Czech Technical University in Prague
Faculty of Electrical Engineering

DIPLOMA THESIS

Using Databases for Description Logics

Prague, 2009 **Author: Marek Šmíd**

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne ____________ ____________________

podpis

Declaration

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

In Prage on ____________ ____________________

signature

Acknowledgements

First, I would like to thank my supervisor Ing. Petr Křemen, whose patience and helpfulness allowed me to work on this project very enjoyably. Gaining some of his vast amounts of theoretical and practical knowledge really enriched my overview of the field of knowledge base systems.

Generally, my acknowledgements go to the whole group Knowledge Base and Software Systems – for being so progressive and running such interesting projects. Not only the group's interesting work attracted me to join it, but also the very friendly atmosphere there, for which thanks belong to all the group members.

Last, but not least, my thanks belong to my family, which supported me all the time of my prolonged studies, including support for two academic trips abroad.

Abstract

This thesis compares different relational schemes and implementations of tools for the *DL−Lite* language, which is suitable for formal representation of large data, making it useful for such distributed environment like the Internet, and it shows the usefulness of the language by its application in a real-life problem.

The key property of the *DL−Lite* language is that for storing large amounts of data can be used the well-established relational databases, and it also allows to answer queries reformulated into SQL directly in the databases. The reasons why the language is better than relational databases themselves is that it provides higher expressivity, while still having tractable reasoning.

The important task is to evaluate the query answering performance for data written in this language with respect to different relational schemes. For this purpose we made use of existing *DL−Lite* reasoner prototypes (Owlgres, QuOnto). Two testing datasets were used (UOB, DBpedia) to accomplish this. The most interesting factor differentiating the implementations (according to the performance) is the data representation model in database, which can be characterised by its relational scheme. One of the implementations (Owlgres) was for this reason extended with two additional relational models. The results clearly show advantages and disadvantages of the models and the implementations.

The last objective is to apply the *DL−Lite* language in a real-life application, within European project NetCarity, which attempts to make life of elderly people on their own more secure. The language is used to represent information regarding sensors (their properties and relations between them) and to store the measured data in a structured way. This allows for complex queries and analyses of the measured data.

Abstrakt

Cílem této práce je porovnat různé relační schémata a technické realizace nástrojů pro jazyk *DL−Lite*, který je vhodný pro formální popis objemných dat, který tak nachází uplatnění zejména na rozprosřených prostředích jako je Internet, a nakonec ukázat vhodnost jeho použití na reálném problému.

Klíčovou vlastností jazyka *DL−Lite* je, že lze využívat pro ukládání velkého množství dat osvědčené relační databáze, a rovněž přímo v ní provádět zodpovídání dotazů přeformulovaných do jazyka SQL. Důvod, proč je zvolený jazyk lepší než samotné relační databáze je, že poskytuje větší expresivitu, při zachování výpočetně zvládnutelného dotazování.

Důležitým bodem je vyhodnotit výkon zodpovídání dotazů nad daty zapsanými v tomto jazyce vzhledem k různým relačním schématům s využitím existujících prototypů (Owlgres, QuOnto). Pro tento účel byla použita testovací data (dataset UOB, DBpedia). Nejzajímavějším prvkem odlišujícím implementace (vzhledem k výkonu) je použitý model reprezentace dat v databázi, který lze vyjádřit relačním schématem. Z tohoto důvodu byla jedna z implementací (Owlgres) rozšířena o dva další relační modely. Naměřené výsledky jasně ukazují výhody a slabiny jednotlivých modelů a implementací.

Poslední úlohou je nasazení jazyka *DL−Lite* do reálné aplikace, v rámci evropského projektu NetCarity, který se zabývá zabezpečením samostatného života starších spoluobčanů. Využití studovaného jazyka je pro popis informací o senzorech zde použitých (jejich vlastností a vztahů mezi nimi) a strukturované ukládání jejich naměřených dat. To umožňuje komplexní dotazy a analýzy nad naměřenými údaji.

The assignment in english

The assignment in czech

Contents

1. Introduction

The need for an intelligent structured way of representing and querying knowledge on the Internet is apparent – there are large volumes of information in natural language and semi-structured form, which are difficult to access. Classical full-text search used on the Internet every day is somewhat limited, since it searches only by a set of keywords, not using their semantics or semantic relations between them.

For this reason, a new group of languages appeared, called OWL (Web Ontology Language, see Section 2.1.2), which is one of the realizations of the Semantic Web[1]. OWL has its grounds in from Description Logics, the languages developed for formal knowledge representation. All these languages allow for knowledge representation understood by computers.

This thesis focuses on the languages family *DL−Lite*, which is a modern descendant of OWL, and is planned to became a part of its upcoming major revision, the OWL 2[2]. The key property of the *DL−Lite* language is, as opposed to most of other description logics, that for storing large amounts of data can be used the well-established relational databases, while preserving correctness and completeness of the reasoning algorithms, and it also allows to answer queries reformulated into SQL directly in the databases. This inevitably puts some restrictions on its expressivity (complexity of constructs of the language), which is still sufficient for many useful applications in different areas (according to (Calvanese et al., 2005) e.g. conceptual data models and object-oriented formalisms) and is strictly higher than expressivity of relational database technology. There is no detailed comparison of *DL−Lite* implementations so far – it is a new technology not widely spread yet.

This thesis starts with an overview of languages known as description logics, focusing on the OWL language family. The details are in Section 2.1. The theoretical background consequently aims to described in details the *DL−Lite* language, which is a fragment of OWL 2, and to compare its capabilities with other members of the OWL family. The

[1] `http://www.w3.org/2001/sw/`

[2] `http://www.w3.org/TR/owl2-profiles/`

DL−Lite is described in detail in Section 2.1.3.

An important insight into the reasoning efficiency can be obtained by comparing inference algorithms based on different relational schemes. The comparison of different relational schemes is one of the key tasks of this thesis, because the scheme, used for representing an ontology in a database, has big impact on the query answering performance.

Two testing datasets were used to accomplish this: UOB[3] (The University Ontology Benchmark) as an artificially generated benchmarking dataset, and DBpedia[4], which consists of real data extracted from Wikipedia (an encyclopedia on the Internet), reformulated into the description logics format (for details see Section 4.1). The most interesting factor differentiating the implementations (according to performance) is the database data representation model, which can be characterized by its relational scheme. For this reason one of the implementations (Owlgres) was extended with two more relational models. The results clearly show advantages and disadvantages of the models and the implementations. For details see Section 4.2, Section 4.3, and Section 4.4.

The last objective is to apply the *DL−Lite* language in a real-life application, the NetCarity project, which attempts to make life of elderly people on their own more secure. Its tasks are for example calling a help when such a person falls down, or set off an alarm when a fire is detected. The application of the language is to represent information regarding sensors used (their properties and relations between them) and to store the measured data in a structured way. This allows for complex queries and analyses of the measured data. The sensors are for example a microphone, an accelerometer, or a camera installed in a house of the monitored person. For details see Section 5.1 and Section 5.2.

[3]`http://www.alphaworks.ibm.com/tech/semanticstk/download`

[4]`http://wiki.dbpedia.org/Downloads32`

2. Background

2.1. Description Logic

Description logics (DL) are languages that provide currently widely accepted means for formalization of semantic web ontologies. The term “ontology” can be defined as an “explicit specification of conceptualization of a knowledge” (Gruber, 1993). It describes a structure of some information domain, and it defines a vocabulary used to express a knowledge base.

An ontology can be expressed in an informal way, e.g. in natural language, or in a formal way, e.g. frames and description logics. The problem of informal languages is they do not allow for automatic processing by computers, while the formal ones do. A rough overview of different languages for expressing ontologies can be found in Fig. 2.1 (Obitko, 2007), where the horizontal axis represents the level of formalization. On the left hand side of the vertical line are the informal languages, as catalogues and glossaries, usually expressed in natural language, which is not formally defined, and hence it is very hard for a computer program to infer knowledge from such documents.

The formal languages on the right hand side restricts documents to certain constructs, resulting in different expressivity, but allowing for automatic knowledge processing. Such languages typically use constructs like is-a relation, taxonomy of classes and relations, class restrictions, etc. The figure mentions e.g. “formal is-a” which is a very simple language, assigning individuals to classes, but as the axis goes right, the languages are becoming more expressive. Below, we will briefly mention “frames”. Description logics, which will be discussed in more details, differ in expressive power, and spread among the axis more on the right. The right end of the axis represents full featured logics, such as first or higher order logics or modal logics, which have very high expressive power, but generally lack decidability. This theoretical problem is a problem for implementation as well, because a query answering system can search the answer for certain queries endlessly.

For more details about description logics, their complexity, applications, etc., see (Baader et al., 2003).

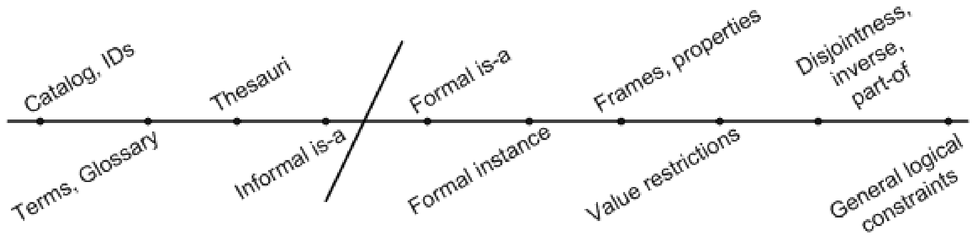

Figure 2.1.: Different ways of describing ontology

2.1.1. Comparison to Other Formalisms

Frames are based on two modeling primitives: frames and slots. A frame represents an entity in the domain. It can be a “class frame” when it represents a class, and an “individual frame” for an individual. This way a class is represented the same way as an individual, making it possible to use the same constructs both for classes and individuals. This is not possible in first order languages, like description logics, where the set of classes and the set of individuals are inherently disjoint.

A frame has assigned a set of slots, each slot represents an attribute with an associated value. Each slot can have a set of facets associated with it, which restrict its values. A slot value can be any frame as well as basic data types. Frames allow for three different types of collections: sets, bags (like sets but with multiple identical elements allowed), and lists. Compared to description logics, frames do not have such precise semantics.

Description logics are generally strict subsets of FOL (first order logic), which allows for decidable reasoning. This is because FOL is undecidable, and the obvious requirement we want from our language on query answering is to finish in (reasonably short) finite time.

Another interesting comparison is of description logics versus database systems (DBMS). DBMS use the closed world assumption (CWA) and support only finite domains, while description logics use the open world assumption (OWA) and the domain can be infinite. Relational databases are also weaker in expressivity.

2.1.2. OWL

The most widely used language based on description logics at the moment is probably OWL (Web Ontology Language), a W3C recommendation[1]. OWL in version 1.0 consti-

[1]`http://www.w3.org/2004/OWL/`

tutes three languages with different expressivity: OWL Lite, OWL DL, and OWL Full (ordered by increasing expressivity). OWL Full provides maximum syntactic freedom of the three, but does not give any computational guarantees. OWL DL provides maximum expressivity while being decidable.

Let us see what is the expressivity of OWL DL, as it is a good example of fairly expressive language. This example will help us understand how the language of our interest, *DL−Lite*, is limited compared to widely used logic like OWL DL. Basic entities of every description logic are concepts, representing sets of individuals, and roles, representing sets of pairs of individuals. In terms of FOL the concepts are unary predicates and the roles are binary predicates.

In OWL DL, a concept can be defined using constructs as in the following list, written in description logics syntax. For details about used symbols and keywords, see (Baader et al., 2003). OWL DL allows constructs from a set indicated by letters $\mathcal{ALC}_{\text{trans}}\mathcal{HOIN}(\mathcal{D})$, as defined in description logics. The attached examples are from the domain of living creatures.

- A (an atomic concept)
  e.g. `Person` representing all persons

- $\neg C$
  e.g. ¬`Person`, representing all individuals except persons

- $C \sqcap D$
  e.g. `Person` ⊓ `Male`, representing all men (male persons)

- $C \sqcup D$
  e.g. `Man` ⊔ `Woman`, representing all persons (men and women)

- $\forall R \cdot C$
  e.g. ∀`hasChild` · `Woman`, representing all parents that have only daughters (or no children at all)

- $\exists R \cdot C$
  e.g. ∃`hasChild` · `Person`, representing all human parents (have at least one person as a child)

- $(\geq nR)$ (number restriction)
  e.g. (≥ 3 `hasChild`), representing all parents with at least 3 children

- $(\leq nR)$ (number restriction)
  e.g. $(\leq 3$ `hasChild`$)$, representing all parents with the maximum of 3 children
- $\{a_1, \ldots, a_n\}$ (nominals)
  e.g. {`class`,`interface`,`enumaration`}, representing the list of basic building blocks in Java
- R^- (role inversion)
  e.g. `hasChild`$^-$, representing the role parent (`hasParent`) (inversion of the role child)

In the list C, D are concepts, and R is a role. The roles in number restrictions can be only simple roles (do not have transitive super-roles). OWL DL also allows datatype roles to be used, where only the first argument comes from the domain, while the second argument is of another data type.

Then a concept can be defined by one of the following axioms:

- $C \sqsubseteq D$ (inclusion)
  e.g. `Human` $\sqsubseteq$ `Mammal`, saying that every human is a mammal
- $C \equiv D$ (equivalence)
  e.g. `Human` $\equiv$ `Person`, saying that all humans are persons, and vice versa

Roles in OWL DL can be defined similarly – either by equivalence to, or by inclusion of another role, or the inverse of a role. Also they can be defined as transitive.

For OWL DL semantics see `http://www.w3.org/TR/owl-semantics/`, as it is not the main focus of this thesis.

2.1.3. *DL−Lite*

DL−Lite is another member of description logics family, which is less expressive than all OWL languages, so that it can be stored in a relational database system, where also queries can be evaluated using standard SQL language. This is a very modern approach, because most of the other languages, though providing higher expressivity, are hardly usable for large datasets. This is a typical case on the Internet, where an ontology is about to store vast amounts of facts. Currently *DL−Lite* is not standardized by W3C, but one of its variants, $DL{-}Lite_R$, is a specified fragment of the up-coming W3C standard of OWL 2.

Because of the application for the Internet, we need feasible worst-case complexities. In order to have an efficient reasoning, because of the size of ABox (number of assertions, which is very big on the Internet), the time complexity of queries with respect to ABox should be as efficient as possible, preferably polynomial in time (polytime). Also the space complexity should be logarithmic (logspace). These restrictions make it possible to reformulate a query into the SQL language. Other languages have worse query answering complexity, which makes them practically unusable for domains with many instances the Internet offers.

Another advantage of this approach becomes obvious when we compare memory management during query answering. $DL{-}Lite$ reformulates a query from e.g. SPARQL into an SQL query, which is evaluated by an underlying DBMS (database management system), keeping the ABox in a persistent storage (e.g. harddrive), only operating on indices of database tables (typically B-trees); just the individuals featuring in the final answer need to be read from the DB. Another key advantage is that SQL query answering is well-settled problem, with many optimizations implemented in most modern DBMS's, making it very robust and efficient. Whereas most of the reasoners for OWL load the whole ABox into memory, where they perform the whole reasoning process. The efficiency of the process is very dependent on the optimizations the reasoner's authors used, and hence it may take some time for the reasoner to "evolve" into a state suitable for practical applications. A bigger issue of this approach is the memory requirements, since it is usually very demanding to store complete ABox of possibly many gigabytes into a computer RAM memory.

To conclude the introduction, if we succeed to fit an ontology into a DBMS, we gain a few advantages. The system would efficiently permanently store large ABoxes in a DB, which is a well-established way of storing strictly organized data. Query answering keeps an ABox in the DB – no need to transfer it into the memory, which would be infeasible because of the memory size restriction anyway. Query answering is inherently of polytime complexity, which is another "must" for large datasets. Let us see, what the language $DL{-}Lite$ satisfying all these conditions can express, as presented in (Calvanese et al., 2007; Calvanese et al., 2006; Calvanese et al., 2005).

The basic version of $DL{-}Lite$ is $DL{-}Lite_{core}$, which satisfies all the aforementioned requirements, has reasonable expressivity, but still can be expanded, while preserving tractability. $DL{-}Lite_{core}$ constructs for defining concepts and roles are:

- $B ::= A | \exists R$

- $C ::= B|\neg B$
- $R ::= P|P^-$
- $E ::= R|\neg R,$

where A denotes an atomic concept, B a basic concept, and C a general concept. Symbol P denotes an atomic role, R a basic role, and E a general role.

The semantics is defined by an interpretation $\mathcal{I}$:

$$\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}).$$

An interpretation assigns sets of individuals to concepts, and sets of pairs of individuals to roles, which is defined by the interpretation function $\cdot^{\mathcal{I}}$. The individuals are members of the domain of the interpretation $\Delta^{\mathcal{I}}$.

Semantics of the used constructs are in Table 2.1.

Table 2.1.: Constructs used in *DL−Lite* and their semantics

Syntax	Semantics	Comment
A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	atomic concept
P	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	atomic role
P^-	$(P^-)^{\mathcal{I}} = \{(b,a)\|(a,b) \in P^{\mathcal{I}}\}$	inverse of an atomic role
$\exists R$	$(\exists R)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}}\|\exists b : (a,b) \in R^{\mathcal{I}}\}$	existential quantification
$\neg B$	$(\neg B)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus B^{\mathcal{I}}$	negation of a basic concept
$\neg R$	$(\neg R)^{\mathcal{I}} = \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \setminus R^{\mathcal{I}}$	negation of a basic role

Then a TBox can be defined by inclusion axioms of the form:

- $B \sqsubseteq C$, with the semantics of $B^{\mathcal{I}} \subseteq C^{\mathcal{I}}$

There are two other versions, $DL{-}Lite_F$, $DL{-}Lite_R$, which are slightly different, but both are at the verge of keeping the complexity requirements satisfied. Their simple union breaks the requirements. But restricting their union gives version $DL{-}Lite_A$, which still satisfies the complexity requirements.

$DL{-}Lite_R$ is extended by the concept construct:

- $R \sqsubseteq E$, with the semantics of $R^{\mathcal{I}} \subseteq E^{\mathcal{I}}$

$DL{-}Lite_F$ is extended by the role restriction:

- (funct R), with the semantics of $\{(a,b),(a,c)\} \subseteq R^{\mathcal{I}} \Longrightarrow b = c$

All the versions can be expanded by data attributes of concepts and roles, which does not affect the complexity.

What can $DL{-}Lite$ express:

- is-a relationship – concept A_1 is subsumed by concept A_2, using $A_1 \sqsubseteq A_2$
- disjoint concepts – $A_1 \sqsubseteq \neg A_2$
- typed roles – $\exists P \sqsubseteq A_1$, $\exists P^- \sqsubseteq A_2$
- mandatory participation in a role – all instances of a concept participates in a role – $A \sqsubseteq \exists P$, $A \sqsubseteq \exists P^-$
- prohibited participation in a role – $A \sqsubseteq \neg\exists P$, $A \sqsubseteq \exists P^-$
- functional restriction of roles – (funct P), (funct P^-) (only in $DL{-}Lite_F$)

Fig. 2.2 (Calvanese et al., 2007) shows the relationship between $DL{-}Lite_{core}$, $DL{-}Lite_F$, $DL{-}Lite_R$, $DL{-}Lite_A$, and some other versions of $DL{-}Lite$, with the arrows meaning "is extended by".

