus only the measurement and storage of the data was done by jMeter.

### 7.1.4 System

#### Architecture

The test system that was used to run the semantic middleware, the reasoners and the test framework was a multi-processor Sun Solaris Server with 8 processors and 32GB memory. The used architecture allowed to reduce the impact of the network latency on the measured times while keeping system performance untouched.

The significance of the problem of network latency can be seen in the following, especially in the evaluation of the cache usage. Here were times measured less than $30ms$ which easily double with a certain degree of latency.

On the other hand placing all applications and reasoners on a single server - to minimize the latency - with too few processors would reduce the performance of each of the components. Comparing (i) 1 reasoner that fully uses 1 processor to answer 2 queries sequentially with (ii) 2 reasoners that share 1 processor and answer the queries in parallel wont show huge, if at all differences in their behavior and response times.

#### Reasoners

For the tests the semantic middleware application managed different numbers of *RacerPro* inference servers in the version 1.9.0.

### 7.1.5 Data

Due to time limitations each evaluation is based on a the measurements of a particular test run and not on the average values of multiple runs. However, for each benchmark several test runs were made, all of which showed very similar values with slight to no variation.

## 7.2 Evaluation of Load Balancing

The whole system and its implementation is based on the assumption that load balancing is able to provide a means for scalability, higher availability, and faster responses than a single reasoner is able to.

Consequently the system will be evaluated

- with regard to load balancing as such, if it can provide a benefit or not
- with regard to the used load balancing strategy, if as proposed an intelligent strategy is more efficient than a ignorant one

The evaluation is based on two benchmarks that will compare the query response times (response time) of systems with different number of reasoners and different load balancing strategies.

### 7.2.1 Benchmark I

Benchmark I measures and compares the response times for a system that implements an *ACO-LB* as it was previously described and a pure Round Robin strategy.

### Test Plan

3 clients concurrently send queries to the system. Each client sequentially sends 3 pairwise different queries, each of which references the same KB. Each client references a different KB, thus in total 3 KBs are referenced. Each client sends the same set of queries in the same order.

At the time when each client's first query arrives at the system, the referenced knowledge base is not yet loaded and the index structures have not been built.

The test plan intentionally uses different knowledge bases to create a more realistic example of a Semantic Web scenario. In order to make the individual results of the clients comparable, each of the clients sends the same queries to the system.

### Setting: ACO-LB

The system is a basic implementation, using a *Ant Colony Optimization Load Balancing Algorithm (ACO-LB)* without further optimization modules. The system implements the paradigm that once a knowledge base is loaded, all queries to that KB are send to it without further load balancing considerations.

The response times for the queries are measured for settings where 1, 2, 3 or 4 reasoners are managed by the semantic middleware. The 1 reasoner system presents an equivalent to a system without any load balancing capabilities, respectively a single reasoner without a middleware. The single reasoner still has to be managed by the middleware to create equal conditions for single and multiple reasoner systems. Besides that it acts as an interface for the SOAP messages, which is needed for the test framework.

### Setting: Round Robin

In the second setting for this benchmark the system bases its load balancing decisions on a pure Round Robin strategy, pure in the way that it bases its load distribution decision only on its server list just like it is done in DNS Load Balancing or Server Load Balancing. The strategy does not make any further considerations or assumptions for the load balancing.

Here tests will be made on systems with 1, 2, and 3 managed reasoners. As argued before, the single reasoner will be as well managed by the middleware to create equal conditions.

This setting is intended to pick up the basic load balancing ideas that are well known and implemented, e.g. in server systems, and evaluate how such a strategy will perform in the Semantic Web scenario. The results will be especially evaluated in comparison to the previous setting ACO. The Round Robin strategy presents one of the best known example for such strategies.

### Results for ACO-LB

Basis for the evaluation of the system and its various variation in which it is tested are the response times as they are measured at the client. Due to the order in which the queries arrived at the system and were processed different waiting and processing times let each measure different developments of the response times. However, it is not the aim of the semantic middleware to improve the processing for a particular client but in average for all clients. Consequently the average response times have to provide the basis for the evaluation[2].

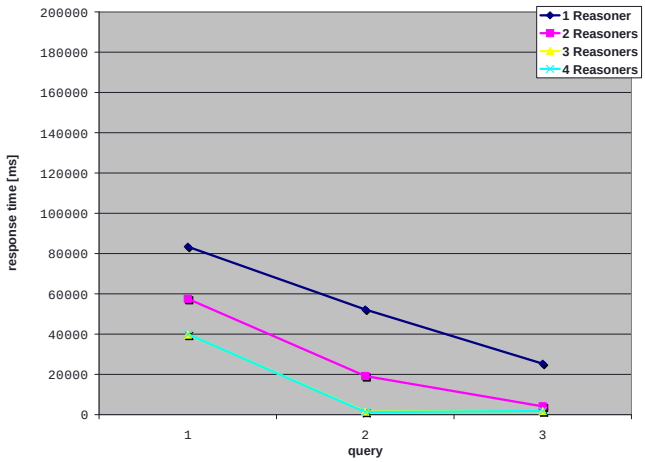


Figure 7.1: Benchmark I. The chart compares the development of the **average ACO-LB** query response times for each of the settings. A single graph for each setting comparing the response time developments for each client can be found in figures C.1, C.2, C.3 and C.4. *Notice that the graph for "3 reasoners" is covered by the one for "4 reasoners".*

Figure 7.1 shows the development of the average response times for the ACO-LB balanced systems with 1, 2, 3 and 4 reasoners. Each of the developments show a clear downwards trend. As the pure processing times for the different queries are more or less equal, and definitely not in a range of about $40000ms$ (which corresponds to the reduction from query 1 to 2) the trend can be explained with the stepwise reduction of waiting time in the reasoner queues. As previously mentioned the first query to a particular knowledge base will cause the reasoner to load the KB and create the index structures. Thus the response times for the first queries consist of the 3 components *waiting time*, *KB loading*, and *processing time*. The response times of the second and third queries, which reach the system when the corresponding previous query was answered, are composed of waiting time and processing time. As there is no further need for KB loading due to the

[2]For further information on the response time developments for every single client see the figures C.1, C.2, C.3 and C.4 in the appendix.

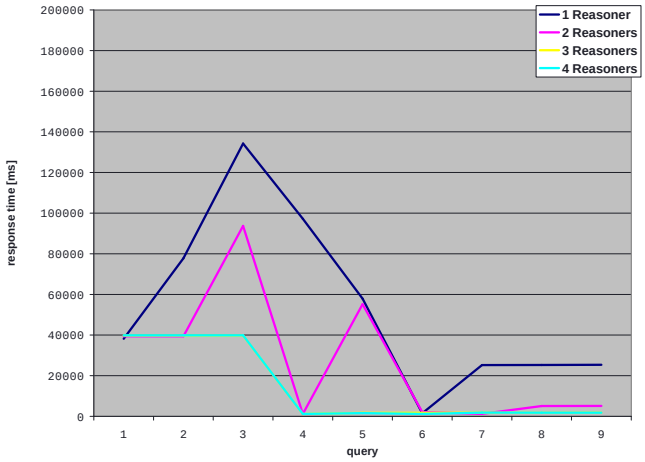


Figure 7.2: Benchmark I. The chart compares the development of the query response times for **ACO-LB** from the system's perspective. It shows the development in the order in which the queries where answered by the reasoners.

algorithm.

The differences in the response times of the three systems for query 1 result from the different waiting times. While the systems with 3 and 4 reasoners are able to answer the 3 concurrent queries in parallel without waiting times, the others can't. In the system with 2 reasoners only 2 queries can be answered in parallel, while the third query has to wait in one of the queues for one of the other queries to finish. This waiting time equals the processing time of the query (which can be assumed to be equal for all three first queries) thus doubling the response time for this particular query.

The same is true for a system with just one reasoner. Here the query 1 that arrives second has to wait for the first to be answered - doubling its response time -, and the query 1 that arrives third has to wait for the first and the second query to be answered - tripling its response time.

The second graph, Figure 7.2 shows this development very clearly. While for systems with 3 and 4 reasoners the response time stays constant for the first 3 answered queries (these are the three query 1) the third in the 2 reasoner system and the second and the third in the 1 reasoner system have response times, with increases in the size of the response time of the previous query(es).

As mentioned earlier query 2 and query 3 only consist of processing time and waiting time. As query 2 is send when query 1 was answered, this means for the system with 3 and 4 reasoners that query 2 reaches the reasoner when it is idle, i.e. when there are no waiting times. Thus the average response time of query 2 for these systems is the pure processing time that is needed to answer the query. This does not change for query 3 as the conditions are equal when they arrive at the system.

This is especially different for the system with 1 reasoner. Here the first query 2 reaches the system when the second query 1 just started to be processed by the reasoner and the third one is still waiting. Thus its response time contains approximately 2 times the response time of query 1 in addition to its own processing time. This causes a significant difference to the 3 and 4 reasoner system, as can be seen in Figure 7.2. The development continues in the same manner for the following queries which - still true for queries 3 - contain a significant amount of waiting time in their response time.

The system with 2 reasoners is a special case. It is kind of a hybrid of the 1 and 3 reasoner system. Due to the 3 KBs that share 2 reasoners, the 3 queries to KB1 behave like they being processed by a 3 reasoner system, while the other 2 share a reasoner and behave like being processed by a 1 reasoner system. These different behaviors are reflected by the zick-zack pattern that the graph describes - on the one side touching the 3/4 reasoners graph on the other side the 1 reasoner graph. The peaks in the graph are caused again by the waiting times of queries to KB2 and KB3.

Both graphs show clearly that a multi-reasoner system with an ACO-LB is able to reduce the response times for concurrent clients by minimizing the waiting times.

A further important result that the benchmark showed, is that - given the fairly few concurrent clients - the maximum number needed to provide an optimal system is equal to the amount of knowledge bases. Due to the paradigm that queries will be send to those reasoners that already have the corresponding KB loaded, no query in this benchmark has reached reasoner 4 at all. Thus the system behaves like a 3 reasoner system and the fourth is just an unused, in a way redundant resource.

### Results for Round Robin

Equal to the previous paragraph here the average response time provides the basis for the analysis and comparison of the different systems. Figure 7.3 shows the development of the average query response times for a Round Robin balanced system with 1, 2 and 3 reasoners [3].

This time there is no clear downwards trend but each system shows its own particular behavior. The system with 1 reasoner shows the same behavior as the 1 reasoner ACO-LB system does. The reasons for this is that they are basically the same. Thus the development of the curve is as well based on the significant accumulations of waiting times that are stepwise decreased over time. The 1 reasoner graph in Figure 7.4 basically matches the corresponding graph of the ACO-LB system.

[3]For further information on the response time developments for every single client see the figures C.7, C.8 and C.9 in the appendix.

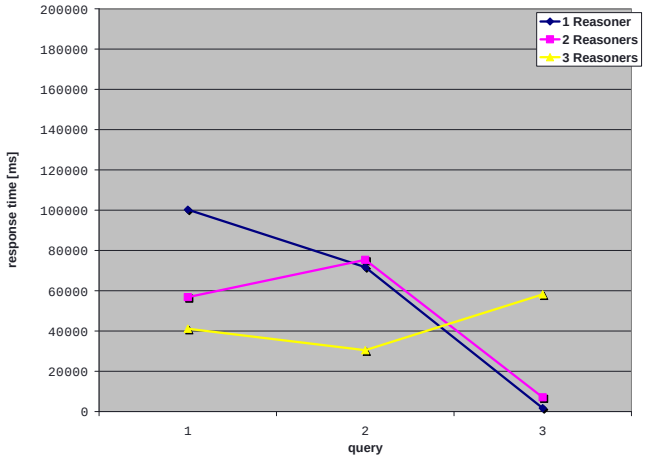


Figure 7.3: Benchmark I. The chart compares the development of the **average Round Robin** query response times for each of the settings. A single graph for each setting comparing the response time developments for each client can be found in figures C.7, C.8 and C.9.

The reason for the other graphs of the Round Robin system not matching the corresponding one in the ACO-LB system are the additional KB loading times that occur in the systems with 2 and 3 reasoners. Due to the blind distribution by the Round Robin strategy also the queries 2 and 3 can cause a reasoner to load and index a knowledge base. As it is not ensured that queries are preferably send to a reasoner that already has the referenced knowledge base loaded, it might be that the KB of query 2 or 3 is still unknown to the reasoner.

The effect for the 2 reasoner system is that - given the fact that queries are alternately send to the reasoners and it is a uneven number of KBs that are concurrently referenced - for query 2 a switch happens. All queries that were first answered by the first reasoner are now answered by the second, and the other way around. This causes a significant increase of the response times for query 2 compared to query 1. Queries have to wait for other queries to be processed and for further KBs to be loaded.

The same effect can be seen for the system with 3 reasoners, but only partly shifted to query 3, due to the different constellation of 3 KBs and 3 reasoners, which makes it more likely for the queries being always answered on the same reasoner - as long as they don't change their order. The change of order definitely happened for query 7, as can be seen in Figure 7.4. It causes a very significant increase in the response time due to existing waiting times and additional KB loading time. Another loading seems to have happened for query 9. All in all this behavior causes an increase of the average response times for query 3.

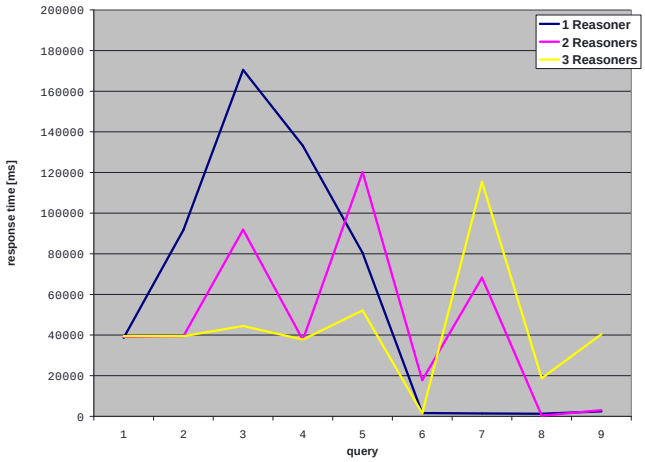


Figure 7.4: Benchmark I. The chart compares the development of the query response times for **Round Robin** from the system's perspective. It shows the development in the order in which the queries where answered by the reasoners.

### Comparing ACO-LB and Round Robin

A very basic difference of both systems in terms of their response times is that the ACO-LB offers a much more predictable behavior than the Round Robin system. Due to the blindness of Round Robin additional KB loading times occur and it is hard to say when a query, or a certain system setting will cause a significant load increase or not.

As Figure 7.5 shows in both cases leads the increase in the number of reasoners to a decrease of the average response times. Due to the additional loading times the graph shows as well that this decrease is much more significant for the ACO-LB balanced system than for one implementing a pure Round Robin strategy. It shows that in the case of the ACO-LB one additional reasoner causes the average response time to drop by half.

## 7.2.2 Benchmark II

Benchmark II measures and compares as well the response times for a system that implements an *ACO-LB* and a pure Round Robin strategy, just like benchmark I. This time the number of clients is increased to 10 to evaluate the impact of a bigger number of clients on the response times.

Benchmark I previously showed that the optimal number of reasoners depends on the number of different knowledge bases that are queried by the clients. Therefore it is no difference in the behavior of system with 2 KBs and 2 reasoners and a system with 20 KBs and 20 reasoners. This benchmark was designed to analyze the systems' behavior for an increased number of clients.

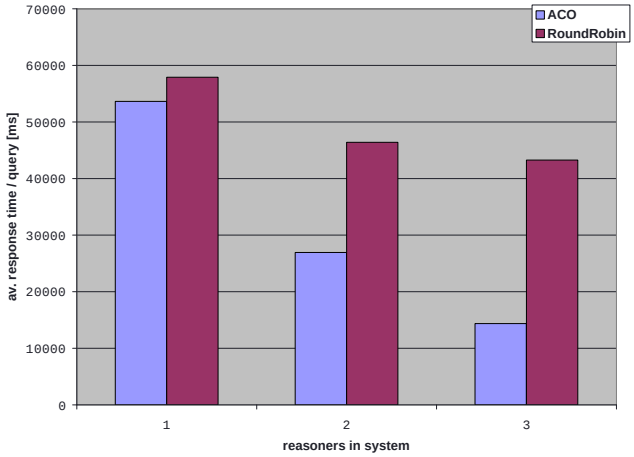


Figure 7.5: Benchmark I. The column chart compares the **average** response times for queries to the **ACO-LB** balanced and the **Round Robin** balanced systems with 1, 2 and 3 reasoners settings.

### Test Plan

10 clients concurrently send queries to the system. Each client sequentially sends 2 pairwise different queries, each of which references the same KB. The 10 clients reference 2 different KBs, 5 clients per KB. Further the queries of the 5 clients that query the same knowledge base are pairwise different, but both KBs will be queried by the same set queries.

At the time when each client's first query arrives at the system, the referenced knowledge base is not yet loaded and the index structures have not been built.

### Setting: ACO-LB

As in benchmark I the system is a basic implementation, using ACO-LB without further optimization modules. As well, the system implements the paradigm that once a knowledge base is loaded, all queries to that KB are send to it without further load balancing considerations.

The response times for the queries are measured for settings where 1, 2, or 3 reasoners are managed by the semantic middleware. Again the 1 reasoner system presents an equivalent to a system without any load balancing capabilities, respectively a single reasoner without a middleware. Still the 1 reasoner will be managed by the middleware, too.

### Setting: Round Robin

The second setting for this benchmark is the analogous setting of the previous benchmark. The system bases its load balancing decisions on a pure

Round Robin strategy. For further information see *benchmark I*.

Again tests will be made on systems with 1, 2, and 3 managed reasoners. As argued before, the single reasoner will be as well managed by the middleware to create equal conditions.

