A Mediator-based Data Integration System for Query Answering using an Optimized Extended Inverse Rules Algorithm

by

Gayathri Jayaraman

A Thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfilment of

the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Ottawa-Carleton Institute for

Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

May 2010

The undersigned recommend to

the Faculty of Graduate Studies and Research

acceptance of the Thesis

A Mediator-based Data Integration System for Query Answering using an Optimized Extended Inverse Rules Algorithm

Submitted by **Gayathri Jayaraman**

in partial fulfilment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Leopoldo Bertossi, Supervisor

Babak Esfandiari, Co-Supervisor

Dr. Howard Schwartz, Department Chair

Carleton University

May 2010

Abstract

A mediator system allows users to pose queries against a global schema and returns answers from multiple data sources. The rewriting of the user query in terms of the local sources uses *mappings*, which in the Local-As-View (LAV) approach, describe the source relations as views over the global schema. Among the existing algorithms that perform query rewriting in LAV, the *Extended Inverse Rules Algorithm* (EIRA) provides the most general approach. Given a set of mappings and database facts, EIRA provides a logic program, which specifies a class of legal instances of the global system. The specification of the legal instances can be used to compute *certain* answers for user queries that are *monotone*.

However, the output of EIRA is only a program specification. Therefore, applying it in a data integration system for query answering requires the design of a system that can store, specify and query the metadata representation. Moreover, it is inefficient to consider all the available mappings and use the facts from all the sources for computing answers to the user query.

In this thesis, we describe the design, representation and implementation of a mediator system, called *Virtual Integration Support System* (VISS), that uses an optimized EIRA for query answering. We describe a general framework for metadata representation in a virtual and relational data integration system under the LAV approach. Specifically, we use *XML* and *RuleML* for representing metadata, viz. the global and local schemas, the mappings between the former and the latter, and global integrity constraints.

We also show how to obtain a *reduced* set of mappings and a subset of available sources for a user query. Using this, we optimize the logic program by generating only the required parts (i.e., those that can be used for answering the query) of

the specification program in EIRA. We also import only the relevant facts using the reduced list of sources for computing the answers.

We describe how *XQuery* can be used to retrieve the relevant information for EIRA based on our optimized approach. The information is then used to build the logic program specification for computing *certain* answers. The implementation of *VISS* uses open-source tools and is used to compute *certain* answers to *Datalog* queries, which are *monotone*.

Acknowledgments

I would like to thank my supervisor and mentor, Professor Leopoldo Bertossi for his wisdom, advice and support throughout my thesis. His guidance and technical questions has been an eye opener for me on many occasions. In addition to his technical guidance, he had provided me excellent ideas and feedback on writing technical papers.

I would like to thank my co-supervisor Professor Babak Esfandiari for his time and feedback. His feedback made me look back and address some unanswered questions. I would like to thank Dr. Harold Boley for providing his time, feedback and support for my work on RuleML. I would like to thank the chair of my thesis examination committee, Professor Gabriel Wainer and the members of my thesis examination committee, Professor Diana Inkpen and Professor Michael Weiss for their time and ideas on the thesis.

I would like to thank Alexander Beauvais for implementing a metadata interface for this research. I would like to thank Monica Caniupan for giving me access to the *Consistency Extractor* system.

Last, but not least, I would like to thank my family - my husband Ram and my 3 kids who have given me a great deal of support without which I would not be where I am today, and my parents who will always be with me in spirit guiding me in difficult times.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Current day computer applications have a need to access, process, report and specially integrate data from various and disparate sources. The data sources are created, maintained and published in formats that adhere to their own organization-specific standards. Data integration systems aim to provide a single unified interface for combining data in various formats from those multiple sources [13]. One of the main approaches in the integration of data is to use a Data Warehouse, where data from multiple data sources are extracted, transformed and projected as a new database, which is a collection of views containing data [48] [70] [72] [66]. The architecture of a Data Warehouse is shown in Figure 1.1. The data in the Data Warehouse may be structured differently from the source. Data is not fresh as the load happens at scheduled times.

Another form of data integration involves peers exchanging data based on certain mappings [23]. When a query is posed to a peer, it sends its data and/or imports data from other peers. The data exchange is an iterative process wherein each peer can in turn import data from its neighboring peers based on trust relationships and there is no central component managing the transfer of data.

The data integration approach we discuss in this research work is the mediation system [69] or virtual data integration system that offers a query interface over a *single global schema*. The global schema consists of relational predicates, in terms of which the user can pose queries. However, there is no actual data contained in them. When the mediator receives a query in terms of the global relations, it produces a

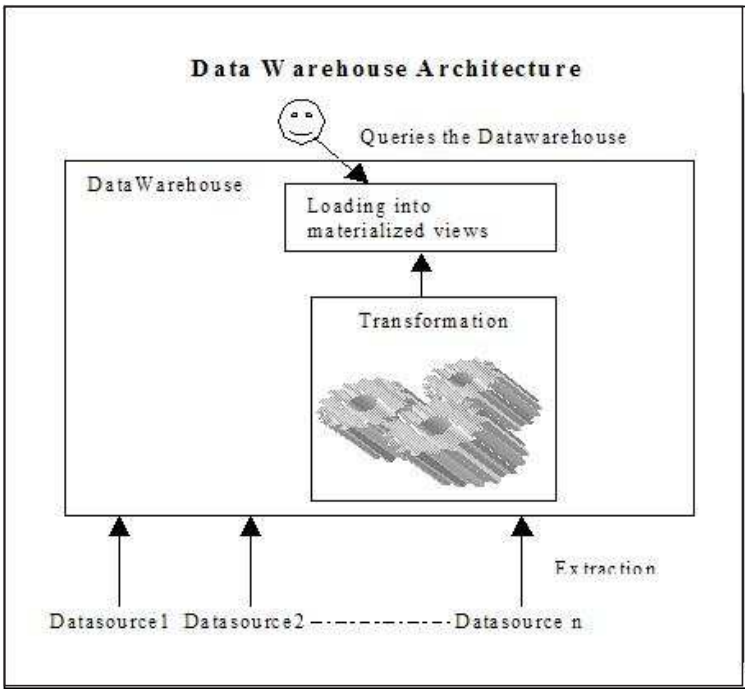


Figure 1.1: Architecture of Data Warehouse.

query plan that identifies the *relevant* data sources and the *relevant* data in them, and specifies how the data obtained from them has to be combined to build the final answer. To produce such a plan, the mediator stores and processes certain *mappings* or view definitions that associate the predicates in the global schema with those in the local sources.

There are different approaches to virtual data integration, depending on how metadata mappings are represented [68] [43]. The *Local-As-View* (LAV) approach, which is used in this research work, defines the local source relations as views over the global schema [15; 50]. In this way, each relevant source relation can be defined independently from other source relations. By doing so, it is easier for any source to join or leave the system, without affecting other source definitions.

The structure of the global schema and mappings constitute the metadata in a mediator system. The metadata will also contain constraints and details for accessing the data sources. The metadata describes the nature of the data sources and data contained in them. The efficient implementation of a query planning mechanism relies on a proper design, representation and querying of metadata to extract required

information.

There are three main algorithms that aim to provide query rewriting under the LAV approach. The *Bucket Algorithm* implemented in the *Information Manifold* [55] mediator considers queries and view definitions that are conjunctive queries with comparison predicates. In the case of the *Extended Minicon Algorithm* [62], the queries and view definitions are conjunctive queries with comparison predicates such as $<, \leq, \neq$. The classic *Inverse-Rules Algorithm* (IRA) accepts view definitions without comparison predicates but handles functional dependencies, recursive queries and binding pattern limitations [53]. Under the LAV approach, the IRA provides all and only *certain* answers to conjunctive queries.

In this research, we use *Extended Inverse Rules Algorithm* (EIRA) for query planning. EIRA is an extended version, introduced in [18] [15], of *Inverse-Rules Algorithm* (IRA) [31]. EIRA inherits the advantages of IRA and also handles *monotone*[1] queries with built-ins. It is based on a logic program specification of *minimal legal instances* [19]. The resulting query plan obtained using this algorithm is expressed as an extended *Datalog* program with stable model semantics [32]. Using EIRA, for a *monotone* query, we obtain *certain* answers, which are true in all *minimal legal instances* of the global system [19].

Given a user query and a set of view definitions in LAV, EIRA performs query rewriting. The view definitions of the source relations are given by conjunctive queries with built-ins. However, using EIRA "as is" can prove inefficient as there may be sources that violate some equality built-in in the query or may contain more than the required data for the query. Using and querying these sources without any pruning, becomes an overhead when we integrate data sources with large amounts of data. Also, if we consider all the available view definitions, the logic program generated using EIRA will contain unnecessary rules. Hence, query answering using EIRA will have to use some optimization that reduces the number of sources based on built-ins in the query and extracts only the required data from these sources.

[1] A query is monotone if its output obtained from a database instance remains true for any superset of that database instance

We use the relations and mappings in the following running example to illustrate the main issues and how we address the issues using our design approach and implementation of a virtual data integration system. Though the example uses relations from the animal domain, in general our mediator approach is applicable to any relational data source.

Example 1 Consider a relational data source, ***animalkingdom***, containing data about animals. It contains the relations **V1** storing information about animals of all classes with attributes *Name, Class, Food*, **V3** contains mammals with attributes *Name, Class, Food* and **V4** contains birds with attributes *Name, Class, Food*. Another data source, ***animalhabitat***, contains relation **V2** with attributes *Name, Habitat* and relation **V5** containing mammals with attributes *Name, Food*.

V1	*Name*	*Class*	*Food*
	dolphin	mammal	fish
	camel	mammal	plant
	shark	fish	fish
	frog	amphibian	insect
	nightingale	bird	insect

V2	*Name*	*Habitat*
	dolphin	ocean
	camel	desert
	elephant	savannah
	giraffe	savannah
	lion	savannah
	whale	water
	parrot	tropical
	nightingale	forest
	deer	forest
	frog	wetlands

V5	*Name*	*Food*
	deer	grass

V3	*Name*	*Class*	*Food*
	dolphin	mammal	fish
	elephant	mammal	plant
	giraffe	mammal	leaves
	lion	mammal	animal

V4	*Name*	*Class*	*Food*
	parrot	bird	nuts
	robin	bird	insect
	nightingale	bird	insect

An information system designer interested in providing information on animals defines the following global schema $\mathcal{G}$: *Animal(Name, Class, Food)*, *Vertebrate(Name)*, *Habitat(Name, Habitat)*. This can be done even before the data sources **animalkingdom** and **animalhabitat** (and possibly others) are available to the system. A mapping that associates the global relations *Animal* and *Vertebrate* with the local

relation **V1** is given as follows:

$$\mathbf{V1}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (1.1)$$

Similarly mappings are defined between the global relations *Animal* and *Habitat* and the other local relations **V2**, **V3**, **V4** and **V5** as follows:

$$\mathbf{V2}(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \quad (1.2)$$

$$\mathbf{V3}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (1.3)$$

$$\mathbf{V4}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "bird". \quad (1.4)$$

$$\mathbf{V5}(Name, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (1.5)$$

Now consider a *Datalog* query, $\Pi(Q)$, posed to the mediator, to get all animals that are mammals with their names and habitat:

$$\begin{aligned} Ans(Name, Habitat) \quad \leftarrow \quad & Animal(Name, Class, Food), Habitat(Name, Habitat), \\ & Class = "mammal". \end{aligned} \quad (1.6)$$

Equation (1.6) is a conjunctive *Datalog* query whose answers cannot be computed by a simple direct computation of the rule body because the data is not stored as material relations over the global schema. Instead, the mappings that describe the source relations have to be used to produce a query plan that eventually queries the (relevant) local sources where the data is stored. □

It can be seen from Example 1 that, from the available sources and view definitions, not all are required to answer the query in Equation (1.6). We can rule out the data source $V4$ as the query in Equation (1.6) specifically asks for *mammal* and $V4$ contains data only about *bird*.

1.1 Problem Statement and Contributions

EIRA handles user queries that are *monotone*, which is a superset of conjunctive queries. EIRA also provides a mechanism for query answering in the presence of global integrity constraints. The logic program approach helps provide further extensions through the use of stable models. These features of EIRA allows us to use it in a general way for query answering in different scenarios. However, currently we do not have a mediator system that applies EIRA for computing *certain* answers. The design of such a mediator will have to address the requirements for storage, specification and querying of metadata. For efficient computation of *certain* answers using EIRA, the design should include some pruning steps when querying the metadata. Given a list of available data sources and view definitions, we need to prune the source relations wherever possible, based on the conditions in the query. And from the relevant sources, we need to extract the relevant data.

In this thesis, we describe a design, representation and implementation of a general mediator system, the *Virtual Integration Support System* (*VISS*), which can be used to integrate multiple relational data sources (or sources wrapped as relational). *VISS* is a first logic programming-based mediator system for the computation of *certain* answers to *monotone* queries using EIRA. The design, representation and implementation of *VISS* addresses some of the main aspects of a general mediator system namely:

1. Identifying and storing metadata information for use in query planning. The metadata is accessed, managed and stored in the mediator. It does not contain actual data.

2. Using expressive enough languages, namely XML and RuleML, and formats for representing metadata contained in the mediator. For example, RuleML has formal semantics to represent integrity constraints [17].

3. Using a standard way to query metadata to extract relevant information.

In *VISS* , we use the languages XML [13] and RuleML [17] to represent metadata and the query language XQuery to query metadata. The metadata about schemas

that are represented and stored as native XML are: (a) The access parameters for data sources (userid, password, etc.); (b) The structure of relations at the sources; and (c) The structure of relations in the global schema. The mappings in *VISS* are represented using specifications in the first-order logic sublanguage and *Datalog* subsets of RuleML [17]. RuleML is an XML-based markup language for representation and storage of rules expressed as formulas of predicate logic. *VISS* is also the first mediator system to use RuleML for representing LAV mappings.

The design of *VISS* also enhances the use of EIRA for query answering as follows:

1. Prunes the query to remove global relations that do not have the "right" variables to provide answers to query.

2. Filters data sources that are not related to global predicates (as specified in their view definitions) in a *monotone* query.

3. Filters data sources whose view definitions violate any of the equality conditions in a conjunctive query thereby reducing the list of data sources used. We extend this approach to disjunctive queries, which contain equality condition.

4. Retrieves data that satisfies built-ins in the query, thereby reducing the amount of data retrieved from data sources. We consider conditions such as $=$, $\geq$, $\leq$, $>$ and $<$ present in the query and apply them when querying data sources.

5. Prunes rules for certain global relations generated by EIRA that are not required for answering the query.

6. Removes rules in the program specification that are redundant, thereby structurally optimizing the logic program.

The *Bucket Algorithm* and *Minicon Algorithm* provide mechanisms to detect and filter irrelevant sources. Our approach partly follows the initial criteria used by these algorithms to filter sources. We also take into account queries that are cartesian products or do not have a join variable. Our approach can also be used for disjunctive queries with equality conditions.

In this thesis, we do not consider determining relevant sources based on global integrity constraints on global schema. But we provide an idea of how this work can be extended when there are global integrity constraints. When filtering sources, we specifically consider equality (=) built-in in the query and view definitions. The equality conditions are of the form $x = c$, where x is a variable and c is a constant. In general, the query and view definitions may contain other built-in operators, such as $\leq, \geq, \neq$.

In order to gather the relevant information needed to compute a query plan, *VISS* uses XQuery to query XML and RuleML metadata. This information is used to build the logic program. *VISS* also generates the *Refined Program Specifications*, as shown in [18] [15], of the *minimal legal instances* as an added feature. The refined specification can be used to extend the functionality of VISS, providing support for *consistent query answering* [16]. *Consistent* answers are defined as those that are true in every repair of all the *minimal legal instances* of a global system.

This thesis is structured as follows. Chapter 2 introduces basic definitions related to virtual data integration and *Extended Inverse Rules Algorithm*. Chapter 3 discusses state of the art approaches used in the design and implementation of mediator systems and techniques used for detecting relevant data sources. Chapter 4 shows how XML and RuleML are used to specify schemas and mappings. Chapter 5 describes the theory behind our pruning approach for detecting source relevance and optimizing query answering using EIRA. Chapter 6 describes how queries in XQuery are used to implement the pruning approach discussed in Chapter 5 in *VISS*. Chapter 7 describes the architecture of *VISS* . Chapter 8 provides details on the component of *VISS* used for building logic programs. Chapter 9 explains the design rationale of *VISS* and provides experimental results on various scenarios of user queries comparing our approach with the original EIRA. Chapter 10 presents some conclusions and future work.

Chapter 2

Preliminaries

This chapter recalls the theoretical ideas and concepts pertaining to a mediator data integration system and its components. We also describe the classification of data sources from existing literature and concepts related to *Extended Inverse Rules Algorithm* and its refined version.

2.1 Basic Notions

In general terms, a virtual data integration system has three main components [50]: (a) A collection of local data sources with a (union) schema $\mathcal{S}$; (b) A global schema $\mathcal{G}$; and (c) A set of mappings $\mathcal{M}$ between the global and source schemas. A data source is an autonomous database that adheres to its own set of integrity constraints (ICs) that enforce consistency of data within that datasource by rejecting undesirable updates. A database is a model of an external domain. It contains data that characterizes and is relevant to the domain. The data may be structured, i.e. data is contained in tables called relations, or unstructured, i.e. data is not easily machine readable such as text files or even a mixture of both. In Example 1, $V1$, $V2$ are predicates in source schema $\mathcal{S}$. Those predicates offered by the global schema $\mathcal{G}$ do not have corresponding material instances. They are available to the user for querying through the mediator interface. In Example 1, *Animal*, *Vertebrate* and *Habitat* are elements of the global schema. In the following, we will also denote the integration system with $\mathcal{G}$.

2.2 Data Source Descriptions

A mediator should contain information on what is available at the sources and how the relations in the sources are mapped to the relations in the global schema. The description of sources determine the computation of the query plan. The description is represented by a set of logical formulas called rules or mappings. A mapping $\mathcal{M}$ between the predicates of local and global schema is a *Datalog* rule of the form:

$$R(X) \longleftarrow \bigwedge_{i=1}^{n} P_i(X_i), \tag{2.1}$$

where, R and $P_1, P_2, ..., P_n$ are predicates in the local and global schemas. X, X_1, $X_2...X_n$ are tuples of variables and/or constants. Each variable occurring in X must occur in one of X_1, $X_2...X_n$ [2].

When defining mappings, the level of detail may differ between the local and global schema. The global schema may also be organized structurally differently from the local schema. That is, a relation in the global schema may be associated with relations in different data sources and hence, contains attributes related to more than one local relation.

There are three main approaches to define mappings between the global schema and local sources [50].

Global-as-View: In the Global-as-View (GAV) approach, the global schema is comprised of views over the tables that are in the union of the local schema [50]. Using GAV, R is a predicate in the global schema and $P_1, P_2, ..., P_n$ are predicates in the local schema in Equation (2.1).

Example 2 We define view *AnimalInfo* using GAV as follows:

$$AnimalInfo(Name, Class, Food) \leftarrow V1(Name, Class, Food).$$

If there are other sources, say $V3(Name, Class, Food)$, containing similar information

as $V1$, then we can define them using GAV as follows:

$$\mathit{AnimalInfo}(Name, Class, Food) \leftarrow \mathit{V3}(Name, Class, Food).$$

Here, *AnimalInfo* is a union of *V1* and *V3*. Similarly, we define the view *AnimalHome* as:

$$\mathit{AnimalHome}(Name, Habitat) \leftarrow \mathit{V1}(Name, Class, Food), \mathit{V2}(Name, Habitat).$$

Here, *AnimalHome* is a join of relations *V1* and *V2* using attribute $Name$. □

The GAV approach is not flexible for insertion or deletion of relations in the source schema. However, query answering in GAV is simple in the absence of integrity constraints and is accomplished by unfolding view definitions to come up with the source relations [22].

Local-as-View: In the *Local-as-View* (LAV) approach, each relation in the local data sources is expressed as a view over the global schema [55]. Using LAV, R is a predicate in the local schema and $P_1, P_2, ..., P_n$ are predicates in the global schema in Equation (2.1). The global relations can be defined even before sources are added to the system. Based on the global schema created, potential sources can be added to contribute data to the system. Each source relation is independently defined from other relations in terms of global relations. Hence, sources can leave or join the system without affecting other view definitions.

Example 3 The global relations *Animal* and *Vertebrate* are associated with local relation $V1$ via a *Datalog* query, which defines $V1$ as a view over $\mathcal{G}$, as containing animals that are vertebrates:

$$\mathit{V1}(\mathit{Name}, \mathit{Class}, \mathit{Food}) \leftarrow \mathit{Animal}(\mathit{Name}, \mathit{Class}, \mathit{Food}), \mathit{Vertebrate}(\mathit{Name}). \quad (2.2)$$

Another mapping describes *V2* as containing animals and their habitat:

$$\mathit{V2}(\mathit{Name}, \mathit{Habitat}) \leftarrow \mathit{Animal}(\mathit{Name}, \mathit{Class}, \mathit{Food}), \mathit{Habitat}(\mathit{Name}, \mathit{Habitat}). \quad (2.3)$$

Equations (2.2) and (2.3) are LAV mappings. Using these mappings, a possible query plan in terms of source relations $V1$ and $V2$ for Equation (1.6) (cf. Example 1 in Chapter 1) is:

$$Ans(Name, Habitat) \leftarrow V1(Name, Class, Food), V2(Name, Habitat). \qquad (2.4)$$

□

Example 3 shows that it is possible to define other sources contributing with information about animals, such as invertebrates. In this sense, the information in the sources *V1* and *V2* can be considered as incomplete with respect to what $\mathcal{G}$ might potentially contain. As shown in Example 1 (cf. Chapter 1), for the *Datalog* query in Equation (1.6), a query plan will have to be generated to extract relevant information from the sources. A query plan is a rewriting of the query as a set of queries to the sources and a way to combine their answers. In the case of LAV, this translates to the problem of *Answering Queries using Views*, which is shown to be NP-complete [54]. Hence, query answering in LAV is more challenging than GAV.

Global and Local-as-View: In Global and Local-as-View (GLAV) approach, a view over the global schema is expressed as a view, which is a query over relations in local datasources [34]. This can be represented as: $\varphi_{\mathcal{S}}(X) \longleftarrow \varphi_{\mathcal{G}}(X_i)$, where, $\varphi_{\mathcal{S}}$ is a query over source schema $\mathcal{S}$, $\varphi_{\mathcal{G}}$ is a query over global schema $\mathcal{G}$ and $X \subseteq X_i$.

Example 4 Consider the source relations in Example 1. Using global relations *Animal* and *Habitat*, we define mappings as follows:

$$V1(Name, Class, Food) \wedge V2(Name, Habitat) \rightarrow Animal(Name, Class, Food) \wedge Habitat(Name, Habitat).$$

□

The GAV and LAV mappings are special cases of GLAV.

2.3 Classification of Sources

A source relation consists of a view definition φ, a label and a view extension v for φ [41]. Based on the labels, the sources contributing data to the system can be classified as *open*, *closed* or *clopen* (both *open* and *closed*). An *open* source contains a subset of the data of its kind in the global instance, a *closed* source contains a superset of the data of its kind in the global instance and a *clopen* source contains exactly all the data of its kind in the global instance.

Example 5 Consider a LAV integration system that combines data sources about vertebrate animals in a global relation $VertebrateAnimal(Name, Habitat)$.

Open Source: A source relation $Mammal$ contains names of animals belonging to class mammalia, given by the mapping:

$$Mammal(Name) \leftarrow VertebrateAnimal(Name, Habitat).$$

Its material extension is given by: $m = \{dolphin, camel, panda\}$. Every animal listed in $Mammal$ corresponds to some tuples that the global instance contains. The source relation $Mammal$ contains a subset of the expected entries in $VertebrateAnimal$. That is, $Mammal \subseteq \Pi_{Name}(VertebrateAnimal)$. Hence, $Mammal$ is said to be *Open*.

Closed Source: Consider a data source $AnimalWorld$, containing animals that are vertebrates and invertebrates. The mapping between global relation $VertebrateAnimal$ and the source relation is given by:

$$AnimalWorld(Name) \leftarrow VertebrateAnimal(Name, Habitat).$$

Its material extension is given by: $aw = \{dolphin, camel, shark, panda, frog, nightingale, spider, snail, squid, crabs\}$. The tuples in data source $AnimalWorld$ form a superset of animal family that are vertebrates. The tuple $\langle spider \rangle$ is in $AnimalWorld$, but not in $VertebrateAnimal$. That is, $AnimalWorld \supseteq \Pi_{Name}(VertebrateAnimal)$. Hence, $AnimalWorld$ is said to be *Closed*.

Clopen Source: Consider a data source $ChordataVertebrata$, containing names of animals that belong to phylum chordata and subphylum vertebrata. The mapping between global relation $VertebrateAnimal$ and the source relation is given by:

$$ChordataVertebrata(Name) \leftarrow VertebrateAnimal(Name, Habitat).$$

Its material extension is given by: $cv = \{dolphin, camel, shark, frog, nightingale, panda\}$. Here, $ChordataVertebrata = \Pi_{Name}(VertebrateAnimal)$. Hence, $ChordataVertebrata$ is said to be *Clopen*. □

Let D be a global instance and (i) v_i^o be a set of material source relations that are *open* defined by the views, $V_i^o(X_i) \longleftarrow \varphi_i^o(X_i^{'})$, $i = 1, ..., n$, where, V_i^o is a source predicate, φ_i^o is a conjunction of global predicates and $X_i \subseteq X_i^{'}$. Then, $V_i^o(D)$ denotes the tuples obtained by applying to D, the view definition V_i^o, (ii) v_i^c be a set of material source relations that are *closed* defined by the views, $V_i^c(X_i) \longleftarrow \varphi_i^c(X_i^{'})$, $i = 1, ..., m$, where, V_i^c is a source predicate, φ_i^c is a conjunction of global predicates and $X_i \subseteq X_i^{'}$. Then, $V_i^c(D)$ denotes the tuples obtained by applying to D, the view definition V_i^c and (iii) v_i^l be a set of material source relations that are *clopen* defined by the views, $V_i^l(X_i) \longleftarrow \varphi_i^l(X_i^{'})$, $i = 1, ..., k$, where, V_i^l is a source predicate, φ_i^l is a conjunction of global predicates and $X_i \subseteq X_i^{'}$. Then, $V_i^l(D)$ denotes the tuples obtained by applying to D, the view definition V_i^l. The global instance D is *legal*, if:
(i) for *open* sources, the computed extension on D of each view V_i^o contains v_i^o,
(ii) for *closed* sources, the computed extension on D of each view V_i^c is contained in v_i^c and
(iii) for *clopen* sources, the computed extension on D of each view V_i^l is the same as v_i^l.

That is,

$$\begin{aligned} Legal(\mathcal{G}) := \quad \{globalD \mid \quad & v_i^o \subseteq V_i^o(D); i = 1, ..., n, \\ & v_i^c \supseteq V_i^c(D); i = 1, ..., m, \\ & v_i^l = V_i^l(D); i = 1, ..., k\}. \end{aligned}$$

Example 6 Consider data sources V_1 and V_2 in Equation (2.2) and (2.3) and the extensions for the source predicates:

$$\begin{aligned} v_1 &= \{(dolphin, mammal, fish), (camel, mammal, plant), (shark, fish, fish), \\ &\quad (frog, amphibian, insect), (nightingale, bird, insect)\}. \\ v_2 &= \{(dolphin, ocean), (camel, desert), (frog, wetlands)\}. \end{aligned}$$

The global instance D_0 is given by:

$$\begin{aligned} Animal &= \{(dolphin, mammal, fish), (camel, mammal, plant), \\ &\quad (shark, fish, fish), (frog, amphibian, insect), \\ &\quad (nightingale, bird, insect), (snake, reptile, frog)\}. \\ Vertebrate &= \{dolphin, camel, shark, frog, nightingale, snake\}. \\ Habitat &= \{(dolphin, ocean), (camel, desert), (frog, wetlands)\}. \end{aligned}$$

The evaluation of the views on D_0 gives:

$$\begin{aligned} V_1(D_0) &= \{(dolphin, mammal, fish), (camel, mammal, plant), \\ &\quad (shark, fish, fish), (frog, amphibian, insect), \\ &\quad (nightingale, bird, insect), (snake, reptile, frog)\}. \\ V_2(D_0) &= \{(dolphin, ocean), (camel, desert), (frog, wetlands)\}. \end{aligned}$$

In this case, $v_1 \subseteq V_1(D_0)$ i.e. V_1 is *open* and $v_2 = V_2(D_0)$ i.e. V_2 is *clopen*. Hence, D_0 is a legal global instance; and all its supersets are also legal instances. □

A query Q is *monotone* if, for every two instances D, D', $D \subseteq D' \Longrightarrow Q[D] \subseteq Q[D']$ [2]. In particular, conjunctive queries are *monotone* [2]. Now, given a global *monotone* query $Q(\bar{X})$, i.e. expressed in terms of global predicates, a tuple $\bar{t}$ is a *certain* answer to Q if for every $D \in Legal(\mathcal{G})$, it holds $D \models Q[\bar{t}]$, i.e. the query becomes true in D with the tuple $\bar{t}$. $Certain_{\mathcal{G}}(Q)$ denotes the set of *certain* answers [1] to Q.

2.4 Program Specification and Query Answering

Extended Inverse Rules Algorithm, introduced in [19], for obtaining *certain* answers from a LAV integration system, is based on a specification as a logic program $\Pi(\mathcal{G})$ with stable model semantics of the legal instances of the system. The stable models of program $\Pi(\mathcal{G})$ [37] are (in correspondence with) the legal instances of $\mathcal{G}$. The specification is inspired by IRA algorithm [31], which introduces Skolem functions to invert view definitions. EIRA [19] replaces functions with auxiliary predicates, whose functionality is enforced in the specification by means of a *choice operator* [39].

The program $\Pi(\mathcal{G})$ for *Simple Specification* contains the following rules [19]:

1. The facts: $dom(a)$, for every constant $a \in U$; and $V(\bar{a})$, whenever $V(\bar{a}) \in v$, for some source extension $v \in \mathcal{G}$.

2. For every view (source) predicate V in the system, with definition $V(\bar{X}) \leftarrow P_1(\bar{X}_1), ..., P_n(\bar{X}_n)$, the rules:

$$P_j(\bar{X}_j) \leftarrow V(\bar{X}), \bigwedge_{X_i \in (\bar{X}_j \setminus \bar{X})} F_i(\bar{X}, X_i), \ j = 1, ..., n.$$

3. For every auxiliary predicate $F_i(\bar{X}, X_i)$ introduced in 2., the rule that makes it functional wrt the dependency of the last argument upon the first arguments:

$$F_i(\bar{X}, X_i) \leftarrow V(\bar{X}), \ dom(X_i), \ choice(\bar{X}, X_i).$$

The *choice operator* picks up only one value for X_i for every combination of values for $\bar{X}$. This operator can be eliminated as such, or equivalently, defined using standard rules. As a result, we obtain a program with stable model semantics whose *stable models* correspond to the *choice models* of the program [39]. We obtain them as follows [19]:

(a) Each *choice* rule $r : H \leftarrow B, choice((\bar{X}), (Y))$ in (3) is replaced by the rule:

$$H \leftarrow B, chosen_r(\bar{X}, Y).$$

(b) For each rule in (a), the following rules are included:

$$chosen_r(\bar{X}, Y) \leftarrow B, not\ diffchoice_r(\bar{X}, Y).$$
$$diffchoice_r(\bar{X}, Y) \leftarrow chosen_r(\bar{X}, Y'), Y \neq Y'.$$

The substitution for *choice operator* and the whole program is in Example 7. These specifications programs can also be modified in order to capture closed and exact sources [15].

Example 7 Program $\Pi(\mathcal{G})$ contains the facts:

$$dom(nightingale).dom(fish).dom(frog).dom(camel).$$
$$dom(amphibian).dom(fish).dom(insect).dom(plant).$$
$$dom(dolphin).dom(wetlands).dom(bird).dom(ocean).$$
$$dom(mammal).dom(shark).dom(desert).$$
$$V1(frog, amphibian, insect).V2(frog, wetlands).$$
$$V1(dolphin, mammal, fish).V2(dolphin, ocean).$$
$$V1(shark, fish, fish).V2(camel, desert).$$
$$V1(nightingale, bird, insect).$$

And the rules:

$$Animal(Name, Class, Food) \leftarrow V1(Name, Class, Food).$$
$$Vertebrate(Name) \leftarrow V1(Name, Class, Food).$$
$$Animal(Name, Class, Food) \leftarrow V2(Name, Habitat), F_1(Name, Habitat, Class), F_2(Name, Habitat, Food).$$
$$F_1(Name, Habitat, Class) \leftarrow V2(Name, Habitat), dom(Class), chosen_1(Name, Habitat, Class).$$
$$chosen_1(Name, Habitat, Class) \leftarrow V2(Name, Habitat), dom(Class), not\ diffchoice_1(Name, Habitat, Class).$$

$$diffchoice_1(Name, Habitat, Class) \leftarrow chosen_1(Name, Habitat, U), dom(Class), U \neq Class.$$

$$F_2(Name, Habitat, Food) \leftarrow V2(Name, Habitat), dom(Food), chosen_2(Name, Habitat, Food).$$

$$chosen_2(Name, Habitat, Food) \leftarrow V2(Name, Habitat), dom(Food), not\ diffchoice_2(Name, Habitat, Food).$$

$$diffchoice_2(Name, Habitat, Food) \leftarrow chosen_2(Name, Habitat, U), dom(Food), U \neq Food.$$

$$Habitat(Name, Habitat) \leftarrow V2(Name, Habitat). \qquad \square$$

In this thesis, we refer to the specification program used for computing *certain* answers as *Simple Specification*. For a *monotone* query, *certain* answers are those that are true in all *minimal legal instances*, i.e. those that do not contain a proper legal instance, of the global system. The answers obtained for a *monotone* query using *Simple Specification*, under the cautious reasoning, correspond to *certain* answers [19]. Cautious reasoning holds as true what is true in all of the stable models of the program.

