FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation of Description Logic Programs using an RDBMS

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Patrik Schneider, BSc.
Matrikelnummer 0627383

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: O.Univ.Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Mitwirkung: Dipl.-Ing. Thomas Krennwallner

Wien, 30.12.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Patrik Schneider
Postfach 668
9490 Vaduz
Liechtenstein

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30.12.2010

(Unterschrift Verfasser)

Abstract

We propose a novel approach to evaluate description logic programs (dl-programs) using a Relational Database Management System (RDBMS). Such dl-programs are a state-of-the-art Semantic Web formalism of combining rules and ontologies, whereby rules are expressed as logic programs and ontologies are expressed in Description Logics (DL). In dl-programs a modular concept of plug-ins was introduced, which allows to combine dl-programs with different DL reasoner. Grounding in dl-programs is still considered a performance bottleneck, caused by having exponentially many rules to process. But for the success of the Semantic Web technologies, it is crucial to efficiently process vast amounts of data. The goal of this work is to show, that dl-programs can be efficiently evaluated by using RDBMSs. For this purpose we report on a prototype implementation, where SQL is generated by an existing DL-Lite reasoner, which is incorporated into a Datalog rewriter. For testing the prototype, we develop a benchmark suite with pure Datalog and Datalog/DL tests. Based on the benchmark suite, we produce experimental results. These results are used to compare the prototype with the reasoners of the DLV family.

Zusammenfassung

Wir entwickeln einen neuen Ansatz, bei dem Description Logic Programs (dl-programs) auf einem relationalen Datenbank System (RDBMS) evaluiert werden. Die dl-programs sind ein vielversprechender Semantik Web Formalismus um Regeln und Ontologien zu kombinieren. Dabei werden die Regeln als Logische Programme und die Ontologien in Beschreibungslogiken (DL) ausgedrückt. Bei der bottom-up Auswertung eines dl-programs gibt es aber immer noch Geschwindigkeitseinbussen, da exponentiell viele Regeln im Verhältnis zur Grösse des Programms entstehen können. Ein wichtiger Faktor für die Akzeptanz von Semantik Web Technologien besteht aber darin, dass grosse Datenmengen effizient verarbeitet werden können. Das Ziel dieser Arbeit ist nun zu zeigen, dass dl-programs mit Hilfe von RDBMS effizient verarbeitet werden können. Um dieses Ziel zu erreichen, wurde ein Prototyp entwickelt, bei dem mit Hilfe eines DL-Lite und eines Datalog Systems, SQL generiert wird. Um die Effizienz des Prototyps zu zeigen, werden Benchmarks entwickelt, welche aus Datalog und Datalog/DL Tests bestehen. Aufgrund dieser Benchmarks wird dann der Prototyp an den Systemen der DLV-Familie gemessen.

Acknowledgements

First of all, I owe my sincerest gratitude to my supervisor, Thomas Eiter, who gave me the opportunity to write my thesis, supported me throughout with his patience and knowledge, while giving me enough space to work in my own pace. He gave me great insight into the field of Knowledge Representation and Reasoning as well as into scientific research as a whole. Moreover, I have learned good practise for scientific writing. Particularly, I enjoyed the session about princesses, dragons, and heroes.

I would also like to thank Thomas Krennwallner, who provided me with many important hints, remarkably, when I exactly needed them. Moreover, I am indebted to the staff of KBS, for supporting me whenever needed.

Lastly, I am truly grateful to my family providing me with support and a loving environment. Writing a thesis is often a long journey with many obstacles, but it is definitely a valuable experience, especially with great backing.

Contents

1. Introduction

The Semantic Web, anticipated by Tim Berners-Lee in 2001, is by 2010 becoming an important topic in the *World Wide Web* (WWW) and the information system community. This is observable through mature standards like the *Resource Description Framework*[1] (RDF) and the *Web Ontology Language*[1] (OWL), but also through evolving projects like *Ontorule*, *DBpedia*, or *SIOC*.[2] In their Scientific American article Berners-Lee et al. capture the Semantic Web as follows [Berners-Lee et al., 2001]: *Knowledge representation* (KR), ontologies and agents are central to the Semantic Web and should lead to an "evolution of knowledge". KR is crucial, so the WWW can be better understood by computer systems and humans. Ontologies, expressed in *Description Logics* (DL), are a central part of the "understanding", adding taxonomies and inference rules to the information of a web page or a document. Software agent will collect and process information using these ontologies. Taking the idea further, an agent could "bootstrap" new reasoning capabilities when discovering new ontologies. Several agents can be linked together creating a "value chain" of information processing, whereby every agent is "adding value" to parts of the information product.

This thesis is inspired by the idea of "adding value" to the information process by extending ontology based inference with *Logic Programming* (LP) based inference rules. In recent years, combining rules and ontologies become an important focus of Semantic Web research. The W3C founded the *Rule Interchange Format* (RIF) working group, which aim is to create a standard for exchanging rules among rule based systems.[3] Extending the idea of exchanging rules, a W3C working draft was written concerning "RIF RDF and OWL Compatibility", which considers the import of RDF and OWL in RIF.[4] The RIF working group also specified different rule languages, which led to a *Core*, a *Basic Logic Dialect* (BLD), and a *Production Rules Dialect* (PRD) dialect [Kifer, 2008]. Related to the RIF and OWL compatibility, we focus on the strain of research, where rules are expressed as logic programs and a loose coupling of rules and ontologies is considered. As the starting point of this thesis, we take the state-of-the-art approach of *description logic programs* (dl-programs) for loose coupling. Dl-programs were introduced by several papers of Eiter et al., describing dl-programs under the *answer set semantics* [Eiter et al., 2004] and under the *well-founded semantics* [Eiter et al., 2009b]. The papers mentioned show that dl-programs regarding to their "advanced" expressive power are still decidable. Furthermore, a modular concept of plug-ins was introduced, which allows to combine

[1] http://www.w3.org/RDF/ and http://www.w3.org/2007/OWL/wiki/OWL_Working_Group

[2] http://ontorule-project.eu/, http://dbpedia.org/, and http://sioc-project.org/

[3] http://www.w3.org/2005/rules/

[4] http://www.w3.org/TR/rif-rdf-owl/

dl-programs with different DL reasoners. The concept of plug-ins was generalized to *HEX-programs* and lead to the successful development of the reasoner *dlvhex*[5] [Eiter et al., 2006].

Another inspiring idea is related to the size of projects like DBpedia. We observe that a vast amount of data is collected and linked to ontologies. For the acceptance of Semantic Web applications it is crucial, that these data is processed efficiently. Efficient data processing was a main reason for the advent of relational *database* (DB) technology, so we will use this technology as the foundation of our inference system. *DL-Lite*, which was introduced by Calvanese et al., builds the link needed between DL-based inference and a *Relational Database Management System* (RDBMS) [Calvanese et al., 2007].

DLV[6] or dlvhex, belonging to the group of *Answer Set* (AS) solvers, which first ground a logic program before the solutions are computed. Even with highly efficient grounding algorithms, we might encounter a *grounding bottleneck*, caused by having exponentially many rules to process [Eiter et al., 2007, 2009a]. If we reason with a large *Knowledge base* (KB), we definitely need a more efficient technique for inference, however without sacrificing the expressibility.

The main aim of this thesis is to show the feasibility of an efficient implementation of dl-programs using an RDBMS. In particular, we answer the emerging questions of how expressible the proof of concept is, what technical limitations are encountered, and how scalable the implementation is.

The following list gives a brief overview of the contributions of this thesis:

1. We show that dl-programs under stratified semantics are rewritable into SQL, with the restriction that the DL-Lite plug-in provides positive Datalog. To achieve this rewriting, we leverage from stratified Datalog and DL-Lite, since both formalisms are rewritable in SQL.
2. We report on a prototype implementation, called *MOR*, where an existing DL-Lite reasoner is incorporated into our Datalog rewriter. The Datalog rewriter takes advantage of linear recursive queries in SQL:1999 [ISO, 1999].
3. For showing the scalability of MOR, we develop a benchmark suite considering expressibility and performance. The suite is separated into a scenario for plain Datalog and two scenarios for Datalog combined with DL-Lite.
4. Based on the benchmark suite, we produce and compare experimental results with MOR and the reasoners of the DLV family. We can remark in advance that the results are encouraging.

An example should illustrate the need for our efforts. Take a "smart" route planner, which should provide the user not just with short routes, but with customized routes. These custom routes, tailored to the needs of an user, could consider environmental, monetary, or even shopping objectives. Ontologies would be needed to define different means of

[5]http://www.kr.tuwien.ac.at/research/systems/dlvhex/
[6]http://www.dbai.tuwien.ac.at/proj/dlv/

transport, geographical locations, and different types of shops. Furthermore ontologies would be needed to link different types of information to each other. The data for this information would be extracted from several external sources (e.g. public rail system, maps, routes, yellow pages, etc.). Then we would need rule based reasoning to calculate the shortest, cheapest, most environmental friendly or most "interesting" way. This could be even extended with the capabilities for user to formulate their own constraints regarding transport, money or their personal interests. This example illustrates that new kinds of applications could be captured by integration ontologies with rules in a scalable fashion.

1.1. Logic Programming

In the 1970s and 1980s, LP evolved out of a debate about two different paradigms of KR. One paradigm, *procedural* knowledge representation, advocated mainly at MIT around Marvin Minsky, features recursive procedures that operate on lists. *Lisp*, based on Lambda Calculus, become the main programming language for this paradigm [McCarthy, 1960]. It is still in considerable use by AI-researchers.[7] The other paradigm, *declarative* knowledge representation, features logic as a declarative language, which is evaluated by a theorem-prover or model-generator. This paradigm was advocated around John McCarthy of Standford, Pat Hayes, and Bob Kowalski of Edinburgh. The main idea evolved from the deduction method *Resolution Principle*, developed by John Alan Robinson in 1965 [Robinson, 1965]. This deduction method was then implemented by Alain Colmerauer in *Prolog*. The principle of Prolog can be subsumed as: ALGORITHM = LOGIC + CONTROL [Colmerauer, 1985]. As Lisp, Prolog attracted and held a strongly devoted user community. Carl Hewitt in [Hewitt, 2009] gives an in-depth view of the developments and controversies in LP research around the 1970s.

1.1.1. Datalog

Strongly influenced by the research in the relational DB field, Datalog restricts LP and particularly Prolog, to a function-free first-order vocabulary. Due to its use in the DB field, facts are not stored in the logic program itself, but kept in an *extensional database* (EDB) usually maintained by an RDBMS. Datalog can be traced back to several researchers, particularly to H. Gallaire and J. Minker, which were researching the intersection between LP and DBs [Gallaire et al., 1977]. The name "Datalog" was coined later by David Maier. An interesting property of Datalog is related to its semantics. Datalog comes with three different equivalent semantics, namely model-theoretic, fixpoint, and proof-theoretic semantics [Abiteboul et al., 1995]. We will have a closer look at Datalog in Chapter 2.

According to Abiteboul et al. Datalog can be distinguished from LP as follows [Abiteboul et al., 1995]:

[7] See the conference for the 50th anniversary of Lisp: http://www.lisp50.org/

- Syntax: In Datalog only relation symbols are allowed, hence functions symbols are excluded. Furthermore variables in Datalog rules have to fulfill certain safeness conditions.
- Model-theoretic semantics: Datalog programs always have finite models, in opposite to infinite models in LP.
- Fixpoint semantics: Fixpoint semantics does not certainly provide a constructive semantics for LP.
- Proof-theoretic semantics: In LP, SLD resolution is crucial, due to the infiniteness of answers, whereby bottom-up approaches are not feasible. In Datalog, resolution is rather used for optimization (e.g. magic sets).
- Expressive power: LP can express all recursive enumerable languages predicates, whereby Datalog's expressive power is in PTIME.

Example 1. The ancestor problem is a well-known example for Datalog:

$parent(a, b).\ parent(b, c).$
$parent(b, d).\ parent(d, e).$
$ancestor(X, Y) \leftarrow parent(X, Y).$
$ancestor(X, Y) \leftarrow parent(X, Z),\ ancestor(Z, Y).$

1.1.2. Answer-Set Programming

Answer-Set Programming (ASP) is a nonmonotonic LP paradigm based on the *Stable Model Semantics* [Gelfond and Lifschitz, 1988] and extended with classical negation in [Gelfond and Lifschitz, 1991]. The development of ASP was surely influenced by certain limitations of Prolog. One limitation is the absence of a purely declarative representation of Prolog programs, because the order of the rules in Prolog are important for their evaluation. If *negation-as-failure* (NAF) is interpreted in the Stabe Model Semantics, the NAF of a literal means that the literal is "not known", which differs to the classical interpretation of the literal's negation. In Prolog classical negation is simply omitted. Moreover the solution of a Prolog program is not encoded in a model. With ASP some limitations of Prolog were overcome and a more general problem solving strategy evolved. This strategy can be outline according to [Eiter et al., 2009a] as follows:

1. A problem instance is encoded in a (nonmonotonic) logic program, such that the solutions are represented by the models of the program;
2. some models of the program are computed using an ASP solver; and
3. a solution for the problem is extracted from the model.

This strategy is well suited to solve NP-complete problems like three-colorability of a graph [Eiter et al., 2009a].

Example 2. To illustrate the method of ASP, we take the strategic companies problem as an example. Central to this problem is the concept of a strategic set, which is a minimal set of companies, being controlled by three other strategic companies. Based on this, it should be determined which companies can be sold, whereby all products still have to be produced and no company should be controlled by its holding company after selling:

$prod\_by(p1, c1, c2).\ prod\_by(p2, c2, c3).$
$prod\_by(p3, c3, c4).\ prod\_by(p4, c4, c5).$
$contr\_by(c1, c2, c3, c4).\ contr\_by(c2, c1, c3, c4).$
$contr\_by(c4, c2, c3, c1).\ contr\_by(c3, c1, c2, c4).$
$strateg(C1) \lor strategic(C2) \leftarrow prod\_by(P, C1, C2).$
$strateg(C) \leftarrow contr\_by(C, C1, C2, C3),\ strateg(C1),\ strateg(C2),\ strateg(C3).$

An ASP solver would return several answer sets for this example.

1.1.3. Tools

For Prolog, SWI-Prolog[8] is a widely used open source implementation. Datalog is more a conceptional language and has effected RDBMS standards. For example the SQL:1999 standard is partly influenced by Datalog [ISO, 1999]. Furthermore Datalog had an impact on deductive DB systems like XSB[9].

ASP has been implemented by the following systems:

- DLV[10], a joint development of University of Calabria and Vienna University of Technology, extends ASP with weak constraints, aggregates, and a SQL front-end [Leone et al., 2006]. DLV^{DB} [11] is an extended development of DLV for evaluating ASP on RDBMS [Terracina et al., 2008]. Finally, in dlvhex[12] the concept of modularization was introduced [Eiter et al., 2006]. We will use DLV, DLV^{DB}, and dlvhex in our experiments.
- Smodels[13] , developed at Helsinki University of Technology, extending ASP with similar functions as DLV [Niemelä and Simons, 1997].
- Clasp[14], developed at University of Potsdam and using a conflict-driven solving technique [Gebser et al., 2007a].

[8] http://www.swi-prolog.org/
[9] http://xsb.sourceforge.net/
[10] http://www.dbai.tuwien.ac.at/proj/dlv/
[11] http://www.mat.unical.it/terracina/dlvdb/
[12] http://www.kr.tuwien.ac.at/research/systems/dlvhex/
[13] http://www.tcs.hut.fi/Software/smodels/
[14] http://www.cs.uni-potsdam.de/clasp/

1.2. Semantic Web Technologies

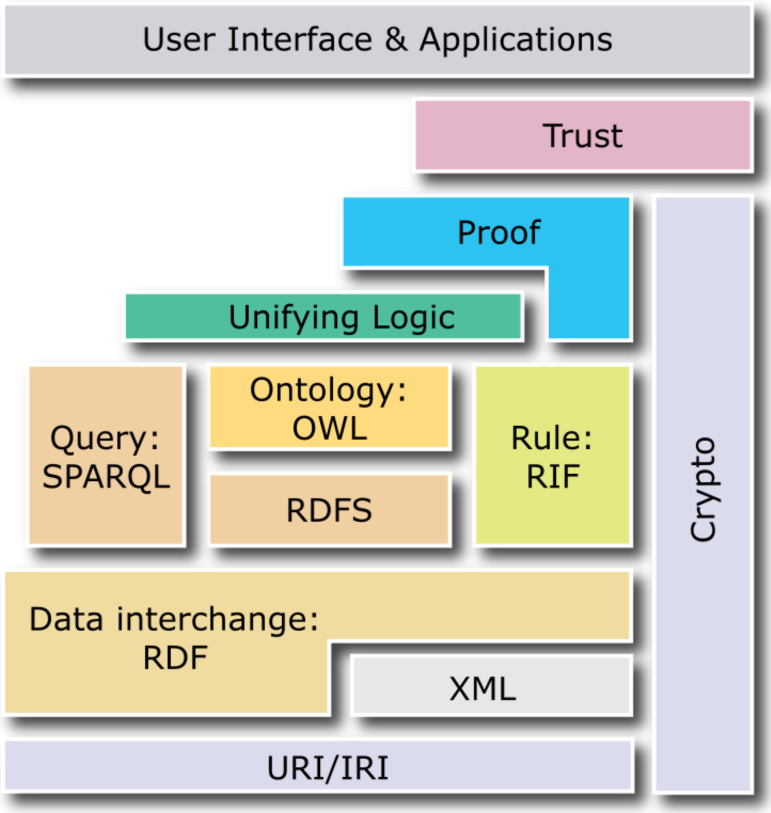

Figure 1.1.: Semantic Web Stack

Created by Tim Berners-Lee, the *Semantic Web Stack*[15] (see Figure 1.1), also called *Semantic Web Layer Cake*, illustrates the architecture of the Semantic Web. Naturally in this stack-based architecture, every technology is based on the layer below. In the following, we give a quick overview of the main technologies:

- URI/IRI (*Uniform / Internationalized Resource Identifier*):[16] This level provides a way to identify a name or resource on the Internet or in a XML-document.
- XML (*Extensible Markup Language*):[17] XML is designed as a general markup language and it is used to specify semi-structured documents in a well-formed format.
- RDF: RDF defines a directed labeled graph, which is represented by statements called *triples*. The triples themselves have the form of (*Subject Predicate Object*),

[15] http://www.w3.org/2007/03/layerCake.svg
[16] http://tools.ietf.org/html/rfc3986 and http://tools.ietf.org/html/rfc3987
[17] http://www.w3.org/XML/

whereby
- the Subject is a vertice and the tail of the edge representing the triple;
- the Object is a vertice and head of the triple's edge;
- the Predicate is defined as the label of the edge.

A Subject is represented by either a *resource* (URI) or a *blank node*; an Object is represented by either a resource, a blank node, or a *data type literal*.

- RDFS (*RDF Schema*):[18] RDFS extends RDF with a basic vocabulary for ontologies, e.g. *Class*, *Property,* and *Label*.
- SPARQL:[19] SPARQL is a RDF query language loosely based on SQL. SPARQL can be seen as the main technology to retrieve information from RDF graphs.
- OWL: In the following section we will have a closer look at it.
- RIF and *Unifying Logic*: RIF serves as an interchange format between existing rule systems. RIF is, as well as Unifying Logic an important focus of current Semantic Web research, whereby Unifying Logic is aiming for a combined formalisms of rules and ontologies. These topics are a key issues of this thesis and will be discussed in the following chapters.
- Proof: This layer is concerned with proof techniques for the underlying ontologies, rules, and unifying logic.
- Trust and Crypto: The provided information are validated and supported regarding sound and complete reasoning and trusted sources.
- User Interface & Applications: This level relates to applications which make use of the Semantic Web or give access to one of the different layers.

1.2.1. OWL

Already ongoing for several years, a main focus of Semantic Web research was shaping an adequate language for ontology modeling. As one of the achievements of this research, in 2004, OWL was for the first time recommended [Schreiber and Dean, 2004]. Now, the most recent recommendation by the W3C is OWL 2 [Krötzsch et al., 2009]. There are still speculations about the confusion of "W" by "O" in OWL. According to Tim Finin, OWL is primary an acronym for the bird, which is easy to illustrate and associated with wisdom. Additionally OWL relates to an early KR language called *One World Language*.[20]

In the OWL 2 Primer the main two alternative semantics for OWL 2 are outlined. A RDF-based semantics, called *OWL 2 Full*, which allows the full expressivity of RDFS, with the unfavorable drawback of being undecidable. On the other hand, a DL based

[18] http://www.w3.org/TR/rdf-schema/
[19] http://www.w3.org/TR/rdf-sparql-query/
[20] http://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0169.html

semantics, called *OWL DL*, which puts syntactic restrictions on RDFS [Krötzsch et al., 2009].

DL is a family of KR formalisms based on a fragment of *first-order logic* (FOL). A DL *knowledge base* (KB) is separated into an *intentional knowledge base* (TBox) and an *extensional knowledge base* (TBox). The base vocabulary of a DL consists of *Individuals*, *Classes,* and *Roles*. Furthermore, Classes and Roles are put in relations to eachother and Individuals are asserted to them [Baader et al., 2004]. We assume that DL was chosen due its well-defined computational properties and modular concept. A modular concept in that sense, that most families of DL are based on the simple DL called $\mathcal{ALC}$, whereas $\mathcal{ALC}$ expressivity is extended with new language constructs. For example $\mathcal{SHOIN}$(D) extends $\mathcal{ALC}$ with transitivity, cardinality, equivalence between Individuals, functional Roles, and more. For further details, we refer the interested reader to [Baader et al., 2004].

The DL $\mathcal{SHOIN}$(D) provides the formal base of OWL DL and enables, due to its computational properties, the development of efficient reasoning systems. Notice that undecidability in OWL Full results mainly from not distinguishing between the sets of Individuals, Classes, and Roles. In contrast to this, these types are pairwise disjunctive sets in $\mathcal{SHOIN}$(D).

With OWL 2 a further extension called *OWL 2 Profiles* was introduced by the W3C [Motik et al., 2009]. The aim of these profiles is, formally called fragments, trading expressive power for lower complexity. As we will see with OWL 2 QL, this trade-off establishes capabilities of using technologies like RDBMS. Note that in DL, most of the reasoning systems are based on tableaux based algorithms.

Figure 1.2 gives an example of a geospatial OWL ontology.[21] In this ontology spatial relationships are defined based on the existing ontologies Geonames[22] and GeoOWL[23]. For example the *RoadFeature* is a subclass of *TypedFeature*, which has the property *hasFeatureCode*.

1.2.2. OWL 2 EL

The authors of [Motik et al., 2009] describe this profile as: "OWL 2 EL is particularly useful in applications employing ontologies that contain very large numbers of properties and/or classes. This profile captures the expressive power used by many such ontologies and is a subset of OWL 2 for which the basic reasoning problems can be performed in time that is polynomial with respect to the size of the ontology [*EL++*]."

Motivated by negative conclusions regarding complexity in DL research, EL^{++} was introduced in [Baader et al., 2006] to provide a tractable formalism, which is expressive

[21] http://www.geospatialmeaning.eu/wp-content/uploads/2008/07/geoconcepts_ontology_skelet.gif
[22] http://www.geonames.org/
[23] http://www.w3.org/2005/Incubator/geo/XGR-geo/

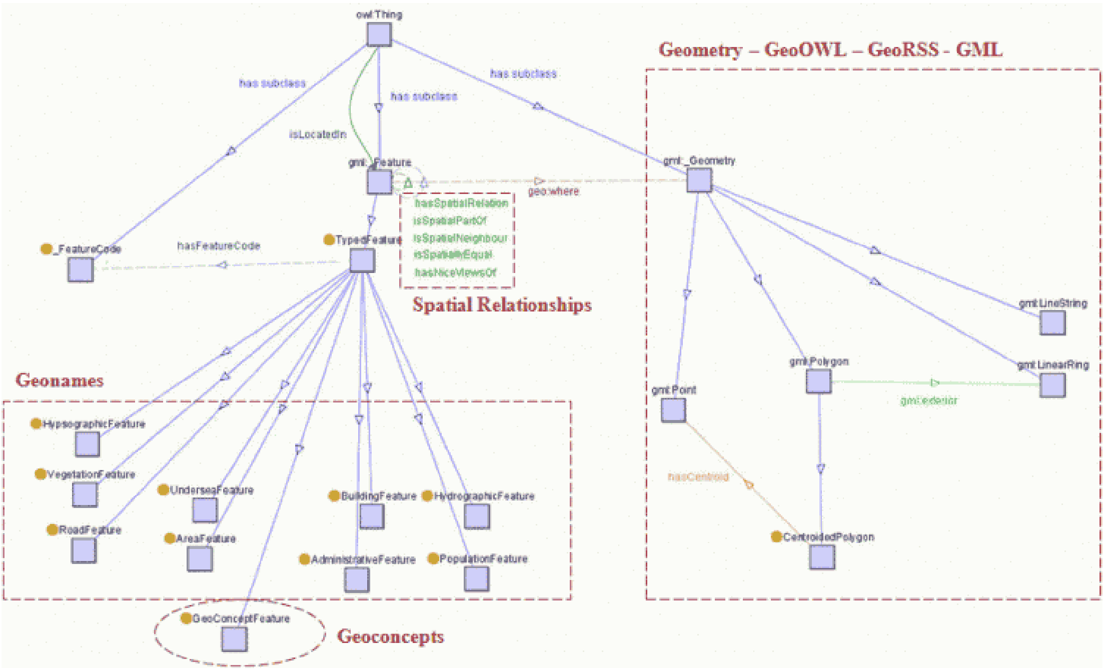

Figure 1.2.: A Geospatial Ontology

enough to capture ontologies used in practice. For example, the biomedical ontology SNOMED CT[24] is expressible in EL^{++}.