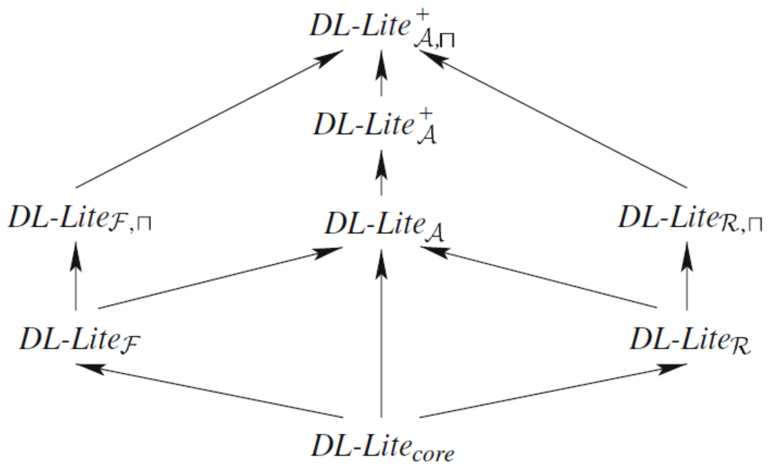
DL-Lite$^+_{\mathcal{A},\sqcap}$
DL-Lite$^+_{\mathcal{A}}$
DL-Lite$_{\mathcal{F},\sqcap}$
DL-Lite$_{\mathcal{A}}$
DL-Lite$_{\mathcal{R},\sqcap}$
DL-Lite$_{\mathcal{F}}$
DL-Lite$_{\mathcal{R}}$
DL-Lite$_{core}$

Figure 2.2.: $DL{-}Lite$ languages family

2.2. *DL−Lite* implementation

2.2.1. Owlgres

Owlgres[2] is an open source Java implementation of *DL−Lite* reasoner developed by Clark & Parsia[3], a small research and development firm specializing in Semantic Web and advanced systems based in Washington, DC.

Owlgres uses a DBMS for persisting data, particularly it is tailored to work with PostgreSQL[4] database (see Section 2.2.1.1). PostgreSQL is a widely used open source object-relational database system available for all major operating systems and platforms.

Owlgres uses a few libraries. It is bundled with Joseki, a SPARQL[5] HTTP server (allowing querying the knowledgebase over HTTP), which we did not test. The key libraries for interfacing with the inputs are OWL API[6] and Jena[7]. OWL API is a Java interface and implementation for OWL, which serves in Owlgres for parsing input data (TBox and ABox) in the RDF/XML format. Jena, specifically its SPARQL processing engine ARQ[8], is used only for parsing SPARQL queries.

There are three typical use-cases of the Owlgres operation:

- Loading a TBox – typically performed only once at the beginning of a knowledgebase life, since it empties the used database, and creates all necessary tables there. Currently Owlgres does not support incremental TBox loading. It simply parses a TBox RDF/XML file using OWL API, puts it into corresponding DB tables, and prepares empty tables for ABox assertions.

- Loading an ABox – can be run incrementally. Owlgres parses an ABox from a RDF/XML file using OWL API, loads the entire TBox from DB, and using the TBox it adds the ABox assertions into DB tables for assertions. Hence it cannot add an ABox assertion for a concept or a role, which was not yet defined in the TBox. This enforces a complete TBox to be loaded in advance.

- Querying an ABox – a query formulated in SPARQL is parsed using Jena; using

[2]`http://pellet.owldl.com/owlgres`

[3]`http://clarkparsia.com/`

[4]`http://www.postgresql.org/`

[5]SPARQL is a query language for RDF, which contains capabilities for graph patterns along with their conjunctions and disjunctions; it also supports value testing and constraining queries

[6]`http://owlapi.sourceforge.net/`

[7]`http://jena.sourceforge.net/`

[8]`http://jena.sourceforge.net/ARQ/`

the entire TBox loaded from the DB, the query is reformulated into an SQL query, which is posed to the DB. The SQL query result is bound the the queried variables, and these variable bindings are presented as the final result.

For more details on Owlgres, its optimizations, details of the used language, and some of the author's tests, see (Stocker and Smith, 2008). Refering to $DL-Lite$ language description in Section 2.1.3, Owlgres uses the version $DL-Lite_R$.

The database scheme Owlgres uses is described in detail in Section 3.1. To summarize, it uses something we call a "single table" model (STM), which means there is a single table for each fact type, e.g. for concept assertions, meaning that all concept assertions for all concepts go into a single table. Thus the table obviously has to have a column for specifying a concept, and a column for specifying an individual. The same is true about object role assertions (a role column and two individual columns), data role assertion (a role column, an individual column, and a data value column), and similarly for annotations. Because repeating individuals' URI identificators in all the assertion tables would unnecessarily occupy a lot of persistent space, there is a table serving as a list of individuals, translating their URIs into generated numerical primary key, which is used instead of URIs in all the assertion tables.

2.2.1.1. PostgreSQL

PostgreSQL is an open source DBMS (database management system), originally developed at the University of California at Berkeley. It offers features like foreign keys, triggers, views, transactional integrity, and multiversion concurrency control. It is a well established DBMS.

The current stable version at the time of this thesis is 8.3.7, but a few first tests were performed on version 8.3.0. This version difference might seem neglectable, but the truth is different. There was a certain query, discussed in Section 4.2, which performed on a testing dataset, took a long time (about 300s). When the PostgreSQL was upgraded to version 8.3.7, the same query under the same conditions took much shorter time (about 5s). All the remaining tests were performed with version 8.3.7. For details, see Section 4.2.

2.2.2. QuOnto

QuOnto[9] is a Java implementation of a *DL−Lite* reasoner made by the authors of the *DL−Lite* specification as stated in the articles (Calvanese et al., 2007; Calvanese et al., 2006; Calvanese et al., 2005) at Sapienza, a university in Rome. It is based on $DL{-}Lite_F$.

QuOnto is wrapped in a package called QToolKit, which is downloadable (after registration) from the QuOnto website. Unfortunately, QuOnto is not open source, and even the QuOnto libraries in QToolKit are obfuscated[10]. This means there is no API and no way how to understand the application details.

It uses H2 database[11] (see Section 2.2.2.1) in the embedded mode, using memory storage only. QToolKit has the H2 library bundled. The memory only mode of H2 means that it cannot be revealed how the data are stored in the DB, since the memory cannot be accessed by another instance of H2 (there is a special mode for dissalowing this, which is unfortunately used in case of QuOnto).

QuOnto uses some other libraries as well, e.g. Jena's ARQ probably for parsing SPARQL queries. But it is hard to guess how exactly and what libraries are used, since the QuOnto's source codes cannot be examined, so only the list of JAR files bundled in QToolKit is obvious.

The only access to the application is its GUI, where TBox and ABox can be loaded into its text panes. Then there is the `Run On QuOnto` button, which does all the loading and processing process. It lets us only to guess what exactly happens inside, it is uncertain whether the loaded TBox is represented only in memory, or is loaded to the H2 database. The ABox does get loaded into the DB (according to some tests performed, see Section 3.2). For reasoning services, it provides consistency check function, intensional reasoning (e.g. checking subsumption, disjointness, etc.), and query answering. We used only the query answering for our tests. A query can be entered in the following languages: Datalog, SPARQL, and SparSQL[12]. We used the last two – SPARQL to test the same queries as with Owlgres, and SparSQL because it offered some interesting observations –

[9] `http://www.dis.uniroma1.it/~quonto/`

[10] Obfuscation of a Java application is a process, where the compiled class files are modified so that their function is the same, while they are very hard to decompile. It for example means that all class, variable, and method names are changed to meaningless short strings, so a reverse engineer would not know what class server what purpose, etc.

[11] `http://www.h2database.com/html/main.html`

[12] SparSQL is a query language developed for *DL−Lite* at Sapienza, a university in Rome. Its syntax is inspired both by SQL and SPARQL. Basic usage looks like a SPARQL statement wrapped in an SQL query.

it allowed us to find out what the relational scheme of the DB is (see Section 3.2), which we would not know otherwise, since it is not documented and the authors do not want to release this information.

A big issue of this implementation is that it does not use OWLAPI, and the authors did not implement a parser fully compatible with RDF/XML. The only input it can handle is OWL functional-style syntax[13], without capability of handling URIs. All the identifiers used in TBox and ABox have to consist of only alphanumeric characters.

The authors also released ROWLKit – another, slightly different implementation of *DL−Lite* reasoner, based on QuOnto. They did so in quite late phase of this thesis assignment, so there was not much time to study it and test it. The main difference is that ROWLKit uses OWLAPI for parsing its input, which makes any testing and usage much easier, allowing for interconnecting with other data and systems. Still it is not an open source, and again the class files are obfuscated. The OWLAPI employment and some other modifications make any testing more feasible, but there was not much attention spent on testing, since there is a big problem: the relational schema is unknown. This is because ROWLKit misses the SparSQL querying language support (has only SPARQL), which allowed for revealing the relational scheme. Still some tests were performed, see Section 4.3.

2.2.2.1. H2

H2 is an open source Java database implementation. It can operate both in embedded and server modes. It can store a database only in memory. It provides transaction isolation and database encryption.

It also provides a console application, which allows for a control of the DB using a browser interface.

A special mode allows the DB to run in memory only, which should provide faster operation, at the price of not having the data persisted on a hard drive. There are two modes of in-memory DB:

named DB – accessed by JDBC string `jdbc:h2:mem:db1`, in which case the same DB can be accessed many times from the same JVM using its name `db1`; another option is to connect to the DB remotely using JDBC string `jdbc:h2:tcp://localhost/mem:db1`

[13]`http://www.w3.org/TR/2008/WD-owl11-syntax-20080108/`

private DB – accessed by JDBC string `jdbc:h2:mem`, in which case the DB opened this way is always unique, and cannot be accessed by someone else (is private)

Unfortunately, QuOnto uses H2 in the private mode.

3. Implementation

3.1. Owlgres

Owlgres is one of the systems tested for performance on $DL-Lite$ query answering. For its introduction, see Section 2.2.1. The version used for testing was 0.1, the newest available at the time the tests were performed. Some parts have been extended or modified in order to support different persistence models, to fix some bugs and issues, and to measure its performance.

The key modification made within the scope of this thesis is the extension of Owlgres model, which is used for persisting an ABox into a database. The TBox representation model in DB was left unchanged, since it is not an important factor in query answering performance, because the complete TBox is loaded and compiled before a query is executed.

But ABox persistence model is very important for a query, because every query is reformulated into an SQL query posed to the DB tables persisting the ABox. And obviously the time and space demands can vary for different SQL queries formulated for different relational schemes.

3.1.1. Original Owlgres DB model

The original model (represented by relational scheme) of Owlgres, as designed and implemented by its authors, is in Fig. 3.1. You can see the TBox is persisted in four tables. The central one `tbox_name` is for storing all TBox entities – concepts, object roles, data roles, and annotation roles.

Follows a brief description of its columns. By the term TBox entity, or just entity, we mean a concept, an object role, a data role, or an annotation role.

id – generated numerical primary key, used in other tables for identification of the entity

type – what type is the entity of, its value can be one of the following list:

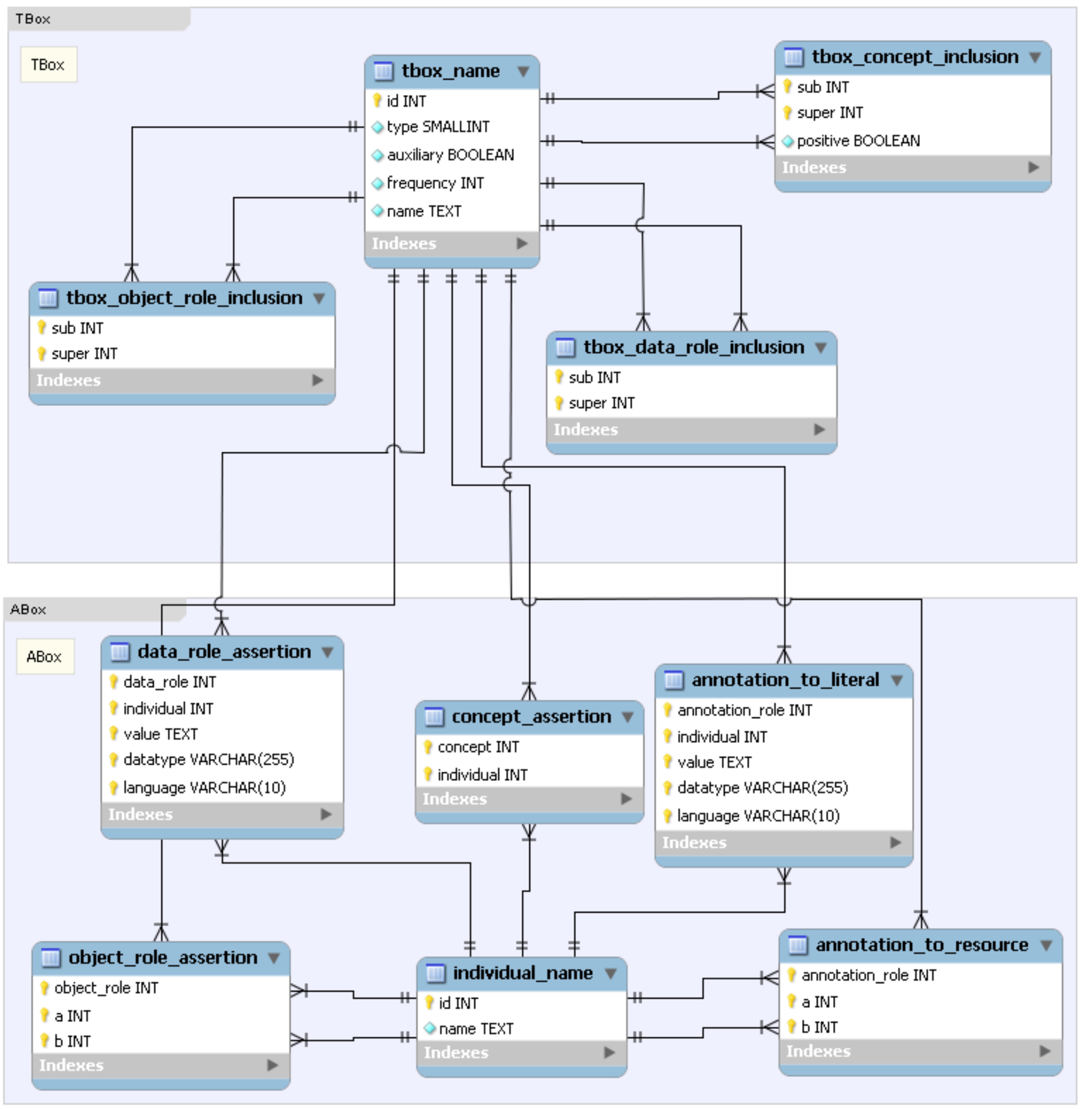

Figure 3.1.: Owlgres original E-R diagram

1 – a concept

2 – an object role

3 – a data role

4 – an annotation role

auxiliary – a boolean, whether the entity (only object role) is auxiliary, Auxiliary roles are used for qualified existential restrictions, which are not expressible in $DL{-}Lite$ directly; they need to be rewritten to fit into $DL{-}Lite$ as $\exists R \cdot C \longrightarrow \exists P$, where $P \sqsubseteq R$ and $\exists P^{-} \sqsubseteq C$, where P is the auxiliary role

frequency – how many times is the entity used in ABox assertions

name – the URI of the entity

The other tables have quite obvious meanings of their columns. The tables are:

tbox_concept_inclusion – for object role inclusions, $C \sqsubseteq D$

tbox_object_role_inclusion – for object role inclusions, $R \sqsubseteq P$

tbox_data_role_inclusion – for data role inclusions, $R \sqsubseteq P$

These tables use generated numerical primary key in `tbox_name` as identification of entities. Note that all these tables use all columns as one joined primary key, so any row duplicities are avoided. Also the tables have indices on all columns separately, to allow fast searching by single column, which is heavily used in SQL queries for query answering.

The only column with not that obvious meaning is the `positive` column in the `tbox_concept_inclusion`. The column contains boolean values specifying, if the inclusions are positive or negative – a negative inclusion contains negation on the right side. I.e.:

- $C \sqsubseteq B$ – a positive inclusion
- $C \sqsubseteq \neg B$ – a negative inclusion

The tables `tbox_concept_inclusion` and `tbox_object_role_inclusion` can have negative id's in their columns. A negative id stands for an entity with a positive id (equal to negative of the negative id), but taken inversely. For `tbox_concept_inclusion` it is because the table is used for existential restrictions also – e.g. of the form $\exists R \sqsubseteq C$, in which case the `sub` column contains a role id instead of an atomic concept id. Then the negative id value means that the role R is inverted, as in $\exists R^- \sqsubseteq C$. This is used quite often, because role domains and ranges are rewritten like this:

- domain$(R) = C \longrightarrow \exists R \sqsubseteq C$
- range$(R) = C \longrightarrow \exists R^- \sqsubseteq C$

In `tbox_object_role_inclusion` a negative id in any of the two columns means the corresponding role is inverted, as for storing axioms like $R \sqsubseteq P^-$ and $R^- \sqsubseteq P$.

The ABox in the original persistence model is stored in fixed number of tables, there is six of them, as opposed to other relational schemes, added to Owlgres for purposes of this thesis. The basic layout is similar to the TBox's, with an individual listing table in the center – table `individual_name`. It simply stores all individual names in the ABox, with generated numerical primary keys used for identification in the other ABox tables. The rest of the tables is for storing assertions:

concept_assertion – for concept assertions (an individual is an instance of a concept), only atomic concepts are allowed, $a \in A^{\mathcal{I}}$, or shortly $A(a)$; a general concept assertion $C(a)$ can be rewritten like $C(a) \longrightarrow A(a)$ where $A \sqsubseteq C$, where A is a fresh concept

object_role_assertion – for object role assertions (a pair of individuals fills a role), only atomic roles are allowed, $(a, b) \in P^{\mathcal{I}}$, or shortly $P(a, b)$; a general role assertion $E(a, b)$ can be rewritten like $E(a, b) \longrightarrow P(a, b)$ where $P \sqsubseteq E$, where P is a fresh role

data_role_assertion – for data role assertions (a pair of an individual and a data value fills a role); it is similar to `object_role_assertion`, except there is a textual data value column `value` instead of role's right hand side individual id column; additional columns are `datatype` and `language` for defining the datatype (e.g. `http://www.w3.org/2001/XMLSchema#dateTime`) and the language of a data value

annotation_to_resource – for annotating an individual with another individual; it has a similar function as `object_role_assertion`, except there is no reasoning (subsumptions etc.) performed on annotations

annotation_to_literal – for annotating an individual with a data value; it has a similar function as `data_role_assertion`, again except there is no reasoning performed on annotations

Unless otherwise noted, these assertion tables use generated numerical primary key in `individual_name` table as an individual identification, and generated numerical primary key in `tbox_name` table as an entity (concept, role, annotation) identification.