In some cases, the openness of sources can be satisfied by the contents of other views and hence, it will not be necessary to compute values for the existential variables. But the *choice operator* in *Simple Specification* may still choose other values for these variables [19]. This may lead to more legal instances than the minimal ones. The instances corresponding to the models of the logic program from the *Simple Specification* form a class between the minimal and the legal instances, and it is a proper subclass of the legal instances. But this does not affect the computation of *certain* answers to *monotone* queries because, the *minimal legal instances* are contained in the subclass of legal instances of the global system specified by *Simple Specification*. Hence, what is true in the *minimal legal instances* of the global system is also true in the subclass of legal instances specified by *Simple Specification*.

The *minimal legal instances* are used to restore consistency of the system for doing

consistent query answering (CQA) [15; 18] (cf. [14] for a survey of CQA). While computing *consistent* answers, using *Simple Specification* may cause more repairs than required. There is a *Refined Version* of the *Simple Specification*, which is described in [19]. It specifies only the collection of *minimal legal instances*. This is achieved by using a stronger condition, $Add_{V_i}(\bar{X})$ in the place of $V(\bar{X})$ in the choice rules. The refined program for an open global system contains the following clauses [19]:

1. Fact $dom(a)$ for every constant $a \in \mathcal{U}$.
2. Fact $V_i(\bar{a})$ whenever $\bar{a} \in v_i$ for some source extension v_i in $\mathcal{G}$.
3. For every view (source) predicate V_i in the system with description, $V_i(\bar{X}) \leftarrow P_1(\bar{X}_1), \ldots, P_n(\bar{X}_n)$:

   (a) For every P_k with no existential variables, the rules:

$$P_k(\bar{X}_k, t_o) \leftarrow V_i(\bar{X}).$$

   (b) For every set S_{ij} of predicates of the description's body that are related by common existential variables $Z_1, \ldots, Z_m$, the rules:

$$\begin{aligned}
P_k(\bar{X}_k, v_{ij}) &\leftarrow add_{v_{ij}}(\bar{X}'), \bigwedge_{Z_l \in (\bar{X}_k \setminus \bar{X}')} F_i^l(\bar{X}', Z_l), \text{for } P_k \in S_{ij}. \\
add_{vij}(\bar{X}') &\leftarrow V_i(\bar{X}), not\ aux_{vij}(\bar{X}), \text{where } \bar{X}' = \bar{X} \cap \{\bigcup_{P_k \in S_{ij}} X_k\}. \\
aux_{vij}(\bar{X}') &\leftarrow \bigwedge_{l=1}^{m} var_{vijZ_l}(\bar{X}_{Z_l}). \\
var_{v_{ij}Z_l}(\bar{X}_{Z_l}) &\leftarrow \bigwedge_{P_k \in S_{ij} \& Z_l \in \bar{X}_k} P_k(\bar{X}_k, nv_{ij}),
\end{aligned}$$

   where $\bar{X}_{Z_l} = \bigcup_{P_k \in S_{ij} \& Z_l \in \bar{X}_k} X_k$, for l = 1,. . . ,m.
4. For every predicate $F_i^l(\bar{X}', Z_l)$ introduced in 3(b), the rules:

$$\begin{aligned}
F_i^l(\bar{X}', Z_l) &\leftarrow add_{v_{ij}Z_l}(\bar{X}'), dom(Z_l), choice((\bar{X}'), (Z_l)). \\
add_{v_{ij}Z_l}(\bar{X}') &\leftarrow add_{vij}(\bar{X}'), not\ aux_{v_{ij}Z_l}(\bar{X}'), \text{for } l = 1, \ldots, m. \\
aux_{v_{ij}Z_l}(\bar{X}') &\leftarrow var_{v_{ij}Z_l}(\bar{X}_{Z_l}), \bigwedge_{Z_k \neq Z_l \& Z_k \in \bar{X}_{z_l}} F_i^k(\bar{X}', Z_k), \text{for } l = 1, \ldots, m.
\end{aligned}$$

5. For every global relation $P(\bar{X})$, the rules:

$$
\begin{aligned}
P(\bar{X}, nv_{ij}) &\leftarrow P(\bar{X}, v_{hk}), \text{for } \{(ij, hk) | P(\bar{X}) \in S_{ij} \cap S_{hk}, ij \neq hk\}. \\
P(\bar{X}, nv_{ij}) &\leftarrow P(\bar{X}, t_o), \text{for } \{(ij) | P(\bar{X}) \in S_{ij}\}. \\
P(\bar{X}, t_d) &\leftarrow P(\bar{X}, v_{ij}), \text{for } \{(ij) | P(\bar{X}) \in S_{ij}\}. \\
P(\bar{X}, t_d) &\leftarrow P(\bar{X}, t_o).
\end{aligned}
$$

Here, $add_{vij}(\bar{X}')$ is true only when the openness of V_i is not satisfied through other views. $add_{vij}(\bar{X}')$ is further specified by means of extra rules. The refined version also uses annotation constants placed as an extra argument in the global relations. Specification programs like these can be evaluated with *DLV* system [52], for example. *DLV* computes *certain* answers wrt the skeptical (or cautious) stable model semantics of disjunctive logic programs with weak negation and program constraints.

Chapter 3

State of the Art

In this chapter, we discuss the design and implementation features of mediator data integration systems by describing their representation and querying of metadata. We mainly analyze mediators related to our line of research by describing those systems that use LAV. We also discuss the current main approaches for detecting relevant sources [46] [62] [6], used for producing a query plan.

3.1 Relational Model with Objects

The *Information Manifold* (IM) [46] is a mediator that uses LAV approach for mappings. IM uses the *Bucket Algorithm* for query answering. The metadata consists of description of source relations in terms of contents and capabilities of information sources. Both the contents and capabilities are represented using a combination of Horn rules and Classic Description Logic. The contents of sources are the LAV mappings. Each source relation is associated with one capability record of the form: $(S_{in}, S_{out}, S_{sel}, min, max)$. The meaning of the capability record is to specify minimum bindings of S_{in} elements required to get any of the elements in S_{out}.

Example 8 Consider the local relation $V1$, which is mapped to global relations $Animal$, $Vertebrate$ as follows:

$$V1(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (3.1)$$

The mapping is represented in IM as:

Contents:

$$V1(Name, Class, Food) \subseteq Animal(Name, Class, Food), Vertebrate(Name).$$

Capabilities:

$$(\{Name, Class, food\}, \{Name, Class, Food\}, \{Name, Class, Food\}, 1, 3).$$

The contents describe the openness of sources using $\subseteq$. □

The CARIN knowledge representation system is used in IM for metadata representation but with limited expressive power. For example, the relational component supports positive *Datalog*-like program and does not have a representation for disjunction and negation.

3.2 Infomaster

Infomaster [38] is a mediator system that uses LAV approach and stores metadata mappings in Knowledge Interchange format. Each attribute in the global and source relation is mapped separately. For accessing metadata for a user query, the mappings are first loaded into a main memory database *Epilog*.

Example 9 The mapping in Equation (3.1) is represented in Infomaster as:

$$(\Leftarrow \quad (Animal.Name\ ?x\ ?y)\ (V1.Name\ ?x\ ?y)).$$
$$(\Leftarrow \quad (Animal.Class\ ?x\ ?y)\ (V1.Class\ ?x\ ?y)).$$
$$(\Leftarrow \quad (Animal.Food\ ?x\ ?y)\ (V1.Food\ ?x\ ?y)).$$
$$(\Leftarrow \quad (Vertebrate.Name\ ?x\ ?y)(V1.Name\ ?x\ ?y)).$$

□

The system does not support built-ins in view definitions. The representation of each attribute in a mapping results in a large knowledge base that will have to be brought

to main memory, when accessed.

3.3 SIMS

SIMS mediator system [6] uses a description logic system, *Loom*, to represent metadata. The query answering approach in SIMS identifies relevant sources by accessing the information sources to search for relevant data. The view definitions are represented by mapping each external relation to a concept in *Loom* using a visual content model. Since only one concept can be associated with a source relation, this approach presents a restriction to express a source relation as a join of two global relations. SIMS uses a key column in source relation to create a *Loom* class. Every other column is viewed corresponding to a *Loom* relation, which describes the relation between item in this column and key column. Figure 3.1 shows an example of content model used in SIMS for Equation (3.1) in Example 8.

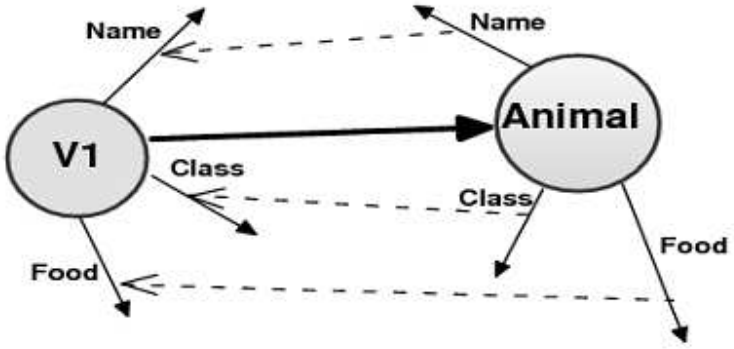


Figure 3.1: SIMS Model

3.4 Agora

Agora mediator system [58] follows LAV approach for mappings and represents relations in global schema as XML documents. The XML DTD (Document Type Definition) information for the XML documents representing each global relation is stored in a set of generic tables. The view definition for a source relation is expressed as a SQL query using attributes in the generic tables. We illustrate the representation

in Agora in Example 10.

Example 10 Consider the following mapping:

$$V1(Name, Class, Food) \leftarrow Animal(Name, Class, Food).$$

The global relation *Animal* is represented in XML format with *Animal* as root element and *Name*, *Class* and *Food* as child elements. The representation of mapping for $V1$ in Agora is given by:

```
select  e1.elID as $m, v4.value as Name, v5. value as Class,
        v6.value as Food
from
        Document d0, URI u0, Value v0, Child c0,
        Element e0, QName q0, Value v1, Child c1,
        Element e1, QName q1, Value v2, Child c2,
        Element e2, QName q2, Value v3, Child c3,
        Value v4, Child c4, Value v5, child c5,
        Value v6
where
        d0.docURIID=u0.uriID and u0.uriValID=v0.valID and
        v0.value="Animal.xml" and d0.docRootID=e0.elID and
        e0.elQNameID=q0.qNameID and q0.qnLocalID=v1.valID
        and v1.value="Animal" and c0.parentID=e0.elID and
        c0.childID=e1.elID and e1.elQNameID=q1.qNameID and
        q1.qnLocalID=v2.valID and v2.value="tuple" and
        c1.parentID=e1.elID and c1.childID=e2.elID and
        e2.elQNameID=q2.qNameID and q2.qnLocalID=v4.valID
        and v4.value="Name" and c3.parentID=e1.elID and
        c2.childValID=v4.valID and c3.parentID=e1.elID and
        c3.childID=e3.elID and e3.elQNameID=q3.qNameID and
        q3.qnLocalID=v5.valID and v5.value="Class" and
        c4.parentID=e3.elID and c3.childValID=v5.valID and
        c4.parentID=e1.elID and c5.childID=e4.elID and
        e4.elQNameID=q4.qNameID and q4.qnLocalID=v6.valID
        and v6.value="Food" and c5.parentID=e5.elID and
        c4.childValID=v6.valID.
```

The tables in *from* clause are the generic tables in the mediator that store the XML DTD of global relations. □

The mediator accepts user queries in XQuery on the global schema and translates them to SQL query on the global schema. This is because the mediator uses LeSelect,

a relational engine, to pose queries to data sources for retrieving data. It is not always possible to translate a query in XQuery to a query in SQL because some of XQuery language features do not have an SQL equivalent [58]. If the query in XQuery could be translated, the SQL query obtained in terms of the global schema after the translation is rewritten into an SQL query in terms of the local sources. A *relational query rewriting algorithm* is used for this purpose. The algorithm uses view definitions that describe a source relation in terms of the elements in the generic tables as shown in Example 10. In the rewriting step, the algorithm does not handle arbitrary levels of nesting or grouping.

3.5 Review of Approach

The main features of systems described in the preceding sections shows that, representation of metadata, viz. description of sources, global schema and mappings is an important aspect in the design of a mediator. A key requirement, as seen in the mediator systems reviewed, is a metadata representation that uses languages that can express the vast metadata stored in a mediator. Another point of note is the requirement of a query language when accessing metadata in a mediator. The query language should also preferably allow some extension, if required, to avoid a customized manipulation of metadata representation. Table 3.1 shows the main features of some of the other existing mediators and *VISS* .

Table 3.1: Mediator Systems

Mediator	**Mappings**	**Metadata Language**	**Metadata Access**	**Query Answering**
Tukwila [44]	LAV	XML	Custom C++ Code	Minicon
PICSEL [40]	LAV	CARIN	Custom Java Code	Query Expansion and Existential Entailment

XML-based Mediation Framework (XMF) [49]	GAV	XMF Mediation Rule (XMR)	XMF Query	Query unfolding
Nimble [30]	GAV	XML	XML-QL	Query unfolding
TSIMMIS [35]	GAV	Mediator Specification Language(MSL)/OEM	LOREL	Query unfolding
Garlic [27]	GAV	Garlic data Language(GDL)	O-SQL	Query unfolding
Hermes [56]	GAV	Prolog-like	Custom C code	Query unfolding
MOMIS [11]	GAV	Object Definition Language(ODL) and Semantic links using STASIS	O-SQL	Query unfolding
Xyleme [29]	GAV	XML in Natix store	OQL and XQL	Query unfolding
Yat [64]	GAV	YAT Language	Custom CGI code	Query unfolding
Automed [45]	BAV	Hypergraph Data Model	Hypergraph Query Language	Schema Transformation Algorithm
MIX [9]	GAV	XML and XML DTD	XML Matching and Structuring Language(XMAS)	Query unfolding
MedMaker [60]	GAV	Mediator Specification Language	Mediator Specification Interpreter	Query unfolding
VISS	LAV	XML and RuleML	XQuery	EIRA

The mediators using GAV are included in Table 3.1 to highlight the metadata representations used. However, specifications such as MSL and GDL are oriented towards representing GAV mappings.

3.6 Pruning Sources using Single Subgoal Buckets (SSB)

The first phase of the *Bucket Algorithm* [55] provides a method to detect relevant sources for a conjunctive query by considering each of the subgoals in the query. A subgoal refers to a predicate in the body of the query or view definition. A bucket called *Single Subgoal Bucket* is created for each subgoal. If a view definition contains a subgoal, which can be mapped to a subgoal in the query, then the view is placed in a bucket for that subgoal. If there are many subgoals that can be mapped to a subgoal in the same view, the view appears in more than one bucket. In the presence of built-ins in the query, views whose subgoals satisfy the built-ins are placed in the bucket.

Example 11 We use source relations $V1$, $V2$, $V3$, $V4$ and $V5$ and the global relations *Animal*, *Habitat* and pose a query to get all animals that are mammals with their names and habitat as follows:

$$\begin{aligned} Ans(Name, Habitat) \quad \leftarrow \quad & Animal(Name, Class, Food), Habitat(Name, Habitat), \\ & Class = "mammal". \end{aligned} \tag{3.2}$$

We use the following view definitions,

$$\mathbf{V1}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \tag{3.3}$$

$$\mathbf{V2}(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \tag{3.4}$$

$$\mathbf{V3}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \tag{3.5}$$

$$\mathbf{V4}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "bird". \tag{3.6}$$

$$\mathbf{V5}(Name, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \tag{3.7}$$

We consider one bucket each for the subgoals *Animal* and *Habitat* as follows:

Table 3.2: Subgoal Buckets.

Animal	Habitat
$V1$	**V2**
V2	
$V3$	
$V5$	

We can see from Table 3.2, that view $V2$ appears in the buckets for *Animal* and *Habitat*. □

The multiple occurrence of a view in the buckets may cause unnecessary candidate rewritings, when the algorithm creates a plan by taking a view from each bucket and combining it with views in the other buckets. If $b_1, b_2, ..., b_l$ are subgoal buckets created, each containing $m_1, m_2, ..., m_n$ number of views, the number of candidate rewritings considered in this algorithm is $m_1 \times m_2 \times \ldots \times m_n$. The algorithm misses the check where we can safely assume that, if a view is placed in a bucket of a subgoal, then it does not appear in the bucket of any other subgoal. This check can reduce the number of candidate rewritings.

3.7 Pruning Sources using Shared Variable Buckets (SVB)

An enhancement to *Bucket Algorithm* [59] for detecting relevant sources looks at shared variables if the condition for constructing single subgoal buckets is not satisfied by the view. A shared variable is one that is used to join two predicates and is identified in the body of a rule by the multiple occurrence of the same variable in more than one predicate. The *Shared Variable Bucket* proceeds by constructing a bucket representing all the subgoals containing a shared variable. A view is placed in a bucket if it covers all the subgoals in the bucket. The built-in conditions in the query and view definitions are also checked before the view is placed in the bucket. *Shared Variable Bucket* is considered only if an existential variable in a query, that is

mapped to a variable in the subgoal of a view also participates in a join.

Example 12 We use the source relations $V1$, $V2$, $V3$, $V4$ and $V5$, the global relations *Animal*, *Habitat*, the view definitions in Example 11 and the query in Equation (3.2). In this case, the shared variable buckets will not be considered because the existential variable in the query, say $Class$(or $Food$), is not a join variable. Hence, SVB approach will proceed similar to SSB and create single subgoal buckets. □

3.8 Pruning Sources using Minicon Descriptions (MCD)

The *Minicon Algorithm* [62] starts with subgoals in the query, and once it finds a partial mapping with a subgoal in a view, looks at the variables in the query. The algorithm considers attributes that are part of a join and finds the minimal additional set of subgoals, called Minicon Descriptions (MCDs), that need to be mapped with subgoals of this view. The *Minicon Algorithm* gives a rewriting for conjunctive queries and conjunctive view definitions with arithmetic comparison predicates. The MCD has similarities with SVB approach.

Example 13 We consider the same view definitions and query in Example 11. First, MCD determines that the view definition $V2$ has all the required predicates in the body that match those in the query. Then, it accepts sources $V1$, $V3$ and $V5$ also as relevant, because of the presence of built-in condition $Class =$ "*mammal*". Hence, MCD detects sources $V1$, $V2$, $V3$ and $V5$ as required sources for computing a query plan. □

3.9 Pruning Query and Inverse Rules

The classic *Inverse Rules Algorithm*(IRA) [31] proceeds by assuming that only required sources are available. But, after computing the inverse rules, it considers only those rules corresponding to the global predicates in the query.

Example 14 Consider the local relation $V1$, which is mapped to global relations $Animal$, $Vertebrate$ as follows:

$$\mathbf{V1}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (3.8)$$

For the query in Equation (3.2) in Example 11, we identified $V1$ as one of the required sources. The inverse rules for $V1$ is given as:

$$Animal(Name, Class, Food) \leftarrow \mathbf{V1}(Name, Class, Food).$$
$$Vertebrate(Name) \leftarrow \mathbf{V1}(Name, Class, Food).$$

But the query in Equation (3.2) does not contain $Vertebrate$. Hence, this rule will be dropped during query evaluation. □

3.10 Review and General Observations

The creation of *buckets* and the *minicon descriptions* are primarily used in the *Bucket* and *Minicon* algorithms respectively for producing candidate rewritings. That is, the sources identified in these approaches are used in the second phase of the algorithm for computing a (union of) conjunctive plan. In the case of EIRA, we use a logic program-based specification, where it is desirable to use only the relevant rules based on the relevant source relations. We do this by using part of the approach used in SSB, SVB and MCDs, namely, checking for view definitions that violate the built-in conditions in the query. We also perform some additional checks to prune the sources, where possible. Then, we use the relevant sources to determine the required rules in *Simple Specification* for computing *certain* answers. At this point, we also prune rules that may be redundant. In the following sections, we present some general observations based on our approach.

3.10.1 Cartesian Product Queries

We consider queries where a predicate does not have a join variable. In such cases, the query itself would require some pruning and we can safely disregard some global relations while computing answers.

Example 15 Suppose we have global relation $FoodList$, containing attribute $Food$ mapped to a source relation $FoodV1$ as follows:

$$FoodV1(Food) \leftarrow FoodList(Food). \tag{3.9}$$

Consider a query that asks for animal names using global relations $Habitat$ and $FoodList$ as follows:

$$Ans(Name) \leftarrow Habitat(Name, Habitat), FoodList(Food). \tag{3.10}$$

Here, the answer does not change in the absence of predicate $FoodList$ in the query. Hence, we could prune predicate $FoodList$ in the query, so we are left with global relation $Habitat$, to get required sources from the view definitions. □

We use a query pruning step to check for those global predicates in a query that are not required for computing answers. The global predicates in the query that do not have a join variable with the predicates containing attributes that appear in the head of the query are pruned.

3.10.2 Data Retrieval

The algorithms discussed in the preceding sections, detect relevant sources for a conjunctive query expressed in terms of the global relations. However, in the context of optimization, we also have to take into account how these sources are finally queried to get relevant data.

Example 16 We use the relations and query in Example 11. The relevant source relations for the query in 3.2 are $V1$, $V2$, $V3$ and $V5$. Querying $V3$ and $V5$ for

all records is acceptable because they contain records of animals belonging to class mammals only. But, $V1$ contains animals belonging to other classes also. Hence, a query to this source relation will have to retrieve only tuples satisfying the condition $Class = "mammal"$. □

In this context, we are formulating a way to query data sources without actually accessing them to analyze the records contained in them. So, a query to two different sources may still retrieve some redundant records. The relation $V1$ may contain a tuple $(V1(dolphin, mammal, fish))$, which is also available in $V3$. We rely solely on the description of the view definitions to understand and formulate the query to source relations.

Chapter 4

Metadata Representation

In this chapter, we first describe XML and RuleML in general. We explain how XML is used to represent part of the metadata and then, we describe how the language specification of RuleML is used for representing mappings under the LAV approach. We also illustrate, using examples, the entire metadata representation in XML and RuleML.

4.1 XML Metadata

XML is recognized as a language of choice to use in information integration applications because of its ability to handle variations in information content [13] [57]. When specifying metadata for integrating data sources, XML offers the flexibility of defining database schemas that vary from source to source, and also within each of them because it offers representation of custom elements to describe the content stored. Also, XML information can be parsed using many available parsers. XML documents have a tree structure that starts at a root element and branches to child elements. The XML elements are defined using tags. [21] gives a formal definition for an XML tree. XML is a W3C recommendation [20] for storing and exchanging information and XML data can be queried using XQuery, which is a W3C recommended query language for XML.

We show how XML is used for describing metadata about source and global schemas. The root element in the metadata representation is **VirInt**. The **VirInt**

element has a child element **Schema**. The elements **Global** and **Local** are child elements of **Schema**. The **Global** element lists details of the global schema, namely the global relations, their attributes and integrity constraints. Within the **Global** element, we represent global relations using element **Rel** and their attributes are represented using element **Var**. **Rel** and **Var** are siblings and their parent element is **Atom**. The usage of **Atom**, **Rel** and **Var** elements follows the specification of RuleML (cf. Section 4.3). The schema diagram for the specification of global schema is shown in Figure 4.1.

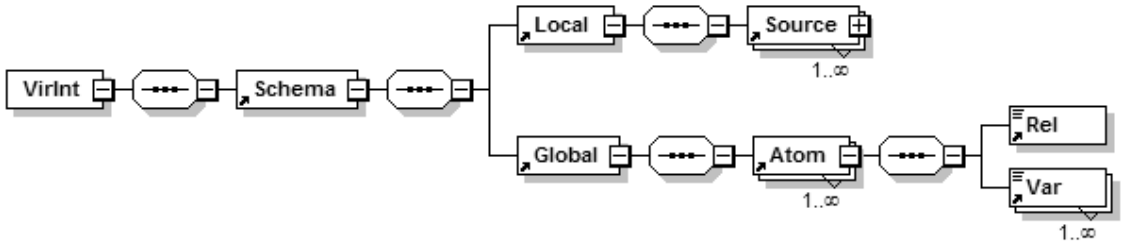


Figure 4.1: Global Schema

Listing 4.1 shows XMLSchema definition (XSD) for specifying global schema in the metadata representation.

Listing 4.1: XMLSchema Definition for describing the Global Schema

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="VirInt">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element ref="Schema" />
7        </xs:sequence>
8      </xs:complexType>
9    </xs:element>
10   <xs:element name="Schema">
11     <xs:complexType>
12       <xs:sequence>
13         <xs:element ref="Local" />
14         <xs:element ref="Global" />
15       </xs:sequence>
16     </xs:complexType>
17   </xs:element>
18   <xs:element name="Global">
19     <xs:complexType>
20       <xs:sequence>
21         <xs:element ref="Atom" maxOccurs="unbounded" />
22       </xs:sequence>
23     </xs:complexType>
24   </xs:element>
25 </xs:schema>
```

We follow the W3C standard for specifying the *XML Schema Definition* (XSD) [65] of metadata. For instance, our XML Schema refers to an element in the metadata representation using *xs:element*. The elements **VirInt**, **Schema** and **Global** are defined as *complex* types because they contain child elements. Lines 10-17 in Listing 4.1 defines **Schema** as a *complex* type containing the *sequence* of child elements **Local** and **Global**. The element **Atom** is defined as occurring multiple times in the metadata representation (cf. Line 21 in Listing 4.1). The structure of **Atom** element and its child elements, **Rel** and **Var**, use RuleML specification, which are explained in detail in Section 4.3.

The details of sources are stored within element **Local**. We describe sources by storing information about type of source (i.e. type of DBMS), connection parameters, and structure of relations. We use descriptive tags for this purpose. The **Source** element, which is a child element of **Local**, is used to list all data sources participating in the system and their details. **Source** element has an attribute *Name*, which specifies the name of data source. The child elements for **Source** are **Type**, **Hostname**, **Databasename**, **Userid**, **Password** and **Atom**.

The **Type** element specifies the name of DBMS used (ex. Mysql, SQL Server, Oracle, etc.). The parameters for connecting to a particular data source are specified by elements such as **Userid**, **Password**, **Hostname** and **Databasename**. The connection parameters are typically provided when registering a source to a mediator. The access information is used when connecting to a data source to extract data from relevant source relations to answer a query. The names of source relations and their attributes are specified similar to the way for the **Global** element. The schema diagram for specification of local schema in the metadata is shown in Figure 4.2.

Listing 4.2 specifies the XSD for local schema in the metadata representation. The elements **Type**, **Hostname**, **Databasename**, **Userid** and **Password** are *simple* elements as they do not have any child elements and are of type *xs:string* (Lines 23-27).

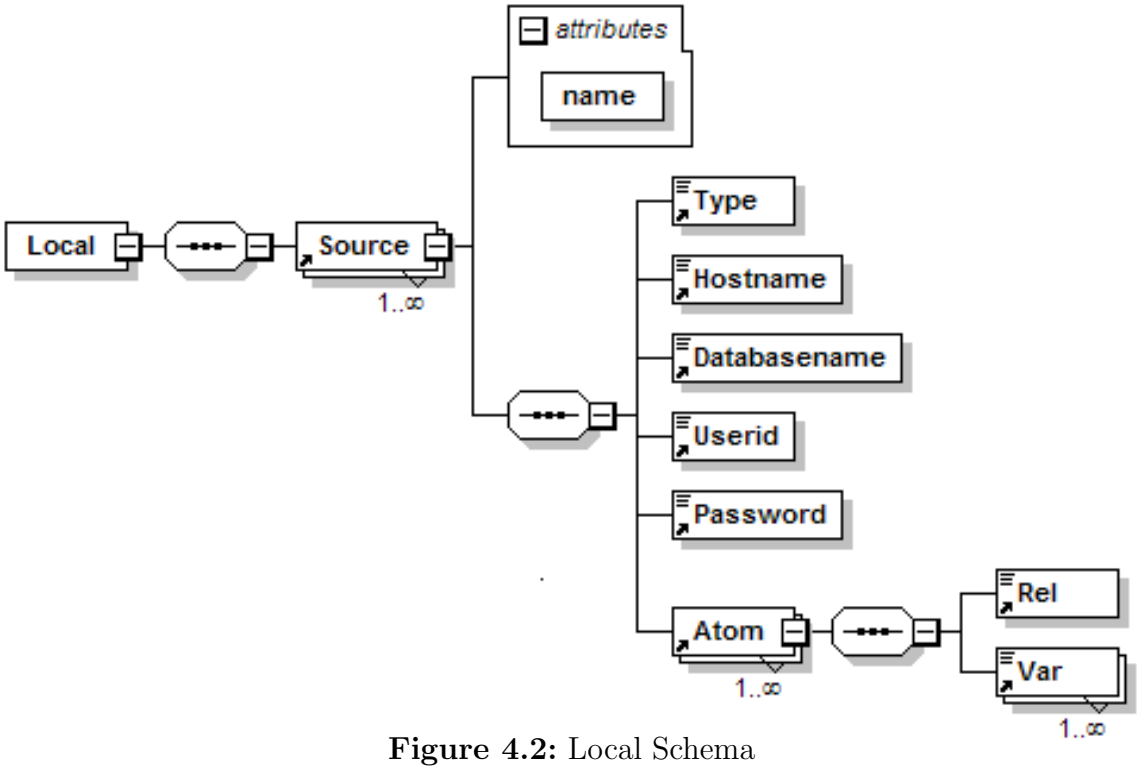


Figure 4.2: Local Schema

Listing 4.2: XMLSchema Definition for describing the Local Schema

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="Local">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element ref="Source" maxOccurs="unbounded" />
7        </xs:sequence>
8      </xs:complexType>
9    </xs:element>
10   <xs:element name="Source">
11     <xs:complexType>
12       <xs:sequence>
13         <xs:element ref="Type" />
14         <xs:element ref="Hostname" />
15         <xs:element ref="Databasename" />
16         <xs:element ref="Userid" />
17         <xs:element ref="Password" />
18         <xs:element ref="Atom" />
19       </xs:sequence>
20       <xs:attribute name="name" type="xs:string" use="required" />
21     </xs:complexType>
22   </xs:element>
23   <xs:element name="Type" type="xs:string"/>
24   <xs:element name="Hostname" type="xs:string"/>
25   <xs:element name="Databasename" type="xs:string"/>
26   <xs:element name="Userid" type="xs:string"/>
27   <xs:element name="Password" type="xs:string"/>
28 </xs:schema>
```

Example 17 The XML metadata about data sources **animalkingdom** and **animalhabitat**, structure of local relations **V1** and **V2**, and global schema consisting of *Animal, Habitat* and *Vertebrate* (cf. Example 1, Chapter 1) are specified as shown in Listing 4.3. □

Listing 4.3: Sample XML describing the Local and Global Schema

```
<VirInt xmlns="http://www.w3.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://localhost/VDI/ metadataVISS.xsd">
<Schema>
    <Local>
        <Source name="animalkingdom">
            <Type>sqlexpress</Type>
            <Hostname>animalkingdom</Hostname>
            <Databasename>animalkingdom</Databasename>
            <Userid>test</Userid>
            <Password>test</Password>
            <Atom>
                <Rel>V1</Rel>
                <Var>Name</Var>
                <Var>Class</Var>
                <Var>Food</Var>
            </Atom>
        </Source>
        <Source name="animalhabitat">
            <Type>mysql</Type>
            <Hostname>animalhabitat</Hostname>
            <Databasename>animalhabitat</Databasename>
            <Userid>test1</Userid>
            <Password>test1</Password>
            <Atom>
                <Rel>V2</Rel>
                <Var>Name</Var>
                <Var>Habitat</Var>
            </Atom>
        </Source>
    </Local>
    <Global>
        <Atom>
            <Rel>Animal</Rel>
            <Var>Name</Var>
            <Var>Class</Var>
            <Var>Food</Var>
        </Atom>
        <Atom>
            <Rel>Habitat</Rel>
            <Var>Name</Var>
            <Var>Habitat</Var>
        </Atom>
        <Atom>
            <Rel>Vertebrate</Rel>
            <Var>Name</Var>
        </Atom>
    </Global>
</Schema> </VirInt>
```

Listing 4.3 describes source **animalkingdom** as a database in *sqlexpress* (a type of Database Management System)(Line 7) and the source **animalhabitat** as a database in *mySql* (Line 20). We list access information for the data sources in **UserId** and **Password** elements (Lines 10-11). Each data source can have many source relations made available to the mediator. Therefore, there may be a sequence of **Atom** elements for a data source that are listed as child elements of **Source** element. The global relations are listed as child elements of **Global** element (Lines 33-47).

4.2 Integrity Constraints (ICs)

The data sources may adhere to their own local integrity constraints but, when integrating data from multiple sources, we may obtain data that is inconsistent wrt to some Global Integrity Constraints (GICs). The Global Integrity Constraints may be persistent, i.e. stored in the mediator or provided with the user query. If the GICs are stored in the mediator, they will have to be represented as part of the metadata. We show how GICs are specified in our metadata representation. However, the GICs are not used for detecting relevant sources. The representation of GICs provides scope to extend our technique for detecting relevant sources in the presence of GICs.[1] First, we look at functional dependencies of the form, $FD : X \rightarrow Y$, where X and Y are attributes of a global relation R. That is, each value of X is associated with only one value of Y. We express this using $IC : \neg(R(X, Y) \wedge R(X, Z) \wedge Y \neq Z)$. The schema diagram for metadata representation of this functional dependency is shown in Figure 4.3.

