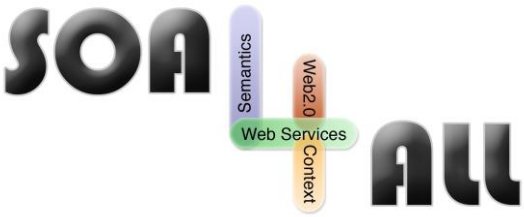


Project Number: **215219**

Project Acronym: **SOA4All**

Project Title: **Service Oriented Architectures for All**

Instrument: **Integrated Project**

Thematic Priority: **Information and Communication Technologies**

D3.2.8 Enhanced Reasoning Framework Core

Activity N: 2	Fundamental and Integration Activities	
Work Package: 3	Service Annotation and Reasoning	
Due Date:		28/02/2011
Submission Date:		28/02/2011
Start Date of Project:		01/03/2008
Duration of Project:		36 Months
Organisation Responsible of Deliverable:		UIBK
Revision:		1.0
Author(s):	Adrian Marte, UIBK	
Reviewers:	Barry Norton, KIT Mateusz Radzimski, ATOS	

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**x**
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	15.01.2011	First draft	Adrian Marte
0.2	08.02.2011	Implementation report, Evaluation	Adrian Marte
0.3	09.02.2011	Internal peer review	Daniel Winkler
0.4	11.02.2011	Corrections after internal peer review	Adrian Marte
0.4	15.02.2011	Internal peer review	Reto Krummenacher
0.5	15.02.2011	Corrections after internal peer review	Adrian Marte
0.6	18.02.2011	Peer review	Mateusz Radzimski
0.7	21.02.2011	Corrections after review	Adrian Marte
0.8	25.02.2011	Peer review	Barry Norton
1.0	28.02.2011	Corrections after review	Adrian Marte

Table of Contents

List of Figures

List of Tables

List of Listings

Glossary of Acronyms

Acronym	Definition
D	Deliverable
EC	European Commission
WP	Work Package
RIF	Rule Interchange Format
BLD	Basic Logic Dialect
DTB	Datatypes and Built-ins
EDB	Extensional Database
IDB	Intensional Database
RHS	Right-hand side
LHS	Left-hand side
RDB	Relational Database
POM	Project Object Model
WSML	Web Service Modeling Language
LUBM	Lehigh University Benchmark
DBLP	Digital Bibliography & Library Project

Executive Summary

The Datalog reasoner IRIS is the main underlying engine used by the WSML2Reasoner framework for reasoning with WSML-Core, WSML-Flight, WSML-Rule and also for WSML-DL language variants. Recently development started on making the IRIS engine a fully compatible – or at least a maximally compatible – RIF-BLD system.

In IRIS the evaluation of Datalog programs is handled only in-memory. As RIF especially targets Web applicability where the amount of data is extremely large, the IRIS reasoner has been extended in order to support data that exceeds the limits of a computer's memory.

This deliverable describes IRIS-RDB, an extension of IRIS, that overcomes this limitation by using a relational database as an underlying system to evaluate Datalog programs. The focus of the deliverable is to compare the original IRIS implementation to IRIS-RDB with respect to the supported features, the performance and the scalability of the software component.

1. Introduction

The Datalog reasoner IRIS is the main underlying engine used by the WSML2Reasoner framework for reasoning with WSML-Core, WSML-Flight, WSML-Rule and also for WSML-DL language variants. Recently, development started on making the IRIS engine a fully compatible – or at least a maximal compatible – RIF-BLD system. RIF-BLD is the Basic Logic Dialect (BLD) of the Rule Interchange Format (RIF), a W3C recommendation for a standard for exchanging rules among rule systems, in particular among Web rule engines.

RIF-BLD[1] extends RIF-Core[2], a common subset of RDF-based rule dialects that supports a rich set of XML Schema and RDF datatypes and XPth built-ins defined in RIF-DTB[3]. RIF-BLD adds features to RIF-Core that are not directly available, such as logic functions, equality in the rule head (rule head equality) and named arguments. Furthermore, it defines the concept of logical entailment, i.e. what it means for a set of RIF-BLD rules to entail another RIF-BLD formula.

IRIS is an extensible reasoning engine for expressive rule-based languages. It supports (un-) safe Datalog with (locally) stratified or well-founded “negation as failure”, function symbols, equality in the rule head and a comprehensive and extensible set of built-ins and datatypes [6]. In IRIS the evaluation of Datalog queries, i.e., the evaluation of queries over a knowledge base, where a knowledge base consists of facts and rules, is only handled in memory. As RIF especially targets application to the Web where the amount of data is extremely large, the IRIS reasoner needs to be extended in order to support data that exceeds the limits of a single computer’s memory.

This deliverable describes the development of IRIS-RDB, an extension of IRIS that uses a relational database as an underlying system to evaluate stratified and recursive Datalog programs using relation algebra.

1.1 Structure of the Document

Section 2 of the deliverable describes the software in detail, where Section 2.4 focuses on the description of the transformation from Datalog programs to relational algebra. Section 3 shows how to install and use the system for evaluating Datalog programs. An evaluation of the software component is provided in Section 4. Section 5 concludes with a short summary of the deliverable.

1.2 Purpose and Scope

This document is a report on the extension of the IRIS reasoner called IRIS-RDB. The object of the report is to give information about the implementation and to show how to use the software for the evaluation of Datalog programs. Further, it defines in an abstract way a transformation from Datalog to relational algebra.

The target audiences of this report are mainly developers who wish to integrate the IRIS-RDB reasoning system into their software components to evaluate Datalog programs, and others who want to understand how the system evaluates Datalog programs using a relational database system.

---

[1] RIF Basic Logic Dialect, http://www.w3.org/TR/2010/REC-rif-bld-20100622/ [last checked 25.01.2011]

[2] RIF Core, http://www.w3.org/TR/2010/REC-rif-core-20100622/ [last checked 25.01.2011]

[3] RIF Datatypes and Built-ins 1.0, http://www.w3.org/TR/2010/REC-rif-dtb-20100622/ [last checked 20.01.2001]

2. Software Description

IRIS-RDB has been developed with the goal to have a more scalable reasoning engine that is able to process knowledge bases that exceed the limits of a single computer's memory. This has been accomplished by exploiting the close relationship of Datalog and relational algebra and, thus, by implementing an evaluation strategy based on a relational database system.