### Results for ACO-LB

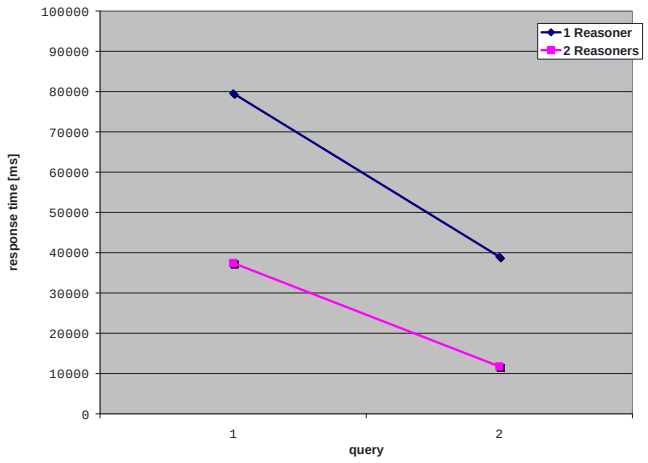


Figure 7.6: Benchmark II[5]. The chart compares the development of the **average ACO-LB** query response times for each of the settings. A single graph for each setting comparing the response time developments for each client can be found in figures C.14 and C.15.

Just like the previous benchmark, Figure 7.6 shows the development of the average response times for the whole system. The resulting graph mirrors the behavior that was already discovered for the ACO-LB in benchmark I. Here again a clear downwards trend can be discovered, due to the same reasons as in benchmark I.

For query 1 the average response time for a single reasoner system is approximately double the time of the 2 reasoner system. This is again due to the additional waiting time for the 8 queries that come after the first two. While the first query has no waiting time, the second query has to wait for the first to be answered, but will as well cause the reasoner to load its KB as it references very probably the other KB. All following have to wait for both to load the KBs and to be answered.

This is different for the system with 2 reasoners. Here the queries are equally distributed by their KB. This will cause the both reasoners to only load one KB, which is done in parallel after the first arrived. Thus all following queries - in both reasoner queues - will only have to wait for 1 KB to be loaded and the previous queries to be answered.

This behavior is as well reflected in Figure 7.7 which shows the development of the response times from the system's perspective, ordering the

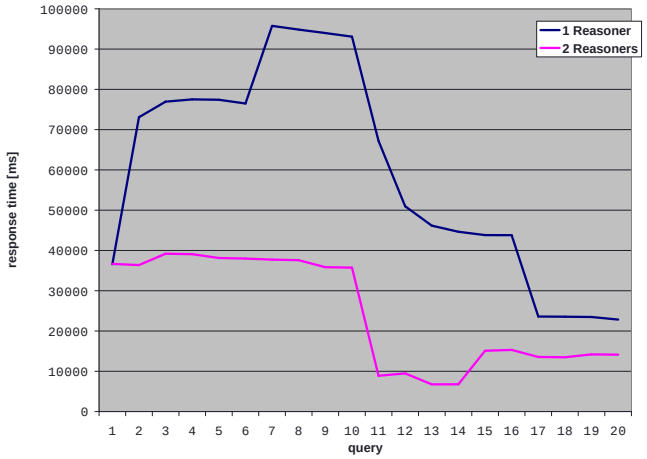


Figure 7.7: Benchmark II. The chart compares the development of the query response times for **ACO-LB** from the system's perspective. It shows the development in the order in which the queries where answered by the reasoners.

queries by the order they were answered. The graph shows the increasing response times for the single reasoner system for the queries 1-10 and shows the effect of decreasing waiting time for the next 10. This is as well true for the 2 reasoner system.

Both graphs in the figure show a change of the slope around the queries 12 to 14, which can be explained with the answering of queries with high processing times.

### Results for Round Robin

Figure 7.8 shows that contrary to the ACO-LB strategy the Round Robin strategy does not show an overall response time decrease with an increasing number of reasoners. While the single reasoner implementation shows as well the the same behavior as for the ACO-LB, the behavior for a 2 reasoner system is a lot less predictable.

Figure 7.9 shows two developments into opposite directions for queries 11 to 17. As queries are alternately answered by reasoner 1 and reasoner 2, the graph shows the differences for the response times of queries that are answered by one or the other reasoner. As the Round Robin strategy distributes the queries blindly, it cause a reasoner here and there to load a further KB - which is the reason for peaks in the response times.

This behavior makes it random and unpredictable how the response time for a particular client will look like, so that the very different response time developments can be noticed regarding each client individually[6].

[6]For further information on the response time developments for every single client see the figures C.19 and C.20 in the appendix.

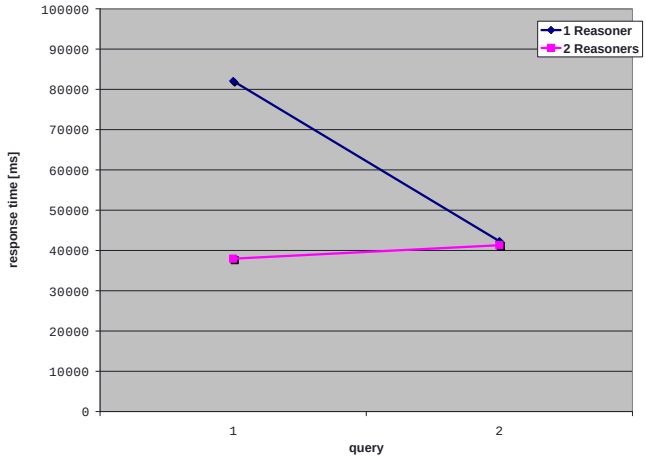


Figure 7.8: Benchmark II. The chart compares the development of the **average Round Robin** query response times for each of the settings. A single graph for each setting comparing the response time developments for each client can be found in Figures C.19 and C.20.

### Comparing ACO-LB and Round Robin

Analogous to benchmark I, benchmark II shows that the behavior of a single reasoner system reasoner is equal for both load balancing strategies. Both strategies behave very similar as there is obviously nothing to balance, if there is only one reasoner.

The measurements clearly show that the ACO-LB system is superior when it comes to the reduction of response times. A comparison of both strategies is illustrated shown in Figure 7.10.

Compared to the Round Robin strategy, the ACO-LB system delivers very predictable results for each client, i.e. the scaling from 1 to 2 reasoners has a positive effect for all queries and all clients in the same way.

## 7.2.3 Evaluation Results

The evaluation of benchmark I and II based on empirical data delivers the following results.

**Scalability** It was proved that a Semantic Web inference system is able to scale by increasing the number of reasoners and distributing the load by means of a load balancing algorithm.

**Availability** The queuing mechanism led to an increased availability by prohibiting the rejection of a query. Instead the query was queued and the service was always available.

**Quality of Service: Faster Responses** The system as it was described was not able to increase the response time of a single particular query

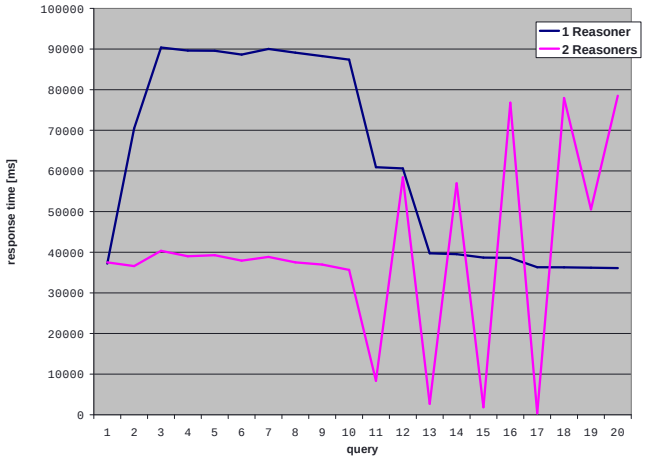


Figure 7.9: Benchmark II. The chart compares the development of the query response times for **Round Robin** from the system's perspective. It shows the development in the order in which the queries where answered by the reasoners.

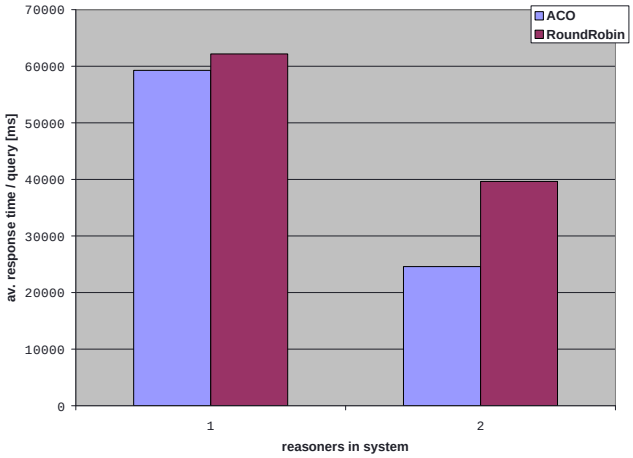


Figure 7.10: Benchmark II. The column chart compares the **average** response times for queries to the **ACO-LB** balanced and the **Round Robin** balanced systems with 1 and 2 reasoners settings.

by decreasing its processing time. A faster response could be achieved by distributing different queries over different reasoners and thus reducing waiting times which otherwise would have occurred for the query.

**Intelligent Query Distribution** The ACO-LB assumption that a query should preferably be answered by the reasoner that has already loaded the referenced knowledge base was proved to be valuable. The performance increase that could be achieved compared to a ignorantly acting Round Robin strategy was significant.

**Waiting Times** Decreasing waiting times showed to be the key element for the decreasing of response times. By adding a further reasoner to the system the average response times could be decreased by more than 50% (Figures 7.10). The impact of the accumulating waiting times is very significant for the performance of a system.

**Optimal Number of needed Reasoners** benchmark I showed that the maximum number of needed reasoners is equal to the number of different KBs that queries reference. This result is only true in so far as the number of clients that concurrently send their queries to a particular knowledge base are fairly limited. In case the load for a single knowledge base becomes too big for a single reasoner it is preferable to have a second reasoner supporting the first.

As a result the system should be able to dynamically add new reasoners in order to handle load increases.

## 7.3 Evaluation of Optimization by Cache Usage

The second principal assumption is that the implementation of a cache in the semantic middleware will help to improve the systems performance by reducing the need for reasoning.

By the means of benchmarks III and IV the cache usage will be evaluated by comparing the response times of a system without cache with a system with cache.

For the evaluation Both benchmarks will instruct different numbers of clients to repeatedly send the same query. This, one might argue, is not an example that is very near to the real world and thus night not be suitable for the evaluation of the test. But it has to be noticed that having one client repeatedly send the same query and this is done in sequence - which is the case as explained previously - is equal to having a set of clients sequentially sending the query. Therefore the used test plans match real world scenarios.

### 7.3.1 Benchmark III

#### Test Plan

10 clients concurrently send queries to the system. Each client sequentially sends 2 completely equal queries. The 10 clients reference 2 different KBs, 5 clients per KB. The queries of the 5 clients that query the same knowledge base are pairwise different. Both KBs will be queried by the same set of queries.

The system has been initialized so that the 2 KBs have been loaded and indexed before the first query was received, each KB by one reasoner.

The queries were chosen so that they offer a chance for the system to profit from the cache.

#### Setting: System without Cache

The system is a basic implementation, using a ACO-LB strategy without further optimization modules, the same as it was used in the previous benchmarks.

Based on the previously found results, the middleware manages 2 reasoners, each of which has been initialized with one KB so that the load balancer will distribute the queries accordingly.

#### Setting: System with Cache

As the purpose of this benchmark is to evaluate the benefit of a cache implementation in the semantic middleware, the second system's settings are equal to the previously described ones, except the fact that a cache is added to the system.

#### Results

The similarities and differences in the behavior of a system with and without cache are best seen in Figure 7.11. It shows the development of the query response times from the system's perspective, i.e. in the order as they were answered by the system.

Both systems show the same behavior for the first 10 queries, as for both the received queries are new and unknown. The development here is the same that already occurred in benchmark I and II. Although the knowledge bases have been preloaded, later queries still have to wait for earlier queries to be processed. These waiting times lead to ever increasing response times for the queries. However, a clear difference can be seen from query 11 on.

The second time the queries arrive the system they are already known. For the system without a cache this has no immediate impact. The queries are will be handled in the same way as they have been, when they arrived for the first time. The existing waiting times lead to a further increase of

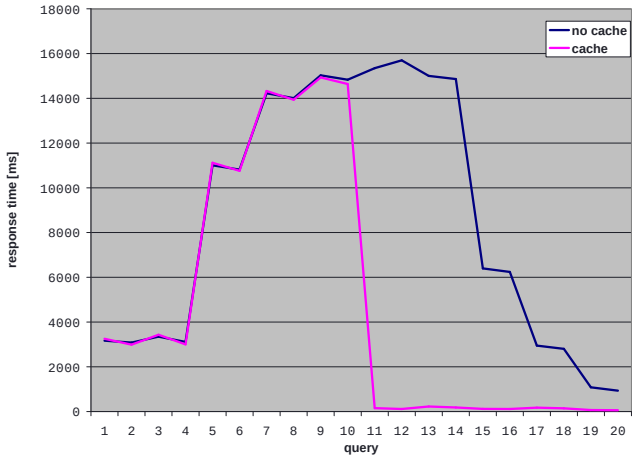


Figure 7.11: Benchmark III. The chart compares the development of the query response times for a system with and without a cache (from the system's perspective). It shows the development in the order in which the queries where answered by the reasoner as they are shown in Figure 7.11. *Note that queries 1-10 are the same as 11-20.*

the response times. Only later a pretty drastic decrease is observed. This leads to the assumption that also for the system without cache there is a difference between a query being answered the first or the second time.

The reasoners also contain a caching mechanism that allows them to answer previously answered queries faster. This explains the behavior of the not cached system (i.e. not cached refers to 'no cache in the middleware'). By speeding up the processing of the queries the reasoners are able to decrease the waiting times and thus the response times significantly.

The system with the cache in the middleware shows a different behavior. Here all queries are able to immediately profit from the cache. As the cache is implemented in the middleware it can be accessed, and the response can be created without consultation of a reasoner. Thus virtually no waiting times occur for the second set of queries. As there exists no waiting time for query 11 and the answer generation in the cache is very fast the further queries as well experience no or a very minimal waiting time.

The significant difference of a cached and a not cached system becomes even clearer when comparing the average response times for both. Figure 7.12 shows that the average response time from the cache was less than a $100ms$, which is significantly less than the average for a not-cached system.

The high impact the reasoner cache had on the reduction of the response times for the queries makes it interesting to compare as well the third step. No waiting times will then exist in either of the systems, which would lead to a direct comparison of the middleware and reasoner cache. The next benchmark will as well address this issue.

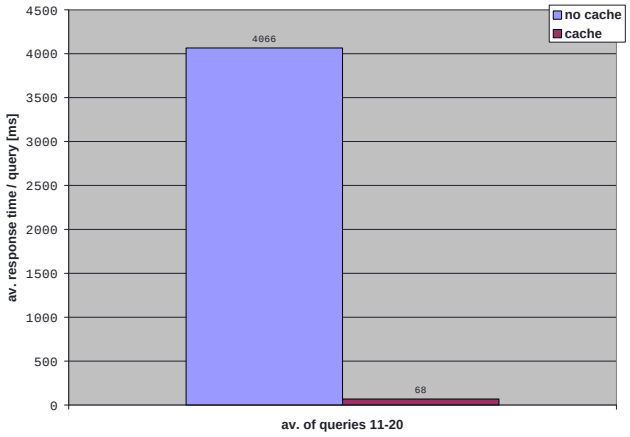


Figure 7.12: Benchmark III. The column chart compares the **average** response times of the queries 11-20.

### 7.3.2 Benchmark IV

#### Test Plan

6 clients concurrently send queries to the system. Each client sequentially sends 3 completely equal queries. The 3 clients reference 2 KB. 3 clients per KB. The queries of the 3 clients that query the same knowledge base are pairwise different. Both KBs will be queried by the same set of queries.

The system has been initialized so that the 2 KBs have been loaded and indexed before the first query was received, each KB by one reasoner.

#### Setting: System without Cache

As in the other cache benchmark the system is a basic implementation, using a ACO-LB strategy without further optimization modules.

Again, based on the previously found results, the middleware manages 2 reasoners - corresponding to the 2 referenced KBs. As well, each of the reasoners has been initialized with one KB so that the load balancer will distribute the queries accordingly.

#### Setting: System with Cache

The settings of the system with cache are equal to those cached system's settings of benchmark III.

#### Results

As in the previous benchmark figure 7.13 shows the development of the response times from the system's perspective, in the order that the queries

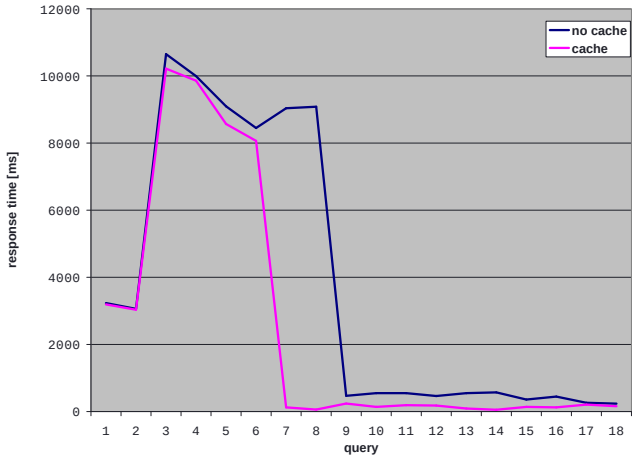


Figure 7.13: Benchmark IV. The chart compares the development of the query response times for a system with and without a cache (from the system's perspective). It shows the development in the order in which the queries where answered by the reasoner. *Note that queries 1-6 are equal to queries 7-12 and 13-18.*

arrived at the system.

What one can see is that the behavior of both systems just mirror the behavior they showed with benchmark III. For the the first time the set of 6 queries are received, both systems show an equal behavior. Due to longer processing times of the particular queries the waiting time for the following queries adds up and thus increases those response times.

For the second time the set of six answers are received the system here shows as well the same behavior as in the previous benchmark. On the one hand the response times of the not-cached system for the queries 7 and 8 first rise due to the existing waiting times, but then drop as well due to the effect of the reasoners' caching. On the other hand the response times for the system with the middleware cache immediately drop, once a known query is received. The system does not have to care for a reasoner to answer it, but creates the response directly from the cache.

While the first two parts of the graph only support the arguments and results that have been found in benchmark III, the third part contains valuable new information.

The third time the set of the six queries reach the system, there exist virtually no to very minimal waiting times for the queries in both systems. Both systems use their caching mechanism to answer the queries, either at the middle-layer or at the reasoner layer.