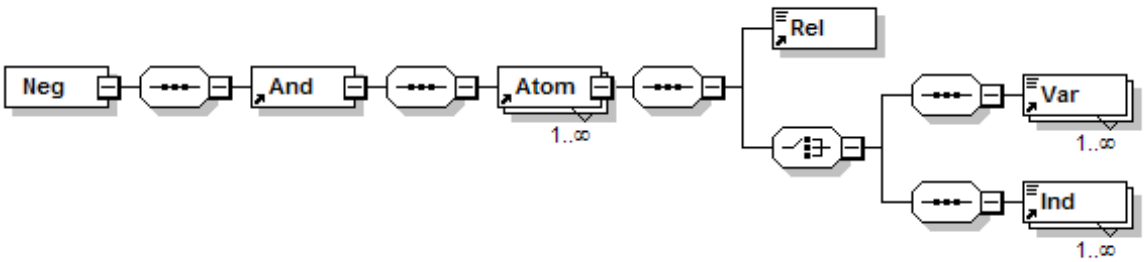


Figure 4.3: IC: Functional Dependency

[1] [24] provides a mechanism to optimize repair programs for *consistent query answering* in the presence of program constraints.

Example 18 We illustrate the violation of functional dependency when integrating data from multiple sources. We consider source relation $V1$ as in Example 17. Suppose we have a relation $V7$ in a data source that contains data about animals similar to $V1$. The extension for $V7$ is given by, $v_7 = \{$**(frog, amphibian, worms)**, (parrot, bird, nuts)$\}$. Both $V1$ and $V7$ satisfy local functional dependency: $Name \rightarrow Food$. The mapping for $V7$ is:

$$V7(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name).$$

The global data, however, does not satisfy the same functional dependency, when considered as a Global IC. This is because $V1$ contains a tuple $\{(frog, amphibian, insect)\}$. □

In cases such as Example 18, it becomes necessary to retrieve those answers that are consistent wrt GICs, at query time. Global ICs can be specified in our XML metadata representation. An example is shown in listing 4.4 for representing the functional dependency illustrated in Example 18.

Listing 4.4: Representation of ICs

```
1  <Rulebase xmlns:rule="http://www.ruleml.org/0.91/xsd"
2  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xsi:schemaLocation="http://www.ruleml.org/0.91/xsd file:datalog.xsd">
4      <Neg>
5          <And>
6              <Atom>
7                  <Rel>Animal</Rel>
8                  <Ind>X</Ind>
9                  <Var>Class</Var>
10                 <Ind>Y</Ind>
11             </Atom>
12             <Atom>
13                 <Rel>Animal</Rel>
14                 <Ind>X</Ind>
15                 <Var>Class</Var>
16                 <Ind>Z</Ind>
17             </Atom>
18         </And>
19     </Neg>
20 </Rulebase>
```

The element **Rulebase** is used to list the sequence of formulae acting as integrity constraints (Line 1). The representation shows that there cannot be two tuples in *Animal*, having values for attribute at position 3 (i.e. *Food*) (Lines 6-11) different for the same value of attribute at position 1 (i.e. *Name*) (Lines 12-17). We also

show representation of GICs, which specify dependency between relations, such as *universal* and *referential* integrity constraints [8].

Example 19 Consider integrity constraints $IC : \forall X(P(X) \rightarrow Q(X)), \forall X(Q(X) \rightarrow R(X)), \forall X(T(X) \rightarrow P(X))$. We represent the dependencies using XML representation shown in Listing 4.5. □

Listing 4.5: Representation of FDs in metadata XML

```
<Rulebase xmlns:rule="http://www.ruleml.org/0.91/xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ruleml.org/0.91/xsd file:datalog.xsd">
    <Implies>
        <head>
            <Atom>
                <Rel>Q</Rel>
                <Var>X</Var>
            </Atom>
        </head>
        <body>
            <Atom>
                <Rel>P</Rel>
                <Var>X</Var>
            </Atom>
        </body>
    </Implies>
    <Implies>
        <head>
            <Atom>
                <Rel>R</Rel>
                <Var>X</Var>
            </Atom>
        </head>
        <body>
            <Atom>
                <Rel>Q</Rel>
                <Var>X</Var>
            </Atom>
        </body>
    </Implies>
    <Implies>
        <head>
            <Atom>
                <Rel>P</Rel>
                <Var>X</Var>
            </Atom>
        </head>
        <body>
            <Atom>
                <Rel>T</Rel>
                <Var>X</Var>
            </Atom>
        </body>
    </Implies></Rulebase>
```

Here, we use RuleML elements to specify GICs. We describe RuleML specification in Section 4.3.

4.3 RuleML and Metadata Mappings

The mappings between global and local schemas in *VISS* is represented using RuleML [17]. RuleML is a markup language for representing logical rules. RuleML 0.9 introduced *First-order logic* sublanguage and its semantics is derived from classical first order logic model theory. The *First-Order Logic* RuleML is a sublanguage of *Derivation* RuleML [17]. *Datalog*, as represented in RuleML, is a sublanguage of *First-Order Logic* RuleML. The RuleML sublanguages and modules are shown in Figure 4.4. The *Datalog* sublanguage of RuleML has the required language specifications for representing view definitions under the LAV approach. In the next sections, we describe the specific modules, from *Datalog* sublanguage of RuleML, that is used for representing LAV mappings.

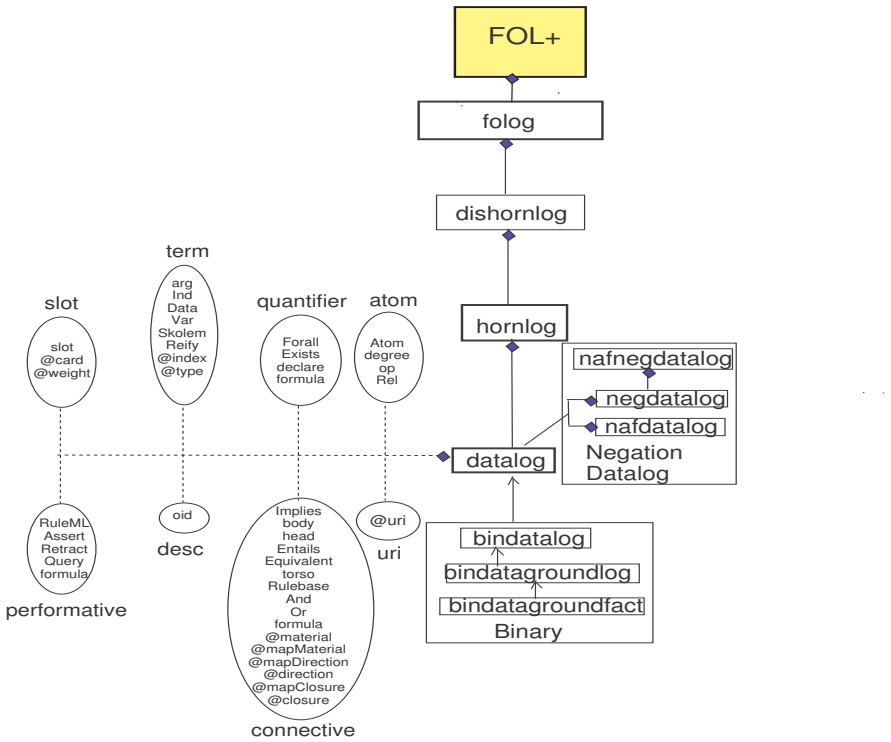


Figure 4.4: RuleML SubLanguages and Modules

4.4 RuleML Modules

We use elements from *connective*, *performative*, *term* and *atom* modules of the *Datalog* sublanguage of RuleML to represent mappings. We explain the specification of each element and how they are used in LAV mappings in the following sections.

4.4.1 Performative Module

The *performative* module [17] contains elements and attributes for RuleML performatives. A *performative* is a clause that does not evaluate to true or false. The elements in *performative* module used to represent LAV mappings are:

RuleML: This element is the n-ary top-level element of a RuleML document. **RuleML** permits ordered sets of performatives such as **Assert**.

Assert: This element acts as a wrapper specifying that its content is a formula, making an implicit Rulebase assumption. The part of the XML Schema for *performative* module that defines **RuleML** and **Assert** elements is shown in Listing 4.6.

Listing 4.6: XMLSchema Definition (XSD) for Performative Module

```
<!-- *** RuleML *** -->
<xs:element name="RuleML" type="RuleML.type"/>
<xs:complexType name="RuleML.type">
    <xs:group ref="RuleML.content"/>
</xs:complexType>
<xs:group name="RuleML.content">
    <xs:sequence>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="Assert"/>
        </xs:choice>
    </xs:sequence>
</xs:group>
<!-- *** Assert *** -->
<xs:element name="Assert" type="Assert.type"/>
<xs:complexType name="Assert.type">
</xs:complexType>
```

4.4.2 Connective Module

The *connective* module [17] contains elements and attributes for RuleML connectives. A connective is an operation on one or more atoms resulting in a compound statement with a truth value. The elements in connective module that are used in the representation of LAV mappings are:

Implies: This element denotes an implication rule. It consists of a conclusion role (head), followed by a premise role (body) or equivalently (since roles constitute unordered elements), a premise role, followed by a conclusion role.

body: This element represents the body of an implication rule denoted by element **Implies**. It contains the premise(s), also known as "antecedent" or "if" part of the rule.

head: This element represents head of an implication rule (**Implies**). It contains the conclusion, also known as "consequent" or "then" part of the rule.

And: This element represents a conjunctive expression. We use this to represent the body of a view definition. If the body of the view definition contains only one predicate, this element can be ignored. That is, `<And>Atom</And>` is equivalent to **Atom**.

The schema diagram for elements **RuleML**, **Assert**, **Implies**, **head** and **body** is given in Figure 4.5.

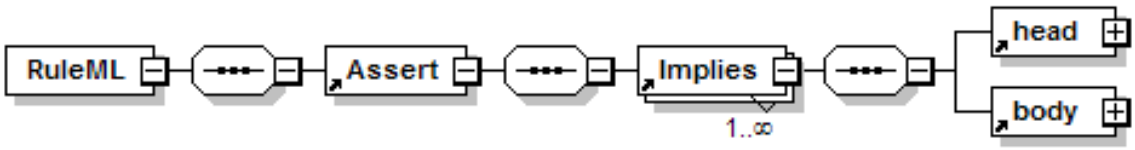


Figure 4.5: LAV Mapping using RuleML Elements

The part of the XML Schema for *connective* module that defines **Implies**, **head**, **body** and **And** elements is shown in Listing 4.7.

Listing 4.7: XMLSchema Definition (XSD) for Connective Module

```
<!-- *** Implies *** -->
<xs:element name="Implies" type="Implies.type"/>
<xs:complexType name="Implies.type">
    <xs:group ref="Implies.content"/>
</xs:complexType>
<xs:group name="Implies.content">
    <xs:sequence>
                <xs:element ref="head"/>
                <xs:element ref="body"/>
    </xs:sequence>
</xs:group>
<!-- *** body *** -->
<xs:element name="body" type="body.type"/>
<xs:complexType name="body.type">
    <xs:group ref="body.content"/>
```

```
</xs:complexType>
<xs:group name="body.content">
    <xs:choice>
        <xs:element name="Atom" type="Atom.type"/>
        <xs:element name="And" type="And-inner.type"/>
        <xs:element name="Or" type="Or-inner.type"/>
    </xs:choice>
</xs:group>
<!-- *** head *** -->
<xs:element name="head" type="head.type"/>
<xs:complexType name="head.type">
    <xs:group ref="head.content"/>
</xs:complexType>
<xs:group name="head.content">
    <xs:choice>
        <xs:element name="Atom" type="Atom.type"/>
    </xs:choice>
</xs:group>
<!-- *** And *** -->
<xs:element name="And" type="And-inner.type"/>
<xs:complexType name="And-inner.type">
    <xs:group ref="And.content"/>
</xs:complexType>
<xs:group name="And.content">
    <xs:sequence>
        <xs:element name="Atom" type="Atom.type"/>
    </xs:sequence>
</xs:group>
```

4.4.3 Atom Module

The *atom* module [17] contains elements and attributes for RuleML atoms. An *atom* is a formula, which does not contain any subformulas. The elements in atom module that are used in the representation of LAV mappings are:

Atom: This element denotes a logical atom, i.e. an expression formed from a predicate (or relation) applied to a collection of its (logical) arguments or attributes.

Rel: This element denotes a relation, i.e. a logical predicate, of an atom (**Atom**).

The part of the XML Schema for *atom* module that defines **Atom** and **Rel** elements is shown in Listing 4.8.

Listing 4.8: XMLSchema Definition (XSD) for Atom Module

```
<!-- *** Atom *** -->
<xs:element name="Atom" type="Atom.type"/>
    <xs:complexType name="Atom.type">
        <xs:group ref="Atom.content"/>
    </xs:complexType>
<xs:group name="Atom.content">
    <xs:sequence>
        <xs:element ref="Rel"/>
        <xs:choice>
            <xs:element ref="Var"/>
            <xs:element ref="Ind"/>
        </xs:choice>
    </xs:sequence>
</xs:group>
<!-- *** Rel *** -->
<xs:element name="Rel" type="Rel.type"/>
<xs:complexType name="Rel.type" mixed="true">
    <xs:group ref="Rel.content"/>
</xs:complexType>
<xs:group name="Rel.content">
    <xs:sequence/>
</xs:group>
```

4.4.4 Term Module

The *term* module [17] contains elements and attributes for RuleML terms. A term is part of a predicate and refers to variables or constant. The elements in term module that are used in the representation of LAV mappings are:

Var: This element denotes a logical variable, as in logic programming. It takes the optional attribute *@type*, which specifies a term's (user-defined) type.

Ind: This element denotes an individual constant, as in predicate logic. It takes the optional attribute *@type*. The attribute *@type* specifies a term's (user-defined) type.

The schema diagram for elements representing the head and body of a view definition is given in Figure 4.6 and 4.7.

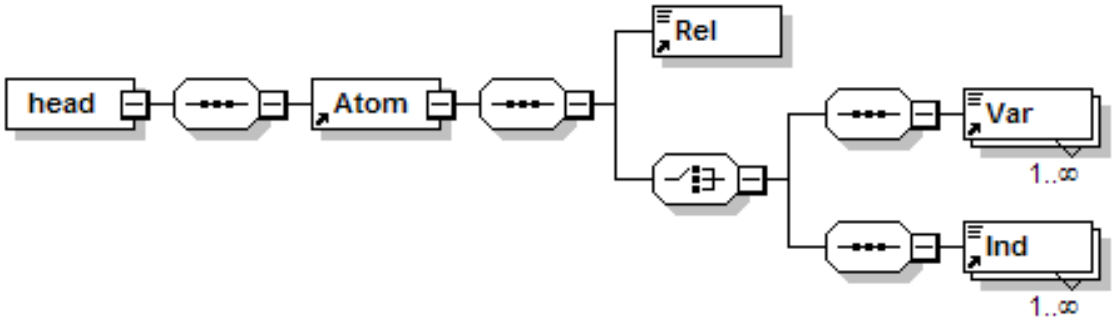


Figure 4.6: Head of Mapping using RuleML Elements

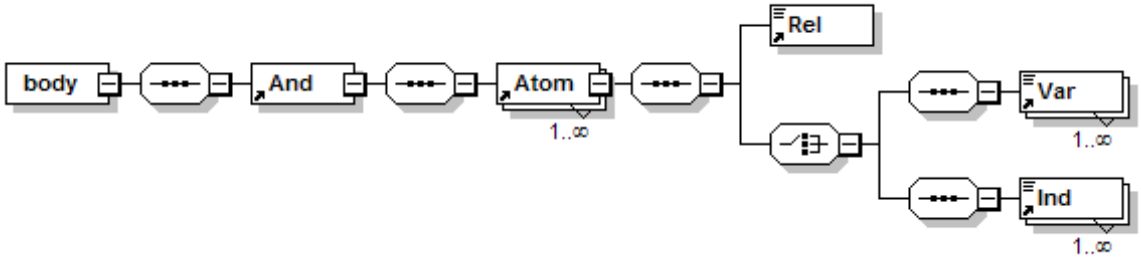


Figure 4.7: Body of Mapping using RuleML Elements

The part of the XML Schema for *term* module that defines **Var** and **Ind** elements is shown in Listing 4.9.

Listing 4.9: XMLSchema Definition (XSD) for Term Module

```
<!-- *** Var *** -->
<xs:element name="Var" type="Var.type"/>
<xs:complexType name="Var.type" mixed="true">
    <xs:group ref="Var.content"/>
    <xs:attributeGroup ref="Var.attlist"/>
</xs:complexType>
<xs:attributeGroup name="Var.attlist">
    <xs:attributeGroup ref="type.attrib"/>
</xs:attributeGroup>
<xs:group name="Var.content">
    <xs:sequence/>
</xs:group>
<!-- *** Ind *** -->
<xs:element name="Ind" type="Ind.type"/>
<xs:complexType name="Ind.type" mixed="true">
    <xs:group ref="Ind.content"/>
    <xs:attributeGroup ref="Ind.attlist"/>
</xs:complexType>
<xs:attributeGroup name="Ind.attlist">
    <xs:attributeGroup ref="type.attrib"/>
</xs:attributeGroup>
<xs:group name="Ind.content">
    <xs:sequence/>
</xs:group>
```

4.4.5 Representing Mappings using RuleML

We illustrate the use of RuleML elements for representing mappings under the LAV approach using Example 20.

Example 20 (Example 1 continued) We use the view definitions of source relations **V1** and **V5**. These source relations are described as views of the global schema, consisting of *Animal* and *Vertebrate* as follows:

$$\mathbf{V1}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (4.1)$$

$$\mathbf{V5}(Name, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (4.2)$$

We represent the above mappings using RuleML specification as shown in Listing 4.10. □

Listing 4.10: Representation of LAV Mappings using RuleML

```
<RuleML xmlns:rule="http://www.ruleml.org/0.91/xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ruleml.org/0.91/xsd
file:datalog.xsd">
<Assert>
    <Implies>
        <head>
            <Atom>
                <Rel>V1</Rel>
                <Var>Name</Var>
                <Var>Class</Var>
                <Var>Food</Var>
            </Atom>
        </head>
        <body>
            <And>
                <Atom>
                    <Rel>Animal</Rel>
                    <Var>Name</Var>
                    <Var>Class</Var>
                    <Var>Food</Var>
                </Atom>
                <Atom>
                    <Rel>Vertebrate</Rel>
                    <Var>Name</Var>
                </Atom>
            </And>
        </body>
    </Implies>
    <Implies>
        <head>
            <Atom>
                <Rel>V5</Rel>
```

```
34                      <Var>Name</Var>
35                      <Var>Food</Var>
36                  </Atom>
37              </head>
38              <body>
39                  <And>
40                      <Atom>
41                          <Rel>Animal</Rel>
42                          <Var>Name</Var>
43                          <Ind>mammal</Ind>
44                          <Var>Food</Var>
45                      </Atom>
46                  </And>
47              </body>
48          </Implies>
49  </Assert>
50  </RuleML>
```

Lines 1-4 show that, the representation follows the XML Schema definition of *Datalog* sublanguage of RuleML and uses elements from *http://www.ruleml.org/0.91/xsd* namespace. As per the specification of *Datalog* sublanguage of RuleML, the database predicates are represented using **Rel** (Line 9, 18), their attributes are represented using **Var** (Lines 10-12) and the built-in values using **Ind** (Line 43). **And** indicates conjunction of predicates in the body of the view definition (Line 16). **Implies** describes the view definition is an implication (Line 6), and hence, contains **head** and **body** elements. **Assert** opens and closes the list of view definitions (Line 5).

RuleML specifies the representation of built-in conditions as shown in Listing 4.10. Here, the attribute at position 2 for relation *Animal* (which is *Class*), has a built-in value represented by element **Ind**. The value of the built-in is shown as equal to "*mammal*". There is another way of representing the built-in value, using RuleML specification. We show this for the condition $Class =$ "*mammal*" as follows:

```
<Atom>
    <Rel>Class</Rel>
    <Ind>mammal</Ind>
</Atom>
```

Here, the built-in condition is specified using an object-oriented approach using attributes as objects (i.e. **Rel**). We follow the approach in Listing 4.10 as this is more straight-forward.

Chapter 5

Optimizing the EIRA

In this chapter, we describe the theory behind our pruning approach to detect relevant sources and query the required data from these sources, which can be used with the logic program from *Simple Specification* (cf. Section 2.4 in Chapter 2). We discuss how the pruning approach is actually implemented in *VISS* when querying the metadata in Chapter 6. The specification of a subclass of legal instances of the integration system is used to compute *certain* answers to *monotone* queries. We refer to the specification as *Simple Specification*. We describe how we can prune rules for some global predicates in the logic program obtained, based on a reduced list of sources. We also show how sources are queried based on built-ins in the query. We start by defining some of the terms used in this chapter.

5.1 Definitions

A source relation V is defined in terms of global relations in LAV as:

$$V(X) \leftarrow P_1(X_1), P_2(X_2), .., P_k(X_k), C_v. \tag{5.1}$$

Here, the body is a conjunction of global relations $P_i(X_i), i = 1, .., k$ and built-in conditions C_v. We assume that, the rules are safe, i.e. all variables occurring in the head of the rule are also present in the body of the rule. We also assume that, if a variable is part of a built-in condition, it also appears as part of a predicate in that

rule. A join variable is one that occurs in more than one predicate in the body of the rule. A query, $Q(\bar{X})$ is given as:

$$Q(\bar{X}) \leftarrow P_1(X_1), P_2(X_2), .., P_n(X_n), C_q. \tag{5.2}$$

Here, $P_i, i = 1, .., n$, are global relations and $X_i, i = 1, .., n$, are variables in the body of the query. The body of the query is a conjunction of atoms on global relations. C_q is a conjunction of built-in conditions, namely, any of $=, \leq, \geq$ and $\neq$. We define source relevance for answering a query using dependency graph [24].

Definition 1 A source predicate V is *relevant* to obtain *certain* answers to a *Datalog* query Q given by Equation (5.2), if there exists a mapping given by Equation (5.1) such that, P_i is in the antecedent of the mapping and V appears in the consequent of the mapping and P_i appears in Q.

Example 21 We consider the global relations *Animal(Name, Class, Food)*, *Habitat(Name, Habitat)* and the source relations $V1$, $V2$ in our running example given as:

$$V1(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \tag{5.3}$$

$$V2(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \tag{5.4}$$

We consider the *Datalog* query:

$$Ans(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \tag{5.5}$$

$V1$ is the consequent of the mapping whose antecedent contains the global predicate *Animal* and $V2$ is the consequent of the mapping whose antecedent contains the global predicates *Animal* and $Habitat$. Both *Animal* and $Habitat$ appear in the body of the query $Ans(Name, Habitat)$. Using Definition 1, we get $V1$ and $V2$ as required sources. □

Table 5.1: Optimization Steps

Ref. Name	**Description**
QP	(1) Retain global predicate in the query that are joined by atleast one variable **or** contains a variable that appears in the head of the query.
SP	(1) Identify list of source relations whose view definitions have atleast one global predicate in the body same as the predicates in the body of the query. (2) Identify view definitions whose head contains atleast one variable that is in the head of the query **or** contains a variable that is one of the join variables in the body of the query. (3) Identify view definitions whose body (if it contains equality built-ins) does not violate any of the equality built-ins in the query.
SQC	(1) Apply the built-in conditions in the query to the appropriate variables in the source relations when querying to retrieve data from the source.
RP	(1) Generate logic program for *Simple Specification* using the reduced list of sources and retain the rules for the global predicates in the body of the query. (2) Prune redundant rules from the logic program from *Simple Specification*.

Our optimization approach starts by analyzing the query and pruning the query, if possible. We refer to this step as Query Pruning (QP). The required view definitions are determined by looking at the global relations in the query and checking certain conditions. We refer to this step as Source Pruning (SP). We then generate statements

to query source relations from the SP step and apply built-in conditions from the query. This step is referred as Source Query Condition (SQC). The logic program from *simple specification* is generated for the source relations obtained from the SP step and rules are further pruned, if possible. This step is referred as Rules Pruning (RP). The steps and their criteria are listed in Table 5.1. We explain each step in detail in the subsequent sections.

5.2 Query Pruning

We show how to reduce the query itself, where possible, as a first step in our optimization process. Pruning the query involves removing those global predicates from the body of the query, that are not involved in the computation of answers to the query.

Algorithm 1 Query Pruning

Require: $Q(\bar{X})$,V_i

```
1:  procedure QueryPruning(RelSource)
2:      RelSource = 0
3:      RelPred = false
4:      PredHead = List of predicates in query body with var appearing in X̄
5:      JoinVar = List of variables in PredHead
6:      for each predicate P_i in body of Q(X̄) do
7:          for each variable X_j in P_i do
8:              if X_j ∈ JoinVar then
9:                  RelPred = true
10:             else if X_j ∈ X̄ then
11:                 RelPred = true
12:             end if
13:         end for
14:         if RelPred = false then
15:             Remove P_i from body of Q(X̄)
16:         end if
17:         RelPred = false
18:     end for
19:     return RelSource
20: end procedure
```

Algorithm 1 verifies that both the following conditions holds true for pruning the global predicates in the query:

(a) The global predicate in the body of the query does not contain a join variable(s) with a global predicate that has variables appearing in the head of the query (Lines 7-9).

(b) The global predicate in the body of the query does not contain a variable that appears in the head of the query (Lines 10-13).

The first condition ensures that global predicates in the query program, which share a join variable with a global predicate that provides tuples in the answer for the query, are considered. If the first condition fails, the second condition checks if the variables in the global predicate are in the head of the query. These conditions are effective in queries involving cartesian products, where only a subset of the predicates in the body of the query actually participate in providing answers.

Example 22 Suppose the mediator contains only the following mappings:

$$V1(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (5.6)$$

$$ClassV1(Class) \leftarrow ClassList(Class). \quad (5.7)$$

$$HabitatV1(Habitat) \leftarrow Environment(Habitat). \quad (5.8)$$

$$WaterAvail(Habitat, Avail) \leftarrow Water(Habitat, Avail). \quad (5.9)$$

Consider a *Datalog* query, $\Pi(Q)$, to get the name of animals and the class, where the animals belong:

$$\begin{aligned} Ans(Name, Class) \quad \leftarrow \quad & Animal(Name, Class, Food), ClassList(Class), \\ & Environment(Habitat), Water(Habitat, Avail). \end{aligned} \quad (5.10)$$

The global predicates *Animal* and *ClassList* are joined by the variable *Class* and contain atleast one of the attributes $Name$ or $Class$, which appear in the head of the query. But, the global predicate *Environment* does not have a join variable with *Animal* or *ClassList*. Also, the head of the query does not contain variable *Habitat*. Hence, *Environment* can be pruned from the query. The global predicate *Water* can

also be pruned for the same reasons, even though it contains a join variable with *Environment*. So, we now have global predicates *Animal* and *ClassList* to get the required sources.

$$Ans(Name, Class) \leftarrow Animal(Name, Class, Food), ClassList(Class). \quad (5.11)$$

Conditions, such as the predicates *Water* and *Environment* in Equation (5.10), are not common; and it does not make much sense to use them in practice. □

If t is a tuple obtained for the query in Equation (5.10), then the same tuple t is obtained from Equation (5.11), after pruning the global predicates.

Consider a conjunctive query without built-ins given by: $Q(\bar{X}) \leftarrow \bigwedge_{i=1}^{n} P_i(X_i), \bigwedge_{j=1}^{m} R_j(Y_j)$, where, P_i, R_j are predicates, X_i, Y_j are tuples of variables/constants and $\bar{X} \subseteq X_i$. If the predicates R_j have non-empty extensions and do not have a join variable with any of the predicates P_i (i.e. Y_j and X_i are disjoint) and the predicates R_j do not provide tuples in the head of the query (i.e. $\bar{X}$ and Y_j are disjoint), then the pruned query $Q'(\bar{X}) \leftarrow \bigwedge_{i=1}^{n} P_i(X_i)$ produces the same set of tuples as Q.

5.3 Source Pruning

We now detect the required source relations based on the information available in the view definitions. Our approach does not provide a minimal set of sources but gives a reduced set, based on certain conditions. When there are built-in conditions in the query, we consider only the equality conditions. Identifying relevant source relations based on operators, such as $\leq, \geq$ and $\neq$ in the query becomes more complex as shown in Example 23.

Example 23 Suppose we have a condition $x \geq c_1$ in the query, where x is a variable in the global relation g_i and c_1 is a constant. The query is given as: $Q(x) \leftarrow g_i(x), x \geq c_1$. If there is a source relation V_i, defined in terms of g_i with a built-in condition $x \geq c_2$ as: $V_i(x) \leftarrow g_i(x), x \geq c_2$, where c_2 is a constant. If $c_1 < c_2$, all tuples in V_i

become relevant to the query. But, if $c_1 > c_2$, V_i may still contain *some* relevant tuples to provide answers for the query. Hence, we will have to consider source relation V_i as required in both cases. □

For ease of description, we assume that for the same global predicate in the query and view definition, the variable names for the global relation in the query and the variable names for the global relation in the view definition are the same. In Algorithm 1, we also get a list of relevant source relations when we check if the predicate in the body of the query appears in the body of a view definition for a source relation (cf. Line 8 in Algorithm 1).

Algorithm 2 Source Pruning

Require: $Q(\bar{X})$, RS
1: **procedure** SOURCEPRUNING($RS, RelSrc$)
2: $RelSrc = 0$
3: $JoinVar = List\ of\ join\ variables\ in\ the\ body\ of\ Q(\bar{X})$
4: $HeadVar = List\ of\ variables\ in\ the\ head\ of\ Q(\bar{X})$
5: **for** each variable X_k in predicate V_i in RS **do**
6: **if** $X_k \in HeadVar$ **then**
7: $RelSrc \leftarrow V_i$
8: exit loop
9: **else if** $X_k \in JoinVar$ **then**
10: $RelSrc \leftarrow V_i$
11: exit loop
12: **end if**
13: **end for**
14: **for** each predicate V_i in $RelSrc$ **do**
15: **for** each equality built-in $C_q i$ in $Q(\bar{X})$ **do**
16: **if** $C_v i$ is true and $C_v i$ satisfies $C_q i$ **then**
17: **else if** $C_v i$ is true and $C_v i$ not satisfy $C_q i$ **then**
18: Remove V_i from $RelSrc$
19: exit loop
20: **end if**
21: **end for**
22: **end for**
23: **return** $RelSrc$
24: **end procedure**

Algorithm 2 uses the list of sources obtained as *RelSource* from Algorithm 1 and applies the following conditions to further reduce the number of source relations:

(a) If the head of a view definition contains a variable from the head of the query or a join variable in the query, then that source is selected.

(b) Using the equality built-ins in the body of the query, and from the list of sources, *RelSrc*, obtained from (a), we check if the body of the view definition contains a equality condition on the same variable as the query built-in. If it does, we check if the value of the built-in violates the equality condition in the query. If this is the case, we remove that source relation from *RelSrc*. The sources that do not have a built-in condition or do not violate the equality built-in condition are retained in the list. We illustrate these steps using the following example.

Example 24 We consider the global relations *Animal(Name, Class, Food)*, *Habitat(Name, Habitat)* and the source relations $V1$, $V2$, $V3$, $V4$, $V5$ in our running example and a source relation $V6(Class)$ whose extension is "mammal", "bird", "amphibian". The mappings are given as:

$$V1(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (5.12)$$

$$V2(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \quad (5.13)$$

$$V3(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (5.14)$$

$$V4(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "bird". \quad (5.15)$$

$$V5(Name, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (5.16)$$

$$V6(Class) \leftarrow Animal(Name, Class, Food). \quad (5.17)$$

We consider the *Datalog* query:

$$\begin{aligned} Ans(Name, Habitat) \quad \leftarrow \quad & Animal(Name, Class, Food), Habitat(Name, Habitat), \\ & Class = "mammal". \end{aligned} \quad (5.18)$$

From Algorithm 1, we get the list of sources, $V1$, $V2$, $V3$, $V4$, $V5$ and $V6$, to consider for pruning. The variables in the head of the query are $Name, Habitat$ and the join variable is $Name$. Using these, we look at the view definitions and select $V1$, $V2$, $V3$, $V4$ and $V5$. We eliminate $V6$ because it does not contain the head or join variables in

the query. Next, we check if the built-ins in the query are used in the view definition for $V3$, $V4$ and $V5$. But, $V4$ violates the equality condition and hence, we eliminate this relation. The reduced set of sources obtained are: $V1$, $V2$, $V3$ and $V5$. □

In some cases, the view definitions that contain variables in the head, which are all the variables in the head of the query is sufficient to compute the answers. In Example 24, if we do not have the built-in $Class = "mammal"$, then just $V2$ is sufficient to answer the query.

5.4 Source Query Condition

We showed how we obtain a reduced list of sources based on the conditions in the query and view definitions. We use the sources from the SP step to retrieve data used as facts in the logic program for computing *certain* answers. However, retrieving all the data, when we require only a subset of data is inefficient. Hence, the next step in our optimization is to apply the built-in conditions from the query to the sources to retrieve the appropriate data.

Algorithm 3 Source Query

Require: $Q(\bar{X})$, $RelSrc$

```
1: procedure SourceQuery(RelSrc, Q)
2:     BltVar = List of built − in variables in the body of Q(X̄)
3:     for each predicate V_i in RelSrc do
4:         for each variable X_k in predicate V_i do
5:             if X_k ∈ BltVar then
6:                 Concatenate data retrieval query for V_i with condition for X_k
7:             end if
8:         end for
9:     end for
10: end procedure
```

First, we look at source relations that contain the built-in variables from the query. For these source relations, we apply built-in conditions from the query for the appropriate variables. The source relations that do not contain the built-in variables, are queried for all the data. In this step, we consider all built-in operators, viz.

$=, \geq, \leq, >, <$ and $\neq$ that are present in the query. We show the steps in Algorithm 3.

Example 25 We use the list of sources obtained in Example 24, viz. $V1$, $V2$, $V3$ and $V5$. The query in Equation (5.18) in Example 24, contains a built-in $Class =$ "$mammal$". The source relations $V1$ and $V3$ contain variable $Class$. Hence, the query statements generated for the source relations are:

$$
\begin{aligned}
&Select\ distinct\ *\ from\ V1\ where\ Class = "mammal"\\
&Select\ distinct\ *\ from\ V2\\
&Select\ distinct\ *\ from\ V3\ where\ Class = "mammal"\\
&Select\ distinct\ *\ from\ V5
\end{aligned}
$$

$V3$ uses the condition $Class =$ "$mammal$" even though it contains only animals that are mammals. In this case, incorporating this condition does not make a difference in data retrieval. □

The queries obtained in Example 25, uses the clause *distinct* to retrieve non-redundant tuples from a relation.

