Accelerating Rule-Based Reasoning in Disk-Resident RDF Knowledge Bases

Masterarbeit im Fach Informatik
Master’s Thesis in Computer Science
von / by
Mohamed Yahya

angefertigt unter der Leitung von / supervised by
Dr. Martin Theobald

begutachtet von / reviewers
Dr. Martin Theobald
Prof. Dr. Gerhard Weikum

August 2010
Non-plagiarism Statement

I, Mohamed Yahya, hereby confirm that this thesis is my own work and that I have documented all sources used.

Signed: 

Date: 
“A few observation and much reasoning lead to error; many observations and a little reasoning to truth.”

Alexis Carrel; 1912 Nobel Prize in Physiology or Medicine.
Abstract

Collections of tens of millions of automatically extracted facts represented using the subject-predicate-object RDF model are available for several domains. As big as these collections are, they are unable to capture all information about a domain, simply because the sources from which they were extracted are incomplete. This can be tackled by creating knowledge bases where facts are enforced with rules showing how new facts can be generated from existing ones and constraints which must hold in the relevant domain. Querying such knowledge bases is expensive for two main reasons. First, data is disk resident, which makes access to it slow. Secondly, rule definitions can be recursive, which requires special query evaluation techniques and renders traditional cost-based query optimization and join-ordering techniques less effective.

This thesis presents the implementation of a query processor for such a setting. We show how we integrated the RDF-3X RDF query engine into our query processor. We also present optimizations along several dimensions: (i) query evaluation techniques, (ii) caching to reduce both disk access and rule evaluation, (iii) a classification of predicates which allows better utilization of the underlying storage engine’s ability to optimize traditional relational queries and (iv) a novel probabilistic way of looking at join ordering and cost estimation in this context.
Acknowledgements

First and foremost, I am deeply grateful to my mother, father and siblings for their support. I would not be where I am today without their prayers, love and encouragement.

I am indebted to my advisor, Dr. Martin Theobald, for his guidance and support during my thesis. He provided a nice balance of motivation to keep me going and space to pursue things that interest me. He also gave me enough room to make mistakes that I now value as precious experience.

I am also thankful to Prof. Gerhard Weikum and the members of the Database and Information Systems Group. They provided a motivating work environment. I have learned a great deal about conducting research by observing the members of the group.

I would also like to thank Prof. Dittrich, whose lecture on Database Systems was key for me to pursue research in this field. My tutoring experience in the second offering of the same lecture was a valuable experience.

My friends at MPI and Saarland University were my family away from home. I am thankful to every single one of them for their support.

Finally, I would also like to thank the German Academic Exchange Service (DAAD), particularly Dr. Helga Baumgarten, for supporting me during my Master’s studies. I am also grateful to Mrs. Jennifer Gerling and Mrs. Stephanie Jörg of the International Max Planck Research School for Computer Science for their help.
# Contents

Abstract iii  

Acknowledgements iv  

List of Figures viii  

List of Tables ix  

1 Introduction 1  
1.1 Motivation and Problem ........................................ 1  
1.2 Contributions ..................................................... 2  
1.3 Outline .............................................................. 3  

2 Related Work: Deductive Databases and the Semantic Web 4  
2.1 Datalog: the Language Of Deductive Databases ............... 4  
2.1.1 Syntax .......................................................... 5  
2.1.2 Semantics ....................................................... 6  
2.1.3 Deductive Database Systems versus Logic Programming ... 8  
2.2 The RDF Data Model ............................................... 9  
2.3 Datalog and the Semantic Web .................................... 10  
2.4 Recursive Query Evaluation ..................................... 11  
2.4.1 The Proof Theory of Datalog ................................. 11  
2.4.2 Bottom-up Query Evaluation .................................. 13  
2.4.3 Top-down Query Evaluation .................................. 14  
2.4.4 On the Complexity of Recursive Query Evaluation ....... 17  
2.5 Join Ordering in Deductive Databases ......................... 18  
2.6 Built-in Predicates .................................................. 21  

3 URDF: Efficient Reasoning in Uncertain RDF Knowledge Bases with Soft and Hard Rules 22  
3.1 Motivation ........................................................... 22  
3.2 Representation Model ............................................. 24  
3.2.1 Extensional Database .......................................... 24  
3.2.2 Soft Rules ...................................................... 24  
3.2.3 Hard Rules ...................................................... 25  
3.3 Reasoning Model ................................................... 27
## Contents

4 RDF-3X: a RISC-style Engine for RDF

4.1 Storage and Indexing .................................................................. 31
4.2 Query Processing ...................................................................... 32
4.3 Query Optimization ................................................................... 33
4.4 Querying Large RDF Graphs

4.4.1 Sideways Information Passing ............................................. 34
4.4.1.1 Run-Time Handling of Merge Joins ............................... 35
4.4.1.2 Run-Time Handling of Hash Joins ............................... 35
4.4.2 Selectivity Estimation .......................................................... 36

5 Systems and Integration

5.1 System Architecture ................................................................. 38
5.2 Rule Store ............................................................................... 40
5.3 Storage Backend

5.3.1 Pre-processing .................................................................... 40
5.3.2 Query time ......................................................................... 42
5.4 Caching .................................................................................. 43
5.5 Subquery Scheduler ................................................................. 43
5.6 Recursive Query Processor

5.6.1 SLD-resolution .................................................................. 44
5.6.2 QSQR ................................................................................. 45
5.6.3 Memory Management ......................................................... 49
5.7 Reasoner Modes ..................................................................... 49

6 Optimizing Recursive Queries

6.1 Subquery Scheduling: Seeking Failure .................................... 52
6.2 Reasoner Modes ..................................................................... 58
6.3 Unification .............................................................................. 58

7 Experimental Evaluation and Results

7.1 Setting ...................................................................................... 61
7.1.1 System Configuration ........................................................ 61
7.1.2 Data Sets .......................................................................... 62
7.1.3 Experiments ...................................................................... 62
7.2 Results .................................................................................... 63
7.2.1 Handling of Extensional Queries ................................. 63
7.2.2 Unification ......................................................................... 63
7.2.3 Evaluation of LPF Subquery Scheduling Strategy .......... 65
7.2.4 Chaining .......................................................................... 68
7.3 Comparison ............................................................................ 70

8 Conclusion and Future Work

8.1 Future Work ........................................................................... 77

A Queries

A.1 Extensional Queries ................................................................. 80
# List of Figures

2.1 RDF triples and graphs ................................................. 9
2.2 SLD-resolution example .............................................. 16
2.3 Example of an infinite SLD derivation ............................... 16
2.4 Importance of atom ordering ......................................... 19
2.5 Sample rules for domain knowledge approach ...................... 20
3.1 URDF ................................................................. 23
3.2 Predicate renaming ................................................... 25
3.3 A knowledge base ..................................................... 27
3.4 The dependency graph of \( q = \leftrightarrow \text{bornIn}(Al\_Gore, ?y) \) with the knowledge base from Figure 3.3 ...................... 28
4.1 Join patterns in SPARQL queries .................................. 31
5.1 RDRD architecture .................................................... 39
5.2 Reasoner timeline .................................................... 39
5.3 A counterexample to the completeness of the formulation of QSQR in [AHV95] and [MBN08] ................................. 48
5.4 UML representation of a substitution ............................... 50
6.1 Predicate promise ..................................................... 56
6.2 Example of chaining .................................................. 57
6.3 Rule merging .......................................................... 58
7.1 Query times for extensional queries based on YAGO ............. 64
7.2 Times for \( 10^6 \) executions of MGU and NAIVE_UNIFY .......... 65
7.3 Query times in milliseconds for recursive queries based on YAGO using LPF and MBF .............................................. 67
7.4 Number of intermediate subqueries for recursive queries based on YAGO using LPF and MBF .............................................. 67
7.5 Query times for recursive queries based on YAGO with and without chaining ................. 69
7.6 Number of intermediate subqueries recursive queries based on YAGO with and without chaining ................................. 70
7.7 Comparison to other systems using queries based on YAGO - cold cache . 72
7.8 Comparison to other systems using queries based on YAGO - warm cache 73
7.9 Comparison to other systems using queries based on LUBM - cold cache . 75
7.10 Comparison to other systems using queries based on LUBM - warm cache 75
List of Tables

2.1 Complexity of queries .................................................. 18
7.1 Query times in milliseconds for extensional queries based on YAGO . . 64
7.2 Times for $10^6$ executions of MGU and NAIVE_UNIFY is milliseconds . . 65
7.3 Query times in milliseconds for recursive queries based on YAGO using LPF and MBF ................................................. 66
7.4 Number of intermediate subqueries for recursive queries based on YAGO using LPF and MBF ................................................. 66
7.5 Query times in milliseconds for recursive queries based on YAGO with and without chaining .............................................. 68
7.6 Number of intermediate subqueries recursive queries based on YAGO with and without chaining .............................................. 69
7.7 Comparison to other systems using queries based on YAGO, times in milliseconds ......................................................... 71
7.8 Comparison to other systems using queries based on LUBM, times in milliseconds ......................................................... 74
Chapter 1

Introduction

1.1 Motivation and Problem

RDF provides a simple, yet powerful, graph-based triples model for representing semantic web data. Information extraction has led to the availability of large knowledge bases, such as YAGO [SKW07], which are represented using the RDF data model. Because of their size, such knowledge bases are usually stored in disk-based database management systems. Special purpose RDF storage engines, such as RDF-3X [NW08], facilitate efficient storage and querying of large knowledge bases with billions of triples. Such engines generally rely on storage schemes and optimizations that are well understood from relational databases.

Such knowledge bases can exhibit incompleteness. This issue can be tackled by appending rules, which describe the knowledge base’s domain, to the knowledge base. Rules remedy incompleteness by allowing new facts to be established from existing ones. Take for instance a knowledge base dealing with a population which does not include the birth place of a specific person. One can add the following rules to the knowledge base to solve this problem:

\[
\text{bornIn}(\text{?person, ?place}) \leftarrow \text{isMarriedTo}(\text{?person, ?spouse}) \land \text{bornIn}(\text{?spouse, ?place})
\]

\[
\text{bornIn}(\text{?person, ?place}) \leftarrow \text{hasChild}(\text{?person, ?child}) \land \text{bornIn}(\text{?child, ?place})
\]

Stating that a person is born in the same place as his or her spouse or child.

These rules can give rise to inconsistencies, where constraints that must hold in a domain are violated. For instance, a person can have one birth place at most. For this, we can add additional rules describing constraints that should hold in a domain. The rule
based approach is a very natural one for RDF data, which is based on binary predicates (triples).

Rules defining predicates can be recursive: the definition of a predicate can, directly or indirectly, depend on itself. Both rules describing the `bornIn` predicate above are recursive. This moves us from the traditional relational setting, to a Datalog-like recursive one. In such a setting, traditional query optimization techniques do not work well because of the different access patterns to data and the difficulty of obtaining reliable statistics, which form the basis of query optimization. Moreover, disk access patterns are different than those expected in a relational setting.

### 1.2 Contributions

We make the following contributions in this thesis:

- An implementation of a recursive query processor for RDF data based on RDF-3X, which we call RDRD (Rule-based query processor for Disk-Resident Data), which has the ability to maintain lineage. This recursive query processor can serve either as a general-purpose deductive database for RDF or as a reasoner for the URDF framework (Chapters 5 and 6).

- We describe how we integrated RDF-3X into our system to serve as the storage backend for our recursive query processor (Section 5.3).

- We show the incompleteness of the presentation of the Query Subquery Recursive (QSQR) recursive query evaluation strategy in [AHV95], one of the most cited sources on QSQR, and propose a fix for this issue (Subsection 5.6.2).

- We propose a probabilistic join-ordering heuristic which (Section 6.1):
  - is dynamic, allowing decisions on join ordering to be made based on the latest binding patterns of a query,
  - uses a probabilistic model that focuses on detecting the failure of a subquery as soon as possible, and
  - allows for the chaining of extensional predicates, allowing for better utilization of the storage engine underlying the deductive database to optimize joins.
Chapter 1. Introduction

1.3 Outline

This thesis is organized as follows: In Chapter 2 we give background information on deductive databases and the semantic web. We introduce deductive databases, their language called Datalog and the RDF data model, which forms the basis of the semantic web, and related semantic web standards through the lens of Datalog. We also discuss query evaluation in a deductive database setting and consider the issue of join ordering. Chapter 3 presents the URDF framework for efficient reasoning in RDF knowledge bases with soft and hard rules. There, we show how URDF enables query answering in a setting where the base data of automatically extracted facts exhibits incompleteness and inconsistency. Chapter 4 presents RDF-3X, an engine for efficiently managing and querying large RDF graphs, which forms the storage backend of our reasoner for URDF. In Chapter 5 we present our implementation of a recursive query processor for URDF, which also serves as a deductive database. Chapter 6 introduces optimizations we added to the recursive query processor. There, we give a probabilistic approach to join ordering which focuses on detecting failure of a query as soon as possible. We give an experimental evaluation of our reasoner and a discussion of results in Chapter 7. Finally, in Chapter 8, we give the conclusions of this thesis and propose directions for future work.
Chapter 2

Related Work: Deductive Databases and the Semantic Web

A deductive database is a database where new facts can be derived from facts explicitly stated in the database using rules [GMN84]. This chapter introduces the basics of deductive databases and related semantic web standards through the lens of deductive databases.

Section 2.1 introduces the syntax of Datalog, the language for expressing facts, rules and queries in deductive databases, and discusses the differences between deductive databases and logic programming systems, such as Prolog. Section 2.2 briefly presents the data model behind RDF, and how such a model can be queried. Section 2.3 discusses the relation between Datalog and semantic web standards for representing ontologies and rules. Section 2.4 presents the evaluation of queries over deductive databases and the complexity of doing this. The problem of join ordering in deductive databases is presented in Section 2.5, along with some approaches for dealing with it. Finally, Section 2.6 discusses the handling of built-in predicates, such as equality, when evaluating queries over deductive databases.

2.1 Datalog: the Language Of Deductive Databases

Datalog is the language used for expressing facts, rules and queries in deductive databases. In this section, we introduce the syntax and semantics of Datalog.
2.1.1 Syntax

The syntax of Datalog is based on three sets:

- **Var**: an infinite alphabet consisting of all finite strings starting with a ‘?’. A member of Var is called a variable.
- **Const**: an infinite alphabet consisting of all finite strings not starting with a ‘?’. A member of Const is called a constant.
- **Pred**: an infinite set, \( \text{Pred} \subseteq \text{Const} \). A member of Pred is called a predicate.

**Definition 2.1** (Term). A term is a logical expression which refers to an object. In Datalog, a term \( t \) is defined as follows:

\[
t :::= v | c
\]

where \( v \in \text{Var} \), \( c \in \text{Const} \).

**Definition 2.2** (Predicate). A predicate \( p \in \text{Pred} \) with \( \text{arity}(p) = n \) is a mapping \( \text{Dom}^n \rightarrow \{ \text{true}, \text{false} \} \), where \( \text{Dom}^n \) is a domain. We say that \( p \) is an \( n \)-ary predicate.

**Definition 2.3** (Atom). An atom \( p(t_1, ..., t_n) \) is an \( n \)-ary predicate \( p \) applied to a tuple of terms \( (t_1, ..., t_n) \). Examples of atoms are: \( \text{hasWon}(\text{Max}\_\text{Planck}, \text{NobelPrize}) \), \( \text{visited}(?x, \text{Germany}) \).

**Definition 2.4** (Literal). A literal is an atom \( p(t_1, ..., t_n) \), called a positive literal, or its negation \( \neg p(t_1, ..., t_n) \), called a negative literal.

**Definition 2.5** (Clause). A clause is a finite set of literals.

An example of a clause is \( \{ \neg\text{eats}(?x, \text{meat}), \text{vegetarian}(?x) \} \). Clauses can be written in implication form. The clause \( c = A_1, A_2, ..., A_m, \neg B_1, \neg B_2, ..., \neg B_n \) can be written as:

\[
A_1, A_2, ..., A_m \leftarrow B_1, B_2, ..., B_n
\]

In this thesis, we will use the implication form of writing clauses. The clause given earlier can be written as

\[
\text{vegetarian}(?x) \leftarrow \text{eats}(?x, \text{meat})
\]

A term, atom, literal or clause are called **ground** if they contain no variables.
Definition 2.6 (Horn Clause). A horn clause is clause with at most one positive literal. Three kinds of horn clauses can be distinguished:

- **Facts.** Facts are clauses with exactly one positive literal. In this thesis we will use $A$ to denote the fact $A \leftarrow$. We will also restrict ourselves to ground facts.

- **Rules.** A rule is a definite clause: a horn clause with exactly one positive literal and at least one negative literal. The positive literal of a rule is called the head and the negative literals the body.

- **Queries.** A query (or a goal) is a clause with no positive literals. A subset of the literals in a query is called a subquery (or a subgoal). In this thesis we will write the query $\leftarrow B_1, B_2, ..., B_n$ as $\leftarrow B_1, B_2, ..., B_n$.

Definition 2.7 (Datalog Program). A Datalog program is a finite set of facts and safe rules in which all variables occurring in the head of a rule also occur in its body.

Predicates in a Datalog program can be divided into two sets: $EPred$ and $IPred$, where $EPred \cup IPred \subseteq Pred$ whereas $EPred \cap IPred = \emptyset$. $EPred$ contains predicates occurring only in the bodies of rules and in facts in a Datalog program. These predicates are said to be extensionally defined, they correspond to relations stored in a database. $IPred$ contains predicates occurring in the heads of rules in a Datalog program. These predicates are said to be intensionally defined through rules.

2.1.2 Semantics

The model-theoretic semantics of Datalog allows us to state which facts are true given a Datalog program. We first start with some basic definitions [LB10].

Definition 2.8 (Interpretation). An interpretation (or possible world) $I$ is a triple $< Dom, \phi, \pi >$, where:

- $Dom$ is a nonempty set called the domain.

- $\phi$ is a function, $\phi : Const \rightarrow Dom$

- $\pi$ is a function which assigns to each $n$-ary predicate a function $Dom^n \rightarrow \mathbb{B}$, where $\mathbb{B} = \{false, true\}$

Definition 2.9 (Variable Assignment). A variable assignment $\mathcal{I}$ is a function from variables to constants.
Definition 2.10 (Satisfaction). A ground atom \( p(t_1, \ldots, t_n) \) is not satisfied (false) by an interpretation \( I \) if \( \pi(p)(\phi(t_1), \ldots, \phi(t_n)) = false \), and it is satisfied (true) otherwise. A ground rule \( A \leftarrow B_1, B_2, \ldots, B_n \) is not satisfied by interpretation \( I \) if \( A \) is false in \( I \) and each \( B_i \) is true in \( I \), and it is satisfied otherwise. A non-ground clause is satisfied by an interpretation \( I \) if it is true for all variable assignments.

A Datalog program \( D \) is satisfied by an interpretation \( I \) if every clause in \( D \) is true in \( I \).

Definition 2.11 (Model). A model of a Datalog program is an interpretation in which all the clauses are true.

Definition 2.12 (Logical Consequence). If \( D \) is a Datalog program and \( q \) is a fact then \( q \) is a logical consequence of \( D \), written as \( D \models q \), if \( q \) is true in every model of \( D \).

Definition 2.12 tells us when a fact follows from a Datalog program, which is sufficient to define a semantics of Datalog from a theoretical point of view. This definition would require that constants be interpreted with arbitrary elements of the domain, which is clearly not what we want. To restrict the number of models we need to consider for defining the semantics of Datalog, we first restrict ourselves to looking at a special kind of interpretations: Herbrand interpretations.

Definition 2.13 (Herbrand Interpretation). A Herbrand interpretation is an interpretation where \( \phi \) maps every \( c \in Const \) to the lexicographic string making up \( c \).