3.1.2. Modified DB models

In order to test different relational schemas, Owlgres was extended as part of this thesis with another two DB models. Having in mind conjunctive queries, it may be resonable to assume that the query answering would be more efficient with an ABox representation of one table per TBox entity, which makes filtering of ABox individuals easier.

The basic difference is that the schemes use multiple tables for ABox assertions, the TBox tables remain the same. Multiple ABox tables means there is a single table for every TBox entity (i.e. for every concept, every object/data role, and every annotation). This drops the need to have columns in the ABox tables for identifying which TBox entity is the assertion of; instead, the entity is identified in the assertion table name, e.g. ABox assertions table `concept_52` and `object_role_43`, where 52 is the id of a concept, resp. 43 is the id of an object role. Here only the assertions of the concept 52 are stored, resp. the assertions of the object role 43 are stored.

The reason why these schemes are tested and what makes them interesting is that in ontologies suitable for *DL−Lite* there is much higher number of ABox assertions than of TBox axioms. Therefore it may be practical to classify individuals according to their concept belonging (resp. role belonging) to separate tables, since typical queries on the ABox ask about individuals of certain concepts; it is infrequent to ask about all individuals, regardless of concept classification or role membership. Thus, such a relational model corresponds well to the conjunctive query pattern $\bigwedge_k C_k(x_i) \wedge \bigwedge_k R_k(x_i, x_j)$. This means these schemes will result in reasonable number of tables, containing large amounts of individuals.

There are two variants of this multiple table approach. The first is in Fig. 3.2, which follows the original Owlgres model in that it contains the `individual_name` table. Thus the individual id's are used in the assertion tables.

The other variant is in Fig. 3.3. It uses an idea implemented in QuOnto (see Section 3.2), where no such table as `individual_name` exists, thus there is no translation from individuals to numerical keys. The assertion tables use directly the individual names.

Besides these differences in the two additional schemes the rest of the table columns have a similar meaning.

Note that the TBox and the ABox schemes in Fig. 3.2 and Fig. 3.3 are disconnected, the ABox tables do not refer to the tables in TBox using the standard DB approach, i.e. foreign keys. It is because the reference, represented by entity ID from the `tbox_name` primary key, is used in the ABox table names. This completely erases the need to have the TBox in the DB. Having a TBox in a DB is just a convenient and efficient way of storing it. It could be stored by some other means, e.g. in a standard RDF/XML file, because the TBox is typically very small compared to the ABox. Thus the way a TBox is stored does not affect the query performance over the corresponding ABox in almost

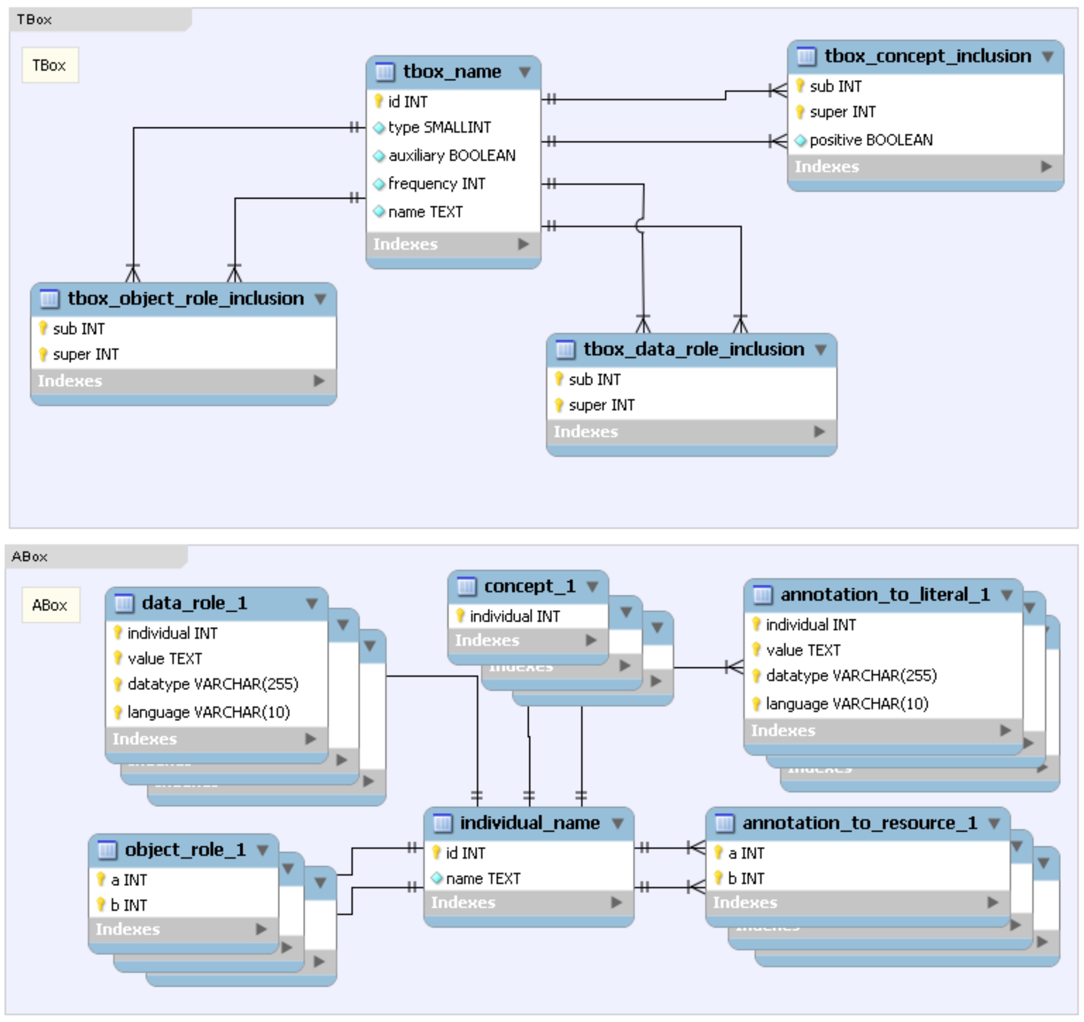

Figure 3.2.: Owlgres modified E-R diagram - multiple tables

any way. The thing that matters the most is the way the ABox is stored.

Changing the original DB model meant not only modifying the piece of code where the tables get created and filled in, but also the builder of an SQL query from a *DL−Lite* query. The original SQL builder basically takes all entities appearing in the query, generates a certesian product (for the SQL `SELECT ... FROM` command) of tables corresponding to entity types of the entities, and restricts the product using many conditions in the `WHERE` clause. This means a cartesian product of many large tables, since for each entity there is a single, large assertion table.

For example, for a query containing two class restrictions and an object role restrictions, e.g. the one in Listing 3.1, the builder generates an SQL query containing cartesian product of two copies of `concept_assertion` table and a copy of `object_role_assertion`,

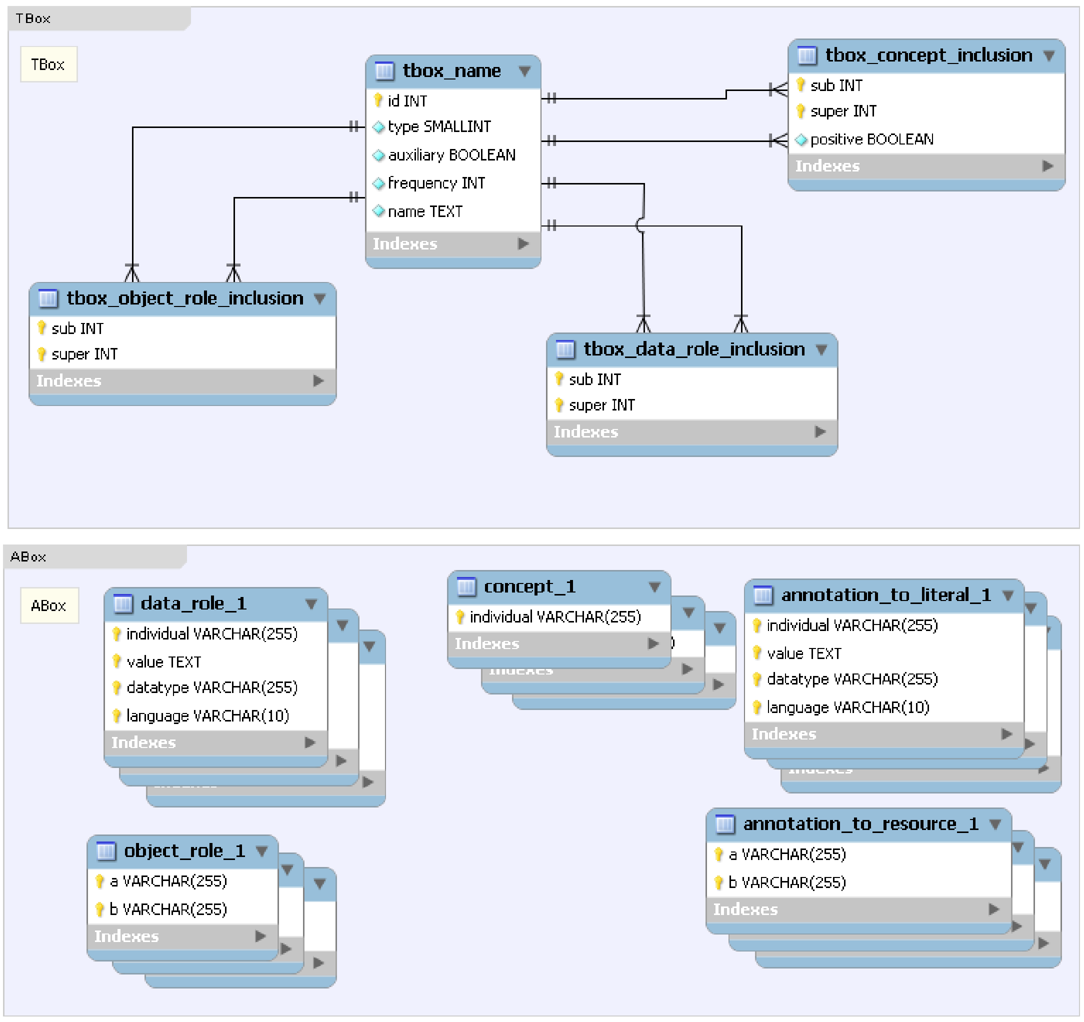

Figure 3.3.: Owlgres modified E-R diagram - multiple table QuOnto style

for an example see Listing 3.2. You can see another example in Appendix B.

Listing 3.1: An example SPARQL query

```
SELECT ?p ?a
WHERE {
  ?p rdf:type :Publication .
  ?p publicationAuthor ?a .
  ?a rdf:type :Professor .
}
```

Listing 3.2: Generated SQL for single table model for an example query

```
SELECT ca_0.individual AS x1, ca_1.individual AS x2
FROM  concept_assertion ca_0, concept_assertion ca_1,
   object_role_assertion ora_0
WHERE ca_0.concept=33
AND ora_0.object_role=62
AND ca_0.individual=ora_0.a
AND ca_1.concept=49
AND ca_1.individual=ora_0.b
```

The builder for the multiple tables models does a similar things, but instead of taking the complete tables for entity assertions, one table for each entity type, it takes assertion tables specific for each entity, thus typically significantly smaller. So, for the example query in Listing 3.1 used above, the cartesian product would consist of two smaller concept assertion tables for specific concepts, e.g. `concept_33` and `concept_49`, and a smaller object role assertion table for specific object role, e.g. `object_role_62`. This also drops some of the SQL `WHERE` condition clauses, since the restriction is performed by calling the specific table by its unique name. For an example, see Listing 3.3. You can compare other SQL queries for different models in Appendix B.

Listing 3.3: Generated SQL for multiple table model for an example query

```
SELECT ca_0.individual AS x1, ca_1.individual AS x2
FROM  concept_33 ca_0, concept_49 ca_1, object_role_62 ora_0
WHERE ca_0.individual=ora_0.a
AND ca_1.individual=ora_0.b
```

The difference in SQL builders for the two modified schemes, multiple table (MT) and multiple table QuOnto style (MTQ), is that the former needs to translate the found id's of individuals answering the query to their URI strings, while the latter does not. The translation is carried out by wrapping the generated SQL query (finding the id's) as inner query into another query, which takes the id's found and maps them to URIs using the `individual_name` table.

All builders must deal with a case, when there are more options for an individual in which table to be found. E.g. when a query restricts a variable to be a member of certain concept, which has multiple subsumed concepts. Then the builders generate an SQL block for each such subsumed concept, gluing them together with the SQL command `UNION`.

The advantages of the multiple table schemes do not come free. The schemes bring also some drawbacks to a DBMS. The most important one is that they issue many tables to the DB, in case of our tests with a large dataset it means around 1000 tables. It might be slow for a DBMS to find a certain table, among so many – it applies a heavy load to the filesystem support of the OS. So there is a dilemma whether to filter individuals by TBox entities using an *indexed column* of a few large tables, or to filter them by choosing a table from many, retrieval of which from a filesystem may be slow – a task which probably DBMS delegates to OS's *filesystem*, which is probably *not indexed.* But for real-case *DL−Lite* ontologies, the TBox contains typically small number of entities (let us say with the maximum of around 10^4), therefore this amount of files should not pose a problem to a DBMS or a filesystem support.

On the other hand, an advantage of the multiple table schemes might be that the smaller assertion tables for often used TBox entities (e.g. an often used concept table `concept_41`) get cached in the memory by the OS and/or the DB, making the query answering faster.

3.1.3. Measuring Performance

Some minor modifications were performed in order to test time spent in different parts of query answering. Measuring time of main entry points to the implementation, like connecting the DB, loading a TBox, and loading a query, was easy. Besides that, to obtain times of some interesting execution parts, e.g. performing the SQL query itself, it was necessary to implement "temporal probes" into a few classes.

For testing purposes, we measure eight different times obtained from the system, which are listed in Table 3.1.

Table 3.1.: Measured times in Owlgres query answering

Name	Abbreviation	Description
dbConnectTime	conn	The time it takes to connect to DB
tboxReadTime	tbox	Time of loading a TBox from DB
queryParseTime	qparse	Query parsing time, by Jena
queryReformulationTime	qref	Time of query reformulation into *DL−Lite*
sqlGenTime	sqlgen	Time of converting a query into an SQL
sqlTime	sql	SQL query execution time
queryExecTime	qexec	Total time of executing query, includes qref, sqlgen, and sql times
totalTime	total	Total time of whole query answering process

The distribution of the measured times, with the inclusions depicted, is in Fig. 3.4.

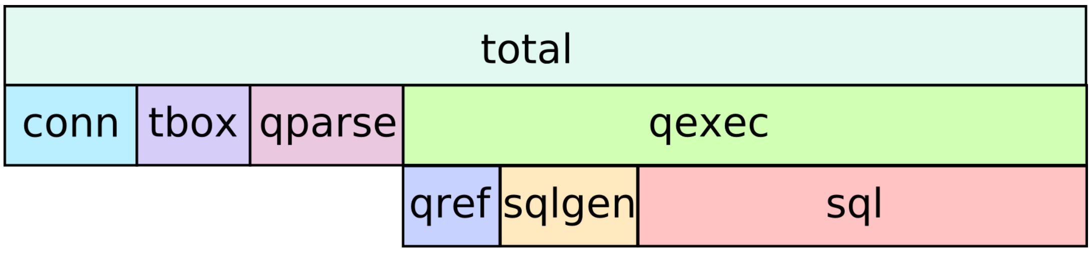

Figure 3.4.: Measured Owlgres query answering times distribution

3.1.4. Owlgres Issues and Bugs

The first issue was found when ABox loading performance was examined, because it seemed surprisingly slow. Authors mention in (Stocker and Smith, 2008) that the speed is relatively slow, but still it seemed there must be a problem with it. It was discovered that the class `ABoxAdditionProcessor`, which takes care of the loading process, probably contains a bug. As a result of time-profiling it showed up that each time an ABox assertion is inserted, the number used as primary key for `individual_name` table for the assertion is calculated, using the SQL `SELECT max(id) FROM individual_name` (in function `idForIndividual`), which is obviously taking some time.

I think the author's intention was to run the SQL only for the first time, and then just increment the value. The issue is in using the `maxIndividualId` variable, because it gets reset to zero every time an assertion is added (in function `add(DLLABoxAssertion assertion, Store store)`), so the new `maxIndividualId` needs to be recalculated by the SQL. The fix I have made results in about 1/3 faster processing on a large data set.

Another bug found was in the SQL query builder (class `SQLQueryBuilder`). Let us start with an example of SQL in Listing 3.4 generated for a test query posed to Owlgres.

Listing 3.4: A piece of generated SQL to demonstrate an Owlgres bug

```
SELECT name_0.name AS x1, name_1.name AS x2
FROM (
 ...
) as innerRel , individual name_0, individual name_1
WHERE  innerRel.x1=name_0.id
AND innerRel.x2=name_1.id

UNION

SELECT x1, name_1.name AS x2
```

```
11 FROM (
12  ...
13 ) as innerRel , individual name_0 , individual name_1
14 WHERE  innerRel.x2=name_1.id
15 ...
```

It is a union of a couple of blocks, from which only the first two are displayed. The query asks for annotation of some objects, and because it does not know if the annotation is a literal or a resource, it generates such union.

The issue is in the second block. In the first one, there are used two copies of the `individual_name` table in outer query (cartesian product) (line 4), which are then filtered using `WHERE` (lines 5 and 6). But in the second one there are still two copies of `individual_name` (line 13), but only one is used for `WHERE` condition (line 14) and for output in `SELECT` (line 10).

Thus the second block generates a huge result caused by the unrestricted cartesian product (in our case, the result of the rest of the query gave 5 rows, but with the second `individual_name`, the cartesian product gave 5 times the size of ABox. And since the second `individual_name` does not show in the output, it was the 5 rows repeated many many times, and hence filtered on the final output as duplicities, showing only the 5 rows. But the size of the result of this sub-query is huge, and takes the DB a long time to evaluate (in our case about 100s).

So the `SQLQueryBuilder` class was modified in the method `getSQL()`, where the unnecessary tables in cartesian product were removed.

There might be another problem with querying annotation data. When a certain query (giving a few results) was extended by adding a SPARQL statement asking for an object's label, e.g. by SPARQL row `?a rdfs:label ?label`, the query answer gave no results, even though the ABox contained the label annotation assertions for the individuals contained in the result of the original query. But this case would need more time to analyse the problem.

The last issue found is a problem with loading a TBox containing an axiom with complement of a concept. When there is such an axiom, the visitor design pattern for loading the TBox gets in an infinite loop. It is because the visitor's visit method `TBoxLoader.NormalizingDescriptionVisitor.visit(OWLObjectComplementOf desc)` calls itself by executing `desc.accept( this )`. This has been fixed by replacing it with the command `desc.getOperand().accept(this)`.

There is one more thing, but it might be not a problem of Owlgres, but of Jena's class `ResultSetFormatter` instead. The discovered behaviour is that when a result was printed using the `ResultSetFormatter` class, some columns with values got switched, making them not corresponding to their headers.