1.2.3. OWL 2 QL

In [Motik et al., 2009] this profile was summarized as: "OWL 2 QL is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. In OWL 2 QL, conjunctive query answering can be implemented using conventional relational database systems. Using a suitable reasoning technique, sound and complete conjunctive query answering can be performed in LOGSPACE with respect to the size of the data (assertions)."

OWL 2 QL will be one of the main focus points of this thesis, due its capabilities of answering conjunctive query over a DL KB using an RDBMS. In Chapter 2 we will have a in-depth look at the DL-Lite family and particularly at $DL\text{-}Lite_R$ [Calvanese et al., 2007].

1.2.4. OWL 2 RL

Again in [Motik et al., 2009] this profile is characterized as: "OWL 2 RL is aimed at applications that require scalable reasoning without sacrificing too much expressive power.

[24]http://www.ihtsdo.org/snomed-ct/

It is designed to accommodate OWL 2 applications that can trade the full expressivity of the language for efficiency, as well as RDF(S) applications that need some added expressivity. OWL 2 RL reasoning systems can be implemented using rule-based reasoning engines. The ontology consistency, class expression satisfiability, class expression subsumption, instance checking, and conjunctive query answering problems can be solved in time that is polynomial with respect to the size of the ontology."

According to Motik et al. the design of OWL 2 RL was influenced by *Description Logic Programs* and *pD**, which enables the implementation of reasoning capabilities by rule-based reasoner [Motik et al., 2009].

1.2.5. Tools

Depending on the level of the Semantic Web Stack, different tools come into use. For editing RDF any XML-Editor can be used, for editing OWL the open-source editor *Protégé*[25] is convenient. For developing Semantic Web applications the *Jena*[26] framework provides a favorable starting poing.

However, our interest is more related to the DL reasoning systems. For OWL DL well-known systems are:[27]

- FaCT++[28],
- Hermit[29],
- KAON2[30],
- Pellet[31], and
- RacerPro[32], which will be used in combination with dlvhex for parts of our experiments.

The following OWL 2 Profiles are supported by:

- CEL supports OWL EL[33],
- QuOnto[34] and Owlgres[35] supports DL-Lite$_{\mathrm{R}}$and OWL QL, whereas we will have a in-depth look at Owlgres in Chapter 4, and
- ORACLE 11g[36] supports OWL RL.

[25] http://protege.stanford.edu/
[26] http://jena.sourceforge.net/
[27] A comprehensive list of DL reasoners can be found on http://www.cs.manchester.ac.uk/~sattler/reasoners.html
[28] http://owl.man.ac.uk/factplusplus/
[29] http://hermit-reasoner.com/
[30] http://kaon2.semanticweb.org/
[31] http://clarkparsia.com/pellet/
[32] http://www.racer-systems.com/products/racerpro/
[33] http://lat.inf.tu-dresden.de/systems/cel/
[34] http://www.dis.uniroma1.it/~quonto/
[35] http://pellet.owldl.com/owlgres/
[36] http://www.oracle.com/database/

1.3. Combining Rules and Ontologies

In the past couple of years, another focus of Semantic Web research was towards finding a combined KR formalism for rules and ontologies. The discussion was encouraged by certain shortcomings of DL. The authors of [Motik et al., 2006] point out some reasons for extending DL with rules:

- Higher Relational Expressivity: DL is designed to model relations in a tree-like manner, whereby in LP general relational structures can be defined.
- Polyadic Predicates: In DL only unary (Classes) and binary (Roles) predicates are intended. Particularly in the DB field, larger predicates are common.
- Closed-World Reasoning: Again in the relational and deductive DB field, it is desired, that if no proof of a positive ground literal is found, then the negation of that literal is assumed true [Reiter, 1977].
- Integrity Constraints: In FOL expressing constraints as used in ASP, is not possible. In ASP, constraints are special rules with an empty head and effect the filtering of unwanted models.
- Modeling Exceptions: In DL being a strict subset of FOL, NAF is not expressible, which is considered an important capability of non-monotonic formalisms. The famous “usually birds fly, but penguins cannot fly” is used to illustrate this shortcoming.

The *Semantic Web Rule Language* (SWRL) was one of the first proposals to overcome these limitations. The rule layer in SWRL was set on top of OWL, achieved by allowing material implication of OWL expressions [Horrocks et al., 2004]. This leads to undecidability in general, however, fragments of SWRL are implemented in several DL reasoner (e.g. KAON2 and Pellet).

In Eiter et al. [Eiter et al., 2008b] an interesting taxonomy of combination approaches is given. The approach is mainly based on the level of integration. Another taxonomy is given by the authors of [Mei et al., 2006], which differs between homogeneous and hybrid approaches, taking into account safeness condition and information flow. We will have a detailed look at the taxonomy of [Eiter et al., 2008b].

1.3.1. Loose Coupling

The rule and DL KB are kept as separate, independent components. An interface mechanism connects both components allowing the exchange of knowledge between them. The interface is designed in a way, that decidability is guaranteed for the combined KB. Furthermore the knowledge flow can be uni- or bi-directional.

Resulting from dl-programs [Eiter et al., 2004], HEX-programs [Eiter et al., 2006] belong to the loose coupling approach. In Chapter 3 we will view dl-programs more detailed.

1.3.2. Tight Semantic Integration

In this approach the rule and DL KB are kept distinct. The integration will not occur through an interface mechanism, but through the integration of the rule and DL models, whereby each model should satisfy its domain and “agree” with the other model.

CARIN [Levy and Rousset, 1998] and $\mathcal{DL} + log$ [Rosati, 2006] represent this approach.

1.3.3. Full Integration

The authors of [Eiter et al., 2006] describe full integration the following way: “Full integration approaches are mostly distinct by the absence of separation between two vocabularies at hand: the two universes are treated to a large extent in a homogeneous way”.

Description Logic programs (DLP) [Grosof et al., 2003], *Hybrid MKNF knowledge bases* [Motik et al., 2006], *first-order Autoepistemci Logic* [de Bruijn et al., 2006], and *Open Answer Set Programs* [Heymans et al., 2007] can be counted to this approach.

1.4. Structure of the Thesis

The structure of the thesis is the following. Chapter 2 provides the formal foundation of combining rules and ontologies. In Chapter 3 the evaluation of dl-programs under stratified Datalog is presented. Furthermore, the reformulation of dl-programs to SQL is described. Chapter 4 highlights the technical aspects of the prototype MOR. In Chapter 5 we introduce a benchmark suite regarding the evaluation of rule based and combined programs. In Chapter 6 we report on the empirical results relating the performance of MOR in comparison with similar reasoning systems. In Chapter 7 the main results are summarized and we outline future work and further studies.

2. Preliminaries

In Chapter 1 we gave a brief introduction to Datalog, which will be further elaborated in this chapter. In Datalog it is feasible to express program classes with *unstratified negation* (e.g. *normal programs* under *well-founded semantics* [Gelder et al., 1991] or under stable model semantics [Gelfond and Lifschitz, 1991]). However, we consider the less expressive class of *stratified programs,* which impose some syntactic restrictions on normal programs [Apt et al., 1988]. For current SQL standards, namely SQL:1999, stratified programs suffice to capture the expressivity of SQL [ISO, 1999]. Afterwards, we capture the DL-Lite family and the notion of *First-Order Reducibility* (FOL-reducibility) for different reasoning tasks. This introduction is kept close to the paper of [Calvanese et al., 2007]. Then, we introduce the formalism of dl-programs. The main idea of dl-programs is combining normal programs under well-founded semantics with different fragments of DL [Eiter et al., 2009b]. Again we remain close to the mentioned paper.

2.1. Relational Algebra

General relational algebra is a well studied field and already Tarski was concerned with it. However his algebra is solely based on binary relations [Tarski, 1941]. *Codd's Relational Algebra* (RA) as an algebraic notation is associated with the *Relational Data Model* and still is an important formalism in the DB field [Codd, 1970]. Despite RA was introduced in 1970, it is used as a basic concept of RDBMS. We follow the definitions for RA from [Ceri et al., 1990] and [Ullman, 1988]. Particularly Ullmann elaborated the relation between RA and Datalog and showed that except recursion RA is as expressive as Datalog [Ullman, 1988].

Definition 3. [Ullman, 1988] The relational data model consists of *relations* and *domains*. Let $D_1, ..., D_n$ be sets, called domains, where $D = D_1 \times ... \times D_n$. A relation defined on $D_1, ..., D_n$ is any subset R of D. Elements of relations are called *tuples*, defined as $\langle d_1, ..., d_n \rangle$ with $d_1 \in D_1, ..., d_n \in D_n$. The columns of a relation are called *attributes*. The set of attributes for a relation R is denoted the *schema* of R. The attributes of R can either be referred by their name or by their position in the schema.

We assume that domains are finite sets, because in the context of RDBMS infinite domains can be neglected. Since a relation is a set, the tuples are distinct but not ordered.

Definition 4. [Ullman, 1988] Relational Algebra has the following basic operators:

- *Union* ($\cup$): Given relations R and S, $R \cup S$ is the *set-theoretic union* of the tuples belonging to R and S. To ensure the result is again a relation, R and S must have identical schemas. This condition is also called *union-compatibility.*
- *Difference* ($-$): Given relations R and S, $R - S$ is the *set-theoretic difference* of the tuples belonging to R and S. Union-compatibility must be fulfilled.
- *Cartesian product* ($\times$): Given relations R and S, $R \times S$ is the set of all tuples t such that t is the concatenation of a tuple $r \in R$ and a tuple $s \in S$.
- *Projection* (π_L): Given a list of attributes L, the tuples of the result are derived from the tuples of the operand relation by elimination of the attributes which are not in L.
- *Selection* (σ_F): Let F be a formula involving operands that are constants, attributes, arithmetic comparison operators (e.g. $<, >, \leq, ...$), and logical operators (e.g. $\neg, \vee, \wedge$). Then, the result of a selection σ_F on a relation R is the set of tuples of R which fulfill formula F.

Definition 5. [Ullman, 1988] The Relational Algebra operators *Intersection* ($\cap$), *Complement* ($\setminus$), *Natural join* ($\bowtie$), *θ-join* (θ), *Semijoin* ($\ltimes$), *Antijoin* ($\triangleright$), and *Outer join* can be derived from by the basic operators.

Definition 6. [Ceri et al., 1990] If we exclude the difference operator, we obtain the sublanguage *Positive Relational Algebra* (RA^+).

Due to the success of SQL, RA never become a query language for RDBMS, nevertheless it is often used for the internal representation of queries. Following the definitions from [Ceri et al., 1990], we show that SQL can be interpreted in RA.

Definition 7. [Ceri et al., 1990] Let Q be a set of relations and let S be a SQL block of the form:

$$\texttt{SELECT}\ \langle A \rangle\ \texttt{FROM}\ \langle Q \rangle\ \texttt{WHERE}\ \langle F \rangle,$$

where S is interpreted by applying selection σ_F to Q and projection π_A to Q. Q is defined as the Cartesian product of all relations in Q. If F contains a join condition, instead of the Cartesian product a *θ-join* can be directly evaluated. Furthermore, several SQL blocks can be combined with the set-based operators union, natural join, and difference to build new SQL blocks.

Out of Definition 7, we can also apply the reversal to rewrite RA expressions into SQL.

2.2. Stratified Programs

First we give a brief introduction to positive programs and extend them to stratified programs.

2.2.1. Syntax and Semantics of Positive Programs

The following definitions are taken from [Eiter et al., 2009a].

Definition 8. A program P is defined on an *alphabet* $\Phi = (\mathcal{S}, \mathcal{V}, \mathcal{C})$, consisting of the nonempty sets of *predicates* $\mathcal{S}$, *variables* $\mathcal{V}$, and *constants* $\mathcal{C}$. A *term* is either a constant or a variable. An *atom* is defined as $p(t_1, ..., t_k)$, where $p \in \mathcal{S}$, each $t_1, ..., t_k$ is a term, and k is called the arity of p. A *classical literal* is a positive (resp. negative) atom a (resp. $\neg a$). A *negation-as-failure (NAF) literal* has the form of the *default-negated atom* a, denoted as *not* a. *Propositional atoms* are atoms with arity $k = 0$.

Definition 9. A *positive program* is a finite set of *rules* (clauses) of the form:

$$a \leftarrow b_1, ..., b_m. ,$$

where $a, b_1, ..., b_m$ are atoms based on alphabet Φ. We refer to $H(C)$ as the *head* of C and the conjunction $b_1, ..., b_m$ is denoted as the *body* $B(C)$. We denote rule C as a *fact* iff $m = 0$.

Definition 10. Given a program P, the *Herbrand universe* HU_P is the set of all ground terms which can be formed from the alphabet Φ. The *Herbrand base* HB_P is the set of all ground atoms which can be formed from predicates in $\mathcal{S}$ and the terms in HU_P. For any rule $C \in P$, we call $ground(C)$ the set of all possible ground instances of C.

Definition 11. A *(Herbrand) interpretation* is an interpretation I over HU_P, such that I is a subset of HB_P.

Definition 12. A ground rule C is *satisfied* in an interpretation I, if the head literal is true in I or at least one body literal is false in I. It is *falsified* if the head literal is false in I and all body literals are true in I.

Definition 13. Let I be an interpretation. Then I is a *model* of

- a ground rule $C = a \leftarrow b_1, ..., b_m$, denoted $I \vDash C$, if the rule C is satisfied;
- a rule C, denoted $I \vDash C$, if $I \vDash C'$ for every $C' \in ground(C)$;
- a program P, denoted $I \vDash P$, if $I \vDash C$ for every rule $C \in P$.

In LP there is usually no distinction between predicates appearing in the head or body. By contrast in Datalog predicates are separated into distinct sets, according to Abiteboul et al. defined as:

Definition 14. [Abiteboul et al., 1995] Let P be a *Datalog program*, the *extensional database* denoted $EDB(P)$ (resp. *intensional database* $IDB(P)$) is the set of all predicates $p \in P$, iff there exists a rule $C \in P$ such that $p \in B(C)$ (resp. $p \in H(C)$).

Definition 15. [Ceri et al., 1990] Let P be a Datalog program, rule $C \in IDB(P)$ has to satisfy the following conditions:

(i) The predicate occurring in the head of C belongs to $IDB(P)$.

(ii) All variables which occur in the head of C also occur in the body of C. (ii) is called *safety condition.*

2.2.2. Dependency Graphs of Logic Programs

Adopted from [Ullman, 1988] we give the definition for a dependency graph of a logic program.

Definition 16. [Ullman, 1988] Let P be a program. The *dependency graph* of P is defined as a directed labeled graph $G = (V, E, L)$, where V consist of all predicates of P, $L = \{+, -\}$ such that, (i) for all $p, q \in V$, $\langle p, q, +\rangle \in E$, iff there is a rule $C \in P$ such that $p \in head\, of\, C$ and $q \in postive\, body\, of\, C$, (ii) for all $p, q \in V$, $\langle p, q, -\rangle \in E$, iff there is a rule $C \in P$ such that $p \in head\, of\, C$ and $q \in negative\, body\, of\, C$.

In the context of a dependency graph the notion of topological sorting is interesting.

Definition 17. [Tarjan, 1976] Let $G(V, E)$ be a directed acyclic graph. A *topological sort* of G is the sequence $S = \left\{v_1, v_2, ..., v_{|V|}\right\}$ in which each vertex of V appears exactly once. For every pair v_i and v_j of distinct vertices in S, if there is an edge in G from v_i to v_j, then $i < j$.

2.2.3. Fixpoint Theory

Knaster-Tarski Theorem and Kleene's fixed-point Theorem are used in several proofs.

Definition 18. Let X be a set and the operator $T : \mathcal{P}(X) \to \mathcal{P}(X)$ be a function . We say that T is *monotone*, if for all X,Y,$X \subseteq Y$ it follows $T(X) \subseteq T(Y)$. We say that T is *finitary* if for every infinite sequence $S_0 \subseteq S_1 \subseteq ...$,

$$T\left(\bigcup_{n=0}^{\infty} S_n\right) \subseteq \bigcup_{n=0}^{\infty} T(S_n)$$

holds. If T is both monotonic and finitary then it is called *continuous.* Another often used equal definition of continutiy is for every infinite sequence $S_0 \subseteq S_1 \subseteq ...$, it is

$$T\left(\bigcup_{n=0}^{\infty} S_n\right) = \bigcup_{n=0}^{\infty} T(S_n).$$

Observe that each continuous operator is also monotone, but the other direction does not hold.

Theorem 19. *[Knaster-Tarski's Fixpoint Theorem] Let T be a monotonic operator on a nonempty set X. Then T has a least fixpoint, denoted $lfp(T)$:*

$$lfp(T) = \bigcap\{X : T(X) \subseteq X\} = \bigcap\{X : T(X) = X\}$$

Definition 20. Let T be a monotonic operator on a nonempty set X. For each finite and transfinite ordinal the *ordinal power* of T is defined as follows, where n is an arbitrary ordinal and ω is an arbitrary limit ordinal:

$$\begin{array}{rcl} T\uparrow_0(X) & = & X \\ T\uparrow_{n+1}(X) & = & T(T\uparrow_n(X)) \\ T\uparrow_\omega(X) & = & \bigcup\limits_{n<\omega} T\uparrow_n(X) \end{array}$$

Theorem 21. *[Kleene's Fixpoint Theorem] Let T be a continuous operator. Then $lfp(T) = T\uparrow_\omega$ holds (ω is the first limit ordinal, the one corresponding to $\mathbb{N}$).*

2.2.4. Syntax of Stratified Programs

In stratified programs positive programs are extended with NAF literal, keeping certain syntactic restrictions regarding the NAF literal.

Definition 22. [Eiter et al., 2009a] A *normal program* is a finite set of rules based on the alphabet Φ, where a rule is in the form:

$$a \leftarrow b_1, ..., b_k, not\, b_{k+1}, ..., not\, b_m. ,$$

where $a, b_1, ..., b_m$ are atoms and $m \geqslant k \geqslant 0$.

By convention, which is also used in the DLV family, variable names start with uppercase letters, whereas predicate and constant names start with lowercase letters. Furthermore underscore "_" denotes an *anonymous variable*, which stands for a variable which is not used anywhere else in the program.

Definition 23. [Eiter et al., 2009a] For a rule C of a normal program, we refer to $H(C)$ as the *head* of C, the conjunction of $b_1, ..., b_k, not\, b_{k+1}, ..., not\, b_m$ is denoted as the *body* $B(C)$. $B(C)$ can be separated into $B^+(C)$ and $B^-(C)$, where the former represents all positive atoms $b_1, ..., b_k$ and the later all default-negated atoms $not\, b_{k+1}, ..., not\, b_m$.

Definition 24. [Apt et al., 1988] Let P be a normal program. A *stratification* is a partition $P = P_1, ..., P_n$ such that for $i = 1, ..., n$ holds:

(i) if a positive literal occurs in a clause in P_i then its relation symbol is defined within $\bigcup\limits_{j \leq i} P_j$.

(ii) if a negative literal occurs in a clause in P_i then its relation symbol is defined within $\bigcup\limits_{j < i} P_j$.

Note that P_1 can be empty. We denote each P_i as a *stratum*.

Lemma 25. *[Apt et al., 1988] A normal program P is stratified iff its dependency graph G_P has no cycle containing a negative labeled edge.*

Lemma 26. *[Apt et al., 1988] A normal program P is stratified iff there exists a stratification of P.*

2.2.5. Semantics of Stratified Programs

Stratification is a syntactic property, however it also has "nice" semantical properties. Apt et al. introduced an iterated fixpoint semantic for stratified programs [Apt et al., 1988]. They provide the notions and results, which are recalled in shortened form for this thesis. For this section, we denote by I a Herbrand interpretation following Definition 11.

Definition 27. An interpretation I of program P is *supported* iff for each atom $a \in I$ there exists a ground clause C with $a \in H(C)$ and $B(C)$ is true in I .

Lemma 28. *Let P be a program. Then I is a model iff $T_P(I) \subseteq I$.*

Proof. See [Lloyd, 1984] for the proof for programs without negation. □

Lemma 29. *Let P be a program. Then I is supported iff $T_P(I) \supseteq I$.*

Proof. Direct from definition. □

Operators are studied over an arbitrary, fixed, complete lattice. The least element is denoted as ϕ and the elements of the lattice as I, J, M. The order relation on the lattice is denoted as $\subseteq$.

Lemma 30. *If T is finitary then for all I*

$T(T \uparrow_\omega (I)) \subseteq T \uparrow_\omega (I)$.

Proof. See Lemma 4 in [Apt et al., 1988]. □

Lemma 31. *If T is growing then for all I.*

$T(T \uparrow_\omega (I)) \subseteq I \cup T \uparrow_\omega (I)$.

Proof. See Lemma 5 in [Apt et al., 1988]. □

Now we take the fixed, complete lattice I, J, M and introduce the notion of *iterations*. Let $T_1, ..., T_n$ be operators . We put

$N_0 = I$
$N_1 = T_1 \uparrow_\omega (N_0)$
...
$N_n = T_n \uparrow_\omega (N_{n-1})$

Notice that N_n is computed using T_i in an iterative fashion, which is expressed by the operator $iter(T_1, ..., T_n, I)$. We need to define the properties *local* and *growing* for this operator.

Definition 32. A sequence of operators $T_1, ..., T_n$ is *local,* if for all I, J and $i = 1, ..., n$

$I \subseteq J \subseteq N_n$ implies $T_i(J) = T_i(J \cap N_i)$.

Local means that each T_i is determined by its values on the subsets of N_i.

Lemma 33. *Suppose that the sequence $T_1, ..., T_n$ is local and that all T_i are finitary. Then*
$(\bigcup_{i=1}^{n} T_i)(iter(T_1, ..., T_n, I)) \subseteq iter(T_1, ..., T_n, I)$.

Proof. See Lemma 6 in [Apt et al., 1988]. □

Lemma 34. *Suppose that the sequence $T_1, ..., T_n$ is local and each T_i is growing. Then*

$iter(T_1, ..., T_n, I) \subseteq I \cup (\bigcup_{i=1}^{n} T_i) iter(T_1, ..., T_n, I))$.

Proof. See Lemma 7 in [Apt et al., 1988]. □

Corollary 35. *Suppose that sequence $T, ..., T_n$ is local and that all T_i are finitary and growing. Then*

$iter(T_1, ..., T_n, I) \subseteq I \cup (\bigcup_{i=1}^{n} T_i) iter(T_1, ..., T_n, I))$.

Thus for a local sequence $T, ..., T_n$ of finitary and growing operators $iter(T_1, ..., T_n, \phi)$ is a fixed point of $\bigcup_{i=1}^{n} T_i$.

Theorem 36. *Suppose that the sequence $T_1, ..., T_n$ is local and that all T_i are growing. If*

$I \subseteq J \subseteq iter(T_1, ..., T_n, I)$ *and*
$(\bigcup_{i=1}^{n} T_i)(J) \subseteq J$ *then*
$J = iter(T_1, ..., T_n, I)$.

Proof. See Theorem 1 in [Apt et al., 1988]. □

To relate $iter(T_1, ..., T_n, I)$ with $(\bigcup_{i=1}^{n} T_i) \uparrow_\omega (I)$, we need the following notion.

Definition 37. A sequence of operators $T_1, ..., T_n$ is *raising* if for all I, J, M and $i = 1, ..., n$

$I \subseteq J \subseteq M \subseteq N_n$ implies $T_i(J) = T_i(M)$.

Apt et al. introduce two equivalent definitions for the *minimal model* of a stratified program. We focus on the main definition which is more operational, since it is based on the iterations of the operator T_P [Apt et al., 1988]. This definition shows that for a program P stratified by $P = P_1, ..., P_n$ the interpretation of P is:

$M_1 = T_{P_1} \uparrow_\omega (\phi)$
$M_2 = T_{P_2} \uparrow_\omega (M_1)$
...
$M_n = T_{P_n} \uparrow_\omega (M_{n-1})$.

Let $M_P = M_n$.

Theorem 38. *For all programs P, T_P is finitary.*

Proof. See Theorem 4 in [Apt et al., 1988]. □

Definition 39. A program P is called *semi-positive,* if none of its negated relation symbols occurs in a head of a clause. Furthermore we define:

$Neg_P = \{A$: $\neg A$ is a variable-free instance of a negative literal in a clause in $P\}$ and

$Def_P = \{A$: A is a variable-free instance of a head of a clause in $P\}$.

Lemma 40. *Let P be a subprogram of P'. Then*

$I \subseteq J \subseteq U_{P'}$ and $I \cap Neg_P = J \cap Neg_P$ implies $T_P(I) \subseteq T_P(J)$.

Proof. See Lemma 10 in [Apt et al., 1988]. □

Informally, P' and $U_{P'}$ are used to regard T_P on a larger space.

Theorem 41. *If P is semi-positive, then T_P is growing.*

Proof. See Theorem 5 in [Apt et al., 1988]. □

Theorem 42. *If the sequence $P_1, ..., P_n$ defines new relations, then the sequence of the operators $T_{P_1}, ..., T_{P_n}$ considered on the space $U_{P_1 \cup ... \cup P_n}$ is local.*

Proof. See Theorem 6 in [Apt et al., 1988]. □

Theorem 43. *1. M_P is a model of P.*

2. M_P is supported.

Proof. See Theorem 7 in [Apt et al., 1988]. □

Theorem 44. *M_P is a minimal model of P .*

Proof. See Theorem 8 in [Apt et al., 1988]. □

We have not shown yet that the model M_P does not depend on the explicit way how P is stratified.