5.5 Rule Pruning

The reduced list of sources obtained from SP step, is used to shortlist the mappings, required to generate the *Simple Specification* (cf. Section 2.4 in Chapter 2). From the list of mappings, we remove those global predicates that do not appear in the body of the query. We then generate the logic program for the *Simple Specification* (cf. Section 2.4 in Chapter 2) with some modifications as follows:

1. The facts: We use query commands with built-ins, obtained from SQC step, for every source in $RelSrc$. As a result, we do not generate the logic program with domain constants and program facts, which means listing all the data in the source relations in the logic program. Instead, we use *import* commands that can be run in DLV [52], a disjunctive *Datalog* system, which directly brings the required data from the source relations into main memory, when running the logic program. Thus, we eliminate the use of $V(\bar{a})$ in the *Simple Specification*.

2. For every view (source) predicate V in the system with definition, $V(\bar{X}) \leftarrow P_1(\bar{X}_1), ..., P_n(\bar{X}_n)$, the rules:

$$P_j(\bar{X}_j) \leftarrow V(\bar{X}), \bigwedge_{X_i \in (\bar{X}_j \setminus \bar{X})} \mathbf{choice_i}(\bar{X}, X_i),\ j = 1, ...n.$$

Here, instead of having two rules, one mapping to auxiliary predicate F_i, which then maps to the *choice* operator, we replace the auxiliary predicate F_i with the *choice* operator directly.

The RP step and the whole logic program is illustrated in Example 26.

Example 26 (Example 25 continued) We use the mappings for source relations $V1$, $V2$, $V3$ and $V5$ that are identified as relevant for answering the query in Equation (5.18). The mapping for $V1$ contains the global relation $Vertebrate$ in the body, which does not appear in the body of the query. Hence, the rule corresponding to this global relation is not considered. We generate the logic program for *Simple Specification* using view definitions for $V1, V2, V3$ and $V5$ as follows:

$$\begin{aligned}
Animal(Name, Class, Food) &\leftarrow V1(Name, Class, Food).\\
Animal(Name, Class, Food) &\leftarrow V2(Name, Habitat),\\
&\quad chosen1(Name, Habitat, Class),\\
&\quad chosen2(Name, Habitat, Food).\\
chosen1(Name, Habitat, Class) &\leftarrow V2(Name, Habitat), dom(Class),\\
&\quad not\ diffchoice1(Name, Habitat, Class).\\
diffchoice1(Name, Habitat, Class) &\leftarrow chosen1(Name, Habitat, U),\\
&\quad dom(Class), U \neq Class.\\
chosen2(Name, Habitat, Food) &\leftarrow V2(Name, Habitat), dom(Food),\\
&\quad not\ diffchoice2(Name, Habitat, Food).\\
diffchoice2(Name, Habitat, Food) &\leftarrow chosen2(Name, Habitat, U),\\
&\quad dom(Food), U \neq Food.\\
Habitat(Name, Habitat) &\leftarrow V2(Name, Habitat).\\
Animal(Name, \text{"}mammal\text{"}, Food) &\leftarrow V3(Name, Class, Food).\\
Animal(Name, \text{"}mammal\text{"}, Food) &\leftarrow V5(Name, Food),\\
&\quad chosen3(Name, Food, Class).
\end{aligned}$$

$$
\begin{aligned}
\mathit{chosen}3(Name, Food, Class) \quad &\leftarrow \quad V5(Name, Food), dom(Class), \\
&\qquad\quad \mathit{not}\ \mathit{diffchoice}3(Name, Food, Class). \\
\mathit{diffchoice}3(Name, Food, Class) \quad &\leftarrow \quad \mathit{chosen}3(Name, Food, U), \\
&\qquad\quad dom(Class), U \neq Class.
\end{aligned}
$$

□

The *dom* predicates in the logic program are defined by additional rules as illustrated in Example 27.

5.6 The Domain Predicate

The $dom(a)$ atoms are obtained from rules that define *dom* predicates using the *relevant* source predicates. To do this, the existential variables are identified from the view definitions of the *relevant* source relations. The source predicates whose extensions contain values for the existential variables are used for defining the *dom* predicate. The source predicates are loaded with values from the *relevant* data sources using *import* commands.

Example 27 (Example 26 continued) From the *relevant* source relations identified for answering the query in Equation (5.18), $V2$ and $V5$ have existential variables $Class$ and $Food$ in their view definitions (cf. Equation (5.13) and (5.16)). The source relations $V1$, $V3$ and $V5$ contain values for the attributes $Class$ and $Food$. Hence, the rules for *dom* predicates are defined as follows:

$$
\begin{aligned}
dom(Class) &\leftarrow V1(\_, Class, \_). \\
dom(Food) &\leftarrow V1(\_, \_, Food). \\
dom(Class) &\leftarrow V3(\_, Class, \_). \\
dom(Food) &\leftarrow V3(\_, \_, Food). \\
dom(Food) &\leftarrow V5(\_, Food).
\end{aligned}
$$

The *import* command for a source relation loads data into the corresponding source predicate when running the logic program in DLV. The *import* command for $V1$ is as follows:

$$\#import \quad (animalkingdom, "test", "test", "Select\ Distinct\ *\ From\ V1 \\ where\ Class\ =\ 'mammal'", \mathbf{V1}, type : Q\_Const, Q\_Const, \\ Q\_Const).$$

The *import* command loads the result of the query statement against the source relation $V1$ in the datasource *animalkingdom* into the predicate in the logic program ($V1$ in bold). We use facts from the *relevant* source relations in the active domain and this reduces the number of ground atoms in the stable models. The "_" in the body of the rules is used to mask the other attributes, which do not appear anywhere else in the same rule [52]. We combine the rules for the *dom* predicates with the logic program in Example 26, the query commands obtained in Example 25 and the query in Equation (5.18) in Example 24, to compute *certain* answers for Equation (5.18).□

5.7 Disjunctive Queries

We consider disjunction of conjunctive queries in the presence of equality built-ins. The query may also contain other type of built-in operators such as $\geq$, $\leq$ and $\neq$. The query is given as:

$$Q(\bar{X}) \leftarrow \bigwedge_{i=1}^{n} P_i(\bar{Y}), C_j, C_k.$$
$$Q(\bar{X}) \leftarrow \bigwedge_{l=1}^{m} R_l(\bar{Z}), C_j, C_t.$$

Here, C_j is a conjunction of built-in conditions using equality operator and C_l, C_t are conjunction of built-in conditions using $\geq$ and $\leq$. We consider each conjunctive query at a time and use the query pruning (QP) and source pruning (SP) steps (cf. Section 5.2 and 5.3).

Example 28 Consider a disjunctive query that asks for animals belonging to class

mammal and those belonging to class bird,

$$Ans(Name, Food) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat), Class = "mammal". \quad (5.19)$$

$$Ans(Name, Food) \leftarrow Animal(Name, Class, Food), Class = "bird". \quad (5.20)$$

In this case, none of the global predicates in the query is pruned in the QP step. Using SP step, we get $V1, V2, V3, V5$ as the list of required sources for Equation (5.17) and $V1, V2, V4$ as the list of required sources for Equation (5.18). In the SQC step, we get the following query commands for the sources:

$$SELECT\ Distinct\ *\ FROM\ V1\ where\ Class = "mammal".$$
$$SELECT\ Distinct\ *\ FROM\ V3\ where\ Class = "mammal".$$
$$SELECT\ Distinct\ *\ FROM\ V2.$$
$$SELECT\ Distinct\ *\ FROM\ V5.$$
$$SELECT\ Distinct\ *\ FROM\ V1\ where\ Class = "bird".$$
$$SELECT\ Distinct\ *\ FROM\ V4\ where\ Class = "bird".$$

The rules for view definitions $V1, V2, V3, V4, V5$ are generated using *Simple Specification* in the RP step. Combining and running the query commands, rules and query program in DLV, gives *certain* answers for the disjunctive query. □

When a disjunctive query is posed in terms of the global relations in *VISS*, the metadata is queried to identify the relevant source relations and retrieve data from those source relations. The *import* commands containing query statements, such as the *Select* clauses shown in Example 28, are generated. We describe the functions in *VISS* that are used to identify relevant sources and explain how they are applied for disjunctive queries in Chapter 6 (cf. Remark 1 at the end of Section 6.3).

Chapter 6

Extracting Relevant Information using XQuery

This chapter describes XQuery FLWOR expressions and how they are used to query XML metadata following the pruning steps. When computing *certain* answers to a query, the XML metadata is queried to get the relevant information using XQuery, a W3C recommended query language for XML documents. After extracting relevant information, the logic program $\Pi(\mathcal{G})$ for *Simple Specification* is generated. Chapter 5 described the theory about the pruning approach. In this chapter, we describe how we actually implement the pruning approach in *VISS* using XQuery.

6.1 XQuery FLWOR Expressions

We use a particular class of queries expressed in XQuery called *FLWOR* expressions [28], which are of the form:

```
FOR <var> IN <expr>
    LET <var> := <expr>
    WHERE <expr>
    ORDER BY <expr>
RETURN <expr>.
```

The term *FLWOR* stands for the keywords FOR, LET, WHERE, ORDER and RETURN. The FOR clause iterates over a sequence of atomic values, such as numbers or XML elements and returns a set of values or XML elements as specified in the RETURN clause [28]. The FOR clause consists of one or more bound variables

prefixed with *$*. W3C defines the binding sequence of a variable in the FOR clause as the value of the expression associated with the variable in that clause [28]. *FLWOR* expressions can be nested to describe various types of joins. An example is shown below:

```
FOR $i IN VirInt/Schema/Local,
    $j IN VirInt/RuleML/Assert/Implies/head
    WHERE $i/Rel=$j/Rel
RETURN $i/Rel.
```

Here, i and j are the variables bound to the XML elements `VirInt/Schema/Local` and `VirInt/RuleML/Assert/Implies/head` respectively and the (equi) join is based on the value of element *Rel*.

LET clause contains one or more variables and each variable is associated with an expression [28]. Unlike a FOR clause, a LET clause binds each variable to the result of the equated expression without iteration. The variable in a FOR and LET clause covers all subexpressions in the inner *FLWOR* expressions that appear after the variable binding [28]. We show an example using LET clause below:

```
LET $i := VirInt/Schema/Local
RETURN $i/Rel.
```

The WHERE clause specifies the condition for filtering the result of the FOR clause. The condition in the WHERE clause is evaluated once for each value or element retrieved in the FOR clause. If the expression in the WHERE clause evaluates to true, then the value or element is retained and its corresponding variables are used to evaluate the result of the RETURN clause [28]. If the expression evaluates to false, the value is discarded. The WHERE clause in a *FLWOR* expression is optional [28]. We show an example using WHERE clause below:

```
FOR $i IN VirInt/Schema/Local,
    $j IN VirInt/RuleML/Assert/Implies/head
    WHERE $i/Rel=$j/Rel
RETURN $i/Rel/text().
```

An ORDER BY clause contains one or more ordering specifications called *orderspecs* [28]. For each value or element in the set of values obtained after filtering in the WHERE clause, the *orderspecs* are evaluated using the values in the result. The

Function Name	Description
string-join	Returns a string by concatenating the strings in the arguments and uses an optional argument as a separator.
insert-before	Returns a new sequence constructed from the list of item arguments, with the new value argument inserted in the position specified by the position argument.
remove	Returns a new sequence constructed from the list of item arguments, with the value specified by the position argument removed from the sequence.
tokenize	Splits a string into a list of strings using the character argument specified as a separator.
replace	Returns a string by replacing a pattern in a given string with the replace argument.
concat	Returns a string by concatenating the string arguments.
count	Returns the count of nodes.
distinct-values	Returns distinct values of nodes or atomic values.
codepoints-to-string	Converts Unicode code point to string.
index-of	Returns the positions of a search item within a list of items.
data	Returns a sequence of atomic values corresponding to a sequence of items.
Following-sibling	Returns a list of all nodes that have the same parent node as the context node and appear after the context node.
Preceding-sibling	Returns a list of all nodes that have the same parent node as the context node and appear before the context node.

Table 6.1: XQuery Functions

relative order of two atomic values is determined by comparing the values of their *orderspecs*, working from left to right until a pair of unequal values is encountered. The ORDER BY clause is optional and if it is not specified, the order of the result set is determined by the FOR and LET clauses and by ordering mode [28]. In nested *FLWOR* expressions, ordering can be applied at multiple levels of an element hierarchy. We show an example using ORDER BY clause below:

```
FOR $i IN VirInt/Schema/Local
ORDER BY $i/Rel descending
RETURN $i/Rel/text().
```

The RETURN clause specifies the projected attributes in the result. The RETURN clause of a *FLWOR* expression is evaluated once for each value in the result of the WHERE and ORDER BY clauses, and the output of these evaluations are combined to form the result of the *FLWOR* expression [28].

XQuery supports conditional expressions using keywords IF, THEN and ELSE as shown below:

```
IF <expr> THEN <expr> ELSE <expr>
```

The expression following IF clause is called a test expression and the result of a conditional expression is determined based on the evaluation of the test expression to true or false. If the test expression evaluates to true, the value of the THEN expression is returned and if not, the value of the ELSE expression is returned [28]. The main functions in XQuery [28], that are used when querying the metadata are shown in Table 6.1.

6.2 Identifying Relevant Sources

For a given query $\Pi(Q)$, only a relevant portion of available view definitions is required. In particular, we require ones whose body contains the global predicates that appear in $\Pi(Q)$. We are not considering GICs as they are not used to compute *certain* answers. The detection of relevant source relations becomes more complicated in the presence of GICs as discussed later in Section 6.5. We use the pruning techniques listed in Chapter 5 to detect relevant sources. We assume the query $\Pi(Q)$ is already pruned using QP step (cf. Section 5.2 in Chapter 5). We now use SP step (cf. Section 5.3 in Chapter 5) and show how relevant source relations can be determined using queries in XQuery. For this, we use the metadata representation described in Chapter 4 as input. First we identify source relations, whose view definitions contain the global relations in $\Pi(Q)$. Next, we apply the criteria in SP step (cf. Section 5.3 in Chapter 5) to further reduce the list of relevant source relations.

Example 29 Consider the following query:

$$Ans(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat), Class = "mammal". \quad (6.1)$$

From the body of the query, we get global predicates *Animal, Habitat*. After applying SP, we obtain **V1**, **V3**, **V5** and **V2** as relevant source relations for this query. □

Listing 6.1: Step 1:Retrieving relevant sources

```
declare function local:getAllSource($g as element()*) as element()*
{
for $f in doc("mappings.xml")/VirInt/RuleML/Assert/Implies
    where ((some $x in $g/preceding-sibling::Var/@Attr
    satisfies $x=$f/head/Atom/Rel/following-sibling::Var ) or
    (some $y in $g/preceding-sibling::Join/@Attr
    satisfies $y=$f/head/Atom/Rel/following-sibling::Var )) return
        <sources>
        {
        for $c in $f, $a in $g/Atom
            where $a/@val = $c/body/And/Atom/Rel/text()
            return
                <source so='{$f/head/Atom/Rel}'>
                {
                for $d in $f/body/And/Atom/Rel
                for $e in $d
                    return $e[text()=$d]/following-sibling::*
                }
                </source>
        }
        </sources>
};

declare function local:gSrcIndx($e as element()*, $g as element()*)
as element()*
{
for $a in $e/source
    where some $b in
    distinct-values($g/Atom/Var[@Op="eq"]/@Attr)
    satisfies $a/Var[text() = distinct-values($b)]
    return
    <source s='{$a/@so}'>
    {
    for $c in
    distinct-values($g/Atom/Var[@Op="eq"]/@Attr)
        return
        <Indx var='{distinct-values($a/Var[text()=$c])}'>
        {
        min(index-of($a/Var/text(),$c))
        }
        </Indx>
    }
    </source>
};
```

Step 1: We describe the user-defined functions used to retrieve *relevant* source relations for a user query. First, we obtain the list of source relations whose view definitions have variables in the head that also appear in the head of the query or are join variables in the body of the query (Lines 4-7 in Listing 6.1) (cf. Section 5.3 in Chapter 5 for source pruning). We define a function *getAllSource* to retrieve the source relations and the values for the variables that appear in equality conditions in the body of the view definition (Lines 1-22). Using this list, we determine the position of the variables in the global predicate that are involved in the equality conditions. We use function *gSrcIndx* to get the position of the variables (Lines 24-44, Listing 6.1).

Example 30 (Example 29 continued) We use the query in Equation (6.1) and the view definitions in Example 1 in Chapter 1. In this case, functions *getAllSource* and *gSrcIndx* gives the following output:

```
<!-- fn:getAllSource Output -->
<source so="V1">
    <Var>Name</Var>
    <Var>Class</Var>
    <Var>Food</Var>
    <Var>Name</Var>
</source>
<source so="V3">
    <Var>Name</Var>
    <Ind>mammal</Ind>
    <Var>Food</Var>
</source>
<source so="V4">
    <Var>Name</Var>
    <Ind>bird</Ind>
    <Var>Food</Var>
</source>
<source so="V5">
    <Var>Name</Var>
    <Ind>mammal</Ind>
    <Var>Food</Var>
</source>
<source so="V2">
    <Var>Name</Var>
    <Var>Class</Var>
    <Var>Food</Var>
    <Var>Name</Var>
    <Var>Habitat</Var>
</source>

<!-- fn:gSrcIndx Output -->
<source s="V1">
    <Indx var="Class">2</Indx>
</source>
<source s="V3">
    <Indx var="Class">2</Indx>
</source>
<source s="V4">
    <Indx var="Class">2</Indx>
</source>
<source s="V5">
    <Indx var="Class">2</Indx>
</source>
<source s="V2">
    <Indx var="Class">2</Indx>
</source>
```

□

Step 2: The XML output obtained from functions *getAllSource* and *gSrcIndx*, shown in Example 30, is used as input in function *gSrcComp* (Lines 1-18, Listing 6.2) to check, if the variables in the equality condition in the body of the query is also present in a equality condition in the body of the view definition. If it does,

the function *gSrcComp* generates a result containing the source relation name, the variable name and its value. The function *gRelWOCond* (Lines 20-30, Listing 6.2) gives a list of source relations having view definitions whose head does not have the variables appearing in any of the built-ins in the query.

Listing 6.2: Step 2: Retrieving relevant sources

```
1  declare function local:gSrcComp($f as element()*, $g as element()*)
2  as element()*
3  {
4  for $b in $f/source, $c in $g
5      where $c/@s=$b/@so
6      return
7          <src s = '{$b/@so}'>
8          {for $d in $b
9              return
10                 for $e in $c/Indx
11                 return
12                     <Var var='{$e/@var}'
13                     ind='{$d/child::*[position()=$e/text()]
14                     [name() != "Var"]/text()}' >
15                     </Var>
16         }
17         </src>
18 };
19
20 declare function local:gRelWOCond($f as element()*,
21 $g as element()*, $x as item()*)
22 {
23 for $b in $f/source,
24     $d in doc("mappings.xml")/VirInt/RuleML/Assert/Implies/head/Atom
25     where $d/Rel/text() = $b/@so
26     and $d/Rel/text()[not(. = $x)]
27     and (every $c in $d/Rel/following-sibling::Var/text()
28     satisfies $c[not(. = $g/Atom/Var/@Attr )])
29     return distinct-values($b/@so)
30 };
```

Example 31 (Example 30 continued) Using the XML result shown in Example 30 as input, the function *gSrcComp* gives the following output:

```
<!-- fn:gSrcComp output -->
<src s="V3">
    <Var var="Class" ind="mammal"/>
</src>
<src s="V4">
    <Var var="Class" ind="bird"/>
</src>
<src s="V5">
    <Var var="Class" ind="mammal"/>
</src>
```

The function *gRelWOCond* returns $V2$ and $V5$ as they do not contain the built-in variable, $Class$ in the head of the view definition. □

Step 3: The XML output of *gSrcComp* shown in Example 31 is used in function *gIRelSrc* (Lines 1-16 in Listing 6.3) to get the source relations, which violate any of the equality built-ins in the query. For the query in Equation (6.1) (cf. Example 29), the function returns $V4$ as this source contains only birds, thus violating the equality condition $Class =$ "*mammal*". We retrieve the remaining list of source relations, those that satisfy the equality built-ins in the query using the function *gRelSrc* (Lines 18-36 in Listing 6.3). The function returns **V1**, **V2**, **V3** and **V5**, which are the required sources for the query in Equation (6.1).

Listing 6.3: Step 3: Retrieving relevant sources

```
1  declare function local:gIRelSrc($f as element()*, $g as element()*)
2  {
3  for $b in $f/Var
4  let $c := $g/Atom/Var
5      where
6      (distinct-values($b/@ind) != "" and
7      not(starts-with(distinct-values($b/@ind), "ge ")) and
8      not(starts-with(distinct-values($b/@ind), "le ")) and
9      not(starts-with(distinct-values($b/@ind), "lt ")) and
10     not(starts-with(distinct-values($b/@ind), "gt ")) and
11     not(starts-with(distinct-values($b/@ind), "ne ")) and
12     distinct-values($b/@ind) !=
13     distinct-values($c[@Op="eq"][@Attr=$b/@var]/@Ind) )
14     return $b/parent::*/@s
15 };
16
17 declare function local:gRelSrc($f as item()*, $g as element()*)
18 {
19 (for $b  in local:getAllSource($g)/source
20     where $b/@so[not(. = $f)]
21     return $b/@so
22 )
23 union
24 (
25 for $a in local:gSrcComp
26  (local:getAllSource($g),local:gSrcIndx(
27  local:getSrcAttr($g), $g))
28     where
29     distinct-values($a/Var/@ind) =
30     distinct-values($g/Atom/Var[@Op="eq"][@Attr=$a/Var/@var]/@Ind)
31     return $a/@s
32 )
33 };
```

6.3 Querying Relevant Sources

We describe how user-defined functions in XQuery are used in the SQC step (cf. Section 5.4 in Chapter 5 for querying relevant sources). Once the relevant source relations have been detected, they are queried to retrieve the required data. We apply the built-ins in the query to the source relations, where applicable. For relevant source relations, whose attributes do not appear in the built-ins in the query, all tuples are retrieved from the source relation. We use a function *gRelCmd* (cf. Listing 6.4) to generate the query commands to the source relations.

Listing 6.4: Query to generate *import* commands

```
1  declare function local:gRelCmd($f as item()*,
2  $g as element()*) as element()*
3  {
4  for $a in doc("mappings.xml")/VirInt/Schema/Local/Source/Atom
5  let $c := $g
6  let $j := local:gSrcWOAC()
7      where $a/Rel/text()[(. = $f)] and
8      $a/Rel/text()[not(. = $j)]
9      return
10         <clause>
11             <clause1>
12             {concat('#import(', data($a/../@name), ', \"',
13             data($a/../Userid),'\", \"', data($a/../Password),'\",
14             \"SELECT Distinct * FROM ', data($a/Rel))}
15             </clause1>
16             <clause11>
17             {if
18             (count($a/Rel/following-sibling::Var[text()=
19
20             $c/Atom/Var/@Attr]/text()) > 0)
21             then
22                 (' where ')
23             else()
24             }
25             </clause11>
26             <clause2>
27             {
28             distinct-values(
29             for $x in $c/Atom/Var
30             for  $b in $a/Rel/following-sibling::Var
31                 return
32                     if ((count($b[text()=$x/@Attr]/text()) > 0) and
33                     ($b[text()=$x/@Attr]/text() != ''))
34                     then
35                         (if ($b/text()=$x[@Op="le"]/@Attr)
36                         then
37                         (concat($b[text()=$x[@Op="le"]/@Attr]/text(),
38                         ' <= '
39                         ,distinct-values($x[@Attr=$b/text()]
40                         [@Op="le"]/@Ind),''',''))
```

```
41                          else if ($b/text()=$x[@Op="ge"]/@Attr)
42                          then
43                          (concat($b[text()=$x[@Op="ge"]/@Attr]/text(),'
44                          >= ''' ,distinct-values($x[@Attr=$b/text()]
45                          [@Op="ge"]/@Ind),''',','))
46                          else if ($b/text()=$x[@Op="eq"]/@Attr)
47                          then
48                          (concat($b[text()=$x[@Op="eq"]/@Attr]/text(),'
49                          = ''' ,distinct-values($x[@Attr=$b/text()]
50                          [@Op="eq"]/@Ind),''',','))
51                          else()
52                          )
53                    else())
54              }
55              </clause2>
56              <clause3>
57              {
58              concat('\", ',data($a/Rel),', type : ',
59              string-join(insert-before(remove(tokenize(replace
60              (string-join($a/Rel/following-sibling::Var,',
61              '),'.+?, ','Q_CONST, '),',','),
62              count($a/Rel/following-sibling::*)),
63              count($a/Rel/following-sibling::*),
64              'Q_CONST'),', '),').', codepoints-to-string(10))}
65              </clause3>
66          </clause>
67 };
```

Listing 4.3 shows the function *gRelCmd*, which first extracts connection information, i.e. *hostname*, *databasename*, *userid*, for the relevant sources obtained in Section 6.2 (Lines 11-15). The metadata representation of sources (cf. Section 4.1 in Chapter 4) is used as input to get this information. The query commands generated are SQL statements that retrieves all non-redundant tuples from the source relations (Line 14). These SQL statements are concatenated with the built-in conditions from the body of the query (Lines 26-55). The output of the function are *import* commands that extract from the data sources, the facts used in the logic program.

The *import* commands directly retrieves the database facts and can be run in DLV [52]. They are of the form: #*import*(*databasename*,"*username*", "*password*","*query*", *predname*, *typeConv*), where *databasename, username, password* are read from the XML metadata. *query* is an SQL statement that retrieves data from the source relation, *predname* defines the name of the predicate that will be used, and *typeConv* specifies the conversion for mapping DBMS data types to *Datalog* data types for each column.

Example 32 We obtain $V1$, $V2$, $V3$ and $V5$ as required sources for query in Equation (6.1) (cf. Example 29). The function *gRelCmd* uses the XML metadata in Listing 4.3 (cf. Section 4.1 in Chapter 4) as input and generates the *import* commands for the source relations as follows:

$$
\begin{aligned}
&\#import \quad (animalkingdom, "test", "test", "Select\ Distinct\ *\ From\ V1 \\
&\qquad where\ Class\ =\ 'mammal'", V1, type : Q\_Const, Q\_Const, \\
&\qquad\qquad Q\_Const). \\
&\#import \quad (animalkingdom, "test", "test", "Select\ Distinct\ *\ From\ V3 \\
&\qquad where\ Class\ =\ 'mammal'", V3, type : Q\_Const, Q\_Const, \\
&\qquad\qquad Q\_Const). \\
&\#import \quad (animalhabitat, "test1", "test1", "Select\ Distinct\ *\ From\ V2", \\
&\qquad\qquad V2, type : Q\_Const, Q\_Const). \\
&\#import \quad (animalhabitat, "test1", "test1", "Select\ Distinct\ *\ From\ V5", \\
&\qquad\qquad V5, type : Q\_Const, Q\_Const). \\
&dom(u).
\end{aligned}
$$

Here, Q_Const is a conversion type specifying that the column is converted to a string with quotes. This can be used generally for all data types. □

An extra atom $dom(u)$, where u is a constant not appearing in any source, is generated with the *import* commands in Example 32 because even though we need a finite domain to run the logic programs, we still have to capture the potential infiniteness of the domain and the openness of the sources [19]. In the case of conjunctive queries, this one extra constant is sufficient to compute *certain* answers [19].

Remark 1 In the case of disjunctive queries, for each conjunctive query in the disjunction, the user-defined function *gRelSrc* (cf. Listing 6.3 in Section 6.2) returns the relevant sources. The *import* commands for the source relations (cf. Example 28 in Chapter 5) are generated using the *gRelCmd* function (cf. Listing 6.4). The relevant source relations are used to generate the required portions of the logic program for computing certain answers.

6.4 Identifying Rules

We now identify the required parts of the logic program for *Simple Specification* (cf. Section 2.4 in Chapter 2). We describe how we use a query in XQuery to perform the initial part of the RP step (cf. Section 5.5 in Chapter 5). We described how to generate *import* commands in the preceding section. These commands are used in the logic program of *Simple Specification* instead of listing all the facts explicitly in the program. Next, mappings are selected based on the source relations obtained in SP step (cf. Section 5.3 in Chapter 5). A query in XQuery is presented in Listing 6.5, which is used to extract the required mappings from the metadata representation (cf. Section 4.3 in Chapter 4).

Listing 6.5: Querying XML and RuleML metadata using XQuery

```
1  for $n in doc("mappings.xml")/VirInt
2      return
3  (: List all rules :)
4          <rules>
5          {for $x in $n/RuleML/Assert/Implies
6              where distinct-values($x/body/And/Atom/Rel) =
7              (<<List of global relations in the query>>)
8              and $x/head/Atom/Rel=(<<List of sources from SP step>>)
9              return
10                 <rule>
11                 (: Describe the rule :)
12                 {for $d in $x
13                     return
14                         <head r1='{$d/head/Atom/Rel}'>
15                         {for $v in $x/head
16                             where $v/Atom/Rel=$d/head/Atom/Rel
17                             return
18                             concat($d/head/Atom/Rel,'(',
19                             string-join($v/Atom/Var,',',')')
20                         }
21                         </head>
22                 }
23                 {for $b in distinct-values($x/body/And/Atom/Rel)
24                     where $b = (<<List of global relations
25                     in the query>>)
26                     return
27                     (: Describe the body of the rule :)
28                         <body r1='{$b}'>
29                         {for $m in $n/Schema/Global/Atom
30                             where $m/Rel=$b
31                             RETURN
32                             concat($b,'(', string-join(
33                             $m/Rel/following-sibling::Var,',',')')
34                         }
35                         </body>
36                 }
```

```
37                  (: Describe built-ins in the rule :)
38                  <body r1=''>
39                  {distinct-values
40                  (for $q in $x/body/And/Atom/Ind
41                  let $l as item()* := index-of($x/body/And/Atom/*,$q)
42                  let $r := $q/preceding-sibling::Rel/text()
43                  for $y in $l
44                  for $s in
45                  $n/Schema/Global/Atom/Rel[text()=$r]/../Var[$y - 1]
46                      return
47                          if ($q != '') then
48                          (
49                          (string-join(('(',$s,'=',$q,'),'),''))
50                          )
51                          else ()
52                  )
53                  }
54                  </body>
55                  </rule>
56          }
57          </rules>
```

The query in Listing 6.5 does the following:

1. Obtain an XML output containing the mappings corresponding to the source relations from the SP step (cf. Section 5.3 in Chapter 5) and whose body contains the global relations in the user query (Lines 5-21).

2. From the body of the mappings, use only the global relations that appear in the body of the query (Lines 23-35) in the XML output.

3. The built-ins in the view definitions, if present, are also generated in the XML output (Lines 38-54).

Example 33 We obtained $V1$, $V2$, $V3$, $V5$ as required sources for the query in Equation (6.1) (cf. Example 29). The global relations in the query are *Animal* and *Habitat*. Using the query in Listing 6.5, we query metadata mappings in Listing 4.10 (cf. Section 4.4.5 in Chapter 4). The output obtained in XML format is shown in Figure 6.1. □

In the XML output in Figure 6.1, the body for $V1$ does not contain $Vertebrate$ even though its view definition does (cf. Equation (4.1) in Example 20 in Chapter 4). This is because, *Vertebrate* does not appear in the query in Equation (6.1) and hence, we prune it when generating the program for *Simple Specification* . The XML result in Figure 6.1 is processed by the *Program Builder*, a *VISS* component that generates the

```
<rules>
    <rule>
        <head r1="V1">V1(Name,Class,Food)</head>
        <body r1="Animal">Animal(Name,Class,Food)</body>
        <body r1=""/>
    </rule>
    <rule>
        <head r1="V3">V3(Name,Class,Food)</head>
        <body r1="Animal">Animal(Name,Class,Food)</body>
        <body r1="">(Class=mammal),</body>
    </rule>
    <rule>
        <head r1="V5">V5(Name,Food)</head>
        <body r1="Animal">Animal(Name,Class,Food)</body>
        <body r1="">(Class=mammal),</body>
    </rule>
    <rule>
        <head r1="V2">V2(Name,Habitat)</head>
        <body r1="Animal">Animal(Name,Class,Food)</body>
        <body r1="Habitat">Habitat(Name,Habitat)</body>
        <body r1=""/>
    </rule>
</rules>
```

Figure 6.1: Pruned Rules

Simple Specification program. We describe the components in Chapter 7 and building the logic program in Chapter 8.

6.5 Dependency Graph

We describe an idea for determining relevant predicates in the presence of Global Integrity Constraints (GICs) using user-defined functions in XQuery[1]. In the presence of GICs, the relevant predicates for answering a query can be determined from the relationship between the predicates in the query and the predicates in the ICs. We analyze this with the help of a dependency graph [24]. In a directed dependency graph, each database predicate is a node. There is an edge from a predicate P_i to P_j,

[1]We do not consider GICs in the mediator system *VISS* described in this thesis and hence consistent query answering is not implemented in *VISS*

iff there exists a constraint such that P_i is the antecedent of the rule and P_j appears in the consequent of the rule. A predicate P is required to obtain *certain* or *consistent* answers to a *Datalog* query Q in the presence of GICs, if P is in a connected component c of the dependency graph for the IC and there is a predicate P' appearing in Q and c [24]. We show how the presence of GICs affect the determination of relevant sources required for answering a query using Example 34.