The trends of both systems are very equal and stable, showing both no bigger variations up or down. However the response times in this section are still higher for the system without a middleware cache. This result is even clearer reflected in Figure 7.14 where the average response times for the last

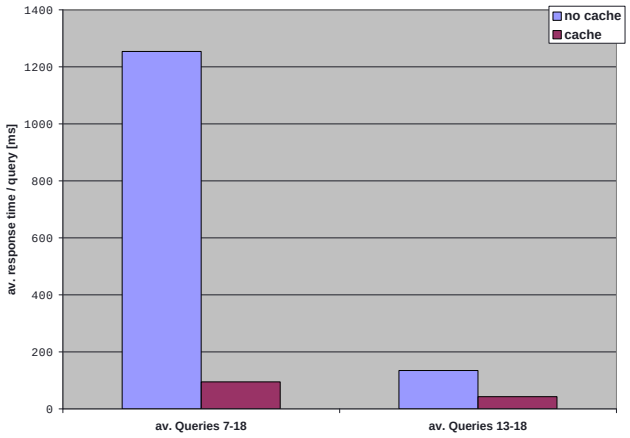


Figure 7.14: Benchmark IV. The column chart compares the **average** response times of the queries 7-18 and 13-18 (which are the intervals where differences occured due to the cache/no cache implementation) as they are shown in Figure 7.13. *Note that the difference between the both is that in the comparison 7-18 the accumulated waiting times of queries 1-6 affect the response times for the queries 7-12, while 13-18 shows a cache/no cache comparison free of waiting times.*

to sections are compared. It can be seen that there still exists a significant difference in the response times for the third segment, but the real effect of a cache reveals itself in the second section, by helping the queries to avoid long waiting times.

### 7.3.3 Evaluation Results

The two benchmarks have both explicitly shown the benefits of a system which implements a cache in the middleware over a system that just relies on the reasoners caching mechanism. However there are three cases that have to be distinguished.

The

In the case of existing or emerging waiting times due to former queries the middleware-cached system has a clear advantage over the other one. The cache provides an alternative source for information that can act independently of the reasoners. Thus it is as well independent of the queues and the waiting times. As it was already shown in the *evaluation of load balancing* in section 7.2, the waiting times are an important factor with a big influence on the response times of a system. Here, due to the possibility of the system to answer the query from either one of the caches very fast, this factor gains even more importance. Consequently the impact of the cache in the middleware is very high, as it lets the query avoid these waiting times.

The second case is when there are no queries waiting in the queues for

both systems. Here the advantage of the middleware cache becomes smaller, as it cannot differentiate from a reasoner by creating responses without waiting times. However it shows that a response from the middleware cache is still faster than a cached response from the reasoner. Obviously this is the result of the communication and translation overhead in case reasoner caches are exploited. Depending on the network latency between middleware and reasoner this effect can significantly increase.

Considering both cases it can be stated that the use of a cache always can be considered as a benefit for creating faster responses.

## 7.4 Evaluation of Optimization by Subsumption

By proving to provide a mayor improvement in the quality of service regarding the decrease of the response times, the cache provides the needed basis for the evaluation of the optimization by subsumption. As a mayor part of this optimization involves the usage of the cache it was very important that the cache as such did not harm to the system by increasing response times or similar.

Thus the benchmarks V and VI will compare the response times for systems with and without a subsumption optimizer implemented. The first of the two benchmarks will compare only the results for one client with incrementally subsuming queries. The second benchmark tests the systems with the same incremental queries but interfered by queries without any subsumption relationship.

### 7.4.1 Benchmark V

#### Test Plan

In this test plan only 1 single client queries the system. The client sequentially sends 4 queries to the system, each to the same KB. The queries incrementally subsume each other and want to retrieve instances of: AssociateProfessor, Professor, Faculty, Person. Between those concepts the following relationships exist:

$$Person \subseteq Faculty \subseteq Professor \subseteq AssociateProfessor$$

The system has been initialized so that the KB has been loaded and indexed before the first query was received.

#### Setting: System Without Subsumption

The system is a basic implementation, using a ACO-LB strategy with an implemented cache. This cache wont have an effect on any of the queries, as no repetition of known queries occurs, but is part of the system to ensure

better comparability to the system with subsumption. No further features are implemented.

### Setting: System with Subsumption

The second system is equal to the first with the addition of a subsumption module. This subsumption module will test the query first for previously answered more specific queries (children) from which the query could profit in combination with the cache. Secondly the query will be tested for previously answered more general queries (parents) in order to profit from a smaller search space.

### Results

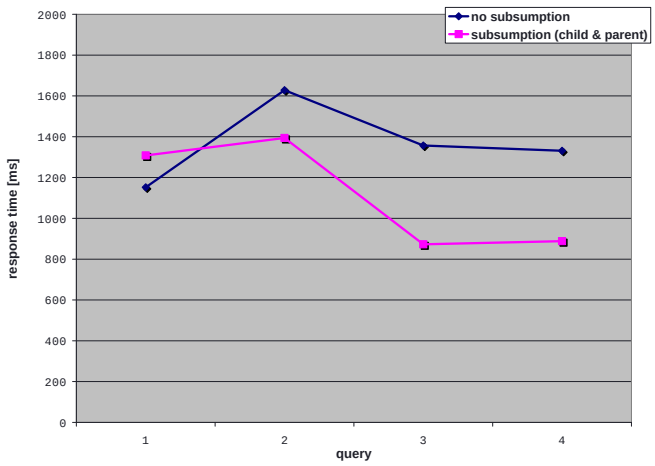


Figure 7.15: Benchmark V. The chart compares the development of the response times of the four queries for the system without and with subsumption optimization.

Figure 7.15 shows the development of the response times for the system with and without the subsumption module. In order to explain the behavior, it is useful to look at two different parts of the graph individually.

The first part is query 1. This query is not able to profit from any subsumption relationship to a previous query as it is the first. This makes it equal for both systems in the way that indipendent of how much subsumption checking will be made, there is no way to profit for the query. Thus it will be processed by a reasoner to provide the responses anyways. This is then the reason for the subsumption-system's higher response time of query 1. The module that checks the query for subsumption adds additional preprocessing time on the processing time of the query and thereby increases the response time.

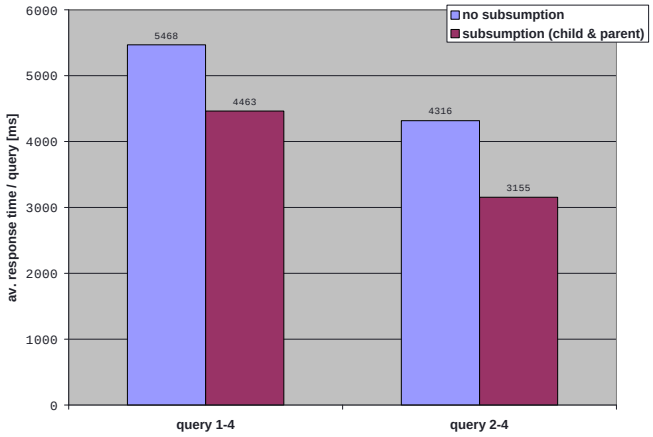


Figure 7.16: Benchmark V. The chart compare the average response times for all four queries of a system with and without subsumption optimization. The second pair of columns does the same comparison for the queries 2 to 4.

The second part are queries 2 to 4. These queries have a subsumption relationship with all of their predecessors, thus they have a chance to profit from the extra effort. This is as well reflected in 7.15. Both graphs show approximately the same developing, the subsumption-system's graph shifted down by a few hundred milliseconds.

All in all, as Figure 7.16 shows, the introduction of a module for subsumption checking payed off. The response times for queries that have a subsumption (2-4) relationship could be lowered by more than 25%.

### 7.4.2 Benchmark VI

#### Test Plan

In this test plan only 3 concurrent clients query the system. The first client sequentially sends - equal to the previous benchmark - the 4 incrementally subsuming queries to the system. The other two clients send queries without any subsumption relationship to any other query. All queries go to the same KB.

The system has been initialized so that the KB has been loaded and indexed before the first query was received.

#### Setting: System Without Subsumption

Equal to the previous benchmark the system is a basic implementation, using a ACO-LB strategy with an implemented cache.

### Setting: System with Subsumption - Parent & Child

The same is true for the second setting. The system is equal to the first setting with the addition of a subsumption module that checks queries for subsumption (child and parent).

### Setting: System with Subsumption - Child

As the previously described system, this system here implements as well a subsumption module. The difference however is that here only the child relation to a previous query will be checked.

### Results

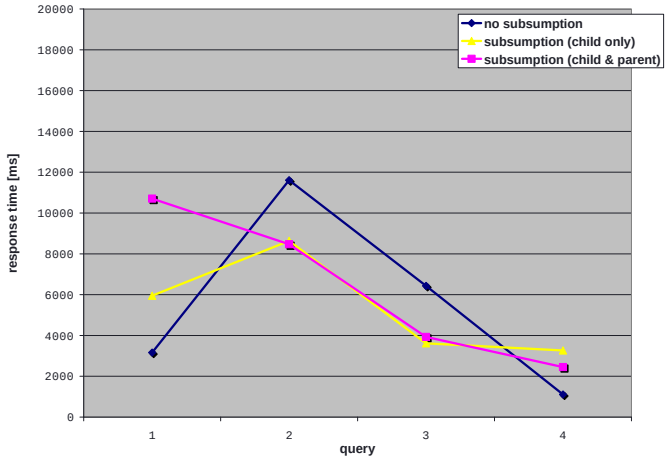


Figure 7.17: Benchmark VI. The chart compares the development of the response times of the four queries for the system without subsumption optimization, with child subsumption and with full child & parent subsumption optimization.

Similar to the previous benchmark Figure 7.17 shows the development of the response times for all three settings - this times the displayed response times are the averages of the three clients.

The graphs reflect great differences and similarities in the behavior of the systems. In order to find reasons for this behavior the graph will be analyzed in three steps.

The fist step is again query 1. Here the same behavior can be found, that was already reflected in the previous benchmark. Due to the additional preprocessing time that is invested into the first query, which shows - due to the not existing subsumption relationship - no benefit, the response times for query 1 increase for the systems with subsumption optimization. The

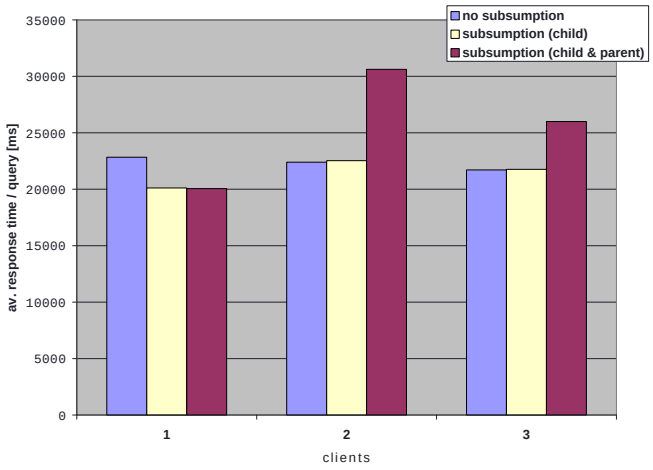


Figure 7.18: Benchmark VI. The chart compares the average response times for each client individually in regard to the different system settings.

graph shows as well that analysis of a query for child and parent relationship with a previous query causes higher response times than the optimization that only concentrates on the child relationship.

In addition to that the presence of further queries adds a waiting time component to the response times, as all queries will be answered by the same reasoner.

The second step includes queries 2 and 3. Here again the same behavior as in the previous benchmark is reflected. The subsumption relationship is detected in both subsumption-systems and lead to a reduction of the response times. When comparing the particular developments of each client it can be seen that this has as well a positive effect on the queries without a subsumption relationship that are answered concurrently. Those profit from the reduced waiting times in the system.

The fact that the graphs for queries 2-3 are approx. equal can be explained with the implementation of the modules. In the module that checks for both relationships, the child relationship is checked for first and dominant. Which means whenever a child relation is detected the rest of the checking is canceled. Thus both behave equally for these queries.

The third step includes only query 4. Here a new behavior is reflected that was not seen in the previous benchmark. Although a subsumption relationship exists to the predecessors of query 4, the system without subsumption checking responded faster to all 3 queries than the other systems. Given as well the fact that the three query 4's showed all in general very small response times let conclude that the extra effort for subsumption checking in combination with a cache lookup took longer than the simple and plain reasoning for these queries.

The overview on the average query response times in Figure 7.18 underlines the analysis that was made so far. The system that implements both child and parent subsumption checking causes higher response times for those clients that can not profit from such a relationship. It further shows that in the case of only checking for the child relationship the response times stay equal compared to the system without subsumption checking. The reason therefore must be seen in the waiting times. The avoidance of reasoning for the first client let to a reduction of waiting times for the second and third client, thus equaling out the increases that were made to their response times by the additional, yet useless subsumption checking.

As already seen in the previous benchmark, for the client 1 the subsumption checking led to a clear reduction of response times.

### 7.4.3 Evaluation Results

The analysis of benchmark V and VI does not paint a picture that is as clear as in the for the previous evaluation. On the one hand the exploitation of subsumption relationships pays off for those queries that actually have such a relationship to an earlier one. On the other hand it was shown that the system, implemented in the way it is, meant even a disadvantage for those queries that do not have such a relationship.

**Benefit for Subsumption-Queries** It was demonstrated in both benchmarks that the exploitation of an existing subsumption relationship proved to be a benefit for the system by decreasing the response times and decreasing waiting times for other queries.

**Disadvantage for Low-Load Queries** However, the previous point has to be limited to the extend that this benefit reveals itself only above a certain threshold value of the time needed for reasoning. Thus this benefit is not applicable for queries that mean a very small amount of load for the system.

**Unavoidable Extra Effort without Profit** The subsumption optimizer proved to be an immediate disadvantage for all those queries that can not profit from it. The development of the average response times from benchmark V to VI shows that with an increasing number of queries that can not profit from the optimizer the benefits for the whole system decrease and turn more and more into a reason for increased response times.

It must be concluded that subsumption optimization is not suitable for every system. The benefit is dependent on the ratio of queries with a subsumption relationship to a previous one and those which don't. The components that were used in the implementation do not allow for a more efficient

way of checking the relationships between the queries. Thus it has to be decided from application to application if a use of this optimization means a advantage or disadvantage.

A possible solution may be to introduce the subsumption optimization into the system, when it has reached a certain degree of maturity. This would mean that already many queries had been answered and thus the chances increase for a new query to have a subsumption relationship to one of the earlier queries.

The system may offer as well a higher probability for the subsumption checking to be advantageous, when the the reasoners are not local but remote. This could lead to significant increases of the response time due to network latency. Therefore the subsumption optimizer would not only help to avoid the processing time, but also the time for transportation.

A further scenario that shows a high probability for the beneficial use of subsumption optimization is the involvement of payments in the use of reasoning capabilities. If a service provider offers the use of its reasoner for the payment of a certain amount (may be even dependent on the number of requested answers) the middleware that connects to this server will try to avoid the reasoning as far as possible in order to reduce the costs of payments. This will shift the measurement for the evaluation by response times to the evaluation of made payments. The additional preprocessing time that the subsumption optimizer creates looses its importance as it is more important to answer queries as much as possible from the local cache.

# Chapter 8

# Conclusion and Future Work

## 8.1 Conclusion

Based on the requirements of different scenarios for Semantic Web applications the thesis proposed the implementation of a Semantic Web middleware system (Chapter 3). The middleware was intended to support scalability, availability and a reasonable quality of service for multi-user inference systems, while offering an open platform for the addition of further modules.

### 8.1.1 The Benefit of Load Balancing

The empirical *evaluation of the benefit of load balancing* in section 7.2 confirmed that a semantic middleware system is able provide the required scalability and availability of a multi-user system.

By implementing a load balancing strategy that distributes the load over a set of reasoners the empirical testing showed that the implemented middleware was able to

- concurrently handle multiple client requests, thus providing a much higher availability than a single reasoner
- scale well by increasing the number of reasoners to handle a higher load, as well for scenarios involving multiple knowledge bases
- provide significantly faster response times than a single reasoner system in multi-user scenarios

The empirical testing showed further the superiority of the proposed intelligent load balancing algorithm, which was based on Ant Colony Optimization (the ACO-LB). The blind and ignorant acting Round Robin strategy, which was used in some of the already well-known systems (see chapter 2, showed a very unpredictable behavior and far slower response times than the ACO-LB. These results confirmed the assumption that distributing the

queries preferably onto reasoners that already have the required knowledge base loaded is a benefit for the system.

### 8.1.2 Optimization

The characteristics of todays reasoners, as they were explained in chapter 3, made it very likely to improve an inference system's overall performance by investing into extensive preprocessing of the queries. The empirical results for the cache evaluation supported this assumption entirely. The testings showed that the implementation of a cache in the middleware provides an enormous decrease of the response times of recurring queries, which also has a beneficial effect on the whole system. The middleware-cached system not only showed to be superior in high load situations where a lot of waiting times occur, it showed also faster response times than an fairly unloaded system which profits from the caching mechanisms in the reasoners.

Different from the cache, the subsumption optimizer could not deliver the expectations that were formulated in section 4.3. While providing those queries with an advantage that had a subsuming relationship with one or more of the earlier queries, those queries that did not have it, experienced an increase of their response times. It was shown that the benefit that would be provided by the subsumption optimizer for an entire system does very much depend on the particular scenario and composition of the involved queries.

During the *evaluation of load balancing* the result was found that an the optimal number of reasoners heavily depend on the number of different referenced knowledge bases. This seems to make the implementation of a knowledge base distributor obsolete. However, it is important to consider that benchmarks always simulate a reasonable, yet artificial scenario. In various Semantic Web scenarios different knowledge bases will be queried with different intensities, thus making it still promising to further distribute heavily queried KBs. The effect here will be then similar to scaling the system from one to two or more reasoners, making the load distribute over all of the entities.

In consequence the conclusion for the knowledge base distributor is a similar as the one for the subsumption optimizer. The benefit that may come from using this component is very much dependent on the particular scenario and can neither be confirmed nor denied in general.

All-in-all the semantic middleware proved to fulfill all of the basic requirements, while being able to increase the quality of service. The theoretical considerations in combination with empirical results and the implemented system will provide a good foundation for further investigation and improvements in the field of multi-user inference systems.

## 8.2 Future Work

The thesis discussed the very basics of load balancing in multi-user inference systems. As this field promises a lot of potential there were many topics that could not be addressed in detail or at all in the limits of this thesis. Further various new ideas and topics were discovered during the analysis of the different aspects of load balancing.