A Herbrand interpretation can be thought of as an interpretation which interprets a constant as itself. Based on Theorem 2.14, it is possible for us to limit ourselves only to Herbrand interpretations when defining the semantics of Datalog.

Theorem 2.14. If any interpretation satisfies a given set of clauses \( S \), then there is a Herbrand interpretation that satisfies them.

All Herbrand interpretations have the same \( Dom \) and \( \phi \) as given in Definition 2.8, they only differ in \( \pi \), that is, they differ in the set of facts to which they assign true. A herbrand interpretation can be completely characterized by the set of facts to which it assigns true. Restricting ourselves to Herbrand interpretations corresponds to the very well known unique name assumption (UNA) where constants with different names are considered different.

We are now ready to define the (model-theoretic) semantics of Datalog.

Definition 2.15 (Semantics of Datalog). Let \( D \) be a Datalog program. The semantics of \( D \) is the minimum Herbrand model of \( D \).
Chapter 2. Related Work: Deductive Databases and RDF

The semantics of a Datalog program $D$, which itself is a Herbrand model of $D$, is equal to the intersection of all Herbrand models of $D$ and is always defined\(^1\). Defining the semantics of a Datalog program as the minimal Herbrand model (equivalently, the intersection of all Herbrand models) corresponds to the very well known **closed world assumption (CWA)** in which only those facts which must be true in all worlds modeled by the (deductive) database are taken to be true, while all others are defined to be false.

### 2.1.3 Deductive Database Systems versus Logic Programming

From the introduction of Datalog, it is possible to see a great resemblance to logic programming languages such as Prolog. Logic programming greatly influenced the development of Datalog for deductive database systems [Das92]. However, the two are directed towards different audiences and have different application domains.

Syntactically, terms in Datalog were defined as being either variables or constants. In logic programming, terms also admit function symbols. The safety condition, which requires that all variables in the head of a Datalog rule exist in its body also is not there in logic programming. Those two restrictions mean that Datalog programs will always have finite models, whereas this is not true for logic programs in general.

The terminology of deductive databases mostly comes from database terminology. Logic programming terminology finds its roots in mathematical logic and theorem proving. For instance, queries in deductive databases correspond to goals in logic programming, arguments to attributes, and facts to ground unit clause etc. Logic programming does not make the distinction between extensional and intensional predicates, while in deductive databases this distinction helps in efficient query processing.

An implementation of a deductive databases typically stresses efficiency. A deductive database is generally assumed to contain a large number of facts, making efficient data management the key issue. Logic programming systems, on the other hand, stress functionality and the efficiency there is focused on the inference mechanism. Deductive databases typically have interfaces which allow the manipulation of data they store, while logic programming systems usually lack the concept of data and have limited interfaces for manipulating clauses.

---

\(^1\)See [AHV95] for the proof.
2.2 The RDF Data Model

RDF was designed to facilitate the exchange of information about resources on the web. Due to its simplicity, it has been utilized in domains other than the web such as representing ontologies, social networks and personal information management.

In this section we will introduce the data model behind RDF through the lens of Datalog. We will not go into details of the syntax and semantics of the RDF standard available in [CK04] and [Hay04].

**Definition 2.16 (RDF Triple).** An **RDF triple** $p(s,o)$ is a fact whose positive literal is composed of a binary predicate $p$ and two arguments: the subject $s$ and the object $o$.

Conceptually, the subject corresponds to an entity, the predicate to a property of the entity and the object to a value of the property. Another way of writing down the RDF triple $p(s,o)$ is $spo$, which resembles Notation-3 syntax of RDF and the syntax of SPARQL. In this thesis we will use both notations interchangeably.

```
bornIn(Al_Gore,Washington_D.C.)
isMarriedTo(Al_Gore,Tipper_Gore)
bornIn(Tipper_Gore,Washington_D.C.)
hasParent(Al_Gore,Albert_Gore,Sr.)
```

(a) RDF Triples

```
Albert_Gore,Sr. hasParent Al_Gore bornIn Washington_D.C.
```

(b) RDF Graph

![RDF Graph](example_graph.png)

**Figure 2.1:** RDF triples and graphs

**Definition 2.17 (RDF Graph).** An **RDF Graph** is a set of RDF triples.

Figure 2.1 shows an example of a set of RDF triples (Figure 2.1(a)) and the corresponding RDF graph (Figure 2.1(b)).

The strength of RDF as a data model stems from its ability to represent a labeled multigraph where predicates act as edges connecting nodes, which, in turn, act as subjects or objects. In the rest of this thesis, we will deal exclusively with RDF predicates.

The standard language for querying an RDF graph is SPARQL [PS08]. A SPARQL query is composed of a set of triple patterns, each of which can contain either a constant or a variable in each of the subject, predicate and object positions. A subgraph of an RDF graph **matches** a triple pattern if terms from the subgraph can be substituted.
for the variables in the triple pattern resulting in an RDF graph equivalent to the
subgraph. SPARQL allows expressing conjunctions and disjunctions of patterns and
optional patterns.

2.3 Datalog and the Semantic Web

The semantic web has standards for representing both ontologies (OWL), and rules
(RuleML). OWL can be divided into three increasingly expressive and computationally
demanding subsets: OWL Lite, OWL DL, and OWL Full [MvH04]. OWL DL gets
its name from its correspondence with Description Logics (DL), and offers maximum
expressiveness while still retaining computational completeness (decidability), something
that OWL Full does not guarantee. OWL provides constructs for expressing individuals,
classes, (binary) properties, class and property inclusion axioms, such as subClassOf
for expressing class hierarchies, and constructing classes, such as intersectionOf to
construct the intersection of two classes.

RuleML is based on Datalog restricted to unary and binary predicates [Rul10]. RuleML
allows for the expression of rules in XML. Datalog, the basis for RuleML, and description
logics, the basis for OWL, have different expressive powers. For example, Datalog is
unable to express cardinality restrictions, which description logics can do.

Still, some statements of description logics about properties and classes can be mapped to
Datalog and vice versa. Take for example the expression of transitivity. The description
logic expression $P \sqsubseteq^+ P$ states that $P$ is a transitive property, which in Datalog can be
written as $P(?x, ?z) \leftarrow P(?x, ?y), P(?y, ?z)$.

The discussion above is based on that in [GHVD03], where the authors also introduce
Description Logic Programs (DLP), which are contained in the expressive intersection
of description logics and Datalog. On the other hand, the proposed Semantic Web Rule
Language (SWRL) standard is an effort to create a language combining both OWL DL
and RuleML [HPSB+04].

For us, DLP is interesting as it shows that Datalog can also be exploited to express and
query ontologies, albeit in a more restricted manner compared to OWL.
2.4 Recursive Query Evaluation

Given a Datalog program, we are interested in answering queries over this program. This section presents how facts can be derived from Datalog programs and how queries can be answered.

2.4.1 The Proof Theory of Datalog

In this section we present the proof theory of Datalog, which shows how new facts can be produced from a Datalog program.

Definition 2.18 (Substitution). A substitution \( \theta \) is a finite set of variables and terms denoted by \( \{?x_1/t_1, \ldots, ?x_n/t_n\} \), where each \(?x_i\) is a distinct variable and each \(t_i\) is a term.

A substitution all of whose terms \(t_1, \ldots, t_n\) are constants is called a **ground substitution**.

Definition 2.19 (Substitution Application). Let \( \theta = \{?x_1/t_1, \ldots, ?x_n/t_n\} \) be a substitution and \( L \) be a literal, then the application of \( \theta \) to \( L \), denoted \( L\theta \), is the literal obtained from \( L \) by simultaneously replacing each occurrence of \(?x_i\) in \( L \) by the corresponding \( t_i \). \( L\theta \) is called an instance of \( L \).

A substitution is applied to a clause by applying it to each literal in the clause.

Algorithm 2.1 gives a for generating new facts from a Datalog program in one step. MGU denotes the function which finds the most general unifier of two clauses. A substitution \( \theta \) is called the most general unifier (mgu) of a set of expressions \( S \) if it unifies the members of \( S \), and for any unifier \( \omega \) of \( S \), there is a unifier \( \lambda \), such that \( \omega = \theta\lambda \). The term \( \theta\lambda \) denotes the composition of the two substitutions, \( \theta \) and \( \lambda \), which is the substitution that produces the same result as applying \( \theta \) then \( \lambda \) to any expression. This method serves as
the basis for the concept of inference given in the following definition.

**Algorithm 2.1: STEP_INFER(D)**

**Input:** A Datalog program $D$

**Output:** Set of all facts which can be inferred from $D$ in one step

1. \textbf{begin}
   2. \text{result} := \emptyset;
   3. \textbf{foreach} rule $R: A \leftarrow B_1, ..., B_n$ \textbf{do}
       4. \textbf{foreach} $n$-tuple $<F_1, ..., F_n>$ of ground facts \textbf{do}
           5. $\theta := \text{MGU}((B_1, ..., B_n), (F_1, ..., F_n));$
           6. \textbf{if} $\theta \neq \text{FAIL}$ \textbf{then}
               7. \text{result} := \text{result} $\cup \{A\theta\}$
   8. \textbf{return} result
\textbf{end}

**Definition 2.20 (Inference).** Given a Datalog program $D$ and a ground fact $F$, $F$ can be \textit{inferred} from $D$, denoted $D \vdash F$, if $F \in \text{INFER}(D)$ as given in Algorithm 2.2.

**Algorithm 2.2: INFER(D)**

**Input:** A Datalog program $D$

**Output:** Set of facts that can be inferred from $D$

1. \textbf{begin}
   2. old := $\emptyset$;
   3. new := $D$;
   4. \textbf{while} new $\neq$ old \textbf{do}
       5. old := new;
       6. new := new $\cup$ STEP_INFER(new);
   7. result = all facts in new;
   8. \textbf{return} result
\textbf{end}

\text{INFER} applies \text{STEP_INFER} repeatedly to come with new inferences until no new inferences are possible. If we focus on one fact, then we can view the repeated applications of \text{STEP_INFER} until that fact is produced as a proof that the fact is a consequence of the given Datalog program. In this context, we can define a proof tree for a fact:

**Definition 2.21 (Proof Tree).** A \textbf{proof tree} of a fact $F$ from a Datalog program $D$ is labeled tree where:
• each vertex of the tree is labeled by a fact,
• each leaf is labeled by a fact in $D$,
• the root is labeled by $F$, and
• for each internal vertex, there exists a grounding of a rule $F_0 \leftarrow F_1, ..., F_n$ in $D$ such that the vertex is labeled $F_0$ and its children are labeled $F_1, ..., F_n$ respectively.

By presenting the INFER algorithm, we have shown how to find all facts that follow from a Datalog program and how to find whether a fact follows from a Datalog program $D$ by checking if it is in INFER($D$). Having presented the model-theoretic semantics of Datalog in 2.1.2, we state two theorems which show the equivalence between the two semantics. The proof for those theorems can be found in [CGT90].

**Theorem 2.22 (Soundness of Inference).** If $D$ is a Datalog program and $F$ is a ground fact, then $D \vdash F \Rightarrow F \models D$.

**Theorem 2.23 (Completeness of Inference).** If $D$ is a Datalog program and $F$ is a ground fact, then $F \models D \Rightarrow D \vdash F$.

By definition, a Datalog program is a finite structure containing finitely many constants, including predicates$^2$. This means, as Theorem 2.24 states, that finitely many facts can be inferred from it.

**Theorem 2.24 (Finiteness of Inference Over A Datalog Program).** If $D$ is a Datalog program then INFER halts on $D$.

Theorem 2.24 also implies that Datalog is decidable. There is always a fixpoint which INFER will reach, and after which no new inferences can be made.

### 2.4.2 Bottom-up Query Evaluation

In Section 2.4.1 we showed how to establish whether a fact follows from a Datalog program using inference. We now want to move to asking queries over a Datalog program. We define an answer to a query as follows:

**Definition 2.25 (Query Answer).** An answer to the query $? \leftarrow Q$ in the context of a Datalog program $D$ is a substitution $\theta$ such that $D \models Q\theta$ is true

---

$^2$This is related to the domain closure assumption (DCA), which says that there are no other constants other than those in the database.
Answers to queries can be generated in two ways: bottom-up and top-down. This section presents bottom-up query evaluation while the next presents top-down query evaluation.

Answering a query in a bottom-up fashion involves:

1. starting from extensional facts and using the rules to generate all facts which can be inferred by a Datalog program and
2. performing a selection on the facts to return answers to the query.

The INFER algorithm described in Section 2.4.1 gave us a naive method of performing the first step. When this method is used, the bottom-up query evaluation is said to be **naive**.

We can improve on this method by making a simple observation. Every iteration of the INFER algorithm invokes STEP_INFER. To infer a new fact in iteration $i + 1$ of INFER, at least one fact generated in iteration $i$ must be used. This observation leads us to **semi-naive** bottom-up query evaluation.

Semi-naive bottom-up query evaluation can be achieved by labeling facts with the iteration in which they were generated starting from 1. Extensional facts are given the label 0. Now, in iteration $i$ of INFER, the second `foreach` loop of STEP_INFER (Line 4) has to use at least one fact generated in iteration $i - 1$.

Although semi-naive evaluation improves on naive evaluation, there is still a great deal of redundancy when it comes to query answering. Generating all facts that can be inferred by a program means that the structure of the query is being ignored. We would expect that, in a deductive database, most facts are irrelevant to answering a specific query.

### 2.4.3 Top-down Query Evaluation

After introducing bottom-up query evaluation in the previous section, we introduce top-down query evaluation here. In top-down query evaluation, the starting point is the query and (ideally) only facts relevant to answering the query are generated.

We start our discussion of SLD-resolution, which is the basis of top-down approaches. SLD-resolution processes queries in a manner similar to how a human thinks about answering queries. However, it suffers one major issue: it is not guaranteed to terminate on an arbitrary Datalog program and query because it can run into cycles. This issue is solved using tabling, which we briefly discuss here. Section 5.6.2 presents QSQR, a concrete top-down query evaluation strategy which uses tabling to guarantee termination.
SLD stands for Linear resolution with Selection function for Definite clauses. It is one of several resolution techniques and is the most widely used form in logic programming systems, particularly Prolog\(^3\).

SLD-resolution answers a query by constructing an SLD-refutation, where facts are derived from rules until answers to the query are reached. The definitions that follow formalize the concepts of derivation and refutation for SLD-resolution.

**Definition 2.26 (Derived Goal).** Let \( q = \varphi \leftarrow A_1, ..., A_i, ..., A_n \) be a goal and \( \varphi = A \leftarrow B_1, ..., B_n \) where \( n \geq 0 \). If \( \theta \) is the most general unifier of \( A_i \) and \( A \), then \( q' = ( \leftarrow A_1, ..., A_{i-1}, B_1, ..., B_n, A_{i+1}, ..., A_k ) \theta \) is said to be a derived goal from \( q \).

The manner in which \( i \) is chosen in Definition 2.26 is determined by a selection function. The selection function \( i := 1 \) is called the leftmost selection function.

**Definition 2.27 (SLD-derivation).** An SLD-derivation from a goal \( q \) with a program \( D \) is a sequence \( q_0 = q, q_1, ... \) of goals and \( \theta_1, \theta_2, ... \) of substitutions such that for each \( i \), \( q_{i+1} \) is derived from \( q_i \) using \( \theta_{i+1} \).

**Definition 2.28 (SLD-refutation).** An SLD-refutation of a goal \( q \) and a program \( D \) is a finite SLD-derivation from \( q \) with \( D \) which has the empty clause as the last goal in the derivation.

Intuitively, Definition 2.28 states that an SLD-refutation of a goal in the context of a program \( D \) is a derivation that leads to facts, which can be seen as rules with empty bodies as shown in Definition 2.6.

An answer to a query \( q \) can be obtained from an SLD-refutation by restricting the composition \( \theta_1 \theta_2 ... \theta_n \) of the substitutions \( \theta_1, \theta_2, ..., \theta_n \) in the SLD-refutation to the variables in \( q \).

Figure 2.2 gives an example of answering a query using SLD-resolution. Figure 2.2(a) gives the setting for an example, with one rule and two facts. Figure 2.2(b) shows the query asking for where Al Gore lives. Figure 2.2 (c), (d), and (e) show the steps of the SLD-refutation that results in the answer substitution in Figure 2.2(e).

An SLD-derivation can be finite or infinite. An SLD-refutation is a successful SLD-derivation in the sense that it shows that the goal follows from a program. An SLD-derivation fails if it ends with a non-empty goal clause with a selected atom that does not unify with any of the clauses of the program.

\(^3\)The SLD-resolution evaluation strategy itself is sometimes called Prolog, as in [BR86].
r1 : \text{livesIn}(\?x, \?y) \leftarrow \text{isMarriedTo}(\?x, \?z), \text{livesIn}(\?z, \?y)

f1 : \text{livesIn}(\text{Tipper\_Gore}, \text{Washington\_D.C.}) \leftarrow

f2 : \text{isMarriedTo}(\text{Al\_Gore}, \text{Tipper\_Gore}) \leftarrow

(a) Setting

q = ? \leftarrow \text{livesIn}(\text{Al\_Gore}, \?loc)

(b) Query

q = ? \leftarrow \text{isMarriedTo}(\text{Al\_Gore}, \?z), \text{livesIn}(\?z, \?loc)

\theta = \{?x/\text{Al\_Gore}\}

(c) First SLD-refutation with r1

q = ? \leftarrow \text{livesIn}(\text{Tipper\_Gore}, \?loc)

\theta = \{?x/\text{Al\_Gore}, ?z/\text{Tipper\_Gore}\}

(d) Second SLD-refutation with f2

q = ? \leftarrow \text{empty clause}

\theta = \{?x/\text{Al\_Gore}, ?z/\text{Tipper\_Gore}, \?loc/\text{Washington\_D.C.}\}

(e) Final SLD-refutation with f2

\theta = \{?loc/\text{Washington\_D.C.}\}

(f) Query Answer

Figure 2.2: SLD-resolution example

SLD resolution is both sound and complete (see Theorems 2.22 and 2.23). However, it suffers from the problem that it is not database-complete. A procedure for answering a query is \textbf{database-complete} (\textbf{DB-complete}) if, whenever there are finitely many answers to a query it terminates after returning all the answers [Vie87]. Infinite SLD-derivations are possible either due to cycles in the relations or cycles in the rules resulting from recursive definitions. An example of the latter case is given in Figure 2.3, where the evaluation of \( q \) using the rule \( r_2 \) will result in generating a new subquery \( \text{ancestor}(\text{Edward\_IV\_of\_England}, \?w) \) that is the same as the original query (up to variable renaming), resulting in a cycle. It is also possible that using one selection function will result in cycles while another would not.

\[ D \]
\[ r_1 : \text{ancestor}(\?x, \?y) \leftarrow \text{parent}(\?x, \?y) \]
\[ r_2 : \text{ancestor}(\?x, \?y) \leftarrow \text{ancestor}(\?x, \?z), \text{parent}(\?z, \?y) \]
\[ \ldots \]
\[ q \leftarrow \text{ancestor}(\text{Edward\_IV\_of\_England}, \?x) \]

Figure 2.3: Example of an infinite SLD derivation

DB-completeness can be achieved by using tabling (also known as memoing), in which two tables are maintained. One table is used to keep track of subqueries currently being evaluated so that cycles can be avoided. The other table stores the intermediate
Two different types of complexity can be distinguished when discussing the complexity of query evaluation: data complexity and expression complexity [Var82, DEGV01, GP03].

The distinction is based on looking at the deductive database and query as two distinct parts: (i) an extensional database which accounts for all constants which can show up in all query results, and (ii) a set of rules which define new relations between those constants and the query, collectively called the query expression.

Data complexity is the complexity of evaluating a query as a function of the size of the extensional database. It shows how a query will perform on an arbitrary database instance with a fixed query expression.