2.1 Datalog

According to [1], Datalog is in many respects a simplified version of general Logic Programming. A logic program consists of a finite set of facts and rules. In Datalog both facts and rules are represented as Horn clauses of the general shape

$$L_0 \text{ :- } L_1, \ldots, L_n$$

where each L_i is a literal. A literal is either an atomic formula or a negated atomic formula, where an atomic formula is of the form $p_i(t_1, \ldots, t_{ki})$ such that p_i is a predicate symbol and the t_j are terms. A non-negated atomic formula is also referred to as a positive literal, whereas a negated atomic formula is called a negative literal. A term is either a constant, variable or, in the case of function-aware systems, a function symbol. In IRIS the constant symbols are represented by data values of the set of XML Schema, RDF (rdf:PlainLiteral and rdf:XMLLiteral) and RIF (rif:local and rif:iri) datatypes. The left-hand side (LHS) of a clause is called the rule head, whereas the right-hand side (RHS) is called the rule body. Clauses with an empty body represent facts, and clauses with at least one literal in the body represent rules. A literal or atomic formula which does not contain any variables is called ground. In the following we may also refer to literals in the rule body as subgoals.

In Datalog there are three types of predicates, Extensional Database (EDB), Intensional Database (IDB) and built-in predicates. The predicates denoting the ground facts are EDB predicates, while the ones defined by rules are IDB predicates. Built-in predicates are expressed by special predicate symbols such as <, >, or = with a predefined meaning. Arithmetic built-in predicates can be written in infix notation, e.g. `?X<?Y` rather than `<(?X, ?Y)`, whereas other built-ins are written like `rif-pred:is-literal(?X)`. Each non-built-in predicate in a Datalog program is either an EDB predicate or an IDB predicate, but not both. We further assume, that each predicate symbol is associated with a particular number of arguments that it takes, and we may denote that number as the arity of the predicate.

2.2 Features

IRIS-RDB is an extension of the IRIS reasoner that uses the database engine H2[4] as an underlying relational database system to evaluate Datalog programs.

H2 is an open-source relational database implemented in Java. It is a very fast[5] and feature-rich[6] system that supports persistent and in-memory storage and has both an embedded and a server mode. H2 is dual licensed under a modified version of the MPL 1.1 (Mozilla Public License) and under the (unmodified) EPL 1.0 (Eclipse Public License). Since IRIS-RDB uses SQL as query language it is, in principle, easy to align the system such that it is possible to use it with a different relational database engine than H2.

[4] H2 Database Engine, http://www.h2database.com [last checked 19.01.2011]

[5] H2 Performance, http://www.h2database.com/html/performance.html [last checked 19.01.2011]

[6] H2 Features, http://www.h2database.com/html/features.html [last checked 19.01.2011]

IRIS-RDB can evaluate safe or unsafe Datalog (without function symbols) and equality in rule heads, supports XML, RDF, and RIF data types, built-in predicates (those supported by IRIS) and (locally) stratified negation as failure, see Table 1.

IRIS is a highly modular system, that provides well-defined Java interfaces which allow the implementation and integration of additional evaluation strategies or storage mechanisms. IRIS-RDB makes heavy use of these interfaces and provides implementations and extensions of such where possible, which allows for the seamless integration of the extension into the IRIS code base.

Table 1: Comparison of IRIS and IRIS-RDB features.

Feature	IRIS	IRIS-RDB
Unsafe rules	Yes	Yes
Globally unstratified programs	Yes	No
Locally unstratified program	Yes	Yes
Function symbols	Yes	No
RIF list terms	Yes	No
RIF-BLD datatypes	Yes	Yes
RIF-BLD built-ins	Yes	Yes
Rule head equality	Yes	Yes

2.2.1 Supported Datatypes

IRIS, and also IRIS-RDB, support all types defined in RIF-DTB, in particular all XML Schema 1.1, RDF (`rdf:XMLLiteral` and `rdf:PlainLiteral`) and RIF (`rif:iri` and `rif:local`) datatypes.

2.2.2 Built-in Predicates

IRIS comes with a rich set of built-in predicates that can be used in the bodies of rules, both in a positive and in a negative literal. They include:

- Equality, inequality, assignment, and unification.
- Addition, subtraction, multiplication, division and modulus.
- All built-ins defined in RIF-DTB, including arithmetical built-ins, guard predicates for datatypes, built-ins for datatype conversion and casting and special functions and predicates on various RDF and XML Schema datatypes.

IRIS-RDB uses the built-in infrastructure of the original IRIS and, therefore, takes advantage of all the built-ins mentioned above.

2.2.3 Rule Head Equality

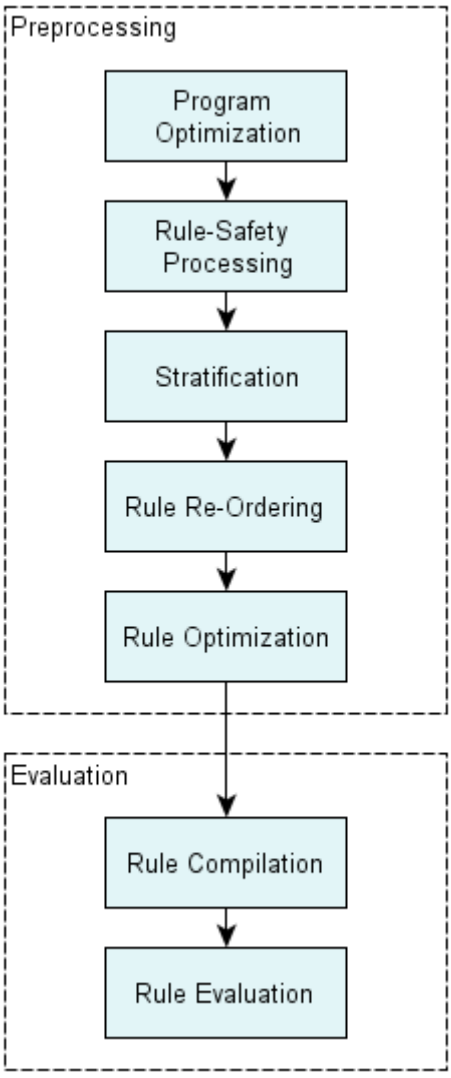
In order to support rules with equality in the head, IRIS-RDB uses the rewriting technique defined in [2, Section 4.1] where additional rules are created for each predicate occurring in the Datalog program in order to resolve equivalent terms.