In the following a list of the main topics and concerns for future work will be given and shortly explained.

### 8.2.1 Single vs. Multiple Semantic Middleware Systems

The evaluation of the implemented system showed that the semantic middleware is able to provide the system with a good degree of scalability and availability. However it has to be assumed that also this system can not scale infinitely and will may reach its limit at a certain point. Further there will exist scenarios where reasoners or reasoner clusters are locally distributed, e.g. being placed in the internal network of different company locations.

These considerations open up a whole area for further investigation:

- what are the limits of a single semantic middleware, i.e. in terms of the numbers of managed reasoners, queries (respectively throughput) per second, etc.
- when is it more efficient to replace a single semantic middleware application with multiple, parallel middleware applications
- what is an efficient way to balance the load on parallel semantic middleware systems
- how can the load be balanced efficiently between locally distributed reasoners or reasoning cluster

These investigations have to be considered as vital for the application of the semantic middleware in the business context.

### 8.2.2 Developing the Semantic Middleware

The proposal for the semantic middleware system in chapter 3 listed more requirements than just the basic ones: scalability, availability, quality of service. Various additional functions were suggested to be integrated into the single middlelayer application, including functions for security and accounting.

In order to develop the concept of a semantic middleware further, those components, their requirements and application areas have to be analyzed and evaluated. Such an evaluation also has to include the consideration if

such a single system is able to handle all these entirely by itself, or if it is more efficient to outsource parts of the functions to other existing or new components.

### 8.2.3 Publish and Subscribe Service

The *publish and subscribe* mechanism is a very common service that is offered in the area of information retrieval, content syndication and various other applications. The client instructs the system to send him further information, whenever something new was added to the field he subscribed for.

A publish and subscribe service should also be considered and evaluated for inference systems, especially as it is not as straight forwarded as for other systems. For instance, such a service is very unlikely to provide a benefit for users that use the incremental query answering. It may only then become of interest for the client, if he already received all answers from the system. It may then be applicable to offer the user a way to subscribe for further answers, when they occur.

All of this will only then makes sense, if the system, different from the current implementation, considers and allows knowledge bases to be altered - which is another interesting topic for further investigation.

### 8.2.4 Knowledge Base Alternation

As mentioned in the previous paragraph, the current system implementation considers the knowledge bases to remain unchanged throughout the system's complete lifecycle.

Thus, further effort has to be put into the issue of exchanging whole knowledge bases or altering their content to provide the semantic middleware with a wider area of application.

Part of this effort will be to reevaluate the cache and its implementation, considering its efficiency especially with regard to the expiration of data items.

### 8.2.5 Support for Server-Side Knowledge Base Selection

Chapter 3 proposed, based on the arguments that were given in the OWL-QL specification[13], the support for server-side knowledge base retrieval. In his query, the client would instruct the inference system to chose the best suitable knowledge base for the optimal answer retrieval. Such a function brings up various issues that have to be analyzed carefully in order to provide the basis for such an implementation, issues such like:

- where does the system find the knowledge bases, are they contained in a local repository, thus providing a finite set of KBs, or has the suiting

KB to selected from any resource on the net

- how is such a knowledge base handled after the query was answered - will be forgot or kept for further answering
- how does the system decide that one knowledge base provides better suiting information than another

### 8.2.6 High Availability Design

One reason to implement a load balancing system is to increase its availability. Due to health checking that will be conducted by the middleware the failure of reasoners can be detected and recovered reasoners can dynamically be reintegrated into the load distribution process.

An important issue that has to be addressed in this context is the failure of the middleware load balancing system. The middleware application so far represents a single point of failure, which when failing, will make the whole application fail.

Following the example of the in section 2.6 discussed active-standby and active-active systems such a scenario has to be analyzed as well for an load balanced inference system. Methods to provide efficient redundancy and recovery methods have to be found and implemented to address this very important issue.

### 8.2.7 Implementation

Besides the needed basic research in various topics, further effort has to be also put into the implemented system in order to provide the features and methods to work in a business environment.

#### Health Checking

In chapter 2Health checking was introduced as a mayor feature of load balancing systems in well-known environments. This issue could not be addressed at all in the scope of this thesis. However, it is a very important issue as well in Semantic Web reasoning systems.

In order to further improve the availability and flexibility of the system further effort has to be put into the investigation of efficient methods for

**Reasoner Failure** In order to provide a high degree of availability, the system has to implement a method that recognize the failure of a reasoner. In such a case this reasoner has to be automatically excluded from the load distribution process - which will involve further consequences for the system, such as cache expiration.

**Reasoner Recovery** In addition to the identification of a reasoner failure, the system should implement a mechanism that allows it to automatically trigger the recovery of a failed reasoner. Moreover the recovered reasoner has to be dynamically reintegrated into the system in order to reestablish the required scale and quality of service.

**Monitoring** The current implementation only provides a very basic way of monitoring. In order to let such a system be integrated in a business environment, a more extensive and sophisticated way of monitoring has to be implemented.

### Data Expiration

The implementation of a cache can lead to an enormous use of resources. Moreover, in the case the system will be extended to support alteration of loaded knowledge bases, the cache must adapt to this as well. It must provide functions that allow cache elements to expire when a KB was altered and prohibit the cache from growing unlimited.

Right now the cache is implemented to store the inferred knowledge volatile in memory. In order to prevent all cached information from vanishing in case the middleware crashes a solution could be to persistently store all inferred knowledge in a database.

Besides the cache, the subsumption optimizer has to be considered for data expiration as well. When a dialog expires in the cache it has to be ensured that the referenced query will be deleted from the subsumption reasoner. Otherwise inconsistencies will occur, if the subsumption reasoner finds relatives of the current query that do not exist in the cache anymore.

### Interfaces to Other Reasoner Products

By now the implemented OWL-QL reasoner RacerManager just provides an interface to instances of the reasoner product RacerPro. In the future additional interfaces should be developed to improve the possibilities to integrate the semantic middleware with existing reasoning infrastructure.

### DIG 2.0 Client Interface

Besides the implementation of various backend interfaces, the expansion of the system by implementing further ontology query language protocols.

With regard to the near future of modern DL systems, a promising candidate for such an extension is the existing standard interface for accessing DL reasoners (DIG). It is realistic to expect that the upcoming version, DIG 2.0[19], will replace OWL-QL as a standard query language. The upcoming version of DIG offers many features that have been found to be essential such as iterative query answering.

# Appendix A

# Additional Algorithms

**Algorithm 6** *continuationWorkflow.execute(query)*:

$session := getSessionId(query)$
$dialog := getDialog(session)$

$requestedAnswerSize := getAnswerSize(query)$
$deliveredAnswerSize := getAnswerSizeFromSession(session)$

$answers := getResponsesFromCache(dialog, requestedAnswerSize, deliveredAnswerSize)$
$query.setAnswers(answers)$

$responseQueue := query.getResponseQueue()$
$responseQueue.put(query)$

**Algorithm 7** *knownQueryWorkflow.execute(query)*:

$dialog := search\_dialog(query)$
$query.setSession(createNewSessionId(dialog))$
$requestedAnswerSize := getAnswerSize(query)$

$answers := getResponsesFromCache(dialog, requestedAnswerSize, 0$
$query.setAnswers(answers)$

$responseQueue := query.getResponseQueue()$
$responseQueue.put(query)$

**Algorithm 8** *newQueryToKnownKBWorkflow.execute(query)*:

```
subsumee := null
subsumer := null
subReasonerQueue := null
newKB := false

subsumee := getSubsumee(query)
if subsumee ≠ null then
  createSubsumptionDialog(query, subsumee)
  knownQueryWorkflow.execute(query)
else
  subsumer := getSubsumer(query)
  if subsumer ≠ null then
    subReasonerQueue := getReasonerQueue(subsumer)
    subReasonerQueue.put(query)
  else
    loadBalancer.balance(query, newKB)
  end if
end if
```

**Algorithm 9** *newQueryToNewKBWorkflow.execute(query)*:

```
subsumptionReasoner.loadKB(query)
query.setSession(createNewSessionId(createNewDialog(query)))
loadBalancer.balance(query, newKB)
```

# Appendix B

# Benchmarking Result Tables

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172029724416 | 472 | 1:lubm-Q01 | Client 1 |
| 2 | 1172029725178 | 3056 | 2:lubm-Q02 | Client 1 |
| 3 | 1172029728236 | 1131 | 3:lubm-Q05 | Client 1 |
| 4 | 1172029729369 | 1309 | 4:lubm-Q06 | Client 1 |
| 5 | 1172029730679 | 488 | 5:lubm-Q07 | Client 1 |
| 6 | 1172029731169 | 10732 | 6:lubm-Q09 | Client 1 |
| 7 | 1172029741902 | 1822 | 7:lubm-Q10 | Client 1 |
| 8 | 1172029743726 | 395 | 8:lubm-Q11 | Client 1 |
| 9 | 1172029744122 | 301 | 9:lubm-Q15 | Client 1 |
| 10 | 1172029744425 | 302 | 10:lubm-Q16 | Client 1 |

Table B.1: Results for: Response times on unloaded system.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172069557636 | 77800 | 1:lubm-Q02 | Client 1 |
| 2 | 1172069635438 | 57927 | 2:lubm-Q05 | Client 1 |
| 3 | 1172069693366 | 25291 | 3:lubm-Q06 | Client 1 |
| 4 | 1172069557636 | 134243 | 1:lubm-Q02 | Client 2 |
| 5 | 1172069691882 | 1483 | 2:lubm-Q05 | Client 2 |
| 6 | 1172069693368 | 25386 | 3:lubm-Q06 | Client 2 |
| 7 | 1172069557636 | 38210 | 1:lubm-Q02 | Client 3 |
| 8 | 1172069596133 | 97221 | 2:lubm-Q05 | Client 3 |
| 9 | 1172069693356 | 25205 | 3:lubm-Q06 | Client 3 |

Table B.2: Results for: 3 KBs, 3 parallel clients, 3 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172070998110 | 39842 | 1:lubm-Q02 | Client 1 |
| 2 | 1172071038249 | 55107 | 2:lubm-Q05 | Client 1 |
| 3 | 1172071093358 | 5153 | 3:lubm-Q06 | Client 1 |
| 4 | 1172070998110 | 39842 | 1:lubm-Q02 | Client 2 |
| 5 | 1172071038250 | 1160 | 2:lubm-Q05 | Client 2 |
| 6 | 1172071038371 | 2148 | 3:lubm-Q06 | Client 2 |
| 7 | 1172070998110 | 94092 | 1:lubm-Q02 | Client 3 |
| 8 | 1172071092205 | 1219 | 2:lubm-Q05 | Client 3 |
| 9 | 1172071093426 | 5179 | 3:lubm-Q06 | Client 3 |

Table B.3: Results for: 3 KBs, 3 parallel clients, 3 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172071413511 | 39783 | 1:lubm-Q02 | Client 1 |
| 2 | 1172071453582 | 1160 | 2:lubm-Q05 | Client 1 |
| 3 | 1172071453703 | 1832 | 3:lubm-Q06 | Client 1 |
| 4 | 1172071413511 | 39803 | 1:lubm-Q02 | Client 2 |
| 5 | 1172071453582 | 1741 | 2:lubm-Q05 | Client 2 |
| 6 | 1172071455325 | 1756 | 3:lubm-Q06 | Client 2 |
| 7 | 1172071413511 | 39783 | 1:lubm-Q02 | Client 3 |
| 8 | 1172071453582 | 1741 | 2:lubm-Q05 | Client 3 |
| 9 | 1172071455325 | 1815 | 3:lubm-Q06 | Client 3 |

Table B.4: Results for: 3 KBs, 3 parallel clients, 3 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172072589879 | 39900 | 1:lubm-Q02 | Client 1 |
| 2 | 1172072630040 | 1608 | 2:lubm-Q05 | Client 1 |
| 3 | 1172072631650 | 1792 | 3:lubm-Q06 | Client 1 |
| 4 | 1172072589879 | 39873 | 1:lubm-Q02 | Client 2 |
| 5 | 1172072630040 | 1173 | 2:lubm-Q05 | Client 2 |
| 6 | 1172072630165 | 1766 | 3:lubm-Q06 | Client 2 |
| 7 | 1172072589879 | 39873 | 1:lubm-Q02 | Client 3 |
| 8 | 1172072630041 | 1168 | 2:lubm-Q05 | Client 3 |
| 9 | 1172072630161 | 1853 | 3:lubm-Q06 | Client 3 |

Table B.5: Results for: 3 KBs, 3 parallel clients, 3 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172091072415 | 91720 | 1:lubm-Q02 | Client 1 |
| 2 | 1172091164138 | 80401 | 2:lubm-Q05 | Client 1 |
| 3 | 1172091244542 | 1286 | 3:lubm-Q06 | Client 1 |
| 4 | 1172091072415 | 38666 | 1:lubm-Q02 | Client 2 |
| 5 | 1172091111369 | 133170 | 2:lubm-Q05 | Client 2 |
| 6 | 1172091244542 | 2427 | 3:lubm-Q06 | Client 2 |
| 7 | 1172091072415 | 170524 | 1:lubm-Q02 | Client 3 |
| 8 | 1172091242942 | 1597 | 2:lubm-Q05 | Client 3 |
| 9 | 1172091244541 | 1389 | 3:lubm-Q06 | Client 3 |

Table B.6: Results for: 3 KBs, 3 parallel clients, 3 queries per client. Scenario with RR.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172092467512 | 39437 | 1:lubm-Q02 | Client 1 |
| 2 | 1172092506951 | 120213 | 2:lubm-Q05 | Client 1 |
| 3 | 1172092627167 | 340 | 3:lubm-Q06 | Client 1 |
| 4 | 1172092467512 | 91883 | 1:lubm-Q02 | Client 2 |
| 5 | 1172092559397 | 68243 | 2:lubm-Q05 | Client 2 |
| 6 | 1172092627642 | 3010 | 3:lubm-Q06 | Client 2 |
| 7 | 1172092467512 | 39097 | 1:lubm-Q02 | Client 3 |
| 8 | 1172092506900 | 37609 | 2:lubm-Q05 | Client 3 |
| 9 | 1172092544512 | 17809 | 3:lubm-Q06 | Client 3 |

Table B.7: Results for: 3 KBs, 3 parallel clients, 3 queries per client. Scenario with RR.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172093879671 | 39472 | 1:lubm-Q02 | Client 1 |
| 2 | 1172093919432 | 52154 | 2:lubm-Q05 | Client 1 |
| 3 | 1172093971588 | 40386 | 3:lubm-Q06 | Client 1 |
| 4 | 1172093879671 | 44489 | 1:lubm-Q02 | Client 2 |
| 5 | 1172093924162 | 1483 | 2:lubm-Q05 | Client 2 |
| 6 | 1172093925647 | 115406 | 3:lubm-Q06 | Client 2 |
| 7 | 1172093879671 | 39475 | 1:lubm-Q02 | Client 3 |
| 8 | 1172093919432 | 37775 | 2:lubm-Q05 | Client 3 |
| 9 | 1172093957213 | 18749 | 3:lubm-Q06 | Client 3 |

Table B.8: Results for: 3 KBs, 3 parallel clients, 3 queries per client. Scenario with RR.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172076059648 | 39591 | 1:lubm-Q01 | Client 1 |
| 2 | 1172076099527 | 67634 | 2:lubm-Q09 | Client 1 |
| 3 | 1172076060493 | 76534 | 1:lubm-Q01 | Client 2 |
| 4 | 1172076137030 | 60354 | 2:lubm-Q09 | Client 2 |
| 5 | 1172076061503 | 80458 | 1:lubm-Q02 | Client 3 |
| 6 | 1172076141963 | 55475 | 2:lubm-Q10 | Client 3 |
| 7 | 1172076062503 | 79678 | 1:lubm-Q02 | Client 4 |
| 8 | 1172076142183 | 55296 | 2:lubm-Q10 | Client 4 |
| 9 | 1172076063503 | 79650 | 1:lubm-Q05 | Client 5 |
| 10 | 1172076143155 | 67656 | 2:lubm-Q11 | Client 5 |
| 11 | 1172076064513 | 95467 | 1:lubm-Q05 | Client 6 |
| 12 | 1172076159983 | 50866 | 2:lubm-Q11 | Client 6 |
| 13 | 1172076065523 | 99219 | 1:lubm-Q06 | Client 7 |
| 14 | 1172076164744 | 46176 | 2:lubm-Q15 | Client 7 |
| 15 | 1172076066532 | 98317 | 1:lubm-Q06 | Client 8 |
| 16 | 1172076164851 | 46130 | 2:lubm-Q15 | Client 8 |
| 17 | 1172076067533 | 97453 | 1:lubm-Q07 | Client 9 |
| 18 | 1172076164988 | 46064 | 2:lubm-Q16 | Client 9 |
| 19 | 1172076068532 | 96531 | 1:lubm-Q07 | Client 10 |
| 20 | 1172076165066 | 46045 | 2:lubm-Q16 | Client 10 |

Table B.9: Results for: 2 KBs, 10 parallel clients, 2 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172076425651 | 39062 | 1:lubm-Q01 | Client 1 |
| 2 | 1172076465000 | 6707 | 2:lubm-Q09 | Client 1 |
| 3 | 1172076426489 | 38224 | 1:lubm-Q01 | Client 2 |
| 4 | 1172076465000 | 8459 | 2:lubm-Q09 | Client 2 |
| 5 | 1172076427499 | 41482 | 1:lubm-Q02 | Client 3 |
| 6 | 1172076468983 | 2717 | 2:lubm-Q10 | Client 3 |
| 7 | 1172076428499 | 40301 | 1:lubm-Q02 | Client 4 |
| 8 | 1172076468802 | 6652 | 2:lubm-Q10 | Client 4 |
| 9 | 1172076429500 | 40412 | 1:lubm-Q05 | Client 5 |
| 10 | 1172076469914 | 1863 | 2:lubm-Q11 | Client 5 |
| 11 | 1172076430509 | 39198 | 1:lubm-Q05 | Client 6 |
| 12 | 1172076469709 | 15268 | 2:lubm-Q11 | Client 6 |
| 13 | 1172076431519 | 40005 | 1:lubm-Q06 | Client 7 |
| 14 | 1172076471526 | 369 | 2:lubm-Q15 | Client 7 |
| 15 | 1172076432529 | 38705 | 1:lubm-Q06 | Client 8 |
| 16 | 1172076471236 | 14223 | 2:lubm-Q15 | Client 8 |
| 17 | 1172076433529 | 38137 | 1:lubm-Q07 | Client 9 |
| 18 | 1172076471668 | 362 | 2:lubm-Q16 | Client 9 |
| 19 | 1172076434529 | 36856 | 1:lubm-Q07 | Client 10 |
| 20 | 1172076471387 | 14132 | 2:lubm-Q16 | Client 10 |