3.2. QuOnto

QuOnto is another system tested for performance on *DL−Lite* query answering. Its version 0.137 alpha was used. Since QuOnto misses any API, and lacks any command-line capabilities, the only way how to use it is through its GUI. The GUI provides rather simple interface consisting of three text panes, and a menu bar.

The three text panes are used for inserting a TBox, an ABox, and something called ECBox. ECBox stands for EQL constraints, which is a special language of the QuOnto authors, described in (Corona et al., 2008), for constraining the ABox using a combination of SQL and SPARQL similar to SparSQL.

The menu bar contains buttons for loading and saving a Tbox and an ABox, but these buttons only manipulate the contents of the text panes. The important buttons are `Run On QuOnto` and `Stop On QuOnto`, which runs processing of the TBox and the ABox. It probably loads an TBox from corresponding text pane just into internal representation in memory, instead of storing it in the DB, but it is uncertain. What is known for sure is that the ABox is really stored in the DB, the schema of which was revealed by chance due to bugs in the implementation described below.

There is a window for entering a query in different formats (Datalog, SPARQL, and SparSQL), from which we used only SPARQL and SparSQL languages. An important button is surprisingly the button `Generate Log File` in the menu. The log file itself is not very descriptive of what the application is doing, but it is the log file where all the Java exception generated in the application appear; the exceptions mostly do not appear elsewhere.

This way we found out the QuOnto's relational scheme, which is depicted in Fig. 3.5. As you can see, it is a multiple table model, where individual names are stored directly into assertion tables. The bug that allowed us to find the scheme was that the assertion tables use directly the names of the TBox entities they represent, and there is no restriction on the names. E.g. when there is a concept named "Woman," it creates a table named `Woman`. And since it uses the standard SQL commands, one can imagine, what would happen when there is a concept called “Select”. It tries to create the corresponding table using the SQL `CREATE TABLE SELECT`, where “select” is an SQL keyword. This command of course fails, generating an exception, which is recorded in the log. This property was found, when a testing TBox contained the object property “like”, which is also an SQL keyword. The SQL query for creating a table for object property “like”, as cought in an exception, looks like:

```
CREATE TABLE like (term1 varchar(255), term2 varchar(255), PRIMARY KEY(term1, term2))
```

The assertion tables are similar to the ones we used in Owlgres modification calling them multiple table QuOnto style (see Section 3.1.2), though these are a little simpler.

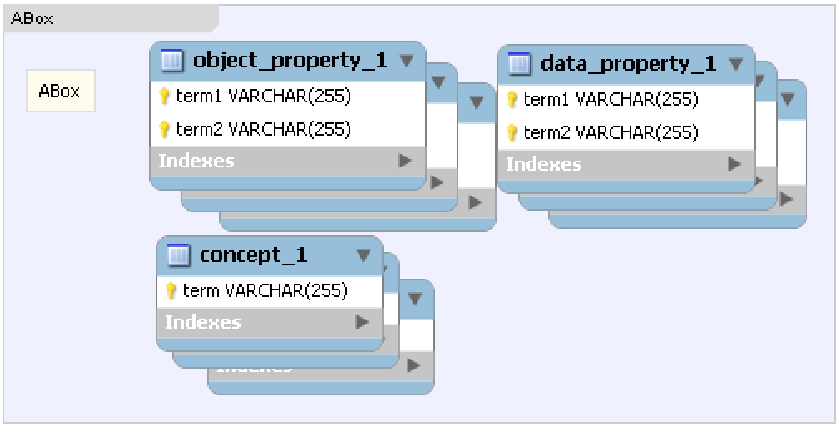

Figure 3.5.: QuOnto E-R diagram

Another useful bug in QuOnto allowed us to see what an SQL query generated from a *DL−Lite* query looks like. It is because of support for SparSQL querying language. Its syntax and usage are described in (Pasquale, 2008), and in (Corona et al., 2008). Simplifying it, it is basically a SPARQL query wrapped in a SQL query, where the SPARQL query is embedded by the construct “`sparqltable` (SPARQL query),” which behaves in the SQL context as an ordinary table. This is also confirmed by the observed behaviour of QuOnto, that the SPARQL query part is reformulated into SQL by the *DL−Lite* reasoner, and the resulting reformulated SQL is substituted into the SparSQL query in the place of `sparqltable` command, leaving the rest of the SparSQL query as is, and posing the final result to DB. This was discovered, when a mistake was made in a SparSQL query in the non-SPARQL part, because the part was not checked for correct syntax, and resulting concatenated query was posed to DB, giving an exception of wrong SQL statement, and the exception was found in the log.

For example an erroneous SparSQL is in Listing 3.5, where the inner SPARQL query has an output column named pn, while the outer SQL query asks for pnn column, which is an error not caught by QuOnto, but the query gets into DB, where it produces an exception.

Listing 3.5: An erroneous SparSQL query

```
select people.pnn
  from sparqltable (
    SELECT ?pn WHERE {
      ?p :httpuobiodtibmcomunivbenchliteowlname ?pn.
      ?p rdf:type 'httpuobiodtibmcomunivbenchliteowlPublication'.
      ?p :httpuobiodtibmcomunivbenchliteowlpublicationAuthor ?a.
      ?f :httpuobiodtibmcomunivbenchliteowlisFriendOf ?a.
      ?f :httpuobiodtibmcomunivbenchliteowllike
        'httpuobiodtibmcomunivbenchliteowlPainting'.
  }) people
```

Which results in the final SQL query posed to the DB in Listing 3.6. The DB then gives an exception stating that the column people.pnn is not found.

Listing 3.6: The resulting erroneous SQL

```
select people.pnn
 from (SELECT DISTINCT alias_1.term2 AS pn
  FROM httpuobiodtibmcomunivbenchliteowlpublicationAuthor alias_0 ,
     httpuobiodtibmcomunivbenchliteowlname alias_1 ,
     httpuobiodtibmcomunivbenchliteowlisFriendOf alias_2 ,
     httpuobiodtibmcomunivbenchliteowllike alias_3
  WHERE alias_0.term1=alias_1.term1  AND alias_0.term2=alias_2.term2
     AND alias_2.term1=alias_3.term1  AND alias_3.term2='
     httpuobiodtibmcomunivbenchliteowlPainting'
 ) people
```

This again confirms the relation scheme to be of the type called multiple table model, where every TBox entity is represented by a single table having the entity name.

3.2.1. Data Preprocessing

Since QuOnto does not use OWLAPI and does not provide full RDF/XML support by other means, both TBox and ABox need to be accommodated to serve as QuOnto input. QuOnto requires them in functional syntax, but the support for it is rather limited – all names cannot be URIs, but identifiers consisting of alphanumeric characters only, and no namespaces are supported. Because both testing datasets use URIs as entity and individual identificators, another conversion needs to be performed. One more complication is that QuOnto uses the TBox entity identifiers directly for table names, which are not case sensitive. Therefore entity names cannot differ just in character case (e.g. when a concept and an object role name is the same, except the first letter is capital for the concept name).

Conversion from the RDF/XML format of used datasets to functional syntax was easy, because it is fully supported by OWLAPI. The complicated part was to adjust the names to fit into QuOnto.

First, all namespace prefixes of names had to be replaced by the namespace name. Then, capital letters of names were doubled, to distinguish the names from the same names with lower case letters. Another thing to modify was to remove the data role values quotation marks (not the escaped ones), since it is not recognized by QoOnto. Then all non-alphanumeric characters were removed. The axioms containing too long names had to be removed, too, because the assertion tables do not support names longer than 255 characters. Things to artifically add to the TBox were all data role ranges, when missing, because QuOnto fails to compile the TBox when there is a range axiom missing. There was a couple of other things to change or remove.

When converting one of the testing datasets, DBpedia (see Section 4.1.2), which contains a lot of different namespace prefixes used, a bug in OWLAPI was discovered. This was an issue in the latest OWLAPI release available, version 2.2.0. Also the latest build from OWLAPI SVN was tried, but it gave an exception without much useful information. Because of the need to quickly convert the DBpedia data to functional syntax, the problem was solved by using an older version of OWLAPI, 2.1.1.

The problem with namespaces in OWLAPI of version 2.2.0 was debugged in detail, and the results were sent to its authors, who confirmed this as a bug. The bug appears as freezing during the conversion. For a reference, it spends all the time when frozen in method `NamespaceUtil.generatePrefix`, where it generates huge amount of prefixes.

4. Testing

From all reasoning procedures (e.g. consistency checking), only the query answering was tested, as being the most interesting and complex one.

4.1. Datasets

4.1.1. UOB

University Ontology Benchmark (UOB) is a dataset developed by IBM for testing performance of OWL-DL reasoners. It is an extension of the well-known Lehigh University Benchmark[1] (LUBM).

UOB is a part of IBM Integrated Ontology Development Toolkit, downloadable at `http://www.alphaworks.ibm.com/tech/semanticstk/download`.

It provides two TBox files with expressivity levels of OWL-Lite – $\mathcal{SHIF}(\mathcal{D})$ and OWL-DL – $\mathcal{SHION}(\mathcal{D})$. The OWL-Lite version was simplified to fit the $DL-Lite$ expressivity, and used for all tests. The simplifications made are removing unsupported role properties (symmetric, functional, and transitive), removing role inclusion axioms, reformulating/removing number restrictions, and reformulating equivalence axioms.

The ABox contains various facts about university students and staff, their publications, courses their take/teach, research groups they belong to, etc. There are three versions of ABox, with one, five, and ten universities. A university has 20 departments on average, another division entity of a university is a college.

There are two queries posed to the tested reasoners formulated for UOB data. The first one in Listing 4.1 asks for all publications and their names, which have an professor author with a student friend, who share the same hobby.

The query in Listing 4.2 retrieves all students that are taught by, have published

[1] `http://swat.cse.lehigh.edu/projects/lubm/`

something and are teaching with someone from the same university part; it also includes advisors of such students. Note that this query uses an undistinguished variable (`_:a`), which is probably not supported (as far as a source code investigation can tell; nothing about it was found in its documentation). The query was originally designed for LUBM dataset, on which it gives a single result. Here it unfortunately gives no results, which might be because of the lack of undistinguished variables support.

Listing 4.1: UOB SPARQL query 1

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX uob:  <http://uob.iodt.ibm.com/univ-bench-lite.owl#>

SELECT ?p ?pn
WHERE {
  ?p uob:name ?pn .
  ?p rdf:type uob:Publication .
  ?p uob:publicationAuthor ?a .
  ?f uob:isFriendOf ?a .
  ?f rdf:type uob:Student .
  ?f uob:like ?hobby .
  ?a uob:like ?hobby .
  ?a rdf:type uob:Professor .
}
```

Listing 4.2: UOB SPARQL query 2

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ub:  <http://uob.iodt.ibm.com/univ-bench-lite.owl#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>

SELECT *
WHERE {
        ?L rdf:type ub:University.
        _:a ub:subOrganizationOf ?L.
        ?E ub:isMemberOf _:a .
        ?A ub:isMemberOf _:a .
        ?E ub:takesCourse ?Z .
        ?A ub:teacherOf ?Z .
        ?A ub:teacherOf ?Z2 .
        ?E ub:isAdvisedBy ?B .
        ?E ub:teachingAssistantOf ?Z2 .
        ?Q ub:publicationAuthor ?E .
        ?Q ub:publicationAuthor ?A .
}
```

4.1.2. DBpedia

DBpedia is a community effort to extract structured information from Wikipedia[2] and to make this information available on the Web[3].

It consists of some structured data from Wikipedia, obtained from a few language versions of Wikipedia, namely English, German, French, Spanish, Italian, Portuguese, Polish, Swedish, Dutch, Japanese, Chinese, Russian, Finnish and Norwegian, which gives around 274 million RDF triples in total. It describes over 2.6 million things, including more than 213,000 persons, 328,000 places, 57,000 music albums, 36,000 films, 20,000 companies, etc.

The current dataset, which was used for testing, is of version 3.2, and can be downloaded at `http://wiki.dbpedia.org/Downloads32`.

Because the complete DBpedia dataset is huge, and the testing of it would be impractical, only a subset has been selected. First, only some parts of it were used (each DBpedia part is in a separate file). The selected files are in Table 4.1. Next, not the entire files were used, but the maximum of 100,000 triples per file was employed for testing.

[2]`http://www.wikipedia.org/`

[3]`http://wiki.dbpedia.org/About`

Table 4.1.: Used DBpedia files

Label	File name
Titles	`articles_label_en.nt.bz2`
Categories (Labels)	`categories_label_en.nt.bz2`
Disambiugation Links	`disambiguation_en.nt.bz2`
Persondata	`persondata_en.nt.bz2`
Redirects	`redirect_en.nt.bz2`
Ontology Types	`types-mappingbased.nt.bz2`
External Links	`externallinks_en.nt.bz2`
Homepages	`homepage_en.nt.bz2`
Images	`image_en.nt.bz2`
Links to Wikipedia Article	`wikipage_en.nt.bz2`
Ontology Infoboxes	`infobox-mappingbased-loose.nt.bz2`

The ABox files are zipped, containing the data in N-triple format. OWLAPI does not support this format, so Jena was used to convert the N-triples into RDF/XML format. Unfortunately, Jena apparently works the way that it loads the entire input file into memory before commencing the conversion, which prohibited to convert some of the selected files (the biggest one has over 800MB, which in internal representation in memory takes even more). Therefore this was another reason for the afore-mentioned 100,000 triple limit on a file.

The website also provides an ontology TBox for the data, which is labeled DBpedia Ontology, and resides in the file `dbpedia-ontology.owl.bz2`. Though the TBox does not use complex language, it does not fit into $DL\!-\!Lite$ expressivity – it uses concept union on the top of it, so the TBox was simplified for the tests. The simplifications made are rewriting axioms containing union of concepts (role domain axioms, object role range axioms) to axioms containing a fresh auxiliary concept instead, where the auxiliary concept is on the left hand side of multiple fresh concept inclusion axioms, with the right hand sides of all the concepts of the union. Another modification is removing all date role range axioms.

There are two queries posed to the tested reasoners formulated for DBpedia data. Note that both use the `distinct` keyword to display only distinct results, which was added in order to try a different feature in the queries. The first one in Listing 4.3 asks for two individuals. The first of them has to have his given name equal to someone else's surname, and the second individual has to have his surname equal to someone else's given name. Additionally, the first individual has to be disambiguated (a Wikipedia disambiguation article has to lead to him). And the final restriction is that an artwork of the first individual and an artwork of the second individual have to have similar genres. Similar in this case means that the genres meet in being defined as genres of a single object.

The other query in Listing 4.4 just adds the restriction, that both the individuals have to be artists. This might seem like a simple and insignificant modification, but it actually complicates the resulting SQL query quite a lot (see test results in Section 4.2).

Listing 4.3: DBpedia SPARQL query 1

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?a1 ?a2
WHERE {
  ?a1 foaf:givenname ?name1 .
  ?jmenovec1 foaf:surname ?name1 .
  ?a2 foaf:surname ?name2 .
  ?jmenovec2 foaf:givenname ?name2 .

  ?d dbp:disambiguates ?a1 .

  ?dilo1 dbo:artist ?a1 .
  ?dilo2 dbo:associatedMusicalArtist ?a2 .

  ?dilo1 dbo:genre ?g1 .
  ?dilo2 dbo:genre ?g2 .
  ?cosi1 dbo:genre ?g1 .
  ?cosi1 dbo:genre ?g2 .
}
```

Listing 4.4: DBpedia SPARQL query 2

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?a1 ?a2
WHERE {
  ?a1 foaf:givenname ?name1 .
  ?jmenovec1 foaf:surname ?name1 .
  ?a2 foaf:surname ?name2 .
  ?jmenovec2 foaf:givenname ?name2 .

  ?d dbp:disambiguates ?a1 .

  ?dilo1 dbo:artist ?a1 .
  ?dilo2 dbo:associatedMusicalArtist ?a2 .

  ?dilo1 dbo:genre ?g1 .
  ?dilo2 dbo:genre ?g2 .
  ?cosi1 dbo:genre ?g1 .
  ?cosi1 dbo:genre ?g2 .

  ?a1 rdf:type dbo:Artist .
  ?a2 rdf:type dbo:Artist .
}
```

4.2. Owlgres

The performance tests of Owlgres were carried out on a standard laptop computer with an Intel Core2 Duo CPU running at 1.6 GHz and 2 GB of RAM, with two operating systems installed: Microsoft Windows XP Professional SP3 (32-bit) and Kubuntu 9.04 (Jaunty Jackalope) Beta 64-bit, a distribution of Linux.

Running Owlgres requires Java Virtual Machine (JVM), the used one was Java from Sun of version 1.6.0_10. The DB used was PostgreSQL of version 8.3.0 for a few first tests on Windows, but it was upgraded to version 8.3.7, which sped up one of the tests; the version on Linux was always 8.3.7.

The tests were performed for all combinations of the following:

- Two datasets (UOB and DPpedia) – abbreviated by `uob` and `dbp`
- Two queries (two for each dataset) – denoted by number 1 and 2

- Three DB schemes (single table, multiple table, multiple table QuOnto style) – abbreviated by `st`, `mt`, and `mtq`

Owlgres was run in the command-line mode, with a JVM option allowing it to use up to 1 GB of RAM. Several Owlgres execution scenarios were tried, but in order to keep the repeated tests as independent as possible, the JVM for Owlgres was restarted for each set of tests (where a set is a sequence of 12 tests: combinations of two datasets, two queries, and three DB schemes), and also the DBMS (PostgreSQL) was restarted after each set. There was 20 of these sets repeated – to have more data allowing for some simple statistical processing (computing standard deviation). Restarting the DBMS suppressed DBMS using caches for tables and their indices for subsequent tests, letting them to read the data from persistent storage every time. Restarting the JVM prevented from caching objects and probably some other optimizations a JVM may do.

The first chart in Fig. 4.1 compares total times of all datasets, all queries, and all schemes both in Windows and Linux. The total times mean times of complete querying process, i.e. connecting to the DB, loading a TBox, processing a query and running a reformulated query on the DB. Note that there are standard deviations of the measured times (with respect to the 20 repeated test sets) marked using the thin line in the shape of the letter "I".

There was a problem measuring the time of the second query on DBpedia on single table scheme DB. It took so long it was not practical to wait for a result (it was run overnight). This case is marked in all charts with full-height bars with a discontinuity mark (shaped like two thunderbolts).