Theorem 45. *Let P be a stratified program. Then the model M_P is independent of the stratification of P.*

Proof. We refer to Theorem 11 in [Apt et al., 1988]. □

2.2.6. Complexity of Stratified Programs

We assume the reader is familiar with the concept of *Computational Complexity* and complexity classes (cf. [Papadimitriou, 1993]). We follow mostly [Dantsin et al., 1997]. Due to our focus on Datalog and RDBMS, we mainly consider the data complexity.

Definition 46. The *data complexity* is the complexity of checking whether $D_{in} \cup P \vDash A$ for a fixed Datalog program P and variable EDB D_{in} and ground atoms A.

The *program complexity* is the complexity of checking whether $D_{in} \cup P \vDash A$ for variable Datalog programs P and ground atoms A over a fixed EDB D_{in}. We recall that if D_{in} is fixed, then the set of constants that may appear in P and A is fixed too.

The *combined complexity* is the complexity of checking whether $D_{in} \cup P \vDash A$ for variable Datalog programs P, ground atoms A, and EDB D_{in}.

Theorem 47. *Datalog is data complete in P.*

Proof. See Theorem 3.4 in [Dantsin et al., 1997]. □

Theorem 48. *Datalog is program complete in $\mathsf{DEXPTIME}$.*

Proof. See Theorem 3.5 in [Dantsin et al., 1997]. □

Theorem 49. *Stratified propositional LP is P-complete. Stratified Datalog is data complete in P and program complete in $\mathsf{DEXPTIME}$.*

Proof. Implicit in [Apt et al., 1988]. □

2.3. DL-Lite and the Notion of FOL-Reducibility

Calvanese et al. introduced in 2005 a new family of Description Logics (DL), called DL-Lite [Calvanese et al., 2005]. DL-Lite is designed for tractable reasoning and efficient query answering. An interesting feature of DL-Lite is, while keeping a low complexity for reasoning a variety of ontology languages are still representable. Namely conceptual data models (e.g. Entity-Relationship-Models [Abiteboul et al., 1995]) and object-oriented models (e.g. basic UML class diagrams [Larman, 2001]) are still covered by DL-Lite. In the development of DL-Lite the focus was put on answering conjunctive queries over DL KB. This is again an interesting issue for this thesis, due the capabilities of DL-Lite to maintain an ABox in an RDBMS and rewriting conjunctive queries into SQL.

2.3.1. The DL-Lite Family

In [Calvanese et al., 2007] the DL-Lite family was further refined. A base DL called $DL\text{-}Lite_{core}$ was extended to $DL\text{-}Lite_F$ and $DL\text{-}Lite_R$. Our focus will be mainly on $DL\text{-}Lite_R$, because it is the logical foundation of OWL 2 QL. The following definitions are taken from [Calvanese et al., 2007].

2.3.1.1. Syntax of $DL\text{-}Lite_{core}$ and $DL\text{-}Lite_R$

We first describe the syntax of $DL\text{-}Lite_{core}$.

Definition 50. Let $\Psi = (A, P)$ be the base vocabulary, where A denotes an *atomic concept*, P denotes an *atomic role* and P^- the inverse of the atomic role P.

Definition 51. Based on the vocabulary Ψ, the following syntax can be formed:

$$
\begin{aligned}
B &\longrightarrow A \mid \exists R \\
C &\longrightarrow B \mid \neg B \\
R &\longrightarrow P \mid P^- \\
E &\longrightarrow R \mid \neg R
\end{aligned}
$$

where B denotes a *basic concept*, R denotes a *basic role*, C denotes a *general concept*, E denotes a *general role,* and $\exists R$ is an unqualified existential quantification on a basic role.

Furthermore the authors use the notation R^-, which means that $R^- = P^-$ if $R = P$, and $R^- = P$, if $R = P^-$. A similar notation is used for $\neg C$ and $\neg E$.

Definition 52. A DL KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ consists of a $DL\text{-}Lite_{core}$ or $DL\text{-}Lite_R$ TBox $\mathcal{T}$, the intentional knowledge, and an ABox $\mathcal{A}$, the extensional knowledge, where:

(i) The $DL\text{-}Lite_{core}$ TBox is defined as a finite set of *inclusion assertions* of the form: $B \sqsubseteq C$.

(ii) The *DL-Lite*$_R$ TBox is defined as a finite set of *inclusion assertions* of the form: (i) or $R \sqsubseteq E$.

(iii) The ABox is defined as a finite set of *membership assertions* of the form: $A(a)$ and $P(a, b)$, where a and b are constants.

The set of inclusion assertions can be extended with $B_1 \sqcup B_2 \sqsubseteq C$ which is equivalent to $B_1 \sqsubseteq C$ and $B_2 \sqsubseteq C$, and with $B \sqsubseteq C_1 \sqcap C_2$ which is equivalent to $B \sqsubseteq C_1$ and $B \sqsubseteq C_2$. We can use the constructs $\top$ to shorten $A \sqcup \neg A$ and $\bot$ to shorten $A \sqcap \neg A$.

Definition 53. A *conjunctive query* (CQ) q is a query of the form:

$$\left\{\vec{x} \mid \exists\vec{y}.conj(\vec{x}, \vec{y})\right\}$$

where $conj(\vec{x}, \vec{y})$ is a conjunction of atoms and equalities with free variables $\vec{x}$ and $\vec{y}$.

A *union of conjunctive queries* (UCQ) q is defined as:

$$\left\{\vec{x} \mid \bigvee_{i=1,\ldots,n} \exists\vec{y_i}.conj_i(\vec{x}, \vec{y_i})\right\}$$

where each $conj_i(\vec{x}, \vec{y_i})$ is defined as before.

2.3.1.2. Semantics of *DL-Lite*$_{core}$ and *DL-Lite*$_R$

We now define the semantics of *DL-Lite*$_{core}$, which is straightforward extendable to *DL-Lite*$_R$.

Definition 54. An *interpretation* $I = (\Delta^I, \cdot^I)$ consists of a non-empty *interpretation domain* Δ^I and an *interpretation function* $\cdot^I$ that assigns to each concept C a subset C^I of Δ^I, and to each role R a binary relation R^I over Δ^I. For the constructs of *DL-Lite*$_{core}$ we have:

$$\begin{array}{rcl}
A^I & \subseteq & \Delta^I; \\
P^I & \subseteq & \Delta^I \times \Delta^I; \\
(P^-)^I & = & \left\{(a,b) | (b,a) \in P^I\right\}; \\
(\exists R)^I & = & \left\{x | \exists y : (x,y) \in R^I\right\}; \\
(\neg B)^I & = & \Delta^I \setminus B^I; \\
(\neg R)^I & = & \Delta^I \times \Delta^I \setminus R^I.
\end{array}$$

An interpretation I is a *model* of an inclusion assertion $B \sqsubseteq C$, if $B^I \subseteq C^I$. This can be extended to a more general form. An interpretation I is a model of $C_1 \sqsubseteq C_2$ (resp. $E_1 \sqsubseteq E_2$), where C_1 and C_2 (resp. E_1 and E_2) are general concepts (resp. general roles), if $C_1^I \subseteq C_2^I$ (resp. $E_1^I \subseteq E_2^I$).

For membership assertions the interpretation function is extended to constants by assigning to each constant a a *distinct* object $a^I \in \Delta^I$. This implies that the *unique name* *assumption* [Baader et al., 2004] on constants is enforced. An interpretation I is a model of a membership assertion $A(a)$, (resp. $P(a,b)$) if $a^I \in A^I$ (resp. $(a^I, b^I) \in P^I$).

Given any assertion α, and an interpretation I, we denote by $I \models \alpha$ that I is a model of α. Given a finite set of assertions κ, we denote by $I \models \kappa$ that I is a model of every assertion in κ. A *model* of a KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is an interpretation I such that $I \models \mathcal{T}$ and $I \models \mathcal{A}$, furthermore we write $I \models \mathcal{K}$ if $I \models \mathcal{T}$ and $I \models \mathcal{A}$. A KB $\mathcal{K}$ is *satisfiable,* if it has at least one model. A KB $\mathcal{K}$ (resp. a TBox $\mathcal{T}$) *logically implies* an assertion α, written $\mathcal{K} \models \alpha$ (resp. $\mathcal{T} \models \alpha$), if all models of $\mathcal{K}$ (resp. $\mathcal{T}$) are also models of α.

It is important to note that by the extended inclusion assertions in $DL\text{-}Lite_R$ the semantics does not need to be reformulated. Furthermore $DL\text{-}Lite_{core}$ and $DL\text{-}Lite_R$ enjoy the finite model property [Baader et al., 2004], due to the absence of assertions of the form $(funct\ R)$.

Definition 55. Given a conjunctive FOL query q and a KB $\mathcal{K}$, the answer to q over $\mathcal{K}$ is the set $ans(q, \mathcal{K})$ of tuples $\vec{a}$ of constants appearing in $\mathcal{K}$ such that $\vec{a}^{\,\mathrm{M}} \in q^{\mathrm{M}}$, for every model M of $\mathcal{K}$.

Definition 56. Given a conjunctive FOL query q and a KB $\mathcal{K}$, the set of all possible tuples of constants in $\mathcal{K}$ whose arity is the one of q is denoted $AllTup(q, \mathcal{K})$.

2.3.2. Reasoning in $DL\text{-}Lite_R$

The following reasoning task are covered by the DL-Lite family:

- *Knowledge base satisfiability*, i.e. decide if a given KB $\mathcal{K}$ is satisfiable.
- *Logical implication of KB assertions*, which covers as well:
  - instance checking; and
  - subsumption of concepts or roles.
- *Query answering*, i.e. given a KB $\mathcal{K}$ and a query q over $\mathcal{K}$, compute the set $ans(q, \mathcal{K})$.

In this thesis the main focus will be on query answering, especially with the capabilities of $DL\text{-}Lite_R$ to deal with large volumes of membership assertions stored in an RDBMS.

2.3.3. FOL-Reducibility

Before we can cover the reasoning task the notion of FOL-reducibility has to be defined.

Definition 57. Given an ABox $\mathcal{A}$, an *interpretation* $db(\mathcal{A}) = \left\langle \Delta^{db(\mathcal{A})}, \cdot^{db(\mathcal{A})} \right\rangle$ is defined as follows:

(i) $\Delta^{db(\mathcal{A})}$ is the non-empty set consisting of all constants in $\mathcal{A}$,

(ii) $a^{db(\mathcal{A})} = a$, for each constant a,

(iii) $A^{db(\mathcal{A})} = \{a \mid A(a) \in A\}$, for each atomic concept A, and

(iv) $P^{db(\mathcal{A})} = \{(a_1, a_2) \mid P(a_1, a_2) \in A\}$, for each atomic role P .

Notice that the interpretation $db(\mathcal{A})$ is a *minimal model* of $\mathcal{A}$.

Definition 58. Satisfiability in a DL $\mathcal{L}$ is *FOL-reducible*, if for every TBox $\mathcal{T}$ expressed in $\mathcal{L}$, there exists a Boolean FOL query q over the alphabet of $\mathcal{T}$, such that for every non-empty ABox $\mathcal{A}$, $\langle \mathcal{T}, \mathcal{A} \rangle$ is satisfiable iff q evaluates to $false$ in $db(\mathcal{A})$.

Definition 59. Query answering in a DL $\mathcal{L}$ for unions of conjunctive queries is FOL-*reducible*, if for every union of conjunctive queries q and every TBox $\mathcal{T}$ expressed in $\mathcal{L}$, there exists a FOL query q_1, over the alphabet of $\mathcal{T}$, such that for every non-empty ABox $\mathcal{A}$ and every tuple of constants $\vec{a}$ occuring in $\mathcal{A}$, $\vec{a} \in ans(q, \langle \mathcal{T}, \mathcal{A} \rangle)$, iff $\vec{a}^{\,db(A)} \in q_1^{db(A)}$.

The idea behind FOL-reducibility is the following: instead of using common DL techniques (e.g. tableau calculus) for *satisfiability* or query answering, a FOL query is evaluated over the ABox, which is viewed as a relational DB.

2.3.4. KB Satisfiability is FOL-Reducible in $DL\text{-}Lite_R$

As a starting point it can be shown that KB Satisfiability is FOL-reducible. The concepts of positive inclusion (PI) and negative inclusion (NI) are crucial for this, where a positive inclusion (resp. negative inclusion) is an assertion of the form $B_1 \sqsubseteq B_2$ (resp. $B_1 \sqsubseteq \neg B_2$) or $R_1 \sqsubseteq R_2$ (resp. $R_1 \sqsubseteq \neg R_2$). Calvanese et al. recognized that the NIs have to be closed with respect to the PIs. They introduced NI-closure as a function of the original TBox.

Definition 60. The NI-closure of a *DL-Lite$_R$* TBox $\mathcal{T}$, denoted by $cln(\mathcal{T})$, is defined inductively as following:

1. all NI assertions in $\mathcal{T}$ are also in $cln(\mathcal{T})$;
2. if $B_1 \sqsubseteq B_2$ is in $\mathcal{T}$ and $B_2 \sqsubseteq \neg B_3$ or $B_3 \sqsubseteq \neg B_2$ is in $cln(\mathcal{T})$, then also $B_1 \sqsubseteq \neg B_3$ is in $cln(\mathcal{T})$;
3. if $R_1 \sqsubseteq R_2$ is in $\mathcal{T}$ and $\exists R_2 \sqsubseteq \neg B$ or $B \sqsubseteq \neg \exists R_2$ is in $cln(\mathcal{T})$, then also $\exists R_1 \sqsubseteq \neg B$ is in $cln(\mathcal{T})$;
4. if $R_1 \sqsubseteq R_2$ is in $\mathcal{T}$ and $\exists R_2^- \sqsubseteq \neg B$ or $B \sqsubseteq \neg \exists R_2^-$ is in $cln(\mathcal{T})$, then also $\exists R_1^- \sqsubseteq \neg B$ is in $cln(\mathcal{T})$;
5. if $R_1 \sqsubseteq R_2$ is in $\mathcal{T}$ and $R_2 \sqsubseteq \neg R_3$ or $R_3 \sqsubseteq \neg R_2$ is in $cln(\mathcal{T})$, then also $R_1 \sqsubseteq \neg R_3$ is in $cln(\mathcal{T})$;
6. if one of the assertions $\exists R \sqsubseteq \neg \exists R$, $\exists R^- \sqsubseteq \neg \exists R^-$ or $R \sqsubseteq \neg R$ is in $cln(\mathcal{T})$, then all three such assertions are in $cln(\mathcal{T})$.

To fully understand the NI-closure we have to consider $can(\mathcal{K})$, the *canonical interpretation* of $\mathcal{K}$. We can see $can(\mathcal{K})$ as an application of PIs on the ABox. This this is done stepwise, creating new membership assertions out of PIs (see Definition 62).

Definition 61. The function ga is defined as follows:

$$ga(R, a, b) = \begin{cases} P(a, b), & if\ R = P \\ P(b, a), & if\ R = P^- \end{cases}$$

where R is a basic role, a and b are constants, and the result P is a membership assertion.

Definition 62. Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a *DL-Lite$_R$* KB, let $\mathcal{T}_p$ be the set of PI assertions in $\mathcal{T}$. Let n be the number of membership assertions in $\mathcal{A}$, where the membership assertions are numbered from 1 to n according to their lexicographic order. Let Γ_N be the set of constants defined above. Consider next the following definition:

- $S_0 = \mathcal{A}$,
- $S_{j+1} = S_j \cup \{f_{new}\}$, where f_{new} is a membership assertion numbered with $n+j+1$ in S_{j+1} and obtained as follows:
  Let f be the first membership assertion in S_j such that there exists a PI $\alpha \in \mathcal{T}_p$ applicable in S_j to f; let α be the lexicographically first PI applicable in S_j to f; and let α_{new} be the constant of Γ_N that follows lexicographically all constants occurring in S_j.
  Case α, f of
  (cr1) $\alpha = A_1 \sqsubseteq A_2$, $f = A_1(a)$ then $f_{new} = A_2(a)$;
  (cr2) $\alpha = A \sqsubseteq \exists R$ and $f = A(a)$ then $f_{new} = ga(R, a, a_{new})$;
  (cr3) $\alpha = \exists R \sqsubseteq A$ and $f = ga(R, a, b)$ then $f_{new} = A(a)$;
  (cr4) $\alpha = \exists R_1 \sqsubseteq \exists R_2$ and $f = ga(R_1, a, b)$ then $f_{new} = ga(R_2, a, a_{new})$;
  (cr5) $\alpha = R_1 \sqsubseteq R_2$ and $f = ga(R_1, a, b)$ then $f_{new} = ga(R_2, a, b)$.

Then, we define *chase of* $\mathcal{K}$, denoted $chase(\mathcal{K})$, as follows: $chase(\mathcal{K}) = \bigcup_{j \in \mathbb{N}} S_j$.

Definition 63. Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a *DL-Lite$_R$* KB. We define the *canonical interpretation* $can(\mathcal{K})$ as the interpretation $\langle \Delta^{can(\mathcal{K})}, \cdot^{can(\mathcal{K})} \rangle$, where:

(i) $\Delta^{can(\mathcal{K})}$ is the set consisting of all constant symbols in $\mathcal{A}$,

(ii) $a^{can(\mathcal{K})} = a$, for each constant a occurring in $chase(\mathcal{K})$,

(iii) $A^{can(\mathcal{K})} = \{a \mid A(a) \in chase(\mathcal{K})\}$, for each atomic concept A, and

(iv) $P^{can(\mathcal{K})} = \{(a_1, a_2) \mid P(a_1, a_2) \in chase(\mathcal{K})\}$, for each atomic role P .

Lemma 64. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a DL-Lite$_R$ KB and let $\mathcal{T}_p$ be the set of positive inclusion assertions in $\mathcal{T}$. Then, $can(\mathcal{K})$ is a model of $\langle \mathcal{T}_p, \mathcal{A} \rangle$.*

Proof. See Lemma 7 in [Calvanese et al., 2007]. □

Lemma 65. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a DL-Lite$_R$ KB. Then, $can(\mathcal{K})$ is a model of $\mathcal{K}$ iff* $db(\mathcal{A})$ is a model of $\langle cln(\mathcal{T}), \mathcal{A} \rangle$.

Proof. See Lemma 12 in [Calvanese et al., 2007]. □

Now the algorithm `Consistent(`$\mathcal{K}$`)` can be introduced. It computes $db(\mathcal{A})$ and $cln(\mathcal{T})$, then evaluates over $db(\mathcal{A})$ the union of all FOL formulas as a Boolean FOL query. The FOL formulas are created by the following function δ.

Definition 66. Translation function δ rewrites assertions of $cln(\mathcal{T})$ to FOL formulas as follows:

(i) $\delta(B_1 \sqsubseteq \neg B_2) = \exists x.\gamma_1(x) \wedge \gamma_2(x)$,

(ii) $\delta(R_1 \sqsubseteq \neg R_2) = \exists x, y.\rho_1(x, y) \wedge \rho_2(x, y)$,

where
$\gamma_i(x) = A_i(x)$ if $B_i = A_i$,
$\gamma_i(x) = \exists y_i.P_i(x, y_i)$ if $B_i = \exists P_i$,
$\gamma_i(x) = \exists y_i.P_i(y_i, x)$ if $B_i = \exists P_i^-$,

and
$\rho_i(x, y) = P_i(x, y)$ if $R_i = P_i$,
$\rho_i(x, y) = P_i(y, x)$ if $R_i = P_i^-$.

Algorithm 2.1 Consistent

```
Input:  DL-Lite_R KB K = ⟨T, A⟩
Result:  true, if K is satisfiable, false otherwise
q_unsat ← ⊥;
foreach α ∈ cln(T) do
  q_unsat ← q_unsat ∨ δ(α);
end
if q_unsat^db(A) = ∅ then
  return true;
else
  return false;
end
```

Lemma 67. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be DL-Lite$_R$ KB. Then, algorithm `Consistent(`$\mathcal{K}$`)` terminates, and $\mathcal{K}$ is satisfiable, iff `Consistent(`$\mathcal{K}$`)` $= true$.*

Proof. See Lemma 16 in [Calvanese et al., 2007]. □

Lemma 68. *Knowledge base satisfiability in DL-Lite$_R$ is FOL-reducible.*

Proof. A direct consequence of Lemma 67. □

2.3.5. Query Answering over $DL\text{-}Lite_R$ Ontologies

Query answering in $DL\text{-}Lite_R$ is realized in two steps. First, the TBox axioms are rewritten into the main query, which results in a union of queries. Second, the result of the first step is evaluated over the ABox. For reformulating query q the function $gr(g, I)$ is central and used in Algorithm 2.2.

Definition 69. Let I be an inclusion assertion that is applicable to atom g. Then, $gr(g, I)$ rewrites g as follows:

1. if $g = A(x)$ and $I = A_1 \sqsubseteq A$, then $gr(g, I) = A_1(x)$;
2. if $g = A(x)$ and $I = \exists P \sqsubseteq A$, then $gr(g, I) = P(x, \_)$;
3. if $g = A(x)$ and $I = \exists P^- \sqsubseteq A$, then $gr(g, I) = P(\_, x)$;
4. if $g = P(x, \_)$ and $I = A \sqsubseteq \exists P$, then $gr(g, I) = A(x)$;
5. if $g = P(x, \_)$ and $I = \exists P_1 \sqsubseteq \exists P$, then $gr(g, I) = P_1(x, \_)$;
6. if $g = P(x, \_)$ and $I = \exists P_1^- \sqsubseteq \exists P$, then $gr(g, I) = P_1(\_, x)$;
7. if $g = P(\_, x)$ and $I = A \sqsubseteq \exists P^-$, then $gr(g, I) = A(x)$;
8. if $g = P(\_, x)$ and $I = \exists P_1 \sqsubseteq \exists P^-$, then $gr(g, I) = P_1(x, \_)$;
9. if $g = P(\_, x)$ and $I = \exists P_1^- \sqsubseteq \exists P^-$, then $gr(g, I) = P_1(\_, x)$;
10. if $g = P(x_1, x_2)$ and either $I = P_1 \sqsubseteq P$ or $I = P_1^- \sqsubseteq P^-$, then $gr(g, I) = P_1(x_1, x_2)$;
11. if $g = P(x_1, x_2)$ and either $I = P_1 \sqsubseteq P^-$ or $I = P_1^- \sqsubseteq P$, then $gr(g, I) = P_1(x_2, x_1)$.

Similar to some dialects in Datalog, the symbol underscore denotes an *non-distinguished, non-shared variable* and shows that an argument is *unbound*. Particularly function τ and *reduce* in Algorithm 2.2 make use of unbound variables. Function τ replaces all unbound variables in a conjunctive query with underscores. Function *reduce* calculates the *most general unifier* (mgu) of the atoms g_1 and g_2 in a conjunctive query. Note that by unifying g_1 and g_2, each underscore symbol in g_1 is replaced with the corresponding argument of g_2 and vice-versa.

Algorithm 2.2 PerfectRef

```
Input:  Conjunctive query q, DL-Lite_R TBox 𝒯
Result:  Union of conjunctive queries PR
PR ← {q};
repeat
  PR′ ← PR;
  foreach query q ∈ PR′ do
    /* Step (a) */
    foreach atom g in q do
      foreach PI I in 𝒯 do
        if I is applicable to g then PR ← PR ∪ {q[g/gr(g, I)]};
      end
    end
    /* Step (b) */
    foreach atom g1, g2 in q do
      if g1 and g2 unify then PR ← PR ∪ {τ(reduce(q, g1, g2))};
    end
  end
until PR′ = PR
return PR;
```

Lemma 70. *Let $\mathcal{T}$ be $DL\text{-}Lite_R$ TBox, and let q be a conjunctive query over $\mathcal{T}$. Then, the algorithm* `PerfectRef(`$q, \mathcal{T}$`)` *terminates.*

Proof. See Lemma 34 in [Calvanese et al., 2007]. □

The second step of query answering is simple. Algorithm 2.3 computes the answer for a union of conjunctive queries over a $DL\text{-}Lite_R$ KB. Furthermore algorithm `Consistent(`$\mathcal{K}$`)` is used to determine whether a KB is satisfiable; if not, all tuples of constants are returned.

Algorithm 2.3 Answer

```
Input:  Union of conjunctive queries Q, DL-Lite_R KB 𝒦 = ⟨𝒯, 𝒜⟩
Result:  set of tuples ans(Q, 𝒦)
if not Consistent(𝒦) then
  return AllTup(Q, 𝒦);
else
  return ( ⋃_{q_i ∈ Q} PerfectRef(q_i, 𝒯))^{db(𝒜)};
end
```

Lemma 71. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a $DL\text{-}Lite_R$ KB, and let Q be a union of conjunctive queries. Then, the algorithm* `Answer(`$Q, \mathcal{K}$`)` *terminates.*

Proof. See Lemma 37 in [Calvanese et al., 2007]. □

The correctness of `Answer`$(Q, \mathcal{K})$is illustrated by the following theorem:

Theorem 72. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a DL-Lite$_R$ KB, Q be a union of conjunctive queries, and $\vec{t}$ a tuple of constants in $\mathcal{K}$. Then, $\vec{t} \in ans(Q, \mathcal{K})$ iff $\vec{t} \in Answer(Q, \mathcal{K})$. Therefore answering unions of conjunctive queries in DL-Lite$_R$ is FOL-reducible.*

Proof. See Theorem 40 and 41 in [Calvanese et al., 2007]. □

2.3.6. Complexity Results for *DL-Lite$_R$*

The complexity results are one of the main reasons for our interest in *DL-Lite$_R$*. The authors of [Calvanese et al., 2007] point out that the worst-case complexity of query answering is exponential in the size of the queries. This is unavoidable, because it is given by the complexity of relational DB query evaluation.