Expression complexity (also known as program complexity) is the complexity of evaluating a query as a function of the size of the query expression. It shows how a query performs on an arbitrary query expression and a fixed database instance.

Traditionally in deductive databases, data complexity was considered to dominate expression complexity [AHV95] as the sizes of extensional database instances tend to dominate rule sizes. This is highly dependent on the nature of the rules in the deductive database.

Table 2.4.4 shows the complexity of query answering in two settings (query languages). The first row is for queries over general Datalog programs. The second row shows the complexity of answering queries in relational algebra.

As one goes down in the table, the complexity of query answering decreases while the expressive power of the language increases. For example, one well known result from [AU79]
Table 2.1: Complexity of queries

<table>
<thead>
<tr>
<th></th>
<th>Data complexity</th>
<th>Expression complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Datalog</td>
<td>P-complete</td>
<td>EXPTIME-complete</td>
</tr>
<tr>
<td>Relational algebra</td>
<td>LOGSPACE-complete</td>
<td>PSPACE-complete</td>
</tr>
</tbody>
</table>

is that relational algebra is unable to express transitive closure queries over relations, while this is possible in Datalog.

2.5 Join Ordering in Deductive Databases

The issue of join ordering (or subquery scheduling) is an important and difficult one for a recursive query setting. In this section we present the problem and some proposed approaches. Subsection 6.1 presents our approach to subquery scheduling.

We start this section with a simple example shown in Figure 2.4 to demonstrate the importance of the issue of subquery scheduling. The example shows two rules in a Datalog program which define the ancestor relation, and a query over the same relation. Assume that the Datalog program contains many extensional facts, which we do not show here. The query in the example is asking for all the ancestors of Charles, Prince of Wales. If the evaluation of \( r_2 \) is done from left to right as the rule is expressed, then the query \(? ← ancestor(?x, ?y)\) will be issued, asking for the ancestry of everyone in the knowledge base, which is clearly a large query that will return a large number of results that will not contribute to the final result.

This problem is caused by the leftmost atom (or left-to-right) selection function which we used. With this selection function, the problem could have been avoided if we supplied \( r_2 \) with a different order. It is undesirable that the user has to care about the order in which rules are written. In a declarative language, such as Datalog, the manner in which queries and rules are written should only dictate the logic behind the computation and not the computation itself. One order that might work well for one query, with a certain binding pattern, can easily fail for another query with a different binding pattern. During recursive query processing, it is also possible that the same rule will be triggered multiple times with different binding patterns. Logic programming systems such as Prolog use the leftmost selection function, as do some deductive database systems, such as CORAL \[RSSS94\].

Another possible selection function is the most-bound-atom-first (MBF), where the atom with the largest number of constants is evaluated first, in the hope of returning the smallest intermediate relation. This selection function was used in the Stanford’s NAIL!
deductive database system [SAdM08]. This selection function would have worked well for the case given in Figure 2.4. After unification, the second atom generates the query $\texttt{?} \leftarrow \texttt{parent(?z, Charles, _Prince_of_Wales)}$, which is constrained by the original query. After answering this subquery, bindings would be obtained for $?z$, which would then be used to answer the first subquery generated from rule $r_2$.

The most-bound-atom-first attempts to resemble query optimization in relational databases. In a relational database, one of the most important and successful heuristics for join ordering is to look for the smallest intermediate relations. In general, join ordering in a relational setting relies heavily on statistics. One of the most important statistics is relation size [GMUW00].

If we try to map such techniques to a recursive setting, we are confronted with the following problem: unlike extensional relations, the sizes of intensionally defined relations are not known until they are evaluated. One approach to deal with this problem is to estimate the size of an intensional relation by means of sampling, where a set of queries over the relation are issued, and the size of the relation is estimated using the results of such queries.

Adaptive sampling is a fairly novel strategy to estimate the size of a (intensional) relation [LN90, LNS90]. This technique is said to be adaptive because it adapts the number of samples taken based on the cost of the samples, rather than starting with a statistically predetermined number of samples. The justification for this in our setting is that the time to compute different samples can vary a lot. This means that starting with a predetermined number of samples to reach a certain accuracy can have unpredictable running time.

In adaptive sampling, the cost of a sample is a function of its size. Assume we want to estimate the size of relation $R$. This relation has to be partitionable.

**Definition 2.29** (Partitionable Relation). A relation $R$ over a database $D$ is $n$-partitionable if:

- $R$ can be partitioned into $n$ disjoint subsets $R_i$, for $1 \leq i \leq n$;
• There exists a function $b(D,R)$ such that for $1 \leq i \leq n$, such that $0 \leq |R_i| \leq b(D,R)$;

• It is possible to randomly select a subset $|R_i|$ and compute its size.

If $R$ is an intensional binary relation (as in RDF), then it is $n$-partitionable as follows: take argument 1 (the subject) of $R$ and partition it based on the constants appearing in the subject, which can be determined from the rules defining $R$. The size of each such partition of $R$ is at most $|Const|$, corresponding to the number of entries the single remaining argument (the object), can take.

Algorithm 2.3 gives the adaptive sampling algorithm. Note the guarantees given by the algorithm, which are derived in [LN90]. The algorithm essentially performs a number of selections on the relation $R$.

**Algorithm 2.3**: ADAPTIVE_SAMPLING($R, D$)

**Input**: An $n$-partitionable relation, $R$

**Input**: A Datalog program, $D$

**Input**: $d, d > 0$

**Input**: $p, 0 \leq p < 1$

**Output**: Estimate of $|R|$ to within $|R|/d$ with probability $p$

```
begin
  \[ \alpha = \frac{d(d + 1)}{(1 - \sqrt{p})} \]
  \[ S = 0 \]
  \[ m = 0 \]
  repeat
    randomly choose some $i \in \{1, ..., n\}$
    compute $s = |R_i|$ \\
    $S = S + \max(1, s)$ \\
    $m = m + 1$
  until $S \geq \alpha b(D, Q)$ ;
  return $nS/m$
end
```

Another approach to dealing with the sizes of intensional relations is that given in [SAdM08], where Datalog is used for program analysis. There, domain knowledge is assumed. For instance, the authors use the example in Figure 2.5, which gives rules defining the transitive closure of the $\text{hasSubtype}$ relation.

\[
\begin{align*}
\text{hasSubclassPlus}(&\ ?\text{super}, \ ?\text{sub}) \leftarrow \text{hasSubtype}(\ ?\text{super}, \ ?\text{sub}) \\
\text{hasSubclassPlus}(&\ ?\text{sup}, \ ?\text{sub}) \leftarrow \text{hasSubclassPlus}(\ ?\text{sup}, \ ?m), \text{hasSubtype}(\ ?m, \ ?\text{sub})
\end{align*}
\]

**Figure 2.5**: Sample rules for domain knowledge approach

---

\(^1\)Tighter bounds are also possible by typing predicate arguments or looking at the structures of the rules defining a predicate.

\(^4\)Hence the exponent 1
Based on their domain knowledge, they know that, for example, a supertype has, on average, 5.9 immediate subtypes. Additionally, they know that there are no chains of length more than 3. Such statistics allow for the computation of the size of the \textit{hasSubclassPlus} relation.

Derr [Der93] presents how different (legacy) deductive database systems deal with the issue of join ordering.

### 2.6 Built-in Predicates

Consider the following rule:

\[
\text{hasSibling}(\textit{?x}, \textit{?y}) \leftarrow \text{hasParent}(\textit{?x}, \textit{?z}), \text{hasParent}(\textit{?y}, \textit{?z}), \text{NOT_EQUAL}(\textit{?x}, \textit{?y}).
\]

and the query \(q = \textit{?} \leftarrow \text{hasSibling}(\textit{a}, \textit{?y})\). The rule states that two people are siblings if they have a common parent and are not the same person (siblinghood is not reflexive). The atom \text{NOT_EQUAL}(\textit{?x}, \textit{?y}) does not correspond to a traditional extensional or intensional predicate. It is intended to be evaluated only when bindings for both \textit{?x} and \textit{?y} are available.

If we deal with the atom \text{NOT_EQUAL}(\textit{?x}, \textit{?y}) as an extensional atom, which when evaluated would return bindings for variables, then answering \(q\) under a selection function which schedules \text{NOT_EQUAL}(\textit{?x}, \textit{?y}) to be evaluated first would result in a large output.

Predicates such as \textit{EQUAL}, \textit{NOT_EQUAL}, \textit{LESS_THAN} etc. are called 
\begin{highlight}
\textbf{built-in predicates}
\end{highlight}
or \begin{highlight}
\textbf{arithmetic predicates}
\end{highlight}. Such predicates are defined to have a \begin{highlight}
\textbf{permissible binding}
\end{highlight} pattern. The above mentioned predicates all have a permissible binding pattern of bound, bound (bb): both arguments have to be bound to evaluate the predicate. A selection function strategy may not schedule an atom with such a predicate for evaluation until its permissible binding pattern is met.
Chapter 3

URDF: Efficient Reasoning in Uncertain RDF Knowledge Bases with Soft and Hard Rules

This chapter presents the URDF framework for efficient reasoning in RDF knowledge bases with soft and hard rules [TSSN10]. Work on this thesis developed from the need to improve the performance of the grounding component of URDF. Section 3.1 discusses the problem which URDF is trying to solve, namely, query answering over large knowledge bases which exhibit both inconsistency and incompleteness. Section 3.2 presents the representation model of URDF. There, we formalize the concept of a knowledge base and discuss each of its components: facts, soft rules and hard rules. In Section 3.3 we present the reasoning model of URDF based on the concept of a dependency graph.

3.1 Motivation

The process of information extraction is not a perfect one. Incompleteness and inconsistency are two problems that can be faced when performing information extraction. Incompleteness is our inability to explicitly capture all information about the domain we are dealing with. Inconsistency arises when the extracted information violates some properties of the domain in question. Formally, an inconsistency arises when the conjunction of all statements in a domain (including those describing constraints in that domain) is unsatisfiable.

Incompleteness and inconsistency are unavoidable issues in information extraction. Sources from which information is extracted usually do not contain complete information about
the domain they are dealing with. The sources themselves might contain incorrect or contradicting information, leading to inconsistent facts being extracted. Extraction tools and techniques might not be able to capture all information in the sources or might not be able to handle how the information is presented, leading to incompleteness and inconsistency.

Asking queries in such a setting might lead to empty results due to incompleteness or contradicting answers due to inconsistency. Starting with a knowledge base containing only extracted facts with confidences, URDF takes the following approach in dealing with this problem:

- to overcome incompleteness, the knowledge base resulting from information extraction is extended with weighted soft rules that describe relations between facts in the corresponding domain. Such rules allow for new facts to be generated from existing ones exactly like deductive databases.

- to detect inconsistency, hard rules are added to the knowledge base describing any integrity constraints which must hold in the domain in question, so that inconsistencies can be detected. Hard rules provide us with negated statements, which, when violated result in an unsatisfiable setting, i.e., the conjunction of all statements in the setting evaluates to false.

When querying a URDF knowledge base, multiple weighted answers will be returned, which, taken together, can violate integrity constraints or contain contradictions. Therefore, as a preprocessing step, URDF adds an approximate weighted MAX-SAT solver as a post processing step to grounding. This special MAX-SAT solver is capable of dealing with integrity constraints by ensuring that the answers it produces do not violate any integrity constraints.

Figure 3.1 depicts the different components of URDF. In this chapter, we will not consider the special weighted MAX-SAT solver, but instead focus on the grounding step.
3.2 Representation Model

A knowledge base in URDF is defined as a triple $\mathcal{KB} = (\mathcal{F}, \mathcal{S}, \mathcal{C})$ consisting of a set of extensionally defined facts, $\mathcal{F}$, a set of soft rules $\mathcal{S}$, and a set of hard rules (or integrity constraints) $\mathcal{C}$.

3.2.1 Extensional Database

The set $\mathcal{F}$ of facts in a URDF knowledge base contains RDF facts. Each fact has a confidence value, which reflects how reliable that fact is. A higher weight expresses a more reliable fact than one with a smaller weight. The assignment of weights to facts is beyond the scope of this thesis, but in the context of information extraction it generally depends on such factors as the source from which a fact was extracted, the format in which a source presents facts, and the techniques used for extraction.

3.2.2 Soft Rules

Soft rules are used to generate new facts. They correspond to Datalog rules with weights. An example of a soft rule is:

$$\text{livesIn}(?x, ?y) \leftarrow \text{isMarriedTo}(?x, ?z) \land \text{livesIn}(?z, ?y)[0.4]$$

This rule states that a person lives in the same lane as his or her spouse. A soft rule has a non-negative real valued weight. A higher weight indicates that a fact reached using this rule has higher confidence than one reached through a rule with a smaller weight.

The weight $w$ of a soft rule $S = A \leftarrow B_1 \land \ldots \land B_n$ is estimated using the extensional database as follows:

$$w(S) = \frac{\text{numberOfGroundings}(A \land B_1 \land \ldots \land B_n)}{\text{numberOfGroundings}(B_1 \land \ldots \land B_n)}$$

Where $\text{numberOfGroundings}(E)$ corresponds to the number of distinct extensional groundings found for $E$, i.e., a conjunction of atoms.

One difference between URDF and Datalog (as expressed in Chapter 2) is that in URDF, predicates which occur in the extensional database can also have rules which define them intentionally, while in our presentation of Datalog we stated that the set of intensional predicates and the set of extensional predicates are non-intersecting.
In the context of query processing in deductive databases, this distinction is not a fundamental one. In the rest of this thesis the following definitions will hold:

- Extensional predicate: a predicate defined only using facts in the extensional database.
- Intensional predicate: a predicate defined by at least one (soft) rule. An intensional predicate can have extensional occurrences
- Purely intensional predicate: an intensional predicate with no extensionally defined instances.

A simple renaming technique could have been used to preserve the original definitions, as shown in Figure 3.2, but we think that it would be much simpler to keep the names of predicates for readability.

\[
S = \{ \text{bornIn}(\text{Tipper Gore,Washington D.C.}), \text{bornIn}(\text{Albert Einstein,Ulm}) \}
\]

(a) Without Renaming

\[
\begin{align*}
r_1 : \text{bornIn}(x,y) & \leftarrow \text{hasParent}(x,z), \text{bornIn}(z,y) \\
r_2 : \text{bornIn}(x,y) & \leftarrow \text{isMarriedTo}(x,z), \text{bornIn}(z,y) \\
f_1 : \text{bornIn}(\text{Tipper Gore,Washington D.C.}) \\
f_2 : \text{bornIn}(\text{Albert Einstein,Ulm})
\end{align*}
\]

(b) With Renaming

\[
\begin{align*}
r_1 : \text{bornIn}(x,y) & \leftarrow \text{hasParent}(x,z), \text{bornIn}(z,y) \\
r_2 : \text{bornIn}(x,y) & \leftarrow \text{isMarriedTo}(x,z), \text{bornIn}(z,y) \\
r_2 : \text{bornIn}(x,y) & \leftarrow \text{bornIn}_{DB}(x,y) \\
f_1 : \text{bornIn}_{DB}(\text{Tipper Gore,Washington D.C.}) \\
f_2 : \text{bornIn}_{DB}(\text{Albert Einstein,Ulm})
\end{align*}
\]

Figure 3.2: Predicate renaming

### 3.2.3 Hard Rules

Hard rules are used to enforce mutual exclusion integrity constraints.

**Definition 3.1** (Ground Hard Rule). A **ground hard rule** (competitor set) is a set of facts.

**Definition 3.2** (Truth of a Ground Hard Rule). A ground hard rule is true in an interpretation if at most one fact is true in the hard rule.

An example of a ground hard rule is

\[
\{ \text{bornIn}(\text{Al Gore,USA}), \text{bornIn}(\text{Al Gore,Italy}), \text{bornIn}(\text{Al Gore,Spain}) \}.
\]
In the context of URDF’s reasoning model, a ground hard rule needs to be generated only when one of the facts contained in it is generated. One important class of integrity constraints which URDF focuses on is that of expressing functional constraints on predicates. For that, we define an ungrounded hard rule from which we can generate a grounded hard rule. Before doing so, we make a restriction on the definition of the set \( \text{Var} \) defined in Section 2.1.1 and define a new set \( \text{Double-Var} \):

- \( \text{Var} \) is an infinite alphabet consisting of all finite strings starting with a single ‘?’.
  A member of \( \text{Var} \) is called a variable.

- \( \text{Double-Var} \) is an infinite alphabet consisting of all finite strings starting with ‘??’. A member of \( \text{Double-Var} \) is called a double variable.

Now, an ungrounded hard rule can be defined as follows:

**Definition 3.3 (Unground Hard Rule).** A unground hard rule (or simply a hard rule) is a binary predicate \( p \) with exactly one argument from \( \text{Var} \cup \text{Const} \) and exactly one argument from \( \text{Double-Var} \).

To express that a person can be born in exactly one place, we can use the following ungrounded hard rule

\[
\text{bornIn}(?x,??y).
\]

This hard rule states that each \( x \) can be born in a single \( y \). During query processing, once a fact \( f \) is derived, it is unified with a hard rule \( h \) to generate an atom \( A \) which will trigger the query \( q = ? \leftarrow A \) as follows:

- If \( f \) and \( h \) do not have the same predicate, then unification fails.

- If \( h \) has a constant for argument \( i, i \in \{1, 2\} \):
  - If \( f \) does not have the same constant for argument \( i \), then unification fails.
  - If \( f \) has the same constant for argument \( i \), then \( A \) is generated by replacing the double variable in \( h \) with a variable.

- If \( h \) has a variable for argument \( i, i \in \{1, 2\} \) the \( A \) is generated by replacing that variable with the constant in the corresponding position in \( f \) and the double variable with a variable.

For example, if the fact \( f = \text{bornIn}(\text{Al Gore, USA}) \) is derived, then unification with the hard rule \( f = \text{bornIn}(?x,??y) \) results in the query \( q = ? \leftarrow \text{bornIn}(\text{Al Gore, ?z}) \). This
query can then be used to generate all ground facts in the same competitor set as \( f \). For this, this query is issued as if it were a user-submitted query, and the resulting facts end up as member of a competitor set.

The class of integrity constraints discussed above, namely functional constraints, is a special class. In the general case, ground hard rules express the mutual exclusion between a set of facts.