2.3 Evaluation Process

IRIS-RDB evaluates queries over sets of facts (ground atomic formulas) and rules, which together we call a *knowledge base*. A knowledge base can be created directly via the Java API or can be parsed from a Datalog program in textual form using the parser provided by IRIS. For each query that is evaluated over the knowledge base, IRIS-RDB returns the set of tuples that can be found or inferred from the knowledge base that satisfy the query.

IRIS-RDB supports semi-naive bottom-up evaluation using a (locally) stratified technique, see Annex A. The implementation of this evaluation strategy is based on the original semi-naive implementation of IRIS and makes heavy use of the involved interfaces and classes. See Figure 1 for a depiction of the steps involved in the process of evaluation.

Figure 1: Stratified evaluation strategy



2.3.1 Program Optimization

The Magic Sets optimization technique [7] re-writes the rule-set according to the query so that only tuples likely to be involved in satisfying the query are computed. The disadvantage of this approach is that a new sub-set of the model must be computed for each new query. Therefore, Magic Sets allows faster knowledge-base initialization times at the expense of

longer query times. IRIS can be configured programmatically whether to use the Magic Sets optimization or not.

Another simpler program optimization technique is rule-filtering that simply removes those rules that cannot influence the query result, thus reducing the size of the minimal model computation. This technique is usually used in combination with Magic Sets.

2.3.2 Rule-Safety Processing

The algorithm for detecting unsafe rules was used from the original IRIS implementation, which is based on the algorithm and the definition of unsafe rules defined in [1, page 105]. According to this definition, a rule is safe if all its variables are limited, where limited variables are defined as follows:

1. Any variable that appears as an argument in an ordinary predicate of the body is limited.
2. Any variable `X` that appears in a subgoal `X=a` or `a=X`, where `a` is a constant, is limited.
3. Variable `X` is limited if it appears in a subgoal `X=Y` or `Y=X`, where `Y` is a variable already known to be limited.

In order to support unsafe rules, IRIS provides an *augmenting rule processor*, which is based on the technique suggested by [3] that adds a "universe" predicate for each unbound variable to the body of the rule. This "universe" predicate contains all constants appearing in the input program or that are created during the evaluation of the program, see also Section 2.4.1. For instance, consider rule

```
q(?X) :- not p(?X).
```

which unsafeness is directly visible, as variable X is not limited, as it does not appear in any non-negated ordinary predicate, nor is it equated with a constant or a variable known to be limited. However, using the aforementioned technique the rule can be made safe by adding a universe predicate and, thus, limiting the variable X, such that the new rule looks like

```
q(?X) :- universe(?X), not p(?X).
```

2.3.3 Stratification

IRIS has the concepts of globally and locally stratified logic programs. A globally stratified program is one where all rules can be grouped into strata using, for instance, the algorithm defined in [1, page 133]. This algorithm computes a stratification of the rules of a program. It groups the predicates into strata, which are the largest sets of predicates, such that:

1. If a predicate `p` is the head of a rule with a subgoal that is a negated `q`, then `q` is in a lower stratum than `p`.
2. If predicate `p` is the head of a rule with a subgoal that is a non-negated `q`, then the stratum of `p` is at least as high as the stratum of `q`.

Given stratified predicates we can also group the rules into strata, by assigning rules `r` to stratum `i`, where `i` is the stratum assigned to the head predicate of `r`. A positive side effect of this stratification is that the strata give an order in which the rules should be evaluated, as all rules in each stratum can be evaluated before the rules of the higher stratum. If no stratification of the rules can be computed, the program is not globally stratified [1, p. 134].

Local stratification is needed when the head predicate of a rule has a negative direct or indirect dependency on itself, but the presence of constants allow the separation of the domain of tuples used as input to the rule and the domain of tuples produced by the rule [6, page 7]. For instance, the following rule appears to be unstratified:

```
p(2, ?X) :- q(?X), not p(3, ?X).
```

because the head predicate of the rule has a direct negative dependency on itself. However, as the rule can only produce tuples whose first term value is `2` and can only use input tuples whose first term is `3`, there exists no (local) recursive dependency and the rule can be evaluated normally.

2.3.4 Rule Re-Ordering

After stratification of a program, the performance of the evaluation can be further improved, by changing the order in which the rules are evaluated. IRIS provides a rule optimizer, which re-orders the rules in a way such that those rules that produce tuples that feed the other rule bodies, are evaluated earlier. For example, the rule

```
p(?X, ?Z) :- r(?X, ?Y), s(?Y, ?Z).
```

is evaluated before the rule

```
q(?X, ?Y) :- p(?X, ?Y), t(?X).
```

as the tuples generated by the first rule can immediately used when evaluating the second rule, which reduces the number of runs required by the semi-naive evaluation algorithm.

2.3.5 Rule Optimization

IRIS provides further optimization techniques, which optimize the evaluation on a per rule basis. The supported optimizers are listed below.

- **Join condition:** This optimizer reduces the number of equality relations by substituting the occurrences of variables `Y` of a built-in predicate `X=Y` with the variable `X`, e.g.:

  ```
  p(?X) :- q(?X), r(?Y), ?X = ?Y.
  ```

  would be changed to

  ```
  p(?X) :- q(?X), r(?X).
  ```

  This can significantly reduce the number of intermediate tuples produced during a sequence of Cartesian products. In the case of IRIS-RDB it also improves performance, as in most cases the join will be handled implicitly by the database system, instead of the IRIS built-in for equality, see Section 2.3.6 for more information on evaluating rules with built-in predicates.

- **Replace variables with constants:** Similar to the above optimization, this optimization reduces the number of equality relations, by substituting the occurrences of variables `X` of a built-in predicate `X=a` with `a`, where `a` is a constant. For instance,

  ```
  p(?X) :- q(?X, ?Y), ?Y = 2.
  ```

  would be changed to

  ```
  p(?X) :- q(?X, 2).
  ```

- **Re-order literals:** This optimization re-arranges the literals in a rule body, such that the most restrictive literals appear first. The preferred order is: positive literals with no variables, built-ins with no variables, positive literals, built-ins and negated literals. Negated literals and built-ins can be pushed earlier into the rule body as soon as all their variables are bound.