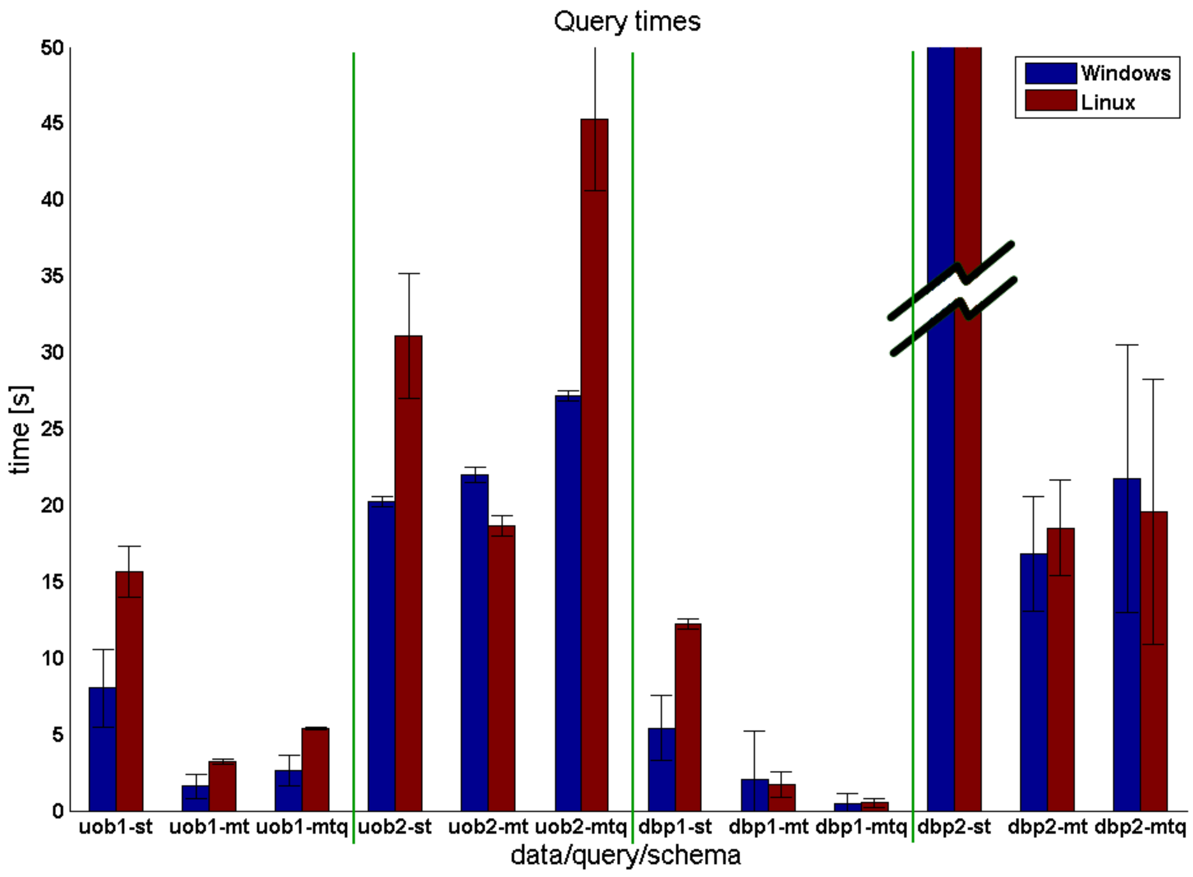

Figure 4.1.: Owlgres query times

Follow two another charts in Fig. 4.2 and Fig. 4.3, each for an operating system. These show how is the total time of query execution divided into steps of the query answering. For explanation of the abbreviations used for the answering steps see Section 3.1.3.

It manifests that the most of the query answering time takes executing an SQL query by the DB. almost all other steps have neglectable times, except the query reformulation of the second query for the UOB dataset (part of the three bars is yellow). It is because the query is very complex, and its reformulation to *DL−Lite* requires many query modifications. The process of reformulation is of course independent of the DB model used, thus the three yellow parts are of about the same height.

The non-SQL steps for the first query of the UOB dataset, single table model (the leftmost bar), also consumed more time then usual in other cases. But this is because this test was always the first in the set of tests (the set was run repeatedly, but always in the same order), and the time spent here is probably mostly due to JVM loading Java classes, libraries, and performing other initializations.

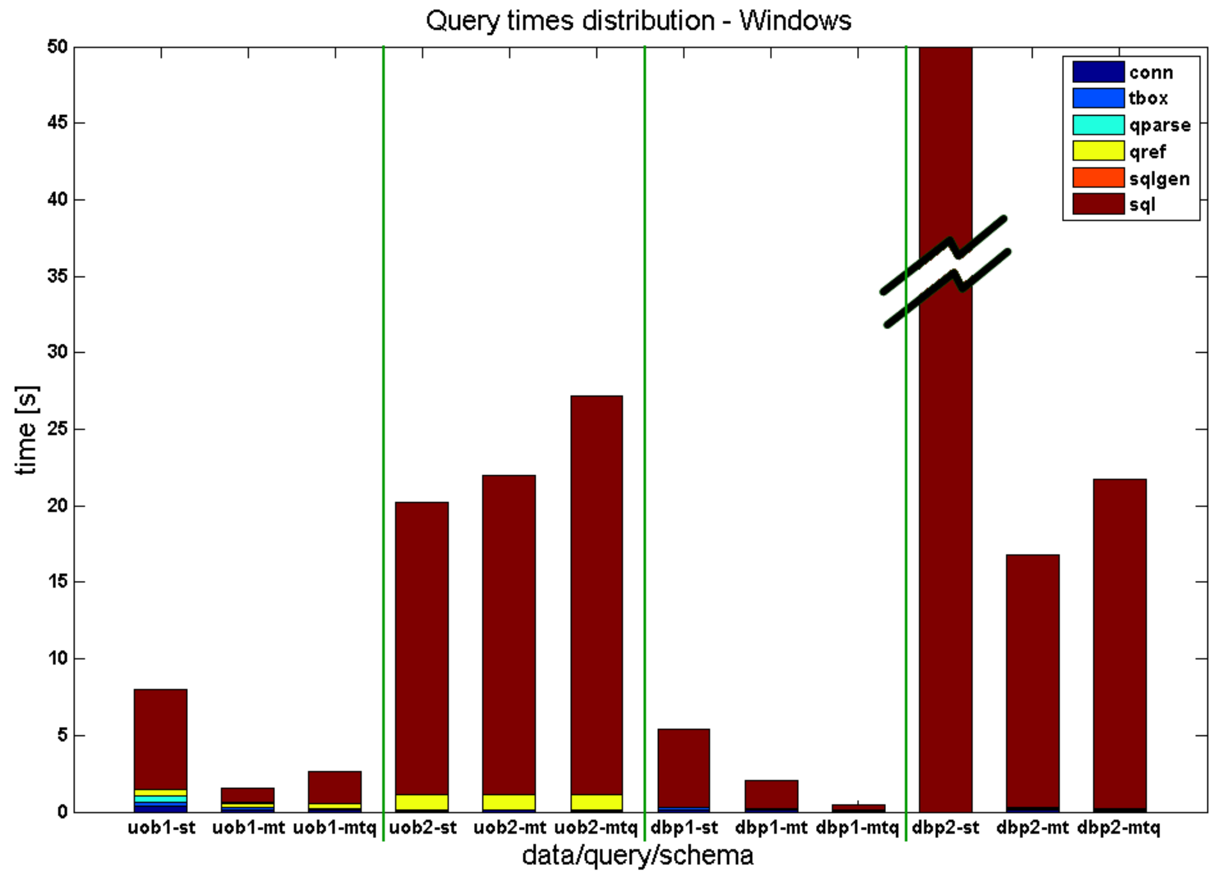

Figure 4.2.: Owlgres query times distribution on Windows

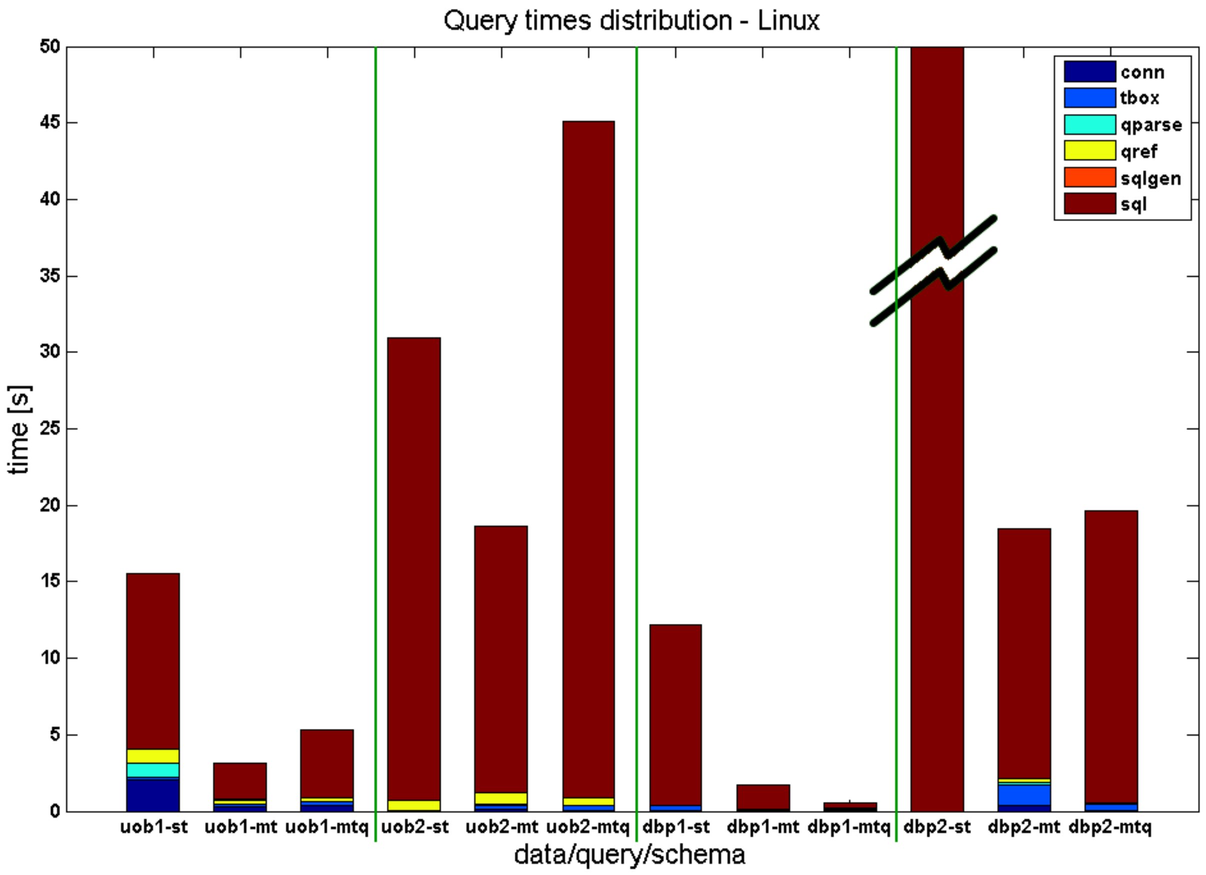

Figure 4.3.: Owlgres query times distribution on Linux

An interesting situation appears in Fig. 4.4. It shows times measured in very early tests, when on the Windows platform accidentally there was an older version of PostgreSQL, version 8.3.0. Most of the results are basically similar, except for the first test query for DBpedia dataset with single table model, for which the total time was significantly higher then in all other tests. Of course all the long time was spent processing an SQL query by the DB. It is hard to estimate what is the cause for it being so long, probably the older version of DB missed some minor optimization, which applied to such complex query for the single table model (the SQL queries for multiple table models are quite simpler). It may be similar problem with the second DBpedia query (taking practically immeasurable amount of time), which is not very complex, but maybe would need another database optimization, which PostgreSQL could be missing. Note that the times cannot be exactly compared to the results in the previous charts, because the testing conditions were different (the DBMS and the JVM was not restarted).

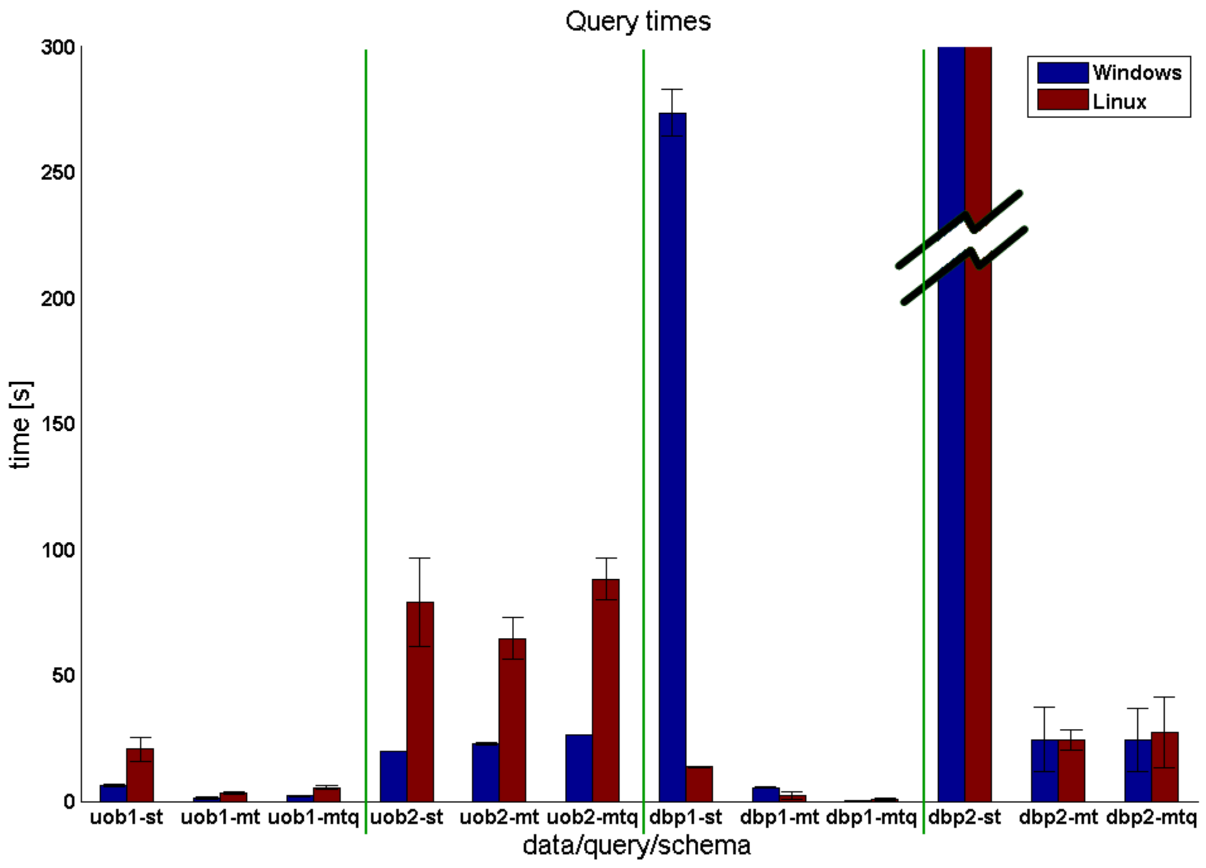

Figure 4.4.: Owlgres query times – first test, with Windows PostgreSQL version 8.3.0

4.3. QuOnto

The tests were performed on the same machine as described in Section 4.2. The JVM version was 1.6.0_10 again. The database QuOnto uses is the H2 embedded Java database. The version of H2 bundled with QuOnto is 1.0.74. A newer version 1.1.112 of H2 was tried out, with a hope it might speed up the tests, but with no success. All the QuOnto tests were performed on Windows platform, since it completely runs in a JVM, which erases most of the platform differences. Note that most of the tests in this section are performed with QuOnto coming from the QToolKit package. Some tests were performed with another system implementing QuOnto, the ROWLKit, which are at the end of this section.

All QuOnto testing was complicated, and a few bugs were encountered. With some of them the QuOnto authors were confronted, resulting in a brief response stating that this prototype should be understood as an evidence that there is a real implementation of $DL-Lite$, that it is not just a theory in the authors' papers.

The problems resulted in rather complex data preprocessing, as described in Section 3.2.1. Another difficulty is with measuring its performance – there is no easy way how to measure the time. There is no way how to measure times of different steps of the query answering process (query reformulation, SQL execution, etc.), since QuOnto does not provide any API. And also measuring the total answering time was not easy, since the only way how to execute a query is pressing a button in its GUI, with the answering finish demonstrated only by showing results. And since it generally takes a long time to answer a query in QuOnto, it is impractical to wait with a stopwatch until an answer appears. Thus a very course solution was used: a simple utility measuring CPU load was made, which shows the time when the CPU load goes over/under certain threshold. And because the query answering in QuOnto heavily utilizes the CPU, using the utility allowed to measure the time of the query answering.

Due to big differences in time performance the results are not displayed in charts. Instead, our analysis is supported by a few tables – Table 4.2, Table 4.3, Table 4.4, and Table 4.5, that depict the performance of the most interesting tests.

First, the same datasets as for Owlgres were used, with the full ABoxes. The resulting total query answering times are in Table 4.2. You can see here, that only the time for the UOB dataset query 2 was obtained, while the others were inaccessible, because took too long (one of the tests was run overnight). Which is a real surprise, since the query 2 of UOB was one of the most difficult queries, and it finished earliest in this test (though later than in any of the Owlgres tests).

Table 4.2.: Measured times of QuOnto query answering, full ABox

Dataset	Query	Time
UOB	1	> 11h
	2	**316s**
DBP	1	> 1h
	2	> 1.5h

In order to find out what the unmeasured times are like, the reduction of ABox was administered (instead of simplifying the queries, which may be quite unpredictable regarding the answering complexity – it is dependent on usage frequency of TBox entities, etc.). First, the ABoxes of both datasets were reduced to 1/10 of the original ones (10%). From the Table 4.3 you can see it was still impossible to find out DBpedia queries answering times. Note that in the tests for DBpedia with the full ABox, the times for which the queries were run (wihtout an answer) are shorter than for the 1/10 ABox case. It is because we expected the queries for 1/10 of ABox to finish, thus we waited for a longer time.

Table 4.3.: Measured times of QuOnto query answering, 1/10 of ABox

Dataset	Query	Time
UOB	1	**1449s**
	2	**8s**
DBP	1	> 14h
	2	N/A

Thus the ABoxes were reduced once again, to 1/20 of the original ones (5%). And the result is that the DBpedia query 1 time is obtained, but not the time for the query 2, as seen in Table 4.4.

For this size of ABoxes, also a newer version of H2 database was tried. It has been tested on both queries for DBpedia, with worse results: for query 1, it did not finish in one hour (as opposed to 1370s with the older version), and for query 2 it did not give a result in reasonable time as well.

Table 4.4.: Measured times of QuOnto query answering, 1/20 of ABox

Dataset	Query	Time
UOB	1	**185s**
	2	**6s**
DBP	1	**1370s**
	2	> 4h

Therefore another strong reductions of the ABoxes to 1/400 of the original ones were made (0.25%) (using 1/100 still did not lead to a result for the DBpedia query 2 in a reasonable time). Now all the times are known, as seen in Table 4.5. Note that the time measuring granularity is 1s, thus the low times of 2s are very coarse-grained.

Table 4.5.: Measured times of QuOnto query answering, 1/400 of ABox

Dataset	Query	Time
UOB	1	**2s**
	2	**2s**
DBP	1	**10s**
	2	**181s**

The QuOnto implementation ROWLKit offers a lot of advantages compared to QToolKit. Some of them are the support for OWLAPI (this eliminates the need for data preprocessing), and a more informative GUI, which shows the time of the end of a query answering process (no need for CPU load monitoring). The important disadvantage is the database model is not known, because no flaws revealing it were found, and again there are no source codes nor any detailed documentation.