Table B.10: Results for: 2 KBs, 10 parallel clients, 2 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172097727578 | 37069 | 1:lubm-Q01 | Client 1 |
| 2 | 1172097764930 | 221589 | 2:lubm-Q09 | Client 1 |
| 3 | 1172097728424 | 73800 | 1:lubm-Q01 | Client 2 |
| 4 | 1172097802229 | 226445 | 2:lubm-Q09 | Client 2 |
| 5 | 1172097729433 | 117350 | 1:lubm-Q02 | Client 3 |
| 6 | 1172097846786 | 181941 | 2:lubm-Q10 | Client 3 |
| 7 | 1172097730433 | 146712 | 1:lubm-Q02 | Client 4 |
| 8 | 1172097877148 | 151619 | 2:lubm-Q10 | Client 4 |
| 9 | 1172097731433 | 172349 | 1:lubm-Q05 | Client 5 |
| 10 | 1172097903784 | 125080 | 2:lubm-Q11 | Client 5 |
| 11 | 1172097732443 | 199434 | 1:lubm-Q05 | Client 6 |
| 12 | 1172097931879 | 97011 | 2:lubm-Q11 | Client 6 |
| 13 | 1172097733453 | 202550 | 1:lubm-Q06 | Client 7 |
| 14 | 1172097936005 | 92953 | 2:lubm-Q15 | Client 7 |
| 15 | 1172097734464 | 226700 | 1:lubm-Q06 | Client 8 |
| 16 | 1172097961167 | 67854 | 2:lubm-Q15 | Client 8 |
| 17 | 1172097735463 | 226943 | 1:lubm-Q07 | Client 9 |
| 18 | 1172097962408 | 66682 | 2:lubm-Q16 | Client 9 |
| 19 | 1172097736463 | 247956 | 1:lubm-Q07 | Client 10 |
| 20 | 1172097984422 | 44726 | 2:lubm-Q16 | Client 10 |

Table B.11: Results for: 2 KBs, 10 parallel clients, 2 queries per client. Scenario with RR.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172096656502 | 37588 | 1:lubm-Q01 | Client 1 |
| 2 | 1172096694382 | 95185 | 2:lubm-Q09 | Client 1 |
| 3 | 1172096657351 | 36739 | 1:lubm-Q01 | Client 2 |
| 4 | 1172096694382 | 98240 | 2:lubm-Q09 | Client 2 |
| 5 | 1172096658351 | 62352 | 1:lubm-Q02 | Client 3 |
| 6 | 1172096720707 | 49388 | 2:lubm-Q10 | Client 3 |
| 7 | 1172096659351 | 62199 | 1:lubm-Q02 | Client 4 |
| 8 | 1172096721552 | 94885 | 2:lubm-Q10 | Client 4 |
| 9 | 1172096660361 | 80091 | 1:lubm-Q05 | Client 5 |
| 10 | 1172096740455 | 53848 | 2:lubm-Q11 | Client 5 |
| 11 | 1172096661371 | 79674 | 1:lubm-Q05 | Client 6 |
| 12 | 1172096741047 | 88258 | 2:lubm-Q11 | Client 6 |
| 13 | 1172096662371 | 81180 | 1:lubm-Q06 | Client 7 |
| 14 | 1172096743553 | 63749 | 2:lubm-Q15 | Client 7 |
| 15 | 1172096663380 | 80875 | 1:lubm-Q06 | Client 8 |
| 16 | 1172096744257 | 63111 | 2:lubm-Q15 | Client 8 |
| 17 | 1172096664380 | 79312 | 1:lubm-Q07 | Client 9 |
| 18 | 1172096743694 | 85798 | 2:lubm-Q16 | Client 9 |
| 19 | 1172096665391 | 79000 | 1:lubm-Q07 | Client 10 |
| 20 | 1172096744393 | 97941 | 2:lubm-Q16 | Client 10 |

Table B.12: Results for: 2 KBs, 10 parallel clients, 2 queries per client. Scenario with RR.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|

Table B.13: Results for: 2 KBs, 6 parallel clients, 3 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172339927367 | 3194 | 1:lubm-Q02 | Client 1 |
| 2 | 1172339930845 | 126 | 2:lubm-Q02 | Client 1 |
| 3 | 1172339930973 | 90 | 3:lubm-Q02 | Client 1 |
| 4 | 1172339929223 | 10217 | 1:lubm-Q06 | Client 2 |
| 5 | 1172339939443 | 238 | 2:lubm-Q06 | Client 2 |
| 6 | 1172339939683 | 138 | 3:lubm-Q06 | Client 2 |
| 7 | 1172339931222 | 8568 | 1:lubm-Q07 | Client 3 |
| 8 | 1172339939792 | 191 | 2:lubm-Q07 | Client 3 |
| 9 | 1172339939985 | 207 | 3:lubm-Q07 | Client 3 |
| 10 | 1172339928233 | 3033 | 1:lubm-Q02 | Client 4 |
| 11 | 1172339931268 | 60 | 2:lubm-Q02 | Client 4 |
| 12 | 1172339931330 | 55 | 3:lubm-Q02 | Client 4 |
| 13 | 1172339930232 | 9860 | 1:lubm-Q06 | Client 5 |
| 14 | 1172339940094 | 140 | 2:lubm-Q06 | Client 5 |
| 15 | 1172339940235 | 126 | 3:lubm-Q06 | Client 5 |
| 16 | 1172339932242 | 8065 | 1:lubm-Q07 | Client 6 |
| 17 | 1172339940310 | 178 | 2:lubm-Q07 | Client 6 |
| 18 | 1172339940490 | 158 | 3:lubm-Q07 | Client 6 |

Table B.14: Results for: 2 KBs, 6 parallel clients, 3 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172103554200 | 3170 | 1:lubm-Q02 | Client 1 |
| 2 | 1172103557655 | 15347 | 2:lubm-Q02 | Client 1 |
| 3 | 1172103555049 | 3349 | 1:lubm-Q05 | Client 2 |
| 4 | 1172103558399 | 15002 | 2:lubm-Q05 | Client 2 |
| 5 | 1172103556047 | 11005 | 1:lubm-Q06 | Client 3 |
| 6 | 1172103567054 | 6396 | 2:lubm-Q06 | Client 3 |
| 7 | 1172103557057 | 14223 | 1:lubm-Q09 | Client 4 |
| 8 | 1172103571282 | 2946 | 2:lubm-Q09 | Client 4 |
| 9 | 1172103558056 | 15029 | 1:lubm-Q10 | Client 5 |
| 10 | 1172103573087 | 1085 | 2:lubm-Q10 | Client 5 |
| 11 | 1172103559066 | 3084 | 1:lubm-Q02 | Client 6 |
| 12 | 1172103562152 | 15698 | 2:lubm-Q02 | Client 6 |
| 13 | 1172103560067 | 3114 | 1:lubm-Q05 | Client 7 |
| 14 | 1172103563183 | 14860 | 2:lubm-Q05 | Client 7 |
| 15 | 1172103561086 | 10812 | 1:lubm-Q06 | Client 8 |
| 16 | 1172103571900 | 6240 | 2:lubm-Q06 | Client 8 |
| 17 | 1172103562086 | 14009 | 1:lubm-Q09 | Client 9 |
| 18 | 1172103576097 | 2804 | 2:lubm-Q09 | Client 9 |
| 19 | 1172103563086 | 14830 | 1:lubm-Q10 | Client 10 |
| 20 | 1172103577918 | 935 | 2:lubm-Q10 | Client 10 |

Table B.15: Results for: 2 KBs, 10 parallel clients, 2 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172103074526 | 3253 | 1:lubm-Q02 | Client 1 |
| 2 | 1172103078064 | 155 | 2:lubm-Q02 | Client 1 |
| 3 | 1172103075377 | 3432 | 1:lubm-Q05 | Client 2 |
| 4 | 1172103078811 | 225 | 2:lubm-Q05 | Client 2 |
| 5 | 1172103076377 | 11119 | 1:lubm-Q06 | Client 3 |
| 6 | 1172103087498 | 121 | 2:lubm-Q06 | Client 3 |
| 7 | 1172103077387 | 14327 | 1:lubm-Q09 | Client 4 |
| 8 | 1172103091716 | 177 | 2:lubm-Q09 | Client 4 |
| 9 | 1172103078386 | 14932 | 1:lubm-Q10 | Client 5 |
| 10 | 1172103093321 | 69 | 2:lubm-Q10 | Client 5 |
| 11 | 1172103079396 | 2992 | 1:lubm-Q02 | Client 6 |
| 12 | 1172103082391 | 114 | 2:lubm-Q02 | Client 6 |
| 13 | 1172103080397 | 3003 | 1:lubm-Q05 | Client 7 |
| 14 | 1172103083402 | 182 | 2:lubm-Q05 | Client 7 |
| 15 | 1172103081406 | 10757 | 1:lubm-Q06 | Client 8 |
| 16 | 1172103092165 | 117 | 2:lubm-Q06 | Client 8 |
| 17 | 1172103082417 | 13937 | 1:lubm-Q09 | Client 9 |
| 18 | 1172103096356 | 146 | 2:lubm-Q09 | Client 9 |
| 19 | 1172103083416 | 14640 | 1:lubm-Q10 | Client 10 |
| 20 | 1172103098059 | 60 | 2:lubm-Q10 | Client 10 |

Table B.16: Results for: 2 KBs, 10 parallel clients, 2 queries per client. Scenario with ACO.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172268204037 | 1308 | 1:getAssociateProfessor | Client 1 |
| 2 | 1172268205628 | 1394 | 2:getProfessor | Client 1 |
| 3 | 1172268207024 | 873 | 3:getFaculty | Client 1 |
| 4 | 1172268207898 | 888 | 4:getPerson | Client 1 |

Table B.17: Results for: 1 KBs, 1 parallel clients, 4 queries per client. Scenario with 1 reasoners.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172267468647 | 1152 | 1:getAssociateProfessor | Client 1 |
| 2 | 1172267470081 | 1628 | 2:getProfessor | Client 1 |
| 3 | 1172267471711 | 1357 | 3:getFaculty | Client 1 |
| 4 | 1172267473070 | 1331 | 4:getPerson | Client 1 |

Table B.18: Results for: 1 KBs, 1 parallel clients, 4 queries per client. Scenario with 1 reasoners.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172270925151 | 1310 | 1:getAssociateProfessor | Client 1 |
| 2 | 1172270926743 | 9814 | 2:getProfessor | Client 1 |
| 3 | 1172270936562 | 6637 | 3:getFaculty | Client 1 |
| 4 | 1172270943202 | 2294 | 4:getPerson | Client 1 |
| 5 | 1172270926010 | 12501 | 1:lubm-Q02 | Client 2 |
| 6 | 1172270938513 | 10995 | 2:lubm-Q05 | Client 2 |
| 7 | 1172270949510 | 4824 | 3:lubm-Q06 | Client 2 |
| 8 | 1172270954336 | 2303 | 4:lubm-Q07 | Client 2 |
| 9 | 1172270926011 | 18309 | 1:lubm-Q09 | Client 3 |
| 10 | 1172270944322 | 4601 | 2:lubm-Q10 | Client 3 |
| 11 | 1172270948926 | 326 | 3:lubm-Q11 | Client 3 |
| 12 | 1172270949254 | 2762 | 4:lubm-Q15 | Client 3 |

Table B.19: Results for: 1 KBs, 3 parallel clients, 4 queries per client. Scenario with 1 reasoners.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172276253177 | 1275 | 1:getAssociateProfessor | Client 1 |
| 2 | 1172276254740 | 10060 | 2:getProfessor | Client 1 |
| 3 | 1172276266802 | 4704 | 3:getFaculty | Client 1 |
| 4 | 1172276271508 | 4077 | 4:getPerson | Client 1 |
| 5 | 1172276254026 | 6429 | 1:lubm-Q02 | Client 2 |
| 6 | 1172276260457 | 9297 | 2:lubm-Q05 | Client 2 |
| 7 | 1172276269757 | 3786 | 3:lubm-Q06 | Client 2 |
| 8 | 1172276273545 | 3017 | 4:lubm-Q07 | Client 2 |
| 9 | 1172276254035 | 10148 | 1:lubm-Q09 | Client 3 |
| 10 | 1172276264185 | 6543 | 2:lubm-Q10 | Client 3 |
| 11 | 1172276270730 | 2361 | 3:lubm-Q11 | Client 3 |
| 12 | 1172276273093 | 2705 | 4:lubm-Q15 | Client 3 |

Table B.20: Results for: 1 KBs, 3 parallel clients, 4 queries per client. Scenario with 1 reasoners.

| No. | TimeStamp | Response Time | Query | Client |
|---|---|---|---|---|
| 1 | 1172270196836 | 1197 | 1:getAssociateProfessor | Client 1 |
| 2 | 1172270198324 | 6744 | 2:getProfessor | Client 1 |
| 3 | 1172270205070 | 13319 | 3:getFaculty | Client 1 |
| 4 | 1172270218393 | 1574 | 4:getPerson | Client 1 |
| 5 | 1172270197687 | 3074 | 1:lubm-Q02 | Client 2 |
| 6 | 1172270200764 | 14063 | 2:lubm-Q05 | Client 2 |
| 7 | 1172270214830 | 4311 | 3:lubm-Q06 | Client 2 |
| 8 | 1172270219143 | 949 | 4:lubm-Q07 | Client 2 |
| 9 | 1172270197697 | 5218 | 1:lubm-Q09 | Client 3 |
| 10 | 1172270202917 | 14022 | 2:lubm-Q10 | Client 3 |
| 11 | 1172270216941 | 1662 | 3:lubm-Q11 | Client 3 |
| 12 | 1172270218605 | 805 | 4:lubm-Q15 | Client 3 |

Table B.21: Results for: 1 KBs, 3 parallel clients, 4 queries per client. Scenario with 1 reasoners.

# Appendix C

# Benchmarking Result Charts

## C.1 Charts for 7.2.1 Benchmark 1

### C.1.1 ACO-LB Result Charts

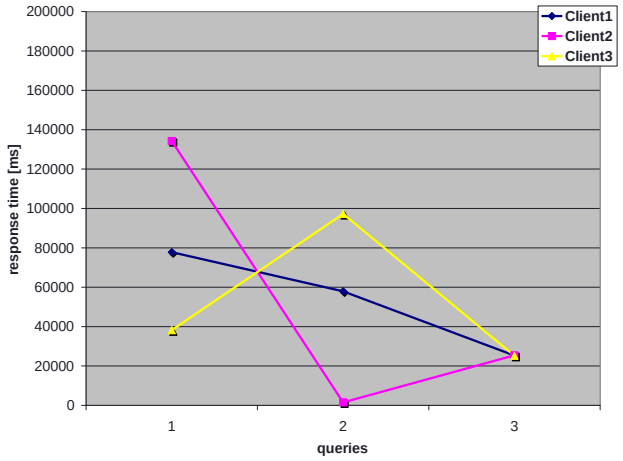


Figure C.1: Scenario: 3 concurrent clients send 3 pairwise different queries sequentially (the next query is send when the answer to the previous was received) to 1KB each (i.e. to 3 KBs in total). The 3 knowledge bases are identical. System Setting: ACO-LB manages 1 reasoner. The graph shows the development of the response times from a the perspective of each client.

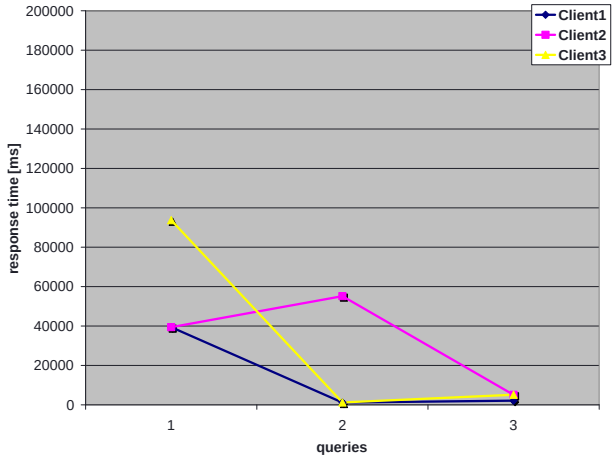


Figure C.2: Scenario: 3 concurrent clients send 3 pairwise different queries sequentially (the next query is send when the answer to the previous was received) to 1KB each (i.e. to 3 KBs in total). The 3 knowledge bases are identical. System Setting: ACO-LB manages 2 reasoners. The graph shows the development of the response times from a the perspective of each client.

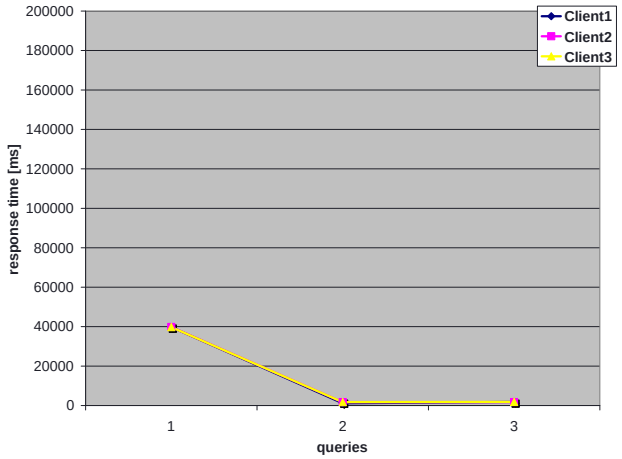


Figure C.3: Scenario: 3 concurrent clients send 3 pairwise different queries sequentially (the next query is send when the answer to the previous was received) to 1KB each (i.e. to 3 KBs in total). The 3 knowledge bases are identical. System Setting: ACO-LB manages 3 reasoners. The graph shows the development of the response times from a the perspective of each client.