- **Remove duplicate literals:** In order to avoid unnecessary joins, this optimizer removes any literal in a rule body that appears twice with the same variables or constants.

2.3.6 Rule Compilation

In the original IRIS a rule is transformed into a compiled rule that gets evaluated by a rule evaluator. The compiler inspects each body literal and creates a *view* on the corresponding relation that filters the tuples of the relation according to the view criteria given by the arguments of the literal. For instance, for the literal `q(?X, ?X)` the compiler creates a view on the relation of `q`, where only those tuple are returned, where both terms are equal. In the next step, the compiler looks for all matching variables between two adjacent views, calculates the join indices and creates indices. For built-in predicates, as there is usually no relation associated with a built-in, the compiler uses the corresponding implementation of the built-in for evaluation.

In IRIS-RDB, the compilation of a rule is performed similarly. However, instead of creating dedicated objects that take care of filter or joining relations, the various operations are represented by relational algebra operations. The rule compiler of IRIS-RDB creates a relation for the rule body as described in the algorithm in Section 2.4.2. In principle, the compiler creates a relational view for each intermediate `A` computed in the process of the rule compilation. The final `A` is then the relation/view representing the relation of the rule body, and can eventually be used to project the values into the head of the rule, see Section 2.4.3.

Since IRIS supports a rich set of built-ins and provides means to easily implement further built-ins, we have decided to use this infrastructure to evaluate built-in predicates, rather than having only a restricted set of built-ins given by the underlying database system. However, this approach comes with the cost of reduced performance when evaluating rules with built-ins, as the tuples may then only be processed one-by-one, which might be quite inefficient compared to the set-oriented methods used by a relational database system.

2.3.7 Rule Evaluation

The original IRIS supports two rule evaluation techniques, the “naive” and the “semi-naive” evaluator. The “naive” [1, page 119] evaluator simply applies all facts to all rules in each round of evaluation and stops when no new facts are computed. The “semi-naive” [1, page 127] evaluator is an extension of the “naive” algorithm that takes advantage of incremental relations and tries to avoid computing tuples that have been computed before.

IRIS-RDB provides an implementation of the “semi-naive” algorithm defined in Annex A.

2.4 Translation of Datalog Programs into Relational Algebra

The following three sections describe the transformation of Datalog rules to relational algebra and the corresponding SQL expressions.

2.4.1 The Relation of a Predicate

Ground facts are intended to be stored in a relational database, therefore we assume that each corresponding EDB-predicate `r` corresponds to exactly one relation `R` in the database, such that each fact `r(c1, ..., ck)` is stored as a tuple `(c1, ..., ck)` of `R`. In general, IDB predicates correspond to relational views and are not stored explicitly. However, in our implementation we also have for each IDB predicate `s` occurring in a program a corresponding relation `S` in the database. Although this can result in reduced performance, this approach allows recursive programs to be computed by simple non-recursive relational expressions. This clearly avoids problems with the underlying database systems, as only a few database systems support recursive SQL and those which do usually need some kind of termination argument, such as an integer that is increased in each recursive step and determines the number of maximal recursive steps.

2.4.1.1 The Universe Relation

In order to support the method implemented in IRIS for processing unsafe rules, a special predicate and corresponding relation is used to store and retrieve the constant symbols appearing in and generated by the program. This *universe* relation stores the string representation of the *common* and *canonical* value and the *datatype* URI of a constant term. The *common* value is the lexical representation of the constant term casted to the most general type of that constant term. For instance, the most general type of numeric terms is `xsd:decimal`, as all numerical values can be represented as a decimal value. The *canonical* value is the canonical, lexical representation of the data value (as defined by the specification of such). For example, the lexical representation of a `xsd:duration` of 1 month is "P1M". The *datatype* URI is the absolute URI of the data type of the constant term. For instance, the data type URI of `xsd:duration` is `http://www.w3.org/2001/XMLSchema#duration`.

The schema of the *universe* relation is depicted in Table 2, where "id" is an auto-increasing integer representing the primary key, "common" is the column storing the *common* value, "canonical" is the column storing the *canonical* value, and "type" is the column storing the *datatype* URI of the constant term, each in a string representation. In order to enable equality checking and to reduce redundancy, a unique index is created on the columns "canonical" and "type", which ensures that there are no two terms of the same type with the same canonical value.

Table 2: Schema of the universe relation

Universe			
id: INT	common: VARCHAR	canonical: VARCHAR	type: VARCHAR
486	1337.0	1337	http://www.w3.org/2001/XMLSchema#int
...	...	...	...

2.4.1.2 The Relation for a Predicate

The relations corresponding to a predicate are created according to the following schema: a predicate `p` with arity `n` is associated with a relation `P` in the database, that has `n` columns, where the column for the first term has the identifier `attr1`, for the second `attr2` and so on. The value of each column is the foreign key (ID in the universe relation) of the tuple in the universe relation corresponding to the term.

This enables us to use two different implementations of joins. The first solution – which is actually the one currently used by IRIS-RDB – joins two tables on the integers values of the attribute columns, which means that those tuples are joined, where the attribute at the specified position has the same canonical value and data type URI. The second method uses the normalized string values of the attributes that is being joined on. This however would require additional joins to the universe relation when joining two relations and, therefore, may significantly reduce performance.

2.4.2 The Relation Defined by a Rule Body

A rule may be evaluated by computing a relation for the rule body using relational algebra operations. We have implemented a modified version of the algorithm defined in [1, page

109]. This version of the algorithm requires the literals of the rule body to be reordered such that all but the allowed unbound variables of a built-in are already bound by a preceding literal. The symbols π and σ refer to the standard relational operations *projection* and *selection*, respectively, as defined in [1, page 56].

Input: The body of a Datalog rule r, which consists of subgoals $S_1, \ldots, S_n$ containing variables $X_1, \ldots, X_m$, where each variable appears only once in the list. For each $S_i=p_i(c_{i1}, c_{iki})$ with a non-built-in predicate, there is a relation R_i already computed, where the c's are arguments, either variables or constants.