But ROWLKit gave much better results for the full ABox of UOB dataset, which are in Table 4.6. It probably uses the H2 more efficiently, but it is hard to know how, since the DB schemes are not known, nor are the SQL queries posed to the DB. Unfortunately, it was impossible to even load the DBpedia data, because the ROWLKit froze up, perhaps because of lack of memory. An attempt on Linux was made, assigning more memory to the JVM than on Windows (up to 3GB), but with no success, the application froze up again (endless execution overnight).

Table 4.6.: Measured times of ROWLKit query answering, full ABox

Dataset	Query	Time
UOB	1	**22s**
	2	**8s**

4.4. Comparison

Comparing the results of two different systems implementing $DL-Lite$ – Owlgres in Section 4.2 and QuOnto in Section 4.3 – is not entirely fair, because the systems use quite different data storage.

One could anticipate that the QuOnto times may be shorter, since it does not persist knowledge base on the harddrive (as Owlgres), but it keeps it in the RAM, which is much faster. than in contrast to Owlgres using a database with permanent hard-drive storage. The disadvantage of this approach is that it consumes a lot of RAM. As seen in the results below, sometimes the application froze up. The answer might be that though the H2 database uses much faster RAM storage, it lacks many optimizations that other well-established databases have. Another explanation is that the Java based H2 is being interpreted, which might be slower than the native code of PostgreSQL.

In spite of the anticipations, Owlgres was more efficient in most cases. There are two major explanations for that: Owlgres has a better reasoner, reformulating queries into more optimized SQL queries, than QuOnto has, or that the H2 is not as optimized as the PostgreSQL database is.

Both things have something to do with optimizations. One opinion could be that when there is not very optimized SQL query generated by a reasoner, the database would internally optimize it, making up for the reasoner; this capability may be missing in H2. But this is not entirely true, since there are optimizations a database cannot make, those that require the knowledge of TBox, which only reasoners have. For example, Owlgres keeps track of frequencies – numbers, how many times each TBox entity is used, which helps formulating the SQL query in order to start with the least frequently used entity (concept, object role, etc.) first.

Even though QuOnto has generally much longer query answering times, one of the result is significantly better then other Quonto times – the result for the UOB dataset, query 2. While Owlgres has for UOB query 2 almost the worst time (except for DBpedia query 2), and in general, query 2 for UOB is very difficult, which is also confirmed by the long time Owlgres needs to reformulate it into *DL−Lite*. This means the QuOnto reasoner has probably some smart optimization for this case, which Owlgres does not have. Note that we are not absolutely certain about it, because the query does not return any results in both cases, thus one of the reasoners might work incorrectly, and would return no results even when there should be some.

A more interesting will be the comparison of different database relational schemes in Owlgres now. First notes are regarding the query answering time distribution. As Fig. 4.2 and Fig. 4.3 show, most of the time spent during a query answering is in executing its SQL query. Besides a few deviations (e.g. a bit longer initial time for the first test (UOB,1,st) probably because of JVM caching, and some unresolved slightly longer times mainly for Linux testing), the only significant exception is query reformulation for UOB dataset, query 2, all schemes, which only proves that the query is rather complex (also confirmed by the complex SQL query in Appendix B), and it has nothing to do with the scheme.

The most important result is that the multiple table schemes are significantly faster then the single table scheme in most cases. The only case where it is not entirely true is the aforementioned query 2 for UOB dataset, where the single table model is a few percent faster on Windows, but still is slower with higher difference on Linux, resulting in a rather balanced performance. As mentioned before, the query is more complex than the others, which probably somehow erases some scheme differences (for Windows, the UOB query 2 provides the most balanced results among different schemes). The single table scheme even completely fails for the query 2 of DBpedia, for which case the database was not able to give an answer (when run overnight) – the cartesian product of huge tables generated by the generated SQL query was probably so big the DB was unable to cope with it.

Comparing the two multiple table schemes, the one with an individual names table and the QuOnto style, a little better results give the one containing and individual names table (called simply multiple table model). It gives shorter times for most of the tests except the one for query 1 for DBpedia, where it is significantly slower. An explanation is that using the individual names table (thus using numerical identifiers for individuals in all assertion tables) makes looking for the actual query answer easier (using DB indices for numeric columns), with an additional query translating the resulting numeric identifiers into the actual individual names using the individual names table. This makes answering most of the queries faster because of the more efficient query for individual identifiers. While the DBpedia query 1 is faster because the query is rather simple, so most of the time takes translation from individual identifiers to their actual names, whereas the multiple table QuOnto style scheme leaves this out, producing the individual names directly.

Other tests could be performed to find out how the query atoms reordering affects the query answering performance, since a reasoner can have a query atoms reordering optimization. But these tests have not been performed, because having in mind how a query gets reformulated into an SQL query, we assume that the optimization is performed by a DBMS (in ordering the tables in a cartesian product in the query).

5. Application

5.1. Netcarity

Netcarity[1] is a European research project for helping older people to improve their safety at home using modern technologies. Participants include academic institutions (e.g. Czech Technical University in Prague (CTU), Eberhard Karls Universitaet Tuebingen and Universita degli Studi di Pavia), organisations (e.g. Provincia Autonoma di Trento in Italy), firms (e.g. Stichting Verpleging en Verzorging Eindhoven e.o. De Archipel in Netherlands), and other companies and authorities.

The project investigates ways how to make living of elderly people on their own possible, without the need to aggregate them in care homes. Because self-sustainability of older people is usually somehow impaired, there are some problems when living alone, e.g. the risk of injury (when falling down, etc.), and other every-day problems that older people find often difficult. These problems must be taken care of, and Netcarity attempts to resolve them using new and existing modern electronic technologies. It tries to improve elderly people wellbeing, independence, safety and health at home.

The main task of CTU within the project is to build the central server for the system, accessed through several web services as well as web interface. The developed server is using PostgreSQL database for data storage and GlassFish application server. The stored data consist mainly of messages between the household and central server which are mostly observations of sensors, and commands.

The result of this thesis can be applied to the sensoric data stored within the Netcarity system, in order to allow intelligent and efficient data retrieval and complex querying. This means developing an appropriate structure of sensors, represented in a sensor ontology. The ontology must include the information about the actual sensors (type, manufacturer, location, range, operating conditions, etc.), ontologies of the quantities they measure (time ontology, representing vectors, arrays, binary data, etc.), and some rela-

[1]`http://www.netcarity.org/`

tions of physical principles they use for measuring. The interconnection of this extension with the current system of storing sensor data, i.e. the ABox storage model, needs to be as seamless as possible, to allow for efficient retrieval and querying, and to reduce the original DB scheme modifications.

The choice of a suitable TBox constructed is a matter in Section 5.2.1. Next, Owlgres (the only open source implementation available) was selected as the reasoning and query answering system. It had to undergo some modifications, to allow a smooth integration. The implemented adaptation of Owlgres is in Section 5.3.

5.2. Sensors

A brief introduction to sensors used in the Netcarity project follows, as described in its internal documentation. Currently, the sensor system is focused towards people protection against an accident of falling down and fire protection. The fall detection is based on three kinds of information:

- Motion acceleration of the monitored person
- Sound recording
- Visual surveillance

Combination of the three information gives high rate of reliability, eliminating false alarms. It might seem the readings of the motion acceleration of a person would be enough for detecting a fall, but it also would give some false alarms (false positive results).

The appropriate sensors are obvious: an accelerometer attached to the monitored person, and a camera and a microphone installed in every room. The topology of the system has a root in a communication device called “home gateway” (HGW), which is connected to the Internet, and distributes information between all devices installed in the house and the central server.

The accelerometer is a tree axes type, connected with a preprocessing CPU unit and wireless transmitter, sending the measured and processed values to an embedded PC. The embedded PC is in every room, combining the wirelessly received acceleration data with image data from a camera and waveforms from a microphone installed in every room as well. The camera and the microphone have their preprocessing units also. The preprocessing units reduce the data flow in the system by extracting the usefull information from the measured data – a case of data extraction. Note that the camera is 3D – among standard 2D color information it also senses the depth of the scanned area.

5.2.1. State of the Art in Sensor Ontologies

In order to categorize the sensors, their properties, and the data obtained from them, a suitable ontology TBox had to be found. It must support defining different sensor properties (e.g. range, location, physical principle), their interfaces, physical processes they measure, and representation of sampled data and their properties (e.g. time data, vectors, arrays), etc. Because developing such ontology is not easy and takes a lot of time, first some research was conducted, which is described in the following list of interesting efforts found.

W3C Incubator

The Semantic Sensor Network Incubator Group, part of the W3C[2] Incubator Activity, is an interesting website leading to a few other sites concerned with semantic description of sensors, and it is a prove that this task is an up-to-date problem, which is widely interesting, since the W3C is a leading organisation of developing standards for the Internet.

Here are some interesting links to projects regarding sensor ontologies, and an ontology prototype

- `http://www.w3.org/2005/Incubator/ssn/` – the incubator main site
- `http://www.w3.org/2005/Incubator/ssn/charter` – Semantic Sensor Network Incubator charter, with some interesting references
- `http://www.w3.org/2005/Incubator/ssn/wiki/Semantic_Sensor_Network_Ontology` – a not very useful sensor ontology by Nguyen (follows OWL-S significantly)
- `http://en.wikipedia.org/wiki/Semantic_Sensor_Web` – an unrelated site, Semantic Sensor Web – a Wikipedia article with some interesting links

The W3C Incubator is currently alive, with its end planned on 2.3.2010.

[2]The World Wide Web Consortium (W3C), at `http://www.w3.org/`, is an organisation developing interoperable technologies for the Web

SensorML

SensorML is a very good standard for sensor description defined by Open Geospatial Consortium[3]. It is very detailed and general, which allows many sensor ontologies to use this standard. Unluckily, it does not come in the form of a formal ontology, but it is a definition of modeling and XML encoding of sensor systems. Basically all ontologies found during this research follow this standard in some way.

- `http://www.opengeospatial.org/projects/groups/sensorweb` – Open Geospatial consortium - Sensor Web Enablement
- `http://www.opengeospatial.org/standards/sensorml` – SensorML – a very good standard for sensor systems (not in OWL)
- `http://portal.opengeospatial.org/files/?artifact_id=21273` – link for downloading the standard
- `http://lists.opengeospatial.org/pipermail/sensorml/2008-March/000392.html` – a mailing list conversation asking whether there is "SensorML model in OWL or RDF format ?", containing some interesting references

Marine Metadata

Marine Metadata Interoperability[4] develops the "Ontology for Devices," an ontology of oceanographic sensors and samplers. It is still a work in progress, with the current version of the ontology in its SVN, but apparently not complete yet. Web site contains a lot interesting UML diagrams, term dictionaries etc., and more importantly – links to other similar standards (the way the OntoSensor, the best ontology found, was discovered).

- `http://marinemetadata.org/community/teams/ontdevices` – Main website of the project
- `http://marinemetadata.org/node/2307/documents/General+Documents` – General documents

[3]The Open Geospatial Consortium (OGC), at `http://www.opengeospatial.org/`, is a non-profit, international, voluntary consensus standards organization that is leading the development of standards for geospatial and location based services

[4]Marine Metadata Interoperability, at `http://marinemetadata.org/`, is a project promoting the exchange, integration and use of marine data through enhanced data publishing, discovery, documentation and accessibility

- `http://marinemetadata.org/community/teams/ontdevices/ontdevrel` – Website linking to other websites with device related ontologies
- `http://sourceforge.net/svn/?group_id=170123` – MMI SVN link (or run `svn co https://mmi.svn.sourceforge.net/svnroot/mmi mmi`)

Marine Metadata Device Ontologies Working group is currently alive.

OntoSensor

Probably the most advanced sensor ontology found. Though its development is still in progress, it contains many useful things from SensorML, data representation etc. May be neccesary to extend, but a very good base for developing ontology for our needs. An article (below) describes a use-case similar to the case of this thesis – data storage in DB, which are queried.

There had been a problem with the ontology regarding the `hasValue` property – the ontology editor Protege 4.0 Beta[5] reports an error upon loading it. It is because the authors used Protege of version 3.3.1, which is more benevolent in OWL parsing. They defined the `hasValue` property as an object property with the range of `CapabilitiesDescription` concept, but then used it as a datatype role with string values. We we changed the definition to datatype role with the string range, and Protege 4.0 Beta loads the ontology without any error.

- `http://www.memphis.edu/eece/cas/projects.php` – its project homepage at Memphis university
- `http://www.memphis.edu/eece/cas/onto_sensor/OntoSensor.txt` – the ontology itself
- `http://www.ontologyportal.org/translations/SUMO.owl.txt` – the SUMO ontology, which is a dependency of the OntoSensor ontology (but not accessible at its standard URL)
- `http://ipsn.acm.org/2006/WIP/goodwin_1568983444.pdf` – the interesting article (2 pages) regarding using the OntoSensor in conjunction with storing sensor data in DB, etc.; in references as (Goodwin and Russomanno, 2006)

The OntoSensor project at the Memphis University seems dead, since no documents are available on the project website, except the ontology itself. The interesting article was written in the year 2006.

[5] `http://protege.stanford.edu/`

5.3. Owlgres Implementation

Owlgres is a logical choice for the sensor ontology reasoning, because it is open source, provides an API, and is extensible. The implementation used for integrating into Netcarity project is based on Owlgres extended with the multiple table ABox data scheme (not the QuOnto version), because of two reasons:

- The testing proved it is the most efficient scheme for most cases (combinations of datasets and queries used), see Section 4.4
- As opposed to the single tabel model, it allows to have different database data representation of values of different data property roles, i.e. each data property role assertions table can have different type of its `value` column (e.g. types of double (floating point numeric), text, binary array, timestamp, etc.); the single table model has to have a single datatype of the common `value` column of all data property role assertions (typically of type text)

For the base of multiple table scheme implementation, see Section 3.1.2. Now follows the description of two main features added to the system: support for data property role ranges and the mapping of TBox entities to their corresponding assertion tables.

In order to store measured observations from sensors efficiently, it is necessary to support data property ranges. For example, it is inappropriate to store scalar values obtained from a thermometer (temperature values) and image data acquired by a camer in the same table. Therefore the sensor ontology should contain data property range axioms for the relevant data property roles, e.g. binary array type as data range for the data property role `hasDataImage` (associating an individual representing an image taken by a camera to its actual picture data), or double precision number type for the data property role `hasScalarValue` (used e.g. for associating a single numeric value to a thermometer or single axis accelerometer observations).

When there is such data property range axiom in the TBox, not only the table corresponding to the entity occurring in the axiom has to have an appropriate value column (having an SQL data type according to the type specified in the axiom), but also it has to be stored somewhere permanently, where from it could be retrieved easily. This is required because of the query builder class, which needs to know data role datatypes, for formulating an SQL query, and for exporting the output from an SQL result. The easiest way was used – to store the data property ranges into the table `tbox_name`, where all

TBox entities appear. Thus the `rangedatatype` column was added, containing a string applicable only to data property roles, stating data type their ranges.

The other feature added is the mapping of entities to their assertion tables. In the original multiple table scheme, as described in Section 3.1.2, every ABox assertion table was named with a prefix (representing what type is the entity of – concept, object/data property, or annotation), followed by a number, which was equal to the corresponding entity ID (as written in `tbox_name` table). This is not practical, first because it is unreadable (lot of tables with similar names, differing only in a number in it), and mainly because it is hard to interconnect such mechanism with existing tables already filled in with data.

Therefore, a mapping had to be developed. In order to store the mapping information (which entity is mapped to what table), the `tbox_name` table was chosen again. Another column, called `aboxmapping`, was added. It defines the table name used for assertions for each TBox entity. If no value is given, the system automatically uses the default name as before, consisting of a prefix and the ID number.

There needs to be a way, how to formulate such mapping information in sensor ontology TBox. There is no designated axiom for it defined in standard OWL languages (to which the implementation sticks, since it uses a parser for OWL, the OWLAPI), like in the previous case with data property range axioms. A suitable way is to use an annotation property. For this, a new annotation property `aboxMapping` was assigned. It can be used in a TBox for an entity, defining in what table are its assertions. For an example see Listing 5.1, line 4, which defines that all assertions of the object property role `hasObservation` are in a table called `observations`.

Listing 5.1: An RDF/XML snippet, defining an ABox mapping for an object property

```
1 <owl:AnnotationProperty rdf:about="#aboxMapping"/>
2
3 <owl:ObjectProperty rdf:about="#hasObservation">
4     <aboxMapping>observations</aboxMapping>
5     <rdfs:range rdf:resource="#Observation"/>
6     <rdfs:domain rdf:resource="#Sensor"/>
7 </owl:ObjectProperty>
```

To continue the example, a query asking all observations taking part in the `hasObservation` role is in Listing 5.2, with the resulting SQL in Listing 5.3. Note the second line in the SQL listing, where the table name is `observations`, as defined in the annotation above.

Listing 5.2: An example query asking all observations from a sensor ontology

```
SELECT ?s ?o
WHERE {
  ?s s:hasObservation ?o .
}
```

Listing 5.3: The generated SQL for an example query on a sensor ontology

```
SELECT ora_0.a AS x1, ora_0.b AS x2
FROM observations ora_0
```

The suitable ontology (OntoSensor) found during the the research in Section 5.2 was not used for testing the implementation; instead a simple ontology was developed as part of this thesis. Its concept hierarchy is in Fig. 5.1 (as displayed in Protégé[6]). It has the object role `hasObservation` with the domain `Sensor`, and the range `Observation`. It contains the following data roles (with their domains and ranges noted):

hasImageData domain: Observation, range: binarydata

hasScalarValue domain: Observation, range: double

hasTime domain: Observation, range: dateTime

hasWaveformData domain: Observation, range: binarydata,

where the binarydata datatype is a custom datatype for representing binary data (of e.g. an image or a waveform).

The datatypes (as data role ranges) used in an ontology used for the Netcarity Owlgres extension has to be translated into SQL datatypes by the extension – they are used for the value columns of data role assertion tables. Currently, only the following datatypes are supported: binarydata (a custom datatype), double, and dateTime (standard XSD[7] datatypes).

The *DL−Lite* language allows the sensor ontology for example to use the class hierarchies and role restrictions (e.g. role domains, role ranges).

[6] `http://protege.stanford.edu/`
[7] `http://www.w3.org/TR/xmlschema11-2/`

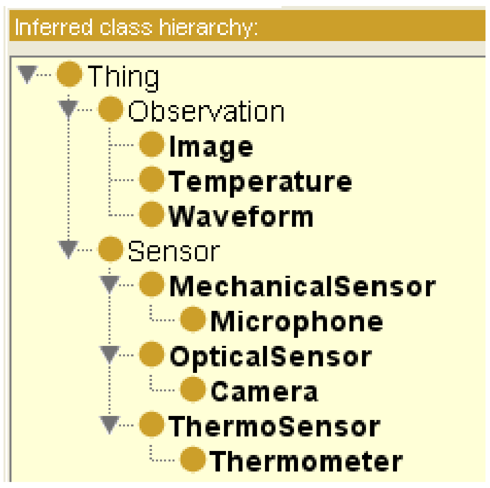

Figure 5.1.: Concept hierarchy of a simple ontology for Netcarity implementation testing

6. Conclusion

As part of this thesis, the original Owlgres relational scheme (single table model) was extended by two other schemes: the multiple table model, and the multiple table QuOnto style model. It is planned to include the models implemented within this thesis into the Owlgres distribution.