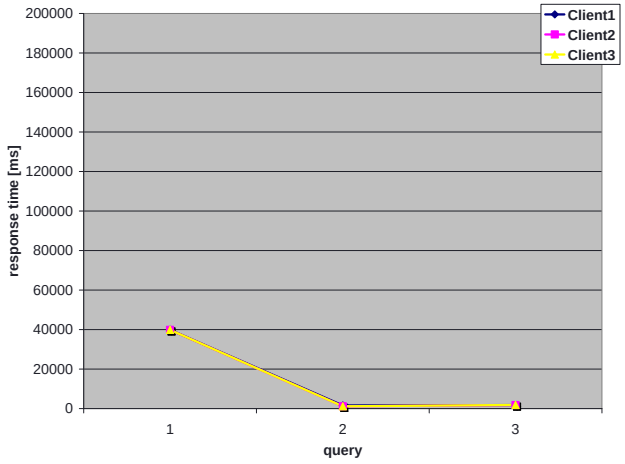


Figure C.4: Scenario: 3 concurrent clients send 3 pairwise different queries sequentially (the next query is send when the answer to the previous was received) to 1KB each (i.e. to 3 KBs in total). The 3 knowledge bases are identical. System Setting: ACO-LB manages 4 reasoners. The graph shows the development of the response times from a the perspective of each client. *Important to notice here is that there is no change at all to the measured times in the 3 reasoner scenario, Fig. C.3.*

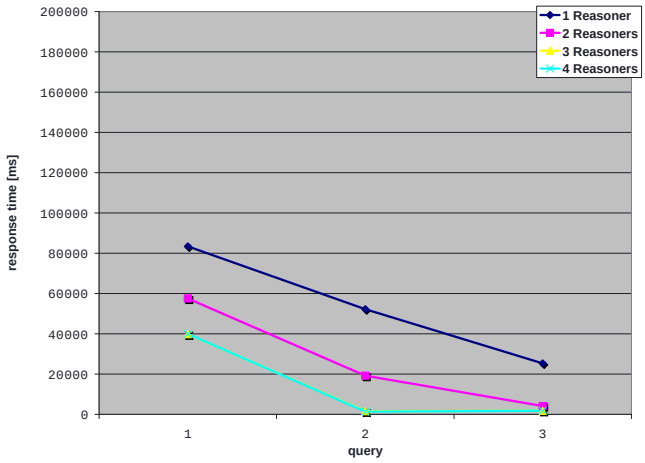


Figure C.5: The chart compares the development of the average ACO-LB query response times for each of the settings shown in figures C.1, C.2, C.3 and C.4. *Notice that the graph for "3 reasoners" is covered by the one for "4 reasoners".*

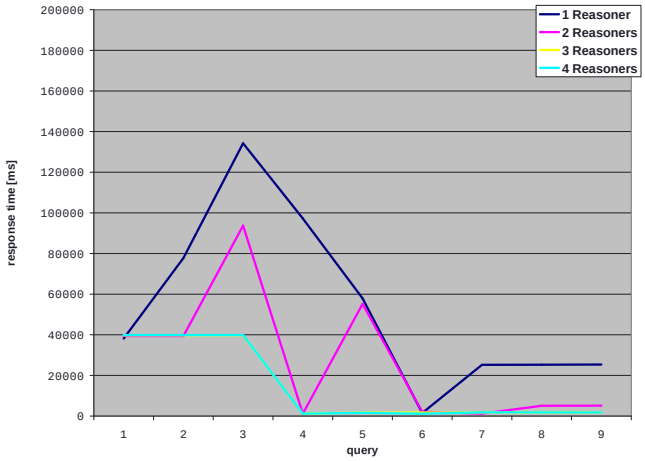


Figure C.6: The chart compares the development of the query response times for ACO-LB from the system's perspective. It shows the development in the order in which the queries where answered by the reasoner. *Note that the order 1,2,3,4... of the queries in the graph do not correspond to the order in the data tables. It is based on the, by time-stamp reordered tables.*

### C.1.2 RoundRobin Result Charts

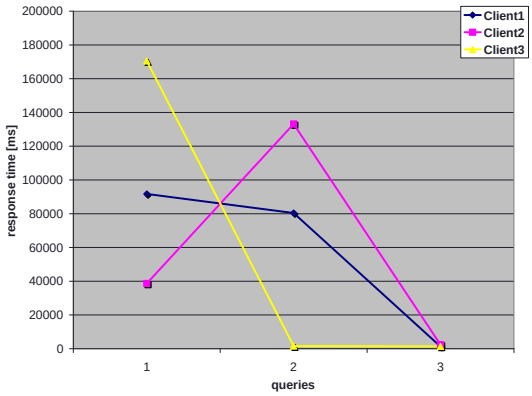


Figure C.7: Scenario: 3 concurrent clients send 3 pairwise different queries sequentially (the next query is send when the answer to the previous was received) to 1KB each (i.e. to 3 KBs in total). System Setting: RoundRobin manages 1 reasoner. The graph shows the development of the response times from a the perspective of each client.

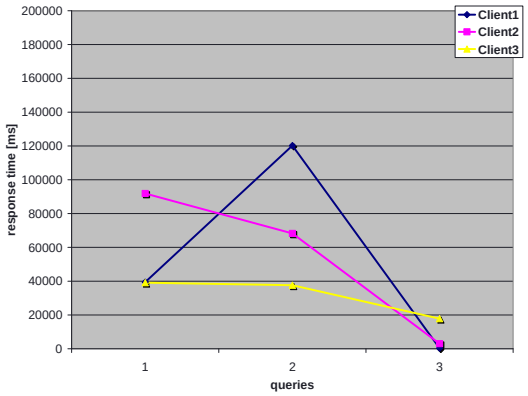


Figure C.8: Scenario: 3 concurrent clients send 3 pairwise different queries sequentially (the next query is send when the answer to the previous was received) to 1KB each (i.e. to 3 KBs in total). System Setting: RoundRobin manages 2 reasoners. The graph shows the development of the response times from a the perspective of each client.

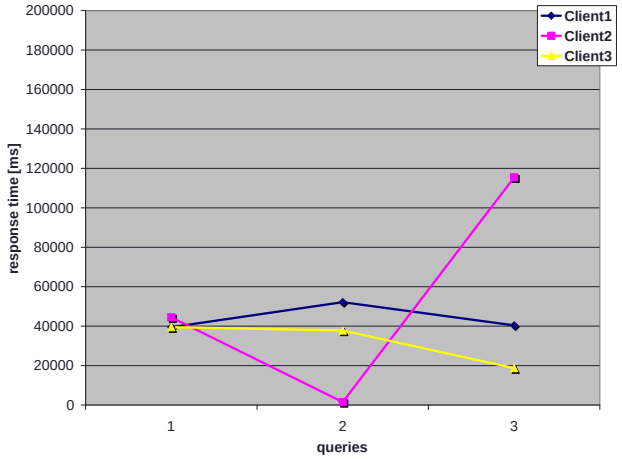


Figure C.9: Scenario: 3 concurrent clients send 3 pairwise different queries sequentially (the next query is send when the answer to the previous was received) to 1KB each (i.e. to 3 KBs in total). System Setting: RoundRobin manages 3 reasoners. The graph shows the development of the response times from a the perspective of each client.

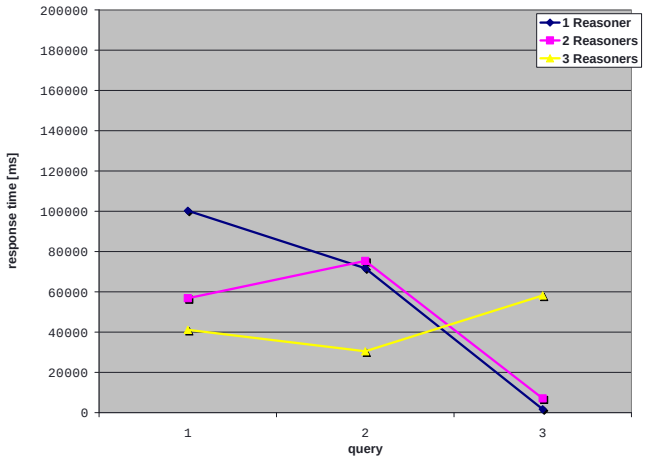


Figure C.10: The chart compares the development of the average RoundRobin query response times of each of the settings shown in figures C.7, C.8 and C.9.

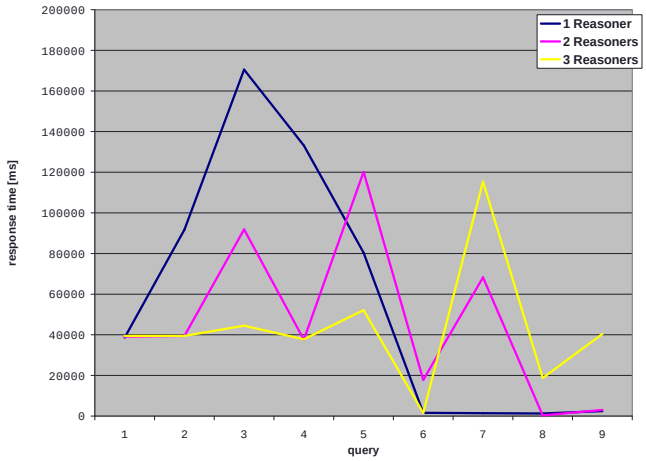


Figure C.11: The chart compares the development of the query response times for RR from the system's perspective. It shows the development in the order in which the queries where answered by the reasoner. *Note that the order 1,2,3,4... of the queries in the graph do not correspond to the order in the data tables. It is based on the, by time-stamp reordered tables.*

### C.1.3 Comparison of the ACO-LB and RoundRobin Results

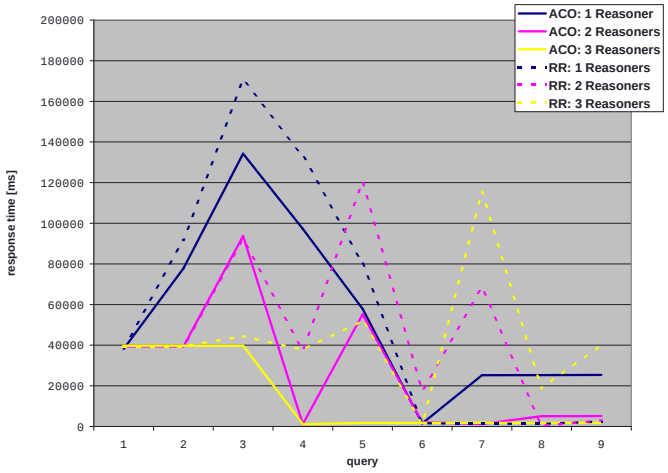


Figure C.12: The chart shows a layering of the figures C.6 and C.11 in order to compare the performance of an ACO-LB managed system and a system with a pure RoundRobin.

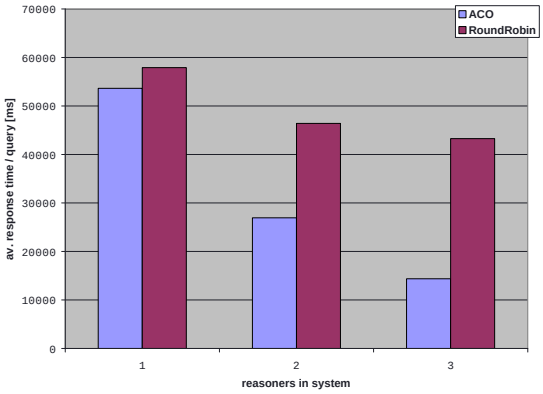


Figure C.13: The column chart compare the average response times for the query of the ACO-LB balanced and the RR balanced system for the a 1, 2 and 3 reasoners setting.

## C.2 Charts for 7.2.2 Benchmark 2

### C.2.1 ACO-LB Result Charts

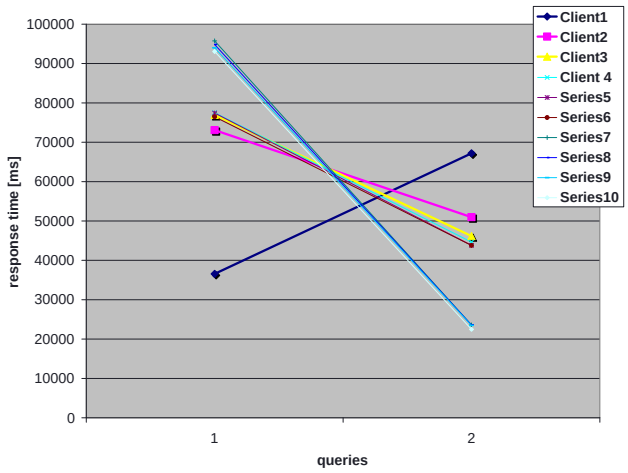


Figure C.14: Scenario: 10 concurrent clients send 2 queries each sequentially(the next query is send when the answer to the previous was received) to 2KBs. 5 clients to KB1, 5 clients to KB2. The 2 knowledge bases are identical. The queries to KB1 are the same as to KB2. System Setting: ACO-LB manages 1 reasoner. The graph shows the development of the response times from a the perspective of each client.

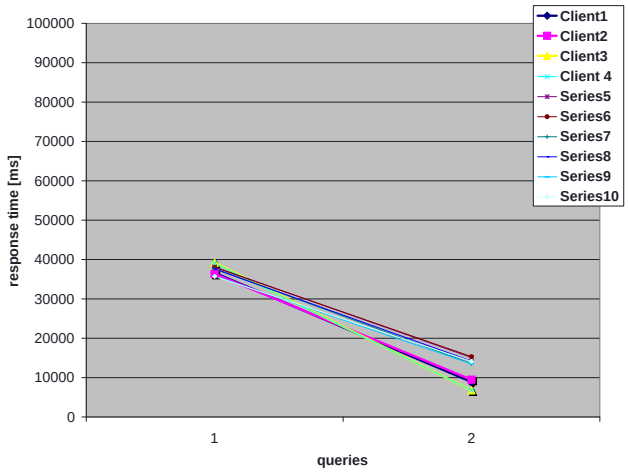


Figure C.15: Scenario: 10 concurrent clients send 2 queries each sequentially(the next query is send when the answer to the previous was received) to 2KBs. 5 clients to KB1, 5 clients to KB2. The 2 knowledge bases are identical. The queries to KB1 are the same as to KB2. System Setting: ACO-LB manages 2 reasoners. The graph shows the development of the response times from a the perspective of each client.

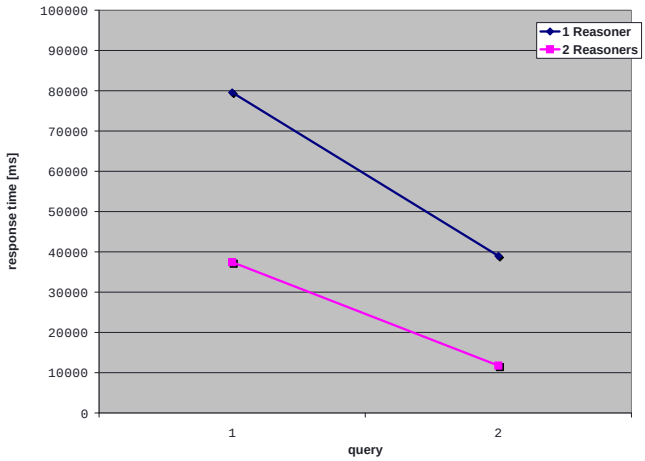


Figure C.16: The chart compares the development of the average ACO-LB query response times for each of the settings shown in figures C.14 and C.15.

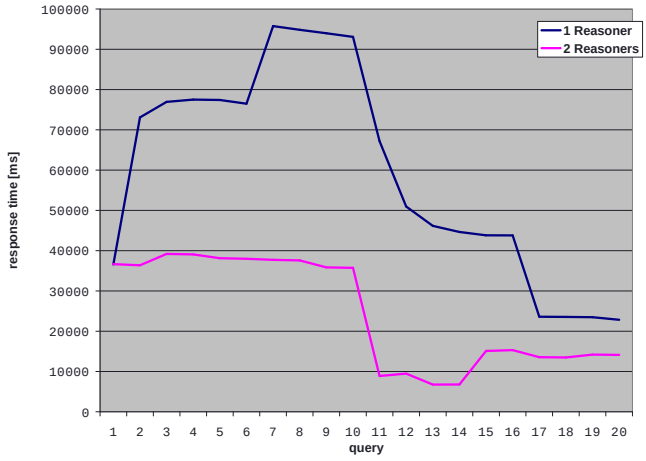


Figure C.17: The chart compares the development of the query response times for ACO-LB from the system's perspective. It shows the development in the order in which the queries where answered by the reasoner. *Note that the order 1,2,3,4... of the queries in the graph do not correspond to the order in the data tables. The charts is based on the, by time-stamp reordered tables.*

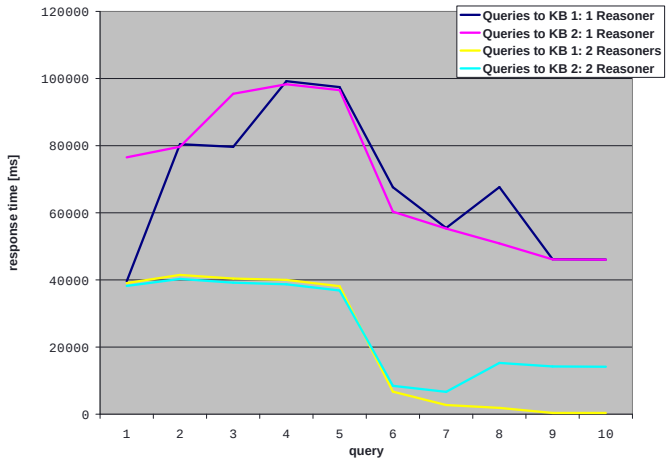


Figure C.18: This chart shows the development from the same perspective as fig. C.17 but for each KB individually. The graph for 1 reasoner of fig. C.17 thus splits into 2 graphs, as does the one for 2 reasoners. Consequently this chart can only show the development over 10 queries, because the 20 queries that reach the whole system split into 10 per KB.