Output: An expression of relational algebra, which we call

$$\text{EVAL-RULE } (r, R_1, \ldots, R_n)$$

that computes from the relations $R_1, \ldots, R_n$ a relation $R(c_1, \ldots, c_k)$ with all and only the tuples $(a_1, \ldots, a_k)$ such that, when we substitute a_j for each c_j, where c_j is a variable and $1 <= j <= k$, all the subgoals $S_1, \ldots, S_n$ are made true.

Method: The expression is constructed by the following steps.

A := ∅

E := ∅

O := empty tuple

for i := 1 to n **do**

if S_i is positive **then**

if p_i is not a built-in predicate **then**

Let T be the expression $\sigma_F(R_i)$. F is the conjunction (logical AND) of the following conditions:

1. If position k of S_i has constant a, then F has the term $k=a
2. If position k and l of S_i both contain the same variable, then F has the term $k=$l

if A != ∅ **then**

Let $Y_1, \ldots, Y_r$ be the variables occuring in O, where each variable appears only once in the list.

Let G be the conjunction (logical AND) of the following conditions:

1. If some Y is also in $X_1, \ldots, X_m$ and let a be the position of Y in O and b be the position of Y in (c_{i1}, c_{iki}) then G has the term $a=$b

If G is not empty then let A be the expression ΩF(A x T) otherwise let A be the expression (A x T).

else

Let A be T.

fi

else

Let Q be a new relation in the database with arity c + d, where c is the arity of A and d is the arity of the built-in predicate. Here A is not empty, as the rule is safe.

Let $Y_1, \ldots, Y_r$ be the variables occuring in O, where each variable appears only once in the list.

Let F be the conjunction (logical AND) of the following conditions:

1. If some Y is also in $X_1, \ldots, X_m$ and let a be the position of Y in O and b be the position of Y in $(c_{i1}, \ldots, c_{iki})$, then F has the term $a=$b

Let T be $\sigma_F(A)$ and and let $(a_1, \ldots, a_c)$ be the tuples from T and let $(b_1, \ldots, b_d)$ be the tuples from the result of the evaluation of the built-in where some of $b_1, \ldots, b_d$ are the input (extracted from $(a_1, \ldots, a_c)$) and some are the output terms of the built-in. Add all tuples $(a_1, \ldots, a_c, b_1, \ldots, b_d)$ to Q for which $(b_1, \ldots, b_d)$ the built-in holds.

Let A be Q.

fi

Let O be the concatenation of O and $(c_{i1}, \ldots, c_{iki})$.

else

if p_i is not a built-in predicate **then**

Let T be the expression $\sigma_F(R_i)$, where F is the conjunction (logical AND) of the following conditions:

1. If position k of S_i has constant a, then F has the term $k!=a
2. If position k and l of S_i both contain the same variable, then F has the term $k!=$l

if A != ∅ **then**

Let $Y_1, \ldots, Y_r$ be the variables occurring in O, where each variable appears only once in the list.

Let G be the conjunction (logical AND) of the following conditions:

1. If some Y is also in $X_1, \ldots, X_m$ and let a be the position of Y in $X_1, \ldots, X_m$ and b be the position of Y in $Y_1, \ldots, Y_r$ then G has the term $a!=$b

Let A be the expression $\pi_{1\ldots r}\sigma_G(A \times T)$

else

Let A be T.

fi

else

Let Q be a new relation in the database with arity c, where c is the arity of A.

Let $Y_1, \ldots, Y_r$ be the variables occurring in O, where each variable appears only once in the list.

Let F be the conjunction (logical AND) of the following conditions:

1. If some Y is also in $X_1, \ldots, X_m$ and let a be the position of Y in O and b be the position of Y in $(c_{i1}, \ldots, c_{iki})$, then F has the term $\$a=\$b$

Let T be $\sigma_F(A)$ and and let $(a_1, \ldots, a_c)$ be the tuples from T and let $(b_1, \ldots, b_d)$ be the tuples from the result of the evaluation of the built-in where some of $b_1, \ldots, b_d$ are the input (extracted from $(a_1, \ldots, a_c)$) and some are the output terms of the built-in. Add all tuples $(a_1, \ldots, a_c)$ to Q for which the built-in does not hold.

Let A be Q.

fi

fi

Let E be A.

end

2.4.3 The Relational Views for a Rule

In principle, the system uses SQL to create relational views for each intermediate A and the final E computed by the algorithm defined in Section 2.4.2. For instance, for the rule body of the rule

```
p(?X, ?Y) :- q(?X, ?X), r(?X, 'a', ?Y).
```

a relational view is created with the SQL expression

```
CREATE VIEW body(attr1, attr2, attr3, attr4, attr5) AS
    SELECT left.attr1 AS attr1, left.attr2 AS attr2,
           right.attr1 AS attr3, right.attr2 AS attr4,
           right.attr3 AS attr5
        FROM q_filter AS left, r_filter AS right
        WHERE left.attr2 = right.attr1
```

where `q_filter` and `r_filter` are relational views created by the SQL expressions

```
CREATE VIEW q_filter(attr1, attr2) AS
    SELECT attr1, attr2 FROM relation_for_q WHERE attr1 = attr2
```

```
CREATE VIEW r_filter(attr1, attr2, attr3) AS
    SELECT attr1, attr2, attr3
        FROM relation_for_r
        WHERE attr2 = '234'
```

where `234` is the ID of the tuple in the universe relation corresponding to the constant `a`.

In a final step, the relation of the body is projected into the rule head by, again, creating a relational view for the head of the rule using the SQL expression

```
CREATE VIEW p(attr1, attr2) AS SELECT attr1, attr5 FROM body
```

This enables the optimizer of the database system to find a well-performing execution plan for the final SQL query.

3. Installation and Configuration

3.1 Installation

IRIS is an open-source Datalog reasoner developed under the GNU Lesser General Public License (LGPL) and provided as a Java implementation that can be downloaded in both, source and binary form, from the Sourceforge project page[7]. The extension described in this deliverable, called IRIS-RDB, is an additional module of IRIS and is, therefore, also licensed under LGPL and is available on the IRIS Sourceforge page.