Theorem 73. *Answering unions of conjunctive queries in DL-Lite$_R$ is PTIME in the size of the TBox, and LOGSPACE in the size of the ABox (data complexity).*

Proof. See Theorem 43 in [Calvanese et al., 2007]. □

Theorem 74. *Answering unions of conjunctive queries in DL-Lite$_R$ is NP-complete in combined complexity.*

Proof. See Theorem 44 in [Calvanese et al., 2007]. □

2.4. Description Logic Programs

Introduced by [Eiter et al., 2004], dl-programs combine DL and normal programs under stable model semantics. Later they were extended in [Eiter et al., 2009b] to well-founded semantics. Due to the strict semantic seperation of the DL KB and logic program, dl-programs belong to the loose coupling approaches.

2.4.1. Syntax of Description Logic Programs

Definition 75. A *dl-program* consists of a $KB = (\mathcal{L}, P)$, where P denotes a generalization of a normal program as in Definition 22 and $\mathcal{L}$ a DL KB. The specification of $\mathcal{L}$ can be found in Definition 52.

Notice that the DL KB $\mathcal{L}$ could also be replaced with more expressive DLs, like $\mathcal{SHIF}(D)$ or $\mathcal{SHOIN}(D)$. In our case this is not desired, because the focus of this thesis is primarily on *DL-Lite*.

Definition 76. [Eiter et al., 2009b] To couple P and $\mathcal{L}$ we introduce the notion of a *dl-query* $Q(t)$, which is either:

(i) a concept inclusion assertions F or its negation $\neg F$, where t is empty; or

(ii) of the forms $C(x)$ or $\neg C(x)$, where C is a concept, and x is a term and equal to t; or

(iii) of the forms $R(x_1, x_2)$ or $\neg R(x_1, x_2)$, where R is a role, and x_1 and x_2 are terms and elements of argument list t; or

(iv) of the forms $= (x_1, x_2)$ or $\neq R(x_1, x_2)$, where x_1 and x_2 are terms and elements of argument list t.

Definition 77. [Eiter et al., 2009b] Extending Definition 22, we introduce a new type of atoms, called *dl-atom*. A dl-atom solely occurs in the rule body and has the form:

$$DL[S_1 op_1 p_1, ..., S_m op_m p_m; Q](t), m \geq 0,$$

where each S_i is either a concept or a role, $op_i \in \{\uplus, \cup\!\!\!\!-\}$, p_i is a unary resp. binary predicate symbol, and $Q(t)$ is a dl-query.

Roughly speaking, $p_1, ..., p_m$ are the input predicate symbols modifying the ABox of $\mathcal{L}$ by adding positive ($\uplus$) resp. negative ($\cup\!\!\!\!-$) assertion to the concepts or roles of $S_1, ..., S_m$. In [Eiter et al., 2004] a nomonotonic operator $\cap\!\!\!\!-$ is defined, however it is not considered in this thesis.

2.4.2. Well-Founded Semantics for Description Logic Programs

Eiter et al. generalized the well-founded semantics for ordinary programs to dl-programs [Eiter et al., 2009b]. They introduced the notion of unfounded set for dl-programs; first we need some preliminary definitions.

Definition 78. [Eiter et al., 2009b] Let $KB = (\mathcal{L}, P)$ be a dl-program and P be a normal program. We denote HB_P the *Herbrand base* of P, *ground*(P) the set of all ground instances in P and, Lit_P the set of all ground literals in P. A set of ground literals $S \subseteq Lit_P$ is *consistent* iff $S \cap \neg.S = \emptyset$, where $\neg.S = \{\neg.l \mid l \in S\}$. We call I a *(three-valued) interpretation* relative to P, where $I \subseteq Lit_P$.

Definition 79. [Eiter et al., 2009b] Let $I \subseteq Lit_P$ be consistent. A set $U \subseteq HB_P$ is an unfounded set of $KB = (\mathcal{L}, P)$ relative to I iff the following holds:

for every atom $a \in U$ and every rule $r \in ground(P)$ with $H(r) = a$, either

(i) $\neg b \in I \cup \neg.U$ for some ordinary atom $b \in B^+(r)$, or

(ii) $b \in I$ for some ordinary atom $b \in B^-(r)$, or

(iii) for some dl-atom $b \in B^+(r)$, it holds that $S^+ \not\models_{\mathcal{L}} b$ for every consistent $S \subseteq Lit_P$ with $I \cup \neg.U \subseteq S$, or

(iv) for some dl-atom $b \in B^-(r)$, $I^+ \models_{\mathcal{L}} b$.

From Definition 79 the first lemma can be derived.

Lemma 80. *[Eiter et al., 2009b] Let $KB = (\mathcal{L}, P)$ be a dl-program and let $I \subseteq Lit_P$ be consistent. Then, the set of unfounded sets of KB relative to I is closed under union.*

Proof. We refer to Lemma 4.5 in [Eiter et al., 2009b]. □

Definition 81. [Eiter et al., 2009b] The operators T_{KB}, U_{KB} and W_{KB} on all consistent $I \subseteq Lit_P$ are defined as follows:

(i) atom $a \in T_{KB}(I)$ iff $a \in HB_P$ and some rule $r \in ground(P)$ exists s.t.
(a) $H(r) = a$,
(b) $I^+ \models_{\mathcal{L}} b$. for all $b \in B^+(r)$,
(c) $\neg b \in I$ for all ordinary atoms $b \in B^-(r)$, and
(d) $S^+ \not\models_{\mathcal{L}} b$ for each consistent $S \subseteq Lit_P$ with $I \subseteq S$, for all dl-atoms $b \in B^-(r)$;

(ii) $U_{KB}(I)$ is the greatest unfounded set of KB relative to I; and

(iii) $W_{KB}(I) = T_{KB}(I) \cup \neg.U_{KB}(I)$.

Lemma 82. *[Eiter et al., 2009b] Let $KB = (\mathcal{L}, P)$ be a dl-program. Then T_{KB}, U_{KB} and W_{KB} are monotonic.*

Proof. We refer to Lemma 4.7 in [Eiter et al., 2009b]. □

Due to the monotonicity of operator W_{KB}, it has a least fixpoint. Based on the least fixpoint the well-founded semantics for dl-programs is defined as follows:

Definition 83. Let $KB = (\mathcal{L}, P)$ be a dl-program. The *well-founded semantics* of KB, denoted $WFS(KB)$, is defined as $lfp(W_{KB})$. An atom $a \in HB_P$ is *well-founded* (resp. *unfounded*) relative to KB iff a (resp. $\neg a$) belongs to $WFS(KB)$.

3. Combining Datalog with DL-Lite

In this chapter we present our approach of combining Datalog with $DL\text{-}Lite_R$. We first exhibit the rewriting of stratified Datalog to RA extended with fixpoint evaluation [Ullman, 1988]. Then, we introduce an improvement for query answering over $DL\text{-}Lite_R$ ontologies. This rewriting refinement produces nonrecursive Datalog queries for a $DL\text{-}Lite_R$ KB and conjunctive queries [Rosati and Almatelli, 2010]. Afterwards we highlight the results of [Eiter et al., 2009b], which show that acyclic dl-programs coupled with $DL\text{-}Lite_R$ are FO-rewritable. The combination of these results will lead to the evaluation of dl-programs under stratified Datalog. Finally we outline the straightforward conversion of RA to SQL, which is needed for the evaluation of dl-program using an RDBMS.

3.1. Rewriting Datalog to Relational Algebra extended with Fixpoint Evaluation

In Chapter 3 of [Ullman, 1988] the connection between Datalog and RA is illustrated. Ullman provides the necessary results and algorithms for rewriting nonrecursive Datalog into RA. Furthermore Ullman introduces a fixpoint evaluation on RA expressions to deal with recursion in Datalog. Based on his work, we give a step-wise introduction of the algorithms and results. First we show how to rewrite a single rule and a positive nonrecursive program into RA (Algorithm 3.1, 3.2, and 3.3). Second, we extend positive programs with recursion (Algorithm 3.4 and 3.5). Third, we integrate the handling of negation in the algorithms (Algorithm 3.6). Finally the algorithm for rewriting stratified Datalog programs is introduced (Algorithm 3.7).

3.1.1. Nonrecursive Datalog

Algorithm 3.1 is used in 3.3 for computing the RA expression for a single rule body.

Algorithm 3.1 EvalRule

```
Input: Rule body B(r), set of computed relations R
Result: Relational algebra expression I
sets Q ← E ← G ← ∅;
/* Handle terms appearing in body predicates (Section A) */
X ←set of appearing variables in B(r);
foreach predicate p ∈ B(r) do
  sets F_i ← V_i ← ∅;
  T_i ←set of terms in p;
  R_i ←relation for p in R;
  foreach variable k ∈ T_i do
    if another variable l ∈ T_i is the same as k then
      /* Selection of variables */
      add (position(k, R_i) = position(l, R_i)) to F_i;
    end
    add position(k, R_i) to V_i; /* Projection of variables */
  end
  foreach constant c ∈ T_i do
    add (position(c, R_i) = c) to F_i; /* Selection of constants */
  end
  add π_{V_i}(σ_{F_i}(R_i)) to Q;
end
/* Handle variables not appearing in body predicates (Section B) */
T ←set of all terms in B(r);
foreach variable x ∈ X such that x ∉ T do
  find a term y which is equated to x through a sequence of atoms;
  if y is a constant then add {y}(x) to Q;
  if y is a variable in R_i then add π_{position(y, R_i)}(R_i) to Q;
end
/* Build the RA natural joins of the expressions (Section C) */
foreach expression q ∈ Q do
  E ← E ⋈ q;
end
/* Handle build-in predicates (Section D) */
foreach predicate (X_B operator Y_B) ∈ B(r) do
  G ← G ∧ X_B operator Y_B;
end
I ← σ_G(E);
return I;
```

Lemma 84. *[Ullman, 1988] Algorithm 3.1 is correct, in the sense that the relation R produces has all and only those tuples $\{a_1, ..., a_m\}$ such that, when we substitute each a_j for X_j every subgoal S_i is made true.*

Proof. We refer to Theorem 3.1 in [Ullman, 1988]. □

Algorithm 3.2 rectifies rules to enforce constraints in the head predicate to the body predicates.

Algorithm 3.2 RectifyRules

```
Input: Datalog program P
Result: Rectified Datalog program P′
P′ ← P;
foreach rule r ∈ P′ do
  T ←set of terms in H(r);
  /* Handle constants in the head */
  foreach constant c ∈ T do
    c′ ← c;
    replace c with new variable x in H(r);
    B(r) ← B(r) ∧ (x = c′);
  end
  /* Handle identical variables in the head */
  foreach variable v ∈ T such that w ∈ T and v = w do
    v′ ← v;
    replace v with new variable x in H(r);
    B(r) ← B(r) ∧ (x = v′);
  end
end
return P′;
```

Lemma 85. *[Ullman, 1988] Algorithm 3.2 is rectifying each rule r in P into r' such that:*

(i) If r is safe, so is r';

(ii) Rules r and r' are equivalent, in the sense that, given relations for the predicates of their subgoals, there is substitution for the variables of r that makes all its subgoals true and makes the head become $p(c_1, ..., c_n)$ iff there is some substitution for the variables of r' that makes the head of r' become $p(c_1, ..., c_n)$.

Proof. See Lemma 3.1 in [Ullman, 1988]. □

In Algorithm 3.3 all parts are put together to evaluate an nonrecursive and positive program.

Algorithm 3.3 NonrecursivePositiveEval

```
Input:  Nonrecursive positive Datalog program P, set of EDB relations R_E
Result:  Set I of relational algebra expressions
set I ← ∅;
set R ← R_E;
P' ←RectifyRules(P);
/* Order of evaluation */
G ←dependency graph of P';
O ←topological sort of G;
/* Build the RA unions of all rules with the same predicate in the head */
foreach predicate p ∈ O do
  E ← ∅;
  foreach rule r ∈ P' such that p = H(r) do
    E_r ←EvalRule(B(r), R);
    X ←set of appearing variables in B(r);
    foreach relation q in E_r such that q ∉ R_E
      /* Replace already created atoms */
      replace q with related RA expressions e ∈ I;
    end
    E ← E ∪ π_X(E_r);
  end
  add p to R;
  add E to I;
end
return I;
```

Note that the symbol ∪ is purposely chosen as the RA union operator.

Theorem 86. *[Ullman, 1988] Algorithm 3.3 correctly computes for a positive nonrecursive Datalog program P the relation for each predicate, in the sense that the expression it constructs for each IDB predicate yields both:*

(i) The set of facts for that predicate that can be proved from the database, and

(ii) The unique minimal model of the rules.

Proof. We refer to Theorem 3.2 in [Ullman, 1988]. □

3.1.2. Positive Recursive Datalog

It is well known that conventional RA is not sufficient expressible to capture recursion in Datalog. We will show according to Ullman that recursion can be expressed with RA, if we extend RA with a fixpoint evaluation. Recursion was introduced in SQL with version SQL:1999. Unfortunately, the form of recursion in SQL:1999 is limited to linear recursion. Indirect recursion over several predicates is not covered by SQL:1999. Grosof et al. discuss this limitation in their paper on DLP and give some methods e.g. magic template procedure to overcome it [Grosof et al., 2003]. Note that DLP should not be mistaken with dl-programs, since only dl-programs support NAF.

Definition 87. Linear recursion can be formulated in Datalog as:

$$\begin{aligned} t(X,Y) &\leftarrow g(X,Y). \\ t(X,Y) &\leftarrow g(X,Z),\ t(Z,Y). \end{aligned}$$

where g is a graph and t its transitive closure [Abiteboul et al., 1995].

It is obvious that recursive rules will result in a cyclic dependency graph, hence a topological sort is impossible and Algorithm 3.3 is not suitable for this class of programs. A fixpoint evaluation overcomes the need of a dependency graph and gives a powerful machinery for handling recursion in general.

Ullman provides a "naive" fixpoint evaluation algorithm, which calculates a set of tuples [Ullman, 1988]. We extend in Algorithm 3.4 and 3.5 Ullman's version to calculate also the RA expressions. From an implementation perspective the RA expressions can be neglected, but we need the RA expressions to show that the merging of Datalog and DL-Lite is feasible.

Algorithm 3.4 FixpointEvalSub

```
Input: Predicate p, set of rules U, set of old tuples Q, set of EDB relations R
Result: Set of tuples T, relational algebra expressions I
set I ← ∅;
/* Bottom-up evaluation of rules for a predicate */
foreach rule r ∈ U such that p = H(r) do
  X ←set of appearing variables in B(r);
  E_r ←EvalRule(B(r), R);
  I ← I ∪ π_X(E_r);
end
T ←calculate result tuples for I incorporating Q;
return T,I;
```

Lemma 88. *[Ullman, 1988] The relational algebra operators union, Cartesian product, projection, and selection are monotone.*

Proof. We refer to Theorem 3.3 in [Ullman, 1988]. □

Lemma 89. *[Ullman, 1988] The relational algebra operators natural join and θ-join are monotone.*

Proof. Both operations are composites of the monotone operations defined in Lemma 88. □

Algorithm 3.5 NaiveFixpointEvaluation

```
Input: Recursive positive Datalog program P, set of EDB relations R, set of existing
tuples C
Result: Set of tuples T, set of relational algebra expressions I
set I ← ∅;
m ←count predicates in P;
/* Initialize tuple sets, add precalculated tuples */
foreach i ← 1 to m such that predicate p_i ∈ P do
  if C is empty then
    T_i ← ∅;
  else
    T_i ←get tuples from C for p_i;
  end
end
repeat
  /* Save old tuple sets */
  foreach i ← 1 to m do
    Q_i ← T_i;
  end
  /* Bottom-up evaluation of rules for prediactes */
  foreach i ← 1 to m such that predicate p_i ∈ P do
    U ←set of appearing rules in P;
    T_i, F_i ←FixpointEvalSub(p_i,U,Q_i,R);
    add F_i to I;
  end
  /* Stop if tuple sets are not altered by evaluation anymore */
until ∀_{i≤m}(Q_i = T_i)
T ← T_1 ∪ ... ∪ T_m;
return T, I;
```

Note, RA in general is non-monotonic due to the difference operator.

Lemma 90. *[Ullman, 1988] The operation* `FixpointEvalSub` *of algorithm 3.5 is monotone.*

Proof. By taking all RA operators used in algorithm 3.1 and 3.4, only union, Cartesian product, projection, selection, natural join, and θ-join are used. All of this operators are monotone, hence operation `FixpointEvalSub` is also monotone. □

Theorem 91. *[Ullman, 1988] Algorithm 3.5 produces the least fixpoint of a positive Datalog program, with respect to the given EDB relations.*

Proof. We refer to Theorem 3.4 in [Ullman, 1988]. □

3.1.3. Datalog with Negation

Again Ullman gives the intuitive idea, that a negated atom's predicate in a rule can be seen as the complement of a relation, in relation to a domain of possible values [Ullman, 1988].

Definition 92. [Ullman, 1988] Let r be a rule of the following form:

$a \leftarrow b,\ not\, c.$

where a, b, and c are atoms. An atom is defined as in Definition 8.

Then r can be rewritten in RA as:

$A(t_1, ..., t_n) = B(t_1, ..., t_n) - C(t_1, ..., t_n).$

There is one case not covered by this simple translation. A variable appearing only in the positive body predicate, but not in any negated predicate.

Definition 93. [Ullman, 1988] Let r be a rule of the following form:

$a \leftarrow b,\ not\, c.$

where a and b are defined as in Definition 92 with the exception that atom c is as $p(t_1, ..., t_m)$ and $n > m$.

Then r can be rewritten in RA as:

$A(t_1, ..., t_n) = B(t_1, ..., t_n) - (C(t_1, ..., t_m) \times \pi_{t_{m+1},...,t_n}(B(t_1, ..., t_n))).$

To rewrite a single negated atom, section C in Algorithm 3.1 has to be replaced with Algorithm 3.6, however we do not yet consider Definition 93 for our algorithm.

Algorithm 3.6 EvalNegativeRule

```
foreach expression q ∈ Q do
  if literal t for q is negative then
    E ← E − q;
  else
    E ← E ⋈ q;
  end
end
```

Minimal model semantics of positive programms do not suffice to capture negation in Datalog. According to [Ceri et al., 1990] there are two semantics for dealing with negation. Namely the approaches are stratified evaluation of Datalog (*stratified Datalog*) and *inflationary semantics* for Datalog. We will focus solely on stratified Datalog.

3.1.4. Stratified Datalog

Algorithm 3.7 is based on the concept of splitting a program P into strata, which are evaluated sequently as subprograms. The result of this algorithm reflects two purposes. Namely, the tuples represent the supported minimal model of the program and the RA expressions represent a sequence of algebraic expressions.

Algorithm 3.7 StratifiedEvaluation

```
Input:  Stratified Datalog program P, set of EDB relations R
Result:  Set of tuples T, set of relational algebra expressions I
I ← ∅;
T ← ∅;
O ←stratification of P;
foreach strata p ∈ O do
  /* Fixpoint evaluation with tuples from last strata */
  T_p, F_p ←NaiveFixpointEvaluation(p, R, T);
  add F_p to I;
  T ← T ∪ T_p;
end
return T, I;
```

Theorem 94. *Algorithm 3.7 correctly computes the supported minimal model of a stratified Datalog program, with respect to its EDB relations.*

Proof. (Sketch) Let $P_1, ..., P_n$ be the stratification of P. We define $M_1, ..., M_n$ as the sequence of minimal models relating to the strata $P_1, ..., P_n$ as follows:

$M_1 = lfp(T_{P_1 \cup EDB})$;
$M_2 = lfp(T_{P_2 \cup M_1})$;
...;
$M_n = lfp(T_{P_n \cup M_{n-1}}) = M_P$,

where M_P is the supported minimal model by Theorem 44. The existence of a fixpoint for a stratum P_i is given by Theorem 91, which shows that `NaiveFixpointEvaluation` produces the least fixpoint for $T_{P_i \cup M_{i-1}}$. □

3.2. An Algorithm for Improving Query Answering over $DL\text{-}Lite_R$ Ontologies

The perfect reformulation of conjunctive queries as shown in Algorithm 2.2 and the storage of the ABox in an RDBMS allows to process very large $DL\text{-}Lite_R$ ABoxes. However the authors of [Rosati and Almatelli, 2010] point out that there is a serious bottleneck in the algorithm. Namely, the computed perfect reformulation of conjunctive queries increases exponentially with the number of atoms in the queries. They refer to empirical studies, which show that queries with more than 5-7 atoms lead to FOL queries too large to be handled by current RDBMS (e.g. a union of thousands of conjunctive queries).

In [Rosati and Almatelli, 2010] they introduced the algorithm `Presto(`$Q, \mathcal{T}$`)` to overcome the above mentioned limitation. See Algorithm 3.8 for a detailed description. In `Presto(`$Q, \mathcal{T}$`)`, instead of a union of conjunctive queries a nonrecursive Datalog query is generated. Employing this technique, the exponential blow-up by using the disjunctive normal form is avoided.

First, we need to address a few functions used in `Presto(`$Q, \mathcal{T}$`)` [Rosati and Almatelli, 2010]:

- Function $Rename(Q)$ replaces every role $R(t, t_1)$ (resp. concept $A(t)$) of query Q by a new ontology-annotated predicate $p^2_R(t, t_1)$ (resp. $p^1_A(t)$).
- By introducing new predicates with lower arity function $DeleteUnboundVars(Q)$ is used to eliminate unbound variables of query Q in a systematic way.
- $DeleteRedundantAtoms(Q, \mathcal{T})$ eliminates redundant atoms of query Q taking inclusion assertions of the TBox $\mathcal{T}$ into account. Three of several elimination rules are, where rule $r \in Q$:
  - If $p^2_R(t_1, t_2)$ and $p^2_S(t_1, t_2)$ occur in r and $\mathcal{T} \models R \sqsubseteq S$, then eliminate $p^2_S(t_1, t_2)$ from r ;
  - If $p^1_B(t)$ and $p^1_C(t)$ occur in r and $\mathcal{T} \models B \sqsubseteq C$, then eliminate $p^1_C(t)$ from r ;
  - If $p^1_B(t)$ and p^0_α occur in r and $\mathcal{T} \models B^0 \sqsubseteq \alpha^0$, then eliminate p^0_α from r ;
- In function $Split(Q)$, the body of every rule in Q is split into a subset of atoms connected by bound join variables. For every subset a new rule with an auxiliary predicate in the head and the subset of atoms in the body is created.
- Function $EliminateEJVar(r, x, \mathcal{T})$ handles a sequence of resolution steps taken from `PerfectRef(`$q, \mathcal{T}$`)` and *Requiem* [Pérez-Urbina et al., 2009]. This function implements a crucial optimization for the *reduce* rule of `PerfectRef(`$q, \mathcal{T}$`)`. This is done by using the *most general subsumees* of concept and role expressions with respect to the TBox, hence useless unifications are avoided.

We have a closer look at function $DefineAtomView(V, \mathcal{T})$, because we will need this definition later.