### 3.3 Reasoning Model

The role of the recursive query processor in URDF is to produce all possible answers to a query, the soft rules which contributed to the generation of the answers, and the competitor sets which contain at least one fact generated during query processing. The result of grounding is a dependency graph. Before formally defining the concept of a dependency graph, we will motivate it with an example.

\[
\mathcal{F} = \\
f_1 : \text{hasParent}(\text{Al Gore, Albert Gore, Sr.})[0.9] \\
f_2 : \text{isMarriedTo}(\text{Al Gore, Tipper Gore})[0.4] \\
f_3 : \text{bornIn}(\text{Albert Gore, Sr., Granville})[0.9] \\
f_4 : \text{bornIn}(\text{Tipper Gore, Washington D.C.})[0.9] \\
f_5 : \text{bornIn}(\text{Albert Einstein, Ulm})[0.8] \\
\ldots
\]

\[
\mathcal{C} = \\
C_1 : \text{bornIn}(?x, ??y)
\]

\[
\mathcal{S} = \\
S_1 : \text{bornIn}(?x, ?y) \leftarrow \text{hasParent}(?x, ?z), \text{bornIn}(?z, ?y)[0.3] \\
S_2 : \text{bornIn}(?x, ?y) \leftarrow \text{isMarriedTo}(?x, ?z), \text{bornIn}(?z, ?y)[0.7]
\]

**Figure 3.3: A knowledge base**

If the query \( q = \text{bornIn}(\text{Al Gore, } ?y) \) is issued on the knowledge base in Fig. 3.3, then two answers will be obtained: \( \text{bornIn}(\text{Al Gore, Granville}) \) (using \( f_1, f_3, \) and \( S_1 \)) and \( \text{bornIn}(\text{Al Gore, Washington D.C.}) \) (using \( f_2, f_1, \) and \( S_2 \)). No answers to \( q \) are available directly from the extensional database \( \mathcal{F} \). Finally, \( C_1 \) will generate the competitor set \( \{ \text{bornIn}(\text{Al Gore, Granville}), \text{bornIn}(\text{Al Gore, Washington D.C.}) \} \) stating that any truth assignment will have to take into consideration that at most one of the two possible birth places can be assigned true. All the grounded clauses relevant to the query can now be combined into a CNF-like formula with special annotations as shown in Fig. 3.3.
Chapter 3. URDF

\[
\neg \text{bornIn}(\text{Al Gore}, \text{Granville}) \\
\neg \text{bornIn}(\text{Al Gore}, \text{Washington D.C.}) \Box \\
\land \\
\neg \text{hasParent}(\text{Al Gore}, \text{Albert Gore, Sr.}) \\
\neg \text{bornIn}(\text{Albert Gore, Sr.}, \text{Granville}) \\
\text{bornIn}(\text{Al Gore}, \text{Granville})][0.3] \\
\land \\
\neg \text{isMarriedTo}(\text{Al Gore}, \text{Tipper Gore}) \\
\neg \text{bornIn}(\text{Tipper Gore}, \text{Washington D.C.}) \\
\text{bornIn}(\text{Al Gore}, \text{Washington D.C.})[0.7]
\]

Figure 3.4: The dependency graph of \( q = \neg \text{bornIn}(\text{Al Gore}, y) \) with the knowledge base from Figure 3.3

The first clause, marked with \( \Box \), corresponds to a competitor set generated from a hard rule. The remaining two clauses correspond to rule groundings. Note that fact \( f_5 \) is not part of the CNF-like formula of Fig. 3.3, as it is not relevant to answering the query. Facts in this formula are the dependency graph for \( q \) when issued over the knowledge base in Fig.3.3. We are now ready to formally define a dependency graph.

**Definition 3.4 (Dependency Graph).** Given a KB = \( \langle F, S, C \rangle \) and a conjunctive query \( Q \), the dependency graph is a triple \( D = \langle F_G, S_G, H_G \rangle \) where:

- \( F_G \) is the set of all base facts from \( F \) which are relevant for finding an answer to \( Q \).
- \( S_G \) is the set of all ground soft rules relevant for finding an answer to \( Q \).
- \( H_G \) is the set of all competitor sets relevant for finding an answer to \( Q \).

We say that a fact (respectively, a ground soft rule) is relevant to finding an answer to a conjunctive query if the proof tree of an answer to the query contains that fact (respectively, a subtree corresponding to that ground soft rule). A ground hard rule is relevant to finding an answer to a conjunctive query if at least one fact in the ground hard rule is in the proof tree of an answer to the query.

A top-down recursive query evaluation algorithm, such as SLD-resolution for example, can be modified to return the dependency graph corresponding to answers to a query over a knowledge base. It is not difficult to see how this can be done for \( F_G \) and \( S_G \). For, \( H_G \), this can be done by unifying derived facts with hard rules and then recursively answering the generated queries. In this case, \( H_G \) would contain the set of answer sets to all such queries. Note that a ground hard rules express mutual exclusion between its members without the need to flatten it to pairs of mutually exclusive facts, which would be quadratic in the size of the competitor set.

The final answer to a query is generated by the weighted MAX-SAT solver, which takes as input the dependency graph generated by the reasoner. URDF’s weighted MAX-SAT
solver will find an assignment of truth values to ground clauses such that the sum of the weights of the satisfied clauses is maximized, without violating any of the integrity constraints expressed by the hard rules, in the form of competitor sets.
Chapter 4

RDF-3X: a RISC-style Engine for RDF

In Section 2.2 we introduced the RDF data model. One of the distinguishing characteristics of RDF data is that predicates, which are analogous to relations and attributes in the relational model, are very diverse. Unlike traditional relational data, which are stored in tables with a fixed pre-defined schema, RDF data is not meant to go through a schema-design phase. An RDF graph can be extended by adding new triples to it, possibly introducing new predicates in the process. This property makes RDF very flexible when it comes to updates and changes in the data.

This flexibility comes at a price; queries over RDF data tend to include a large number of joins, a big part of which are used to construct the ‘original relations’ from triples, resulting in star-shaped joins, as shown in Fig. 4.1. This chapter introduces one of the currently most performant RDF engines, RDF-3X, which focuses on managing large-scale RDF data with a RISC-style architecture [NW08]. We use RDF-3X as the storage backend for our recursive query processor, RDRD.

Section 4.1 presents the index-based storage infrastructure of RDF-3X. Section 4.2 briefly presents the basics of query processing in RDF-3X, while Section 4.3 presents how RDF-3X optimizes queries. Section 4.4 presents, in more detail, optimizations added to RDF-3X to allow faster execution of queries over a billion triples. We do not give a complete description of RDF-3X in this chapter, we focus on the parts relevant for integrating RDRD with RDF-3X.
Chapter 4. RDF-3X

4.1 Storage and Indexing

Conceptually, RDF-3X uses a single large triples table for storing RDF triples. The triples table is composed of three attributes: subject, predicate, and object. An alternative to this approach would be to store triples with a common predicate in a distinct table known as a *property table* [WW06]. A third approach, which is a hybrid of the first two, is to cluster predicates, usually based on the domains they deal with or their co-occurrence in workloads, and to store triples with predicates from each cluster in a distinct *cluster-property table*.

RDF-3X does not store strings directly in its triples tables. Instead, the strings are mapped to ids and a mapping dictionary is maintained to convert strings to ids and ids back to strings.

Because ids are used for storing triples, RDF-3X uses index-only relations. Six clustered B+-tree indexes containing all triples are maintained by RDF-3X, which correspond to the six possible collation orders of the subject (S), predicate (P), and object (O). This means that every query over a triple pattern can be answered using a single index scan, where the constants in the triple pattern form the prefix of the index.

Answering some queries will not require information about complete triples. Take for instance the query:

```
SELECT ?x WHERE {?x ?y Max_Planck}
```
This query is asking about all entities connected to Max Planck through some relation, but the relation itself is not relevant to answering the query. Answering such queries can be achieved by maintaining aggregated indexes which store two out of the three columns in a triple, which is what RDF-3X does. RDF-3X maintains six additional B\(^+\)-tree indexes over all possible collation orders of pairs of S, P and O. An entry in the leaf of an aggregated index is composed of the values of the two fields over which the index is constructed, and a count. The count represents the number of times a pair occurs in the full set of triples. The count is needed to respect the semantics of SPARQL with respect to duplicates [PS08].

The final set of indexes which RDF-3X maintains are fully aggregated indexes. These are B\(^+\)-tree indexes over one of the three fields: S, P, and O. These indexes store the value of the corresponding field and a count in a manner similar to that of aggregated indexes.

In all, RDF-3X uses six complete triples indexes, six aggregated indexes, and three fully aggregated indexes for a total of 15 indexes. RDF-3X uses various compression schemes for complete triples and aggregated indexes which utilize the fact that neighboring entries in the leaf pages of an index are tuples which are likely to have common prefixes, thus allowing for prefix compression. The result is that the total size of the 15 indexes is smaller than that of the original RDF data.

### 4.2 Query Processing

RDF-3X accepts SPARQL queries. The query string is translated into a set of triple patterns, each of which contains variables and constants. The constants in the triple patterns are represented using their ids in the mapping dictionary.

The triple patterns generated are used to infer a query graph where two triple patterns share an edge in the query graph if they have a variable in common. A naive execution plan can be constructed from a query graph by translating nodes into range scans, edges into joins, and merging disconnected components using cross products. The next section presents how RDF-3X performs query optimization.

RDF-3X can also handle disjunctive queries supplied using SPARQL’s \text{UNION} and \text{OPTIONAL} keywords. Triple patterns within a \text{UNION} or \text{OPTIONAL} expression are treated as nested subqueries which are optimized on their own and then treated as base input relations for further optimization. As RDF-3X targets conjunctive queries due to their prominence, no sophisticated schemes are used to optimize disjunctive queries.
Chapter 4. RDF-3X

4.3 Query Optimization

As we discussed before, joins are the critical operation in queries over RDF data. As star-shaped subqueries are common in SPARQL queries, bushy trees are desirable. It is also desirable that the query processor utilizes (sort-)merge-joins as much as possible, since RDF-3X already contains sorted indexes on all possible orders of the data.

RDF-3X uses bottom-up dynamic-programming to generate a (near) optimal plan. This allows fast plan enumeration compared to transformation-based top-down approaches. Bottom-up dynamic-programming organizes a DP table by subgraphs of the query graph, maintaining for each subgraph the optimal plan(s).

The DP table is seeded with scans corresponding to triple patterns in the query. The optimizer analyzes the query to check which variables have to be preserved and which can be projected away. Variables that are not projected by the query or are not used in a join between triples patterns can be projected away. Triple patterns with variables that are projected away are answered using aggregated indexes, while the rest are answered using the complete triple indexes.

The optimizer generates plans for every index that can answer a triple pattern (taking into consideration variables that have been projected away). While the constants in the triple pattern will form a prefix for one of the possible indexes, allowing for range scans in that index, other indexes might produce results with an order suitable for a subsequent merge-join, resulting in lower overall cost. The optimizer keeps track of the costs of the generated plans using its cost model, which rewards possibly interesting orders. Equivalent plans that are dominated by cheaper ones are pruned.

An important part of query optimization is selectivity estimates for decisions about join ordering. RDF-3X’s selectivity estimation scheme changed as the amount of data expected to be managed by RDF-3X increased. Section 4.4.1 present how RDF-3X performs selectivity estimation when managing data in the billion triples range.

4.4 Querying Large RDF Graphs

As efficient as the initial version of RDF-3X was compared to other systems [NW10], it was originally designed with datasets containing tens of millions of triples in mind. When we go into the billion triples range, the performance degrades. Two main factors contribute to this degradation:
1. Large fractions of the indexes need to be scanned by the join operators even if the result size is very small. In other words, the amount of data which join operators go through to process a query depends on the size of the data. It is desirable that the amount of data which join operators go through depends on the result size.

2. When the size of the data is too large, traditional methods of selectivity estimation, which rely on collecting statistics, become very inaccurate. The reason is that a limited amount of memory is allocated for storing such statistics (in the form of histograms, for example). This translates into suboptimal join-order optimization decisions.

The performance degradation comes in the form of longer query execution times, resulting in non-interactive response times for complex queries on large RDF graphs. The next two subsections will present RDF-3X’s solutions to both problems [NW09].

4.4.1 Sideways Information Passing

One way of optimizing joins is by having a join operator consider non-local information in an approach called *sideways information passing (SIP)*\(^1\). SIP involves having operators that are possibly not within the same sub-tree in the operator tree communicate to avoid producing outputs that will not contribute to the final result of the operator tree.

In the context of RDF-3X, index scan operators are allowed to exchange information about the data they see. This way, the index scan operators can avoid passing entries to the join operators in the pipeline which will not contribute to the final result. This implies that index scan operators will be able to skip reading parts of the index, allowing them to save very expensive disk access.

The SIP strategy used in RDF-3X, termed *ubiquitous SIP (U-SIP)*, is done in two stages: compile-time preparation and run-time handling of merge and hash joins (the two types of joins used in RDF-3X).

During query compile-time, a decision is made about which operators in the operator tree need to exchange information and should therefore have bidirectional U-SIP edges between them. For this, the concept of *equivalent variables* of a query (or a query plan) is used, where two variables \(v_1\) and \(v_2\) are equivalent \((v_1 = v_2)\) if they must have the

\(^1\)A different definition of sideways information passing will be given in Subection 5.6.2 in the context of recursive query processing. Later U-SIP will be introduced and will be used in the rest of this thesis for this context.
same bindings. An equivalence class \( E(v) \) of a variable \( v \) is computed as follows:

\[
\begin{align*}
E_0(v) & := \{v\} \\
E_n(v) & := \{v_2 \mid v_1 \in E_{n-1}(v) \land v_1 \equiv v_2\} \\
E(v) & := \bigcup_{n=0}^{\infty} E_n(v)
\end{align*}
\]

Equivalence classes tell us which variables need to have the same bindings to produce an output result.

In RDF-3X, a bidirectional U-SIP edge is added between two index scan operators \( o_1 \) and \( o_2 \) in an operator tree \( i f f \):

1. they have variables in the same equivalence class and
2. there is no pipeline-breaking operator, such as a hash join operator, in the subtree rooted at the lowest common ancestor of \( o_1 \) and \( o_2 \).

The second stage of U-SIP takes place after query compilation, while the query plan is being executed. Merge and hash joins are handled differently as we present next.

### 4.4.1.1 Run-Time Handling of Merge Joins

Merge join operators get their inputs in ascending order of the join-variable values. Each index scan operator serving as an input to a join operator has a cursor for its current binding. Index scan operators can exchange those values, using a shared memory structure. An operator will compare its current binding with the bindings received from other operators, compute the maximum (\( m \)) and change its own binding to a value equal to or greater than \( m \) (this value is called the operator’s next binding). All intermediate values can be safely skipped as they will not contribute to the result of the join.

### 4.4.1.2 Run-Time Handling of Hash Joins

The run-time handling of hash joins is very different from that of merge joins because data is not sorted in the first. We use a global domain filter for each equivalence class of variables consisting of:

1. range bounds (minimum and maximum values seen), and
2. a bloom filter for each equivalence class of variables
Bloom filters will be used for skipping values in index scans with variables in the same equivalence class as one of the variables on which a hash-table was built. For that, the used hash functions will be distance-preserving which allows us to map a run of 0’s in the bloom filter to a number of values (a distance) that can be skipped in an index scan.

**Definition 4.1** (Distance preserving hash function). A hash function \( h(x) \) for mapping values \( x, y \) onto a bit-vector \( D[0 \ldots m-1] \) is distance preserving if a run of zero bits in the bit vector between \( h(x) \) and \( h(y) \), where runs can be extended modulo \( m \), denotes that the next set bit corresponds to a value \( y \) with a specific distance to \( x \) provided that \( y > x \).

Every hash-join build operator creates its own private Bloom filter. The observed domain of each hash-join build operator is initialized to \((\text{inf}, 0, 0^*)\) as no values have been seen yet. After that, each private filter is intersected with the Bloom filter in the domain filter of its equivalence class. The potential domain of variables in an equivalence class is initialized to \((0, \text{inf}, 1^*)\), as all values are possible. After the hash table is built using variable \( v \), we can skip entries in any index scan with a variable in \( E(v) \) that has values not seen in the build phase. For that, we need to be able to decode what “next” \((\text{next}(v))\) value we expect to have been seen in the build phase from the Bloom filter. Given a scan’s current binding, \( \text{cur}(v) \), according to its cursor position, we can compute a lower bound for the gap that we can skip. The next possible binding for \( v \) is:

\[
\text{next}(v) = \begin{cases} 
\text{next}(D.min) & \text{if } \text{cur}(v) < D.min \\
\infty & \text{if } \text{cur}(v) > D.max \\
\text{cur}(v) & \text{if } D.filter \text{ contains } \text{cur}(v) \\
\text{cur}(v) + \frac{1}{a}(x1\Theta h(\text{cur}(v))) & \text{otherwise}
\end{cases}
\]

where \( x1 \) is the position of the next one-bit entry in the bit-vector modulo \( m \), \( \Theta \) is the modulo-\( m \) difference and \( h(x) = ax \mod m \).

### 4.4.2 Selectivity Estimation

Initial versions of RDF-3X relied on equi-depth histograms for selectivity estimates, which are crucial for join-order optimization. As the number of triples being managed by RDF-3X increases to the billion triples range, we have to make a compromise between the accuracy of the estimation histograms and the space dedicated to them as we want the histograms to be always read from main memory.

A different approach is used for tackling the problem of selectivity estimation for large data sets. It relies on the fact that RDF-3X already has compressed indexes on all SPO
permutations, binary, and unary projections with counts. In this approach, the join selectivity of a triple pattern is taken as the *selectivity of that triple pattern with all other triples*.

To show how this works, let us take a triple pattern \((c_1, c_2, v)\) where \(c_1\) and \(c_2\) are constants and \(v\) is a variable. Now, as an example, to estimate its selectivity when joining it with all other triples \((s_2, p_2, o_2)\) based on their subject \(s_2\), where \(s_2, p_2\) and \(o_2\) stand for variables, we compute:

\[
\text{sel}\left((c_1, c_2, v) \bowtie_{v=s_2} (s_2, p_2, o_2)\right) = \frac{|((c_1, c_2, v) \bowtie_{v=s_2} (s_2, p_2, o_2))|}{\sum_{x \in \Pi_v (c_1, c_2, v)} |(x, p_2, o_2)|}
\]

where we can use the unary projection indexes to get \(|(x, p_2, o_2)|\) (for each value of \(x\) of \(v\)) with one point query into the index.

These join selectivities are pre-computed for all possible choices of triple patterns with one or two variables and for all permutations of SPO fields. The results are stored in compressed B+-trees indexed by the constants in the triple pattern. For the example case above, the leaves of the B+-tree have entries of the following form: \((c_1, c_2, s_s, s_p, s_o)\), i.e., the constants and the size of the join result when comparing \(v\) with the subject, predicate or object of all triples in the database. The case where we have two variables is similar, but instead of three join possibilities there would be six. In the case where we have three variables in a triple pattern (all fields are variable), there are nine possible ways to join it with all other patterns, and we keep an index on those as well.
Chapter 5

Systems and Integration

For this thesis, we implemented a recursive query processor, called RDRD (Rule-based query processor for Disk-Resident Data), for the URDF framework presented in Chapter 3 on top of the RDF-3X engine presented in Chapter 4. This chapter presents the implementation of RDRD, and the next chapter presents some optimizations we added to it. RDRD is essentially a deductive database for RDF data, with the ability to maintain a dependency graph which URDF needs. Section 5.1 gives a brief view of the architecture of the system. Section 5.2 presents the rule store. Section 5.3 presents how RDF-3X was integrated as storage backend of RDRD, and what changes and additions we had to make to it. Section 5.5 describes our dynamic subquery scheduler. Section 5.6 describes the component of the system that handles recursive query processing in a top-down fashion. There, we present the QSQR algorithm for recursive query evaluation, which we use in our system, argue why its presentation in [AHV95] is incomplete and suggest a fix for this. Finally, Section 5.7 describes the different modes in which our system can operate, either as a normal deductive database or as a special one with the ability to maintain lineage and dependency graph information for URDF.

5.1 System Architecture

Figure 5.1 shows a high-level view of the architecture of RDRD. It is composed of the following components:

1. A rule store where rules are kept. This component is discussed in Section 5.2.

2. A fact store which stores extensionally defined facts. We use RDF-3X, presented in Chapter 4 as our storage backend for facts. Section 5.3 discusses how we integrated RDF-3X into RDRD.
3. The subquery scheduler, which is responsible for determining the order of evaluation of atoms in a conjunctive query. The query itself may be the one issued directly by the user or a query generated from the body of a rule during the recursive evaluation of a query submitted by the user. Section 5.5 presents this component in detail.

4. The recursive query processor, which is composed of implementations of two two top-down recursive query processing algorithms: SLD-resolution and QSQR. Section 5.6 this part of the system. Both implementations support caching of answered subqueries.

RDRD has two stages: the pre-processing stage and the query stage. During pre-processing, a user submits an RDF-3X database file containing the extensional facts and the set of rules, both known collectively as the knowledge base (see Section 3.2), to the reasoner. Figure 5.2 shows a timeline with the operations performed in each stage.
5.2 Rule Store

We assume, as is common for deductive databases, that the number of rules in our system is much smaller than that of the number of base facts. Based on this, our rules are assumed to be memory resident during query processing.