Since version 0.7.0 IRIS is delivered and maintained as an Apache Maven[8] project. For IRIS-RDB an additional module has been added to the IRIS project. To get releases and snapshots of IRIS-RDB and the dependent components, the following repositories have to be added to the project object model (POM) file:

```
<repositories>
  <repository>
    <id>sti2-archiva-external</id>
    <url>http://maven.sti2.at/archiva/repository/external</url>
  </repository>
  <repository>
    <id>sti2-archiva-snapshots</id>
    <url>http://maven.sti2.at/archiva/repository/snapshots</url>
  </repository>
</repositories>
```

The standard SOA4All project setup should have the SOA4All NEXUS repository[9] hosted by TIE in its configuration, which mirrors both STI repositories, thus they do not need to be added explicitly. The repositories that should be used in the configuration for mirroring are:

- `http://coconut.tie.nl:8080/nexus-webapp-1.3.1/content/groups/public/`
- `http://coconut.tie.nl:8080/nexus-webapp-1.3.1/content/groups/public-snapshots/`

The current stable version of IRIS and IRIS-RDB, as of 09.02.2011, is version 0.8.0. Ongoing development is committed to the snapshot version `0.8.1-SNAPSHOT`. IRIS-RDB can be added as dependency by adding `at.sti2.iris:iris-rdb` as dependency to the POM file:

```
<dependency>
      <groupId>at.sti2.iris</groupId>
      <artifactId>iris-rdb</artifactId>
      <version>0.8.0</version>
</dependency>
```

[7] IRIS project page at Sourceforge, http://sourceforge.net/projects/iris-reasoner/ [last checked 24.01.2011]

[8] Apache Maven, http://maven.apache.org/ [last checked 06.02.2011]

[9] SOA4All NEXUS repository, http://coconut.tie.nl:8080/nexus-webapp-1.3.1 [last checked 24.01.2011]

3.2 Configuration

IRIS and IRIS-RDB can be programmatically configured when initializing a knowledge base. All configuration parameters are collected together in a single configuration object that is passed to the knowledge base, thus allowing a highly flexible combination of standard and user-provided components. The configuration class contains these categories of parameters:

- **Factories** for evaluation strategies, rule compilers, rule evaluators, relations and indexes. N.B., not used in IRIS-RDB.
- **Termination parameters** for termination conditions (time out, maximum tuples, maximum complexity).
- **Numerical behaviour** determining significant bits of floating point precision for comparison, divide by zero behaviour.
- Collections of **program optimizers**, **rule optimizers** and rule **re-ordering optimizers**.
- Collection of **rule stratifiers**.
- **Rule-safety processor** for detecting unsafe rules or making unsafe rules safe.
- Unlike the original IRIS, IRIS-RDB provides **no** support for **external data sources**.

Furthermore, the IRIS-RDB knowledge base can be configured to use

- A newly created embedded database stored in the temporary directory of the user running the Java program.
- An in-memory H2 database.
- An already existing database referenced by a `java.sql.Connection` object. N.B., IRIS-RDB has currently only been tested with the H2 database system.

3.3 Usage Example

Listing 1 gives an example Java program that creates an IRIS-RDB knowledge base for the program shown in Listing 2, executes all the queries defined in this program over the previously created knowledge base and outputs the resulting relation to the console. For the sake of simplicity, exceptions are not handled in this example.

Listing 1: Usage example.

```
public class Example {
	public static void main(String[] args) throws Exception {
		// Create a Reader on the Datalog program file.
		File program = new File("datalog_program.iris");
		Reader reader = new FileReader(program);

		// Parse the Datalog program.
		Parser parser = new Parser();
		parser.parse(reader);

		// Retrieve the facts, rules and queries from the
		// parsed program.
		Map<IPredicate, IRelation> factMap = parser.getFacts();
		List<IRule> rules = parser.getRules();
		List<IQuery> queries = parser.getQueries();

		// Create a default configuration.
		Configuration configuration = new Configuration();

		// Enable Magic Sets together with rule filtering.
		configuration.programOptmimisers.add(new RuleFilter());
		configuration.programOptmimisers.add(new MagicSets());

		// Convert the map from predicate to relation to a
		// IFacts object.
		IFacts facts = new Facts(factMap,
			configuration.relationFactory);

		// Create the knowledge base.
		IKnowledgeBase knowledgeBase = new RdbKnowledgeBase(facts,
			rules, configuration);

		// Evaluate all queries over the knowledge base.
		for (IQuery query : queries) {
			List<IVariable> variableBindings =
				new ArrayList<IVariable>();
			IRelation relation = knowledgeBase.execute(query,
				variableBindings);

			// Output the variables.
			System.out.println(variableBindings);

			// For performance reasons compute
			// the relation size only once.
			int relationSize = relation.size();

			// Output each tuple in the relation, where the term
			// at position i corresponds to the variable at
			// position i in the variable bindings list.
			for (int i = 0; i < relationSize; i++) {
				System.out.println(relation.get(i));
			}
		}
	}
}
```

The Datalog program used in this example is shown in Listing 2. The program creates 20 pairs of X and Y, where 0 <= X < 200 and Y = X + 1 and then computes all possible paths by the recursive joining of all pairs.

Listing 2: Recursive Datalog program using built-ins.

```
p(0, 1).
p(?X1, ?Y1) :- p(?X, ?Y), ?X + 1 = ?X1, ?Y + 1 = ?Y1, ?X < 200.

path(?X, ?Y) :- p(?X, ?Y).
path(?X, ?Y) :- path(?X, ?Z), path(?Z, ?Y).

?- path(?X, ?Y).
```

Listing 3 shows a part of the output produced by the Java program defined in Listing 1, which is the result of the query `?- path(?X, ?Y)` that gives all the possible 20503 transitive paths.

Listing 3: Part of the output of Java program defined in Listing 1.

```
[?X, ?Y]
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(0, 5)
(0, 6)
(0, 7)
(0, 8)
(0, 9)
(0, 10)
```

4. Evaluation

IRIS-RDB has been developed with the goal to have a more scalable Datalog reasoner, which can process Datalog programs that contain and produce facts that do not fit in the memory of a single computer. The evaluation focuses on the comparison of the original IRIS with IRIS-RDB with respect to the performance and, more importantly, to the scalability of the system.

All test results were produced using the rule-filtering and Magic Sets optimization techniques and were run on a system with:

- Intel ® Core™ i7-620M 2x 2.66GHz,
- Windows 7 (64-bit),
- 4 Gbyte DDR2 RAM,
- Oracle Java SE Development Kit (JDK) 6 Update 23 (32-bit).