Example 34 Consider the GICs $IC : \forall X(P(X) \rightarrow Q(X)), \forall X(Q(X) \rightarrow R(X)), \forall X(R(X) \rightarrow S(X)), \forall X(T(X) \rightarrow P(X)), \forall X(R(X) \rightarrow W(X))$. For a query $Ans(X) \leftarrow R(X)$, apart from the global relation R in the body of the query, we have to consider the global relations in the GICs whose antecedent is R and apply the same process recursively for each global relation obtained in the result. Here, the relevant global predicates required to answer the query are $\{R, S, W\}$. When we determine the source relations required for answering the query, we have to take into account the view definitions that contain the global predicate in the query (i.e. R) and the global predicates based on the dependencies (i.e. S, W). □

Listing 6.6: Query to obtain relevant predicates based on ICs

```
1  declare function local:relPred($e as  element()*) as element()*
2  {
3  let $a := local:getPred($e)
4      return $a | $a/local:relPred(.)
5  };
6  declare function local:getPred($b as element()*) as element()*
7  {
8  for $n in doc("mappings.xml")/VirInt/Schema/Global/Rulebase/Implies
9      where $n/body/Atom/Rel = $b
10     return $n/head/Atom/Rel
11 };
```

The XML metadata representation described in Chapter 4 shows Listing 4.5 for representing GICs in Example 34. In Listing 6.6, we show the functions *relPred* and *getPred* that uses the XML metadata representation of GICs to obtain the relevant global predicates. The function *relPred* (Listing 6.6 Line 1) takes the global relation in the body of the query as argument and calls the function *getPred* (Line 3). The function *getPred* returns the predicate consequent for the predicate antecedent passed as argument using the XML metadata (Lines 9-11). We do this recursively for each global relation (Line 4) to obtain the relevant global predicates for the query

$Ans(X) \leftarrow R(X)$ based on the GICs. For the query, $Ans(X) \leftarrow R(X)$, *getPred* takes the global predicate in the query, namely R, as an argument and computes the dependencies from the XML metadata representation of ICs (cf. Listing 4.5 in Chapter 4) and returns the relevant global predicates $\{R, S, W\}$.

Chapter 7

Architecture of VISS

This chapter describes the general architecture of *VISS* , a mediator system for integrating relational data sources or sources that can be wrapped as relational. We also present the implementation architecture of *VISS* and explain each component.

7.1 General Architecture

When a *Datalog* query is posed to the mediator *VISS*, the latter analyzes the query and determines the source relations required to answer it. Then, the metadata is queried for the access information of those relations and import commands are generated to read tuples from the source relations and store them as facts in the logic program. These facts form the extensional database used by the *Simple Specification* program, which becomes the query plan. The *Simple Specification* program is run in a logic-based system and the result of the program evaluation is the set of *certain* answers to the *Datalog* query. We assume that all data sources are relational or wrapped as relational using appropriate wrappers. We show the general architecture of *VISS* in Figure 7.1.

7.1.1 User Query

VISS computes *certain* answers to *monotone* queries with built-ins. *Monotone* queries are characterized by *Datalog* programs without negation. Some examples of *monotone* queries are union (disjunctive), cartesian product and conjunctive queries. Queries

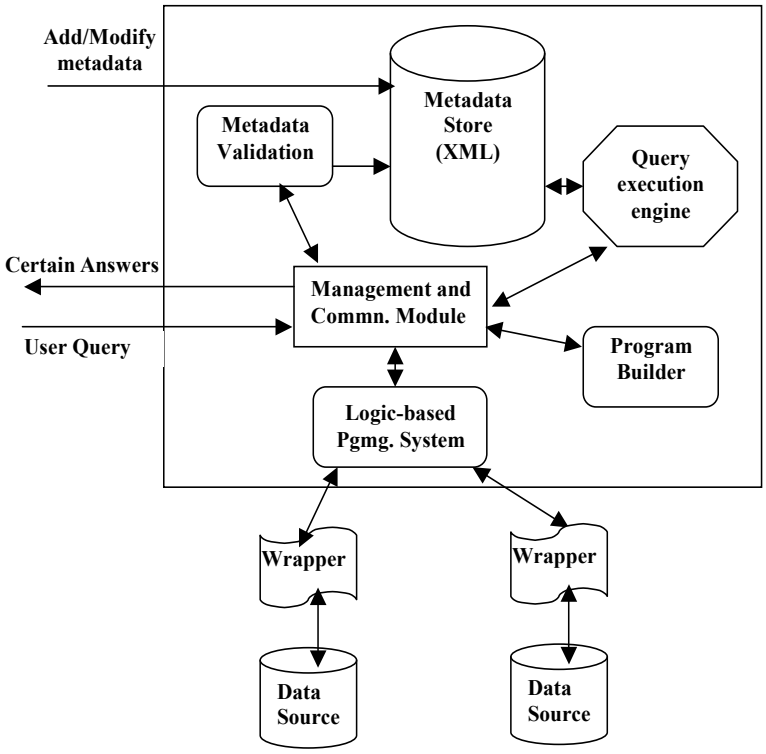


Figure 7.1: General Architecture of VISS

can also be recursive. The query is posed in terms of the predicates in the global schema.

Example 35 A user query for listing all animals that live in "$ocean$" or "$forest$" is given by the disjunctive query:

$$\begin{aligned} Ans(Name, Habitat) &\leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat), \\ &\quad Habitat = "ocean". \\ Ans(Name, Habitat) &\leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat), \\ &\quad Habitat = "forest". \end{aligned}$$

$Ans(Name, Habitat)$?

7.1.2 Management and Communication Module

The *Management and Communication Module* (MCM) acts as the central component of the mediator, which invokes or calls the other components. The main functionality of MCM is to manage the execution of other components of the mediator and initiate

calls to the appropriate components at different stages of computation of answers. For validating the XML metadata, the MCM invokes the *Metadata Validator* and if the validation fails, MCM stops further execution and sends an error message. The MCM also initiates calls to the *Query Execution Engine* for executing a query in XQuery against the XML metadata. Once the required information, such as the *import* commands and mappings are gathered, MCM uses the *Program Builder* to build the *Simple Specification* program. MCM, then combines the *import* commands, logic program for the *Simple Specification* and the user query and runs them, under skeptical stable model semantics, using a *Logic-based Programming System*.

7.1.3 Metadata Validation

The metadata in *VISS* is stored in XML format, which contains custom XML elements and RuleML elements. We described the XML Schema Definitions (XSDs) for this metadata in Chapter 4. The *Metadata Validator* verifies that the XML metadata in *VISS* adheres to the schema specifications (cf. Chapter 4). If the validation succeeds, a success message is sent to MCM. If the validation fails, an error file is generated listing the errors and a failure message is sent to MCM.

7.1.4 Metadata Store

The *Metadata Store* contains the XML metadata (cf. Chapter 4), and makes it available for access by other components. The metadata consists of the global schema, local schema, access information of sources and GICs, all using custom XML elements and RuleML elements. The *Metadata Store* provides static, persistent storage of metadata but it does not contain actual data from the sources. Typically, the metadata is created and updated in the *Metadata Store* using a metadata interface, which allows a designer to enter the metadata information. The metadata information is entered in *Datalog* format, which is translated by the interface to XML and RuleML.

7.1.5 Query Execution Engine

The *Metadata Store* is accessed by the *Query Execution Engine* to determine the relevant source relations by querying the XML metadata. The execution engine can process queries in XQuery language. The *Query Execution Engine* also generates the necessary *import* commands and sends the output to the MCM. The queries run by the *Query Execution Engine* performs read operations on the metadata but does not update the metadata.

7.1.6 Program Builder

The *Program Builder* is invoked by the MCM after the relevant sources are determined by querying the metadata. The *Program Builder* builds the logic program for *Simple Specification* based on the XML output obtained in the RP step (cf. Figure 6.1 in Chapter 6). The *Program Builder* also prunes the redundant rules as specified in the RP step (cf. Section 5.5 in Chapter 5), by replacing the function predicate F_i with predicate $chosen_i$ directly. The output of *Program Builder* is the program for *Simple Specification* .

7.1.7 Logic-based Programming System

The MCM combines the logic program output of the *Program Builder*, the *import* commands generated to query the source relations and the user query. The combined program is run in a *Logic-based Programming System* (LPS) for deductive databases that evaluates the complete program and generates *certain* answers to user query. The *Logic-based Programming System* uses the *Datalog* language with negation. The LPS also interfaces with the wrappers for the data sources to pull the required facts into the logic program for evaluation.

7.1.8 Wrappers and Data Sources

The data retrieved from the source relations are provided to the mediator in relational format using wrappers. The format in which the data is presented to the mediator

may be different from the actual format of data in the data source. The wrappers perform the required transformation to relational format that is understood by the mediator. We do not delve into the transformation process in the wrapper. We assume that the data sources have suitable wrappers that export the data from the source to the mediator with the right structure.

7.2 Implementation Architecture

The *VISS* system is implemented in C++. *VISS* uses Oracle's Berkeley DB XML (BDBXML) [73], an open source XML database, for storing the XML metadata related to a global schema. The implementation architecture of *VISS* is shown in Figure 7.2. We describe the main tools used to implement the components of the general architecture of *VISS* .

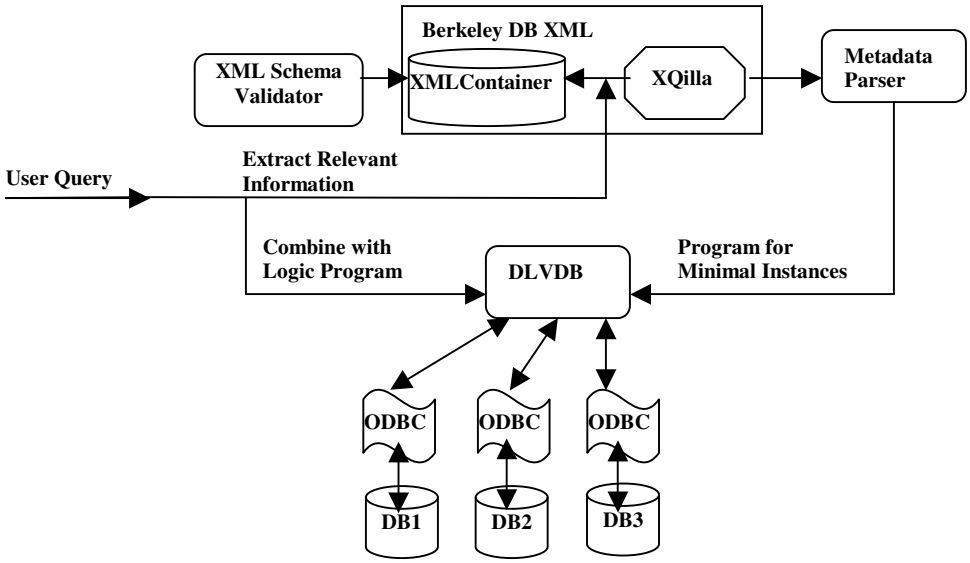


Figure 7.2: Implementation of VISS

7.2.1 XMLSchema Validator

In *VISS* , the metadata is an XML document containing custom XML elements and RuleML elements (cf. Chapter 4). The metadata document is validated using the XML Schema Validator, (*XSV*), recommended by W3C [75]. *XSV* checks the metadata with the schema definitions described in Chapter 4. The output from *XSV* on

successful validation of the metadata XML document is shown in Listing 7.1.

Listing 7.1: Sample output from XML Schema Validator in VISS

```
1 <xsv xmlns="http://www.w3.org/2000/05/xsv" docElt="{None}VirInt"
2       instanceAssessed="true" instanceErrors="0" schemaErrors="0"
3       target="file:///mappings.xml" validation="lax"
4 </xsv>
```

Line 2 in Listing 7.1 shows that there are no errors after validating the metadata. The term *instanceErrors* indicates errors in the overall structure of the metadata document and *schemaErrors* indicates violation of the XML Schema Definitions (listed in Chapter 4).

7.2.2 Berkeley DB XML

Oracle Berkeley DB XML 2.4.16 (BDBXML) [73] is an open source, embeddable XML database for storing XML documents. BDBXML is a library of C++ APIs and is supported on a very large number of platforms. BDBXML includes an XML Document Manager and an XQuery engine. A set of application programming interfaces (APIs) allow querying and retrieval of data from XML documents. The main BDBXML objects for managing XML documents are *XMLManager*, a high-level class for managing container; and *XMLContainer*, which is used for storing XML documents. A container is a collection of XML documents and information about those documents [73]. BDBXML uses XQilla 2.0, an open source XQuery execution engine that conforms to the XQuery W3C standard [28]. The results of a query are obtained in the form of an *XMLResults* object, a resultset containing XML elements or atomic values, which can be iterated over to retrieve each value in the set.

VISS uses BDBXML as the *Metadata Store* and the metadata is stored in a *XMLContainer*. The BDBXML APIs are used to call the *Query Execution Engine* and other components of *VISS* and hence, BDBXML also acts as the *Management and Communication Module*. Queries in the mediator *VISS* are executed using XQilla.

7.2.3 Metadata Interface

The *Metadata Interface* is implemented using the approach of POSL translator [7], which translates POSL to RuleML. The *Metadata Interface*[1] accepts mappings in *Datalog* format as input and translates them to RuleML elements (as specified in Chapter 4 Section 4.3). The interface first prompts the user to specify the number of data sources as shown in Figure 7.3. The value from this user input is used to generate the access information in XML format for the metadata.

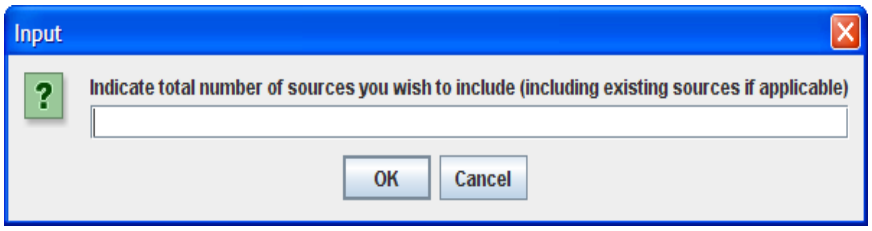


Figure 7.3: Specifying Number of Sources

After specifying the number of data sources, the XML metadata (specifications are described in Chapter 4) is generated as shown in Figure 7.4.

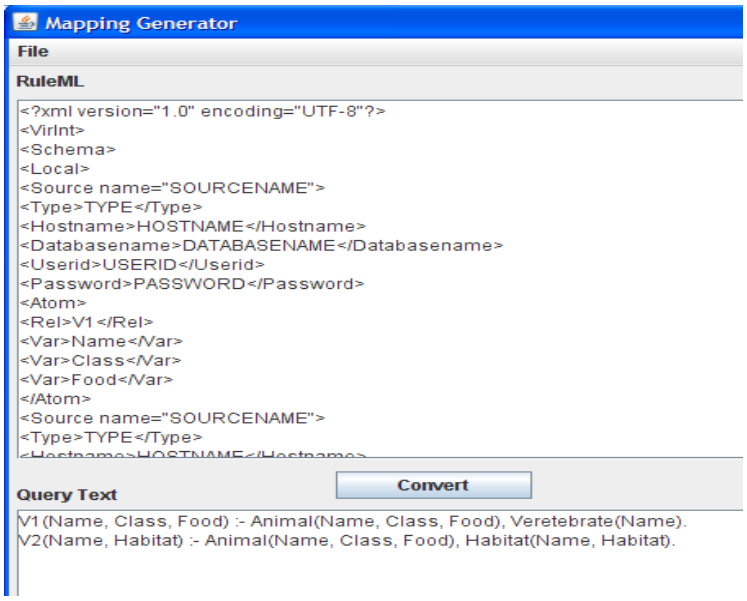
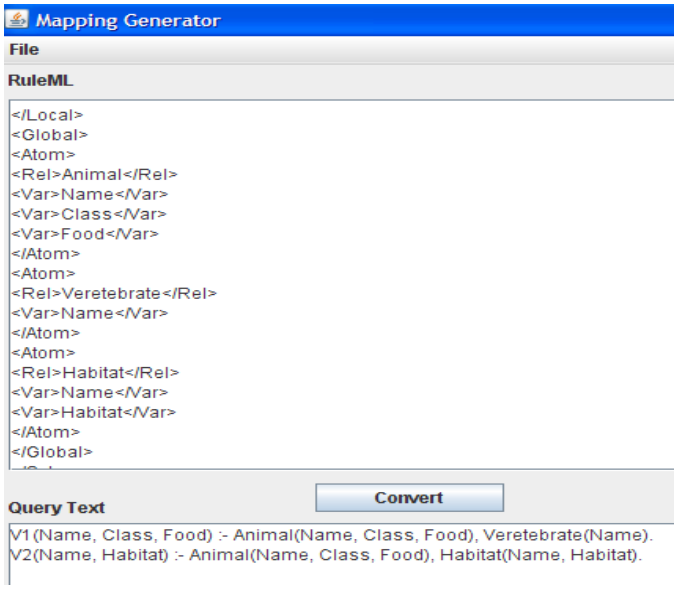


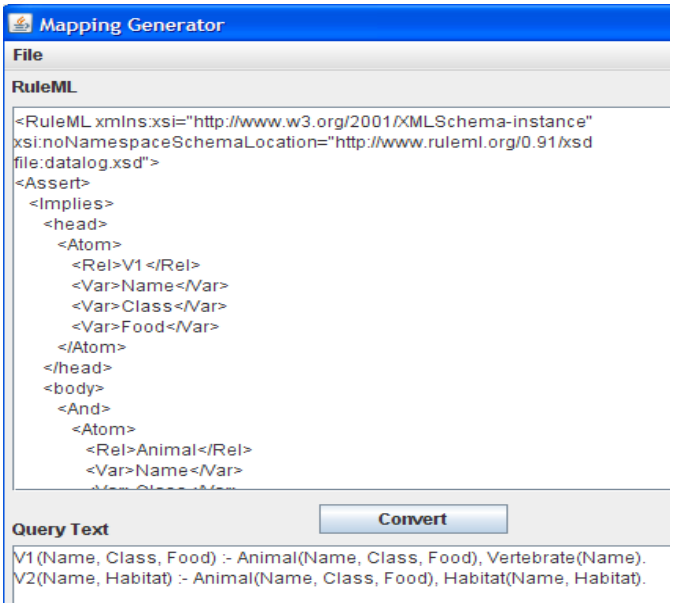
Figure 7.4: Description of Sources

The user can edit the values for **Databasename**, **Hostname**, **Userid** and **Password** elements for each source in the upper text window shown in Figure 7.4.

[1]This interface was implemented as part of an undergraduate honours project by A. Beauvais



(a) Description of Sources



(b) RuleML Elements for Mappings

Figure 7.5: Metadata Interface

The interface generates the XML representation of global relations based on the body of the view definitions in the user input text window shown in Figure 7.5 (a). The XML representation follows the specification described in Section 4.1 in Chapter

4. The view definitions given in *Datalog* format are translated to RuleML elements (cf. Section 4.3 in Chapter 4). The **head** and **body** elements are generated for the view definitions as shown in Figure 7.5 (b).

7.2.4 Metadata Parser

The *Metadata Parser* is implemented in C++ by leveraging the *Xerces C++ SAX Parser*. The parser builds the program for *Simple Specification* . The *Metadata Parser* also generates the *inverse rules* program [53] and the *refined specification* [19] program in *VISS* . The construction of the logic programs is described in detail in Chapter 8.

7.2.5 DLVDB

DLV allows for the evaluation of disjunctive $Datalog^{not}$ programs [32], and provides an easy interface to external databases using ODBC drivers. The logic program generated by the *Metadata Parser*, $\Pi(\mathcal{G})$ specifying the class of legal instances is combined with the query program in *Datalog*, $\Pi(Q)$ and with the *import* commands, which load data (the program facts) that are *relevant* (Cf . Chapter 6) to a user query. The data is stored in relational DBMSs, with which *DLV* is able to interact with through the wrappers. The combined program is run in *DLV* [51] invoked in *VISS*. The result of this program evaluation is the set of *certain* answers to the global query.

Chapter 8

Program Builder

This chapter describes the *Program Builder* component of the mediator *VISS* . The *Program Builder* constructs the logic program for *Simple Specification*, which is used to compute *certain* answers to a user query. We construct *Simple Specification* program using the modifications described in Section 5.5 in Chapter 5, but without the ground facts. We also describe how the *Inverse Rules* [53] and *Refined Specification* program [19] for specifying the class of minimal legal instances are generated in *VISS* [1].

8.1 Preliminaries

The input to the *Program Builder* is the set of pruned rules or view definitions obtained in XML format (cf. Figure 6.1 in Section 6.4 in Chapter 6). The XML result containing the pruned rules conforms to the schema specification listed in Listing 8.1. The XML Schema Definition shows (XSD) that elements **rules** and **rule** are complex types containing sequence of child elements (Listing 8.1 Lines 5-11). **rule** is a child element of **rules** and contains each of the pruned view definitions in the child elements **head** and **body** (Listing 8.1 Lines 12-23). Both the **head** and **body** elements contain an attribute *r1*. *r1* stores the name of the source predicate, when used in the **head** element (Listing 8.1 Lines 16-17) or the name of the global predicate, when used in the **body** element (Listing 8.1 Lines 27-28). Each **rule** element contains only

[1] Currently *VISS* computes only the certain answers even though we generate the refined version of the specification.

one **head** element and one or more **body** elements.

Listing 8.1: XMLSchema Definition (XSD) for the Pruned Rules XML Output

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="rules">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded"
        name="rule">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="0" name="head">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="r1" type="xs:string"
                      use="required" />
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element minOccurs="0" maxOccurs="unbounded"
              name="body">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="r1" type="xs:string"
                      use="optional" />
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Example 36 We use the XML output obtained in Section 6.4 in Chapter 6. Here, the **rule** element contains the head and body of the view definition for $V1$. The view definition is pruned to remove $Vertebrate$ from the body as it does not appear in the body of the query. The view definitions for $V2$, $V3$ and $V5$ is also listed as these are obtained as relevant sources for the query,

$$Ans(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat), Class = "mammal". \quad (8.1)$$

Listing 8.2: Pruned Rules XML Output

```
<rules xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:noNamespaceSchemaLocation='mappingrule.xsd'>
    <rule>
        <head r1="V1">V1(Name, Class ,Food)</head>
        <body r1="Animal">Animal(Name, Class ,Food)</body>
        <body r1=""/>
    </rule>
    <rule>
        <head r1="V3">V3(Name, Class ,Food)</head>
        <body r1="Animal">Animal(Name, Class ,Food)</body>
        <body r1="">(Class=mammal),</body>
    </rule>
    <rule>
        <head r1="V5">V5(Name, Food)</head>
        <body r1="Animal">Animal(Name, Class ,Food)</body>
        <body r1="">(Class=mammal),</body>
    </rule>
    <rule>
        <head r1="V2">V2(Name, Habitat)</head>
        <body r1="Animal">Animal(Name, Class ,Food)</body>
        <body r1="Habitat">Habitat(Name, Habitat)</body>
        <body r1=""/>
    </rule>
</rules>
```

The attribute $r1$ stores the name of the predicates or empty string ($r1 = ""$) in the case of built-ins. □

The schema diagram for the pruned rules XML output is shown in Figure 8.1.

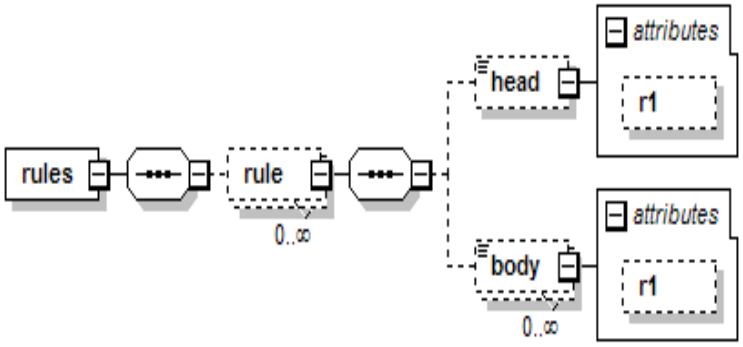


Figure 8.1: XML Output of Rules

The *Program Builder* first parses the XML file in Listing 8.2 using the *Xerces C++ SAX2 (Simple API for XML) API* - an open source library that provides the capability to parse, generate, manipulate and validate XML documents. The initial steps to setup the parser is shown in Listing 8.3.

Listing 8.3: Initial Parser Setup

```
1  SAXParser* parser = new SAXParser;
2  parser->setValidationScheme();
3  parser->setDoNamespaces();
4  parser->setDoSchema();
5  parser->setValidationSchemaFullChecking();
6  try
7  {
8  VISSHandler handler(encodingName, unRepFlags, parseOpt);
9        parser->setDocumentHandler(&handler);
10       parser->setErrorHandler(&handler);
11       parser->parse(xmlFile);
12 }
13 catch (const XMLException &toCatch)
14 {
15         // Display error message and stop execution
16 }
17 delete parser;
18 XMLPlatformUtils::Terminate();
```

The SAXParser class implements the XML parser (Listing 8.3 Line 1) and requires a handler class, which is publicly inherited from the HandlerBase class (Listing 8.3 Line 8), for processing the XML file. We implemented a VISSHandler class whose member variables and functions implement the guts of the Program Builder. If the XML file (cf. Listing 8.2) does not conform to the schema (cf. Listing 8.1), the *Handler* throws an error and the program fails with a message that the input is not valid (Listing 8.3 Lines 13-16). The data in a valid XML file, such as the one shown in Listing 8.2, is parsed and stored in an internal data structure (Listing 8.3 Line 9-11).

Listing 8.4: The *ruleAtom* Class Defintion

```
1  class ruleAtom
2  {
3  public:
4      string predicate;
5      vector<string> attributes;
6      // print the rule. For e.g. V1(a,b,c)
7      void print(string& outStr);
8      // print just the attributes. For e.g. a,b,c
9      void printAttributes(string& outStr);
10  };
```

The basic data structure used to process the **rule** element (cf. Listing 8.2) is a *ruleAtom* (Listing 8.4 Line 1), which has a predicate string and a list of attribute strings (Listing 8.4 Lines 4-5). For instance, $V1(Name, Class, Food)$ is a *ruleAtom* with a predicate $V1$ and a list of attributes - $Name$, $Class$ and $Food$. A *rule*, in

turn, has one *ruleAtom* representing the head and a list of *ruleAtom*s representing the body. The *string* and *vector* classes are defined in the C++ Standard Template Library [74].

When the *Handler* reads a **head** or **body** element within a **rule** element (cf. Listing 8.2), it stores the string in an internal list. A *tokenize()* function processes the string into a *ruleAtom*. For instance, the *tokenize()* function would process the string $V1(Name, Class, Food)$ into a *vector* with strings "$V1$", "$Name$", "$Class$" and "$Food$". After reading in the **head** and **body** elements of a **rule** element (and processing each into its *ruleAtom*), the *Program Builder* generates the logic program.

8.2 Building the Simple Specification Program

The text in the **head** and **body** elements of all the **rule** elements has already been read from the XML file (cf. Listing 8.2). The *legalInstance()* function (Listing 8.5) first prints the body atom, followed by ":-" and then the head atom (Listing 8.5 Lines 5-7). For instance, *legalInstance()* inverts the view definition for $V2$ and prints $Animal(Name, Class, Food)$:- $V2(Name, Habitat)$ first. The function then checks if each attribute in a body atom is also present in the head (Listing 8.5 Lines 8-10). If the attribute is not present, extra rules are printed (Listing 8.5 Lines 27-41). For example, attribute $Class$ and $Food$ are not present in $V2(Name, Habitat)$. So the *chosen* predicate is introduced, one for each existential attribute (i.e. $Class$ and $Food$) not present in the (inverted) body atom.

We directly use the *chosen* predicate, eliminating the rule using F_i predicates as shown in RP. Then, each of the *chosen* predicates is defined using additional rules as per *Simple Specification*. If an existential attribute occurs more than once in the same view definition, only one *chosen* predicate is used for that attribute. Hence, we check for multiple occurrence of the existential attribute in the same view definition (Listing 8.5 Lines 12-13). The rules for the *dom* predicates are generated for each existential variable in the mappings of the relevant source relations (Listing 8.5 Lines 42-44).

Listing 8.5: Rules Construction for Simple Specification

```
1
2  VISSHandler::legalInstance( ruleAtom* p_head, ruleAtom* p_body )
3  {
4          p_body->print()
5          print " :- "
6          p_head->print()
7  for each attribute in p_body->attributes
8  {
9          if p_body attribute is NOT found in p_head
10         {
11                 if attribute was processed earlier
12                         get the function name for the attribute
13                 else
14                 {
15                         get a new chosen name
16                         store the chosen name and attribute name
17                 }
18                 print the chosen name
19                 str1 = "(" + p_head->printAttributes() + ")"
20                 str2 = "(" + p_head->printAttributes()
21                  + p_body attribute + ")"
22                 print "("
23                 p_head->printAttributes()
24                 print p_body attribute
25                 print ")"
26         if p_body attribute NOT processed earlier
27                 {
28                         // print 2 extra rules
29                         // rule 1
30                         print "chosen" + chosen suffix + str2 +
31                         " :- " + p_head->predicate +
32             str1 + ", dom(" +
33             p_body attribute + "), not diffchoice" +
34                         chosen suffix + str2 + "."
35                         // rule 2
36                         print "diffchoice" + chosen suffix + str2 +
37                         " :- " + "chosen" + chosen suffix +
38                         "(" + p_head->printAttributes() +
39             ",U, dom(" + p_body attribute +
40             ", U !=" + p_body attribute + "."
41             print "dom(" + p_body attribute_Ex +
42                         " :- " + p_head_Ev->predicate_Rel +
43             str11 + "."
44                 }
45         }
46 }
47 print "."
48 }
```

The string *attribute_Ex* (Listing 8.5 Line 42) contains the existential variables in the mappings (e.g. $Class$ and $Food$ in $V2$). The string *predicate_Rel* contains the source relations (e.g. $V1$, $V3$ and $V5$) from which values are loaded into the *dom* predicate and *str11* is a string containing the existential variable with the remaining

variables (e.g. $Name$ in $V1$) masked using "_" (Listing 8.5 Lines 43-44).

Example 37 The *Simple Specification* program generated with the XML file in Listing 8.2 as input for the *legalInstance()* function is:

$$
\begin{aligned}
Animal(Name, Class, Food) &:- V1(Name, Class, Food).\\
Animal(Name, \textbf{"mammal"}, Food) &:- V3(Name, Class, Food).\\
Animal(Name, \textbf{"mammal"}, Food) &:- V5(Name, Food),\\
&\quad chosen1(Name, Food, Class).\\
chosen1(Name, Food, Class) &:- V5(Name, Food), dom(Class),\\
&\quad not\ diffchoice1(Name, Food, Class).\\
diffchoice1(Name, Food, Class) &:- chosen1(Name, Food, U),\\
&\quad dom(Class), U! = Class.\\
Animal(Name, Class, Food) &:- V2(Name, Habitat),\\
&\quad chosen2(Name, Habitat, Class),\\
&\quad chosen3(Name, Habitat, Food).\\
chosen2(Name, Habitat, Class) &:- V2(Name, Habitat), dom(Class),\\
&\quad not\ diffchoice2(Name, Habitat, Class).\\
diffchoice2(Name, Habitat, Class) &:- chosen2(Name, Habitat, U),\\
&\quad dom(Class), U! = Class.\\
chosen3(Name, Habitat, Food) &:- V2(Name, Habitat), dom(Food),\\
&\quad not\ diffchoice3(Name, Habitat, Food).\\
diffchoice3(Name, Habitat, Food) &:- chosen3(Name, Habitat, U),\\
&\quad dom(Food), U! = Food.\\
Habitat(Name, Habitat) &:- V2(Name, Habitat).\\
dom(Class) &:- V1(\_, Class, \_).\\
dom(Food) &:- V1(\_, \_, Food).\\
dom(Class) &:- V3(\_, Class, \_).\\
dom(Food) &:- V3(\_, \_, Food).\\
dom(Food) &:- V5(\_, Food).
\end{aligned}
$$

The value in the built-in $Class$ = "$mammal$", is directly substituted for the variable $Class$ in the rule for $V3$ and $V5$ above. The resulting program is combined with the query:

$$
\begin{aligned}
Ans(Name, Habitat) \leftarrow\ & Animal(Name, Class, Food), Habitat(Name, Habitat),\\
& Class = "mammal".
\end{aligned}
$$

The *import* commands for retrieving facts from the source relations (cf. Example 32 in Chapter 6) is given by:

$$\#import(animalkingdom,"test","test","Select\ Distinct\ *\ From\ V1 \\ where\ Class\ =\ 'mammal'",V1,type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","Select\ Distinct\ *\ From\ V3 \\ where\ Class\ =\ 'mammal'",V3,type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test1","test1","Select\ Distinct\ *\ From\ V2", \\ V2,type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test1","test1","Select\ Distinct\ *\ From\ V5", \\ V5,type:Q\_Const,Q\_Const).$$

The facts are retrieved during runtime and directly loaded into main memory. The combined program (*lglinst.dlv*) is run in *DLV* [52] under cautious reasoning to get *certain* answers as follows:

$$dl.exe\ -silent\ -cautious\ lglinst.dlv$$
$$"dolphin","ocean"$$
$$"camel","desert"$$
$$"elephant","savannah"$$
$$"giraffe","savannah"$$
$$"lion","savannah"$$
$$"deer","forest"$$

□

8.3 Building the Refined Specification Program

The *Program Builder* in *VISS* also generates the *refined specification* of the program for minimal instances (cf. Section 2.4 in Chapter 2). The specification program for minimal instances can be used to compute *consistent* answers, which are true in all repairs of the minimal legal instances[2].

The *Program Builder* first reads the head and body of all the rules from the XML file (cf. Listing 8.2). For each rule, a string is constructed out of the head and all the body *ruleAtoms* before the *refSpecs()* function (for generating the *refined specification*) is called. The function *lowerString()* (Listing 8.6 Line 2) transforms the upper case characters in a string into lower case. The function *refSpecs()* is called with *p_head*, *p_body* and *lastRule* as parameters (Listing 8.6 Lines 3-5). For each body *ruleAtom*, a string called *lastRule* is constructed as shown in Listing 8.6. First, the

[2]This is scope for future work to incorporate consistent query answering in *VISS* .

rule is inverted by printing the body atom, followed by ":-" and then the head atom (Listing 8.6 Lines 7-9).