The user submits the rules as strings using a syntax similar to that we presented for Datalog. After the rules are parsed, they are analyzed and their predicates are classified as described in Subsection 3.2.2. The classification is important for two reasons:

- There is no need to query the fact store for purely intensional atoms, as it would always return empty answers.
- Extensional atoms can be grouped together (chained), but not with other kinds of atoms, to allow for more efficient access to disk. This point will be presented in more detail later in Section 6.1.

RDRD’s representation of atoms, which are the building blocks of rules, uses ids rather than strings, in a manner similar to that of RDF-3X. Extensional and intensional predicates and constants in rules use RDF-3X’s ids. Two other dictionaries are also maintained, one for variables and another for purely intensional predicates. Each atom’s predicate also has a flag to indicate the type of the predicate.

5.3 Storage Backend

RDRD uses RDF-3X as its storage backend for extensional RDF facts. This choice was made based on RDF-3X’s performance statistics in comparison to other RDF engines. RDRD is tightly integrated with RDF-3X, access to RDF-3X is done using its native C++ interfaces.

5.3.1 Pre-processing

A user is assumed to have an RDF-3X database file, which can be compiled from a plain text RDF file in Turtle syntax [DB08] by the application rdf3xload. When the user starts RDRD, he submits the location of this file, along with the rules that will be used for reasoning.

RDF-3X is first used during the translation of rules into RDRD’s special representation, as described in Section 5.2. During this translation, predicates are classified into extensional, intensional and purely intensional. Predicates in rules are first collected and
then the fully aggregated index on predicates is queried for each of those predicates. If a
predicate exists in this index, then it is classified as extensional or intensional, depending
on whether there are any rules that define it. If the predicate does not exist in the index,
then it is classified as a purely intensional predicate (if it is defined by a rule). A similar
process is done when the query is submitted to classify the predicates in its atoms.

Internally, RDRD represents constants and variables in rules and queries using integers
for efficient comparison. The rules use RDF-3X’s integer representation for extensional
and intensional predicates and constants occurring as arguments. This serves to speed
interaction with RDF-3X and to avoid any translation step in the middle. We added
interfaces on top of RDF-3X’s dictionary to translate strings to ids during pre-processing
and to translate ids back to strings when answers need to be displayed to the user.

We also have special interfaces with RDF-3X to identify domain sizes. Our subquery
scheduling scheme, which we present later in Section 6.1, requires that we know:

1. the sizes of sets defining domains, and
2. the sizes of the active domains of the arguments of extensional predicates.

For instance, assume we have the following domain definitions for two extensional pred-
icates:

\[
bornIn(\text{PERSON, CITY})
\]

\[
hasMayor(\text{CITY, PERSON})
\]

which state that the domains of the subject of the \text{bornIn} predicate and the object
of the \text{hasMayor} predicate are the set \text{PERSON}, and the domains of the object of
the \text{bornIn} predicate and the subject of the \text{hasMayor} predicate are the set \text{CITY}.
Assuming that no other extensional predicates exist which have arguments with one of
the domains above, the sizes of \text{PERSON} and \text{CITY} in our extensional database are
equal to the cardinalities of the results of the following two queries respectively:

\[
\text{SELECT DISTINCT ?domainMember WHERE } \{ {?domainMember bornIn ?unused} \text{ UNION } \{?unused CITY ?domainMember\} \}
\]

\[
\text{SELECT DISTINCT ?domainMember WHERE } \{ {?unused bornIn ?domainMember} \text{ UNION } \{?domainMember CITY ?unused\} \}
\]

To compute the number of \text{PERSONs} for whom the \text{bornIn} relation is defined, i.e., the
size of the active domain of \text{bornIn}, the following query can be issued:

\[
\text{SELECT DISTINCT ?activeDom WHERE } \{ ?activeDom bornIn ?unused\}
\]
To this end, we implemented an interface which takes definitions of predicate domains, and returns domain and active domain sizes. For this interface we are interested only in result sizes and not actual values, for this, we implemented a count aggregation operator for RDF-3X.

5.3.2 Query time

During query time, RDF-3X is accessed for answering extensional and intensional atoms in a query (potentially recursively). RDF-3X was designed with queries composed of large joins in mind. However, the query pattern which we expect from our setting is much different. We expect that small queries, often composed of a single atom, will be issued to RDF-3X.

As we expect single-atom queries to occur frequently, we made sure that they can be performed as fast as possible. When a single-atom query is issued to RDF-3X during recursive query processing, it is handled by a special method which directly issues an index scan on the appropriate full triples index. The choice of index is based on the binding pattern of the atom: bound entries of the atom should form a prefix of the chosen index. Issuing an index scan directly avoids logical query plan generation and query optimization, which are pure overhead for a single-atom query, but are needed for the general case.

This manner of handling single-atom queries allows us to issue single-atom queries composed of a ground atom, which correspond to an ASK query in SPARQL. Issuing such queries is not yet supported by RDF-3X’s query interface.

For queries containing more than a single atom, it is required that RDF-3X’s query processing infrastructure is exploited to produce both a good join ordering and a good choice of physical join operators. In Section 4.3, we explained how RDF-3X’s query optimizer generates query plans in a bottom-up manner considering every index that can answer a triple pattern. For us, this means that six indexes have to be considered as we query for full triples. The plan generator has to compute the cost of each index scan that can answer a triple pattern, which requires disc access because of the manner in which RDF-3X maintains its statistics (see Subsection 4.4.2).

Observing that the predicate is always given, we reduced the number of indexes considered from six to two, namely: the PSO and POS indexes. While this can result in some interesting orders not being considered, we have observed that for small join patterns, this restriction helped in reducing the time required for query optimization.
The code we use for generating query graphs is based on that used by RDF-3X’s query interface, but we have stripped it to the absolute minimum needed. Parts of this code responsible for encoding such things as filter conditions, OPTIONAL and UNION parts have all been left out as we do not need them for our queries.

An answer to a query is a set of variable bindings, as defined in Section 2.4. We implemented an operator which is added to the physical query plan, whose role is to construct bindings (mappings from variables to constants) from the answers returned by RDF-3X and the query. This is done by enforcing an order on variables that matches the order that RDF-3X keeps on the registers holding variable bindings.

5.4 Caching

In processing SPARQL queries, RDF-3X is unlikely to access the same disk page multiple times. This is very different for recursive queries, where the same disk page can be accessed multiple times. This is due to the manner in which variable bindings propagate and new subqueries are generated, often sharing the same constants with previous queries.

RDF-3X did not maintain any caches. It operates on top of a memory mapped file, which means that the operating system can perform some caching. Index pages are kept compressed on disk and are uncompressed once a page is read. We added caching to RDF-3X’s indexes. When caching is enabled, a hash table of cached pages is maintained. The key into the hash table is a page number and the values are uncompressed pages. Caching can be configured on a per-index basis. If caching is enabled for an index, then, when the page is requested, the hash table is queried for that page. If it exists, then the page is served from cache, otherwise, the page is read into memory, decompressed and then added to the cache. Caching was the only modification we had to make to RDF-3X’s infrastructure.

We added caching to RDF-3X’s full triples, aggregated, fully aggregated and statistics indexes. Each index type has a different page layout depending on the number of fields maintained per entry, requiring slightly different caching implementations.

5.5 Subquery Scheduler

The subquery scheduler is responsible for selecting a subquery from the current query for evaluation. Our approach to subquery scheduling is a *dynamic one*: decisions are made during query processing time. This allows for the consideration of the query’s current bindings, which changes as more subqueries of the query are evaluated.
Our subquery scheduler takes as input a conjunctive query and returns two conjunctive subqueries:

- The chosen subquery, which is a conjunctive subquery that will be evaluated next.
- The remaining subquery, which is a conjunctive subquery and which will be evaluated after the chosen subquery is evaluated successfully.

The chosen subquery can be composed of a single intensional, or a single purely intensional atom, or of one or more extensional atoms, as we show in Section 6.1.

If the evaluation of the chosen subquery succeeds, it will result in bindings for variables in the chosen subquery. These variables will usually occur in the remaining subquery, which means that the binding pattern of the remaining subquery changes. Because of this, once the remaining subquery has to be evaluated, it is sent to the subquery scheduler, with the latest bindings, and a subquery thereof is chosen.

As we discussed in Section 2.5, several subquery scheduling strategies exist. Because of this, our implementation allows the user to select a subquery scheduling strategy for answering a query. Our implementation comes with three subquery scheduling strategies implemented: left-most atom, most-bound-atom-first, and the seeking failure strategy which we present in Section 6.1.

5.6 Recursive Query Processor

The recursive query processor uses two top-down algorithms: SLD-resolution and QSQR. The choice between the two is done by the user, as there is no syntactic characterization of queries that result in cycles to automate this selection [BR86].

During query evaluation in both algorithms, subquery scheduling is done dynamically at every step when a new subquery has to be selected for evaluation. This allows for the consideration of newly found bindings at runtime to produce a good ordering.

5.6.1 SLD-resolution

To improve the performance of SLD-resolution, we added a rule cache which keeps track of answered subqueries. If a subquery was already answered, then any subsequent attempts to evaluate the same subquery are answered from cache.
The user can also choose to activate cycle detection in SLD-resolution. When the user chooses to do so, a stack of subqueries currently being evaluated is maintained. Every time a new subquery is issued, it is compared to the subqueries on the stack, and if there is a match then the user is warned that a cycle was detected and the cycle gets broken by failing the current subquery. This can result in the loss of an arbitrary number of answers [CK94], but can still be useful for a user not interested in the complete answer set.

5.6.2 QSQR

Query-Subquery (QSQ) refers to a family of top-down datalog query evaluation techniques. This family includes both iterative and recursive algorithms each of which can be applied in a tuple-at-a-time (depth-first) or a set-at-a-time (breadth-first) fashion. QSQ is based on SLD-resolution but it is DB-complete, which it achieves using tabling.

The initial QSQ algorithm was query-subquery recursive (QSQR) presented in [Vie86]. This algorithm was later found to be incomplete and was corrected [Vie87, Nej87]. One of the most cited source on the QSQR algorithm is [AHV95], which presents the algorithm using the same setting as the one used to present magic sets in [BMSU86], including adornments and sideways information passing. All recent works that use QSQR which we came across base their work on the description of the algorithm in [AHV95]. We claim that this description of QSQR is incomplete and propose a fix to this problem. To the best of our knowledge, we are the first to detect this incompleteness, which we communicated to the authors of [MBN08], resulting in a modification of their work¹.

We first introduce some concepts needed for presenting QSQR. We then present the complete version of the algorithm and finally show why the algorithm’s description in [AHV95] is incomplete using an example.

If we have the subquery atom $R_0(a, ?y)$, where $a$ is a constant and $?y$ is a variable. We say that the first argument of $R_0$ in the query is bound ($b$) and the second argument is free ($f$), denoted as $R_0^{bf}$. The superscript is called an adornment.

**Definition 5.1.** If $p$ is an n-ary predicate then an adornment $a$ is a an n-tuple of $b$’s and $f$’s, denoted $p^a$, where a $b$ (resp. $f$) indicates that the corresponding argument of the predicate is bound to a constant (or free, respectively).

Given a rule and an adornment for the atom forming the rule’s head, an adorned rule is formed by adding adornments to the rule body as follows:

¹See [http://www.mimuw.edu.pl/~nguyen/QSQS-revised-long.pdf](http://www.mimuw.edu.pl/~nguyen/QSQS-revised-long.pdf), which is a revised and extended version of [MBN08] with proofs for completeness
1. All occurrences of each bound variable in the rule head are bound.

2. All occurrences of constants are bound.

3. If a variable \( ?x \) occurs in the rule body, then all subsequent occurrences of \( ?x \) in subsequent literals are bound.

Take the rule

\[
R_0(?x, ?y) \leftarrow R_1(?x, ?z), R_2(?z, ?w), R_3(?w, ?y).
\]

If the rules head has the adornment \( R_{bf}^0 \), then the corresponding adorned rule is

\[
R_{bf}^0(?x, ?y) \leftarrow R_{bf}^1(?x, ?z), R_{bf}^2(?z, ?w), R_{bf}^3(?w, ?y).
\]

Two important observations can be made here. First, item 3 in the definition of adorned rules implies that different orderings of the atoms in the rule body imply different adornments. A second observation, also regarding item 3, is that of the idea of sideways information passing (SIP).

Item 3 states that once an atom is grounded, obtaining bindings for its variables (free arguments), the resulting bindings can be used to bind the same variables in subsequent predicates. This idea is known as sideways information passing. A sideways information passing strategy is simply a decision on the order in which atoms in a query will be evaluated [BR87].

A set of single-atom subqueries with the same adornment can be denoted as \((R^\gamma, J)\), where \( \gamma \) is an adornment of the query predicate \( R \), and \( J \) is a set of tuples with the values of the entries bound by \( \gamma \). \((R^\gamma, J)\) is called a generalized subquery. If \( R \) is a predicate and \( \gamma \) is an adornment for \( R \), then \( \text{bound}(R, \gamma) \) denotes the coordinates of \( R \) bound in \( \gamma \).

**Supplementary relations** keep track of variable bindings during left-to-right rule evaluation. An adorned rule with \( n \) atoms in its body has \( n + 1 \) supplementary relations: \( \text{sup}_0 \) through \( \text{sup}_n \), as shown below:

\[
R_{bf}^0(?x, ?y) \leftarrow R_{bf}^1(?x, ?z), R_{bf}^2(?z, ?w), R_{bf}^3(?w, ?y)
\]

\[
\uparrow \quad \uparrow \quad \uparrow \quad \uparrow
\]

\[
\text{sup}_0[?x] \quad \text{sup}_1[?x, ?z] \quad \text{sup}_2[?x, ?] \quad \text{sup}_3[?x, ?y]
\]

The attributes of the 0\( \text{th} \) supplementary relation, \( \text{sup}_0 \), are those variables bound in the head of the adorned rule. The attributes of the \( n\text{th} \) supplementary relation, \( \text{sup}_n \), are all the variables in the head of the adorned rule. For \( i \in [1, n - 1] \), the attribute set of
the $i^{th}$ supplementary relation, $\text{sup}_i$, is the set of variables which occur in both (i) $\text{sup}_0$ or one of $B_1,...B_i$ and (ii) $B_{i+1},...B_n$ or $\text{sup}_n$.

We are now ready to present QSQR in Algorithm 5.1, which takes as input a datalog program and a query over an intensional predicate. This algorithm uses the global variables $\text{ans}_p^\delta$ and $\text{input}_p^\delta$ for each adorned predicate $p^\delta$. Algorithm 5.1 calls Algorithm 5.2 which processes a generalized subquery in a set-at-a-time fashion by calling Algorithm 5.3 on rules.

$\text{ans}_r$ relations keep track of answers found for an adorned predicate, while $\text{input}_r$ relations keep track of subqueries that have already been evaluated to avoid running into cycles, as SLD does. Contrary to $\text{ans}_r$ and $\text{input}_r$ relations, supplementary relations are used as local variables in Algorithm 5.3.

Algorithm 5.1: QSQR($D, q$)

<table>
<thead>
<tr>
<th>Input:</th>
<th>A datalog program $D$ and an intensional query $q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>The global $\text{ans}_r$ and $\text{input}_r$ relations</td>
</tr>
<tr>
<td>Output:</td>
<td>All answers for $q$</td>
</tr>
</tbody>
</table>

1. begin
2. Set all $\text{ans}_r$ relations to be empty.
3. Set $(R^\gamma, J)$ to be the generalized query corresponding to $q$.
4. repeat
5. Set all $\text{input}_r$ relations to be empty.
6. Call QSQR/EVALUATE_GENERALIZED($D, (R^\gamma, J)$).
7. until Until no $\text{ans}_r$ relation has changed in the last iteration ;
8. return All answers for $q$ by performing a selection on $\text{ans}_r^\gamma$ using $J$
9. end

Algorithm 5.2: QSQR/EVALUATE_GENERALIZED($D, gq$)

| Input: | A datalog program $D$ and a generalized query $gq = (R^\gamma, J)$ |
| Input: | The global $\text{ans}_r$ and $\text{input}_r$ relations |

1. begin
2. Remove from $J$ all tuples in $\text{input}_r R^\gamma$.
3. if $J$ is empty then
4. exit
5. $\text{input}_r R^\gamma := \text{input}_r R^\gamma \cup J$.
6. foreach rule $\varphi$ defining $R$ do
7. Call QSQR/EVALUATE_RULE($D, \varphi, (R^\gamma, J)$)
8. end

The description of QSQR in [AHV95] and [MBN08] omits line 5 of Algorithm 5.1, which causes the algorithm to be incomplete as we demonstrate using the example setting of Figure 5.3(a).
Algorithm 5.3: QSQR_EVALUATE_RULE($D, \varphi, gq$)

**Input:** A datalog program $D$, a rule $\varphi$ and a generalized query $gq = (R^q, T)$

**Input:** The global $\text{ans}$ and $\text{input\_relations}$

1. **begin**
2. Remove from $T$ all tuples that do not unify with the head of the rule.
3. Set $\text{sup}_0 := T$.
4. Proceed sideways across the body $B_1, \ldots, B_n$ of the rule to the final supplementary relation $\text{sup}_n$ as follows:
   - **foreach** Atom $B_j$ do
     1. if $B_j$ has predicate $R' \in \text{EPred}$ then
        a. populate $\text{sup}_j$ using the extensional database.
     2. if $B_j$ has predicate $R^\delta, R' \in \text{IPred}$ then
        a. set $S := \text{sup}_{j-1}[\text{bound}(R', \delta)]$
        b. recursively call algorithm QSQR_EVALUATE_GENERALIZED($D, (R^\delta, S)$)
        c. use $\text{sup}_{j-1}$ and the current value of global variable $\text{ans}_R^\delta$ to populate $\text{sup}_j$
     3. Add tuples produced for $\text{sup}_n$ into the global variable $\text{ans}_R^q$.

**end**

\[ D \]
\[
\begin{align*}
    r_1 &: p(?x, ?y) \leftarrow s(?x, ?y) \\
    r_2 &: s(?x, ?y) \leftarrow u(?x, ?y) \\
    r_2 &: s(?x, ?y) \leftarrow p(?x, ?z), t(?z, ?y) \\
    f_1 &: u(a, b) \\
    f_2 &: t(b, c)
\end{align*}
\]

It is not difficult to see, using bottom-up evaluation, that there are two answers for the query: \{?y/b\} and \{?y/c\}. Omitting line 5 in Algorithm 5.1 would only return one answer, namely \{?y/b\}. The second answer cannot be obtained because in the second iteration of the outer-most loop of QSQR, line 2 of Algorithm 5.2 would result in an empty $J$ as $\text{input}_J$ already contains the tuple $\langle a \rangle$, as shown in Figure 5.3(b). Clearing $\text{input\_relations}$ between iterations of the outermost loop, as line 5 of Algorithm 5.1 does, solves this problem.

The presentation of QSQ here is based on that used to present magic sets in [BMSU86, BR87]. Magic sets is a rule-rewriting technique to optimize bottom-up query evaluation. Before starting bottom-up query evaluation, magic sets performs a rewriting of the rules based on the query, called magic sets rewriting. Then, a bottom-up query evaluation strategy, usually semi-naive is applied to answer the query using the rewritten rules.
We opted to implement QSQR in a tuple-at-a-time manner mainly because RDF-3X does not support set-at-a-time querying, and we wanted to keep the changes to RDF-3X as little as possible. An engine supports set-at-a-time (also known as set-oriented) querying if it is capable of accepting a set of queries at once and returning their answers in a manner which allows answers to be traced back to the original queries. While set-at-a-time evaluation can mean better disk access patterns, we think that the caching we added to RDF-3X reduces the impact of accessing disk in a tuple-at-a-time manner.