Definition 95. [Rosati and Almatelli, 2010] Let $\mathcal{T}$ be a *DL-Lite*$_R$ TBox and let V be an ontology-annotated predicate. Then, the function $DefineAtomView(V, \mathcal{T})$ is defined as follows:

(i) if $V = p^2_R$ with R a role name, then the following set of rules is defined

$$\left\{p^2_R(x, y) \leftarrow P(x, y) \,|\, P \textit{ is a role name and } \mathcal{T} \models P \sqsubseteq R\right\} \cup$$
$$\left\{p^2_R(x, y) \leftarrow P(y, x) \,|\, P \textit{ is a role name and } \mathcal{T} \models P^- \sqsubseteq R\right\};$$

(ii) if $V = p^1_B$ with B a basic concept, then the following set of rules is defined

$$\left\{p^1_B(x) \leftarrow A(x) \mid A\,is\,a\,concept\,name\,and\,\mathcal{T} \models A \sqsubseteq B\right\} \cup$$
$$\left\{p^1_B(x) \leftarrow R(x,\_) \mid R\,is\,a\,role\,name\,and\,\mathcal{T} \models \exists R \sqsubseteq B\right\} \cup$$
$$\left\{p^1_B(x) \leftarrow R(\_,x) \mid R\,is\,a\,role\,name\,and\,\mathcal{T} \models \exists R^- \sqsubseteq B\right\};$$

(iii) if $V = p^0_N$ with N a concept or role name, then the following set of rules is defined

$$\left\{p^0_N \leftarrow A(\_) \mid A\,is\,a\,concept\,name\,and\,\mathcal{T} \models A^0 \sqsubseteq N^0\right\} \cup$$
$$\left\{p^0_N \leftarrow R(\_,\_) \mid R\,is\,a\,role\,name\,and\,\mathcal{T} \models R^0 \sqsubseteq N^0\right\};$$

Algorithm 3.8 Presto

```
Input: Union of conjunctive queries Q, DL-Lite_R TBox T
Result: Nonrecursive Datalog query Q'
Q' ← Rename(Q);
Q' ← DeleteUnboundVars(Q');
Q' ← DeleteRedundantAtoms(Q', T);
Q' ← Split(Q');
repeat
  if there exist r ∈ Q' and existential-join-var x in r such that
    Eliminable(x, r, T) = true and x has not already been eliminated from r then
      Q'' ← EliminateEJVar(r, x, T);
      Q'' ← DeleteUnboundVars(Q'');
      Q'' ← DeleteRedundantAtoms(Q'', T);
      Q' ← Q' ∪ Split(Q'');
  end
until Q' has reached a fixpoint
foreach ontology-annotated predicate p^n_α occurring in Q' do
  Q' ← Q' ∪ DefineAtomView(p^n_α, T);
end
return Q';
```

Theorem 96. *[Rosati and Almatelli, 2010] Let $\mathcal{T}$ be a DL-Lite$_R$ TBox, let Q be an union of conjunctive queries, and let Q' be the nonrecursive Datalog query returned by* `Presto`$(Q,\mathcal{T})$*. Then, for every ABox $\mathcal{A}$ such that $\langle\mathcal{T},\mathcal{A}\rangle$ is a satisfiable DL-Lite$_R$ KB, $\langle\mathcal{T},\mathcal{A}\rangle \models Q$ iff Q' is satisfied in can$(\mathcal{K})$.*

Proof. See Theorem 2 in [Rosati and Almatelli, 2010]. □

Example 97. [Rosati and Almatelli, 2010] Consider the *DL-Lite*$_R$ TBox $\mathcal{T}$, where A, A_1, B, C are concepts and P, R, S, T are roles:

$$\begin{array}{llll}
\mathcal{T}: & A \sqsubseteq A_1 & \exists T \sqsubseteq \exists S & T \sqsubseteq R \\
& A_1 \sqsubseteq B & \exists T^- \sqsubseteq \exists P & T \sqsubseteq R^- \\
& \exists R \sqsubseteq \exists U & \exists T^- \sqsubseteq \exists A_1 & T^- \sqsubseteq S \\
& \exists R \sqsubseteq A & \exists U \sqsubseteq \exists C & T \sqsubseteq P^- \\
& \exists R^- \sqsubseteq A & \exists U^- \sqsubseteq \exists P & U \sqsubseteq S^- \\
& \exists R \sqsubseteq \exists U & T \sqsubseteq P & U \sqsubseteq T^-
\end{array}$$

Let q be a conjunctive query as follows:

$q(y) \leftarrow T(x,w),\ R(y,w),\ A_1(z).$

Applying `Presto(`$q,\mathcal{T}$`)` we will get the following Datalog program:

$$\begin{array}{ll}
(R0) & q(y) \leftarrow q_1(y). \\
(R1) & q_1(y) \leftarrow p^1_{\exists T^-}(w),\ p^2_R(y,w). \\
(R2) & q_1(y) \leftarrow p^1_{\exists T^-}(y). \\
(R3) & q_1(y) \leftarrow p^1_{\exists U}(y).
\end{array}$$

and

$$\begin{array}{lll}
p^1_{\exists T^-}(x) \leftarrow U(x,_). & p^1_{\exists U}(x) \leftarrow T(x,_). & p^2_R(x,y) \leftarrow U(x,y). \\
p^1_{\exists T^-}(x) \leftarrow U(_,x). & p^1_{\exists U}(x) \leftarrow U(x,_). & p^2_R(x,y) \leftarrow T(x,y). \\
p^1_{\exists T^-}(x) \leftarrow R(x,_). & p^1_{\exists U}(x) \leftarrow R(x,_). & p^2_R(x,y) \leftarrow R(x,y). \\
p^1_{\exists T^-}(x) \leftarrow T(x,_). & p^1_{\exists U}(x) \leftarrow U(_,x). & p^2_R(x,y) \leftarrow U(y,x). \\
p^1_{\exists T^-}(x) \leftarrow T(_,x). & p^1_{\exists U}(x) \leftarrow T(_,x). & p^2_R(x,y) \leftarrow U(y,x).
\end{array}$$

3.3. First-Order Rewritable Case of Description Logic Programs

In [Eiter et al., 2009b] the authors show, that computing the general complexity of well-founded semantics for dl-programs over the DL $\mathcal{SHIF}(D)$ is EXPTIME, furthermore deciding whether a literal $l \in WFS(KB)$ holds is EXPTIME-complete.

We focus our work primarily on RDMBS, therefore data complexity is more interesting. Considering this for a dl-program $KB = (\mathcal{L}, P)$, only the facts in P and membership assertion in $\mathcal{L}$ do vary. By choosing $Horn\text{-}\mathcal{SHIQ}$ [Hustadt et al., 2005], reasoning and conjunctive query answering in PTIME under data complexity is feasible [Eiter et al., 2008a]. At this point the data complexity is the same as with ordinary normal programs under well-founded semantics.

Definition 98. Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a DL KB and $MS(\mathcal{K})$ the set of membership assertions. The *closed world assumption (CWA)* of $\mathcal{K}$ is defined as:

$CWA(\mathcal{K}) = \{\neg p \mid p \in MS(\mathcal{K})\}.$

CWA-satisfiable of $\mathcal{K}$ is defined by the entailment relation $\vDash_{cwa}$ for any membership or inclusion assertion α:

$\mathcal{K} \vDash_{cwa} \alpha$ iff $\mathcal{K} \cup CWA(\mathcal{K}) \vDash_{cwa} \alpha.$

Theorem 99. *[Eiter et al., 2009b] Given $KB = (\mathcal{L}, P)$ and a literal $l \subseteq Lit_P$, where every dl-atom in P can be evaluated in* PTIME *, deciding whether $l \in WFS(KB)$ is complete for* PTIME *under data complexity.*

Proof. See Theorem 7.2 in [Eiter et al., 2009b]. □

As mentioned in Theorem 73, answering unions of conjunctive queries in $DL\text{-}Lite_R$ is LOGSPACE in the size of the ABox. $DL\text{-}Lite_R$ data complexity and query answering capabilities makes it a plausible choice to couple with dl-programs. Hence the authors of [Eiter et al., 2009b] introduce the concept of *first-order rewritability*, where a dl-query can be rewritten into a FOL formula over the ABox. To achieve first-order rewritability for a normal program P, they restrict P to be acyclic and rewrite ordinary predicates of P to FOL formulas.

Notice, that (i), (ii) and, (iii) of Definition 76 can be expressed as a conjunctive query. We recall the Theorems 7.3 and 7.4 of [Eiter et al., 2009b] and introduce some minor adaptions.

Theorem 100. *[Eiter et al., 2009b] Let $KB = (\mathcal{L}, P)$ be an acyclic dl-program and a literal $l \in Lit_P$, where*

(i) every dl-query in P is first-order rewritable, and

(ii) if the operator $\uplus$ occurs in P, then $\mathcal{L}$ is defined over a DL, that

(ii.a) is CWA-satisfiable, and

(ii.b) allows for first-order rewritable concept and role membership, deciding whether $l \in WFS(KB)$ is first-order rewritable.

Proof. Since KB is acyclic, there is an acyclic dependency graph G_P. We derive from G_P the strict partial order of G_P the mapping of predicate symbols $\mathcal{K} : \mathcal{P}_P \rightarrow \{0, 1, ..., n\}$. We call $\mathcal{K}(p)$ the *rank* of p. By Theorem 72, every dl-query in P can be expressed in terms of a FOL formula over the set A of all membership and inclusion assertions in $\mathcal{L}$. We now show by induction on $\mathcal{K}(p) \in \{0, 1, ..., n\}$ that each predicate symbol $p \in \mathcal{P}_P$ can be expressed in terms of a FOL formula over the set F of all membership and inclusion assertions in $\mathcal{L}$ and the $EDB(P)$.

Basis: Each predicate $p \in \mathcal{P}_P$ of rank 0 can trivially be expressed in terms of a FOL formula over F.

Induction: We have to consider the evaluation of a dl-atom $DL[\lambda; Q_{Lite}](c)$ and the definition of a predicate $p \in \mathcal{P}_P$ via the set of all rules in P with $p \in H(r)$:

(i) Consider the dl-atom $DL[\lambda; Q_L](c)$ with $\lambda = \lambda^+, \lambda^-$, where $\lambda^+ = S_1 \uplus p_1, ..., S_l \uplus p_l$, $\lambda^- = S_{l+1} \uplus p_{l+1}, ..., S_m \uplus p_m$, and $m \geq l \geq 0$. The dl-query $Q_L(c)$ can be expressed in terms of a FOL formula $\alpha(x)$ over A, that is, $\mathcal{L} \models Q_L(c)$ iff $I_A \models \alpha(c)$. Since the underlying DL allows first-order rewritable membership and inclusion assertions, every

S_i in λ^-, $l < i \leq m$, can be expressed in terms of a FOL formula $\psi_{S_i}(y)$ over A, that is, $\mathcal{L} \models S_i(c)$ iff $I_A \models \psi_{S_i}(c)$ for every c. By the induction hypothesis, every input predicate p_j in λ can be expressed in terms of a FOL formula $\psi_j(x)$ over F, that is, $p_j(c) \in WFS(KB)$ iff $I_F \models \psi_j(c)$. We define the FOL formula $\delta(x)$ for $DL[\lambda; Q_L](x)$ over F as follows:

$$\delta(x) = \alpha^{\lambda^+}(x) \vee \bigvee_{j=l+1}^{m} \exists y(\psi_{S_j}^{\lambda^+}(y) \wedge \psi_j(y)),$$

where β^{λ^+} is obtained from β by replacing every $S_i(s)$ such that S_i occurs in λ^+ by $S_i(s) \vee \psi_{i_1}(s) \vee ... \vee \psi_{i_{k_i}}(s)$, where $S_{i_1}, ..., S_{i_{k_i}}$ are all occurrences of S_j in λ^+.

Note that $I_F \models S_i(c)$ iff $I_F \models S_i(c) \in \mathcal{L}$, for all $1 \leq i \leq l$. Hence,

$I_F \models S_i(c) \vee \psi_{i_1}(c) \vee ... \vee \psi_{i_{k_i}}(c)$

iff $S_i(c) \in \mathcal{L}$ or $p_{i_j}(c) \in WFS(KB)$, for some $1 \leq j \leq k_i$

iff $S_i(c) \in \mathcal{L} \cup \bigcup_{i=1}^{l} A_i(WFS(KB))$

iff $I_{A'} \models S_i(c)$, where $A' = A \cup \bigcup_{i=1}^{l} A_i(WFS(KB))$.

It follows from this that

$I_F \models \alpha^{\lambda^+}(c)$ iff $I_{A'} \models \alpha(c)$ and $I_F \models \psi_{S_j}^{\lambda^+}(c)$ iff $I_{A'} \models \psi_{S_j}(c)$, for all $l < j \leq m$.

This in turn implies that

$I_F \models \delta(c)$ iff (i) $\mathcal{L} \cup A' \models Q_L(c)$, or (ii) $\mathcal{L} \cup A' \models S_j(d)$ and $p_j(d) \in WFS(KB)$, for some $l < j \leq m$ and d.

Let $A'' = A' \cup \bigcup_{j=l+1}^{m} A_j(WFS(KB))$. If $\mathcal{L} \cup A'' \not\models Q_L(c)$, then clearly both (i) and (ii) are false; conversely, if $\mathcal{L} \cup A' \not\models Q_L(c))$ and $\mathcal{L} \cup A' \not\models S_j(d)$ for every $p_j(d) \in WFS(KB)$ where $l < j \leq m$, then $\mathcal{L} \cup A'' \not\models Q_L(c)$ holds since the underlying DL is CWA-satisfiable.

In summary, this shows that $I_F \models \delta(c)$ iff $\mathcal{L} \cup A'' \models Q_L(c)$ iff $WFS(KB)$ satisfies $DL[\lambda; Q_L](c)$. That is, $\delta(x)$ is a FOL formula for $DL[\lambda; Q](x)$ over F.

(ii) Consider next the set of all rules in P with $p \in H(r)$. W.l.o.g., the heads $p(x)$ of all these rules coincide. Let $\alpha(x)$ denote the disjunction of the existentially quantified bodies of these rules, where the default negations in the rule bodies are interpreted as classical negations. By the induction hypothesis, every body predicate in $\alpha(x)$ can be expressed in terms of a FOL formula over F, and the same holds for every dl-atom in $\alpha(x)$. Let $\alpha'(x)$ be obtained from $\alpha(x)$ by replacing all but the predicates of rank 0 by these FOL formulas. Then $\alpha'(x)$ is a FOL formula over F for p. $\square$

Example 101. Consider the *DL-Lite*$_R$ TBox $\mathcal{T}$, where A, A_1, B, C are concepts and P, R, S, T are roles:

Theorem 102. *[Eiter et al., 2009b] Let $KB = (\mathcal{L}, P)$ be an acyclic dl-program and a literal $l \in Lit_P$, where*

(i) $\mathcal{L}$ is defined in the description logic $DL\text{-}Lite_R$, and

(ii) all dl-queries in P are membership or inclusion assertions in $\mathcal{L}$, where concepts and roles are atomic.

Then deciding whether $l \in WFS(KB)$ is first-order rewritable.

Proof. We apply Theorem 67, 72, and 100. Observe that in $DL\text{-}Lite_R$ KB satisfiability and query answering are FOL-reducible, where FOL-reducible corresponds to first-order rewritable. Observe also that $DL\text{-}Lite_R$ is a CWA-satisfiable DL [Calvanese et al., 2007], and thus Theorem 100 also allows operator $\cup\!\!\!\!-$ to occur in P. All dl-atoms with dl-queries of the form $C(t)$ and $R(t_1, t_2)$ are immediately first-order rewritable. Dl-queries of the form $C \sqsubseteq D$ resp. $\neg(C \sqsubseteq D)$ can be reduced to conjunctive queries as follows: $L' \models C \sqsubseteq D$ iff $L' \cup \{C(e), D'(e), D' \sqsubseteq \neg D, A'(d), A' \sqsubseteq \neg A\} \models A(d)$ resp. $L' \models \neg(C \sqsubseteq D)$ iff $L' \cup \{C' \sqsubseteq D, A'(d), A' \sqsubseteq \neg A\} \models A(d)$, where d and e are fresh individuals, and A, A', and D' are fresh atomic concepts. By Theorem 100, it thus follows that deciding whether $l \in WFS(KB)$ is first-order rewritable. $\square$

Example 103. [Eiter et al., 2009b] Consider the DL KB $\mathcal{L} = \langle \mathcal{T}, \mathcal{A} \rangle$, where C is a concept in $\mathcal{T}$ and $C(a)$ is an membership assertion in $\mathcal{A}$. Let $(\mathcal{L}, P)$ be a dl-program as follows:

$p(c)$. $q(b)$.
$r \leftarrow p(x)$.
$r \leftarrow DL\,[C \uplus p; C]\,(x)$.
$s \leftarrow not\ DL\,[C \uplus p, C \cup\!\!\!\!-\, q; C]\,(x)$.

The dl-program $(\mathcal{L}, P)$ can be expressed by the following formulas:

- The query $Q(x)$ is expressed by $\alpha(x) = C(x)$ over $\mathcal{A} = \{C(a)\}$;
- the predicates p and q as $\psi_p(x) = p(x)$ and $\psi_q(x) = q(x)$ over $F = \{C(a), p(c), q(b)\}$;
- the dl-atom $DL\,[C \uplus p; C]\,(x)$ is translated into $\delta_1(x) = \alpha^{\lambda^+}(x) = C(x) \vee p(x)$ over F (note that $m = l$);
- the dl-atom $DL\,[C \uplus p, C \cup\!\!\!\!-\, q; C]\,(x)$ is expressed by $\delta_2(x) = (C(x) \vee p(x)) \vee \exists y((C(y) \vee p(y)) \wedge q(y))$ over F.

3.4. Stratified Evaluation of Description Logic Programs

By Theorem 100 and 102, we have shown that acyclic dl-programs can be coupled with $DL\text{-}Lite_R$. Next, we extend dl-programs to be evaluated under stratified Datalog. The evaluation is split into a preprocessing part and a standard stratified evaluation part. The first part is concerned with the rewriting of the dl-atoms by compiling the $DL\text{-}Lite_R$ KB and the conjunctive query into Datalog. Then, the adapted dl-program can be rewritten into RA expressions extended with fixpoint evaluation. In Algorithm 3.9 we give a detailed description.

Algorithm 3.9 dl-program Evaluation

```
Input:  Stratified dl-program (𝓛, P) with a DL-Lite_R KB 𝓛 = ⟨𝒯, 𝒜⟩,
        set of EDB relations R
Result:  Set of tuples T, set I of relational algebra expressions

/* Step (a):  Preprocessing ABox */
write ABox 𝒜 to EDB R;

/* Step (b):  Preprocessing dl-atoms */
for each dl-atom d ∈ P do
  q_f ← rewrite TBox 𝒯 and conjunctive query Q of d
    to a nonrecursive Datalog query with Presto(Q,𝒯);
  q_d ← rewrite query q_f to include DL update predicates;
  extend P with query q_d;
  replace dl-atom d in P with a new ordinary atom referring to q_d;
end

/* Step (c):  Datalog to RA */
set of RA expressions I=∅;
S = stratification of P;
foreach strata st ∈ S do
  /* Fixpoint evaluation with tuples from last strata */
  T_p, F_p ←NaiveFixpointEvaluation(st, R, T);
  add RA expressions F_p to I;
  T ← T ∪ T_p;
end

return T, I;
```

Theorem 104. *Algorithm 3.9 correctly computes the supported minimal model of a stratified dl-program $KB = (\mathcal{L}, P)$, with respect to its EDB relations* R*, where*

(i) $\mathcal{L}$ is defined in the description logic DL-Lite$_R$,

(ii) dl-queries in P are positive membership or inclusion assertions in $\mathcal{L}$, where concepts and roles are atomic, and

(iii) in dl-atoms, only update operators of the form $\uplus$ are allowed.

Proof. (Sketch) . We need to show that rewriting the ABox($\mathcal{L}$) and dl-atoms in P to nonrecursive Datalog queries is (a) feasible and (b) the queries stay positive, resulting in program P'.

(a) The membership assertions of an ABox($\mathcal{L}$) can be immediately rewritten to EDB facts of P. Next, consider the dl-atom $DL[\lambda; Q_{Lite}](c)$, the dl-query Q_{Lite} can be reduced to the conjunctive query q_{CQ} according to Theorem 102. By Theorem 96 of Algorithm `Presto(`$Q, \mathcal{T}$`)`, the conjunctive query q_{CQ} can be rewritten to a positive nonrecursive Datalog query R. A dl-update λ of the form $S_i \uplus p_i$ rewrites the query R by $(S_i \vee p_i)$ for every occurrence of S_i in the rule bodies of R.

(b) Observe that by condition (ii) and Definition 95 of Algorithm `Presto(`$Q, \mathcal{T}$`)` the nonrecursive Datalog query R is positive. Furthermore by (iii) only positive inclusion assertions by the dl-update λ are allowed.

Thus by (a) and (b) rewriting P to P' preserves the stratification of P, because only positive atoms are introduced. Then by Theorem 94, the supported minimal model for the stratified dl-program P' is computed correctly. □

Finally, we need to rewrite the generated RA expressions with fixpoint evaluation into SQL. This is primarily achieved by applying the reversal of Definition 7 to create SQL statements from the RA expressions.

The RA expressions can be rewritten into a single large SQL statement or to an intermediate DB, containing views for every IDB relation. We favor the second approach. By any approach, the created SQL statements can be evaluated on an RDBMS, exploiting the efficient query optimizers in modern RDBMSs.

Example 105. Consider the DL KB $\mathcal{L} = \langle \mathcal{T}, \mathcal{A} \rangle$, where R and Q are roles and $\mathcal{T}$ is defined as $R \sqsubseteq Q$. Furthermore $Q(c_1, c_2)$ and $Q(c_2, c_3)$ are membership assertion in $\mathcal{A}$. Let $DLP = (\mathcal{L}, P)$ be a dl-program as follows:

$b(c_1).\, b(c_2).$
$a(x, y) \leftarrow b(x),\ not\, s(x, y).$
$s(x, y) \leftarrow DL\,[R \uplus t; R]\,(x, y).$
$b(x, y) \leftarrow DL\,[; Q]\,(x, y).$
$t(x, y) \leftarrow b(x, y).$
$t(x, z) \leftarrow t(x, y),\ t(y, z).$

The dl-program DLP can be rewritten to the following Datalog rules, where the first part is directly taken and the second part is adapoted according to Algorithm 3.9:

$b(c_1).\, b(c_2).$
$q(x, y) \leftarrow a(x),\ not\, s(x, y).$
$t(x, y) \leftarrow b(x, y).$
$t(x, z) \leftarrow t(x, y),\ t(y, z).$

and

$q(c_1, c_2)$. $q(c_2, c_3)$.
$p^2_R(x, y) \leftarrow q(x, y)$.
$s(x, y) \leftarrow dl_1(x, y)$.
$dl_1(x, y) \leftarrow p^2_R(x, y)$.
$dl_1(x, y) \leftarrow t(x, y)$.
$b(x, y) \leftarrow dl_2(x, y)$.
$dl_2(x, y) \leftarrow q(x, y)$.

At this point we could rewrite the Datalog rules into RA expressions using fixpoint evaluation for the rule $t(x, z) \leftarrow t(x, y),\ t(y, z)$.

4. Implementation

4.1. Overview

In this chapter we introduce the experimental implementation of a RDBMS-based solver, called *MOR*. MOR is the abbreviation for MergeOntologyRule. The implementation stands as a proof of concept for the theories developed in Chapter 3.

We setup the following design goals for MOR:

1. Taking fully advantage of RDBMS technology;
2. reusing existing Open Source software components;
3. interfacing a DL reasoner through plug-ins; and
4. using an object-oriented programming language for the implementation.

With respect to Goal 4, Java 1.6 was chosen, because its accepted use in academia and the availability of a wide range of components (particularly the JGraphT library[1]). Crucial to Goal 3 is the existence of DL reasoners supporting $DL\text{-}Lite_R$, namely Owlgres[2] and QuOnto[3]. As for Goal 1, PostgreSQL 8.4[4] was chosen due its support for the SQL:1999 standard and its efficient query optimizer. Particularly the capability of evaluating recursive queries opened up interesting extensions for this thesis.

Resulting from the design goals, the implementation of MOR was carried out in three steps:

1. Developing a basic rewriter for Datalog to SQL.
2. Designing and developing a DL plug-in for using Owlgres 0.1 and possible other OWL2 QL reasoners.
3. Adapting Owlgres 0.1 for extracting rewritten SQL statements,

For the first step, already existing Datalog-based inference engine were considered. Particularly KAON [Bozsak et al., 2002] was promising, but the following limitations speak against it:[5]

[1] http://www.jgrapht.org/
[2] http://pellet.owldl.com/owlgres/
[3] http://www.dis.uniroma1.it/quonto/
[4] http://www.postgresql.org/
[5] http://sourceforge.net/projects/kaon/

- The main developing efforts appear to be in KAON2, while the last changes in KAON happened in 2005,
- KAON is not plug-able, so major changes in the source code would have been necessary, and
- some steps of the query evaluation are not based on recursion based on SQL.

4.2. Design

4.2.1. Architecture

The UML class diagram of Figure 4.1 gives a brief overview of the architecture. We omit the architecture on class level but refer to the source code of MOR (see Appendix A). Briefly, MOR consists of the following libraries:

- The base library,
- the main library, and
- SQL-based plug-in libraries.

The base library is made of the basic classes representing a logic program (e.g. `Rule`, `Predicate`, and `Literal`) and auxiliary classes used by the builder and the plug-ins.

The main library covers the control flow, the access to the RDBMS, the parsing, and the rewriting strategy. Depending on the configuration, plug-ins are loaded and called on demand by this library.

As for the plug-in libraries, the bridge pattern [Gamma et al., 1995] was chosen, so MOR can be extended with different SQL based plug-ins. The idea of using plug-ins with logic programs was adopted from HEX-programs [Eiter et al., 2006].

The first developed plug-in library is the DL plug-in, which encapsulates the $DL\text{-}Lite_R$ reasoner Owlgres. Beside interfacing Owlgres, parsing of DL predicates, creating auxiliary views and reprocessing SQL statements of Owlgres are implemented in this library.

The Owlgres reasoner, a Java library itself, performs the rewriting of conjunctive queries according to $DL\text{-}Lite_R$. A more detailed picture of Owlgres and the plug-in will be given in Section 4.4.

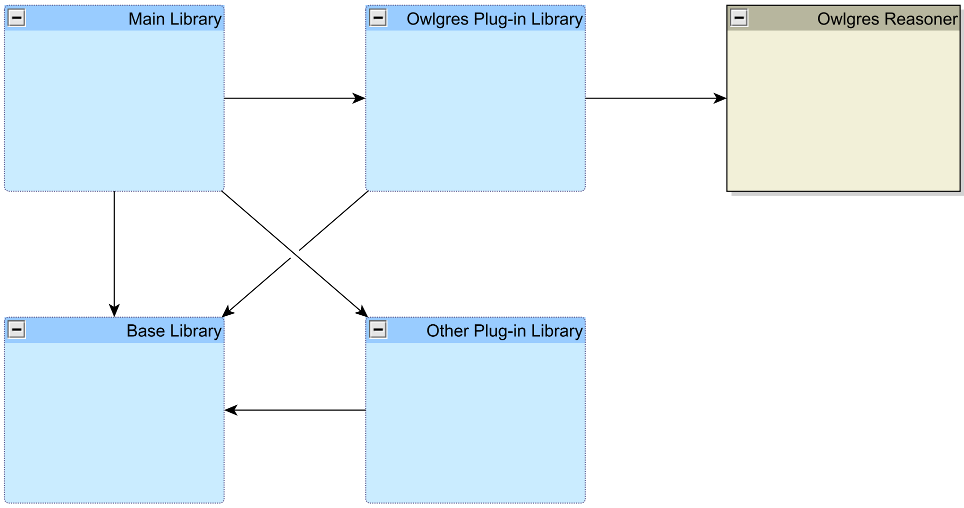

Figure 4.1.: System Architecture of MOR

4.2.2. Data- and Control-Flow

Besides standard RDBMS parameter (e.g. DB name, DB user or DB password) the main input for MOR is a logic program file, containing the program P.

The program P is parsed by the parser module, where plug-ins can overwrite the standard parsing. After successful parsing, P will be separated into a set of EDB facts and IDB rules.

The EDB of P will be directly written to the RDBMS. A valid schema of tables relating to the predicates is required and has to be created by the user beforehand.