The presence of existential variables is checked and if there are no existential variables (i.e. all variables in the body of the rule also appear in the head of the rule), then the inverted rule is printed *as is*. An extra argument $t0$ is added to the head of the inverted rule as per *refined specification* (Listing 8.6 Lines 30-33). The predicate, which contains the argument $t0$ is an obligatory atom in all the minimal instances [19]. If existential variables are present, the *Add* predicate, suffixed with the source relation name, is printed (Listing 8.6 Lines 11-15). The function *refSpecsHelper()* is called to generate the extra rules for the existential variables (Listing 8.6 Lines 18-24). Example 38 illustrates the *refSpecs()* function.

Listing 8.6: Rules Construction for Refined Specification

```
1  lastRule = p_body->predicate + "(" + p_body->printAttributes() + ",n" +
2  lowerString(p_head->predicate + ")"
3  VISSHandler::refSpecs( ruleAtom* p_head,
4                         ruleAtom* p_body,
5                         string lastRule )
6  {
7          string auxRule, refSpecsStr;
8          refSpecsStr = p_body->predicate + "("
9          refSpecsStr += p_body->printAttributes();
10
11 if (p_body has attributes NOT found in p_head)
12 {
13         refSpecsStr += "," + lowerString(p_head->predicate) + ")"
14         refSpecsStr += " :- Add"
15         p_head->print
16 for each attribute attr[i] in p_body->attributes
17 {
18         if (attr[i] NOT found in p_head)
19         {
20                 string ruleEnd;
21                 extraRules = refSpecsHelper( attr[i], p_head,
22 refSpecsStr, auxRule, ruleEnd );
23                 ruleEnd += lastRule;
24         }
25 }
26 refSpecsStr += ".\n"
27 auxRule += ".\n"
28 }
29 else
30 {
31         refSpecsStr += ",t0) :-"
32         refSpecsStr += p_head->print() + ".\n"
33 }}
```

Example 38 We use view definitions for $V1$ and $V2$ from Listing 8.2. The rule for $V1$ does not contain any existential variables and hence, following are the specifications generated,

$$\begin{aligned} &\%\textit{Refined specifications for } V1: \\ &Animal(Name, Class, Food, t0) \ :- \ \mathbf{V1}(Name, Class, Food) \\ &Vertebrate(Name, t0) \ :- \ \mathbf{V1}(Name, Class, Food) \end{aligned}$$

□

Listing 8.7: Rules Construction for Refined Spec. in the Presence of Existential Variable

```
1  string VISSHandler::refSpecsHelper( string& attr, ruleAtom* p_head,
2                                      string& outStr, string& auxRule,
3                                          string& ruleEnd )
4  {string retStr;
5  if attr was processed earlier
6                  get the chosen name for the attribute
7  else{
8                  get a new chosen name
9                  store the chosen name and attribute name
10         }
11         string str1 = "(" + p_head->printAttributes() + ")";
12         string str2 = "(" + p_head->printAttributes() + attr + ")";
13         outStr += "," + chosen name + str2;
14         if attr NOT processed earlier
15         {
16                 retStr += chosen name + str2 + " :- Add" +
17                 p_head->predicate + attr + str1 + ", dom(" +
18                 attr + "), chosen" + chosen suffix + str2 + ".\n"
19                 retStr += "chosen" + chosen suffix + str2 +
20                 " :- " + "Add" + p_head->predicate + attr +
21                 Str1 + ", dom(" + attr + "), not diffchoice" +
22                 chosen suffix + str2 + ".\n"
23                 retStr += "diffchoice" + chosen suffix + str2 +
24                 " :- " + "chosen" + chosen suffix + "(" +
25                 p_head->printAttributes() + ",U), dom(" +
26                 attr + "), U != " + attr + ".\n"
27         retStr += "Add" + p_head->predicate + attr + str1 +
28         " :- " + "Add" + p_head->predicate + str1 +
29         ", not Aux" + p_head->predicate + attr + str1 + ".\n"
30         retStr += "Aux" + p_head->predicate + attr + str1 +
31         " :- " + "Var" + p_head->predicate + attr + str2 + ".\n"
32                 if (auxRule is empty)
33                 {
34             auxRule = "Add" + p_head->predicate + str1 + "
35             :- " + p_head->predicate + str1 + ", not Aux" +
36             p_head->predicate + str1 + ".\n";
37             auxRule += "Aux" + p_head->predicate +
38             str1 + " :- " + "Var" + p_head->predicate + attr + str2;
39                 }
40                 else{auxRule += ", Var" + p_head->predicate + attr + str2;
41             }
42                 ruleEnd = "Var" + p_head->predicate + attr + str2 +
43         " :- ";}
44         return retStr;}
```

The *refSpecsHelper()* function is used to generate the extra rules when an attribute in *p_body* is not found in *p_head*. The *chosen* predicate corresponding to each existential variable and the two extra rules defining it are generated using the *refined specification* (cf. Section 2.4 in Chapter 2) as shown in Listing 8.7 Lines 23-34. The predicate *Add* suffixed with the source relation name (i.e. $AddV2$) is defined by extra rules as shown in Listing 8.7 Lines 35-58. The $AddVi(\bar{X})$ is true only when the openness of Vi is not satisfied through other views [19]. Example 39 illustrates the *refSpecsHelper()* function.

Example 39 (Example 38 continued) We use predicate $AddV2$ to specify the openness of $V2$ and the rule for $V2$ contains the existential variables $Class$ and $Food$, all of which are defined by extra rules as follows,

$$\%\textit{Refined specifications for V2}:$$

$$Animal(Name, Class, Food, v2) :- AddV2(Name, Habitat), chosen1(Name, Habitat, Class), chosen2(Name, Habitat, Food). \quad (1)$$

$$AddV2(Name, Habitat) :- V2(Name, Habitat), not\ AuxV2(Name, Habitat). \quad (2)$$

$$AuxV2(Name, Habitat) :- VarV2Class(Name, Habitat, Class), VarV2Food(Name, Habitat, Food). \quad (3)$$

$$chosen1(Name, Habitat, Class) :- AddV2Class(Name, Habitat), dom(Class), not\ diffchoice1(Name, Habitat, Class). \quad (4)$$

$$diffchoice1(Name, Habitat, Class) :- chosen1(Name, Habitat, U), dom(Class), U! = Class. \quad (5)$$

$$AddV2Class(Name, Habitat) :- AddV2(Name, Habitat), not\ AuxV2Class(Name, Habitat). \quad (6)$$

$$AuxV2Class(Name, Habitat) :- VarV2Class(Name, Habitat, Class). \quad (7)$$

$$VarV2Class(Name, Habitat, Class) :- Animal(Name, Class, Food, nv2), Habitat(Name, Habitat, nv2). \quad (8)$$

$$Habitat(Name, Habitat, t0) :- V2(Name, Habitat). \quad (9)$$

$$chosen2(Name, Habitat, Food) : -AddV2Food(Name, Habitat),$$
$$dom(Food),$$
$$not\ diffchoice2(Name, Habitat, Food). \quad (10)$$
$$diffchoice2(Name, Habitat, Food) : -chosen2(Name, Habitat, U),$$
$$dom(Food), U! = Food. \quad (11)$$
$$AddV2Food(Name, Habitat) : -AddV2(Name, Habitat),$$
$$not\ AuxV2Food(Name, Habitat). \quad (12)$$
$$AuxV2Food(Name, Habitat) : -VarV2Food(Name, Habitat, Food). \quad (13)$$
$$VarV2Food(Name, Habitat, Food) : -Animal(Name, Class, Food, nv2),$$
$$Habitat(Name, Habitat, nv2). \quad (14)$$

The predicate $AddV2$ is used in place of $V2$ (in the *Simple Specification*) and $AddV2$ is defined by extra rules (Lines 2-3). The $V2$ predicate used for defining the $chosen_i$ predicate in *Simple Specification* is replaced with the $AddV2Class$ and $AddV2Food$ in Lines 6-8 and Lines 11-13 respectively. Again, we replace the function predicate F_i used in the *refined specification* [19] with the $chosen_i$ predicate directly, thus eliminating redundant rules. □

8.4 Building the Inverse Rules Program

The *Inverse Rules* [31] are generated only when there are no built-ins in the view definitions. As before, the head and body of all the rules has been read from the XML file (cf. Listing 8.2). The *printInvert()* function simply checks if each attribute in each of the body atom of a rule is also present in the head (Listing 8.8 Lines 6-10). If it is present, the attribute is simply printed as it is (Listing 8.8 Lines 10-11). If the attribute is not present (i.e. it is an existential attribute), it is replaced by a string f_i where $i = 1..n$, n is the number of existential variables and the attributes of the head (Listing 8.8 Lines 13-22). Example 40 illustrates the *printInvert()* function.

Listing 8.8: Function *printInvert()*

```
1  VISSHandler::printInvert( ruleAtom* p_head, ruleAtom* p_body )
2  {
3  print the body predicate
4  print "("
5  for each attribute in p_body->attributes
6  {
7          if( !first attribute )
8          print ","
9                  if p_body attribute is found in p_head
10                         print the attribute
11                 else
12     {
13                 if attribute was processed earlier
14                         print the function name and head attributes
15                 else
16                 {
17                         get a new function name
18                         store the function name and attribute name
19                         print the new function name and head attributes
20                 }
21     }
22         print ") <-" p_head->print()
23 }
24 }
```

Example 40 We use view definitions for $V1$ and $V2$ from Listing 8.2. The rule for $V1$ does not contain any existential variables and rule for $V2$ contains $Class$ and $Food$ as existential variables. There are no built-ins in the view definitions. The *Program Builder* generates the *Inverse Rules* as follows,

$Animal(Name, Class, Food) :- V1(Name, Class, Food).$
$Vertebrate(Name) :- V1(Name, Class, Food).$
$Animal(Name, f1(Name, Habitat), f2(Name, Habitat)) :- V2(Name, Habitat).$
$Habitat(Name, Habitat) :- V2(Name, Habitat).$

□

Chapter 9

Design Rationale and Experiments

In this chapter, we present the rationale behind the design of *VISS* and also experimental results obtained by computing *certain* answers, (a) without optimization, (b) using only the global predicates in the body of the query and (c) using the optimization steps presented in this research (QP, SP, SQC and RP). We look at the gain observed in terms of number of relevant sources considered, number of tuples imported into the logic program, execution time and the size of the logic program.

9.1 Design Rationale of VISS

The choice of specification languages, viz. XML and RuleML in VISS, was based on the versatility offered by them to represent the mappings, integrity constraints as well as the access information of the data sources. We could create custom XML elements for the mappings but RuleML already provides the required specifications. Since RuleML is based on XML, we could represent the entire metadata in a single XML file. More notably, the use of XML and RuleML allows us to use XQuery for querying the metadata. The metadata representations in the mediators discussed earlier (cf. Chapter 3) lack a standard query language. As a result, a custom *Query Execution Engine* was built for these applications. The use of a logic program approach in VISS helps provide further extensions through the use of stable models.

In VISS, the XML and RuleML metadata representation is parsed to build the logic program using open-source parsers. The parser automatically reads the XML

document and performs actions based on the tags encountered in the document. We only need to specify what those actions should be (cf. Chapter 8) in the handler class of the parser. *VISS* uses the Xerces Simple API for XML (SAX) parser, which provides stream-oriented APIs for parsing XML. SAX is more efficient compared to the Xerces DOM (Document Object Model) parser because it bypasses the creation of a tree-based object model and loading it in memory [76].

A metadata representation using *Datalog* might seem a natural choice as we use a logic program to compute *certain* answers and *Datalog* is a language for representing rules. However, the *Datalog* representation would still require custom code to read and process the mappings from scratch and build the logic program specification. Also, using *Datalog* to represent the access information for the sources will result in second-order logic formulas. We illustrate this in Example 41.

Example 41 The representation of the source **animalkingdom** using our XML representation is as follows:

```
<Source name="animalkingdom">
    <Type>sqlexpress</Type>
    <Hostname>animalkingdom</Hostname>
    <Databasename>animalkingdom</Databasename>
    <Userid>test</Userid>
    <Password>test</Password>
    <Atom>
        <Rel>V1</Rel>
        <Var>Name</Var>
        <Var>Class</Var>
        <Var>Food</Var>
    </Atom>
</Source>
```

Representing the structure of the local source in *Datalog* would result in a second-order formula as follows:

$$animalkingdom \quad (sqlexpress, animalkingdom, animalkingdom, test, test, \\ Atom(V1, Name, Class, Food)).$$

□

The representation of access information in second-order logic, even if not used for

any computation in this form, still presents a question of how it can be translated into *Import* commands in the logic program specification.

The Berkeley DB XML (BDBXML) in *VISS* serves the purpose of a *Metadata Store* as well as a *Query Execution Engine* because XQilla, the XQuery execution engine, is already built inside BDBXML.

9.2 Experimental Setup

We describe the experiments performed to analyze the gain obtained using the optimization steps in our design and implementation of VISS. We compare our approach with using the original EIRA and not with the other pruning approaches. The experiments were performed on a PC with Intel Pentium M processor of 1.6 GHz, 1 GB RAM using a Windows XP 2002 SP3 operating system. We use four data sources and their relations as shown in Table 9.1. The global relations defined in the mediator are $Animal$, $Vertebrate$, $Habitat$, $AnimalList$, $FoodList$ and $ClassList$. The view definitions describing the source relations are given by:

$$V1(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (9.1)$$

$$LV1(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (9.2)$$

$$V2(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \quad (9.3)$$

$$LV2(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \quad (9.4)$$

$$V3(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (9.5)$$

$$LV3(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (9.6)$$

$$V4(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "bird". \quad (9.7)$$

$$LV4(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "bird". \quad (9.8)$$

$$V5(Name, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (9.9)$$

$$LV5(Name, Food) \leftarrow Animal(Name, Class, Food), Class = "mammal". \quad (9.10)$$

$$V6(Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \quad (9.11)$$

$$LV6(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "amphibian". \quad (9.12)$$

$$V7(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Class = "reptile". \quad (9.13)$$

$$AnimalV1(Name, Class) \leftarrow AnimalList(Name, Class). \quad (9.14)$$

$$FoodV1(Food) \leftarrow FoodList(Food). \quad (9.15)$$

$$ClassV1(Class) \leftarrow ClassList(Class). \quad (9.16)$$

$$VTest1(Name) \leftarrow Vertebrate(Name). \quad (9.17)$$

$$VTest2(Name) \leftarrow Vertebrate(Name). \quad (9.18)$$

Table 9.1: Data Sources.

Database	Source Relation	Number of Tuples	Description
animalkingdom	V1	500	List of vertebrates
	LV1	5188	List of vertebrates ⊇ V1
	FoodV1	300	List of common animal food
	AnimalV1	300	List of animals and their classes
	ClassV1	300	List of animal classes
	V3	500	List of mammals
	LV3	5138	List of mammals ⊇ V3
	V4	500	List of birds
	LV5	5123	List of mammals ⊇ V5
animalhabitat	V2	1100	List of animals and their habitat
	LV2	5376	List of animals and their habitat ⊇ V2
	V5	100	List of mammals
	V6	1300	List of common animal habitats
mysqltest	LV6	5220	List of amphibians
	V7	5240	List of reptiles
accesstest	LV4	5583	List of birds ⊇ V4
	VTest1	100	List of vertebrates
	VTest2	100	List of vertebrates

We compute *certain* answers for four query test cases that are representative of the type of queries we consider in this thesis. We do this in three scenarios. First scenario is generating the logic program without involving any optimization steps. The second scenario uses the global relations in the body of the query as the only criteria for identifying the sources. The third scenario uses the optimization steps - QP, SP, SQC and RP (cf. Chapter 5). We list the logic program in each test scenario containing the *import* commands, the *Simple Specification* program without the *dom* rules and the query program. The *dom* atoms in the first scenario will include all values from the active domain and for the sake of brevity, we do not show this in the logic program listed in all the scenarios. However, when running the logic program using DLV, the rules for the *dom* predicates will have to be added to all the programs listed in the test cases.

9.2.1 Test Case 1

A conjunctive query that asks for animal names, class, food and habitat.

$$Ans(Name, Class, Food, Habitat) \leftarrow Animal(Name, Class, Food), \\ Habitat(Name, Habitat). \qquad (9.19)$$

Program 9.1a lists the logic program without optimization that imports all the data from all the sources available and using the rules for all view definitions. This, of course, is a naive method for computing answers. The second scenario uses the global relations in the body of the query namely $Animal$ and $Habitat$ and identifies the sources from the view definitions. Only those view definitions that contain the global relations that are in the body of the query are considered. Program 9.1b uses a reduced number of sources but the logic program contains some redundant rules. Program 9.1c uses the optimization step RP and prunes the rules for the function predicate fi and replaces them with *choseni* directly (RP step Chapter 5 Section 5.5). Also, the equality built-ins in the view definition are directly substituted for the variable. For example, the attribute $Class$ is substituted with the value "*mammal*" in the rule for $V3$, $LV3$, $V5$ and $LV5$. Since it is a conjunctive query without any equality built-ins, the sources identified in Program 9.1c are the same as for Program 9.1b.

Program 9.1a for scenario 1:

$$\#import(animalkingdom,"test","test","select\ *\ from\ V1",V1,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ LV1",LV1,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ FoodV1",FoodV1,$$
$$type:Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ AnimalV1",AnimalV1,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ ClassV1",ClassV1,$$
$$type:Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ V3",V3,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ LV3",LV3,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ V4",V4,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ LV5",LV5,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V2",V2,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ LV2",LV2,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V5",V5,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V6",V6,$$
$$type:Q\_Const).$$
$$\#import(mysqltest,"test","test","select\ *\ from\ LV6",LV6,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(mysqltest,"test","test","select\ *\ from\ V7",V7,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(accesstest,"","","select\ *\ from\ VTest1",VTest1,$$
$$type:Q\_Const).$$
$$\#import(accesstest,"","","select\ *\ from\ LV4",LV4,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(accesstest,"","","select\ *\ from\ VTest2",VTest2,$$
$$type:Q\_Const,Q\_Const).$$
$$Vertebrate(Name):-VTest1(Name).$$
$$Vertebrate(Name):-VTest2(Name).$$

$$
\begin{aligned}
Animal(Name, Class, Food) &: -V1(Name, Class, Food).\\
Vertebrate(Name) &: -V1(Name, Class, Food).\\
Animal(Name, Class, Food) &: -LV1(Name, Class, Food).\\
Vertebrate(Name) &: -LV1(Name, Class, Food).\\
Animal(Name, Class, Food) &: -V3(Name, Class, Food).\\
Animal(Name, Class, Food) &: -LV3(Name, Class, Food).\\
Animal(Name, Class, Food) &: -V4(Name, Class, Food).\\
Animal(Name, Class, Food) &: -LV4(Name, Class, Food).\\
Animal(Name, Class, Food) &: -V5(Name, Food), f6(Name, Food, Class).\\
f6(Name, Food, Class) &: -V5(Name, Food), dom(Class),\\
&\quad chosen6(Name, Food, Class).\\
chosen6(Name, Food, Class) &: -V5(Name, Food), dom(Class),\\
&\quad not\ diffchoice6(Name, Food, Class).\\
diffchoice6(Name, Food, Class) &: -chosen6(Name, Food, U),\\
&\quad dom(Class), U! = Class.\\
Animal(Name, Class, Food) &: -LV5(Name, Food), f7(Name, Food, Class).\\
f7(Name, Food, Class) &: -LV5(Name, Food), dom(Class),\\
&\quad chosen7(Name, Food, Class).\\
chosen7(Name, Food, Class) &: -LV5(Name, Food), dom(Class),\\
&\quad not\ diffchoice7(Name, Food, Class).\\
diffchoice7(Name, Food, Class) &: -chosen7(Name, Food, U), dom(Class),\\
&\quad U! = Class.\\
Animal(Name, Class, Food) &: -V2(Name, Habitat),\\
&\quad f8(Name, Habitat, Class),\\
&\quad f9(Name, Habitat, Food).\\
f8(Name, Habitat, Class) &: -V2(Name, Habitat), dom(Class),\\
&\quad chosen8(Name, Habitat, Class).\\
chosen8(Name, Habitat, Class) &: -V2(Name, Habitat), dom(Class),\\
&\quad not\ diffchoice8(Name, Habitat, Class).\\
diffchoice8(Name, Habitat, Class) &: -chosen8(Name, Habitat, U),\\
&\quad dom(Class), U! = Class.\\
f9(Name, Habitat, Food) &: -V2(Name, Habitat), dom(Food),\\
&\quad chosen9(Name, Habitat, Food).\\
chosen9(Name, Habitat, Food) &: -V2(Name, Habitat), dom(Food),\\
&\quad not\ diffchoice9(Name, Habitat, Food).\\
diffchoice9(Name, Habitat, Food) &: -chosen9(Name, Habitat, U),\\
&\quad dom(Food), U! = Food.\\
Habitat(Name, Habitat) &: -V2(Name, Habitat).\\
Animal(Name, Class, Food) &: -LV2(Name, Habitat),\\
&\quad f10(Name, Habitat, Class),\\
&\quad f11(Name, Habitat, Food).
\end{aligned}
$$

$$f10(Name, Habitat, Class) :- LV2(Name, Habitat), dom(Class), chosen10(Name, Habitat, Class).$$
$$chosen10(Name, Habitat, Class) :- LV2(Name, Habitat), dom(Class), not\ diffchoice10(Name, Habitat, Class).$$
$$diffchoice10(Name, Habitat, Class) :- chosen10(Name, Habitat, U), dom(Class), U! = Class.$$
$$f11(Name, Habitat, Food) :- LV2(Name, Habitat), dom(Food), chosen11(Name, Habitat, Food).$$
$$chosen11(Name, Habitat, Food) :- LV2(Name, Habitat), dom(Food), not\ diffchoice11(Name, Habitat, Food).$$
$$diffchoice11(Name, Habitat, Food) :- chosen11(Name, Habitat, U), dom(Food), U! = Food.$$
$$Habitat(Name, Habitat) :- LV2(Name, Habitat).$$
$$Animal(Name, Class, Food) :- V6(Habitat), f12(Habitat, Name), f13(Habitat, Class), f14(Habitat, Food).$$
$$f12(Habitat, Name) :- V6(Habitat), dom(Name), chosen12(Habitat, Name).$$
$$chosen12(Habitat, Name) :- V6(Habitat), dom(Name), not\ diffchoice12(Habitat, Name).$$
$$diffchoice12(Habitat, Name) :- chosen12(Habitat, U), dom(Name), U! = Name.$$
$$f13(Habitat, Class) :- V6(Habitat), dom(Class), chosen13(Habitat, Class).$$
$$chosen13(Habitat, Class) :- V6(Habitat), dom(Class), not\ diffchoice13(Habitat, Class).$$
$$diffchoice13(Habitat, Class) :- chosen13(Habitat, U), dom(Class), U! = Class.$$
$$f14(Habitat, Food) :- V6(Habitat), dom(Food), chosen14(Habitat, Food).$$
$$chosen14(Habitat, Food) :- V6(Habitat), dom(Food), not\ diffchoice14(Habitat, Food).$$
$$diffchoice14(Habitat, Food) :- chosen14(Habitat, U), dom(Food), U! = Food.$$
$$Habitat(Name, Habitat) :- V6(Habitat), f12(Habitat, Name).$$
$$Animal(Name, Class, Food) :- LV6(Name, Class, Food).$$
$$Animal(Name, Class, Food) :- V7(Name, Class, Food).$$
$$AnimalList(Name, Class) :- AnimalV1(Name, Class).$$
$$FoodList(Food) :- FoodV1(Food).$$
$$ClassList(Class) :- ClassV1(Class).$$
$$Ans(Name, Class, Food, Habitat) :- Habitat(Name, Habitat), Animal(Name, Class, Food).$$
$$Ans(Name, Class, Food, Habitat)?$$

Program 9.1b for scenario 2:

$$\#import(animalkingdom,"test","test","select\ *\ from\ V1",V1,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ LV1",LV1,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ V3",V3,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ LV3",LV3,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ V4",V4,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ LV5",LV5,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V2",V2,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ LV2",LV2,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V5",V5,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V6",V6,$$
$$type:Q\_Const).$$
$$\#import(mysqltest,"test","test","select\ *\ from\ LV6",LV6,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(mysqltest,"test","test","select\ *\ from\ V7",V7,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(accesstest,"","","select\ *\ from\ LV4",LV4,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$Animal(Name,Class,Food):-V1(Name,Class,Food).$$
$$Animal(Name,Class,Food):-LV1(Name,Class,Food).$$
$$Animal(Name,Class,Food):-V3(Name,Class,Food).$$
$$Animal(Name,Class,Food):-LV3(Name,Class,Food).$$
$$Animal(Name,Class,Food):-V4(Name,Class,Food).$$
$$Animal(Name,Class,Food):-LV4(Name,Class,Food).$$
$$Animal(Name,Class,Food):-V5(Name,Food),f6(Name,Food,Class).$$
$$f6(Name,Food,Class):-V5(Name,Food),dom(Class),$$
$$chosen6(Name,Food,Class).$$
$$chosen6(Name,Food,Class):-V5(Name,Food),dom(Class),$$
$$not\ diffchoice6(Name,Food,Class).$$
$$diffchoice6(Name,Food,Class):-chosen6(Name,Food,U),$$
$$dom(Class),U!=Class.$$

$$Animal(Name, Class, Food) : -LV5(Name, Food), f7(Name, Food, Class).$$
$$f7(Name, Food, Class) : -LV5(Name, Food), dom(Class), chosen7(Name, Food, Class).$$
$$chosen7(Name, Food, Class) : -LV5(Name, Food), dom(Class), not\ diffchoice7(Name, Food, Class).$$
$$diffchoice7(Name, Food, Class) : -chosen7(Name, Food, U), dom(Class), U! = Class.$$
$$Animal(Name, Class, Food) : -V2(Name, Habitat), f8(Name, Habitat, Class), f9(Name, Habitat, Food).$$
$$f8(Name, Habitat, Class) : -V2(Name, Habitat), dom(Class), chosen8(Name, Habitat, Class).$$
$$chosen8(Name, Habitat, Class) : -V2(Name, Habitat), dom(Class), not\ diffchoice8(Name, Habitat, Class).$$
$$diffchoice8(Name, Habitat, Class) : -chosen8(Name, Habitat, U), dom(Class), U! = Class.$$
$$f9(Name, Habitat, Food) : -V2(Name, Habitat), dom(Food), chosen9(Name, Habitat, Food).$$
$$chosen9(Name, Habitat, Food) : -V2(Name, Habitat), dom(Food), not\ diffchoice9(Name, Habitat, Food).$$
$$diffchoice9(Name, Habitat, Food) : -chosen9(Name, Habitat, U), dom(Food), U! = Food.$$
$$Habitat(Name, Habitat) : -V2(Name, Habitat).$$
$$Animal(Name, Class, Food) : -LV2(Name, Habitat), f10(Name, Habitat, Class), f11(Name, Habitat, Food).$$
$$f10(Name, Habitat, Class) : -LV2(Name, Habitat), dom(Class), chosen10(Name, Habitat, Class).$$
$$chosen10(Name, Habitat, Class) : -LV2(Name, Habitat), dom(Class), not\ diffchoice10(Name, Habitat, Class).$$
$$diffchoice10(Name, Habitat, Class) : -chosen10(Name, Habitat, U), dom(Class), U! = Class.$$
$$f11(Name, Habitat, Food) : -LV2(Name, Habitat), dom(Food), chosen11(Name, Habitat, Food).$$
$$chosen11(Name, Habitat, Food) : -LV2(Name, Habitat), dom(Food), not\ diffchoice11(Name, Habitat, Food).$$
$$diffchoice11(Name, Habitat, Food) : -chosen11(Name, Habitat, U), dom(Food), U! = Food.$$
$$Habitat(Name, Habitat) : -LV2(Name, Habitat).$$
$$Animal(Name, Class, Food) : -V6(Habitat), f12(Habitat, Name), f13(Habitat, Class), f14(Habitat, Food).$$

$$f12(Habitat, Name) : -V6(Habitat), dom(Name), chosen12(Habitat, Name).$$
$$chosen12(Habitat, Name) : -V6(Habitat), dom(Name), not\ diffchoice12(Habitat, Name).$$
$$diffchoice12(Habitat, Name) : -chosen12(Habitat, U), dom(Name), U! = Name.$$
$$f13(Habitat, Class) : -V6(Habitat), dom(Class), chosen13(Habitat, Class).$$
$$chosen13(Habitat, Class) : -V6(Habitat), dom(Class), not\ diffchoice13(Habitat, Class).$$
$$diffchoice13(Habitat, Class) : -chosen13(Habitat, U), dom(Class), U! = Class.$$
$$f14(Habitat, Food) : -V6(Habitat), dom(Food), chosen14(Habitat, Food).$$
$$chosen14(Habitat, Food) : -V6(Habitat), dom(Food), not\ diffchoice14(Habitat, Food).$$
$$diffchoice14(Habitat, Food) : -chosen14(Habitat, U), dom(Food), U! = Food.$$
$$Habitat(Name, Habitat) : -V6(Habitat), f12(Habitat, Name).$$
$$Animal(Name, Class, Food) : -LV6(Name, Class, Food).$$
$$Animal(Name, Class, Food) : -V7(Name, Class, Food).$$
$$Ans(Name, Class, Food, Habitat) : -Habitat(Name, Habitat), Animal(Name, Class, Food).$$
$$Ans(Name, Class, Food, Habitat)?$$

Program 9.1c for scenario 3:

$$\#import(animalkingdom, "test", "test", "select\ *\ from\ V1", V1, type : Q\_Const, Q\_Const, Q\_Const).$$
$$\#import(animalkingdom, "test", "test", "select\ *\ from\ LV1", LV1, type : Q\_Const, Q\_Const, Q\_Const).$$
$$\#import(animalkingdom, "test", "test", "select\ *\ from\ V3", V3, type : Q\_Const, Q\_Const, Q\_Const).$$
$$\#import(animalkingdom, "test", "test", "select\ *\ from\ LV3", LV3, type : Q\_Const, Q\_Const, Q\_Const).$$
$$\#import(animalkingdom, "test", "test", "select\ *\ from\ V4", V4, type : Q\_Const, Q\_Const, Q\_Const).$$
$$\#import(animalkingdom, "test", "test", "select\ *\ from\ LV5", LV5, type : Q\_Const, Q\_Const).$$

$$\#import(animalhabitat,"test","test","select\ *\ from\ V2",V2,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ LV2",LV2,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V5",V5,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V6",V6,$$
$$type:Q\_Const).$$
$$\#import(mysqltest,"test","test","select\ *\ from\ LV6",LV6,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(mysqltest,"test","test","select\ *\ from\ V7",V7,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(accesstest,"","","select\ *\ from\ LV4",LV4,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$Animal(Name,Class,Food):-V1(Name,Class,Food).$$
$$Animal(Name,Class,Food):-LV1(Name,Class,Food).$$
$$Animal(Name,"mammal",Food):-V3(Name,Class,Food).$$
$$Animal(Name,"mammal",Food):-LV3(Name,Class,Food).$$
$$Animal(Name,"bird",Food):-V4(Name,Class,Food).$$
$$Animal(Name,"bird",Food):-LV4(Name,Class,Food).$$
$$Animal(Name,"mammal",Food):-V5(Name,Food),$$
$$chosen6(Name,Food,Class).$$
$$chosen6(Name,Food,Class):-V5(Name,Food),dom(Class),$$
$$not\ diffchoice6(Name,Food,Class).$$
$$diffchoice6(Name,Food,Class):-chosen6(Name,Food,U),$$
$$dom(Class),U!=Class.$$
$$Animal(Name,"mammal",Food):-LV5(Name,Food),$$
$$chosen7(Name,Food,Class).$$
$$chosen7(Name,Food,Class):-LV5(Name,Food),dom(Class),$$
$$not\ diffchoice7(Name,Food,Class).$$
$$diffchoice7(Name,Food,Class):-chosen7(Name,Food,U),dom(Class),$$
$$U!=Class.$$
$$Animal(Name,Class,Food):-V2(Name,Habitat),$$
$$chosen8(Name,Habitat,Class),$$
$$chosen9(Name,Habitat,Food).$$
$$chosen8(Name,Habitat,Class):-V2(Name,Habitat),dom(Class),$$
$$not\ diffchoice8(Name,Habitat,Class).$$
$$diffchoice8(Name,Habitat,Class):-chosen8(Name,Habitat,U),$$
$$dom(Class),U!=Class.$$
$$chosen9(Name,Habitat,Food):-V2(Name,Habitat),dom(Food),$$
$$not\ diffchoice9(Name,Habitat,Food).$$

$$\begin{aligned}
\mathit{diffchoice}9(Name, Habitat, Food) :- & \mathit{chosen}9(Name, Habitat, U),\\
& dom(Food), U! = Food.\\
Habitat(Name, Habitat) :- & V2(Name, Habitat).\\
Animal(Name, Class, Food) :- & LV2(Name, Habitat),\\
& \mathit{chosen}10(Name, Habitat, Class),\\
& \mathit{chosen}11(Name, Habitat, Food).\\
\mathit{chosen}10(Name, Habitat, Class) :- & LV2(Name, Habitat), dom(Class),\\
& not\ \mathit{diffchoice}10(Name, Habitat, Class).\\
\mathit{diffchoice}10(Name, Habitat, Class) :- & \mathit{chosen}10(Name, Habitat, U),\\
& dom(Class), U! = Class.\\
\mathit{chosen}11(Name, Habitat, Food) :- & LV2(Name, Habitat), dom(Food),\\
& not\ \mathit{diffchoice}11(Name, Habitat, Food).\\
\mathit{diffchoice}11(Name, Habitat, Food) :- & \mathit{chosen}11(Name, Habitat, U),\\
& dom(Food), U! = Food.\\
Habitat(Name, Habitat) :- & LV2(Name, Habitat).\\
Animal(Name, Class, Food) :- & V6(Habitat), \mathit{chosen}12(Habitat, Name),\\
& \mathit{chosen}13(Habitat, Class),\\
& \mathit{chosen}14(Habitat, Food).\\
\mathit{chosen}12(Habitat, Name) :- & V6(Habitat), dom(Name),\\
& not\ \mathit{diffchoice}12(Habitat, Name).\\
\mathit{diffchoice}12(Habitat, Name) :- & \mathit{chosen}12(Habitat, U),\\
& dom(Name), U! = Name.
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}13(Habitat, Class) :- & V6(Habitat), dom(Class),\\
& not\ \mathit{diffchoice}13(Habitat, Class).\\
\mathit{diffchoice}13(Habitat, Class) :- & \mathit{chosen}13(Habitat, U),\\
& dom(Class), U! = Class.\\
\mathit{chosen}14(Habitat, Food) :- & V6(Habitat), dom(Food),\\
& not\ \mathit{diffchoice}14(Habitat, Food).\\
\mathit{diffchoice}14(Habitat, Food) :- & \mathit{chosen}14(Habitat, U),\\
& dom(Food), U! = Food.\\
Habitat(Name, Habitat) :- & V6(Habitat), \mathit{chosen}12(Habitat, Name).\\
Animal(Name, \text{"}amphibian\text{"}, Food) :- & LV6(Name, Class, Food).\\
Animal(Name, \text{"}reptile\text{"}, Food) :- & V7(Name, Class, Food).\\
Ans(Name, Class, Food, Habitat) :- & Habitat(Name, Habitat),\\
& Animal(Name, Class, Food).\\
Ans(Name, Class, Food, Habitat)? &
\end{aligned}$$

9.2.2 Test Case 2

A conjunctive query with built-ins that asks for name, class, food and habitat of animals that belong to class amphibians and those that eat insects.

$$\begin{aligned} Ans(Name, Class, Food, Habitat) \leftarrow\ & Animal(Name, Class, Food), \\ & Habitat(Name, Habitat), Food = "insects" \\ & Class = "amphibian". \end{aligned} \tag{9.20}$$

The logic programs for scenario 1 and 2 remains the same as test case 1. This is because scenario 1 considers everything irrespective of the query. Scenario 2 again considers the global relations *Animal* and *Habitat* but not the built-ins in the query.