In choosing a DB-complete strategy to implement, the choice was between QSQR and magic sets. The two algorithms are highly related. Magic sets optimizes bottom-up query evaluation by rewriting rules to integrate information about the query, in a manner similar to top-down evaluation. QSQR achieves DB-completeness by resorting to iterative top-down evaluation, in a manner that resembles semi-naive bottom-up evaluation (see Subsection 2.4.2). We chose to implement QSQR because the of the lack of need for an explicit rule rewriting step, and because it highly resembles SLD-resolution, which we had already implemented.

The \textit{ans} and \textit{input} relations needed for tabling in QSQR are created during runtime, when they are needed. Hash Indexes are kept on each relation’s attributes to speed up point queries, which are issued frequently on those relations.

### 5.6.3 Memory Management

Both SLD-resolution and QSQR are recursive algorithms. Evaluating a query using one of those algorithms can result in very deep recursion. To reduce chances of a stack overflow, data structures whose life span extends beyond the method in which they were declared were allocated on the heap data area of the program, rather than on the stack. To make the allocation of data structures from the heap more efficient, we made extensive use of memory pools.

### 5.7 Reasoner Modes

RDRD is able to maintain the \textit{lineage} of an answer [MTdK+07]. Currently, our representation of lineage is a tree corresponding to the proof tree of the an answer, which also corresponds the the SLD-refutation resulting in the answer. As duplicate answers are not merged, lineage is purely conjunctive. We plan to change this such that duplicates are captured and merged, resulting in disjunctive lineage (in which case the tree structure turns into a DAG), which would also reduce the memory overhead of maintaining
lineage. In the context of URDF’s reasoning model, lineage is one component of the dependency graph, with ground hard rules being the other.

RDRD operates in two modes: standard mode and lineage mode. In standard mode, RDRD serves as a standard datalog reasoner, and does not have a notion of hard rules or maintain a dependency graph. RDRD’s default mode is lineage mode, but can be changed to standard mode if desired.

Lineage mode is used when RDRD runs as part of URDF. This means that in addition to standard datalog-style reasoning, it has to be able to support hard rules and maintain a dependency graph. Lineage mode can also be interesting in itself, if the user wants to see the proof tree corresponding to an answer.

In lineage mode, RDRD has to cope with hard rules. When a fact is generated by the reasoner, it is unified with the hard rules. If this unification is successful, it generates a query that, when answered, will generate a competitor set.

An answer to a query is a substitution, as stated in Definition 2.25. Figure 5.4 shows how we represent a substitution. A substitution is composed of a set of variable bindings and lineage. The lineage of a substitution is set to NULL in standard mode, and maintains a reference to a lineage object in lineage mode. The lineage of a substitution is composed of (i) a set of competitor sets that might be triggered by a fact generated from the substitution, (ii) a set of extensional facts that contributed to finding the substitution, and (iii) a set of ground soft rules which contributed to finding the substitution. The manner in which the ground soft rules are represented gives rise to the tree structure of lineage.

![Figure 5.4: UML representation of a substitution](image_url)
A competitor set is represented by the query atom corresponding to the particular instance of the hard rule, and a set of substitutions with answers to this query.
Chapter 6

Optimizing Recursive Queries

This chapter builds on the previous one and describes the optimizations we added to RDRD. Some minor optimizations, such as caching for SLD-resolution, were already discussed in Chapter 5. Section 6.1 presents our approach to join ordering, one of the main contributions of this thesis, which focuses on trying to detect the failure of a query as soon as possible, and attempts to achieve better utilization of the underlying storage engine’s ability to perform joins on extensional predicates. Section 6.2 presents some optimizations which can be utilized depending of what mode the recursive query processor is operating in. Finally, Section 6.3 discuss how we perform unification between subqueries and rule heads in RDRD.

6.1 Subquery Scheduling: Seeking Failure

Section 2.5 presented the importance of join ordering (subquery scheduling) in deductive databases, and some approaches of doing so, most of which are based on estimating the sizes of intensional relations.

Adaptive sampling gave an elegant way of sampling a relation with guarantees on accuracy. We do not use sampling for two main reasons. First of all, sampling is expensive, especially for our case of recursive relations, where tight bounds on the runtime for evaluating a sample are not available. During sampling time, we do not have the size estimates (which we are trying to obtain), so subquery scheduling will be naive making sampling take a long period of time. If the rule set in a knowledge base changes frequently, as we expect in our setting, then sampling becomes unattractive. Secondly, expressing the cost of a sample as a function of its size does not seem sufficient. With recursion, it is possible, as we have observed, that a query might result in deep recursion, which translates into lengthy runtime, while still returning a small result set.
We also presented an approach for estimating the sizes of intensional relations based on domain knowledge. Such an approach is not well-suited for our needs. Take for instance the knowledge base shown in Figure 3.3. There, we give an intensional definition of the bornIn relation. Our domain knowledge tells us that it is a functional relation, but we have the hard rules as part of the knowledge base namely because we expect this property to be violated.

As a further example, take the ancestor relation shown in Figure 2.4. It is not clear how domain knowledge helps in this particular case. How many ancestors or descendant does a person have? In a realistic knowledge base, can we expect a king, a president and a scientist who are contemporaries to have the same number of ancestors?\footnote{Under the closed world assumption, ancestors that do not exist in the knowledge base are assumed to be non-existent.}

To summarize, a naive approach of scheduling subqueries, such as the leftmost-atom-first approach does not work well for our setting. Approaches completely based on size estimation are also not very well suited for our application. Sampling can be expensive and domain knowledge is insufficient. Moreover, using the approach of relational query optimization, which relies on relation sizes, ignores an important source of overhead in our setting, namely recursion. In our setting, rules can be highly recursive leading to very deep recursion levels during evaluation.

We take a different approach to scheduling subqueries than the ones mentioned above. Our approach is based on one important observation: we are operating under the closed world assumption where all facts that cannot be established are assumed to be false. In a conjunctive query setting, this means that the failure of one query atom causes the entire query to fail. Our approach is to discover failure as soon as possible to avoid deep recursion, which would trigger disk access along the way. Algorithm 6.1 shows our subquery scheduling strategy, one of the main contributions of this thesis, which we discuss in further detail.

The algorithm returns two sets of atoms, each containing conjuncts of a conjunctive subquery, one for the subquery to be issued next and the other for the remaining part of the original query (see Section 5.5). It prioritizes built-in atoms with a permissible binding (Line 3-4). Those atoms are the cheapest to evaluate, they can be evaluated without resorting to disk or triggering recursion. If such an atom fails, then the rest of the query does not need to be evaluated.

If no built-in atom with a permissible binding pattern is available, then the algorithm makes its choice based on the promise of the remaining atoms. Our approach for selecting among predicates is called Least Promising First (LPF), where the atom with the
Chapter 6. Optimizing Recursive Queries

Algorithm 6.1: SELECT_SUBQUERY_LP"(q,chain)

Input: A conjunctive query \( q = \? \leftarrow B_1, \ldots, B_n \)
Input: \( \text{chain} \in \{\text{true, false}\} \)
Output: A double: \( (\text{set of current subquery atoms, set of remaining subquery atoms}) \)

begin

\( Q = \text{set of atoms in } q \)

if \( Q \) contains an atom \( B_i, 1 \leq i \leq n, \) with a built-in predicate and a permissible binding pattern then

\[ \text{return } (\{B_i\}, Q - \{B_i\}) \]

else

\( \text{Evaluable} = \text{set of atoms in } q \) with non-built-in predicates.

\( \text{chosenSubquery} = \{q^\prime\}, q^\prime \in \text{Evaluable and } \forall q'' \in \text{Evaluable, } \text{prom}(q^\prime.\text{predicate}^{\prime}.\text{adornment}) \leq \text{prom}(q''.\text{predicate}^{\prime}.\text{adornment}) \)

if \( \text{chain} = \text{true} \) then

\( \text{chosenSubquery} = \text{CHAIN}(Q, \{\text{chosenSubquery}\}) \)

\[ \text{return } (\text{chosenSubquery}, Q - \text{chosenSubquery}) \]

end

least promise of returning an answer is scheduled first. Before defining the promise of an atom, we introduce some necessary concepts.

The domain of argument \( i \) of predicate \( p \), denoted \( \text{Domain}(p.i) \), is the set of constant values that argument \( i \) can take. The active domain of argument \( i \) of an extensional predicate \( p \), denoted \( \text{ActiveDomain}(p.i) \), is the set of constant values occurring in argument \( i \) [AHV95]. We assume that a domain is defined for every argument of every predicate.

The concept of promise is based on a probabilistic model that relies on statistics from extensional predicates and independence assumptions. If \( A_i, 1 \leq i \leq n \) are independent events then we have the following axioms:

\[ P(\neg A_i) = 1 - P(A_i) \]

\[ P(\bigwedge_{i=1}^{n} A_i) = \prod_{i=1}^{n} P(A_i) \]

\[ P(\bigvee_{i=1}^{n} A_i) = 1 - P(\bigwedge_{i=1}^{n} \neg A_i) = 1 - \prod_{i=1}^{n} (1 - P(A_i)) \]

We are now ready to define the concept of promise.
Definition 6.1 (Promise). The promise \( prom \) of an adorned predicate given a maximum depth \( maxd \) is (recursively) defined as follows:

\[
\begin{align*}
\text{prom}(E^\gamma, d, maxd) &= \prod_{l_i \in \gamma_i, i = b} |\text{Active Domain}(R.i)|, 1 \leq i \leq \text{arity}(R) \\
\text{prom}(\bigwedge_{i=1}^n A_i^\delta_i, d, maxd) &= \prod_{i=1}^n \text{prom}(A_i^\delta_i, d, maxd) \\
\text{prom}(\bigvee_{i=1}^n A_i^\delta_i, d, maxd) &= \begin{cases} 
1 & \text{if } d < maxd \\
\top & \text{otherwise}
\end{cases} \\
\text{prom}(\top, d, maxd) &= 1.0
\end{align*}
\]

Where \( E \) is an extensional predicate, \( \top \) is a special predicate called the cutoff predicate, and \( A_i \) is an arbitrary predicate. The promise of an atom is defined as the promise of the adorned predicate corresponding to the atom.

Intuitively, the promise of an adorned predicate can be interpreted as the probability of finding at least one binding for it given its binding pattern.

We demonstrate the concept of promise with an example. Figure 6.1(a) shows the rules defining the ancestor relation, first introduced in Figure 2.4. It also gives some information about the sizes of the arguments of the extensional parent predicate. Note that both arguments of the parent predicate are of the same domain: \( PERSON \).

The justification for the manner in which the promise of an extensional predicate is computed is as follows: using the example of Figure 6.1(a), if the subquery \( parent^{bf} \) is asked, then the bound argument, the subject, can take one of 1000 possible values in the set \( PERSON \). Now, the subject of the \( parent \) predicate is defined (extensionally) for 300 members of the set \( PERSON \), which means that for the remaining 700, evaluating \( parent^{bf} \), will fail. So, we say that \( parent^{bf} \) has a promise of 300/1000, which corresponds to the probability of not failing (or returning at least one answer).

The definition of the promise of an adorned extensional predicate is based on the assumption of independence between its arguments, hence the product. Note that the empty product, which corresponds to an atom with no bound arguments, evaluates to 1.0, indicating that an extensional atom with no bindings is certain to return an answer.

The promise of an adorned intensional predicate, such as \( ancestor^{bb} \) in Figure 6.1(a), is computed based on the structure of the rules defining that predicate, and the propagation of variables within those rules. The body of a single rule corresponds to a conjunction of atoms, all of which have to be evaluated successfully for the conjunction to return an answer. The promise of this conjunction is computed from the promise of the conjuncts with the assumption of independence between them.
A predicate, such as \textit{ancestor}, can be defined by multiple rules. The evaluation of a query over such a predicate succeds if at least one of the definitions successfully returns an answer, which is where disjunctions comes from. Again, the computation of the promise of a disjunction of adorned predicates assumes independence between the disjuncts.

The term \textit{maxd} is the cutoff depth for computing promise. At that depth, an intensional predicate is not expanded. Instead, the cutoff predicate is plugged. The definition of the promise of a cutoff predicate is a pessimistic one, as it can be interpreted that the probability of returning a result from the part that was cut off is 1.0. Giving the cutoff predicate a promise of 1.0 serves to penalize recursively defined queries by giving them a higher promise and, therefore, making them less likely to be scheduled in the existence of extensional predicates.

Figure 6.1(b) shows the computation of \textit{promise(ancestor\textit{bb}, 0.2)} using a tree. Each internal node is labeled with a conjunction or a disjunction denoting a definition of an intensional predicate. Leaves are labeled with extensional predicates or the cutoff predicate. The root of the tree is labeled with the adorned intensional predicate corresponding to an intensional subquery. The number in the label of each node denotes that node’s promise.

\begin{align*}
  r_1: & \text{ancestor}(\text{?x,?y}) \leftarrow \text{parent}(\text{?x,?y}) \\
  r_2: & \text{ancestor}(\text{?x,?y}) \leftarrow \text{ancestor}(\text{?x,?z}) \land \text{parent}(\text{?z,?y}) \\
  |Active \ Domain(\text{parent.1})| = 300 \\
  |Active \ Domain(\text{parent.2})| = 500 \\
  |Domain(\text{parent.1})| = |Domain(\text{parent.2})| = |\text{PERSON}| = 1000
\end{align*}

(a) Ancestor Relation

\begin{align*}
  \text{ancestor}^{\text{bb}} : & 0.575 \\
  \lor : & 0.575 \\
  \text{parent}^{\text{bb}} : & \begin{array}{c}
    300 \\
    \frac{1000}{1000}
  \end{array} \land : 0.5 \\
  \lor : & 1.0 \\
  \text{parent}^{\text{fb}} : & \frac{500}{1000} \\
  \land : & 0.3 \\
  \land : & 1.0 \\
  \text{parent}^{\text{ff}} : & 1.0 \\
  \lor : & 1.0 \\
  \lor : & 1.0 \\
  \text{parent}^{\text{ff}} : & 1.0
\end{align*}

(b) \textit{promise(ancestor}^{\text{bb}}, 2)
Chapter 6. Optimizing Recursive Queries

\[ q = \left( E_1(a, z) \land E_2(z, y) \land I_2(z, y) \right) \]

(a) Query

\[ q' = \left( E_1(a, z) \right) \]

(b) Selecting a Subquery

\[ q = \left( E_1(a, z) \land E_2(z, y) \right) \]

(c) Chaining \( q' \) from \( q \)

Figure 6.2: Example of chaining

The promise of all binding patterns of predicates is computed during pre-processing time and stored in a hash indexes which can be queried very efficiently during query time when ordering decisions need to be made.

Note that promise considers only the binding pattern and not the actual bindings, which also makes it suitable for join ordering in a set-at-a-time setting.

Going back to Algorithm 6.1, lines 8-9 invoke chaining if requested by the user. A subquery \( q' \) is chained from a query \( q \) by adding all atoms in \( q \) which have a variable occurring in \( q' \) (implying a join condition) to \( q' \) iteratively until no more atoms can be added to \( q' \), as shown in Algorithm 6.2. This recursive algorithm terminates because a query is always finite. Figure 6.2(a) shows a query, from which a subquery is selected in Figure 6.2(b) and then chained in Figure 6.2(c).

Algorithm 6.2: CHAIN(\( q, q' \))

Input: A set of atoms \( q \) for the query
Input: A set of atoms \( q' \) for the subquery to be chained
Output: A chaining of \( q' \) from \( q \)

1 begin
2 \( S = \) set of atoms in \( q \) with variables occurring in \( q \)
3 \( S = S - q' \)
4 if \( S \neq \emptyset \) then
5 \quad return CHAIN(\( q, q' \cup S \))
6 else
7 \quad return \( q' \)
8 end

Chaining allows better utilization of the capabilities of the storage engine to perform joins. Evaluating extensional atoms that share common variables against the storage engine one by one would correspond to performing a nested loop join (NLJ) and would miss the chance for join ordering. Sending a conjunctive query to the storage engine gives the chance for better join strategies and orders.
6.2 Reasoner Modes

The distinction between standard and lineage modes affects the speed of query answering. First of all, there is the overhead of maintaining lineage. Secondly, this distinction decides whether we are looking for all answers (standard mode) or all proofs (lineage mode). For example, in standard mode, if we are trying to answer a query composed of a ground intensional atom, such as ? ← bornIn(Al Gore, Washington, D.C.), then finding an answer in the extensional database means that there is no need to resort to evaluating any rules. On the other hand, if we are operating in lineage mode, where we are interested in all proofs, then we have to show all proof trees of a facts, which means that rules would need to be evaluated for the case mentioned above.

When standard mode is not requested, we also perform rule merging during preprocessing. Rule merging is applied whenever a purely intensional predicate is (i) defined by one rule (ii) and has only extensional atoms in its body.

Merging rules helps eliminate unnecessary nested loop joins. Figure 6.3 shows an example of rule merging. In the example, $E$ denotes and extensional predicate, $I$ an intensional predicate, and $PI$ a purely intensional one. Note the renaming of variables to ensure the correctness of merging.

\[
\begin{align*}
    r_1 &: I_1(?x, ?y) ← I_2(?x, ?z) ∧ E_1(?z, ?y) \\
    r_2 &: PI_2(?x, ?y) ← E_2(?x, ?z) ∧ E_3(?z, ?y)
\end{align*}
\]

(a) Original Rules

\[
    r_1 : I_1(?x, ?y) ← E_2(?x, ?t) ∧ E_3(?t, ?z) ∧ E_1(?z, ?y)
\]

(b) Rules After Merging

\[ \text{Figure 6.3: Rule merging} \]

6.3 Unification

One important operation in a resolution-based system is unification. The purpose of unification is to find if a rule can be used to answer a subquery, and if so, produces a substitution which transforms the rule into a form appropriate for answering the subquery.

The standard procedure for unification is to find the most general unifier (MGU) of the subquery atom and the rule head. This procedure presented in Algorithm 6.3 for the Datalog setting, where terms are either variables or constants. Applying the MGU procedure can be expensive as the unified terms first need to be standardized apart and the iterations of the MGU procedure involve applying all substitutions found so far.
to the terms being unified. Moreover, the procedure has to be applied frequently during query processing, so speeding it up is crucial.

When the rules and query do not contain atoms with multiple occurrences of the same variable, and rules do not contain constant arguments, then unification\(^2\) is done according to Algorithm 6.4. The substitution obtained from NAIVE UNIFY is applied only to the rule \(r\), to generate \(r'\). The body of \(r'\) is sent as a query to the reasoner. Each substitution obtained \(\theta\) is an answer to the head of \(r'\) but not the original subquery, as they differ in their variables. A simple mapping is done to change the substitution to become an answer to the original subquery. Note that a similar mapping is also needed for MGU, as the head of \(r'\) can contain variables from both the subquery atom and the head of the rule \(r\), as a result of unification. Therefore, this step does not introduce any new overhead.

**Algorithm 6.3: MGU\((q, r)\)**

| Input: A subquery \(q = ? \leftarrow C\) with no multiple occurrences of the same variable |
| Input: A rule \(r = A \leftarrow B_1, \ldots, B_n\) with variable arguments and no multiple occurrences of the same variable in the same atom |

| Output: A substitution \(\theta\) or failure |

```python
begin
  \(\theta = \{\}\)
  if \(C.predicate \neq A.predicate\) then
    return fail
  \(i = 1\)
  while \(i \leq \text{arity}(C)\) do
    \(s = |\theta|\)
    if \(A.i \neq C.i\) then
      if isVar\((A.i)\) then
        \(\theta = \theta \cup \{A.i/C.i\}\)
      else if isVar\((C.i)\) then
        \(\theta = \theta \cup \{C.i/A.i\}\)
      else
        return fail
    apply \(\theta\) to both \(C.j\) and \(A.j, j > i\)
    \(i = i + 1\)
  return \(\theta\)
end
```

The decision of whether to use NAIVE UNIFY or the fully fledged MGU procedure is based on the analysis of rules during pre-processing and the query during runtime to see if they match the conditions stated in the description of NAIVE UNIFY.