The focus of this evaluation was to measure the time to do inference rather than loading the test data sets. Thus, we only measured the time it took to evaluate a query, and did not consider the loading of the facts into the database. If not stated otherwise, the results in the tables below show the times measured in seconds.

4.1 OpenRuleBench

The scalability tests were taken from the OpenRuleBench [4] test suite, in particular, we used some of the test cases of the *large join tests* category, which includes large database joins, LUBM-derived tests, the Mondial and the DBLP test. In the detailed report of the OpenRuleBench [5] it has been observed that IRIS could not handle any of the large join tests, due to a timeout. Therefore, these test cases seemed to be a suitable candidate to check if the IRIS-RDB system meets the stated expectations.

Unfortunately, only the Join1 and Join2 tests could be run, as the LUBM-derived tests were not available in an IRIS compatible format, and the Mondial tests contained function symbols, which are not supported by IRIS-RDB. We could not use the DBLP tests as the IRIS parser failed to load the program due to memory limitations.

Unlike the OpenRuleBench we did not use a timeout, which determined the maximum allowed time to run an evaluation, but waited until the system produced a result or until an error occurred. In the tables below, “Error” means, that the system produced an `OutOfMemoryError` after some time, even if we assigned 1536 megabyte of memory to the Java virtual machine.

4.1.1 Join1

The Join1 test has a form of a non-recursive tree of binary joins, which is expressed using the rules shown in Listing 4.

Listing 4: Rules for Join1

```
a(?X, ?Y) :- b1(?X, ?Z), b2(?Z, ?Y).
b1(?X, ?Y) :- c1(?X, ?Z), c2(?Z, ?Y).
b2(?X, ?Y) :- c3(?X, ?Z), c4(?Z, ?Y).
c1(?X, ?Y) :- d1(?X, ?Z), d2(?Z, ?Y).
```

The relations for the predicates `c2`, `c3`, `c4`, `d1`, and `d2` were randomly generated. OpenRuleBench provides three datasets: *data0* with 50 000, *data1* with 250 000 and *data2* with 1250000 tuples. In our evaluation we only used the datasets *data0* and *data1* as the IRIS parser did not manage to process *data2* containing 1 250 000 tuples.

The test further defines nine queries on the predicates `a`, `b1` and `b2`. There are three queries for each predicate where one query has no variable binding, one has a binding on the first variable and one has a binding on the second variable. The queries are shown in Listing 5, where each line in the listing represents a single test.

Listing 5: Queries for Join1

```
?- a(?X, ?Y).
?- b1(?X, ?Y).
?- b2(?X, ?Y).

?- a(1, ?Y).
?- b1(1, ?Y).
?- b1(1, ?Y).

?- a(?X, 1).
?- b1(?X, 1).
?- b2(?X, 1).
```

Table 3 shows the results of the Join1 test with unbound variables in the query. As expected, the original IRIS could not compute the result for query `a` due to an `OutOfMemoryError`. Interestingly, for *data0* we succeeded in computing results for the other two queries, unlike in the evaluation conducted by OpenRuleBench authors, where IRIS did not manage to evaluate any of the programs. This might be the case, since the OpenRuleBench authors only assigned 512MB of memory to the Java virtual machine running the programs, whereas we assigned 1536MB of memory. This also applies for the Join2 test in Section 0.

Table 3: Times for Join1, no query bindings

	data0			data1		
	a	b1	b2	a	b1	b2
IRIS-RDB	1068.875	67.726	7.112	16685.639	474.931	74.71
IRIS	Error	19.069	1.417	Error	Error	45.16

Table 4 shows the results of the Join1 test with a binding on the first variable. In this test, both systems take heavy advantage of the Magic Sets optimization in order to rewrite the program in a way such that the data handled in the process of evaluation is limited by the variable bindings in the query. Thus, the evaluation times are significantly lower than in the test above. For *data0*, IRIS performs better than IRIS-RDB in all three tests, showing a significant difference for the query on predicate `a`. For *data1* IRIS did not manage to compute the first two queries due to an `OutOfMemoryError`. Interestingly, it can be observed, that

the performance difference for query `b2` is less significant than for *data0*. We assume that for larger fact bases the drawback of hard-disk access may be amortized by the set-oriented techniques used by the database system.

Table 4: Times for Join1 with first argument bound

	data0			data1		
	a	b1	b2	a	b1	b2
IRIS-RDB	13.275	0.475	0.178	186.372	0.749	0.187
IRIS	1.332	0.135	0.112	53.711	0.593	0.172

Table 5 shows the results of the final Join1 test where the second variable is bound. In this test, IRIS performs significantly better than IRIS-RDB on all three queries. For query `a` on *data1 IRIS* did not manage to evaluate the program.

Table 5: Times for Join1 with second argument bound

	data0			data1		
	a	b1	b2	a	b1	b2
IRIS-RDB	164.119	21.497	1.061	735.923	454.904	18.611
IRIS	34.725	1.404	0.063	Error	62.681	0.625

4.1.2 Join2

The Join2 test defines the rules and queries shown in Listing 6. The facts for the program consist of the tuples `p(abcd0), ..., p(abcd18)`. The program produces a large intermediate result, but only a small set of answers for the query `?- q(?X).`

Listing 6: Rules and queries for Join2.

```
ra(?A, ?B, ?C, ?D, ?E) :- p(?A), p(?B), p(?C), p(?D), p(?E).
rb(?A, ?B, ?C, ?D, ?E) :- p(?A), p(?B), p(?C), p(?D), p(?E).
r(?A, ?B, ?C, ?D, ?E) :- ra(?A, ?B, ?C, ?D, ?E),
                         rb(?A, ?B, ?C, ?D, ?E).
q(?A) :- r(?A, ?B, ?C, ?D, ?E).
q(?B) :- r(?A, ?B, ?C, ?D, ?E).
q(?C) :- r(?A, ?B, ?C, ?D, ?E).
q(?D) :- r(?A, ?B, ?C, ?D, ?E).
q(?E) :- r(?A, ?B, ?C, ?D, ?E).
```

Table 6 shows the results of the Join2 test. IRIS did not manage to evaluate the program due to an `OutOfMemoryError`.