Program 9.2c for scenario 3:

$$\begin{aligned} & \#import(animalkingdom, "sa", "gj", "Select\ Distinct\ *\ From\ V1\ where \\ & \qquad Class =' amphibian'\ and\ Food =' insects'", \\ & \qquad V1, type : Q\_Const, Q\_Const, Q\_Const). \\ & \#import(animalkingdom, "sa", "gj", "Select\ Distinct\ *\ From\ LV1\ where \\ & \qquad Class =' amphibian'\ and\ Food =' insects'", \\ & \qquad LV1, type : Q\_Const, Q\_Const, Q\_Const). \\ & \#import(mysqltest, "root", "gj", "Select\ Distinct\ *\ From\ LV6\ where \\ & \qquad Class =' amphibian'\ and\ Food =' insects'", \\ & \qquad LV6, type : Q\_Const, Q\_Const, Q\_Const). \\ & \#import(animalhabitat, "root", "gj", "Select\ Distinct\ *\ From\ V2", \\ & \qquad V2, type : Q\_Const, Q\_Const). \\ & \#import(animalhabitat, "root", "gj", "Select\ Distinct\ *\ From\ LV2", \\ & \qquad LV2, type : Q\_Const, Q\_Const). \\ & \#import(animalhabitat, "root", "gj", "Select\ Distinct\ *\ From\ V6", \\ & \qquad V6, type : Q\_Const). \\ & Animal(Name, Class, Food) : -V1(Name, Class, Food). \\ & Animal(Name, Class, Food) : -LV1(Name, Class, Food). \\ & Animal(Name, Class, Food) : -V2(Name, Habitat), \\ & \qquad chosen8(Name, Habitat, Class), \\ & \qquad chosen9(Name, Habitat, Food). \\ & chosen8(Name, Habitat, Class) : -V2(Name, Habitat), dom(Class), \\ & \qquad not\ diffchoice8(Name, Habitat, Class). \\ & diffchoice8(Name, Habitat, Class) : -chosen8(Name, Habitat, U), \\ & \qquad dom(Class), U! = Class. \end{aligned}$$

$$chosen9(Name, Habitat, Food) :- V2(Name, Habitat), dom(Food), \\ not\ diffchoice9(Name, Habitat, Food).$$

$$diffchoice9(Name, Habitat, Food) :- chosen9(Name, Habitat, U), \\ dom(Food), U! = Food.$$

$$Habitat(Name, Habitat) :- V2(Name, Habitat).$$

$$Animal(Name, Class, Food) :- LV2(Name, Habitat), \\ chosen10(Name, Habitat, Class), \\ chosen11(Name, Habitat, Food).$$

$$chosen10(Name, Habitat, Class) :- LV2(Name, Habitat), dom(Class), \\ not\ diffchoice10(Name, Habitat, Class).$$

$$diffchoice10(Name, Habitat, Class) :- chosen10(Name, Habitat, U), \\ dom(Class), U! = Class.$$

$$chosen11(Name, Habitat, Food) :- LV2(Name, Habitat), dom(Food), \\ not\ diffchoice11(Name, Habitat, Food).$$

$$diffchoice11(Name, Habitat, Food) :- chosen11(Name, Habitat, U), \\ dom(Food), U! = Food.$$

$$Habitat(Name, Habitat) :- LV2(Name, Habitat).$$

$$Animal(Name, Class, Food) :- V6(Habitat), chosen12(Habitat, Name), \\ chosen13(Habitat, Class), \\ chosen14(Habitat, Food).$$

$$chosen12(Habitat, Name) :- V6(Habitat), dom(Name), \\ not\ diffchoice12(Habitat, Name).$$

$$diffchoice12(Habitat, Name) :- chosen12(Habitat, U), \\ dom(Name), U! = Name.$$

$$chosen13(Habitat, Class) :- V6(Habitat), dom(Class), \\ not\ diffchoice13(Habitat, Class).$$

$$diffchoice13(Habitat, Class) :- chosen13(Habitat, U), \\ dom(Class), U! = Class.$$

$$chosen14(Habitat, Food) :- V6(Habitat), dom(Food), \\ not\ diffchoice14(Habitat, Food).$$

$$diffchoice14(Habitat, Food) :- chosen14(Habitat, U), \\ dom(Food), U! = Food.$$

$$Habitat(Name, Habitat) :- V6(Habitat), chosen12(Habitat, Name).$$

$$Animal(Name, \text{"}amphibian\text{"}, Food) :- LV6(Name, Class, Food).$$

$$Ans(Name, Class, Food, Habitat) :- Habitat(Name, Habitat), \\ Animal(Name, Class, Food), \\ Class = \text{"}amphibian\text{"}, Food = \text{"}insects\text{"}.$$

$$Ans(Name, Class, Food, Habitat)?$$

9.2.3 Test Case 3

A disjunctive query with built-ins that asks for name, class and habitat of animals that belong to class amphibians or class reptiles.

$$\begin{aligned} Ans(Name, Class, Habitat) &\leftarrow Animal(Name, Class, Food), \\ &\quad Habitat(Name, Habitat), \\ &\quad Class = "amphibian". \\ Ans(Name, Class, Habitat) &\leftarrow Animal(Name, Class, Food), \\ &\quad Habitat(Name, Habitat), \\ &\quad Class = "reptile". \end{aligned} \tag{9.21}$$

In this case also, the logic programs for scenario 1 and 2 remains the same as test case 1. Scenario 1 considers everything irrespective of the query. Scenario 2 again considers the global relations *Animal* and *Habitat* but not the built-ins in the query.

Program 9.3c for scenario 3:

$$\begin{aligned} &\#import(animalkingdom, "sa", "gj", "Select\ Distinct\ *\ From\ V1\ where \\ &\qquad Class =' amphibian'\ and\ Food =' insects'", \\ &\qquad V1, type : Q\_Const, Q\_Const, Q\_Const). \\ &\#import(animalkingdom, "sa", "gj", "Select\ Distinct\ *\ From\ LV1\ where \\ &\qquad Class =' amphibian'\ and\ Food =' insects'", \\ &\qquad LV1, type : Q\_Const, Q\_Const, Q\_Const). \\ &\#import(mysqltest, "root", "gj", "Select\ Distinct\ *\ From\ LV6\ where \\ &\qquad Class =' amphibian'\ and\ Food =' insects'", \\ &\qquad LV6, type : Q\_Const, Q\_Const, Q\_Const). \\ &\#import(animalhabitat, "root", "gj", "Select\ Distinct\ *\ From\ V2", \\ &\qquad V2, type : Q\_Const, Q\_Const). \\ &\#import(animalhabitat, "root", "gj", "Select\ Distinct\ *\ From\ LV2", \\ &\qquad LV2, type : Q\_Const, Q\_Const). \\ &\#import(animalhabitat, "root", "gj", "Select\ Distinct\ *\ From\ V6", \\ &\qquad V6, type : Q\_Const). \\ &\#import(animalkingdom, "sa", "gj", "Select\ Distinct\ *\ From\ V1\ where \\ &\qquad Class =' reptile'", \\ &\qquad V1, type : Q\_Const, Q\_Const, Q\_Const). \\ &\#import(animalkingdom, "sa", "gj", "Select\ Distinct\ *\ From\ LV1\ where \\ &\qquad Class =' reptile'", \\ &\qquad LV1, type : Q\_Const, Q\_Const, Q\_Const). \end{aligned}$$

$$\#import(mysqltest,"root","gj","Select\ Distinct\ *\ FROM\ V7\ where\ Class =' reptile'",V7,type:Q\_Const,Q\_Const,Q\_Const).$$

$$Animal(Name,Class,Food):-V1(Name,Class,Food).$$

$$Animal(Name,Class,Food):-LV1(Name,Class,Food).$$

$$Animal(Name,Class,Food):-V2(Name,Habitat),chosen8(Name,Habitat,Class),chosen9(Name,Habitat,Food).$$

$$chosen8(Name,Habitat,Class):-V2(Name,Habitat),dom(Class),not\ diffchoice8(Name,Habitat,Class).$$

$$diffchoice8(Name,Habitat,Class):-chosen8(Name,Habitat,U),dom(Class),U!=Class.$$

$$chosen9(Name,Habitat,Food):-V2(Name,Habitat),dom(Food),not\ diffchoice9(Name,Habitat,Food).$$

$$diffchoice9(Name,Habitat,Food):-chosen9(Name,Habitat,U),dom(Food),U!=Food.$$

$$Habitat(Name,Habitat):-V2(Name,Habitat).$$

$$Animal(Name,Class,Food):-LV2(Name,Habitat),chosen10(Name,Habitat,Class),chosen11(Name,Habitat,Food).$$

$$chosen10(Name,Habitat,Class):-LV2(Name,Habitat),dom(Class),not\ diffchoice10(Name,Habitat,Class).$$

$$diffchoice10(Name,Habitat,Class):-chosen10(Name,Habitat,U),dom(Class),U!=Class.$$

$$chosen11(Name,Habitat,Food):-LV2(Name,Habitat),dom(Food),not\ diffchoice11(Name,Habitat,Food).$$

$$diffchoice11(Name,Habitat,Food):-chosen11(Name,Habitat,U),dom(Food),U!=Food.$$

$$Habitat(Name,Habitat):-LV2(Name,Habitat).$$

$$Animal(Name,Class,Food):-V6(Habitat),chosen12(Habitat,Name),chosen13(Habitat,Class),chosen14(Habitat,Food).$$

$$chosen12(Habitat,Name):-V6(Habitat),dom(Name),not\ diffchoice12(Habitat,Name).$$

$$diffchoice12(Habitat,Name):-chosen12(Habitat,U),dom(Name),U!=Name.$$

$$chosen13(Habitat,Class):-V6(Habitat),dom(Class),not\ diffchoice13(Habitat,Class).$$

$$diffchoice13(Habitat,Class):-chosen13(Habitat,U),dom(Class),U!=Class.$$

$$
\begin{aligned}
chosen14(Habitat, Food) :- & V6(Habitat), dom(Food), \\
& not\ diffchoice14(Habitat, Food). \\
diffchoice14(Habitat, Food) :- & chosen14(Habitat, U), \\
& dom(Food), U! = Food. \\
Habitat(Name, Habitat) :- & V6(Habitat), chosen12(Habitat, Name). \\
Animal(Name, "amphibian", Food) :- & LV6(Name, Class, Food). \\
Animal(Name, "reptile", Food) :- & V7(Name, Class, Food). \\
Ans(Name, Class, Habitat) :- & Habitat(Name, Habitat), \\
& Animal(Name, Class, Food), \\
& Class = "amphibian", Food = "insects". \\
Ans(Name, Class, Habitat) :- & Habitat(Name, Habitat), \\
& Animal(Name, Class, Food), \\
& Class = "reptile". \\
Ans(Name, Class, Habitat)? &
\end{aligned}
$$

9.2.4 Test Case 4

A cartesian product query that asks for name and habitat of animals.

$$
\begin{aligned}
Ans(Name, Habitat) \ \leftarrow \ & Habitat(Name, Habitat), ClassList(Class), \\
& FoodList(Food), Vertebrate(Name). \qquad (9.22)
\end{aligned}
$$

The logic program for scenario 1 remains the same as test case 1. Scenario 2 considers all the global relations in the body of the query namely $Vertebrate$, $ClassList$, $FoodList$ and $Habitat$.

Program 9.4b for scenario 2:

$$
\begin{aligned}
& \#import(animalkingdom, "test", "test", "select\ *\ from\ V1", V1, \\
& \qquad type : Q\_Const, Q\_Const, Q\_Const). \\
& \#import(animalkingdom, "test", "test", "select\ *\ from\ LV1", LV1, \\
& \qquad type : Q\_Const, Q\_Const, Q\_Const). \\
& \#import(animalkingdom, "test", "test", "select\ *\ from\ FoodV1", FoodV1, \\
& \qquad type : Q\_Const). \\
& \#import(animalkingdom, "test", "test", "select\ *\ from\ ClassV1", ClassV1, \\
& \qquad type : Q\_Const). \\
& \#import(animalhabitat, "test", "test", "select\ *\ from\ V2", V2, \\
& \qquad type : Q\_Const, Q\_Const).
\end{aligned}
$$

$$\#import(animalhabitat,"test","test","select\ *\ from\ LV2",LV2,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V6",V6,$$
$$type:Q\_Const).$$
$$\#import(accesstest,"","","select\ *\ from\ VTest1",VTest1,$$
$$type:Q\_Const).$$
$$\#import(accesstest,"","","select\ *\ from\ VTest2",VTest2,$$
$$type:Q\_Const,Q\_Const).$$
$$Vertebrate(Name):-VTest1(Name).$$
$$Vertebrate(Name):-VTest2(Name).$$
$$Vertebrate(Name):-V1(Name,Class,Food).$$
$$Vertebrate(Name):-LV1(Name,Class,Food).$$
$$Habitat(Name,Habitat):-V2(Name,Habitat).$$
$$Habitat(Name,Habitat):-LV2(Name,Habitat).$$
$$Habitat(Name,Habitat):-V6(Habitat),f12(Habitat,Name).$$
$$f12(Habitat,Name):-V6(Habitat),dom(Name),$$
$$chosen12(Habitat,Name).$$
$$chosen12(Habitat,Name):-V6(Habitat),dom(Name),$$
$$not\ diffchoice12(Habitat,Name).$$
$$diffchoice12(Habitat,Name):-chosen12(Habitat,U),$$
$$dom(Name),U!=Name.$$
$$FoodList(Food):-FoodV1(Food).$$
$$ClassList(Class):-ClassV1(Class).$$
$$Ans(Name,Habitat):-Habitat(Name,Habitat),$$
$$Vertebrate(Name),FoodList(Food),$$
$$ClassList(Class).$$
$$Ans(Name,Habitat)?$$

Program 9.4c for scenario 3:

$$\#import(animalkingdom,"test","test","select\ *\ from\ V1",V1,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalkingdom,"test","test","select\ *\ from\ LV1",LV1,$$
$$type:Q\_Const,Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ V2",V2,$$
$$type:Q\_Const,Q\_Const).$$
$$\#import(animalhabitat,"test","test","select\ *\ from\ LV2",LV2,$$
$$type:Q\_Const,Q\_Const).$$

$$\#import(animalhabitat,"test","test","select * from\ V6",V6,$$
$$type : Q\_Const).$$
$$\#import(accesstest,"","","select * from\ VTest1",VTest1,$$
$$type : Q\_Const).$$
$$\#import(accesstest,"","","select * from\ VTest2",VTest2,$$
$$type : Q\_Const, Q\_Const).$$
$$Vertebrate(Name) : -VTest1(Name).$$
$$Vertebrate(Name) : -VTest2(Name).$$
$$Vertebrate(Name) : -V1(Name, Class, Food).$$
$$Vertebrate(Name) : -LV1(Name, Class, Food).$$
$$Habitat(Name, Habitat) : -V2(Name, Habitat).$$
$$Habitat(Name, Habitat) : -LV2(Name, Habitat).$$
$$Habitat(Name, Habitat) : -V6(Habitat), chosen12(Habitat, Name).$$
$$chosen12(Habitat, Name) : -V6(Habitat), dom(Name),$$
$$not\ diffchoice12(Habitat, Name).$$
$$diffchoice12(Habitat, Name) : -chosen12(Habitat, U),$$
$$dom(Name), U! = Name.$$
$$Ans(Name, Habitat) : -Habitat(Name, Habitat),$$
$$Vertebrate(Name).$$
$$Ans(Name, Habitat)?$$

9.3 Experimental Results

We discuss the results obtained for the test cases in the three scenarios.

Test Case 1: The gain obtained in scenario 3, is in terms of number of rules used in the logic program and a small improvement in execution time.

Test Case 2: In scenario 3, we see improvement because we use the SP, SQC and RP steps and get a reduced set of sources, imported tuples and rules in the logic program.

Test Case 3: In scenario 3, again we see improvement because we use the SP, SQC and RP steps and get a reduced set of sources, tuples and rules for both parts of the disjunctive query (one asking for *amphibians* and the other asking for *reptiles*).

Test Case 4: In scenario 3, again, we see some optimization. We first prune the query using QP step and consider only the global relations $Habitat$ and $Vertebrate$.

We use the following criteria for each query to compare our results,

(a) Number of source relations identified.

(b) Number of tuples imported into the logic program.

(c) Execution time.

(d) Number of Rules generated by the *Simple Specification* program (without *dom* atoms).

The results obtained for the test cases are shown in Table 9.2.

Table 9.2: Experimental Results.

Test Case	Scenario	Number of Source Relations	Imported Tuples	Execution Time	Number of Rules
1	1	18	41968	35.8s	50
	2	13	40868	19.6s	43
	3	13	40868	17.4s	34
2	1	18	41968	33.5s	50
	2	13	40868	19.4s	43
	3	6	6395	8.6s	23
3	1	18	41968	34.7s	50
	2	13	40868	19.8s	43
	3	7	12526	9.8s	24
4	1	18	41968	30.6s	50
	2	9	14264	8.5s	12
	3	7	13664	7.1s	9

From the results, we see performance improvements in every test case listed in Table 9.2. This is especially significant in the case of queries that have built-ins involving the equality operator. We also analyze the cost associated with having equality built-ins in the query. Figure 9.1 shows the execution time for different number of equality built-ins in the query. The execution time increases linearly as the number of equality built-ins in the query.

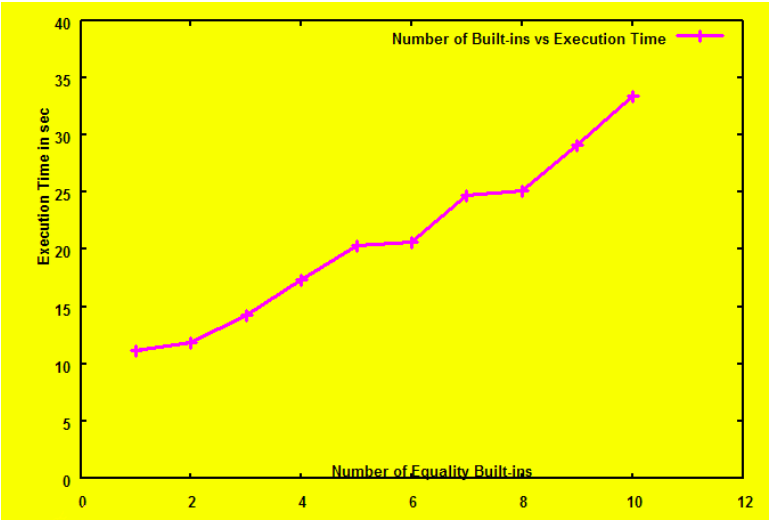


Figure 9.1: Execution time Vs No. Of Equality Built-ins.

Chapter 10

Conclusions

In this thesis, we showed a metadata representation that is used to specify the entire metadata under the LAV approach. The metadata included the structure of global schema, local schema and their access information, integrity constraints and LAV mappings. The metadata representation used a combination of custom XML elements and RuleML elements. The queries on the XML representation employed the query language, XQuery to extract the required information.

We also showed pruning techniques that is used in conjunction with *Extended Inverse Rules Algorithm* (EIRA) for computing *certain* answers. Generally, there will be many sources available to a mediator and the mediator will contain many mappings. A query answering algorithm requires an optimization step to identify those source relations that are required for answering a query. In this respect, we addressed some of the open research issues discussed in [15], such as optimizing the interaction of the logic programming system with the underlying databases.

We showed how we optimize at the query level, source level and the rules. In the case of EIRA, our pruning approach reduces the atoms in the stable models. The specification program contains only the rules corresponding to the relevant source relations. We also import only the relevant data, based on the built-ins in the query, into the logic program. We also showed how a query using cartesian product can be pruned of some global predicates. We showed a modification of *Simple Specification* (that can also be applied to the *refined specification*) that eliminates redundant rules. We showed how source relations that violate the equality built-ins in the query can be

pruned from the list of sources considered. We applied the source pruning technique to conjunctive queries and also showed that it works for disjunctive queries with equality conditions. We presented experimental results proving the optimization techniques makes the computation of *certain* answers more efficient as compared to retrieving the source data using only the global predicates from the body of the query.

We also showed how the XML representation of metadata is queried and then parsed to generate the program for *Simple Specification* . We used the same parser to generate the program for *Refined Specification* and *Inverse Rules*. We also described the general architecture of a mediator *VISS* describing the components and the implementation of the mediator using open source tools. We present some possible future work in this research and other related work in this area.

10.1 Future Work

When representing metadata using XML, we may be forced to specify the data types of attributes in the source relations. The representation of the data types is especially significant when we specify the mappings where similar attributes in two source relations are of different data types. For example, the average life span of an animal could be defined as a number data type (e.g. 25) in one database and a character data type (e.g. 25 yrs) in another. The metadata representation may have to include more specific information, such as the data types, to accurately describe the data.

The source pruning step currently checks the violation of equality built-ins in the query when determining required sources. The SP step could be extended to perform additional checks in the presence of comparison predicates such as <= and >=. For example, consider a query for endangered species of animals whose population does not exceed 100 (i.e. <= 100). If there is a mapping that defines a source relation as containing animals whose current population exceed 2000 (i.e. >= 2000), then we can eliminate this relation for computing answer to the query on endangered species. However, this becomes more complex when there are combinations of comparison predicates in the same query. It would also be interesting to extend and verify the

pruning techniques in the presence of aggregation and grouping functions in queries.

VISS generates the refined specifications of the minimal instances, which in the presence of global integrity constraints can be used for computing *consistent* answers. A similar approach is used in the *Consistency Extractor System* [25], which works on single and possibly inconsistent database. When global integrity constraints are present, the specifications of the class of legal instances has to include the global relations and their dependencies, that are transitively connected to the relations in the query and the integrity constraints. We showed how ICs are specified in the XML metadata and described an idea of obtaining the relevant predicates from the query and ICs using user-defined functions in XQuery. Apart from the ICs specified in the XML metadata, ICs can also be specified along with the user query. These ICs will be in the language of the user query and will have to be taken into account for computing *consistent* answers.

10.2 Comparison to Related Work

We mention existing work in the area of query answering and mediator systems. Queries with comparison predicates have been studied in [71] [47]. [5] discusses queries with aggregations in the presence of functional dependencies. Queries with disjunctive views are analyzed in [4]. We presented metadata representations in other mediator systems and existing approaches for detecting relevant sources that are used in *Bucket* and *Minicon* algorithms. [25] provides optimization techniques using *Magic Sets* in repair programs for computing *consistent* answers. The pruning methods in [25] detect required portions of the logic program based on program constraints (ICs) specified with the user query. [3] discuss optimization methods that use query result caching mechanism to use results from previous execution of queries.

The design issues in data integration systems have been studied in [10] [12] [63]. Mediator systems such as Garlic [42], Disco [67], TSIMMIS [35], Silkroute [33] and Xperanto [26] use the *Global as View* approach. Data integration applications such as e-xmlmedia [36] uses XPath, a language for querying elements in XML documents,

to describe a data source. The mediator uses an XPath guide, which is a list of all paths that can be queried in a data source. The data sources are provided to the mediator as XML documents. [61] presents a mediator architecture using XQuery for metadata mappings under the GAV approach.

10.3 Concluding Remarks

Our metadata representation provided a basis for specifying data source access information, integrity constraints and mappings all within a single XML document stored in the mediator. Thus, querying involves accessing a single metadata store for retrieving all the required information. Additionally, our pruning techniques provide optimization by localizing computations to the relevant parts of the user query, data sources, data and mapping rules. This makes the computation of *certain* answers efficient in the presence of large number of data sources that contain large amounts of data. *VISS* provides an infrastructure for virtually integrating multiple data sources according to the LAV approach. Concrete data integration system is specified in terms of metadata using an XML/RuleML representation.

Bibliography

[1] S. Abiteboul and O. Duschka, "Complexity of answering queries using materialized views," in *ACM Symposium on Principles of Database Systems*. ACM Press, 1998, pp. 254–263.

[2] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian, "Query caching and optimization in distributed mediator systems," in *ACM SIGMOD Conference on Management of Data*. ACM Press, 1996, pp. 137–148.

[4] F. N. Afrati, M. Gergatsoulis, and T. G. Kavalieros, "Answering queries using materialized views with disjunctions," in *7th International Conference on Database Theory*. Springer LNCS 1540, 1999, pp. 435–452.

[5] M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad, "Scalar aggregation in inconsistent databases," *Theoretical Computer Science*, vol. 296, no. 3, pp. 405–434, 2003.

[6] Y. Arens, C. Hsu, and C. Knoblock, "Query processing in the sims information mediator," in *Readings in agents*. Morgan Kaufmann, 1998, pp. 82–90.

[7] M. Ball, H. Boley, D. Hirtle, J. Mei, and B. Spencer, "Implementing ruleml using schemas, translators, and bidirectional interpreters," in *W3C Workshop on Rule Languages for Interoperability*. W3C, 2005.

[8] P. Barcelo, L. Bertossi, and L. Bravo, "Characterizing and computing semantically correct answers from databases with annotated logic and answer sets," in *Semantics of Databases*. Springer LNCS 2582, 2001, pp. 7–33.

[9] C. Baru, A. Gupta, B. Ludascher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu, "Xml-based information mediation with mix," in *SIGMOD Conference*. ACM Press, 1999, pp. 597–599.

[10] C. Batini, M. Lenzerini, and S. B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys*, vol. 18, no. 4, pp. 323–364, 1986.

[11] D. Beneventano and S. Bergamaschi, "The momis methodology for integrating heterogeneous data sources," in *International Federation for Information Processing*, vol. 156. Springer Boston, 2004, pp. 19–24.

[12] S. Bergamaschi, S. Castano, M. Vincini, and D. Beneventano, "Semantic integration of heterogeneous information sources," *Data and Knowledge Engineering*, vol. 36, no. 3, pp. 215–249, 2001.

[13] P. Bernstein and L. Haas, "Information integration in the enterprise," *Communications of the ACM*, vol. 51, no. 9, pp. 72–79, 2008.

[14] L. Bertossi, "Consistent query answering in databases," *ACM Sigmod Record*, vol. 35, no. 2, pp. 68–76, 2006.

[15] L. Bertossi and L. Bravo, "Consistent query answers in virtual data integration systems," in *Inconsistency Tolerance*. Springer LNCS 3300, 2005, pp. 42–83.

[16] L. Bertossi, J. Chomicki, A. Cortes, and C. Gutierrez, "Consistent answers from integrated data sources," in *International Conference on Flexible Query Answering Systems*. Springer LNCS 2522, 2002, pp. 71–85.

[17] H. Boley, S. Tabet, and G. Wagner, "Design rationale for ruleml: A markup language for semantic web rules," in *Semantic Web and Web Services*. Stanford University, 2001, pp. 381–401.

[18] L. Bravo and L. Bertossi, "Logic programs for consistently querying data integration systems," in *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 2003, pp. 10–15.

[19] L. Bravo and L. Bertossi, "Deductive databases for computing certain and consistent answers to queries from mediated data integration systems," *Journal of Applied Logic*, vol. 3, no. 1, pp. 329–367, 2005.

[20] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (xml)," *World Wide Web Journal*, vol. 2, no. 4, pp. 27–66, 1997.

[21] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan, "Reasoning about keys for xml," *Information Systems Journal*, vol. 28, no. 8, pp. 1037–1063, 2003.

[22] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, "On the role of integrity constraints in data integration," *IEEE Data Engineering Bulletin*, vol. 25, no. 3, pp. 39–45, 2002.

[23] D. Calvanese, D. De Giacomo, M. Lenzerini, and R. Rosati, "Logical foundations of peer-to-peer data integration," in *23rd symposium on Principles of database systems*. ACM, 2004, pp. 241–251.

[24] M. Caniupan and L. Bertossi, "Optimizing repair programs for consistent query answering," in *25th International Conference of the Chilean Computer Science Society*. IEEE Computer Society, 2005, pp. 3–12.

[25] M. Caniupan and L. Bertossi, "The consistency extractor system: Querying inconsistent databases using answer set programs," in *Scalable Uncertainty Management Conference*. Springer LNCS 4772, 2007, pp. 74–88.

[26] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian, "Xperanto: Publishing object-relational data as xml," in *Third International Workshop on the Web and Databases*. Citeseer, 2000, pp. 105–110.

[27] M. Carey, L. Haas, P. Schwarz, M. Arya, W. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. Williams, and E. Wimmers, "Towards heterogeneous multimedia information systems: The garlic approach," in *5th International Workshop on Research Issues in Data Engineering: Distributed Object Management*, 1995, pp. 124–131.

[28] D. Chamberlin, M. Stefanescu, D. Florescu, J. Robie, and J. Simeon, "Xquery: A query language for xml," W3C, Tech. Rep., 2001.

[29] C. Delobel, C. Reynaud, M. Rousset, J. Sirot, and D. Vodislav, "Semantic integration in xyleme: a uniform tree-based approach," *Data and Knowledge Engineering*, vol. 44, no. 3, pp. 267–298, 2003.

[30] D. Draper, A. Halevy, and D. Weld, "The nimble xml data integration system," in *International Conference on Data Engineering*, 2001, pp. 155–160.

[31] O. Duschka, M. Genesereth, and A. Levy, "Recursive query plans for data integration," *Journal of Logic Programming*, vol. 43, no. 1, pp. 49–73, 2000.

[32] T. Eiter, G. Gottlob, and H. Mannila, "Disjunctive datalog," *ACM Transactions on Database Systems*, vol. 22, no. 3, pp. 364–418, 1997.

[33] M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. Tan, "Silkroute: A framework for publishing relational data in xml," *ACM Transactions on Database Systems*, vol. 27, no. 4, pp. 438–493, 2002.

[34] M. Friedman, A. Levy, and T. Millstein, "Navigational plans for data integration," in *National Conference on Artificial Intelligence*, 1999, pp. 67–73.

[35] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom, "Integrating and accessing heterogeneous information sources in tsimmis," in *AAAI Symposium on Information Gathering*, 1995, pp. 61–64.

[36] G. Gardarin, A. Mensch, T. Tuyet Dang-Ngoc, and L. Smit, "Integrating heterogeneous data sources with xml and xquery," in *13th International Workshop*

on Database and Expert Systems Applications. IEEE Computer Society, 2002, pp. 839–846.

[37] M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New Generation Computing*, vol. 9, pp. 365–385, 1991.

[38] M. Genesereth, A. Keller, and O. Duschka, "Infomaster: An information integration system," in *ACM SIGMOD Conference*. ACM Press, 1997, pp. 539–542.

[39] F. Giannotti, D. Pedreschi, D. Sacca, and C. Zaniolo, "Non-determinism in deductive databases," in *2nd International Conference on Deductive and Object-Oriented Databases*, 1991, pp. 129–146.

[40] F. Goasdoue, V. Lattes, and M. Rousset, "The use of carin language and algorithms for information integration: The picsel system," *International Journal on Cooperative Information Systems*, vol. 9, no. 4, pp. 383–401, 2000.

[41] G. Grahne and A. O. Mendelzon, "Tableau techniques for querying information sources through global schemas," in *7th International Conference on Database Theory*. Springer LNCS 1540, 1999, pp. 332–347.

[42] L. Haas, R. Miller, B. Niswonger, M. Roth, P. Schwarz, and E. Wimmers, "Transforming heterogeneous data with database middleware: Beyond integration," *IEEE Data Engineering Bulletin*, vol. 22, no. 1, pp. 31–36, 1999.

[43] A. Halevy, "Answering queries using views: A survey," *Journal of Very Large Databases*, vol. 10, no. 4, pp. 270–294, 2001.

[44] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld, "An adaptive query execution system for data integration," in *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1999, pp. 299–310.

[45] E. Jasper, N. Tong, P. McBrien, and A. Poulovassilis, "View generation and optimisation in the automed data integration framework," in *15th Conference on Advanced Information Systems Engineering*, ser. CEUR Workshop Proceedings, vol. 74. CEUR-WS.org, 2003.