### C.2.2 RoundRobin Result Charts

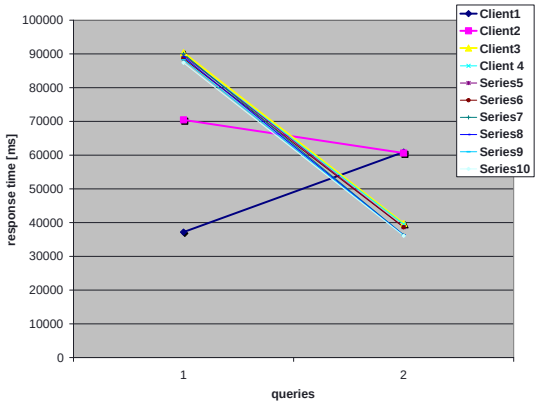


Figure C.19: Scenario: 10 concurrent clients send 2 queries each sequentially (the next query is send when the answer to the previous was received) to 2KBs. 5 clients to KB1, 5 clients to KB2. The 2 knowledge bases are identical. The queries to KB1 are the same as to KB2. System Setting: RoundRobin manages 1 reasoner. The graph shows the development of the response times from the perspective of each client.

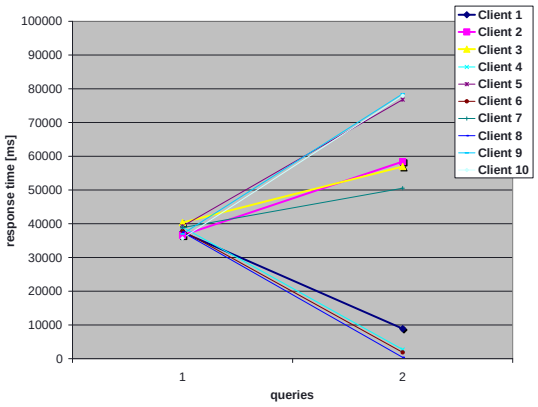


Figure C.20: Scenario: 10 concurrent clients send 2 queries each sequentially (the next query is send when the answer to the previous was received) to 2KBs. 5 clients to KB1, 5 clients to KB2. The 2 knowledge bases are identical. The queries to KB1 are the same as to KB2. System Setting: RoundRobin manages 2 reasoners. The graph shows the development of the response times from the perspective of each client.

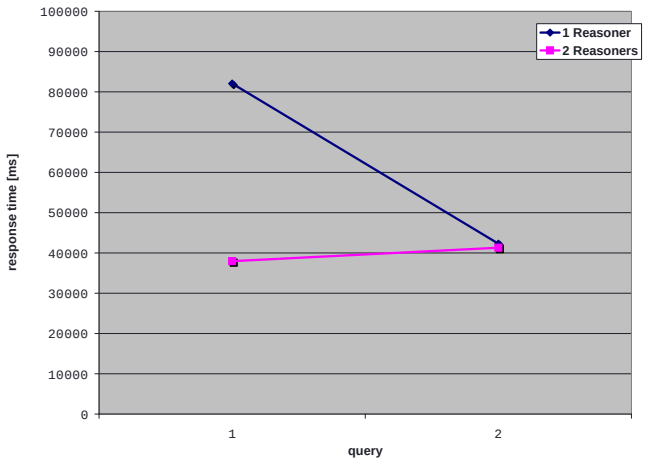


Figure C.21: The chart compares the development of the average RR query response times for each of the settings shown in figures C.19 and C.20 .

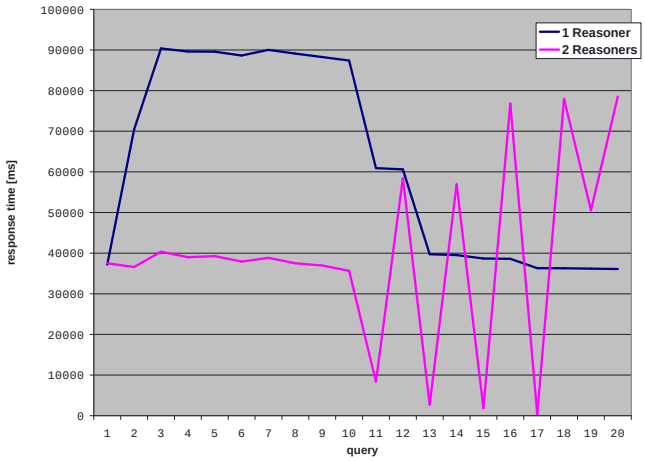


Figure C.22: The chart compares the development of the query response times for RR from the system's perspective. It shows the development in the order in which the queries where answered by the reasoner. *Note that the order 1,2,3,4... of the queries in the graph do not correspond to the order in the data tables. The charts is based on the, by time-stamp reordered tables.*

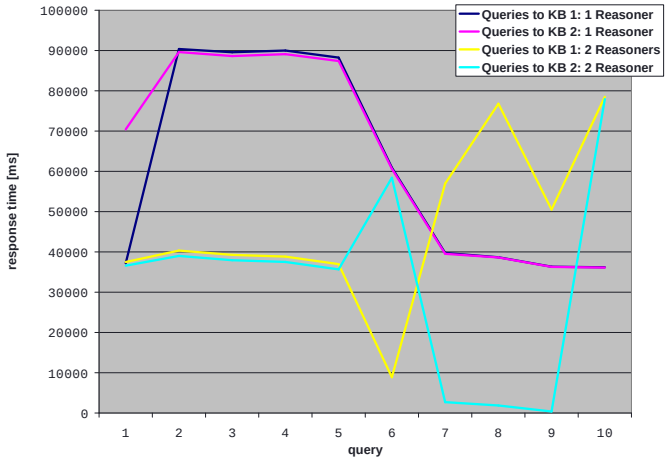


Figure C.23: This chart shows the development from the same perspective as fig. C.22 but for each KB individually. The graph for 1 reasoner of fig. C.22 thus splits into 2 graphs, as does the one for 2 reasoners. Consequently this chart can only show the development over 10 queries, because the 20 queries that reach the whole system split into 10 per KB.

### C.2.3 Comparison of the ACO and RoundRobin Results

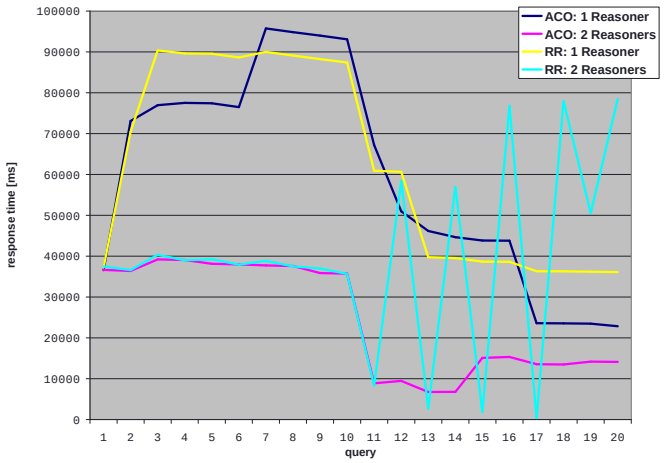


Figure C.24: The chart shows a layering of the figures C.17 and C.22 in order to compare the performance of an ACO-LB managed system and a system with a pure RoundRobin.

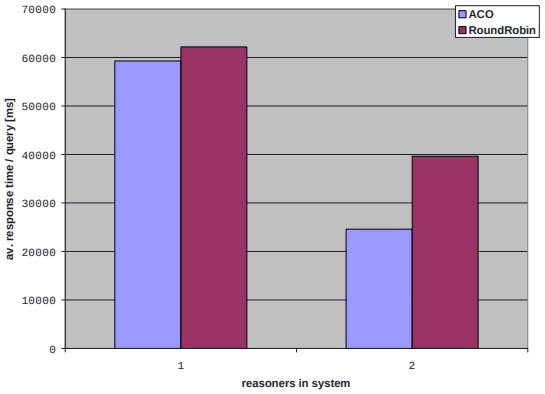


Figure C.25: The column chart compare the average response times for the query of the ACO-LB balanced and the RR balanced system for the a 1 and 2 reasoners setting.

## C.3 Charts for 7.3.1 Benchmark 3

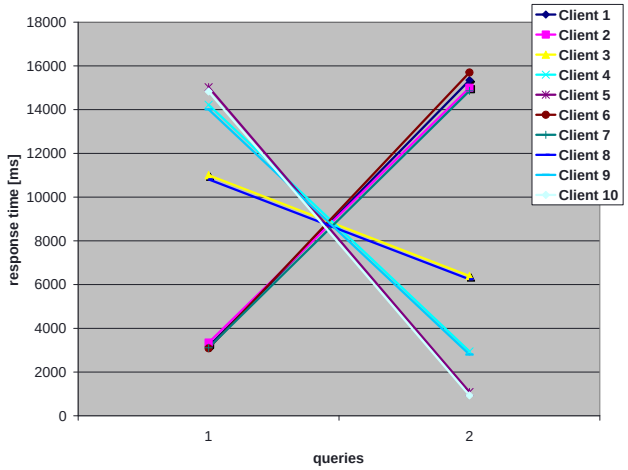


Figure C.26: Scenario: 10 concurrent clients, each of which sequentially (the next query is send when the answer to the previous was received) sends 2 times the same query. 5 clients send to KB1, 3 clients to KB2. The 2 knowledge bases are identical, have been previous loaded - each on one of the reasoners - and the index structures have been built. The queries to KB1 are the same as to KB2, but are pairwise different to the other queries to the same knowledge base. *System Setting: a pure ACO-LB* ***without*** *cache manages 2 reasoners.* The graph shows on the one hand the response times from the client perspective in comparison to the others and on the other hand compares the first time a query was answered with the second (as in this case a client is identical to a particular query). *Note: The first time the query arrives it is new to the system and the reasoner, the second time the query arrives it is known by system/reasoner.*

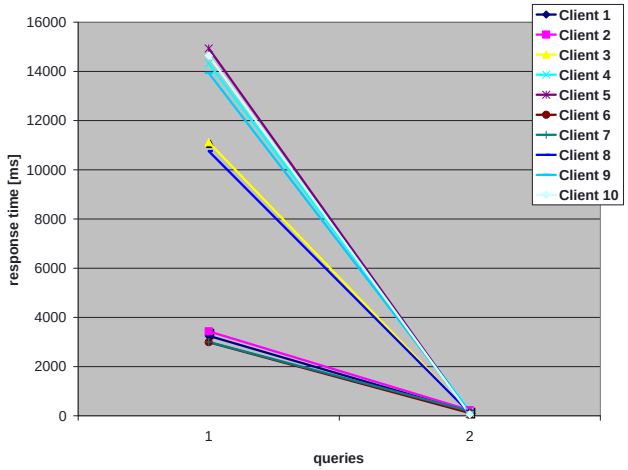


Figure C.27: Scenario: 10 concurrent clients, each of which sequentially (the next query is send when the answer to the previous was received) sends 2 times the same query. 5 clients send to KB1, 3 clients to KB2. The 2 knowledge bases are identical, have been previous loaded - each on one of the reasoners - and the index structures have been built. The queries to KB1 are the same as to KB2, but are pairwise different to the other queries to the same knowledge base. *System Setting: a pure ACO-LB* ***with*** *cache manages 2 reasoners.* The graph shows on the one hand the response times from the client perspective in comparison to the others and on the other hand compares the first time a query was answered with the second (as in this case a client is identical to a particular query). *Note: The first time the query arrives it is new to the system and the reasoner, the second time the query arrives it is known by system/reasoner.*

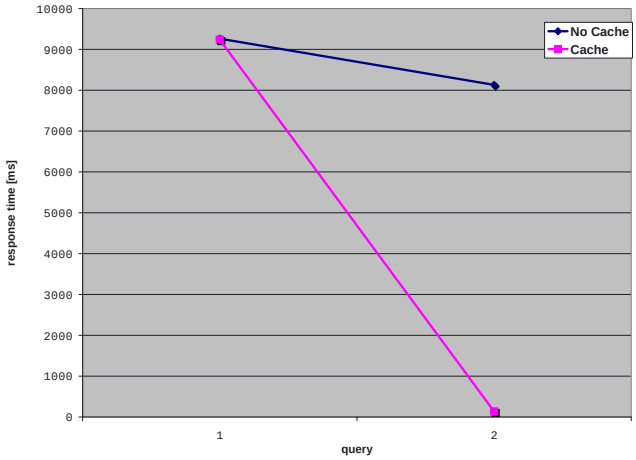


Figure C.28: The chart compares the development of the average query response times shown in figures C.26 and C.27 for a system with cache and without cache.

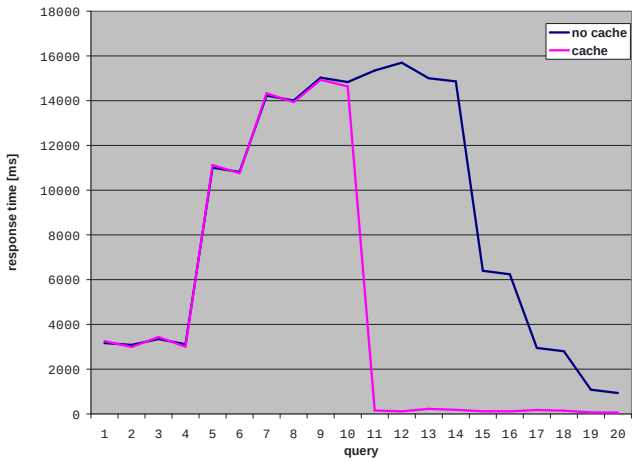


Figure C.29: The chart compares the development of the query response times for a system with and without a cache - from the system's perspective. It shows the development in the order in which the queries where answered by the reasoner. *Note that queries 1-10 are the same as 11-20.*

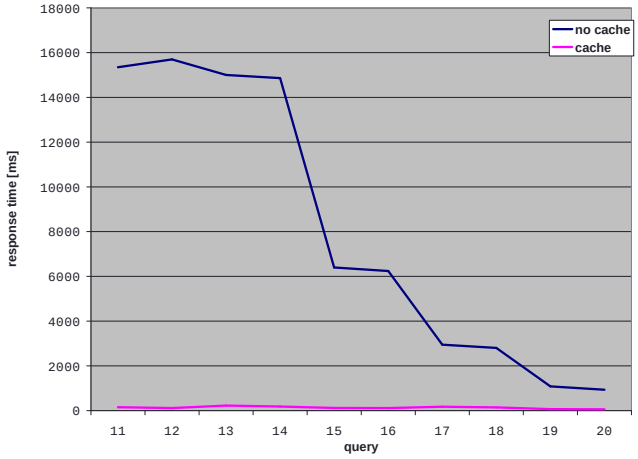


Figure C.30: The chart shows the same comparison as fig. C.29, but focuses on the queries 11-20.

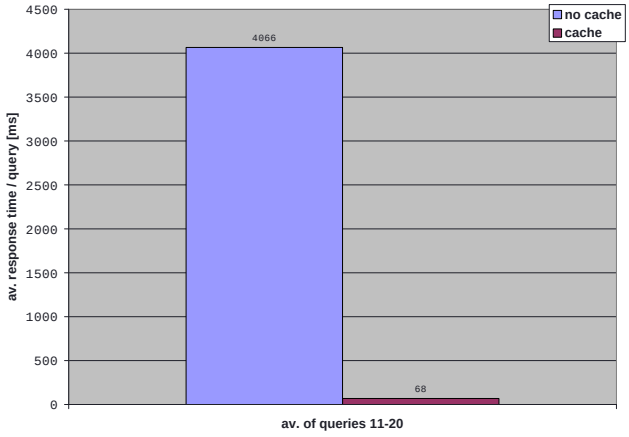


Figure C.31: The column chart compares the average response times of the queries 11-20 as they are shown in fig. C.30.

## C.4 Charts for 7.3.2 Benchmark 4

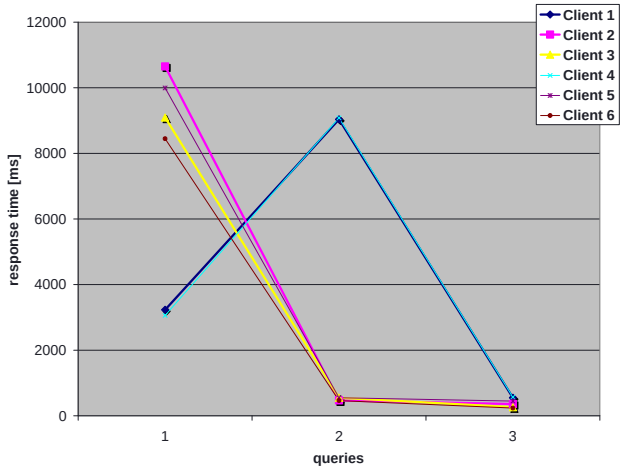


Figure C.32: Scenario: 6 concurrent clients, each of which sequentially (the next query is send when the answer to the previous was received) sends 3 times the same query. 3 clients send to KB1, 3 clients to KB2. The 2 knowledge bases are identical, have been previous loaded - each on one of the reasoners - and the index structures have been built. The queries to KB1 are the same as to KB2, but are pairwise different to the other queries to the same knowledge base. *System Setting: a pure ACO-LB* ***without*** *cache manages 2 reasoners.* The graph shows on the one hand the response times from the client perspective in comparison to the others and on the other hand compares the first time a query was answered with the second and third (as in this case a client is identical to a particular query). *Note: The first time the query arrives it is new to the system and the reasoner, the second and third time the query arrives it is known by system/reasoner.*

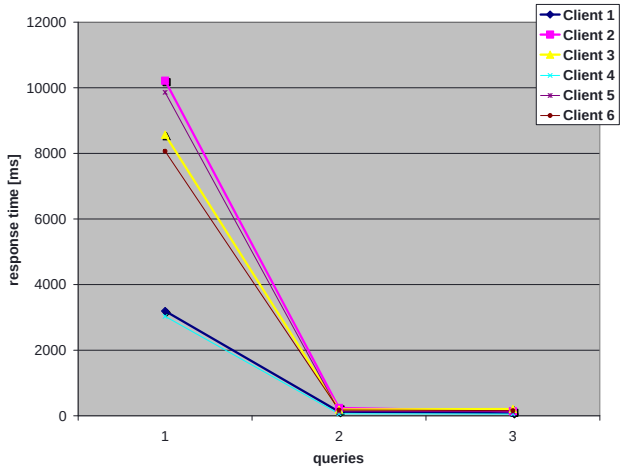


Figure C.33: Scenario: 6 concurrent clients, each of which sequentially (the next query is send when the answer to the previous was received) sends 3 times the same query. 3 clients send to KB1, 3 clients to KB2. The 2 knowledge bases are identical, have been previous loaded - each on one of the reasoners - and the index structures have been built. The queries to KB1 are the same as to KB2, but are pairwise different to the other queries to the same knowledge base. *System Setting: a pure ACO-LB* ***with*** *cache manages 2 reasoners.* The graph shows on the one hand the response times from the client perspective in comparison to the others and on the other hand compares the first time a query was answered with the second and third (as in this case a client is identical to a particular query). *Note: The first time the query arrives it is new to the system and the reasoner, the second and third time the query arrives it is known by system/reasoner.*

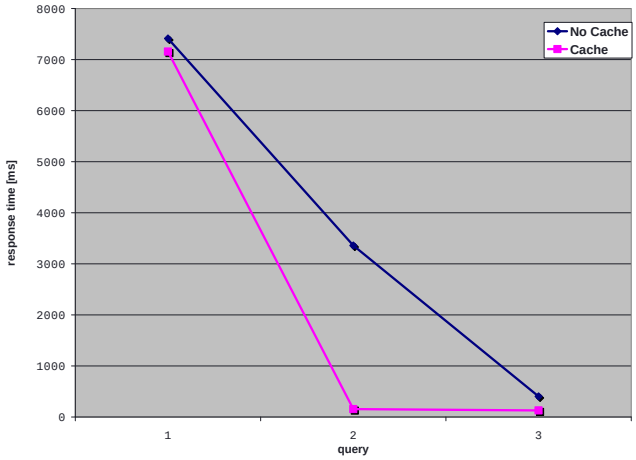


Figure C.34: The chart compares the development of the average query response times shown in figures C.32 and C.33 for a system with cache and without cache.