Table 6: Times for Join2

	q
IRIS-RDB	1773.478
IRIS	Error

4.2 Built-in Predicates

In order to test the performance of Datalog programs with built-in predicates, we have run the program shown in Listing 2, where we have varied the number that limits the range of the variable `X` and, therefore, determines the number of recursive calls of the rule on line 2. Surprisingly, IRIS-RDB performs almost as well as the original IRIS. We expected that, due to the one-tuple-at-a-time iteration, and the continuous hard disk access that is required when evaluating rules with built-in predicates, the performance of IRIS-RDB was significantly worse compared to the in-memory evaluation of IRIS.

Table 7 depicts the results of the evaluations, where the number in parentheses shows the number of tuples in the output relation of the predicate `path`.

Table 7: Times for program with built-in predicates.

	200 (20503)	400 (80601)	800 (321201)	1000 (501501)
IRIS-RDB	14.365	100.216	1056,399	2439.453
IRIS	9.92	86.623	825.255	1897.620

4.3 Evaluation Conclusion

The results show that IRIS-RDB is able to evaluate Datalog programs for which the original in-memory implementation fails to compute a result. However, the results also outline that IRIS performs better than IRIS-RDB in those tests that it manages to process. The reason for this may be that IRIS-RDB requires continuous hard disk access, whereas, IRIS processes everything in-memory. Furthermore, in each run of the semi-naive evaluation, the system copies each delta relation, see Annex A, to a dedicated relation in the database, which may also have an influence on the performance, especially for large and numerous intermediate relations.

We also presume that the performance of IRIS-RDB could be increased by reducing the tuple size of the intermediate relations and by optimizing the SQL expressions by, for instance, changing the order of the joins, such that relations with the smallest size are joined first, which in turn reduces the size of intermediate relations.

An advantage of using a relational database system as underlying engine for evaluating Datalog programs is that the system may benefit of the performance optimizations of future versions of the DBMS.

5. Conclusions

In this deliverable, we have presented IRIS-RDB, an extension of the IRIS reasoner that uses the database engine H2 as an underlying relational database system to evaluate Datalog programs. H2 is a very fast and feature-rich relational database system that supports persistent and in-memory storage and has both an embedded and a server mode, which proved to be suitable for the purpose of IRIS-RDB.

IRIS-RDB can evaluate safe or unsafe Datalog programs containing rules with equality in the head. It supports XML, RDF, and RIF data types, an extensive set of built-in predicates and (locally) stratified negation as failure. Furthermore, the database binding extends IRIS with a persistent data storage facility that enables for processing continuously growing data, which is a necessity when reasoning in the context of millions of services.

We have shown that the system is able to evaluate Datalog programs for which the original IRIS fails to compute a result due to the limits on the data it can process in memory. However, this increased degree of scalability comes at the cost of reduced performance on programs where this does not apply.

6. References

1. Ullman, J. D., *Principles of Database And Knowledge-Base Systems*, W.H. Freeman & Co. Ltd., 1988
2. Winkler, D., Bishop, B., *D3.2.5 Second Prototype Repository Reasoner for WSML-Core v2.0*, 2010
3. Van Gelder, A., Ross, K., Schlipf, J. *The Well-Founded Semantics for General Logic Programs*, in Journal of the AssocM1on for Computing Machinery 38(3), pp. 620-650, 1991
4. Senlin Liang, S., Fodor, P., Wan, H., Kifer, M., *OpenRuleBench: An Analysis of the Performance of Rule Engines*, 2009
5. Senlin Liang, S., Fodor, P., Wan, H., Kifer, M., *OpenRuleBench: Detailed Report,* 2009
6. IRIS – Integrated Rule Inference System – API and User Guide, http://iris-reasoner.org/pages/user_guide.pdf, 2008
7. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J. D., *Magic Sets and Other Strange Ways to Implement Logic Programs*, Proceeding PODS '86 Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems, 1986

Annex A.

Semi-naive Datalog Evaluation as defined in [1, page 125]

Input: A collection of datalog rules with EDB predicates `r1, ..., rk` and IDB predicates `p1, ..., pm`. Also, a list of relations `R1, ..., Rk` to serve as values of the EDB predicates.

Output: The last fixed point solution to the datalog equations obtained from these rules.

Method: We use `EVAL` once to get the computation of relations started, and then use `EVAL-INCR` repeatedly on incremental IDB relations. The computation is shown in the following program, where for each IDB predicate `pi`, there is an associated relation `Pi` that holds all the tuples, and there is an incremental relation `ΔPi` that holds only the tuples added on the previous round.

```
for i := 1 to m do
  ΔPi := EVAL(pi, R1, ..., Rk, Ø, ..., Ø);
  Pi := ΔPi;
end
repeat
  for i := 1 to m do
    ΔQi := ΔPi; /* save old ΔP's */
  for i := 1 to m do begin
    ΔPi := EVAL-INCR(pi, R1, ..., Rk, P1, ..., Pm, ΔQ1, ..., ΔQm);
    ΔPi := ΔPi - Pi; /* remove "new" tuples that appeared before */
  end;
  for i := 1 to m do
    Pi := Pi ∪ ΔPi; /* save old ΔP's */
until ΔPi = Ø for all i
output Pi's
```

The expression `EVAL-RULE(r, R1, ..., Rn)` computes for a rule `r` from the relations `R1, ..., Rn` a relation `R(X1, ..., Xn)` as defined in Section 2.4.2.

`EVAL(pi, R1, ..., Rk, P1, ..., Pm)` is defined as the union of `EVAL-RULE(...)` for each of the rules `r` for a predicate `pi`, projected onto the variables of the head.

The *incremental relation* `EVAL-RULE-INCR` for rule `r` is the union of the `n` relations

$$\text{EVAL-RULE}(r, R_1, \ldots, R_{i-1}, \Delta R_i, R_{i+1}, \ldots, R_n)$$

for `1 <= i <= n`. That is, in each term, exactly one incremental relation is substituted for the full relation. Formally, we define:

$$\text{EVAL-RULE-INCR}(r, R_1, \ldots, R_{i-1}, \Delta R_i, \ldots, \Delta R_n) = \bigcup_{i<=i<=n} \text{EVAL-RULE}(r, R_1, \ldots, R_{i-1}, \Delta R_i, R_{i+1}, \ldots, R_n)$$