A few QuOnto bugs allowed us to see its database relational scheme, as well as the SQL queries generated from the queries posed to the system, as described in Section 3.2. It uses a scheme similar to the multiple table scheme without the individual names table Since QuOnto is unable to parse standard RDF/XML format (does not use OWLAPI), and reads only functional syntax of limited notation, a utility was implemented for converting RDF/XML files into the necessarily simplified functional syntax.

One of the implemented extensions (the multiple table scheme) was further extended, in order to be used for a real-life application within the NetCarity project (described in Section 5.1). First, some research was performed to find a suitable ontology for sensors – their properties, relations between them, and for data measured by them. The sensor information and data representation is one of the tasks NetCarity has to solve. During the research described in Section 5.2, one of a few ontologies found was selected as the best, though still needing some extension. The implemented Owlgres extension includes the support for data role ranges (reflecting in the data type of the value column of the corresponding assertion table), and the support for user definable mapping of TBox entities to their corresponding assertion tables, which is described in Section 5.3.

For testing purposes two datasets, UOB and DBpedia, were used. Two queries per dataset were developed. Section 4.2 compares the three different database schemes implemented in Owlgres, tested on two platforms (Windows, Linux), showing the results in charts. Section 4.3 uncovers the tests performed on QuOnto (only for its original database scheme, only on Windows). The test results are summarized in Section 4.4, with the key observations: QuOnto showed only poor results (may be because of not very strong QuOnto query optimization, or poor H2 database performance), with one exception: for a certain test it did much better then for its other tests, while for Owlgres that certain test was slower than most of its other tests, probably meaning that QuOnto has good optimization for this certain query (it is the UOB query 2). The Owlgres system showed good overall performance, with the winner of *multiple table scheme* (not QuOnto style) as being the *fastest* for most of the tests. An interesting thing is that a query did not finish at all for the single table scheme; because of this and also the other tests it ended up as the worst. In our future work we would like to perform in-depth analysis of ROWLKit.

As a part of the future work, the first version of $DL\text{−}Lite$ integration into Netcarity presented in Section 5.3 will be extended and tested on real data.

Bibliography

Baader, F., Calvanese, D., McGuinness, D. L., Patel-Schneider, P. and Nardi, D. (2003), *The description logic handbook: theory, implementation, and applications*, Cambridge University Press. ISBN 0-521-78176-0.

Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A. and Rosati, R. (2006), Linking data to ontologies: The description logic dl-litea, *in* 'Proc. of the 2nd Int. Workshop on OWL: Experiences and Directions (OWLED 2006)', Vol. 216 of *CEUR Electronic Workshop Proceedings, http://ceur-ws.org/.*

Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M. and Rosati, R. (2005), Dl-lite: Tractable description logics for ontologies, *in* 'Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005)', pp. 602–607.

Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M. and Rosati, R. (2007), 'Tractable reasoning and efficient query answering in description logics: The dl-lite family', *J. of Automated Reasoning* **39**(3), 385–429.

Corona, C., Pasquale, E. D., Poggi, A., Ruzzi, M. and Savo, D. F. (2008), When dl-lite met owl...., *in* Dolbear et al. (2009).

Dolbear, C., Ruttenberg, A. and Sattler, U., eds (2009), *Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC-2008), Karlsruhe, Germany, October 26-27, 2008*, Vol. 432 of *CEUR Workshop Proceedings*, CEUR-WS.org.

Goodwin, C. and Russomanno, D. J. (2006), An ontology-based sensor network prototype environment, *in* 'Proceedings of the 5th International Conference on Information Processing in Sensor Networks', Nashville TN, USA (IEEE), pp. 1–2.

Gruber, T. R. (1993), 'A translation approach to portable ontology specifications', *Knowledge Systems Laboratory, Technical Report, Compurter Science Department, Stanford University, California* **KSL 92-71**, 199–220.

Obitko, M. (2007), *Introduction to Ontologies and Semantic Web*, Marek Obitko, http://www.obitko.com/tutorials/ontologies-semantic-web/. accessed on 30.4.2009.

Pasquale, E. D. (2008), *Query Language on Ontologies: SparSQL*, http://www.dis.uniroma1.it/ degiacom/didattica/semingsoft/materiale/2-ontologies-description-logics/SparSQL-2up.pdf. accessed on 1.12.2008.

Stocker, M. and Smith, M. (2008), Owlgres: A scalable owl reasoner, *in* Dolbear et al. (2009).

List of Figures

A. Contents of the Enclosed CD

A CD is enclosed to this thesis, containing source codes of the implemented extensions, the testing datasets, and this thesis. Its directory structure with brief description of each directory follows.

- **thesis**: A folder containing this thesis in electronic form (a PDF document) including the scanned signed declaration page and the diploma thesis assignment.
- **source_codes**: A folder containing source codes with the following subfolders:
  - **owlgres**: Complete Java source codes of the extensions made to the Owlgres system
  - **owlgres_er_diagrams**: E-R diagrams implemented in Owlgres, designed in MySQL Workbench 5.0
  - **owl_conversion_utilities**: A few Java utilities for converting ontologies in different formats, and tools for data preprocessing for QuOnto
  - **matlab**: Matlab scripts for displaying charts of Owlgres results
  - **CPUloadTimer**: A Java utility monitoring CPU load, recording the times when the CPU load changes
- **test_results**: A folder containing results of Owlgres – the measured times, both in `CSV` format and charts exported from Matlab
- **datasets**: Contains data both for UOB and DBpedia datasets; the source RDF/XML ABox and TBox data, the data converted to functional syntax for QuOnto, and also the test queries are included
- **sensors**: A structured folder containing some of the materials found out during sensor ontology research

B. Generated SQL for the Test Queries

A shortened listing of generated SQL queries for different DB models (Single Table, Multiple Table, Multiple Table QuOnto) for all four testing queries (two for UOB, two for DBpedia) posed to DB by Owlgres and QuOnto, and shortened SQL queries generated by QuOnto for the same four testing queries.

B.1. SQL for UOB Query 1

Listing B.1: Owlgres, Single Table Model SQL

```
SELECT name_0.name AS x1, x2
FROM (
SELECT ora_1.a AS x1, dra_0.value AS x2
FROM  concept_assertion ca_0,  concept_assertion ca_1,  object_role_assertion ora_0,  object_role_assertion ora_1,
     object_role_assertion ora_2,  object_role_assertion ora_3,  data_role_assertion dra_0
WHERE ora_0.object_role=62
AND ca_0.concept=27
AND ora_1.object_role=94
AND ca_0.individual=ora_1.b
AND ora_2.object_role=62
AND ca_0.individual=ora_2.a
AND ora_0.b=ora_2.b
AND dra_0.data_role=96
AND ora_1.a=dra_0.individual
AND ora_3.object_role=91
AND ora_0.a=ora_3.a
AND ca_0.individual=ora_3.b
AND ca_1.concept=22
AND ora_0.a=ca_1.individual
UNION
SELECT ora_1.a AS x1, dra_0.value AS x2
FROM  concept_assertion ca_0,  concept_assertion ca_1,  object_role_assertion ora_0,  object_role_assertion ora_1,
     object_role_assertion ora_2,  object_role_assertion ora_3,  data_role_assertion dra_0
WHERE ca_0.concept=33
AND ora_0.object_role=62
AND ca_0.individual=ora_0.a
AND ora_1.object_role=94
AND ora_2.object_role=62
AND ora_1.b=ora_2.a
AND ora_0.b=ora_2.b
AND ca_1.concept=29
AND ora_1.b=ca_1.individual
AND dra_0.data_role=96
AND ora_1.a=dra_0.individual
AND ora_3.object_role=91
AND ca_0.individual=ora_3.a
AND ora_1.b=ora_3.b
UNION

-- and other 13 similar blocks follow...

) as innerRel , individual_name name_0
WHERE  innerRel.x1=name_0.id
```

Listing B.2: Owlgres, Multiple Table Model SQL

```
SELECT name_0.name AS x1, x2
FROM (
SELECT ora_1.a AS x1, dra_0.value AS x2
FROM objrole_62 ora_0, objrole_94 ora_1, objrole_63 ora_2, objrole_62 ora_3, objrole_72 ora_4, objrole_91 ora_5,
     datarole_96 dra_0
WHERE ora_1.b=ora_2.b
AND ora_1.b=ora_3.a
AND ora_0.b=ora_3.b
AND ora_1.a=dra_0.individual
AND ora_0.a=ora_4.a
AND ora_0.a=ora_5.a
AND ora_1.b=ora_5.b
UNION
SELECT ora_1.a AS x1, dra_0.value AS x2
FROM concept_27 ca_0, objrole_62 ora_0, objrole_94 ora_1, objrole_62 ora_2, objrole_72 ora_3, objrole_91 ora_4,
     datarole_96 dra_0
WHERE ca_0.individual=ora_1.b
AND ca_0.individual=ora_2.a
AND ora_0.b=ora_2.b
AND ora_1.a=dra_0.individual
AND ora_0.a=ora_3.a
AND ora_0.a=ora_4.a
AND ca_0.individual=ora_4.b
UNION

-- and other 13 similar blocks follow...

) as innerRel , individual name_0
WHERE  innerRel.x1=name_0.id
```

Listing B.3: Owlgres, Multiple Table QuOnto Model SQL

```
SELECT ora_1.a AS x1, dra_0.value AS x2
FROM objrole_62 ora_0, objrole_94 ora_1, objrole_63 ora_2, objrole_62 ora_3, objrole_72 ora_4, objrole_91 ora_5,
     datarole_96 dra_0
WHERE ora_1.b=ora_2.b
AND ora_1.b=ora_3.a
AND ora_0.b=ora_3.b
AND ora_1.a=dra_0.individual
AND ora_0.a=ora_4.a
AND ora_0.a=ora_5.a
AND ora_1.b=ora_5.b
UNION
SELECT ora_1.a AS x1, dra_0.value AS x2
FROM concept_27 ca_0, objrole_62 ora_0, objrole_94 ora_1, objrole_62 ora_2, objrole_72 ora_3, objrole_91 ora_4,
     datarole_96 dra_0
WHERE ca_0.individual=ora_1.b
AND ca_0.individual=ora_2.a
AND ora_0.b=ora_2.b
AND ora_1.a=dra_0.individual
AND ora_0.a=ora_3.a
AND ora_0.a=ora_4.a
AND ca_0.individual=ora_4.b
UNION

-- and other 13 similar blocks follow...
```

Listing B.4: QuOnto generated SQL

```
SELECT DISTINCT alias_2.term1 AS p , alias_2.term2 AS pn
FROM httpuobiodtibmcomunivbenchliteowlpublicationAuthor alias_0 , httpuobiodtibmcomunivbenchliteowllike alias_1 ,
     httpuobiodtibmcomunivbenchliteowlname alias_2 , httpuobiodtibmcomunivbenchliteowllike alias_3 ,
     httpuobiodtibmcomunivbenchliteowlisFriendOf alias_4 , httpuobiodtibmcomunivbenchliteowlisAdvisedBy alias_5 ,
     httpuobiodtibmcomunivbenchliteowltakesCourse alias_6
WHERE alias_0.term1=alias_2.term1  AND alias_0.term2=alias_3.term1  AND alias_1.term2=alias_3.term2  AND alias_1.term1
     =alias_4.term1  AND alias_3.term1=alias_4.term2  AND alias_4.term2=alias_5.term2  AND alias_4.term1=alias_6.term1
UNION
SELECT DISTINCT alias_3.term1 AS p , alias_3.term2 AS pn
FROM httpuobiodtibmcomunivbenchliteowlAssistantProfessor alias_0 , httpuobiodtibmcomunivbenchliteowlpublicationAuthor
     alias_1 , httpuobiodtibmcomunivbenchliteowllike alias_2 , httpuobiodtibmcomunivbenchliteowlname alias_3 ,
     httpuobiodtibmcomunivbenchliteowllike alias_4 , httpuobiodtibmcomunivbenchliteowlisFriendOf alias_5 ,
     httpuobiodtibmcomunivbenchliteowltakesCourse alias_6
WHERE alias_0.term=alias_1.term2  AND alias_1.term1=alias_3.term1  AND alias_1.term2=alias_4.term1  AND alias_2.term2=
     alias_4.term2  AND alias_2.term1=alias_5.term1  AND alias_4.term1=alias_5.term2  AND alias_5.term1=alias_6.term1
UNION

-- and other 10 similar blocks follow...
```

B.2. SQL for UOB Query 2

Listing B.5: Owlgres, Single Table Model SQL

```
SELECT name_0.name AS x1, name_1.name AS x2, name_2.name AS x3, name_3.name AS x4, name_4.name AS x5, name_5.name AS
     x6, name_6.name AS x7
FROM (
SELECT ora_1.a AS x1, ora_0.a AS x2, ora_0.b AS x3, ora_4.b AS x4, ora_2.b AS x5, ora_3.b AS x6, ora_1.b AS x7
FROM  object_role_assertion ora_0,  object_role_assertion ora_1,  object_role_assertion ora_2,  object_role_assertion
     ora_3,  object_role_assertion ora_4,  object_role_assertion ora_5,  object_role_assertion ora_6,
     object_role_assertion ora_7,  object_role_assertion ora_8,  object_role_assertion ora_9,  object_role_assertion
     ora_10
WHERE ora_0.object_role=94
AND ora_1.object_role=72
AND ora_2.object_role=77
AND ora_3.object_role=81
AND ora_0.b=ora_3.a
AND ora_4.object_role=63
AND ora_1.a=ora_4.a
AND ora_5.object_role=81
AND ora_0.b=ora_5.a
AND ora_1.b=ora_5.b
AND ora_6.object_role=82
AND ora_0.b=ora_6.a
AND ora_2.a=ora_6.b
AND ora_7.object_role=82
AND ora_1.a=ora_7.a
AND ora_2.a=ora_7.b
AND ora_8.object_role=92
AND ora_2.b=ora_8.b
AND ora_9.object_role=94
AND ora_0.a=ora_9.a
AND ora_1.a=ora_9.b
AND ora_10.object_role=89
AND ora_1.a=ora_10.a
AND ora_3.b=ora_10.b
UNION
SELECT ora_3.a AS x1, ora_4.a AS x2, ora_1.b AS x3, ora_5.b AS x4, ora_0.b AS x5, ora_1.a AS x6, ora_3.b AS x7
FROM  object_role_assertion ora_0,  object_role_assertion ora_1,  object_role_assertion ora_2,  object_role_assertion
     ora_3,  object_role_assertion ora_4,  object_role_assertion ora_5,  object_role_assertion ora_6,
     object_role_assertion ora_7,  object_role_assertion ora_8,  object_role_assertion ora_9,  object_role_assertion
     ora_10
WHERE ora_0.object_role=65
AND ora_1.object_role=66
AND ora_2.object_role=77
AND ora_0.b=ora_2.b
AND ora_3.object_role=72
AND ora_4.object_role=94
AND ora_1.b=ora_4.b
AND ora_5.object_role=63
AND ora_3.a=ora_5.a
AND ora_6.object_role=82
AND ora_1.b=ora_6.a
AND ora_2.a=ora_6.b
AND ora_7.object_role=81
AND ora_1.b=ora_7.a
AND ora_3.b=ora_7.b
AND ora_8.object_role=82
AND ora_3.a=ora_8.a
AND ora_2.a=ora_8.b
AND ora_9.object_role=94
AND ora_4.a=ora_9.a
AND ora_3.a=ora_9.b
AND ora_10.object_role=89
AND ora_3.a=ora_10.a
AND ora_1.a=ora_10.b
UNION

-- and other 14 similar blocks follow...

) as innerRel , individual_name name_0, individual_name name_1, individual_name name_2, individual_name name_3,
     individual_name name_4, individual_name name_6, individual_name name_5
WHERE  innerRel.x1=name_0.id
AND innerRel.x2=name_1.id
AND innerRel.x3=name_2.id
AND innerRel.x4=name_3.id
AND innerRel.x5=name_4.id
AND innerRel.x6=name_5.id
AND innerRel.x7=name_6.id
```

Listing B.6: Owlgres, Multiple Table Model SQL

```
SELECT name_0.name AS x1, name_1.name AS x2, name_2.name AS x3, name_3.name AS x4, name_4.name AS x5, name_5.name AS
     x6, name_6.name AS x7
FROM (
SELECT ora_2.a AS x1, ora_1.a AS x2, ora_0.b AS x3, ora_4.b AS x4, ora_3.b AS x5, ora_0.a AS x6, ora_2.b AS x7
FROM objrole_66 ora_0, objrole_94 ora_1, objrole_72 ora_2, objrole_77 ora_3, objrole_63 ora_4, objrole_82 ora_5,
     objrole_66 ora_6, objrole_82 ora_7, objrole_92 ora_8, objrole_94 ora_9, objrole_89 ora_10
WHERE ora_0.b=ora_1.b
AND ora_2.a=ora_4.a
AND ora_0.b=ora_5.a
AND ora_3.a=ora_5.b
AND ora_2.b=ora_6.a
AND ora_0.b=ora_6.b
AND ora_2.a=ora_7.a
AND ora_3.a=ora_7.b
AND ora_3.b=ora_8.b
AND ora_1.a=ora_9.a
AND ora_2.a=ora_9.b
AND ora_2.a=ora_10.a
AND ora_0.a=ora_10.b
UNION
SELECT ora_1.a AS x1, ora_2.a AS x2, ora_2.b AS x3, ora_4.b AS x4, ca_0.individual AS x5, ora_3.b AS x6, ora_1.b AS x7
FROM concept_45 ca_0, objrole_77 ora_0, objrole_72 ora_1, objrole_94 ora_2, objrole_81 ora_3, objrole_63 ora_4,
     objrole_82 ora_5, objrole_81 ora_6, objrole_82 ora_7, objrole_94 ora_8, objrole_89 ora_9
WHERE ca_0.individual=ora_0.b
AND ora_2.b=ora_3.a
AND ora_1.a=ora_4.a
AND ora_2.b=ora_5.a
AND ora_0.a=ora_5.b
AND ora_2.b=ora_6.a
AND ora_1.b=ora_6.b
AND ora_1.a=ora_7.a
AND ora_0.a=ora_7.b
AND ora_2.a=ora_8.a
AND ora_1.a=ora_8.b
AND ora_1.a=ora_9.a
AND ora_3.b=ora_9.b
UNION

-- and other 14 similar blocks follow...

) as innerRel , individual name_0, individual name_1, individual name_2, individual name_3, individual name_4,
     individual name_6, individual name_5
WHERE  innerRel.x1=name_0.id
AND innerRel.x2=name_1.id
AND innerRel.x3=name_2.id
AND innerRel.x4=name_3.id
AND innerRel.x5=name_4.id
AND innerRel.x6=name_5.id
AND innerRel.x7=name_6.id
```