\(^2\)This algorithm is not strictly a unification algorithm, as the resulting substitution can be applied only to the rule, but not to the subquery, and the resulting rule head matches that of the subquery up to variable renaming.
Algorithm 6.4: NAIVE_UNIFY(q, r)

Input: A subquery q =? \(\leftarrow C\) with no multiple occurrences of the same variable

Input: A rule r = A \(\leftarrow B_1,...,B_n\) with variable arguments and no multiple occurrences of the same variable in the same atom

Output: A substitution \(\theta\) or failure

begin

\[ \theta = \{\} \]
\[ \text{if } C.\text{predicate} \neq A.\text{predicate} \text{ then} \]
\[ \text{return fail} \]
\[ i = 1 \]
\[ \text{while } i \leq \text{arity}(C) \text{ do} \]
\[ \text{if isConst}(C.i) \text{ then} \]
\[ \theta = \theta \cup \{C.i/A.i\} \]
\[ i = i + 1 \]
\[ \text{return } \theta \]

end
Chapter 7

Experimental Evaluation and Results

This chapter presents experimental results for RDRD and specific components of it. We describe the setting for our experiments, present the results, and discuss them.

7.1 Setting

7.1.1 System Configuration

All experiments were conducted on a Dell Optiplex 760 PC with an Intel Pentium Processor E5200, 3.2GB of memory running a 64-bit Linux 2.6.30 kernel. For cold cache experiments, we used the /proc/sys/vm/drop_caches kernel interface before starting each run of an experiment to clear the operating system’s buffer cache. For experiments where time is measured, the lowest of at least five runs of each experiment is shown in the results. All times are in milliseconds.

RDRD was developed entirely in C++, it makes use of the following external libraries: google-sparsehash (for caching and representation of bindings), google-glog (for logging), Boost Pool Library (for memory pools), Boost Smart Pointer (for lineage maintenance), and Boost Spirit parser (for parsing user input). We used the g++ compiler, part of GNU GCC version 4.3.2. Compilation was performed with optimization level 2 (-O2).

For other systems which use Java, Java 1.6.0 was used.
7.1.2 Data Sets

YAGO

YAGO (Yet Another Great Ontology) is an ontology automatically extracted from Wikipedia and WordNet [SKW07]. It contains 20 million facts about more than 2 million entities. YAGO has a manually confirmed accuracy of 95%. YAGO contains both taxonomic relations, such as subclassOf, and non-taxonomic ones, such as bornIn.

LUBM

LUBM (Lehigh University BenchMark) is a benchmark for evaluating OWL knowledge base systems [GPH05]. It generates synthetic data modeling the university domain, with departments, groups, professors, and students etc., and comes with 14 queries. LUBM queries target a subset of OWL lite, with a focus on class and property inclusion, which are also expressable in Datalog (see Section 2.3).

7.1.3 Experiments

We performed the following experiments:

1. comparison of query time for relational queries between RDF-3X, with its complete query optimizer, and our recursive query processor, which skips the optimizer for some queries and uses a scaled down version of it for others,

2. comparison of the performance of the two unification algorithms we use: MGU and NAIVE_UNIFY,

3. comparison between the LPF and MBF subquery scheduling strategies,

4. analysis of the effect of chaining, and

5. comparison with two other systems: Jena and IRIS using both YAGO and LUBM datasets.

Two metrics were used for our experiments:

1. the time required to answer a query, and

2. the Number of Intermediate Subqueries (NIS) issued during query processing, which is equivalent to the number of times subquery scheduling is performed.
Chapter 7. Experimental Evaluation and Results

The first metric is a natural one, most query answering systems have small query response times as their ultimate goal. The second metric is inspired by the fact that, in LPF, we are seeking to detect failure as soon as possible, which we would like to happen in as little time as possible. As we will see, this is not always the case.

Appendix A lists all queries and rules used in our experiments. The rules and queries for the YAGO experiments are taken from [TSSN10], while those for LUBM are based on the queries and ontology described in [GPH05].

7.2 Results

7.2.1 Handling of Extensional Queries

Table 7.1 shows the result of issuing eight extensional queries using both RDF-3X's query infrastructure and that of RDRD with cold cache. In Section 5.3, we describe how we handle single-atom extensional queries (such as QE1-2), where we skip the query optimizer completely. For queries with multiple atoms (such as QE3-8), we consider only two possible indexes rather than the six which RDF-3X considers, thus reducing the overhead of selectivity estimation, which is performed through disk.

The same data is plotted in Figure 7.1. This data shows the high cost of query optimization which RDF-3X performs, for our setting. We expect join queries issued through RDRD to be small. For this class of join queries, the data shows that we benefit from considering less index scans, and, therefore, less query plans.

The eight queries used in this experiment are over the YAGO dataset. They were chosen to be similar to what we expect the storage backend to need to handle. Single atom queries with partial bindings are expected to be most frequent, either because chaining can be disabled or simply because of the structure of the rules, which often does not allow for much chaining. The rest of the queries resemble what we would expect to result from chaining: a set of atoms with common variables (joins), and partial bindings in most cases.

7.2.2 Unification

In Section 6.3, we presented the algorithm we use for unification under specific conditions, which we called NAIVE\textunderscore UNIFY, when the full power of the most general unifier (MGU) is not required.
Table 7.2.2 and the corresponding chart in Figure 7.2 show the result of running each of
the two unification algorithms on the subquery atom $p^{bb} : p(?z, ?w)$, $p^{bf} : p(a, ?w)$, and
$p^{ff} : p(a, b)$ with a rule with the head $p(?x, ?y)$ $10^6$ times.

Note that NAIVY_UNIFY can be used in this case since the rule head has no
constants and neither the rule head nor the subquery atom have multiple occurrences
of the same variable. The results here are as expected: at each step, MGU attempts to
propogate the latest substitution to the remaining part of the terms being unified, resulting
in a constant time for unification across all three cases. NAIVE_UNIFY only
constructs a substitution, without the need for any application during unification. Thus,
NAIVE_UNIFY requires both less time than MGU and less time with a smaller number
of constants.
Experimental Evaluation and Results

### Table 7.2: Times for $10^6$ executions of MGU and NAIVE_UNIFY is milliseconds

<table>
<thead>
<tr>
<th>Unification Algorithm</th>
<th>$p(b,b)$</th>
<th>$p(b,f)$</th>
<th>$p(f,f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGU</td>
<td>507</td>
<td>4837</td>
<td>481</td>
</tr>
<tr>
<td>Naive_Unify</td>
<td>265</td>
<td>133</td>
<td>12</td>
</tr>
</tbody>
</table>

#### 7.2.3 Evaluation of LPF Subquery Scheduling Strategy

In this experiment, we issued 11 queries from [TSSN10] over the YAGO dataset to RDRD using both LPF and MBF as subquery scheduling strategies with cold cache. Our system was designed with the setting for the queries in this experiment in mind. The LPF strategy, which tries to detect the failure of a query as soon as possible assumes that the knowledge base will not contain information to answer many (sub-) queries. Detecting this as soon as possible allows query processing to terminate early, so that the user can, possibly, append more rules to the system in hope of getting an answer to his original query.

The queries we used, along with the rules used to answer each query, are given in Appendix A. The selection of rules was done based on knowledge about entities in the query. For example, Q5, which deals with the academic domain, does not use the rule YR11, which is restricted to the acting domain. Table 7.3 and Figure 7.3 show the query times, and Table 7.4 and Figure 7.4 show the corresponding NIS.
Chapter 7. Experimental Evaluation and Results

<table>
<thead>
<tr>
<th>Query</th>
<th>LPF</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>207</td>
<td>226</td>
</tr>
<tr>
<td>Q2</td>
<td>132</td>
<td>146</td>
</tr>
<tr>
<td>Q3</td>
<td>378</td>
<td>407</td>
</tr>
<tr>
<td>Q4</td>
<td>359</td>
<td>366</td>
</tr>
<tr>
<td>Q5</td>
<td>1110</td>
<td>5290</td>
</tr>
<tr>
<td>Q6</td>
<td>45</td>
<td>60719</td>
</tr>
<tr>
<td>Q7</td>
<td>89626</td>
<td>84913</td>
</tr>
<tr>
<td>Q8</td>
<td>1682</td>
<td>2223</td>
</tr>
<tr>
<td>Q9</td>
<td>253</td>
<td>254</td>
</tr>
<tr>
<td>Q10</td>
<td>260</td>
<td>267</td>
</tr>
<tr>
<td>Q11</td>
<td>1423</td>
<td>2301</td>
</tr>
</tbody>
</table>

Table 7.3: Query times in milliseconds for recursive queries based on YAGO using LPF and MBF

<table>
<thead>
<tr>
<th>Query</th>
<th>LPF</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>123</td>
<td>129</td>
</tr>
<tr>
<td>Q2</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>Q3</td>
<td>804</td>
<td>1924</td>
</tr>
<tr>
<td>Q4</td>
<td>421</td>
<td>421</td>
</tr>
<tr>
<td>Q5</td>
<td>562</td>
<td>28029</td>
</tr>
<tr>
<td>Q6</td>
<td>1</td>
<td>5224142</td>
</tr>
<tr>
<td>Q7</td>
<td>7386444</td>
<td>7524333</td>
</tr>
<tr>
<td>Q8</td>
<td>2083</td>
<td>22436</td>
</tr>
<tr>
<td>Q9</td>
<td>103</td>
<td>136</td>
</tr>
<tr>
<td>Q10</td>
<td>113</td>
<td>113</td>
</tr>
<tr>
<td>Q11</td>
<td>9030</td>
<td>27543</td>
</tr>
</tbody>
</table>

Table 7.4: Number of intermediate subqueries for recursive queries based on YAGO using LPF and MBF

From this data, LPF can be seen as better guided MBF. Queries Q6 and Q7 are two interesting cases. For Q6, the only query here with an empty result, LPF has two options, either the atom with the predicate \textit{bornIn} or that with \textit{locatedIn}. \textit{bornIn}, which is an intensional predicate, has a higher promise than \textit{locatedIn}, an extensional predicate (see Section 6.1). Because \textit{locatedIn} fails, LPF has made the right choice. For MBF, in the case of a tie, MBF goes with the leftmost atom in the query, which is \textit{bornIn}. This results in a deep recursion, but the final query result is empty.

On the other hand, comparing the query time and NIS for Q7, LPF, which is based on detecting failure as soon as possible, can be eager to evaluate extensional predicates by going to disk. This is reflected in the higher query time, yet lower NIS. The same conclusion can be made by observing both the differences in query time and NIS between LPF and MBF, where the differences in NIS tend to be more substantial.
Chapter 7. Experimental Evaluation and Results

**Figure 7.3:** Query times in milliseconds for recursive queries based on YAGO using LPF and MBF

**Figure 7.4:** Number of intermediate subqueries for recursive queries based on YAGO using LPF and MBF
Chapter 7. Experimental Evaluation and Results

<table>
<thead>
<tr>
<th>Query</th>
<th>LPF+NoChain</th>
<th>LPF+Chain</th>
<th>Time in Query Optimizer for Chaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>207</td>
<td>331</td>
<td>124</td>
</tr>
<tr>
<td>Q2</td>
<td>132</td>
<td>181</td>
<td>94</td>
</tr>
<tr>
<td>Q3</td>
<td>378</td>
<td>372</td>
<td>172</td>
</tr>
<tr>
<td>Q4</td>
<td>359</td>
<td>348</td>
<td>131</td>
</tr>
<tr>
<td>Q5</td>
<td>1110</td>
<td>1491</td>
<td>605</td>
</tr>
<tr>
<td>Q6</td>
<td>45</td>
<td>43</td>
<td>1</td>
</tr>
<tr>
<td>Q7</td>
<td>89626</td>
<td>241869</td>
<td>123917</td>
</tr>
<tr>
<td>Q8</td>
<td>1682</td>
<td>1984</td>
<td>478</td>
</tr>
<tr>
<td>Q9</td>
<td>253</td>
<td>195</td>
<td>146</td>
</tr>
<tr>
<td>Q10</td>
<td>260</td>
<td>93</td>
<td>65</td>
</tr>
<tr>
<td>Q11</td>
<td>1423</td>
<td>1755</td>
<td>545</td>
</tr>
</tbody>
</table>

Table 7.5: Query times in milliseconds for recursive queries based on YAGO with and without chaining

7.2.4 Chaining

In Algorithm 6.1, if chaining is enabled, then, once an extensional predicate is chosen to be answered next, all other extensional predicates which (transitively) can be joined with it are grouped into a conjunctive query and sent to the underlying storage engine. The justification for this is that the (relational) storage engine will perform joins efficiently, while sending individual atoms to the query engine necessarily implies nested loop joins.

Table 7.5 shows the time for answering queries Q1-Q11, which we described above, with and without chaining and with cold cache. The last column shows the total amount of time spent in the query optimizer for the case when chaining was enabled (after the modifications given in Section 5.3.2). The same information is given in Figure 7.5. Table 7.6 and Figure 7.6 show the effect of chaining on NIS.

While, as expected, NIS falls when chaining is enabled, the same cannot be said about query time. The time spent in query optimization, which is required for a reasonable join ordering, is too long for RDF-3X. The only query where we see a significant improvement is Q10, which is an extensional query.

One way to remedy this would be to cache query plans, so that once a query is issued, subsequent queries with the same predicates and binding pattern can be executed using the cached query plan.

It is also worth noting that RDF-3X cannot fall back on NLJ when it estimates that the intermediate results of the join are too small. This, however, is a secondary issue as it is necessary to access disk-based statistics before making such a decision.
Chapter 7. Experimental Evaluation and Results

Figure 7.5: Query times for recursive queries based on YAGO with and without chaining

<table>
<thead>
<tr>
<th>Query</th>
<th>LPF+NoChain</th>
<th>LPF+Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>123</td>
<td>108</td>
</tr>
<tr>
<td>Q2</td>
<td>29</td>
<td>19</td>
</tr>
<tr>
<td>Q3</td>
<td>804</td>
<td>510</td>
</tr>
<tr>
<td>Q4</td>
<td>421</td>
<td>321</td>
</tr>
<tr>
<td>Q5</td>
<td>562</td>
<td>498</td>
</tr>
<tr>
<td>Q6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q7</td>
<td>7386444</td>
<td>6347944</td>
</tr>
<tr>
<td>Q8</td>
<td>2083</td>
<td>1822</td>
</tr>
<tr>
<td>Q9</td>
<td>103</td>
<td>48</td>
</tr>
<tr>
<td>Q10</td>
<td>113</td>
<td>53</td>
</tr>
<tr>
<td>Q11</td>
<td>9030</td>
<td>7571</td>
</tr>
</tbody>
</table>

Table 7.6: Number of intermediate subqueries recursive queries based on YAGO with and without chaining
Chapter 7. Experimental Evaluation and Results

7.3 Comparison

We compared our system to two other recursive query processors: Jena and IRIS. We used both the queries for YAGO, which was used in the experiments above, and a new set of queries based on the LUBM benchmark.

**Jena** is an open source Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, SPARQL and includes a rule-based inference engine [Jen10]. Jena’s rule engine supports two main evaluation strategies: bottom-up and top-down with tabling using SLG (see Subsection 2.4.3). In all experiments here, we used the top-down rule engine with tabling enabled. As a storage backend, we used TDB (the Jena Tuple Database), which is a component of Jena that provides for large scale storage and querying of RDF datasets using a pure Java engine [TDB10]. TDB was the best storage engine we tried with Jena in terms of performance, we tried both PostgreSQL and Jena SDB, but the performance of TDB was superior to both. TDB uses three B+-indexs on SPO, POS, and OSP to store the triples of an RDF graph. TDB stores identifiers in the indexes, and uses a dictionary to map between identifiers and terms. We used it for its superior performance when interfaced with Jena. We used Jena 2.6.3 with TDB 0.8.7
Table 7.7: Comparison to other systems using queries based on YAGO, times in milliseconds

<table>
<thead>
<tr>
<th>Query</th>
<th>RDRD without chaining</th>
<th>Jena+TDB</th>
<th>IRIS+PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cold Cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>132</td>
<td>607</td>
<td>3304</td>
</tr>
<tr>
<td>Q10</td>
<td>260</td>
<td>1025</td>
<td>2570</td>
</tr>
<tr>
<td>Q12</td>
<td>401</td>
<td>995</td>
<td>1839</td>
</tr>
<tr>
<td>Q13</td>
<td>1052</td>
<td>3152</td>
<td>1861</td>
</tr>
<tr>
<td></td>
<td>Warm Cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>52</td>
<td>1582</td>
</tr>
<tr>
<td>Q10</td>
<td>4</td>
<td>63</td>
<td>1093</td>
</tr>
<tr>
<td>Q12</td>
<td>77</td>
<td>164</td>
<td>451</td>
</tr>
<tr>
<td>Q13</td>
<td>78</td>
<td>101</td>
<td>443</td>
</tr>
</tbody>
</table>

**IRIS** (Integrated Rule Inference System) is an open-source Datalog engine supporting built-in predicates [IRI10]. It is designed to be highly configurable, allowing for different Datalog evaluation strategies and the definition of custom data types and predicates. We used semi-naive bottom-up evaluation with magic sets rewriting as our evaluation strategy for IRIS version 0.5.8. As a storage backend, we used PostgreSQL 8.4 with a facts table and the following indexes: SPO, SOP, OSP, OPS, PSO, POS.

The two systems were selected because they are open source, with the ability to handle RDF data, which general Prolog engines cannot do without modification. They are also capable of interfacing with a disk-based storage engine, which makes the setting similar to that of RDRD.

Table 7.7 show the results for four queries based on YAGO using cold and warm cache. Figures 7.7 and 7.8 show plots of the same data. Note that we added Q12 and Q13 especially for this test. These two queries are over the famous ancestor relation. Both systems, Jena and IRIS, could not handle most of our YAGO queries (Q1, Q3-Q9, and Q11) which are highly recursive and are unmanageable without join ordering.

For the four queries in the table, the focus is on the efficiency of implementation, not the ability to perform join ordering, which both cannot do. For Q12, we changed the order of atoms in the recursive definition of ancestor when supplied to both Jena and IRIS to reduce the number of intermediate queries they generate for finding the answer (see Section 2.5). RDRD does not require this, as the ordering is dynamic. Q10 is a purely extensional query, and for it, as with all other queries in this test, we disabled chaining in our system.

For IRIS, the database interface was a bottleneck, which explains why Q12 and Q13 perform very similarly. Both RDRD and Jena require more time to answer Q13 than Q14, which is expected as it goes into deeper recursion and returns a larger result set.
Chapter 7. Experimental Evaluation and Results

The results for running the queries based on the LUBM benchmark are given in Table 7.8 on both cold and warm cache. Figures 7.9 and 7.10 show plots for the same data. Running those queries also exposed some issues in the manner in which we perform memory management in our system, which makes heavy use of heap-based memory pools (see Section 5.6.3). This issue arises when there are large intermediate results during query processing, requiring us to make some changes in the manner we perform memory deallocation in our system. This issue did not arise in the queries over the YAGO dataset, which generally have smaller sets of intermediate results.

The rules for LUBM queries are not as recursive as those we saw for the queries for YAGO. In fact, the only rule which can run into cycles (given the information we know about the data), is that expressing the transitivity of the \texttt{subOrganizationOf} relation. All other rules define new classes or express class hierarchies.