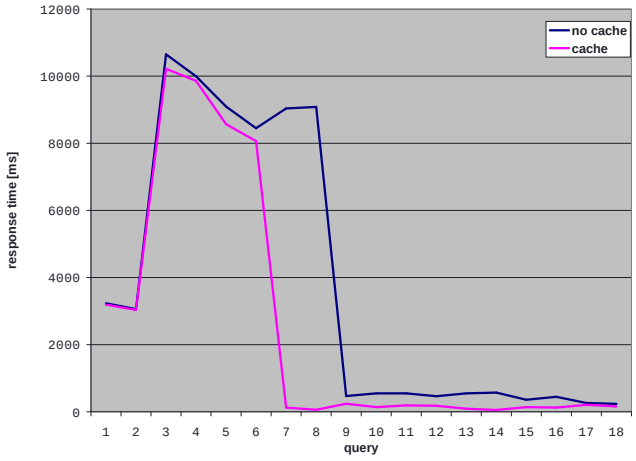


Figure C.35: The chart compares the development of the query response times for a system with and without a cache - from the system's perspective. It shows the development in the order in which the queries where answered by the reasoner. *Note that queries 1-6 are equal to queries 7-12 and 13-18.*

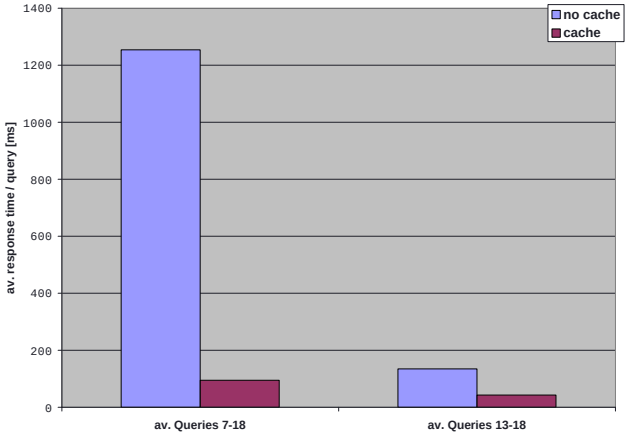


Figure C.36: The column chart compares the average response times of the queries 7-18 and 13-18 (which is the interval where differences occured due to the cache/no cache implementation) as they are shown in fig. C.35. *Note that the difference between the both is that in the comparison 7-18 the accumulated waiting times of queries 1-6 affect the response times for the queries 7-12, while 13-18 shows a cache/no cache comparison free of waiting times.*

## C.5 Charts for 7.4.1 Benchmark 5

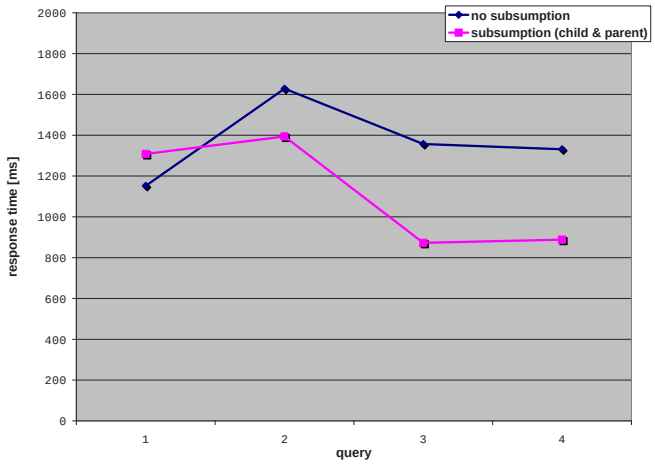


Figure C.37: Benchmark V. The chart compares the development of the response times of the four queries for the system without and with subsumption optimization.

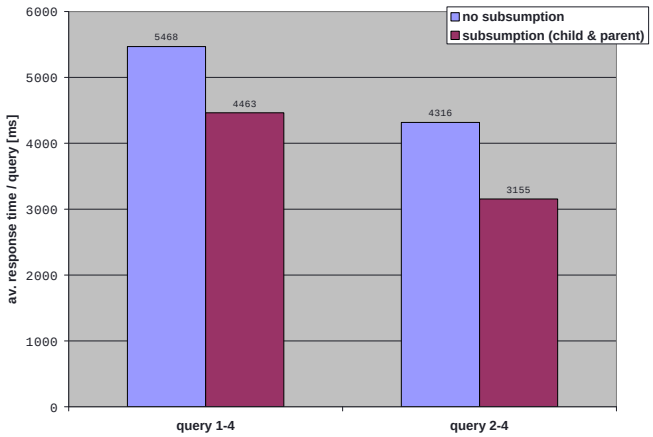


Figure C.38: Benchmark V. The chart compare the average response times for all four queries of a system with and without subsumption optimization. The second pair of columns does the same comparison for the queries 2 to 4.

## C.6 Charts for 7.4.2 Benchmark 6

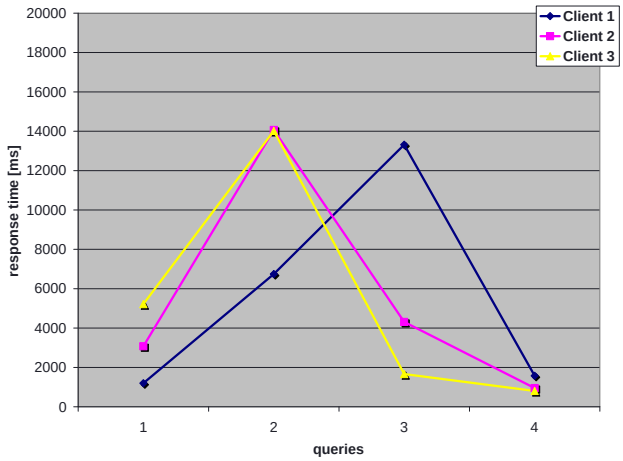


Figure C.39: Benchmark VI. The chart shows the individual developments of the response times for each of the clients in the setting without subsumption optimization.

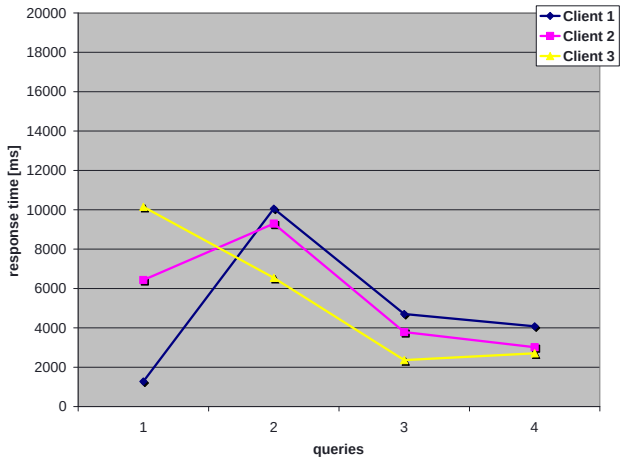


Figure C.40: Benchmark VI. The chart shows the individual developments of the response times for each of the clients in the setting with child subsumption optimization.

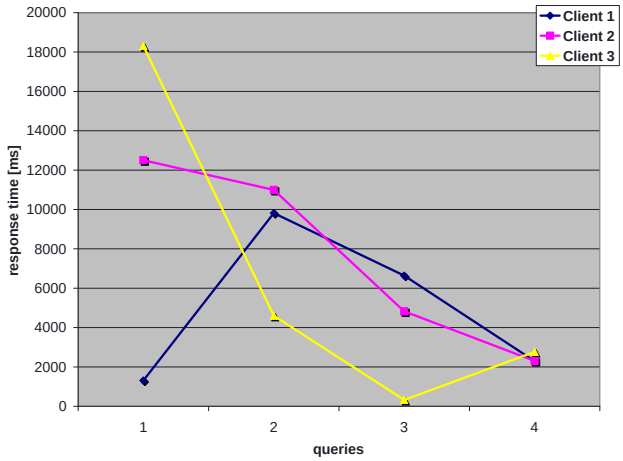


Figure C.41: Benchmark VI. The chart shows the individual developments of the response times for each of the clients in the setting with full child & parent subsumption optimization.

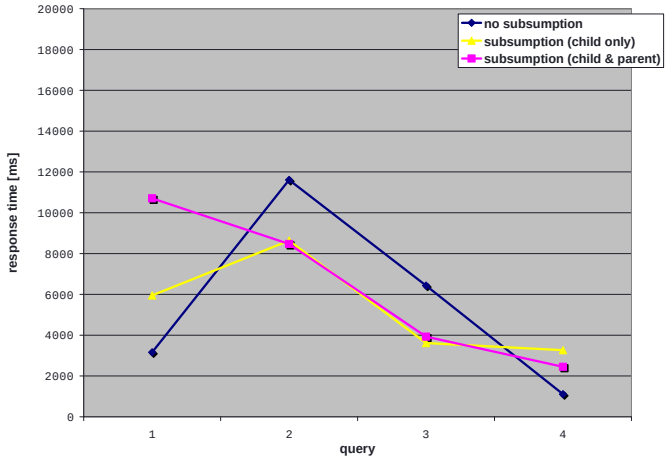


Figure C.42: Benchmark VI. The chart compares the development of the response times of the four queries for the system without subsumption optimization, with child subsumption and with full child & parent subsumption optimization.

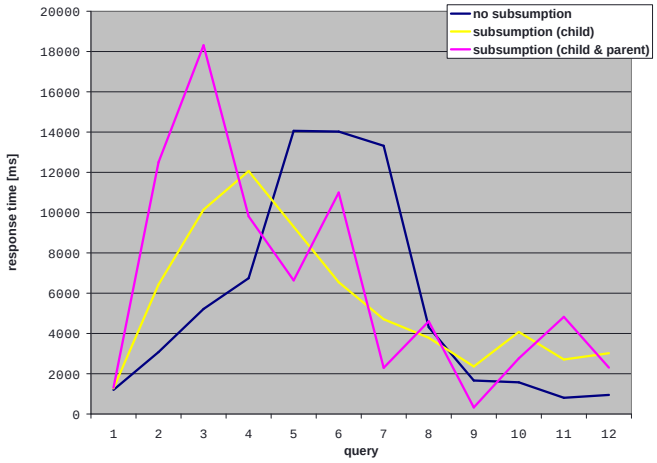


Figure C.43: Benchmark VI. The chart shows the development of the response times from a systems perspective, in the order they were answered. The graph compares the settings without subsumption, with child subsumption and with child & parent subsumption optimization

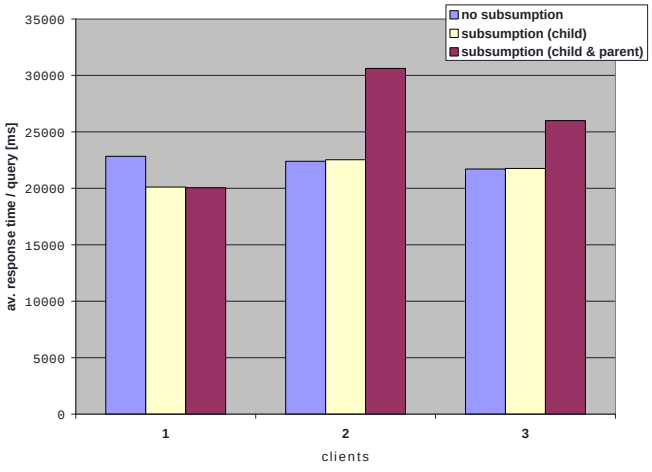


Figure C.44: Benchmark VI. The chart compares the average response times for each client individually in regard to the different system settings.

# Appendix D

# LUBM Queries in OWL QL

### LUBM Query 01

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#GraduateStudent"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:takesCourse rdf:resource="http://www.Department0.University0.edu/GraduateCourse0"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query 02

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#GraduateStudent"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#y">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#University"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#z">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Department"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:memberOf rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#z"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#z">
<uni:subOrganizationOf rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:undergraduateDegreeFrom rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
<var:y/>
<var:z/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query 05

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Person"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:memberOf rdf:resource="http://www.Department0.University0.edu"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query 06

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Student"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query 07

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Student"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#y">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Course"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.Department0.University0.edu/FullProfessor0">
<uni:teacherOf rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:takesCourse rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
<var:y/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query 09

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Student"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#y">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Faculty"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#z">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Course"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:advisor rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:takesCourse rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#z"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#y">
<uni:teacherOf rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#z"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
<var:y/>
<var:z/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query 10

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Student"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:takesCourse rdf:resource="http://www.Department0.University0.edu/GraduateCourse0"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query 11

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#ResearchGroup"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:subOrganizationOf rdf:resource="http://www.University0.edu"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query: Variation to 06 (Q15)

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Student"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#y">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Course"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.Department0.University0.edu/FullProfessor1">
<uni:teacherOf rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:takesCourse rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
<var:y/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query: Variation to 06 (Q16)

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Student"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#y">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Course"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.Department0.University0.edu/FullProfessor2">
<uni:teacherOf rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<uni:takesCourse rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
<var:y/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
0
</answerSizeBound>
</query>
```

## LUBM Query: getAssociateProfessor - Subsumption Query 1

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#AssociateProfessor"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>1</answerSizeBound>
</query>
```

## LUBM Query: getProfessor - Subsumption Query 2

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Professor"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
1
</answerSizeBound>
</query>
```

## LUBM Query: getFaculty - Subsumption Query 3

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Faculty"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
1
</answerSizeBound>
</query>
```

## LUBM Query: getPerson - Subsumption Query 4

```
<?xml version="1.0"?>
<query
xmlns="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:uni="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#">
<queryPattern>
<rdf:RDF>
<rdf:Description
rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
<rdf:type rdf:resource="http://www.lehigh.edu/\%7Ezhp2/2004/0401/univ-bench.owl#Person"/>
</rdf:Description>
  </rdf:RDF>
</queryPattern>
<mustBindVars>
<var:x/>
</mustBindVars>
<answerKBPattern>
<kbRef rdf:resource="file://localhost/~/univ-bench-1.owl"/>
</answerKBPattern>
<answerSizeBound>
1
</answerSizeBound>
</query>
```

# Bibliography


[1] Tony Bourke (2001). *Server Load Balancing*. O'Reilly

[2] Jeremy D. Zawodny, Derek J. Balling (2004). *High Performance MySQL*. O'Reilly

[3] Chandra Kopparapu (2002). *Load balancing servers, firewalls, and caches*. Wiley

[4] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (2003). *The Description Logic Handbook: Theory, Implementation, Applications.*. Cambridge University Press

[5] Volker Haarslev, Ralf Moeller, Michael Wessel (2005). *RacerPro User's Guide*. Racer Systems GmbH & Co. KG, Version 1.9, published on December 8th 2005

[6] Volker Haarslev, Ralf Moeller, Michael Wessel (2005). *RacerPro Reference Manual*. Racer Systems GmbH & Co. KG, Version 1.9, published on December 8th 2005

[7] *Apache Axis Web Services Framework*. Website: http://ws.apache.org/axis/. Last accessed February 27th 2007

[8] *Apache Tomcat Application Server*. Website: http://jakarta.apache.org/tomcat/. Last accessed February 27th 2007

[9] *Apache JMeter User's Manual*. Website: http://jakarta.apache.org/jmeter/usermanual/. Last accessed February 27th 2007

[10] *Jena Semantic Web Framework*. Website: http://sourceforge.net/projects/jena/. Last accessed February 27th 2007

[11] *XMLBeans*. Website: http://xmlbeans.apache.org/. Last accessed February 27th 2007

[12] *Inference Web*. Website: http://iw.stanford.edu/. Last accessed February 27th 2007

[13] Richard Fikes, Patrick Hayes, and Ian Horrocks *OWL-QL A Language for Deductive Query Answering on the Semantic Web*. W3C Recommendation, Website: http://www-ksl.stanford.edu/projects/owl-ql/

[14] Deborah McGuinness and Frank van Harmelen (2004) *OWL Web Ontology Language - Overview*. W3C Recommendation, Website: http://w3.org/TR/owl-features/REC-owl-features-20040210, 2004

[15] Marco Dorigo, Thomas Stuetzle (2004). *Ant Colony Optimization*. The MIT Press

[16] Lawrence Botley (2006) *Artificial intelligence network load balancing using Ant Colony Optimisation*. Website: http://www.codeproject.com/useritems/Ant_Colony_Optimisation.asp

[17] Apache (2006) *Struts 2 Framework*. Website: http://struts.apache.org/2.0.6/

[18] The Semantic Web and Agent Technologies Lab, Lehigh University (2006) *Lehigh University Benchmark*. Website: http://swat.cse.lehigh.edu/projects/lubm/

[19] Anni-Yasmin Turhan, Sean Bechhofer, Alissa Kaplunova, Thorsten Liebig, Marko Luther, Ralf Moeller, Olaf Noppens, Peter Patel-Schneider, Boontawee Suntisrivaraporn, and Timo Weithoener (2006). *DIG 2.0 - Towards a Flexible Interface for Description Logic Reasoners.* In B. Coence Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *OWL: Experiences and Directions 2006*