For determining an efficient rewriting order, a dependency graph of P's IDB is build. The graph is fed into the JGraphT library[6], which computes the topological sorting of P. The careful reader will notice, that a topological sorting of stratified programs is not feasible. To retrieve a sorting, the strongly connected components of the IDB would have to be computed. Due to our current limitation to linear recursion, all cycles in the IDB are between two rules. We take advantage of this restriction, hence every pair of recursive rules can be reduced to a single rule. After reducing all recursive rules, the topological sorting of P is viable.

The detailed implementation of the rewriter is discussed in Section 4.3. Briefly, the rewriter converts the IDB rules into SQL by taking the plug-ins into account.

Finally, the SQL executor runs the generated SQL statements on the RDBMS. After successfully executing them, the results will be outputted.

Figure 4.2 gives a complete overview of the data- and control-flow.

[6]http://www.jgrapht.org/

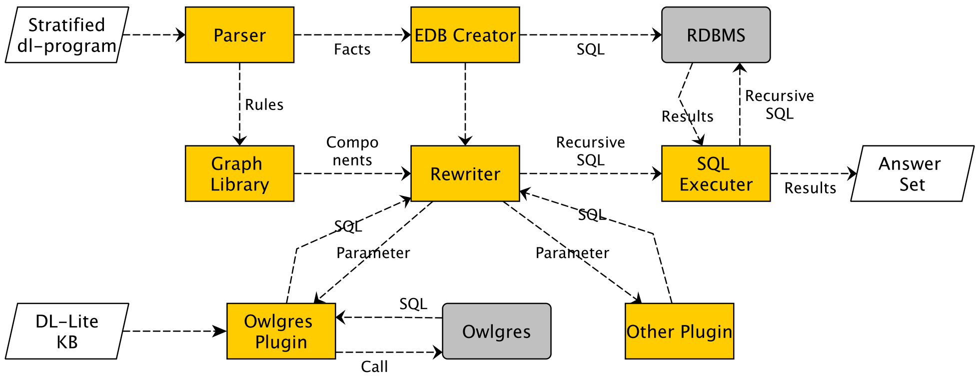

Figure 4.2.: Data- and Control-flow

4.3. Details of Rewriting Datalog to SQL

The theoretical results of Chapter 3 give an appropriate outline for rewriting Datalog to SQL. Based on this results we will define rewriting functions, which convert Datalog directly into SQL omitting RA entirely.

The Datalog program of Figure 4.3 will be taken as a running example.

4.3.1. Datatypes

Datatypes are not covered by RA, but when using an RDBMS every field has datatypes assigned. We choose the simple approach of specifying datatypes according to the constants in the input program. For simplicity, just the following XML datatypes are supported :[7]

- `xsd:string`
- `xsd:integer`
- `xsd:long`
- `xsd:float`
- `xsd:double`

If a variable is unbound, denoted by “_” , it will be ignored for further rewriting.

[7]http://www.w3.org/TR/xmlschema-2/

```
(1) husband_of(greco,pugliese).
(2) husband_of(pietro,famularo).
(3) migrated(pietro).

(4) married(X,Y) :- husband_of(X,Y).
(5) married(Y,X) :- husband_of(X,Y).
(6) married(X,Y) :- wife_of(X,Y,Z), Z > 18.
(7) married(Y,X) :- wife_of(X,Y,Z), Z > 18.
(8) parent(X,Y) :- father_of(X,Y).
(9) parent(X,Y) :- mother_of(X,Y).
(10) parent(X,Y) :- married(X,Z), father_of(Z,Y).
(11) parent(X,Y) :- married(X,Z), mother_of(Z,Y), not migrated(Y).

(12) ancestor(X,Y) :- parent(X,Y).
(13) ancestor(X,Y) :- ancestor(X,U), ancestor(U,Y).
```

Figure 4.3.: A program representing the ancestor problem

4.3.2. Rewriting the EDB

The EDB can be mapped straightforward to entries in related DB tables.

Definition 106. A fact a is defined as an atom $p(c_1, ..., c_n)$, where p is a predicate, each $c_1, ..., c_n$ is a constant, and $n \geqslant 0$. The rewriting function f_{EDB} on P is defined as follows:

$$f_{EDB}(P) = \bigcup_{i=1,\ldots,|EDB(P)|} f_F(a_i)$$

where

$$f_F(a): \quad \texttt{INSERT INTO MOR\_}a\ \texttt{FIELDS}(attr_{a,1}, ..., attr_{a,n})\ \texttt{VALUES}(c_{a,1}, ..., c_{a,n});$$

Notice that the prefix `MOR_` is used to name custom tables and to avoid conflicts with existing DB tables.

Example 107. The following example illustrates the result of function f_F on the EDB facts (1), (2), and (3) of Figure 4.3:

```
INSERT INTO TABLE MOR_husband('greco','pugliese');
INSERT INTO TABLE MOR_husband('pietro','famularo');
INSERT INTO TABLE MOR_migrated('pietro');
```

4.3.3. Rewriting Nonrecursive Rules

In SQL there are several syntactical ways to expess joins. We will use the *implicit join notation*, where the joined tables are simply listed in the `FROM` clause.

Any variable of a body atom is chosen for the projection, if it matches a variable in the head atom. Any variable of a body atom is taken for the selection, if it matches a variable of another body atom. Every unmatched variable is considered unbound and ignored. For rewriting a nonrecursive rule we can distinguish four different cases, depending on the occurance of NAF literals and SQL operators.

Definition 108. Let a rule r be defined as:

$a \leftarrow b_1, ..., b_k, not\, b_{k+1}, ..., not\, b_m, f_r, ..., f_s.$

where $a, b_1, ..., b_m$ are atoms, $f_r, ..., f_s$ are built-in functions, $m \geqslant k \geqslant 0$, and $s \geqslant r \geqslant 0$. We recall Definition 8 for the atoms. A built-in function is defined as $(t_1\, op\, t_2)$, where op is a SQL operator with $op \in \{\geq, >, \leq, <, =, !=, \texttt{LIKE}, \texttt{BETWEEN}, \texttt{IN}\}$ and t_1, t_2 are terms. Furthermore the Datalog safety condition has to be fullfilled.

The following sets are defined over r:

$P = \big\{t_b \mid t_a \in a(t_1, ..., t_{n_a}) \ \wedge \ t_b \in b_j(t_1, ..., t_{n_j}) \ \wedge \ j \leq k \ \wedge \ t_a = t_b\big\}$;

$S^+ = \big\{\langle t_1, t_2\rangle \mid t_1 \in b_i(t_1, ..., t_{n_i}) \ \wedge \ t_2 \in b_j(t_1, ..., t_{n_j}) \ \wedge \ i \leq j \leq k \ \wedge \ b_i \neq b_j \ \wedge \ t_1 = t_2\big\}$;

$S^- = \left\{\left\langle b_j, \vec{t_1}, \vec{t_2}\right\rangle \mid t_1 \in b_i(t_1, ..., t_{n_i}) \ \wedge \ t_2 \in b_j(t_1, ..., t_{n_j}) \ \wedge \ i \leq k < j \ \wedge \ t_1 = t_2\right\}$;

$S^{op} = \{\langle t_1, op, t_2\rangle \mid (t_1\, op\, t_2) \in f_i \wedge \ s \geqslant i \geqslant r\}$.

Then we define the rewriting function f_B on r as the following four cases $f_B^{\emptyset}$, f_B^+, f_B^-, and f_B^{op}:

1. The case $f_B^{\emptyset}$ with $S^- = \emptyset$, $S^+ = \emptyset$, and $S^{op} = \emptyset$ is defined as follows, where $p \in P$:

$$f_B^{\emptyset}(r, P): \quad \texttt{SELECT DISTINCT}\ p_1, ..., p_n\ \texttt{FROM}\ b_1, ..., b_k;$$

2. The case f_B^+ with $S^- = \emptyset$ and $S^{op} = \emptyset$ is defined as follows, where $s^+ \in S^+$ and $p \in P$:

$$\begin{aligned} f_B^+(r, P, S^+): \quad & \texttt{SELECT DISTINCT}\ p_1, ..., p_n\ \texttt{FROM}\ b_1, ..., b_k \\ & \texttt{WHERE}\ s^+_{1,1} = s^+_{1,2}\ \texttt{AND}, ..., s^+_{n,1} = s^+_{n,2}; \end{aligned}$$

3. The case f_B^- with $S^{op} = \emptyset$ is defined as follows, where $s^- \in S^-$, $s^+ \in S^+$, and $p \in P$:

$$\begin{aligned} f_B^-(r, P, S^+, S^-): \quad & \texttt{SELECT DISTINCT}\ p_1, ..., p_n\ \texttt{FROM}\ b_1, ..., b_k \\ & \texttt{WHERE}\ s^+_{1,1} = s^+_{1,2}\ \texttt{AND}, ..., s^+_{n,1} = s^+_{n,2}\ \texttt{AND} \\ & (s^-_{1,2,1}, ..., s^-_{1,2,n})\ \texttt{NOT IN(SELECT}\ s^-_{1,3,1}, ..., s^-_{1,3,n}\ \texttt{FROM}\ s^-_{1,1}) \\ & \texttt{AND}, ..., (s^-_{n,2,1}, ..., s^-_{n,2,n})\ \texttt{NOT IN} \\ & (\texttt{SELECT}\ s^-_{n,3,1}, ..., s^-_{n,3,n}\ \texttt{FROM}\ s^-_{n,1}); \end{aligned}$$

4. The case f_B^{op} with $S^- = \emptyset$ and $S^+ = \emptyset$ is defined as follows, where $s^{op} \in S^{op}$, and $p \in P$:

$$f_B^{op}(r, P, S^{op}): \quad \texttt{SELECT DISTINCT}\ p_1, ..., p_n\ \texttt{FROM}\ b_1, ..., b_k \\ \texttt{WHERE}\ s_{1,1}^{op}\ op_1\ s_{1,2}^{op}.\ \texttt{AND}, ..., s_{n,1}^{op}\ op_n\ s_{n,2}^{op};$$

The four cases just define the basic rewriting functions of a nonrecursive rule. As a consequence we can combine them to extend the rewriting as follows:

$f_B^{+op}(r, P, S^+, S^{op}):\ f_B^+(r, P, S^+)$ and $f_B^{op}(r, P, S^{op})$;

$f_B^{-op}(r, P, S^+, S^-, S^{op}):\ f_B^-(r, P, S^+)$ and $f_B^{op}(r, P, S^{op})$.

Definition 109. Let r be a rule of program P and let p be a predicate as in Definition 108. Let V be the set of all predicates in P. The rewriting function f_{IDB} on P is defined as follows:

$$f_{IDB}(P) = \bigcup_{i=1,...,|V|} f_R(v_i, R_i)$$

where R is the set of rules as follows:

$R = \{r \mid v \in H(r)\}$, for a predicate $v \in V$ and every rule $r \in IDB(P)$.

Then we define the rewriting function f_R on v and R as follows, where $r \in R$:

$$f_R(v, R): \quad \texttt{CREATE OR REPLACE VIEW MOR\_}v(attr_{v,1}, ..., attr_{v,n})\ \texttt{AS} \\ f_B(r_1)\ \texttt{UNION}\ ...f_B(r_n);$$

Roughly speaking, in the function f_{IDB} the rules of a program are divided into subsets according to the criteria of sharing the same predicate in the head atom. For every subset the function f_B is applied to every rule and the results are merged by an `UNION` clause.

Example 110. The Rules (8), (9), (10), and (11) from Figure 4.3 are rewritten the following way:

```
CREATE OR REPLACE VIEW MOR_parent(att1, att2) AS
(SELECT DISTINCT MOR_father_of.att1, MOR_father_of.att2 FROM MOR_father_of
UNION
SELECT DISTINCT MOR_mother_of.att1, MOR_mother_of.att2 FROM MOR_mother_of
UNION
SELECT DISTINCT MOR_married.att1, MOR_father_of.att2 FROM MOR_married, MOR_father_of
WHERE MOR_married.att2=MOR_father_of.att1
UNION
SELECT DISTINCT MOR_married.att1, MOR_mother_of.att2 FROM MOR_married, MOR_mother_of
WHERE MOR_married.att2=MOR_mother_of.att1 AND
(MOR_mother_of.att2) NOT IN (SELECT MOR_migrated.att1 FROM MOR_migrated ));
```

4.3.4. Rewriting Recursive Rules

For this thesis, the rewriting of Datalog to SQL would have been less appealing without the introduction of recursion to the DB field. This occurred with the SQL:1999 standard [ISO, 1999] and fortunately, some RDBMS vendor have implemented SQL:1999 almost entirely. The listed RDBMS support recursive queries:

- PostgreSQL with version 8.4,[8]
- Microsoft SQL Server with version 2008,[9]
- IBM DB2 with version 7.2,[10] and
- Oracle with version 9i release 2.[11]

As already pointed out in Chapter 3, the main drawback of SQL:1999 is its limitation to linear recursion.

Definition 111. In PostgreSQL 8.4, the syntax of linear recursive queries is taken from the upcoming SQL:2008 standard:

$$\begin{array}{c}\texttt{WITH RECURSIVE} < \texttt{query name} > \texttt{AS} \ (< \texttt{table subquery} >) \\ \{...\} \ [\ \texttt{SEARCH clause} \mid \texttt{CYCLE clause} \ ] < \texttt{SELECT body} > \end{array}$$

Definition 112. The rewriting function f_B^{rec} for linear recursive rules is an additional case of function f_B. Let the rules r_1 and r_2 of program P be defined as:

$r_1 : a \ \leftarrow \ b, c_1, ..., c_k.$

$r_2 : a_1 \ \leftarrow \ a_2, a_3, c_1, ..., c_k.$

where the rules, atoms, predicates, and terms are defined as in Definition 108. Furthermore the atoms a, a_1, a_2, and a_3 share the same predicate.

The following sets are defined over r_1 and r_2:

$P^1 = \{t_b \mid r_1 \ \wedge \ t_a \in a(t_1, ..., t_{n_a}) \ \wedge \ t_b \in b(t_1, ..., t_{n_b}) \ \wedge \ t_a = t_b\}\,;$

$P^2 = \{t_c \mid r_2 \ \wedge \ t_a \in a_1(t_1, ..., t_{n_{a_1}}) \ \wedge \ ((t_c \in a_2(t_1, ..., t_{n_{a_2}}) \ \wedge \ t_a = t_c) \vee$

$\qquad (t_c \in a_3(t_1, ..., t_{n_{a_3}}) \ \wedge \ t_a = t_c))\};$

$S = \left\{\langle t_1, t_2\rangle \mid r_2 \ \wedge \ t_1 \in a_2(t_1, ..., t_{n_{a_2}}) \ \wedge \ t_2 \in a_3(t_1, ..., t_{n_{a_3}}) \ \wedge \ t_1 = t_2\right\}\,;$

[8] http://www.postgresql.org/
[9] http://www.microsoft.com/sqlserver/
[10] http://www.ibm.com/db2/
[11] http://www.oracle.com/database/

The function f_B^{rec} is defined as follows, where $s \in S$, $p^1 \in P^1$, $p^2 \in P^2$, and $lm \in \mathbb{N}$:

$$
\begin{array}{ll}
f_B^{rec}(r_1, r_2, P^1, P^2, S, lm): & \texttt{CREATE OR REPLACE VIEW MOR\_}a\texttt{\_}base(attr_{a,1}, ..., attr_{a,n})\ \texttt{AS} \\
& (\texttt{SELECT DISTINCT}\ p^1_1, ..., p^1_n\ \texttt{FROM}\ b); \\
& \\
& \texttt{CREATE OR REPLACE VIEW MOR\_}a(p^2_1, ..., p^2_n)\ \texttt{AS(} \\
& \texttt{WITH RECURSIVE MOR\_}a\texttt{\_}rc(p^2_1, ..., p^2_n)\ \texttt{AS} \\
& ((\texttt{SELECT DISTINCT}\ p^1_1, ..., p^1_n\ \texttt{FROM MOR\_}a\texttt{\_}base) \\
& \texttt{UNION} \\
& (\texttt{SELECT DISTINCT}\ p^2_1, ..., p^2_n\ , \\
& \texttt{FROM MOR\_}a\texttt{\_}rc,\ \texttt{MOR\_}a\texttt{\_}base \\
& \texttt{WHERE}\ s_{1,1} = s_{1,2}\ \texttt{AND}, ..., s_{n,1} = s_{n,2})) \\
& \texttt{SELECT}\ p^2_1, ..., p^2_n\ \texttt{FROM MOR\_}a\texttt{\_}rc\ \texttt{LIMIT}\ lm);
\end{array}
$$

If there are cycles in the EDB, the recursive query evaluation of PostgreSQL will not terminate. In the PostgreSQL documentation this issue is stated as: "When working with recursive queries it is important to be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely."[12] As commented by the PostgreSQL developers, the recursive query evaluation is implemented as follows:[13]

1. Evaluate the non-recursive term of the `SEARCH` clause.
2. Evaluate the `CYCLE` clause by applying a breadth-first search, where a working table and a result table is kept.

We decided to introduce the `LIMIT` parameter, to restrict the rows which are fetched by the parent query. This has the effect, that a circular evaluation will stop after a certain amount of iteration in the breadth-first search. It is crucial to set the `LIMIT` parameter not less than the size of the expected result set. An incorrect parameter will cause a incomplete result set. A better approach and valuable extension of MOR would be the introduction of a native DB function, which tracks all visited vertices to break cycles. See the PostgreSQL documentation for an in-depth discussion of this issue.[14]

4.4. Interfacing Owlgres with the DL Plug-in

As a second step, an intermediate layer between MOR and Owlgres had to be developed. We follow the design of dlvhex and HEX-programs (see [Eiter et al., 2006]) and introduce a DL plug-in to interface with Owlgres and possible other DL reasoner.

The authors of [Stocker and Smith, 2008] put the focus of Owlgres as: "Owlgres aims at both efficient querying over a scalable persistent store and automatic reasoning for RDF and OWL data." Owlgres in its beta-release 0.1 is optimized for PostgreSQL, but

[12]http://www.postgresql.org/docs/8.4/static/queries-with.html
[13]http://archives.postgresql.org/pgsql-hackers/2008-02/msg00642.php
[14]http://www.postgresql.org/docs/8.4/static/queries-with.html

could be adapted to other RDBMS. Owlgres can act as server, awaiting queries through TCP/IP. We decided to used Owlgres directly by referencing it as a component.

The DL plug-in covers two major cases:

- The first case is querying the DL KB. This is realized by calling Owlgres with a given SPARQL query.
- The second case extends the simple query with updates to Owlgres' DL KB.

4.4.1. DL-Atoms

In HEX-programs a strict approach for querying the DL KB with DL atoms was chosen. This has the advantage of uniformly interfacing different DL reasoner with different query languages. Thereby users of the system are strictly guided with the formulation of their DL query. According to Eiter et al. DL atoms of HEX-programs support the following queries [Eiter et al., 2006]:

- Concept queries with the *&dlC* atom,
- object role queries with the *&dlR* atom,
- data role queries with the *&dlDR* atom, and
- conjunctive queries (resp. union of conjunctive queries) with *&dlCQ* (resp. *&dlUCQ)* atom.

In MOR on the contrary, accessing the DL KB is accomplished by SPARQL queries. The main reason for using SPARQL is influenced by the fact, that Owlgres uses it as the language for query answering. Since Owlgres is not full SPARQL-complete, there are some limitation regarding expressivity. For example the expression `OPTIONAL` is not supported by Owlgres [Stocker and Smith, 2008]. Opposed to DL atoms of HEX-programs, the SPARQL queries are stored in external files and referenced in the DL atom. Using SPARQL directly has the nice advantage, that it opens up the full expressibility of the language. Thus any extensions in SPARQL and Owlgres are immediately available in MOR.

We follow the notion of *dl-atom* and *dl-query* introduced in Chapter 3. We use the dl-atom for practical purposes in two forms, namely as a standard atom and an update atom.

Definition 113. The *standard dl-atom* is defined the following way:

$\&\texttt{dlQS}[\texttt{query\_uri}](\texttt{X}_1, ..., \texttt{X}_\texttt{n})$,

where `query_uri` is the URI of a SPARQL query and $\texttt{X}_1, ..., \texttt{X}_\texttt{n}$ is the list of output terms.

Example 114. The following example is a plain dl-program. It references a query do Dbpedia, which retrieves all books of a certain type:

```
book(X,Y) :- &dlQS["books.rq"](X,Y). ,
```

where the SPARQL query `books.rq` is defined as:

```
SELECT ?s ?n WHERE { ?s rdf#type yago:Book106410904 .  ?s dbpedia:name ?n .  }
```

Definition 115. The *update dl-atom* extends the DL KB access with concept, object-role and data-role updates:

&dlQU[query_uri, op1, pred1, op2, pred2, op3, pred3]($X_1, ..., X_n$) ,

with the following parameter:

query_uri ...	is the URI of a SPARQL query.
op1 ...	denoting a positive (+) or negative (-) concept assertions.
pred1 ...	designating the binary predicate for concept extensions, where the first term is a DL class and the second term is a DL individual.
op2 ...	similar to op1 but for object-role assertions.
pred2 ...	designating the ternary predicate for concept extensions, where the first term is a DL individual, the second a DL role and the third again an DL individual.
op3 ...	is similar to op1 but it is intended for data-role assertion.
pred3 ...	is similar to pred2 but the third term is plain text instead of an DL individual.
$X_1, ..., X_n$	denotes the list of output terms. Note that the output terms have to match the variables of the SPARQL query defined in query_uri.

Example 116. The following dl-program extends Example 114 with membership assertions of a Spanish book.

```
book(X,Y) :- &dlQU["data/dbpedia_query.rq",+,updateC,+,updateOR,+,updateDR](X,Y).
updateC(http://dbpedia.org/class/yago/Book106410904,study_Spanish_1).
updateDR(study_Spanish_1,http://dbpedia.org/property/name,einführungSpanish1).
updateOR(study_Spanish_1,http://dbpedia.org/property/successor,study_Spanish_2).
```

4.4.2. Owlgres Overview

The Owlgres DL KB has to be initialized by creating the TBox and filling the ABox. This is accomplished in the command line interface by the following commands [Stocker and Smith, 2008]:

- Creating the TBox is achieved by using the `sh/create` command. The name of the RDBMS and the URI of the OWL file with the TBox definitions have to be provided.
- Loading the ABox assertions is done with `sh/load` command. Furthermore the RDBMS's name and the URI of the OWL file with the ABox definition have to be given.

For each DB instance, the TBox needs to be created once, the ABox can be loaded several times. By loading the TBox, a static DB schema is created, which builds the underlying structure for query answering.

For evaluating the update dl-atom, the DB schema of the Owlgres needs to be temporarily changed. Thus a closer look at the DB schema is needed to illustrate further steps. The relevant DB tables of Owlgres are shown in Table 4.2. Note that the `id` fields are the primary and foreign keys to reference the different tables to each other:

Table	Fields	Description
`concept_assertion`	`concept (id),` `individual (id)`	$Concept(Individual)$

`data_role_assertion`	`data_role (id)`
	`individual (id)`
	`value (text)`
	`datatype (text)`
	`language (text)`
		$DataRole(Individual, Value)$

`individual_name`	`individual (id)` `name (text)`	An id and name is assigned to every $Individual$

`object_role_assertion`	`object_role (id)`
	`a (id)`
	`b (id)`
		$ObjectRole(A, B)$

`tbox_concept_inclusion`	`sub (id)`
	`super (id)`
	`positive (bool)`
		$SubConcept \sqsubseteq SuperConcept$ or $SubConcept \sqsubseteq \neg SuperConcept$

`tbox_data_role_inclusion`	`sub (id)` `super (id)`	$\exists SubDataRole.Concept \sqsubseteq \exists SuperDataRole.Concept$

`tbox_name`	`id (id)`
	`type (text)`
	`auxiliary (text)`
	`frequency (text)`
	`name (text)`
		An id, name, type and its frequency is assigned to every $Class$, $ObjectRole$, $DataRole$, $Namespace$

`tbox_object_role_inclusion`	`sub (id)` `super (id)`	$\exists SubObjectRole.Concept \sqsubseteq \exists SuperObjectRole.Concept$

Table 4.2.: Owlgres 0.1 DB Schema

4.4.3. Owlgres KB Management

As mentioned before, the TBox is closely tied to the DB schema. Thus for every new TBox a new DB instance is needed. For the purpose of our experiments, we did not see any real drawback with this technique.

4.4.4. Rewriting the Standard DL-Atom

The rewriting of the standard dl-atom is straightforward. The `query_uri` and the RDBMS parameter are forwarded to the rewrite function of Owlgres. After Owlgres is called, the resulting SQL statements are embedded into a newly created DB view. The rewrite function of Owlgres is an implementation of the Algorithm `Answer`$(Q, \mathcal{K})$ in $DL\text{-}Lite_R$ (we refer to Chapter 2). As described in [Stocker and Smith, 2008], the standard rewriting of [Calvanese et al., 2007] was extended with three types of optimizations:

- Query simplification,
- selectivity optimization, and
- rewriting a set of conjunctive queries as a single query of `UNION` clauses.

4.4.5. Rewriting the Update DL-Atom

The following requirements have to be considered by rewriting the update dl-atom:

- After updating and accessing the DL KB, we need to transform it in its prior state, and
- as discussed in Section 4.3.3, a cascade of DB views and SQL statements is created and evaluated synchronous.

Note that otherwise the DL KB could be directly manipulated on the RDBMS.

The update process, which is shown in Algorithm 4.1 has the following steps:

- First, the original DB tables are renamed;
- Second, the auxiliary tables are created;
- And third, the auxiliary tables are "unionized" with the renamed tables using their original name.

In Algorithm 4.1 the update of the DL KB occurs in stages. A stage is represented by the variable *lvl*. The introduction of stages is needed, otherwise different update dl-atoms would interfere with each other. Through stages, every update dl-atom constructs its own temporary state of the DL KB.