[46] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava, "The information manifold," in *AAAI Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, 1995, pp. 85–91.

[47] A. Klug, "On conjunctive queries containing inequalities," *Journal of the ACM*, vol. 35, no. 1, pp. 146–160, 1988.

[48] W. Labio, Y. Zhuge, J. Wiener, H. Gupta, H. Garcia-Molina, and J. Widom, "The whips prototype for data warehouse creation and maintenance," in *International Conference on Data Engineering*. IEEE Computer Society, 1997, pp. 557–559.

[49] K. Lee, J. Min, K. Park, and K. Lee, "A design and implementation of xml-based mediation framework (xmf) for integration of internet information resources," in *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002, pp. 2700–2708.

[50] M. Lenzerini, "Data integration: A theoretical perspective," in *ACM Symposium on Principles of Database Systems*. ACM, 2002, pp. 233–246.

[51] N. Leone, V. Lio, and G. Terracina, "Dlvdb: Bridging the gap between asp systems and dbmss," in *International Conference on Logic Programming and Nonmonotonic Reasoning*, 2004, pp. 341–345.

[52] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The dlv system for knowledge representation and reasoning," *ACM Transactions on Computational Logic*, vol. 7, no. 3, pp. 499–562, 2006.

[53] A. Levy, "Logic-based techniques in data integration," in *Logic-based artificial intelligence*. J. Minker (ed.), Kluwer Academic Publishers, 2000, pp. 575–595.

[54] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering queries using views," in *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 1995, pp. 95–104.

[55] A. Y. Levy, A. Rajaraman, and J. J. Ordille, "Querying heterogeneous information sources using source descriptions," in *Twenty-second International Conference on Very Large Databases*. Morgan Kaufmann, 1996, pp. 251–262.

[56] J. Lu, A. Nerode, and V. Subrahmanian, "Hybrid knowledge bases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 5, pp. 773–785, 1996.

[57] B. Ludscher, Y. Papakonstantinou, and P. Velikhov, "Navigation-driven evaluation of virtual mediated views," in *7th International Conference on Extending Database Technology*. Springer LNCS 1777, 2000, pp. 150–165.

[58] I. Manolescu, D. Florescu, and D. Kossmann, "Answering xml queries over heterogeneous data sources," in *International Conference on Very Large Data Bases*, 2001, pp. 241–250.

[59] P. Mitra, "An algorithm for answering queries efficiently using views," in *12th Australasian Database Conference*, 2001, pp. 99–106.

[60] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman, "Medmaker: A mediation system based on declarative specifications," in *Proceedings of the 12th International Conference on Data Engineering*. IEEE Computer Society, 1996, pp. 132–141.

[61] X. Peng, R. Brazile, and K. Swigger, "Using xquery to describe mappings from global schemas to local data sources," in *IEEE International Conference*. IEEE Systems, Man, and Cybernetics Society, 2004, pp. 97–102.

[62] R. Pottinger and A. Halevy, "Minicon: A scalable algorithm for answering queries using views," *The International Journal on Very Large Data Bases*, vol. 10, no. 2-3, pp. 182–198, 2001.

[63] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[64] J. Simeon and S. Cluet, "Using yat to build a web server," in *International Workshop on the Web and Databases*, ser. Lecture Notes in Computer Science, vol. 1590. Springer, 1998, pp. 118–135.

[65] C. M. Sperberg-McQueen, H. S. Thompson, M. Maloney, D. Beech, N. Mendelsohn, and S. Gao, "Xml schema definition language (xsd) 1.1 part 1: Structures," W3C, Tech. Rep., 2009.

[66] D. Theodoratos, S. Ligoudistianos, and T. Sellis, "Designing the global data warehouse with spj views," in *11th Conference on Advanced Information Systems Engineering*. Springer, 1999, pp. 180–194.

[67] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling heterogeneous database and the design of disco," in *16th International Conference on Distributed Computing Systems*, 1996, pp. 449–457.

[68] J. Ullman, "Information integration using logical views," in *6th International Conference on Database Theory*, ser. Lecture Notes in Computer Science, vol. 1186. Springer, 1997, pp. 19–40.

[69] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, vol. 25, no. 3, pp. 38–49, 1992.

[70] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom, "A system prototype for warehouse view maintenance," in *Workshop on Materialised Views: Techniques and Applications*, 1996, pp. 26–33.

[71] H. Z. Yang and P. Larson, "Query transformation for psj-queries," in *13th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1987, pp. 245–254.

[72] G. Zhou, R. Hull, R. King, and J. Franchitti, "Data integration and warehousing using h2o," *IEEE Data Engineering Bulletin*, vol. 18, no. 2, pp. 29–40, 1995.

[73] "Anatomy of an xml database: Oracle berkeley db xml." Available at http://www.oracle.com/technology/products/berkeley-db/xml/index.html, oracle.

[74] "Standard template library programmer's guide," Available at http://www.sgi.com/tech/stl/, 2000.

[75] "Validator for xml schema 20010502 version," Available at http://www.w3.org/2001/03/webdata/xsv, 2007, w3C.

[76] "Xerces-c++ xml parser," *Available at http://xerces.apache.org/xerces-c/*, 2007.

Appendix A: More Experiments

We list experimental results for more query types in this section. We use a black-box approach where we run queries of certain types and compute cost in terms of number of source relations used and number of rules in the logic program excluding the dom rules and query program. We show efficiency of our approach when compared to the original EIRA and not with other approaches. There are currently no benchmark standards for testing data integration systems and so we use test data from various domains (such as movies, animals, conferences) to do our experiments. We use the datasources in Chapter 9 and the following:

Datasource: MovieDB

$$\begin{aligned}
MovieV1(Title, Year, Director) &\leftarrow Movie(Title, Year, Director), \\
&\quad American(Director), Year >= 1960. \\
MovieV6(Title, Year) &\leftarrow Movie(Title, Year, Director), \\
&\quad American(Director), Year >= 1960, \\
&\quad Director = "Spielberg". \\
MovieV3(Title, Year) &\leftarrow Movie(Title, Year, Director), \\
&\quad American(Director), Year >= 1960, \\
&\quad Director = "Scorcese". \\
MovieV5(Title, Year, Director) &\leftarrow Movie(Title, Year, Director), \\
&\quad American(Director).
\end{aligned}$$

Datasource: IMDB

$$\begin{aligned}
MovieV2(Title, Review) &\leftarrow Movie(Title, Year, Director), \\
&\quad Review(Title, Review), Title = "signs", \\
&\quad Year >= 1990. \\
MovieV4(Title, Review) &\leftarrow Movie(Title, Year, Director), \\
&\quad Review(Title, Review), Title = "avatar", \\
&\quad Year >= 1990. \\
ReviewV2(Title, Review) &\leftarrow Review(Title, Review).
\end{aligned}$$

Datasource: ConferenceDB

$$AAAIDB(Title) \leftarrow AAAIPapers(Title).$$

Datasource: PaperDB

$$\begin{aligned}
CitationDB(Title1, Title2) &\leftarrow Cites(Title1, Title2). \\
AwardDB(Title) &\leftarrow AwardPaper(Title).
\end{aligned}$$

Test Case A1: Query with Comparison Operators and Equality Built-ins

$$Ans(Title, Year, Review) \ :- \ Movie(Title, Year, Director), Review(Title, Review), Year >= "1995", Director == "TimBurton".$$

$Ans(Title, Year, Review)$?

Program using VISS:

$$\begin{aligned}
&\#import(moviedb, "sa", "gj", "SELECT\ Distinct\ *\ FROM\ MovieV1 \\
&\qquad where\ Director =' TimBurton'\ and \\
&\qquad Year >=' 1995'", MovieV1, \\
&\qquad type : Q_CONST, Q_CONST, Q_CONST).\\
&\#import(moviedb, "sa", "gj", "SELECT\ Distinct\ *\ FROM\ MovieV5 \\
&\qquad where\ Director =' TimBurton'\ and \\
&\qquad Year >=' 1995'", MovieV5, \\
&\qquad type : Q_CONST, Q_CONST, Q_CONST).\\
&\#import(imdb, "root", "gj", "SELECT\ Distinct\ *\ FROM\ MovieV4", \\
&\qquad MovieV4, type : Q_CONST, Q_CONST).\\
&\#import(imdb, "root", "gj", "SELECT\ Distinct\ *\ FROM\ MovieV2", \\
&\qquad MovieV2, type : Q_CONST, Q_CONST).\\
&\#import(imdb, "root", "gj", "SELECT\ Distinct\ *\ FROM\ ReviewV2", \\
&\qquad ReviewV2, type : Q_CONST, Q_CONST).\\
&Movie(Title, Year, Director) :- MovieV1(Title, Year, Director).\\
&Movie(Title, Year, Director) :- MovieV2(Title, Review), \\
&\qquad chosen1(Title, Review, Year), \\
&\qquad chosen2(Title, Review, Director).\\
&chosen1(Title, Review, Year) :- MovieV2(Title, Review), dom(Year), \\
&\qquad not\ diffchoice1(Title, Review, Year).\\
&diffchoice1(Title, Review, Year) :- chosen1(Title, Review, U), dom(Year), \\
&\qquad U\ != Year.\\
&chosen2(Title, Review, Director) :- MovieV2(Title, Review), dom(Director), \\
&\qquad not\ diffchoice2(Title, Review, Director).\\
&diffchoice2(Title, Review, Director) :- chosen2(Title, Review, U), dom(Director), \\
&\qquad U\ != Director.\\
&Movie(Title, Year, Director) :- MovieV4(Title, Review), \\
&\qquad chosen3(Title, Review, Year), \\
&\qquad chosen4(Title, Review, Director).\\
&chosen3(Title, Review, Year) :- MovieV4(Title, Review), dom(Year), \\
&\qquad not\ diffchoice3(Title, Review, Year).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}3(Title, Review, Year) :- & \mathit{chosen}3(Title, Review, U), \mathit{dom}(Year), \\
& U\ !\!= Year.\\
\mathit{chosen}4(Title, Review, Director) :- & \mathit{MovieV}4(Title, Review), \mathit{dom}(Director), \\
& \mathit{not}\ \mathit{diffchoice}4(Title, Review, Director).\\
\mathit{diffchoice}4(Title, Review, Director) :- & \mathit{chosen}4(Title, Review, U), \mathit{dom}(Director), \\
& U\ !\!= Director.\\
Movie(Title, Year, Director) :- & \mathit{MovieV}5(Title, Year, Director).\\
Review(Title, Review) :- & \mathit{ReviewV}2(Title, Review).
\end{aligned}$$

Test Case A2: Query with Comparison Operators and No Equality Built-ins

$$\begin{aligned}
Ans(Title, Year, Review) \ :- \ & Movie(Title, Year, Director), Review(Title, Review), \\
& Year <= \text{"1995"}.
\end{aligned}$$

$Ans(Title, Year, Review)$?

Program using VISS:

$$\begin{aligned}
& \#import(moviedb, \text{"}sa\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ MovieV1 \\
& \qquad where\ Year <= 1995'\text{"}, MovieV1, \\
& \qquad type : Q_CONST, Q_CONST, Q_CONST).\\
& \#import(moviedb, \text{"}sa\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ MovieV3 \\
& \qquad where\ Year <= 1995'\text{"}, MovieV3, \\
& \qquad type : Q_CONST, Q_CONST).\\
& \#import(moviedb, \text{"}sa\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ MovieV6 \\
& \qquad where\ Year <= 1995'\text{"}, MovieV6, \\
& \qquad type : Q_CONST, Q_CONST).\\
& \#import(moviedb, \text{"}sa\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ MovieV5 \\
& \qquad where\ Year <= 1995'\text{"}, MovieV5, \\
& \qquad type : Q_CONST, Q_CONST, Q_CONST).\\
& \#import(imdb, \text{"}root\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ MovieV4\text{"}, \\
& \qquad MovieV4, type : Q_CONST, Q_CONST).\\
& \#import(imdb, \text{"}root\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ MovieV2\text{"}, \\
& \qquad MovieV2, type : Q_CONST, Q_CONST).\\
& \#import(imdb, \text{"}root\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ ReviewV2\text{"}, \\
& \qquad ReviewV2, type : Q_CONST, Q_CONST).\\
& Movie(Title, Year, Director) :- \mathit{MovieV}1(Title, Year, Director).\\
& Movie(Title, Year, Director) :- \mathit{MovieV}3(Title, Year), \\
& \qquad \mathit{chosen}1(Title, Year, Director).\\
& \mathit{chosen}1(Title, Year, Director) :- \mathit{MovieV}3(Title, Year), \mathit{dom}(Director), \\
& \qquad \mathit{not}\ \mathit{diffchoice}1(Title, Year, Director).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}1(Title, Year, Director) :- & \mathit{chosen}1(Title, Year, U), \mathit{dom}(Director), \\
& U \mathrel{!}= Director.\\
Movie(Title, Year, Director) :- & MovieV3(Title, Year), \\
& \mathit{chosen}2(Title, Year, Director).\\
\mathit{chosen}2(Title, Year, Director) :- & MovieV3(Title, Year), \mathit{dom}(Director), \\
& \mathit{not\ diffchoice}2(Title, Year, Director).\\
\mathit{diffchoice}2(Title, Year, Director) :- & \mathit{chosen}2(Title, Year, U), \mathit{dom}(Director), \\
& U \mathrel{!}= Director.\\
Movie(Title, Year, Director) :- & MovieV2(Title, Review), \\
& \mathit{chosen}3(Title, Review, Year), \\
& \mathit{chosen}4(Title, Review, Director).\\
\mathit{chosen}3(Title, Review, Year) :- & MovieV2(Title, Review), \mathit{dom}(Year), \\
& \mathit{not\ diffchoice}3(Title, Review, Year).\\
\mathit{diffchoice}3(Title, Review, Year) :- & \mathit{chosen}3(Title, Review, U), \mathit{dom}(Year), \\
& U \mathrel{!}= Year.\\
\mathit{chosen}4(Title, Review, Director) :- & MovieV2(Title, Review), \mathit{dom}(Director), \\
& \mathit{not\ diffchoice}4(Title, Review, Director).\\
\mathit{diffchoice}4(Title, Review, Director) :- & \mathit{chosen}4(Title, Review, U), \mathit{dom}(Director), \\
& U \mathrel{!}= Director.\\
Review(Title, Review) :- & MovieV2(Title, Review).\\
Movie(Title, Year, Director) :- & MovieV4(Title, Review), \\
& \mathit{chosen}5(Title, Review, Year), \\
& \mathit{chosen}6(Title, Review, Director).\\
\mathit{chosen}5(Title, Review, Year) :- & MovieV4(Title, Review), \mathit{dom}(Year), \\
& \mathit{not\ diffchoice}5(Title, Review, Year).\\
\mathit{diffchoice}5(Title, Review, Year) :- & \mathit{chosen}5(Title, Review, U), \mathit{dom}(Year), \\
& U \mathrel{!}= Year.\\
\mathit{chosen}6(Title, Review, Director) :- & MovieV4(Title, Review), \mathit{dom}(Director), \\
& \mathit{not\ diffchoice}6(Title, Review, Director).\\
\mathit{diffchoice}6(Title, Review, Director) :- & \mathit{chosen}6(Title, Review, U), \mathit{dom}(Director), \\
& U \mathrel{!}= Director.\\
Review(Title, Review) :- & MovieV4(Title, Review).\\
Movie(Title, Year, Director) :- & MovieV5(Title, Year, Director).\\
Review(Title, Review) :- & ReviewV2(Title, Review).\\
Movie(Title, Year, Director) :- & MovieV6(Title, Year), \\
& \mathit{chosen}7(Title, Year, Director).\\
\mathit{chosen}7(Title, Year, Director) :- & MovieV6(Title, Year), \mathit{dom}(Director), \\
& \mathit{not\ diffchoice}7(Title, Year, Director).\\
\mathit{diffchoice}7(Title, Year, Director) :- & \mathit{chosen}7(Title, Year, U), \mathit{dom}(Director), \\
& U \mathrel{!}= Director.
\end{aligned}$$

Test Case A3: Recursive Query with Equality Built-ins

$$
\begin{aligned}
Papers(Title) &:- AAAIpapers(Title), Title == \text{"1"}.\\
Papers(Title2) &:- Papers(Title1), Cites(Title1, Title2).\\
Ans(Title) &:- Papers(Title), AwardPaper(Title).\\
Ans(Title)?
\end{aligned}
$$

Program using VISS:

$$
\begin{aligned}
&\#import(paperdb, \text{"}root\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ AwardDB\\
&\qquad where\ Title =' 1'\text{"}, AwardDB, type : Q\_CONST).\\
&\#import(conferencedb, \text{"}sa\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ AaaiDB\\
&\qquad where\ Title =' 1'\text{"}, AaaiDB, type : Q\_CONST).\\
&\#import(paperdb, \text{"}root\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ CitationDB\text{"},\\
&\qquad CitationDB, type : Q\_CONST, Q\_CONST).\\
&AAAIpapers(Title) : -AaaiDB(Title).\\
&Cites(Title1, Title2) : -CitationDB(Title1, Title2).\\
&AwardPaper(Title) : -AwardDB(Title).
\end{aligned}
$$

Test Case A4: Query with No Existential Variables

$$
\begin{aligned}
Ans(Name) &:- Vertebrate(Name).\\
Ans(Name)?
\end{aligned}
$$

Program using VISS:

$$
\begin{aligned}
&\#import(animalkingdom, \text{"}sa\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ V1\text{"},\\
&\qquad V1, type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&\#import(animalkingdom, \text{"}sa\text{"}, \text{"}gj\text{"}, \text{"}SELECT\ Distinct\ *\ FROM\ LV1\text{"},\\
&\qquad LV1, type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&Vertebrate(Name) : -V1(Name, Class, Food).\\
&Vertebrate(Name) : -LV1(Name, Class, Food).
\end{aligned}
$$

Test Case A5: Query with Inequality Built-in

$$
\begin{aligned}
Ans(Name, Habitat) &:- Animal(Name, Class, Food), Habitat(Name, Habitat),\\
&\quad Habitat\ !=\text{"}ocean\text{"}.\\
Ans(Name, Habitat)?
\end{aligned}
$$

Program using VISS:

$$\#import(animalhabitat,"root","gj","SELECTDistinct * FROMV2 whereHabitat! ='ocean'",V2, type:Q\_CONST,Q\_CONST).$$
$$\#import(animalhabitat,"root","gj","SELECTDistinct * FROMLV2 whereHabitat! ='ocean'",LV2, type:Q\_CONST,Q\_CONST).$$
$$\#import(animalhabitat,"root","gj","SELECTDistinct * FROMV6 whereHabitat! ='ocean'",V6, type:Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECTDistinct * FROMV1",V1, type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECTDistinct * FROMLV1",LV1, type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECTDistinct * FROMV3",V3, type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECTDistinct * FROMLV3",LV3, type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECTDistinct * FROMV4",V4, type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECTDistinct * FROMLV5",LV5, type:Q\_CONST,Q\_CONST).$$
$$\#import(animalhabitat,"root","gj","SELECTDistinct * FROMV5",V5, type:Q\_CONST,Q\_CONST).$$
$$\#import(mysqltest,"root","gj","SELECTDistinct * FROMLV6",LV6, type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(mysqltest,"root","gj","SELECTDistinct * FROMLV7",LV7, type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(accesstest,"","","SELECTDistinct * FROMLV4",LV4, type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$Animal(Name,Class,Food):-V1(Name,Class,Food).$$
$$Animal(Name,Class,Food):-LV1(Name,Class,Food).$$
$$Animal(Name,Class,Food):-V3(Name,Class,Food).$$
$$Animal(Name,Class,Food):-LV3(Name,Class,Food).$$
$$Animal(Name,Class,Food):-V4(Name,Class,Food).$$
$$Animal(Name,Class,Food):-LV4(Name,Class,Food).$$
$$Animal(Name,Class,Food):-V5(Name,Food),chosen1(Name,Food,Class).$$
$$chosen1(Name,Food,Class):-V5(Name,Food),dom(Class), not\ diffchoice1(Name,Food,Class).$$

$$\begin{aligned}
\mathit{diffchoice}1(Name, Food, Class) :- & \mathit{chosen}1(Name, Food, U), \mathit{dom}(Class), \\
& U! = Class.
\end{aligned}$$

$$\begin{aligned}
Animal(Name, Class, Food) :- & LV5(Name, Food), \\
& \mathit{chosen}2(Name, Food, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}2(Name, Food, Class) :- & LV5(Name, Food), \mathit{dom}(Class), \\
& \mathit{not\ diffchoice}2(Name, Food, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}2(Name, Food, Class) :- & \mathit{chosen}2(Name, Food, U), \mathit{dom}(Class), \\
& U! = Class.
\end{aligned}$$

$$\begin{aligned}
Animal(Name, Class, Food) :- & V2(Name, Habitat), \\
& \mathit{chosen}3(Name, Habitat, Class), \\
& \mathit{chosen}4(Name, Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}3(Name, Habitat, Class) :- & V2(Name, Habitat), \mathit{dom}(Class), \\
& \mathit{not\ diffchoice}3(Name, Habitat, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}3(Name, Habitat, Class) :- & \mathit{chosen}3(Name, Habitat, U), \mathit{dom}(Class), \\
& U! = Class.
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}4(Name, Habitat, Food) :- & V2(Name, Habitat), \mathit{dom}(Food), \\
& \mathit{not\ diffchoice}4(Name, Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}4(Name, Habitat, Food) :- & \mathit{chosen}4(Name, Habitat, U), \mathit{dom}(Food), \\
& U! = Food.
\end{aligned}$$

$$Habitat(Name, Habitat) :- V2(Name, Habitat).$$

$$\begin{aligned}
Animal(Name, Class, Food) :- & LV2(Name, Habitat), \\
& \mathit{chosen}5(Name, Habitat, Class), \\
& \mathit{chosen}6(Name, Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}5(Name, Habitat, Class) :- & LV2(Name, Habitat), \mathit{dom}(Class), \\
& \mathit{not\ diffchoice}5(Name, Habitat, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}5(Name, Habitat, Class) :- & \mathit{chosen}5(Name, Habitat, U), \mathit{dom}(Class), \\
& U! = Class.
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}6(Name, Habitat, Food) :- & LV2(Name, Habitat), \mathit{dom}(Food), \\
& \mathit{not\ diffchoice}6(Name, Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}6(Name, Habitat, Food) :- & \mathit{chosen}6(Name, Habitat, U), \mathit{dom}(Food), \\
& U! = Food.
\end{aligned}$$

$$Habitat(Name, Habitat) :- LV2(Name, Habitat).$$

$$\begin{aligned}
Animal(Name, Class, Food) :- & V6(Habitat), \mathit{chosen}7(Habitat, Name), \\
& \mathit{chosen}8(Habitat, Class), \\
& \mathit{chosen}9(Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}7(Habitat, Name) :- & V6(Habitat), \mathit{dom}(Name), \\
& \mathit{not\ diffchoice}7(Habitat, Name).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}7(Habitat, Name) :- & \mathit{chosen}7(Habitat, U), \mathit{dom}(Name), \\
& U! = Name.
\end{aligned}$$

$$chosen8(Habitat, Class) : -V6(Habitat), dom(Class), \\ not\ diffchoice8(Habitat, Class).$$
$$diffchoice8(Habitat, Class) : -chosen8(Habitat, U), dom(Class), \\ U! = Class.$$
$$chosen9(Habitat, Food) : -V6(Habitat), dom(Food), \\ not\ diffchoice9(Habitat, Food).$$
$$diffchoice9(Habitat, Food) : -chosen9(Habitat, U), dom(Food), \\ U! = Food.$$
$$Habitat(Name, Habitat) : -V6(Habitat), chosen7(Habitat, Name).$$
$$Animal(Name, Class, Food) : -LV6(Name, Class, Food).$$
$$Animal(Name, Class, Food) : -LV7(Name, Class, Food).$$

Test Case A6: Disjunctive Query with Equality Built-ins

$$\begin{aligned}
Ans(Name, Food) &: - Animal(Name, Class, Food), Habitat(Name, Habitat), \\
&\quad Class == "amphibian", Food == "insect". \\
Ans(Name, Food) &: - Animal(Name, Class, Food), Class == "bird". \\
Ans(Name, Food) &: - Animal(Name, Class, Food), Class == "reptile". \\
Ans(Name, Food) &: - Animal(Name, Class, Food), Class == "mammal", \\
&\quad Food == "plant". \\
Ans(Name, Food) &: - Animal(Name, Class, Food), Habitat(Name, Habitat), \\
&\quad Habitat == "ocean", Food == "fish". \\
Ans(Name, Food) &: - Animal(Name, Class, Food), Habitat(Name, Habitat), \\
&\quad Habitat == "desert". \\
Ans(Name, Food) &: - Animal(Name, Class, Food), Habitat(Name, Habitat), \\
&\quad Habitat == "savannah". \\
Ans(Name, Food)?
\end{aligned}$$

Program using VISS:

$$\begin{aligned}
&\#import(animalkingdom, "sa", "gj", "SELECT\ Distinct\ *\ FROM\ V1 \\
&\qquad where\ Class =' amphibian'\ and\ Food =' insect'", \\
&\qquad V1, type : Q_CONST, Q_CONST, Q_CONST). \\
&\#import(animalkingdom, "sa", "gj", "SELECT\ Distinct\ *\ FROM\ LV1 \\
&\qquad where\ Class =' amphibian'\ and\ Food =' insect'", \\
&\qquad LV1, type : Q_CONST, Q_CONST, Q_CONST). \\
&\#import(mysqltest, "root", "gj", "SELECT\ Distinct\ *\ FROM\ LV6 \\
&\qquad where\ Class =' amphibian'\ and\ Food =' insect'", \\
&\qquad LV6, type : Q_CONST, Q_CONST, Q_CONST).
\end{aligned}$$

$$\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ V2",$$
$$V2,type:Q\_CONST,Q\_CONST).$$
$$\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ LV2",$$
$$LV2,type:Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V1$$
$$where\ Class ='\ bird'",$$
$$V1,type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV1$$
$$where\ Class ='\ bird'",$$
$$LV1,type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V4$$
$$where\ Class ='\ bird'",V4,$$
$$type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(accesstest,"","","SELECT\ Distinct\ *\ FROM\ LV4$$
$$where\ Class ='\ bird'",LV4,$$
$$type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V1$$
$$where\ Class ='\ reptile'",V1,$$
$$type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV1$$
$$where\ Class ='\ reptile'",LV1,$$
$$type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(mysqltest,"root","gj","SELECT\ Distinct\ *\ FROM\ LV7$$
$$where\ Class ='\ reptile'",LV7,$$
$$type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V1$$
$$where\ Class ='\ mammal'\ and\ Food ='\ plant'",$$
$$V1,type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV1$$
$$where\ Class ='\ mammal'\ and\ Food ='\ plant'",$$
$$LV1,type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V3$$
$$where\ Class ='\ mammal'\ and\ Food ='\ plant'",$$
$$V3,type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV3$$
$$where\ Class ='\ mammal'\ and\ Food ='\ plant'",$$
$$LV3,type:Q\_CONST,Q\_CONST,Q\_CONST).$$
$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV5$$
$$where\ Food ='\ plant'",LV5,$$
$$type:Q\_CONST,Q\_CONST).$$

$$
\begin{aligned}
&\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ V5\\
&\qquad\qquad\qquad\qquad where\ Food =' plant'",V5,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST).\\
&\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V1\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",V1,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV1\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",LV1,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V3\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",V3,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV3\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",LV3,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V4\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",V4,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV5\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",LV5,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST).\\
&\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ V2\\
&\qquad\qquad\qquad\qquad where\ Habitat =' ocean'",V2,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST).\\
&\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ LV2\\
&\qquad\qquad\qquad\qquad where\ Habitat =' ocean'",LV2,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST).\\
&\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ V5\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",V5,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST).\\
&\quad\#import(mysqltest,"root","gj","SELECT\ Distinct\ *\ FROM\ LV6\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",LV6,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&\quad\#import(mysqltest,"root","gj","SELECT\ Distinct\ *\ FROM\ LV7\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",LV7,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST, Q\_CONST).\\
&\qquad\#import(accesstest,"","","SELECT\ Distinct\ *\ FROM\ LV4\\
&\qquad\qquad\qquad\qquad where\ Food =' fish'",LV4,\\
&\qquad\qquad\qquad\qquad type : Q\_CONST, Q\_CONST, Q\_CONST).
\end{aligned}
$$

$$\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ V2 \\ where\ Habitat =' desert'",V2, \\ type : Q\_CONST, Q\_CONST).$$

$$\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ LV2 \\ where\ Habitat =' desert'",LV2, \\ type : Q\_CONST, Q\_CONST).$$

$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V1",V1, \\ type : Q\_CONST, Q\_CONST, Q\_CONST).$$

$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV1",LV1, \\ type : Q\_CONST, Q\_CONST, Q\_CONST).$$

$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V3",V3, \\ type : Q\_CONST, Q\_CONST, Q\_CONST).$$

$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV3",LV3, \\ type : Q\_CONST, Q\_CONST, Q\_CONST).$$

$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ V4",V4, \\ type : Q\_CONST, Q\_CONST, Q\_CONST).$$

$$\#import(animalkingdom,"sa","gj","SELECT\ Distinct\ *\ FROM\ LV5",LV5, \\ type : Q\_CONST, Q\_CONST).$$

$$\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ V5",V5, \\ type : Q\_CONST, Q\_CONST).$$

$$\#import(mysqltest,"root","gj","SELECT\ Distinct\ *\ FROM\ LV6",LV6, \\ type : Q\_CONST, Q\_CONST, Q\_CONST).$$

$$\#import(mysqltest,"root","gj","SELECT\ Distinct\ *\ FROM\ LV7",LV7, \\ type : Q\_CONST, Q\_CONST, Q\_CONST).$$

$$\#import(accesstest,"","","SELECT\ Distinct\ *\ FROM\ LV4",LV4, \\ type : Q\_CONST, Q\_CONST, Q\_CONST).$$

$$\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ V2 \\ where\ Habitat =' savannah'",V2, \\ type : Q\_CONST, Q\_CONST).$$

$$\#import(animalhabitat,"root","gj","SELECT\ Distinct\ *\ FROM\ LV2 \\ where\ Habitat =' savannah'",LV2, \\ type : Q\_CONST, Q\_CONST).$$

$$Animal(Name, Class, Food) : -V1(Name, Class, Food).$$

$$Animal(Name, Class, Food) : -LV1(Name, Class, Food).$$

$$Animal(Name, Class, Food) : -V3(Name, Class, Food).$$

$$Animal(Name, Class, Food) : -LV3(Name, Class, Food).$$

$$Animal(Name, Class, Food) : -V4(Name, Class, Food).$$

$$Animal(Name, Class, Food) : -LV4(Name, Class, Food).$$

$$Animal(Name, Class, Food) : -V5(Name, Food), \\ chosen1(Name, Food, Class).$$

$$\begin{aligned}
\mathit{chosen}1(Name, Food, Class) :- & V5(Name, Food), \mathit{dom}(Class), \\
& \mathit{not}\ \mathit{diffchoice}1(Name, Food, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}1(Name, Food, Class) :- & \mathit{chosen}1(Name, Food, U), \mathit{dom}(Class), \\
& U! = Class.
\end{aligned}$$

$$\begin{aligned}
\mathit{Animal}(Name, Class, Food) :- & LV5(Name, Food), \\
& \mathit{chosen}2(Name, Food, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}2(Name, Food, Class) :- & LV5(Name, Food), \mathit{dom}(Class), \\
& \mathit{not}\ \mathit{diffchoice}2(Name, Food, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}2(Name, Food, Class) :- & \mathit{chosen}2(Name, Food, U), \mathit{dom}(Class), \\
& U! = Class.
\end{aligned}$$

$$\begin{aligned}
\mathit{Animal}(Name, Class, Food) :- & V2(Name, Habitat), \\
& \mathit{chosen}3(Name, Habitat, Class), \\
& \mathit{chosen}4(Name, Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}3(Name, Habitat, Class) :- & V2(Name, Habitat), \mathit{dom}(Class), \\
& \mathit{not}\ \mathit{diffchoice}3(Name, Habitat, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}3(Name, Habitat, Class) :- & \mathit{chosen}3(Name, Habitat, U), \mathit{dom}(Class), \\
& U! = Class.
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}4(Name, Habitat, Food) :- & V2(Name, Habitat), \mathit{dom}(Food), \\
& \mathit{not}\ \mathit{diffchoice}4(Name, Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}4(Name, Habitat, Food) :- & \mathit{chosen}4(Name, Habitat, U), \mathit{dom}(Food), \\
& U! = Food.
\end{aligned}$$

$$Habitat(Name, Habitat) :- V2(Name, Habitat).$$

$$\begin{aligned}
\mathit{Animal}(Name, Class, Food) :- & LV2(Name, Habitat), \\
& \mathit{chosen}5(Name, Habitat, Class), \\
& \mathit{chosen}6(Name, Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}5(Name, Habitat, Class) :- & LV2(Name, Habitat), \mathit{dom}(Class), \\
& \mathit{not}\ \mathit{diffchoice}5(Name, Habitat, Class).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}5(Name, Habitat, Class) :- & \mathit{chosen}5(Name, Habitat, U), \mathit{dom}(Class), \\
& U! = Class.
\end{aligned}$$

$$\begin{aligned}
\mathit{chosen}6(Name, Habitat, Food) :- & LV2(Name, Habitat), \mathit{dom}(Food), \\
& \mathit{not}\ \mathit{diffchoice}6(Name, Habitat, Food).
\end{aligned}$$

$$\begin{aligned}
\mathit{diffchoice}6(Name, Habitat, Food) :- & \mathit{chosen}6(Name, Habitat, U), \mathit{dom}(Food), \\
& U! = Food.
\end{aligned}$$

$$Habitat(Name, Habitat) :- LV2(Name, Habitat).$$

$$\mathit{Animal}(Name, Class, Food) :- LV6(Name, Class, Food).$$

$$\mathit{Animal}(Name, Class, Food) :- LV7(Name, Class, Food).$$