In this experiment, we compare RDRD with and without chaining enabled to Jena. In both cases, RDRD wins over Jena, with the best of the two performing considerably better than Jena. The comparison of RDRD with and without chaining gives mixed results, with no clear winner. We observed that chaining performed well when the number of intermediate results from individual atoms was large, but it performed worse when intermediate results were small, in which case the time for query optimization dominated that of answering a chained subquery. This is not an issue with chaining, but an issue with the choice of storage engine and the space of query plans it considers.
Chaining would be beneficial for our setting if we are able to detect small intermediate results and fall back on nested loop joins, which are the cheapest in our system, given that they incur no optimization overhead. We plan to integrate RDRD with other storage backends, which make different assumptions about the join patterns than those which RDF-3X does.
<table>
<thead>
<tr>
<th>Query</th>
<th>RDRD without chaining</th>
<th>RDRD with chaining</th>
<th>Jena</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cold Cache</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QL1</td>
<td>68</td>
<td>25</td>
<td>1003</td>
</tr>
<tr>
<td>QL2</td>
<td>23691</td>
<td>448</td>
<td>n/a</td>
</tr>
<tr>
<td>QL3</td>
<td>234</td>
<td>18</td>
<td>2081</td>
</tr>
<tr>
<td>QL4</td>
<td>60</td>
<td>61</td>
<td>590</td>
</tr>
<tr>
<td>QL5</td>
<td>112</td>
<td>120</td>
<td>1150</td>
</tr>
<tr>
<td>QL6</td>
<td>150</td>
<td>144</td>
<td>1140</td>
</tr>
<tr>
<td>QL7</td>
<td>57</td>
<td>45</td>
<td>580</td>
</tr>
<tr>
<td>QL8</td>
<td>59</td>
<td>59</td>
<td>1398</td>
</tr>
<tr>
<td>QL9</td>
<td>1054</td>
<td>209</td>
<td>15317</td>
</tr>
<tr>
<td>QL10</td>
<td>219</td>
<td>192</td>
<td>1802</td>
</tr>
<tr>
<td>QL11</td>
<td>164</td>
<td>140</td>
<td>820</td>
</tr>
<tr>
<td>QL12</td>
<td>50</td>
<td>94</td>
<td>2108</td>
</tr>
<tr>
<td>QL13</td>
<td>15</td>
<td>15</td>
<td>165</td>
</tr>
<tr>
<td>QL14</td>
<td>53</td>
<td>33</td>
<td>898</td>
</tr>
<tr>
<td>Warm Cache</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QL1</td>
<td>21</td>
<td>3</td>
<td>471</td>
</tr>
<tr>
<td>QL2</td>
<td>22657</td>
<td>374</td>
<td>n/a</td>
</tr>
<tr>
<td>QL3</td>
<td>85</td>
<td>4</td>
<td>870</td>
</tr>
<tr>
<td>QL4</td>
<td>22</td>
<td>23</td>
<td>152</td>
</tr>
<tr>
<td>QL5</td>
<td>13</td>
<td>14</td>
<td>721</td>
</tr>
<tr>
<td>QL6</td>
<td>108</td>
<td>110</td>
<td>667</td>
</tr>
<tr>
<td>QL7</td>
<td>25</td>
<td>16</td>
<td>202</td>
</tr>
<tr>
<td>QL8</td>
<td>37</td>
<td>37</td>
<td>885</td>
</tr>
<tr>
<td>QL9</td>
<td>969</td>
<td>174</td>
<td>13715</td>
</tr>
<tr>
<td>QL10</td>
<td>174</td>
<td>159</td>
<td>971</td>
</tr>
<tr>
<td>QL11</td>
<td>113</td>
<td>92</td>
<td>303</td>
</tr>
<tr>
<td>QL12</td>
<td>35</td>
<td>94</td>
<td>105</td>
</tr>
<tr>
<td>QL13</td>
<td>1</td>
<td>1</td>
<td>38</td>
</tr>
<tr>
<td>QL14</td>
<td>16</td>
<td>23</td>
<td>618</td>
</tr>
</tbody>
</table>

Table 7.8: Comparison to other systems using queries based on LUBM, times in milliseconds
Chapter 7. Experimental Evaluation and Results

Figure 7.9: Comparison to other systems using queries based on LUBM - cold cache

Figure 7.10: Comparison to other systems using queries based on LUBM - warm cache
Chapter 8

Conclusion and Future Work

Rule based query processing allows for easy expression of a very rich class of information needs. This ease comes largely from the fact that rules resemble how we, as humans, think about the world. Rules allow us to express connections between relations, and to define new relations. Among other things, they can be used to overcome the incompleteness of knowledge bases resulting from automated information extraction, which gives us even richer knowledge bases.

Rules are a declarative means of expressing information needs, which requires that any rule-based system be able to efficiently handle this class of queries.

The main source of complexity for a rule-based system is recursion, which moves query processing over rules from the traditional relational setting, where efficient query processing is very well understood, to the more demanding recursive setting. First of all, recursive query processing, in a Datalog setting, is computationally very demanding. Secondly, traditional approaches to query optimization, which highly rely on statistics about relations become much less effective, as relations are expressed intensionally, rather than being readily stored for analysis.

Moreover, the core architectural and systems aspects of relational database systems have been, to a large degree, standardized, and issues involved in their implementation are very well understood. In comparison, rule-based systems with support for recursive query processing, mostly come in the form of research prototypes, which attempt to explore the effects of specific new ideas, often blending ideas from the fields of database systems, logic programming and theorem proving.

In this thesis, we implemented RDRD, a rule-based query processor for disk-resident RDF data with the ability to maintain lineage. It is intended to process queries in a rule-based setting over a large number of extensionally defined facts stored in a disk...
based storage engine, RDF-3X. The motivation for developing RDRD is to serve as the backbone of the URDF framework, which attempts to overcome the issue incompleteness and inconsistency in RDF knowledge bases using a rule-based approach.

We described the implementation of RDRD, and optimizations added to it. We showed how we integrated it with RDF-3X, and what changes we had to make to RDF-3X and the manner in which it processes queries.

We also described a dynamic probabilistic approach to join ordering in a recursive setting, which uses a probabilistic model that relies on statistics from extensional predicates and independence assumptions. The approach, called Least Promising Atom First (LPF), attempts to discover the failure of a recursive subquery as soon as possible, to avoid going into deep recursion, which triggers expensive disk access along the way. We also described chaining of extensional predicates during join ordering, to allow for better utilization of the ability of the storage engine underlying the rule-based query processor to perform joins.

In this thesis, we looked at recursive query processing algorithms, and proposed a fix for the incompleteness in the description of the QSQR algorithm in [AHV95]. It uses the framework used to describe the magic sets algorithm, which aids in the implementation of the algorithm.

Finally, we gave experimental evaluation of our system and join ordering strategy. The results of the experiments have shown that our system performs favorably compared to other similar systems. Experiments have also shown that our join ordering strategy, based on the LPF heuristic performs well while having a low overhead. Particularly, the LPF strategy helped reduce the number of steps required to answer a query. Results also showed that our choice of storage backend did not give the full potential of chaining that we expected, which requires more to be done as we described.

We have learned many lessons from working on RDRD and this thesis, and we believe that there is much work that can still be done on several fronts.

### 8.1 Future Work

**Probabilistic grounding model:** In URDF, weights for facts and rules do not play any rule during grounding. An extension of this would be to define a model for how weights, which correspond to our confidence in base facts and rules, can be integrated into the grounding process. Intuitively, we would have less confidence in a fact reached through a long chain of reasoning than one with a shorter chain. Similarly, we have
higher confidence in a fact reached using rules and base facts with high confidences than
one reached using rules and base facts with low confidences. This, requires defining a
clearer semantics of confidences and how they propagate. It also requires a distinction
between rules. The kind of rules we have seen for URDF, such as

$$isMarriedTo(x, y) \land bornIn(x, z) \xrightarrow{p} bornIn(y, z)$$

where $p$ is the probability of the rule holding, are the ones we would target. In ‘classical
Datalog’ rules, such as

$$ancestor(?x, ?y) \land ancestor(?x, ?z) \rightarrow parent(?z, ?y)$$

the $ancestor$ relation holds equally for all facts reached using this rule, regardless of the
length of the chain of reasoning about an answer. Fuhr’s probabilistic Datalog framework
would be a starting point for this work [Fuh95, Fuh96].

**Ranking and top-k style query processing:** Once a clear model can be reached for
propagating confidences during grounding, it is possible to rank query results in such a
setting. We have seen in Section 2.4.4 that query answering in Datalog is a computa-
tionally demanding problem. One interesting issue to investigate is: given a user who
is interested in answers with a specific range of confidence, how a ranking of results
can guide the grounding process, allowing for the pruning of the search space. Ranking
can also give rise to top-k query processing, which presents a chance for approximating
results in a manner similar to that done in [TWS04].

**Systems and Engineering:** The system produced for this thesis is a prototype, we
believe that there are many improvements we can make to it. One important issue
is how to better integrate a recursive query processor with the pipelining architecture
of the relational database used for extensional facts. This would reduce the need for
materialization of intermediate results, which can be a significant overhead for large
intermediate results. There is much to be learned about engineering such a system
based on experiences from the logic pogramming and automated theorem proving. In
the logics part of the system, it would be interesting to see the data structures and
algorithms underlying logic programming systems and automated theorem provers. For
instance, the data structures used for terms, rules and substituteus are crucial in the
efficiency of the system. Algorithms for unification, substitution application etc. are
also crucial for the system’s efficiency, and are greatly influenced by the data structures
in use.

**Selectivity Estimation and Join Ordering:** In Section 6.1, we proposed a join or-
dering scheme that avoids dealing with sizes of relations, and preferred it over sampling
approaches which can be time consuming. A hybrid approach can combine between our approach and approaches based on size estimates. Initially, no size estimations can be given for intensional relations, which means that we would fully rely on an approach such as LPF. As more queries are answered, we start getting size approximations for intensional relations. As we have better size estimations, we can gradually reduce dependence on an approach such as LPF and resort to size-based ordering approaches.
Appendix A

Queries

A.1 Extensional Queries

RDF-3X:

QE1: select ?y where {Tipper_Gore bornIn ?y}
QE2: select ?x where { ?x hasCurrency United_States_dollar}
QE3: select ?y ?z where {Al_Gore isMarriedTo ?y . ?y bornIn ?z}
                           ?x hasCurrency United_States_dollar}
QE5: select ?x ?y where {?x isLeaderOf ?y . ?x bornIn ?y}
                              Arnold_Schwarzenegger actedIn ?y . ?x actedIn ?y }
                           ?y bornIn ?z}
QE8: select ?x ?y ?z where {?x bornIn Ulm . ?x diedIn ?y .
                           ?x hasWonPrize ?z}

Reasoner:

QE1: ? ← bornIn(Tipper_Gore, ?y);
QE2: ? ← hasCurrency(?x, United_States_dollar);
QE3: ? ← isMarriedTo(Al_Gore, ?y), bornIn(?y, ?z);
QE4: ? ← hasOfficialLanguage(?x, Portuguese_language),
                           hasCurrency(?x, United_States_dollar);
QE5: ? ← isLeaderOf(?x, ?y), bornIn(?x, ?y);
QE6: ? ← actedIn(Arnold_Schwarzenegger, ?y), actedIn(?x, ?y), bornIn(?x, ?z);
QE7: ? ← hasAcademicAdvisor(?x, ?y), bornIn(?x, ?z), bornIn(?y, ?z);
QE8: ? ← diedIn(?x, ?y), hasWonPrize(?x, ?z), bornIn(?x, Ulm);
A.2 Recursive Queries

A.2.1 Yago

Rules

YR1:  \( \text{diedIn}(?x, ?y) \leftarrow \text{bornIn}(?x, ?y), \text{livesIn}(?x, ?y) \);

YR2:  \( \text{bornIn}(?x, ?z) \leftarrow \text{isCitizenOf}(?x, ?y), \text{locatedIn}(?z, ?y), \text{livesIn}(?x, ?z) \);

YR3:  \( \text{bornIn}(?x, ?z) \leftarrow \text{isMarriedTo}(?x, ?y), \text{bornIn}(?y, ?z) \);

YR4:  \( \text{bornIn}(?c, ?y) \leftarrow \text{livesIn}(?x, ?y), \text{livesIn}(?z, ?y), \text{isMarriedTo}(?x, ?z), \text{hasChild}(?x, ?c), \text{hasChild}(?z, ?c) \);

YR5:  \( \text{bornIn}(?a, ?c) \leftarrow \text{isCitizenOf}(?a, ?b), \text{locatedIn}(?b, ?c) \);

YR6:  \( \text{graduatedFrom}(?z, ?a) \leftarrow \text{hasAcademicAdvisor}(?x, ?y), \text{hasAcademicAdvisor}(?z, ?y), \text{graduatedFrom}(?x, ?a) \);

YR7:  \( \text{graduatedFrom}(?a, ?c) \leftarrow \text{worksAt}(?b, ?c), \text{hasAcademicAdvisor}(?a, ?b) \);

YR8:  \( \text{livesIn}(?y, ?z) \leftarrow \text{isMarriedTo}(?x, ?y), \text{livesIn}(?x, ?z) \);

YR9:  \( \text{livesIn}(?y, ?z) \leftarrow \text{isMarriedTo}(?x, ?y), \text{bornIn}(?x, ?z), \text{bornIn}(?y, ?z) \);

YR10: \( \text{isMarriedTo}(?x, ?y) \leftarrow \text{hasChild}(?x, ?z), \text{hasChild}(?y, ?z), \text{NOTEQUALS}(?x, ?y) \);

YR11: \( \text{isMarriedTo}(?x, ?z) \leftarrow \text{actedIn}(?x, ?y), \text{actedIn}(?z, ?y), \text{NOTEQUALS}(?x, ?z) \);

YR12: \( \text{ancestor}(?x, ?y) \leftarrow \text{hasChild}(?x, ?y) \);

YR13: \( \text{ancestor}(?x, ?y) \leftarrow \text{ancestor}(?x, ?z), \text{ancestor}(?z, ?y) \);
Appendix A. Queries

Queries

Q1: ? ← bornIn(Al_Gore, ?x);
   YR2-YR5, YR8-YR11

Q2: ? ← isMarriedTo(Woody_Allen, ?x);
   YR10-YR11

Q3: ? ← actedIn(?x, Total_Recall), bornIn(?x, Thal, _Austria);
   YR2-YR5, YR8-YR11

Q4: ? ← actedIn(Arnold_Schwarzenegger, ?y), actedIn(?x, ?y), bornIn(?x, ?z),
   NOTEQUALS(?x, Arnold_Schwarzenegger);
   YR2-YR5, YR8-YR11

Q5: ? ← bornIn(?x, Oxford), graduatedFrom(?x, ?y), hasAcademicAdvisor(?x, ?z),
   graduatedFrom(?z, University_of_Cambridge);
   YR2 - YR10

Q6: ? ← bornIn(?x, Paris), isCitizenOf(?x, ?y), locatedIn(Paris, ?z),
   NOTEQUALS(?y, ?z);
   YR2-YR5, YR8-YR11

Q7: ? ← actedIn(?x, Total_Recall), bornIn(?x, ?y), livesIn(?x, ?z),
   bornOnDate(?x, ?a), isMarriedTo(?x, ?b);
   YR2-YR5, YR8-YR11

Q8: ? ← hasWonPrize(?x, Nobel_Prize_in_Physics), bornIn(?x, Ulm),
   bornOnDate(?x, ?y);
   YR2-YR5, YR8-YR10

Q9: ? ← isMarriedTo(Emma_Thompson, ?x), actedIn(Emma_Thompson, ?y),
   actedIn(?x, ?y);
   YR2-YR5, YR8-YR11

Q10: ? ← directed(Martin_Scorsese, ?x), actedIn(?y, ?x), actedIn(?z, ?x),
     NOTEQUALS(?y, ?z);

Q11: ? ← diedIn(Ingrid_of_Sweden, ?x);
     YR1-YR5, YR8-YR10

Q12: ? ← ancestor(?x, Charles,_Prince_of_Wales);
     YR12 - YR13

Q13: ? ← ancestor(Earl_IV_of_England, ?x);
     YR12 - YR13
A.2.2 LUBM

Rules

LR1:  \( \text{type}(?x, \text{Professor}) \leftarrow \text{type}(?x, \text{FullProfessor}); \)
LR2:  \( \text{type}(?x, \text{Professor}) \leftarrow \text{type}(?x, \text{AssociateProfessor}); \)
LR3:  \( \text{type}(?x, \text{Professor}) \leftarrow \text{type}(?x, \text{AssistantProfessor}); \)
LR4:  \( \text{type}(?x, \text{Student}) \leftarrow \text{type}(?x, \text{GraduateStudent}); \)
LR5:  \( \text{type}(?x, \text{Student}) \leftarrow \text{type}(?x, \text{UndergraduateStudent}); \)
LR6:  \( \text{subOrganizationOf}(?x, ?z) \leftarrow \text{subOrganizationOf}(?x, ?y), \text{subOrganizationOf}(?y, ?z); \)
LR7:  \( \text{degreeFrom}(?x, ?y) \leftarrow \text{doctoralDegreeFrom}(?x, ?y); \)
LR8:  \( \text{degreeFrom}(?x, ?y) \leftarrow \text{mastersDegreeFrom}(?x, ?y); \)
LR9:  \( \text{degreeFrom}(?x, ?y) \leftarrow \text{undergraduateDegreeFrom}(?x, ?y); \)
LR10: \( \text{hasAlumnus}(?x, ?y) \leftarrow \text{degreeFrom}(?y, ?x); \)
LR11: \( \text{chairOf}(?x, ?y) \leftarrow \text{headOf}(?x, ?y); \)
LR12: \( \text{allMemberOf}(?x, ?y) \leftarrow \text{memberOf}(?x, ?y); \)
LR13: \( \text{allMemberOf}(?x, ?y) \leftarrow \text{worksFor}(?x, ?y); \)
Queries

LQ1:  ? ← type(?x, GraduateStudent),
      takesCourse(?x, http://www.Department0.University0.edu/GraduateCourse0);
      -
LQ2:  ? ← type(?x, GraduateStudent), type(?y, University),
      type(?z, Department), memberOf(?x, ?z), subOrganizationOf(?z, ?y);

LR6
LQ3:  ? ← type(?x, Publication),
      publicationAuthor(?x, http://www.Department0.University0.edu/AssistantProfessor0);
      -
LQ4:  ? ← type(?x, Professor),
      worksFor(?x, http://www.Department0.University0.edu), name(?x, ?l),
      emailAddress(?x, ?m), telephone(?x, ?n);

LR1 - LR3
LQ5:  ? ← allMemberOf(?x, http://www.Department0.University0.edu);

YR12 - YR13
LQ6:  ? ← type(?x, Student);

LR4 - LR5
LQ7:  ? ← type(?y, Course),
      teacherOf(http://www.Department0.University0.edu/AssociateProfessor0, ?y),
      takesCourse(?x, ?y), type(?x, Student);

LR4 - LR5
LQ8:  ? ← memberOf(?x, ?y), subOrganizationOf(?y, http://www.University0.edu),
      emailAddress(?x, ?z), a(?x, Student);

LR4 - LR6
LQ9:  ? ← advisor(?x, ?y), type(?z, Course), takesCourse(?x, ?z),
      teacherOf(?y, ?z), type(?x, Student);

LR4 - LR5
LQ10: ? ← type(?x, Student),
      takesCourse(?x, http://www.Department0.University0.edu/GraduateCourse0);

LR4 - LR5
LQ11: ? ← type(?x, ResearchGroup),
      subOrganizationOf(?x, http://www.University0.edu);

LR6
LQ12: \( ? \leftarrow \text{chairOf}(?,?d), \text{type}(?,\text{Department}), \text{worksFor}(?,?), \text{subOrganizationOf}(?,\text{http://www.University0.edu}); \)

LR6, LR11

LQ13: \( ? \leftarrow \text{hasAlumnus}(\text{http://www.University0.edu},?x); \)

LR7 - LR10

LQ14: \( ? \leftarrow \text{type}(?,\text{UndergraduateStudent}); \)

-
Bibliography