The statements `UNION` (resp. `EXCEPT`) are used to implement positive (resp. negative) membership assertion. As apparent from the DB schema in Table 4.2, there are no negative assertion defined in `concept_assertion`, `object_role_assertion`, and `data_role_assertion`. Thus we decided to use `EXCEPT` for excluding the negative assertions from the result.

Furthermore, for every created individual, a temporary id is created, otherwise the new individual would collide with already existing individuals. For creating new ids without materializing the individuals, we used PostgreSQL's sequence function, which creates unique identifiers on demand.[15]

[15] http://www.postgresql.org/docs/8.4/static/sql-createsequence.html

Algorithm 4.1 Update DL KB

```
Input: Set A of update dl-atoms
Result: Set S of SQL commands, set U of undo SQL commands
lvl = 1;
foreach atom a ∈ A do
  pred1 ← pred1 of a;
  pred2 ← pred2 of a;
  pred3 ← pred3 of a;
  /* Rename current state and create new state of ABox */
  S ∪ RENAME tables individual_name, concept_assertion, data_role_assertion,
      object_role_assertion to in_+lvl, ca_+lvl, da_+lvl, oa_+lvl;
  S ∪ CREATE auxiliary tables aux_in_+lvl, aux_ca_+lvl, aux_da_+lvl, aux_oa_+lvl;
  S ∪ SELECT INTO aux_in FROM view pred1;
  S ∪ SELECT INTO aux_ca FROM INNER JOIN view pred1, table tbox_name;
  S ∪ SELECT INTO aux_da FROM INNER JOIN view pred2, table individual_name,
      table tbox_name;
  S ∪ SELECT INTO aux_oa FROM INNER JOIN view pred3, table individual_name,
      table tbox_name;
  /* Union (except) of old and new state of ABox */
  S ∪ UNION of in_+lvl and aux_in calling it individual_name;
  S ∪ UNION or EXCEPT of ca_+lvl and aux_ca calling it concept_assertion;
  S ∪ UNION or EXCEPT of da_+lvl and aux_da calling it data_role_assertion;
  S ∪ UNION or EXCEPT of oa_+lvl and aux_oa calling it object_role_assertion;
  U ∪ reverse of S;
  increase lvl by 1;
end
return S, U;
```

4.4.6. Adaptions in Owlgres 0.1

The third step is adapting Owlgres 0.1 in a way, such that plain SQL statements can be extracted, and no transactions are performed on the RDBMS. Performing any transaction during the rewriting step would considerable decrease the performance of MOR.

The main adaptions in Owlgres are as follows:

1. In the library `Owlgres_CLI` the new class `queryRewrite` was introduced. This class works as an entry point for external systems (e.g. MOR).
2. In several intermediate classes (e.g. `OWLGres`) a method called `queryRewrite()` was added to forward the rewriting call to the `Owlgres_Core` library.
3. In the class `StoreConnectionBase` of the `Owlgres_Core` library, a `SQLQueryBuilder` is created and the SQL statements retrieved.
4. In the class `DLLKB`, which calls `StoreConnectionBase`, the original design had to be altered. In the original class, calls to rewrite a query always created a `ResultSet`,

which decrease the performance of the overall system. This was adapted, that in case of `queryRewrite()` only a string and not the whole result set is returned.

4.5. Limitations

As already mentioned, MOR is an experimental system, and due to "usual" limitation of resources (e.g. time), we did not implement the following functions yet:

- As discussed in Chapter 3, only linear recursion and not general recursion is supported;
- Algorithm `RectifyRules` of Chapter 3 is not considered;
- Other OWL2 QL reasoners (e.g. QuOnto) could be integrated;
- A generic SQL plug-in is missing;
- The public interface of the MOR libraries have to be reconsidered, so MOR could be used as a software component in other systems;
- Replacing the `LIMIT` parameter with a native DB function, which handles cycles in recursive queries.

5. Experiments

5.1. Methodology

The Asparagus competition was a major step in the field of ASP for benchmarking ASP reasoners [Gebser et al., 2007b, Denecker et al., 2009]. Also for OWL several benchmark suites as Lehigh University Benchmark (LUBM) and Ontology Benchmark (UOBM) have evolved. We refer the interested reader to [Guo et al., 2005, Ma et al., 2006]. For more general rule-based systems, OpenRuleBench was developed to benchmark different types of reasoners, namely Prolog-based, Deductive Database, Productive Rule, and Reactive Rule systems [Liang et al., 2009].

None of the mentioned benchmarks is designed to cover dl-programs. Fortunately, OpenRuleBench includes some useful benchmarks, which could be adopted for these experiments. When reasoning systems are benchmarked, there is an ongoing discussion on the separation of loading and inference time [Liang et al., 2009]. In contrast to OpenRuleBench, this benchmark does not separate loading and inference time. This is done because we focus on RDBMS technology, where important optimizations are done in the loading step.

The experiment is split into four different scenarios, whereby the last does not contain benchmarks:

1. This scenario is composed of ordinary Datalog programs.
2. In this scenario Datalog programs query a DBpedia DL KB.
3. More complex than the second scenario, a LUBM DL KB is queried instead.
4. This scenario shows extensions and limitations of our prototype.

To cover a wide area of testing, both random and "real world" data is used. The generated data for each scenario contains approximately 10,000, 100,000, 500,000, and 1,000,000 facts or assertions.

5.2. Scenario 1 - Datalog

This scenario focuses just on Datalog programs. Based on three benchmarks, some features like recursion and default negation are tested.

MOR is compared to the following systems:

- DLV using its ODBC interface [Leone et al., 2006], and
- DLV^{DB} using the auxiliary directives `USE` for importing the EDB and `CREATE` for defining the IDB [Terracina et al., 2008].

5.2.1. Large Join Benchmark

This benchmark is taken from OpenRuleBench, whereby the EDB consists of the relations `c2`, `c3`, `c4`, `d1`, `d2`, and `ex`. These relations are first randomly generated and then imported to PostgreSQL. Note that the `#import` directive in MOR is used as in DLV, mapping DB relations to the EDB. For implications the symbol `:-` is used instead of ←.

Definition 117. The first benchmark is a non-recursive program evaluating a tree of binary joins:

```
#import(c2). #import(c3). #import(c4). #import(d1). #import(d2).
% Query
result(X,Y) :- b1(X,Z), b2(Z,Y).
% Main
b1(X,Y) :- c1(X,Z), c2(Z,Y).
b2(X,Y) :- c3(X,Z), c4(Z,Y).
c1(X,Y) :- d1(X,Z), d2(Z,Y).
```

5.2.2. Default Negation Benchmark

Definition 118. This benchmark is extending the previous program simply with default negation:

```
#import(c2). #import(c3). #import(c4). #import(d1). #import(d2). #import(ex).
% Query
result(X,Y) :- b1(X,Z), b2(Z,Y).
% Main
b1(X,Y) :- c1(X,Z), c2(Z,Y), not ex(Y).
b2(X,Y) :- c3(X,Z), c4(Z,Y), not ex(Y).
c1(X,Y) :- d1(X,Z), d2(Z,Y), not ex(X).
```

5.2.3. Stratified Negation Benchmark

The well know ancestor problem is taken to show recursion and default negation. Again the relations of the EDB are generated randomly, hence certain instances of the EDB might be cyclic.

Definition 119. This program captures the transitive closure of the relation `parent`. It consists of the following rules:

```
#import(migrated). #import(husband_of). #import(wife_of). #import(father_of).
#import(mother_of).
% Query
result(X,Y) :- married(X,Y), not ancestor(X,Y).
% Main
married(X,Y) :- husband_of(X,Y).
married(Y,X) :- husband_of(X,Y).
married(X,Y) :- wife_of(X,Y,Z), Z > 18.
married(Y,X) :- wife_of(X,Y,Z), Z > 18.
parent(X,Y) :- father_of(X,Y).
parent(X,Y) :- mother_of(X,Y).
parent(X,Y) :- married(X,Z), father_of(Z,Y).
parent(X,Y) :- married(X,Z), mother_of(Z,Y), not migrated(Y).
% Recursion
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,U), ancestor(U,Y).
```

5.3. Scenario 2 - Derived DBpedia

Being the goal of this work, the capabilities of combining Datalog and DL-Lite based on an RDBMS are shown. This scenario is split into three benchmarks, whereby the first is a simple access to the DL layer, the second has access to the DL layer and negation is included, and the third combines access and update to the DL layer.

MOR is benchmarked agains dlvhex, using the Description Logic Plug-in with RacerPro 1.9.2 [Eiter et al., 2006].

We extracted three types of literature (e.g. books, periodicals, and publications) from the DBpedia DB[1]. The extracted data was then imported to Owlgres filling the ABox.

Definition 120. We defined the TBox as follows, because the whole TBox of DBpedia is not capturable with Owlgres:

$$\begin{array}{l} Book \sqsubseteq Publication \\ Periodical \sqsubseteq Publication \\ \exists Name.nameString \\ \langle publication, nameString \rangle : Name \end{array} .$$

5.3.1. Simple Benchmark

Definition 121. This program is simple merging two DL queries for books and journals:

[1] http://wiki.dbpedia.org/OnlineAccess

```
% Query
result(X,Y) :- book(X,Y).
result(X,Y) :- journ(X,Y).
% DL Access
book(X,Y) :- &dlQS["dbpedia_query1"](X,Y).
journ(X,Y) :- &dlQS["dbpedia_query2"](X,Y).
```

5.3.2. Advanced Benchmark

Definition 122. This program selects all available books and excludes the periodicals from the total set. Then a given range of books are chosen from the total set:

```
% Query
result(X,Y) :- journ_sel_a(X,Y).
result(X,Y) :- journ_sel_d(X,Y).
% Main
journ_sel_a(X,Y) :- journ(X,Y), Y > 'Book 1000', Y < 'Book 1500'.
journ_sel_d(X,Y) :- journ(X,Y), Y > 'Book 8000', Y < 'Book 8500'.
journ(X,Y) :- publ(X,Y), not book(X,Y).
% DL Access
book(X,Y) :- &dlQS["dbpedia_query1"](X,Y).
publ(X,Y) :- &dlQS["dbpedia_query3"](X,Y).
```

5.3.3. Update Benchmark

Definition 123. In this program the ABox is extended with three Spanish books:

```
% Query
result(X,Y) :- book(X,Y).
result(X,Y) :- journ(X,Y).
% DL Access and Update
book(X,Y) :- &dlQU["dbpedia_query1",+,updateConcept,+,,+,updateData](X,Y).
journ(X,Y) :- &dlQS["dbpedia_query2"](X,Y).
% Update Predicates
updateConcept(X,Y) :- uc(X,Y).
updateData(X,Y,Z) :- ud(X,Y,Z).
uc(yago:Book106410904,study_Spanish_1).
uc(yago:Book106410904,study_Spanish_2).
uc(yago:Book106410904,study_Spanish_3).
ud(study_Spanish_1,dbpedia:name,einführungSpanish1).
ud(study_Spanish_2,dbpedia:name,einführungSpanish2).
ud(study_Spanish_3,dbpedia:name,enführungSpanish3).
```

5.3.4. DBpedia queries

Definition 124. The following list contains all SPARQL queries used in the DL plug-in for this scenario:

```
dbpedia_query1:  SELECT ?s ?n WHERE
                  {?s rdf:type yago:Book106410904.  ?s dbpedia:name ?n.}
dbpedia_query2:  SELECT ?s ?n WHERE
                  {?s rdf:type yago:Periodical106593296.  ?s dbpedia:name ?n.}
dbpedia_query3:  SELECT ?s ?n WHERE
                  {?s rdf:type yago:Publication106589574.  ?s dbpedia:name ?n.}
```

5.4. Scenario 3 - Derived LUBM

Similar to the previous scenario, MOR is benchmarked against dlvhex.

LUBM is one of the standard data sets to benchmark Semantic Web applications. A data generator[2] is part of LUBM, so an ABox of generic universities can be generated. The DL KB consists mainly of universities, departments, professors, students and courses. We refer to [Guo et al., 2005] for a detailed definition of the TBox.

5.4.1. Simple Benchmark

Definition 125. This program is adopted from the LUBM examples. The program retrieves all students which take certain courses:

```
% Query
result(X) :- takesCourse(X,Y), u_important(Y), graduateStudent(X).
% Main
u_important(http://www.Department0.University0.edu/GraduateCourse0).
u_important(http://www.Department0.University0.edu/GraduateCourse2).
u_important(http://www.Department0.University0.edu/GraduateCourse4).
% DL Access
graduateStudent(X) :- &dlQS["lubm_query1"](X).
takesCourse(X,Y) :- &dlQS["lubm_query2"](X,Y).
```

5.4.2. Advanced Benchmark

Definition 126. This program is also taken from the LUBM, but extended with negation. The program is seeking students which take courses of faculty advisors, whereby the advisors should not be full professors.

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

```
% Query
result(X) :- advisor(X,Y), teacherOf(Y,Z), takesCourse(X,Z),
  student(X), faculty(Y), course(Z), not fullprof(Y).
% DL Access
takesCourse(X,Y) :- &dlQS["lubm_query2"](X,Y).
advisor(X,Y) :- &dlQS["lubm_query3"](X,Y).
teacherOf(X,Y) :- &dlQS["lubm_query4"](X,Y).
student(X) :- &dlQS["lubm_query5"](X).
faculty(X) :- &dlQS["lubm_query6"](X).
course(X) :- &dlQS["lubm_query7"](X).
fullprof(X) :- &dlQS["lubm_query8"](X).
```

5.4.3. Update Benchmark

Definition 127. In this benchmark we overcome the lack of transitivity in DL-Lite$_R$ and OWL 2 QL [Motik et al., 2009]. This is done by calculating the transitive closure of the organization hierarchy in the Datalog program. Then, the results are injected back to the DL KB. Finally, the altered DL KB is queried for the main result. Note, that we introduce a new role called `subOrganizationOfTC`, which assures that the transitive closure is not conflicting with the original role `subOrganizationOf`.

```
% Query
result(X,Y,Z) :- graduateStudent(X), memberOf(X,Z), ugDegreeFrom(X,Y),
  univ(Y), dept(Z), subOrgOf(Y,Z).
% DL Access
graduateStudent(X) :- &dlQS["lubm_query1"](X).
memberOf(X,Y) :- &dlQS["lubm_query9"](Y,X).
ugDegreeFrom(X,Y) :- &dlQS["lubm_query10"](Y,X).
univ(X) :- &dlQS["lubm_query11"](X).
dept(X) :- &dlQS["lubm_query12"](X).
% DL Access and Update
subOrgOf(X,Y) :- &dlQU["lubm_query14",+,,+,updateRole,+,](Y,X).
% Recursion and Update Predicate
updateRole(X,Y,Z) :- u_roletc(Y), updateSubOrg(X,Z).
baseOrg(X,Y) :- &dlQS["lubm_query13"](X,Y).
updateSubOrg(X,Y) :- baseOrg(X,Y).
updateSubOrg(X,Z) :- updateSubOrg(X,Y), updateSubOrg(Y,Z).
u_roletc(lubm:subOrganizationOfTC).
```

5.4.4. LUBM queries

Definition 128. All SPARQL queries for this scenario are defined as follows:

```
 lubm_query1:  SELECT ?s WHERE {?s rdf:type lubm:GraduateStudent.}
 lubm_query2:  SELECT ?s ?n WHERE {?s lubm:takesCourse ?n.}
 lubm_query3:  SELECT ?s ?n WHERE {?s lubm:advisor ?n.}
 lubm_query4:  SELECT ?s ?n WHERE {?s lubm:teacherOf ?n.}
 lubm_query5:  SELECT ?s WHERE {?s rdf:type lubm:Student.}
 lubm_query6:  SELECT ?s WHERE {?s rdf:type lubm:Faculty.}
 lubm_query7:  SELECT ?s WHERE {?s rdf:type lubm:Course.}
 lubm_query8:  SELECT ?s WHERE {?s rdf:type lubm:FullProfessor.}
 lubm_query9:  SELECT ?s ?n WHERE {?s lubm:memberOf ?n.}
lubm_query10:  SELECT ?s ?n WHERE {?s lubm:undergraduateDegreeFrom ?n.}
lubm_query11:  SELECT ?s WHERE {?s rdf:type lubm:University.}
lubm_query12:  SELECT ?s WHERE {?s rdf:type lubm:Department.}
lubm_query13:  SELECT ?s ?n WHERE {?s lubm:subOrganizationOf ?n.}
lubm_query14:  SELECT ?s ?n WHERE {?s lubm:subOrganizationOfTC ?n.}
```

5.5. Scenario 4 - Limitations and Extensions

5.5.1. Well-Founded Semantics

The known win-not-win test introduced by [Gelder et al., 1991] is taken to show recursion and default negation. For this test the generated EDB is cyclic. We expect problems with the rewriting to SQL, because the program is only evaluable under well-founded or stable-model semantics.

Definition 129. The program consists of a single recursive rule:

```
#import(wnw_move).
% Query and Main
win(X) :- wnw_move(X,Y), not win(Y).
```

5.5.2. Combining DLV$^{\mathrm{DB}}$ with generated Owlgres queries

The similarity between MOR and DLV$^{\mathrm{DB}}$ suggest, that DLV$^{\mathrm{DB}}$ could be extendend with a DL plug-in. This test should give a first insight into such an extension. As a first step, all needed DB views are generated by MOR. Then, the DB views will be imported as the EDB to DLV$^{\mathrm{DB}}$. Finally, DLV$^{\mathrm{DB}}$ will be run on top of the Owlgres DB views. We will use the simple LUBM benchmark of Definition 125 with 1,000,000 assertions for this test.

6. Experimental Results

In this chapter, we present and discuss the results for our experiments. The test were performed on a server running on openSUSE 11.1 (x86_64) with the following specification:

- Processor (CPU): Intel® Xeon® CPU E5450 @ 3.00GHz;
- Total memory (RAM): 15.7 GB.

The standard installation of PostgreSQL 8.4 in openSUSE 11.1 was taken. To utilize the available RAM, the parameters `shared_buffers` was set to 4096 MB (from 32 MB) and `work_mem` was set to 512 MB (from 1 MB) . These recommendations for performance optimization were taken from PostgreSQL wiki.[1] Further optimization of PostgreSQL, such as creating DB indices, was not considered for the Datalog scenario, but for the combined scenarios the DL reasoner Owlgres creates useful indices on its initialization.

For every system/benchmark combination, the experiment followed the following procedure:

1. Initializing PostgreSQL to clear its cache.
2. Five rounds of calls with the respective tool and the respective benchmark's program file are executed, whereby start- and finishing times are logged. To avoid any bias in measuring, the result output is turned off for every system.
3. After a timeout of 12 hours the particular experiment is canceled.
4. The average of the five execution times is taken to calculate the final result, which is kept in seconds.

Notice, that we consciously decided to initialize PostgreSQL only at the beginning, so the query optimizer of PostgreSQL could take advantage of cached data. In Section 6.1.1 we will discuss the results related to caching.

To validate the correctness of every benchmark, the resulting sets were counted and compared with the other systems. We did not encounter any differences in the resulting sets.

[1] http://wiki.postgresql.org/wiki/Performance_Optimization

6.1. Scenario 1 - Datalog

6.1.1. Large Join Benchmark

The results (see Figure 6.1) show that MOR and DLV^{DB} have linear runtime behavior[2], whereby MOR is about 45% faster than DLV^{DB}. We explain this difference, even both are based on an RDBMS, that in DLV^{DB} the IDB is temporarily materialized, where in MOR the IDB is rewritten into DB views. We refer regarding the details of materialization in DLV^{DB} to [Terracina et al., 2008]. By importing the EDB from the RDBMS and evaluating the IDB internally, DLV falls behind after 10,000 instances.

n	MOR	DLV	DLV^{DB}
10000	0	2	0
100000	2	270	3
250000	5	2274	9
500000	10	8096	18
1000000	22	32644	42
	Time (s)		

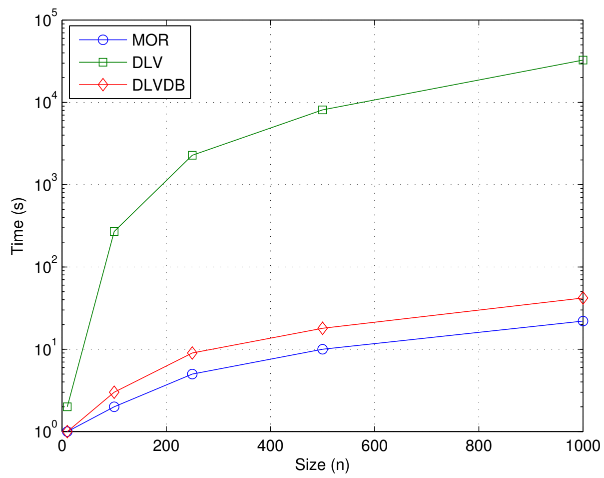

Figure 6.1.: Large Join Results

In Table 6.1 the detailed log for a single experiment with MOR is shown. We suggest, that the results of the particular rounds are almost identical, hence we can conclude that caching with 5 repetition has no effect on the evaluation time.

Round	Start	End	Result (s)
1	19:30:37	19:30:59	00:00:22
2	19:30:59	19:31:21	00:00:22
3	19:31:21	19:31:42	00:00:21
4	19:31:42	19:32:04	00:00:22
5	19:32:04	19:32:26	00:00:22

Table 6.1.: Log for MOR with Large Join Benchmark

[2]We use runtime behavior (e.g. linear) in a descriptive sense and not as the worst-case behavior.

6.1.2. Default Negation Benchmark

This benchmark (see Figure 6.2) gives a similar picture as the previous. An interesting point is that the intermediate result of this benchmark is identical with the previous benchmark. Still the performance for any system is by 50% better. This indicates that the optimizer of PostgreSQL calculates reasonably the set difference before the set union.

n	MOR	DLV	DLV^{DB}
10000	0	1	0
100000	1	105	1
250000	3	977	4
500000	5	2795	9
1000000	11	11446	19
	Time (s)		

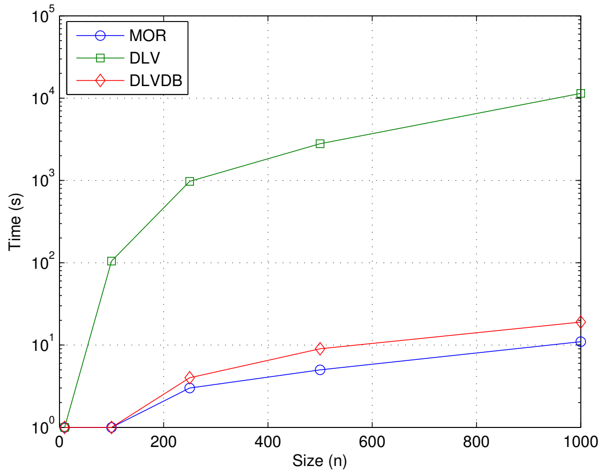

Figure 6.2.: Default Negation Results

6.1.3. Stratified Negation Benchmark

In Figure 6.3 we can observe that MOR and DLV^{DB} have an outlier with 250,000 instances. So we do not have a smooth curve for any system. We still can reveal, that MOR is about 3 times faster than DLV^{DB}. DLV exceeded the timeout of 12 hours with 250,000 instances.

A reason for the uneven curve is related to the test data. The data for this benchmark is randomly generated, so depth and occurrence of cycles can vary from instance to instance.

For example, the instances of size 500,000 and 1,000,000 contain cycles, which have a profound influence on the runtime behavior of MOR. As already discussed in Chapter 4, MOR uses the `LIMIT` parameter for breaking cycles in recursive queries in SQL:1999. After we had changed the `LIMIT` parameter from 100,000,000 to 2,000,000, the runtime behavior of MOR improved drastically:

- For the instance size of 500,000 runtime decreased from 3,221 to 17 seconds;
- For the instance size of 1,000,000 runtime dropped from 16,856 to 21 seconds.

This result indicates, that the `LIMIT` parameter has a strong impact on the performance of recursive query evaluation in PostgreSQL 8.4.

n	MOR	DLV	DLV^{DB}
10000	0	1	1
100000	1	254	5
250000	16	timeout	255
500000	17	timeout	56
1000000	21	timeout	239
	Time (s)		

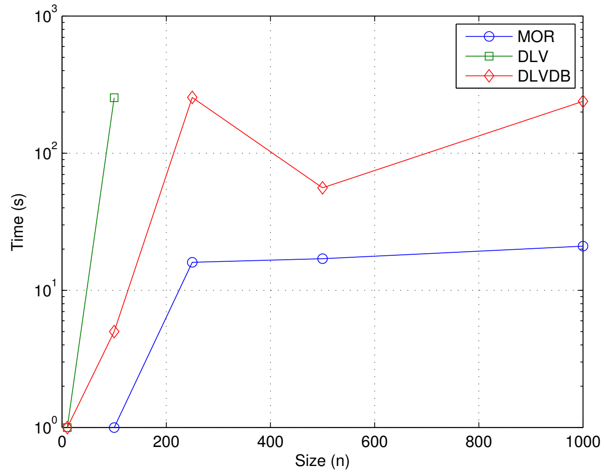

Figure 6.3.: Stratified Negation Results

6.2. Scenario 2 - Derived DBpedia

This benchmark, having a very simple TBox, should not pose any problem for the tested systems. Generic books and periodicals were additionally generated to reach the desired instance size. Note that the instance size is measured in total of assertions and not total of individuals.

6.2.1. Simple Benchmark

The result in Figure 6.4 shows that MOR performs nicely and exhibits almost linear runtime behavior.