Listing B.7: Owlgres, Multiple Table QuOnto Model SQL

```
SELECT ora_2.a AS x1, ora_1.a AS x2, ora_0.b AS x3, ora_4.b AS x4, ora_3.b AS x5, ora_0.a AS x6, ora_2.b AS x7
FROM objrole_66 ora_0, objrole_94 ora_1, objrole_72 ora_2, objrole_77 ora_3, objrole_63 ora_4, objrole_82 ora_5,
     objrole_66 ora_6, objrole_82 ora_7, objrole_92 ora_8, objrole_94 ora_9, objrole_89 ora_10
WHERE ora_0.b=ora_1.b
AND ora_2.a=ora_4.a
AND ora_0.b=ora_5.a
AND ora_3.a=ora_5.b
AND ora_2.b=ora_6.a
AND ora_0.b=ora_6.b
AND ora_2.a=ora_7.a
AND ora_3.a=ora_7.b
AND ora_3.b=ora_8.b
AND ora_1.a=ora_9.a
AND ora_2.a=ora_9.b
AND ora_2.a=ora_10.a
AND ora_0.a=ora_10.b
UNION
SELECT ora_1.a AS x1, ora_2.a AS x2, ora_2.b AS x3, ora_4.b AS x4, ca_0.individual AS x5, ora_3.b AS x6, ora_1.b AS x7
FROM concept_45 ca_0, objrole_77 ora_0, objrole_72 ora_1, objrole_94 ora_2, objrole_81 ora_3, objrole_63 ora_4,
     objrole_82 ora_5, objrole_81 ora_6, objrole_82 ora_7, objrole_94 ora_8, objrole_89 ora_9
WHERE ca_0.individual=ora_0.b
AND ora_2.b=ora_3.a
AND ora_1.a=ora_4.a
AND ora_2.b=ora_5.a
AND ora_0.a=ora_5.b
AND ora_2.b=ora_6.a
AND ora_1.b=ora_6.b
AND ora_1.a=ora_7.a
AND ora_0.a=ora_7.b
AND ora_2.a=ora_8.a
AND ora_1.a=ora_8.b
AND ora_1.a=ora_9.a
AND ora_3.b=ora_9.b
UNION

-- and other 14 similar blocks follow...
```

Listing B.8: QuOnto generated SQL

```
SELECT DISTINCT alias_10.term2 AS L , alias_9.term2 AS E , alias_8.term2 AS A , alias_9.term1 AS Q , alias_8.term1 AS
     Z , alias_5.term2 AS Z2 , alias_0.term2 AS B
FROM httpuobiodtibmcomunivbenchliteowlisAdvisedBy alias_0 , httpuobiodtibmcomunivbenchliteowlisMemberOf alias_1 ,
     httpuobiodtibmcomunivbenchliteowlsubOrganizationOf alias_2 , httpuobiodtibmcomunivbenchliteowlpublicationAuthor
     alias_3 , httpuobiodtibmcomunivbenchliteowlteachingAssistantOf alias_4 ,
     httpuobiodtibmcomunivbenchliteowlteacherOf alias_5 , httpuobiodtibmcomunivbenchliteowlisMemberOf alias_6 ,
     httpuobiodtibmcomunivbenchliteowltakesCourse alias_7 , httpuobiodtibmcomunivbenchliteowlisTaughtBy alias_8 ,
     httpuobiodtibmcomunivbenchliteowlpublicationAuthor alias_9 ,
     httpuobiodtibmcomunivbenchliteowlhasDoctoralDegreeFrom alias_10
```

```
WHERE alias_1.term2=alias_2.term1  AND alias_1.term1=alias_3.term2  AND alias_0.term1=alias_4.term1  AND alias_3.term2
     =alias_5.term1  AND alias_4.term2=alias_5.term2  AND alias_4.term1=alias_6.term1  AND alias_2.term1=alias_6.term2
       AND alias_6.term1=alias_7.term1  AND alias_7.term2=alias_8.term1  AND alias_5.term1=alias_8.term2  AND alias_3.
     term1=alias_9.term1  AND alias_7.term1=alias_9.term2  AND alias_2.term2=alias_10.term2
UNION
SELECT DISTINCT alias_3.term2 AS L , alias_10.term2 AS E , alias_9.term2 AS A , alias_10.term1 AS Q , alias_8.term1 AS
      Z , alias_9.term1 AS Z2 , alias_0.term2 AS B
FROM httpuobiodtibmcomunivbenchliteowlisAdvisedBy alias_0 , httpuobiodtibmcomunivbenchliteowlisMemberOf alias_1 ,
     httpuobiodtibmcomunivbenchliteowlsubOrganizationOf alias_2 , httpuobiodtibmcomunivbenchliteowlhasMasterDegreeFrom
      alias_3 , httpuobiodtibmcomunivbenchliteowlpublicationAuthor alias_4 ,
     httpuobiodtibmcomunivbenchliteowlteachingAssistantOf alias_5 , httpuobiodtibmcomunivbenchliteowlisMemberOf
     alias_6 , httpuobiodtibmcomunivbenchliteowltakesCourse alias_7 , httpuobiodtibmcomunivbenchliteowlisTaughtBy
     alias_8 , httpuobiodtibmcomunivbenchliteowlisTaughtBy alias_9 ,
     httpuobiodtibmcomunivbenchliteowlpublicationAuthor alias_10
WHERE alias_1.term2=alias_2.term1  AND alias_2.term2=alias_3.term2  AND alias_1.term1=alias_4.term2  AND alias_0.term1
     =alias_5.term1  AND alias_5.term1=alias_6.term1  AND alias_2.term1=alias_6.term2  AND alias_6.term1=alias_7.term1
       AND alias_7.term2=alias_8.term1  AND alias_4.term2=alias_8.term2  AND alias_5.term2=alias_9.term1  AND alias_8.
     term2=alias_9.term2  AND alias_4.term1=alias_10.term1  AND alias_7.term1=alias_10.term2
UNION

-- and other 10 similar blocks follow...
```

B.3. SQL for DBP Query 1

Listing B.9: Owlgres, Single Table Model SQL

```
SELECT DISTINCT name_0.name AS x1, name_1.name AS x2
FROM (
SELECT DISTINCT ora_0.b AS x1, dra_1.individual AS x2
FROM  object_role_assertion ora_0,  object_role_assertion ora_1,  object_role_assertion ora_2,  object_role_assertion
     ora_3,  object_role_assertion ora_4,  object_role_assertion ora_5,  object_role_assertion ora_6,
     data_role_assertion dra_0,  data_role_assertion dra_1,  data_role_assertion dra_2,  data_role_assertion dra_3
WHERE ora_0.object_role=463
AND dra_0.data_role=900
AND dra_1.data_role=900
AND ora_1.object_role=483
AND dra_1.individual=ora_1.b
AND ora_2.object_role=668
AND ora_0.a=ora_2.a
AND ora_3.object_role=668
AND ora_1.a=ora_3.a
AND ora_4.object_role=467
AND ora_0.b=ora_4.b
AND dra_2.data_role=779
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_5.object_role=668
AND ora_2.b=ora_5.b
AND ora_6.object_role=668
AND ora_5.a=ora_6.a
AND ora_3.b=ora_6.b
AND dra_3.data_role=779
AND dra_1.value=dra_3.value
) as innerRel , individual_name name_0, individual_name name_1
WHERE  innerRel.x1=name_0.id
AND innerRel.x2=name_1.id
```

Listing B.10: Owlgres, Multiple Table Model SQL

```
SELECT DISTINCT name_0.name AS x1, name_1.name AS x2
FROM (
SELECT DISTINCT ora_0.b AS x1, dra_1.individual AS x2
FROM objrole_463 ora_0, objrole_483 ora_1, objrole_668 ora_2, objrole_668 ora_3, objrole_467 ora_4, objrole_668 ora_5,
      objrole_668 ora_6, datarole_900 dra_0, datarole_900 dra_1, datarole_779 dra_2, datarole_779 dra_3
WHERE dra_1.individual=ora_1.b
AND ora_0.a=ora_2.a
AND ora_1.a=ora_3.a
AND ora_0.b=ora_4.b
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_2.b=ora_5.b
AND ora_5.a=ora_6.a
AND ora_3.b=ora_6.b
AND dra_1.value=dra_3.value
) as innerRel , individual name_0, individual name_1
WHERE  innerRel.x1=name_0.id
AND innerRel.x2=name_1.id
```

Listing B.11: Owlgres, Multiple Table QuOnto Model SQL

```
SELECT DISTINCT ora_0.b AS x1, dra_1.individual AS x2
FROM objrole_463 ora_0, objrole_483 ora_1, objrole_668 ora_2, objrole_668 ora_3, objrole_467 ora_4, objrole_668 ora_5,
      objrole_668 ora_6, datarole_900 dra_0, datarole_900 dra_1, datarole_779 dra_2, datarole_779 dra_3
WHERE dra_1.individual=ora_1.b
AND ora_0.a=ora_2.a
AND ora_1.a=ora_3.a
AND ora_0.b=ora_4.b
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_2.b=ora_5.b
AND ora_5.a=ora_6.a
AND ora_3.b=ora_6.b
AND dra_1.value=dra_3.value
```

Listing B.12: QuOnto generated SQL

```
SELECT DISTINCT alias_8.term2 AS a1 , alias_10.term2 AS a2
FROM httpdbpediaorgontologygenre alias_0 , httpdbpediaorgontologygenre alias_1 , httpxmlnscomfoaf01givenname alias_2 ,
      httpdbpediaorgontologygenre alias_3 , httpxmlnscomfoaf01givenname alias_4 , httpdbpediaorgpropertydisambiguates
     alias_5 , httpxmlnscomfoaf01surname alias_6 , httpdbpediaorgontologygenre alias_7 , httpdbpediaorgontologyartist
     alias_8 , httpxmlnscomfoaf01surname alias_9 , httpdbpediaorgontologyassociatedMusicalArtist alias_10
WHERE alias_0.term2=alias_1.term2  AND alias_1.term1=alias_3.term1  AND alias_2.term1=alias_5.term2  AND alias_2.term2
     =alias_6.term2  AND alias_3.term2=alias_7.term2  AND alias_7.term1=alias_8.term1  AND alias_5.term2=alias_8.term2
       AND alias_4.term2=alias_9.term2  AND alias_0.term1=alias_10.term1  AND alias_9.term1=alias_10.term2
```

B.4. SQL for DBP Query 2

Listing B.13: Owlgres, Single Table Model SQL

```
SELECT DISTINCT name_0.name AS x1, name_1.name AS x2
FROM (
SELECT DISTINCT ora_0.b AS x1, ora_1.b AS x2
FROM  concept_assertion ca_0,  object_role_assertion ora_0,  object_role_assertion ora_1,  object_role_assertion ora_2
     ,  object_role_assertion ora_3,  object_role_assertion ora_4,  object_role_assertion ora_5,
     object_role_assertion ora_6,  data_role_assertion dra_0,  data_role_assertion dra_1,  data_role_assertion dra_2,
      data_role_assertion dra_3
WHERE ora_0.object_role=463
AND dra_0.data_role=900
AND ora_1.object_role=483
AND dra_1.data_role=900
AND ora_1.b=dra_1.individual
AND ora_2.object_role=668
AND ora_0.a=ora_2.a
AND ca_0.concept=251
AND ora_0.b=ca_0.individual
AND ora_3.object_role=668
AND ora_1.a=ora_3.a
AND ora_4.object_role=467
AND ora_0.b=ora_4.b
AND dra_2.data_role=779
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_5.object_role=668
AND ora_2.b=ora_5.b
AND ora_6.object_role=668
AND ora_5.a=ora_6.a
AND ora_3.b=ora_6.b
AND dra_3.data_role=779
AND dra_1.value=dra_3.value
UNION
SELECT DISTINCT ora_0.b AS x1, ora_1.b AS x2
FROM  concept_assertion ca_0,  object_role_assertion ora_0,  object_role_assertion ora_1,  object_role_assertion ora_2
     ,  object_role_assertion ora_3,  object_role_assertion ora_4,  object_role_assertion ora_5,
     object_role_assertion ora_6,  data_role_assertion dra_0,  data_role_assertion dra_1,  data_role_assertion dra_2,
      data_role_assertion dra_3
WHERE ora_0.object_role=463
AND dra_0.data_role=900
AND ora_1.object_role=483
AND dra_1.data_role=900
AND ora_1.b=dra_1.individual
AND ora_2.object_role=668
AND ora_0.a=ora_2.a
AND ora_3.object_role=668
AND ora_1.a=ora_3.a
AND ora_4.object_role=467
AND ora_0.b=ora_4.b
AND dra_2.data_role=779
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_5.object_role=668
AND ora_2.b=ora_5.b
AND ora_6.object_role=668
AND ora_5.a=ora_6.a
AND ora_3.b=ora_6.b
AND ca_0.concept=245
AND ora_0.b=ca_0.individual
AND dra_3.data_role=779
AND dra_1.value=dra_3.value
UNION

-- and other 17 similar blocks follow...

) as innerRel , individual_name name_0, individual_name name_1
WHERE  innerRel.x1=name_0.id
AND innerRel.x2=name_1.id
```

Listing B.14: Owlgres, Multiple Table Model SQL

```
SELECT DISTINCT name_0.name AS x1, name_1.name AS x2
FROM (
SELECT DISTINCT ora_0.b AS x1, ora_1.b AS x2
FROM objrole_463 ora_0, objrole_483 ora_1, objrole_668 ora_2, objrole_668 ora_3, objrole_570 ora_4, objrole_467 ora_5,
      objrole_668 ora_6, objrole_668 ora_7, datarole_900 dra_0, datarole_900 dra_1, datarole_779 dra_2, datarole_779
     dra_3
WHERE ora_1.b=dra_1.individual
AND ora_0.a=ora_2.a
AND ora_1.a=ora_3.a
AND ora_0.b=ora_4.a
AND ora_0.b=ora_5.b
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_2.b=ora_6.b
AND ora_6.a=ora_7.a
AND ora_3.b=ora_7.b
AND dra_1.value=dra_3.value
UNION
SELECT DISTINCT ora_0.b AS x1, dra_1.individual AS x2
FROM objrole_463 ora_0, objrole_415 ora_1, objrole_483 ora_2, objrole_668 ora_3, objrole_668 ora_4, objrole_467 ora_5,
      objrole_668 ora_6, objrole_668 ora_7, datarole_900 dra_0, datarole_900 dra_1, datarole_779 dra_2, datarole_779
     dra_3
WHERE ora_0.b=ora_1.a
AND dra_1.individual=ora_2.b
AND ora_0.a=ora_3.a
AND ora_2.a=ora_4.a
AND ora_0.b=ora_5.b
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_3.b=ora_6.b
AND ora_6.a=ora_7.a
AND ora_4.b=ora_7.b
AND dra_1.value=dra_3.value
UNION

-- and other 17 similar blocks follow...

) as innerRel , individual name_0, individual name_1
WHERE  innerRel.x1=name_0.id
AND innerRel.x2=name_1.id
```

Listing B.15: Owlgres, Multiple Table QuOnto Model SQL

```
SELECT DISTINCT ora_0.b AS x1, ora_1.b AS x2
FROM objrole_463 ora_0, objrole_483 ora_1, objrole_668 ora_2, objrole_668 ora_3, objrole_570 ora_4, objrole_467 ora_5,
      objrole_668 ora_6, objrole_668 ora_7, datarole_900 dra_0, datarole_900 dra_1, datarole_779 dra_2, datarole_779
     dra_3
WHERE ora_1.b=dra_1.individual
AND ora_0.a=ora_2.a
AND ora_1.a=ora_3.a
AND ora_0.b=ora_4.a
AND ora_0.b=ora_5.b
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_2.b=ora_6.b
AND ora_6.a=ora_7.a
AND ora_3.b=ora_7.b
AND dra_1.value=dra_3.value
UNION
SELECT DISTINCT ora_0.b AS x1, dra_1.individual AS x2
FROM objrole_463 ora_0, objrole_415 ora_1, objrole_483 ora_2, objrole_668 ora_3, objrole_668 ora_4, objrole_467 ora_5,
      objrole_668 ora_6, objrole_668 ora_7, datarole_900 dra_0, datarole_900 dra_1, datarole_779 dra_2, datarole_779
     dra_3
WHERE ora_0.b=ora_1.a
AND dra_1.individual=ora_2.b
AND ora_0.a=ora_3.a
AND ora_2.a=ora_4.a
AND ora_0.b=ora_5.b
AND ora_0.b=dra_2.individual
AND dra_0.value=dra_2.value
AND ora_3.b=ora_6.b
AND ora_6.a=ora_7.a
AND ora_4.b=ora_7.b
AND dra_1.value=dra_3.value
UNION

-- and other 17 similar blocks follow...
```

Listing B.16: QuOnto generated SQL

```
SELECT DISTINCT alias_9.term2 AS a1 , alias_11.term2 AS a2
```

```
FROM httpdbpediaorgontologygenre alias_0 , httpdbpediaorgontologygenre alias_1 , httpdbpediaorgontologyassociatedBand
     alias_2 , httpxmlnscomfoaf01givenname alias_3 , httpxmlnscomfoaf01givenname alias_4 , httpdbpediaorgontologygenre
      alias_5 , httpxmlnscomfoaf01surname alias_6 , httpdbpediaorgpropertydisambiguates alias_7 ,
     httpdbpediaorgontologygenre alias_8 , httpdbpediaorgontologyartist alias_9 , httpxmlnscomfoaf01surname alias_10 ,
      httpdbpediaorgontologyassociatedMusicalArtist alias_11
WHERE alias_0.term2=alias_1.term2  AND alias_2.term1=alias_3.term1  AND alias_1.term1=alias_5.term1  AND alias_3.term2
     =alias_6.term2  AND alias_3.term1=alias_7.term2  AND alias_5.term2=alias_8.term2  AND alias_8.term1=alias_9.term1
       AND alias_7.term2=alias_9.term2  AND alias_4.term2=alias_10.term2  AND alias_0.term1=alias_11.term1  AND
     alias_10.term1=alias_11.term2
UNION
SELECT DISTINCT alias_9.term2 AS a1 , alias_11.term2 AS a2
FROM httpdbpediaorgontologyassociatedMusicalArtist alias_0 , httpdbpediaorgontologygenre alias_1 ,
     httpdbpediaorgontologygenre alias_2 , httpxmlnscomfoaf01givenname alias_3 , httpxmlnscomfoaf01givenname alias_4 ,
      httpdbpediaorgontologygenre alias_5 , httpxmlnscomfoaf01surname alias_6 , httpdbpediaorgpropertydisambiguates
     alias_7 , httpdbpediaorgontologygenre alias_8 , httpdbpediaorgontologyartist alias_9 , httpxmlnscomfoaf01surname
     alias_10 , httpdbpediaorgontologyassociatedMusicalArtist alias_11
WHERE alias_1.term2=alias_2.term2  AND alias_0.term2=alias_3.term1  AND alias_2.term1=alias_5.term1  AND alias_3.term2
     =alias_6.term2  AND alias_3.term1=alias_7.term2  AND alias_5.term2=alias_8.term2  AND alias_8.term1=alias_9.term1
       AND alias_7.term2=alias_9.term2  AND alias_4.term2=alias_10.term2  AND alias_1.term1=alias_11.term1  AND
     alias_10.term1=alias_11.term2
UNION

-- and other 10 similar blocks follow...
```