Unfortunately we could not test dlvhex with an instance size bigger than 10,000 (resp. 100,000). The reasoner RacerPro 1.9.1 is used in the dlvhex DL plug-in. The following error was thrown by RacerPro:

```
‘‘Plugin Error in dlC["dbp_100.rdf",a,b,c,d, "yago:Book106410904"](X) in line 10:
An explicit gc call caused a need for 139460608 more bytes of heap.  The
operating system will not make the space available because of a lack of swap
space or some other operating system imposed limit or memory mapping collision.’’
```

The AllegroGraph Library probably causes this error, which is used by RacerPro 1.9.1 as the RDF triple store [Racer Systems GmbH & Co. KG, 2009]. The AllegroGraph Library has in its Free Edition a limited heap size of 60 MB.[3]

[3]http://www.franz.com/downloads/

n	MOR	dlvhex
10000	1	7
100000	1	error
250000	2	error
500000	3	error
1000000	5	error
	Time (s)	

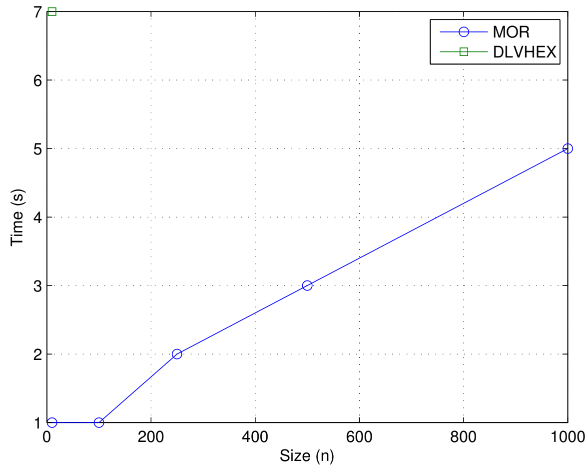

Figure 6.4.: DBpedia Simple Results

6.2.2. Advanced Benchmark

After introducing negation, still linear runtime behavior of MOR can be observed (see Figure 6.5).

n	MOR	dlvhex
10000	1	7
100000	1	error
250000	2	error
500000	4	error
1000000	7	error
	Time (s)	

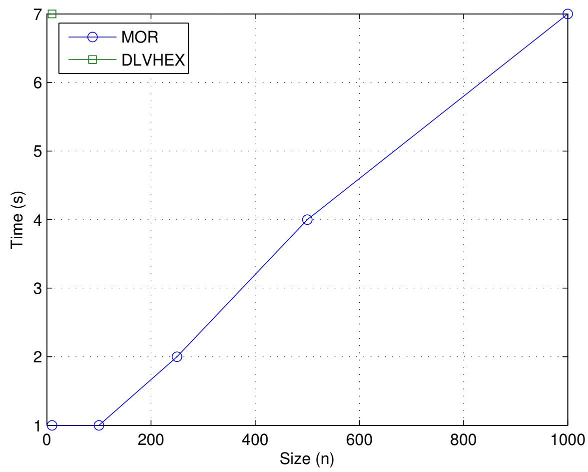

Figure 6.5.: DBpedia Advanced Results

6.2.3. Update Benchmark

Again we observe similar runtime behavior of MOR as in the previous benchmarks. Furthermore we observed an interesting behavior of dlvhex. For the instance size of

10,000, dlvhex has the same runtime in any DBpedia benchmark. This observation indicates, concerning the small instance size, that the main share of runtime is not used for instance retrieval, but for initializing the system.

n	MOR	dlvhex
10000	1	7
100000	2	error
250000	4	error
500000	7	error
1000000	13	error
	Time (s)	

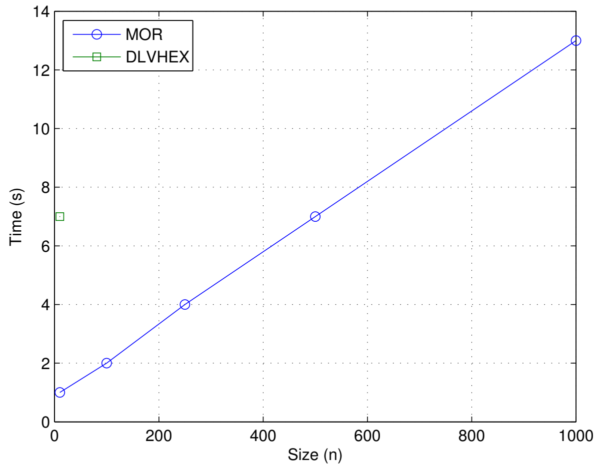

Figure 6.6.: DBpedia Update Results

6.3. Scenario 3 - Derived LUBM

Compared to the DBpedia scenario, the LUBM TBox is considerable more complex. Still the degree of inference is mainly based on sub-classes, sub-properties, and transitivity relations. The standard LUBM TBox could not be loaded because $DL\text{-}Lite_R$ and OWL 2 QL do not support transitive properties and intersection in superclass expressions [Motik et al., 2009]. After removing these expressions from the TBox, loading with Owlgres succeeded.

6.3.1. Simple Benchmark

As already described in Section 6.2.1, we encounter again the error with RacerPro. Fortunately, also instances with size 100,000 could be benchmarked, giving a better insight into runtime behavior of dlvhex.

For MOR, the first three instances are below one second, hence the results are not fine grained enough to draw any conclusion for its behavior. Still we can say, that the overall performance for this benchmark is very good.

n	MOR	dlvhex
10000	1	13
100000	1	41
250000	1	error
500000	2	error
1000000	3	error
	Time (s)	

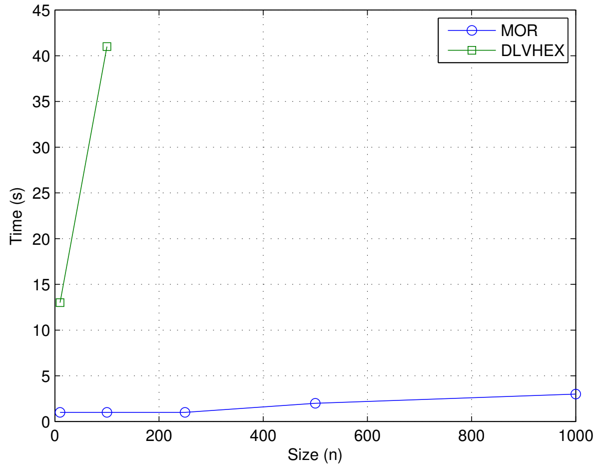

Figure 6.7.: LUBM Simple Results

6.3.2. Advanced Benchmark

The results (see Figure 6.8) illustrate that MOR has again a linear runtime behavior, where MOR is about 30 times faster than dlvhex. Of all the benchmarks, this benchmark has the longest absolute runtime for the large instances. This might be explained by the structure of the program. Four relations and one negated relation are joined, whereby all relations have a low selectivity among the join attributes. The low selectivity cancels the advantage of RDBMS, which use index techniques like the B-tree index.

n	MOR	dlvhex
10000	1	36
100000	4	117
250000	11	error
500000	20	error
1000000	44	error
	Time (s)	

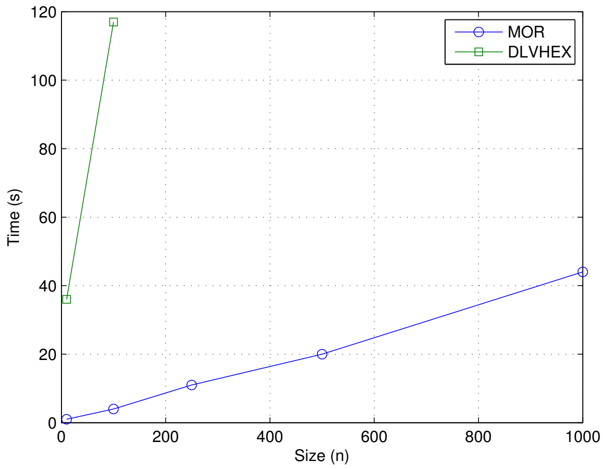

Figure 6.8.: LUBM Advanced Results

6.3.3. Update Benchmark

This benchmark is regarding expressivity the most interesting one. The transitive closure of a DL query is calculated in the Datalog program and injected back to the DL KB. Then the injected tuples are accessed in another DL query. The performance issues from Section 6.1.3 did not effect these results, due to the acyclicity of the generated data. Again we observed nice linear runtime behavior of MOR (see Figure 6.9).

n	MOR	dlvhex
10000	1	35
100000	1	108
250000	2	error
500000	4	error
1000000	11	error
	Time (s)	

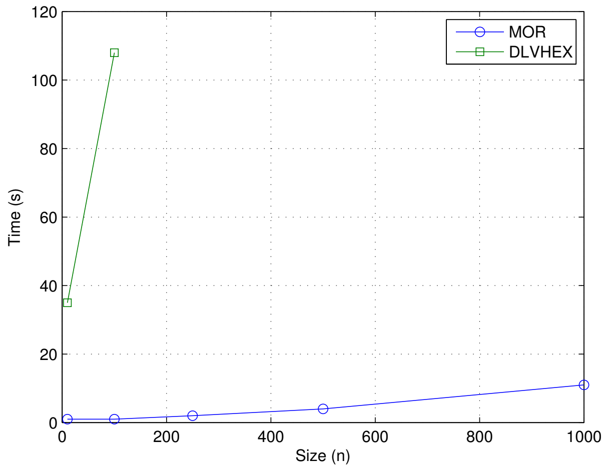

Figure 6.9.: LUBM Update Results

6.4. Scenario 4 - Limitations and Extensions

6.4.1. Well-Founded / Stable-Model Semantics

Even the Datalog program "`win(X) :- wnw_move(X,Y), not win(Y).`" can be rewritten to SQL as:

```
WITH RECURSIVE win(att1) AS (
 SELECT att1 FROM wnw_move AS mp
 UNION
 SELECT m.att1 FROM win AS r, wnw_move as m
   WHERE NOT (m.att2 IN ( SELECT att1 FROM win))
) SELECT * FROM win LIMIT 1000 ;
```

Executing this statement on PostgreSQL will lead to the following error:

"`ERROR: recursive reference to query "win" must not appear within a subquery`".

This shows, that more expressive program classes than stratified programs are rewritable to SQL:1999, but can not be evaluated on an RDBMS like PostgreSQL.

6.4.2. Combining DLV$^{\mathrm{DB}}$ with Owlgres

We compare DLV$^{\mathrm{DB}}$ results with the already existing results of MOR in Benchmark 6.3.1. We choose 1,000,000 instances for the experiment and proceeded the following way:

1. DB views for accessing the DL KB were created. This was achieved within 3 seconds by MOR.
2. At this point it was possible to run DLV$^{\mathrm{DB}}$, where the previous created DB views were imported to DLV$^{\mathrm{DB}}$. Evaluating the program resulted in an average runtime of about 12 seconds (see Table 6.2).

Round	Start	End	Result (s)
1	17:43:10	17:43:22	00:00:12
2	17:43:22	17:43:33	00:00:11
3	17:43:33	17:43:45	00:00:12
4	17:43:45	17:43:56	00:00:11
5	17:43:56	17:44:08	00:00:12

Table 6.2.: Log for DLV$^{\mathrm{DB}}$ with the LUBM Simple Benchmark

The same evaluation in MOR was in average about 4 times faster (3 seconds in MOR to 12 seconds in DLV$^{\mathrm{DB}}$). Yet this result let us conclude, that is feasible and encouraging to combine DL-Lite reasoning with DLV$^{\mathrm{DB}}$.

6.4.3. Summary of Results

The results for MOR indicate, that in most benchmarks almost linear runtime behavior was observed. One exception was Benchmark 6.1.3, where runtime behavior seems to be unpredictable. We suggest, that this is related to the depth and occurrence of cycles in the randomly generated test data. We also observed that the `LIMIT` parameter has a strong impact on the performance of recursive query evaluation in PostgreSQL 8.4. Besides influencing the performance, using the `LIMIT` parameter can lead to incomplete results, if its size is chosen lower than the expected result set. We conclude that the native implementation of linear recursive queries in PostgreSQL 8.4 as a breadth-first search is not favourable (see Section 4.3.4).

The results for the benchmarks in Scenario 1 show, that MOR and DLV$^{\mathrm{DB}}$ have similar runtime behavior. DLV$^{\mathrm{DB}}$ is materializing the IDB temporarily, which leads to the fact, that MOR is constantly about 45% faster than DLV$^{\mathrm{DB}}$. Furthermore we showed in Benchmark 6.4.2, that is encouraging to combine $DL\text{-}Lite_R$ reasoning with DLV$^{\mathrm{DB}}$.

As seen in Section 6.2.1, we encountered an error with the DL plug-in and RacerPro 1.9.1 for large instances, because of the limitation of the heap size in RacerPro. But we observed with the working benchmarks, that MOR is usually about 30 times faster than dlvhex.

In Benchmark 6.3.3 we showed, that with MOR it is possible to update the $DL\text{-}Lite_R$ KB with results, which were calculated in the Datalog program as the transitive closure of a DL query. The calculation of the transitive closure is interesting, because the unsupported transitive properties in $DL\text{-}Lite_R$ KB can be simulated. With Benchmark 6.4.1 we observed that stratified programs are not rewritable with MOR.

7. Conclusion

In this thesis, we presented a novel approach to efficiently evaluate dl-programs on an RDBMS. Due to the expressive power of SQL, we restricted the dl-programs to stratified semantics and linear recursion. To show the feasibility of such an approach, we developed the prototype implementation MOR, which interfaces the DL-Lite reasoner Owlgres and uses the RDBMS PostgreSQL 8.4. We followed with MOR the approach of rewriting the dl-programs fully into a cascade of DB views, not materializing any intermediate results. Still the query evaluation engine of PostgreSQL could process these complex, recursive SQL statements. The prototype was then benchmarked in four different scenarios to the DLV family of reasoners. For the combined scenarios, we designed our own benchmarks based on a DBpedia and LUBM KB.

7.1. Evaluation Results

With the experimental setup, MOR outperformed all the involved systems, namely DLV, DLV^{DB} and dlvhex. The reason for this encouraging results can mainly be accounted to the power of SQL optimizers of modern RDBMSs like PostgreSQL. These RDBMSs are designed to process vast amount of data, hence MOR and the incorporated DL-Lite reasoner Owlgres are focused to rewrite dl-programs to complex SQL statements and receive the evaluated results from the RDBMS.

In the purely Datalog scenario, MOR and DLV^{DB} showed similar runtime behavior, where MOR is constantly about 45% faster. DLV using ODBC, which never was designed for large scale data processing, is considerably slower than the other two systems. Evaluating recursive queries with PostgreSQL needed particularly attention, because cycles in the EDB lead to non-termination. We introduced the `LIMIT` parameter of SQL:1999 to avoid non-termination, however the runtime behavior of MOR is sensitive to this parameter.

In the combined scenarios, MOR performed remarkably better than dlvhex. In all of the benchmarks, MOR showed almost linear runtime behavior. This behavior was even apparent in the most interesting benchmark, where the transitive closure of a role query is calculated in the Datalog program, injected back to the DL KB. Then the DL KB is queried again to obtain the final result.

Finally, we showed the feasibility of using DLV^{DB} with DL-Lite reasoning. This was achieved by prepossessing the DL-Lite queries and saving the created SQL statements in DB views. After that, the DB view were imported by DLV^{DB}. The overall performance of this approach was by a factor four slower than MOR.

7.2. Future Work and Further Studies

We recognize the overlapping fields of theory, implementation, benchmarks, and practice, where research for the evaluation of dl-programs on an RDBMS could be pushed further.

On the theoretical side, it would be appealing to extend the introduced stratified semantics to well-founded or even answer-set semantics. Clearly, the evaluation of these semantics should be natively on the RDBMS, so we could follow our approach of utilizing the power of query optimizers in RDBMSs.

Regarding the implementation of MOR, we need to overcome the limitation mentioned in Chapter 4. For example, a more advanced handling of cycles regarding recursive queries could be developed, a generic SQL plug-in is missing, and the restriction to linear recursion could be overcome. Furthermore, MOR could be adapted, so that it would be usable as a Java component by other systems. For example, MOR could be coupled with the Jena framework, figuring as another rule-based inference engine. Concerning the early release state of Owlgres, further plug-ins to other DL-Lite reasoners (e.g. Quonto) could be written. Moreover, a similar implementation to MOR could be applied to DLV^{DB}.

The benchmarks could be extended with different tests regarding size and scenarios. A very interesting extension could contain large data sets with 200 million assertions as it is used in the Berlin SPARQL Benchmark[1]. Besides, MOR could be compared with other systems (e.g. Prolog based reasoners) besides the DLV family.

Getting back to the introductory example of a "smart" route planner, the use of MOR in a "real-world" project would give further insight into practical issues regarding the combination of rules and ontologies. As Johann Wolfgang von Goethe captures it in Faust [Goethe and Prudhoe, 1974]:

> "Dear friend, all theory is gray, And green the golden tree of life."

[1] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

A. Installation and Use:

Briefly, we outline how MOR can be installed on a Linux system. Since MOR was developed in Java 1.6, it should be easy deployable on other operating systems. The source code and the compiled binary files are available on:

`http://code.google.com/p/dbmor/`

A.1. Prerequisites

Basically, every RDBMS supporting SQL:1999 could be used, but we recommend to use PostgreSQL 8.4. There are several OWL2 QL reasoners available, but only Owlgres 0.1 is supported. It is mandatory to use our branch of Owlgres, due to some adaptions done in the main branch of Owlgres. Furthermore the JGraphT library is needed. We deliver the binary files for Owlgres and JGraphT with the source code.

A.2. Installation

After the binary files are deployed, the following steps have to be processed to setup the installation:

1. Create a PostgreSQL database with `createdb`, for example: `createdb -U myuser mydb`.
2. Check compatibility of the Owlgres TBox by `sh/expchk --tbox data/tbox_sample.rdf`. Note, that Owlgres only supports data serialized in RDF/XML.
3. Create the Owlgres database by loading the TBox: `sh/create --db mydb --user myuser --passwd mypw --tbox data/tbox_sample.rdf`.
4. Load the Owlgres ABox into the database: `sh/load --db mydb --user myuser --passwd mypw --abox data/abox_sample.rdf`.
5. Create custom tables and import the data from other databases. This step is needed, if the dl-program imports the EDB from external data sources.

A.3. Calling MOR from the Command Line

We provide the shell script `mor.sh` to call MOR. The following parameters are valid inputs:

`--pgm`	URI of the dl-program.
`--host`	Database host, where the default is localhost.
`--db`	Database name.
`--user`	Database user.
`--passwd`	Database password.
`--silent`	Status messages are suppressed.
`--keepviews`	Keeps the created database views, otherwise all intermediate views are dropped after execution.

Bibliography

ISO/IEC 9075–1:1999, Information Technology – Database languages – SQL – part 1: Framework. Technical report, 1999.

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995. ISBN 0-201-53771-0.

Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, 1988. ISBN 0-934613-40-0.

Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics. In *Handbook on Ontologies*, pages 3–28. 2004.

Franz Baader, Carsten Lutz, and Boontawee Suntisrivaraporn. Efficient reasoning in $l+$. In *Proceedings of the 2006 International Workshop on Description Logics (DL2006), Windermere, Lake District, UK, May 30 - June 1, 2006*, 2006.

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001. ISSN 0036-8733. URL `http://www.scientificamerican.com/article.cfm?id=the-semantic-web`.

Erol Bozsak, Marc Ehrig, Siegfried Handschuh, Andreas Hotho, Alexander Maedche, Boris Motik, Daniel Oberle, Christoph Schmitz, Steffen Staab, Ljiljana Stojanovic, Nenad Stojanovic, Rudi Studer, Gerd Stumme, York Sure, Julien Tane, Raphael Volz, and Valentin Zacharias. Kaon - towards a large scale semantic web. In *E-Commerce and Web Technologies, Third International Conference, EC-Web 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*, pages 304–313, 2002.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Dl-lite: Tractable description logics for ontologies. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 602–607, 2005.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *dl-lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Springer, 1990. ISBN 3-540-51728-6.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13 (6):377–387, 1970.

Alain Colmerauer. Prolog in 10 figures. *Commun. ACM*, 28(12):1296–1310, 1985.

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. In *IEEE Conference on Computational Complexity*, pages 82–101, 1997.

Jos de Bruijn, Thomas Eiter, Axel Polleres, and Hans Tompits. On representational issues about combinations of classical theories with nonmonotonic rules. In *Knowledge Science, Engineering and Management, First International Conference, KSEM 2006, Guilin, China, August 5-8, 2006, Proceedings*, pages 1–22, 2006.

Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The second answer set programming competition. In *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, pages 637–654, 2009.

Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, June 2-5, 2004*, pages 141–151, 2004.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A prover for semantic-web reasoning under the answer-set semantics. In *2006 IEEE / WIC / ACM International Conference on Web Intelligence (WI 2006), 18-22 December 2006, Hong Kong, China*, pages 1073–1074, 2006.

Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.*, 51(2-4):123–165, 2007.

Thomas Eiter, Georg Gottlob, Magdalena Ortiz, and Mantas Simkus. Query answering in the description logic horn-. In *Logics in Artificial Intelligence, 11th European Conference, JELIA 2008, Dresden, Germany, September 28 - October 1, 2008. Proceedings*, pages 166–179, 2008a.

Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artif. Intell.*, 172(12-13):1495–1539, 2008b.

Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, pages 40–110, 2009a.

Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, and Roman Schindlauer. Well-founded semantics for description logic programs in the semantic web. Technical Report INFSYS RR-1843-09-01, March 2009b. To appear in ACM Transactions on Computational Logic (revised).

Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. An overview and introduction to logic and data bases. In *Logic and Data Bases*, pages 3–30, 1977.

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. ISBN 978-0-201-63361-0.

Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *lasp* : A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, pages 260–265, 2007a.

Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Miroslaw Truszczynski. The first answer set programming system competition. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, pages 3–17, 2007b.

Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pages 1070–1080, 1988.

Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

Johann Wolfgang von Goethe and John Edgar Prudhoe. *Faust, the tragedy. Part 1; translated for the English speaking stage by John Prudhoe.* Manchester University Press; Barnes & Noble, Manchester, New York, 1974. ISBN 0719005701.

Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *Proceedings of the World Wide Web Conference (WWW2003)*, pages 48–57, 2003.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

Carl Hewitt. Middle history of logic programming. *CoRR*, abs/0904.3036, 2009.

Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Open answer set programming for the semantic web. *J. Applied Logic*, 5(1):144–169, 2007.

Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining owl and ruleml. W3C submission, W3C, October 2004. URL `URLhttp://www.w3.org/Submission/SWRL/`.

Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Data complexity of reasoning in very expressive description logics. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 466–471, 2005.

Michael Kifer. Rule interchange format: The framework. In *Web Reasoning and Rule Systems, Second International Conference, RR 2008, Karlsruhe, Germany, October 31-November 1, 2008. Proceedings*, pages 1–11, 2008.

Markus Krötzsch, Peter F. Patel-Schneider, Sebastian Rudolph, Pascal Hitzler, and Bijan Parsia. OWL 2 web ontology language primer. Technical report, W3C, October 2009. URL `http://www.w3.org/TR/2009/REC-owl2-primer-20091027/`.

Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0130925691.

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

Alon Y. Levy and Marie-Christine Rousset. Combining horn rules and description logics in carin. *Artif. Intell.*, 104(1-2):165–209, 1998.

Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 601–610, 2009.

John W. Lloyd. *Foundations of Logic Programming, 1st Edition*. Springer, 1984. ISBN 3-540-13299-6.

Li Ma, Yang Yang, Zhaoming Qiu, Guo Tong Xie, Yue Pan, and Shengping Liu. Towards a complete owl ontology benchmark. In *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings*, pages 125–139, 2006.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

Jing Mei, Harold Boley, Jie Li, Virendrakumar C. Bhavsar, and Zuoquan Lin. Datalog$^{\mathrm{dl}}$: Datalog rules parameterized by description logics. In *Canadian Semantic Web, CSWWS 2006, first Canadian Semantic Web Working Symposium, June 2006, Quebec, Canada*, pages 171–187, 2006.

Boris Motik, Ian Horrocks, Riccardo Rosati, and Ulrike Sattler. Can owl and logic programming live together happily ever after? In *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, pages 501–514, 2006.

Boris Motik, Achille Fokoue, Ian Horrocks, Zhe Wu, Carsten Lutz, and Bernardo Cuenca Grau. OWL 2 web ontology language profiles. W3C recommendation, W3C, October 2009. URL `http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/`.

Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany, July 28-31, 1997, Proceedings*, pages 421–430, 1997.

Christos H. Papadimitriou. *Computational Complexity.* Addison Wesley, December 1993. ISBN 0201530821.

Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. A comparison of query rewriting techniques for dl-lite. In *Proceedings of the DL Home 22nd International Workshop on Description Logics (DL 2009), Oxford, UK, July 27-30, 2009*, 2009.

Racer Systems GmbH & Co. KG. *Release Notes for RacerPro 1.9.2 beta.*, March 2009. URL `http://www.racer-systems.com/products/racerpro/release-notes-1-9-2-beta.pdf`.

Raymond Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.

John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

Riccardo Rosati. Dl+log: Tight integration of description logics and disjunctive datalog. In *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 68–78, 2006.

Riccardo Rosati and Alessandro Almatelli. Improving query answering over dl-lite ontologies. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*, 2010.

Guus Schreiber and Mike Dean. OWL web ontology language reference. W3C recommendation, W3C, February 2004. URL `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`.

Markus Stocker and Michael Smith. Owlgres: A scalable owl reasoner. In *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*, 2008.

Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Inf.*, 6: 171–185, 1976.

Alfred Tarski. On the calculus of relations. *J. Symb. Log.*, 6(3):73–89, 1941.

Giorgio Terracina, Nicola Leone, Vincenzino Lio, and Claudio Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP*, 8(2):129–165, 2008.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988. ISBN 0-7167-8158-1.