Project Number: IST-2001-33570
Project Acronym: INFOMIX
Project Full Name: Infomix: Boosting the Information Integration

Deliverable Number: D3.2
Title: **Query language**
Authors: Nicola Leone, Giovambattista Ianni
Workpackage: WP3
Document Type: Deliverable
Distribution: Infomix Consortium
Status: Final
Document file: WP3_T2_D3.2_0103.ps
Version: 1.3
Number of pages: 22

Due Date: September 30, 2003
Delivery Date: November 11, 2003

Partners contributed: UNICAL, TUWIEN, UNIROME1
Partners owning: UNICAL, TUWIEN, UNIROME1
Man Months: 9

Short Description:

This report provides the formal definition of the query language of the INFOMIX system.
The query language of the INFOMIX system will be formally defined. The language is to be fully declarative, highly expressive, and able to deal with semi-structured data to some extent.

Document Change Record		
Version	**Date**	**Reason for Change**
v.1.3	November 10, 2003	Revised Final version

Contents

1 Introduction **4**
1.1 Requirements for the Infomix Query Language 4

2 Theoretical Background **6**
2.1 The relational Model 6

3 Formal Framework for Information Integration **8**
3.1 Syntax 9
3.2 Semantics 10

4 Query Languages and Interaction with ICs **11**
4.1 Non-recursive Datalog Programs 11
4.2 Recursion 12
4.3 Negation 13
4.4 Aggregate Functions 14

5 IQL grammar **15**

6 IQL Semantics **18**

7 Conclusions **21**

1 Introduction

The formal framework underlying the information integration model (IIM) of the INFOMIX system has been completely specified in the deliverable D3.1, which is mainly devoted to the description of the languages for the mapping specification and of the types of integrity constraints the designer is allowed to specify in the global schema. We briefly recall that this framework accounts for data integration systems consisting of a global schema $\mathcal{G}$, which specifies the global (user) elements, a source schema $\mathcal{S}$, which describes the structure of the data sources in the system, and a mapping $\mathcal{M}$, which specifies the relationship between the sources and the global schema.

In the framework, $\mathcal{M}$ can be expressed in both the two approaches for specifying the mapping commonly adopted in the theory and practice of data integration [9]: the global-as-view (GAV) approach, in which global elements are defined as views over the sources, and the local-as-view (LAV), in which, conversely, source elements are characterized as views over the global schema.

Up to this point, less attention has been paid to the formal description of the user query language, which at large extent has been assumed to be the select-project-join fragment of standard SQL, whose power is equivalent to the one of non-recursive Datalog programs. The main reason for postponing the choice of the user query languages and for temporarily dealing with this language can be understood by looking at the result provided in the deliverables D3.1 and D3.3, where it is put in evidence that the interaction between the classes of constraints allowed on $\mathcal{G}$ makes query answering difficult under the different semantics considered, and may lead to undecidable cases even when the user query language is non-recursive Datalog. The focus of these results is to highlight the intrinsic difficulty of query answering in data integration systems, due to the results of hardness and undecidability holding for not highly expressive query languages.

1.1 Requirements for the Infomix Query Language

In this report we are interested in finding some space for enriching this 'basic' query language with some additional features, still guided by the fact that, in the spirit of the INFOMIX approach, the resulting language must be fully declarative and the DLV system must be able to evaluate it efficiently and preserve scalability as much as possible.

Specifically, we consider the use of (stratified) negation, recursion and aggregate functions for enriching non-recursive Datalog programs, and we investigate the interactions (possible increase of computational complexity or cause of undecidability) of such features with the different kinds of constraints allowed on the global schema. In this analysis, we deal with all the kinds of integrity constraints (ICs in the following) the user is allowed to specify over the global schema in the INFOMIX system:

- *Key Dependencies* (KDs), i.e., constraints expressing that each tuple in any relation can be fully identified by the values of certain attributes.
- *Inclusion Dependencies* (IDs), i.e. constraints expressing that (a projection of) a given relation is included in (a projection of) another relation, and

ICs	UCQs	recursion	stratified negation	aggregates	Datalog$^{\neg s}$ with aggregates
KDs	+	+	+	+	+
IDs	+/-	-	-	-	-
EDs	+	+	+	+	+
EDs and KDs	+	+	+	+	+

Figure 1: Interactions of ICs with the features of the query language.

- *Exclusion Dependencies* (EDs), i.e. constraints expressing that two relations (or projections on them) are disjoint.

We point out that the type of mapping over the sources does not matter for the analysis of the query language as well as the view definition language of the sources.

In Figure 1, we summarize the results described in this report. We put in evidence the categories of ICs (+) allowed in order to maintain the decidability of the query answering as the expressiveness of the query language is changed. The case of the union of conjunctive queries (UCQs) is of particular interest, as it is fully decidable in the case in which IDs are *Non-Key Conflicting Inclusion Dependencies* (NKCIDs), i.e. IDs which do not propagate a proper superset of the key of the relation into its right-hand side. For instance, this class comprises the well known class of foreign-key dependencies.

Conversely, recursion, negation and aggregates can be only used in the absence of inclusion dependencies, no matter of particular restrictions, and can be combined without any decidability problem, leading to the language Datalog$^{\neg s}$ with aggregates.

Although UCQs is the only language (w.r.t. the extensions considered) that is fully decidable in the INFOMIX setting, we decide to not restrict the attention to UCQs and we assume Datalog$^{\neg s}$ with aggregates to be the user query language (from now on we will call this language IQL, Infomix Query Language). This language is indeed powerful enough for expressing significant, real-world queries and it is still decidable in the presence of common ICs such as keys and exclusion dependencies.

Obviously, each time the user wants to specify also IDs, the module which is responsible for the query preprocessing will automatically restrict the language to UCQs, by disallowing the use of the other complex features, and possibly informing the user about the adopted restrictions.

The rest of the present document is structured as follows. In the next section, we briefly recall some preliminaries on the relational model and, specifically, on the syntax and the semantics of the integrity constraints. Section 3 presents an overview of the Information Integration setting of the INFOMIX system, whereas in Section 4 we analyze the basic query language for the INFOMIX system. In Section 5, we formally present the grammar of the IQL in EBNF notation and the semantics, and, finally, in Section 7 we draw our conclusions.

2 Theoretical Background

In this section we recall some theoretical background that will be useful for the following discussions. In particular, we present the basic notions of the relational model, on which we will build in the next section our formal framework for data integration, and briefly recall the complexity classes that will be mentioned in this report. This section is intended to be a brief introduction to such matters; for further background we refer the reader to [7, 1, 12].

2.1 The relational Model

We assume to deal with an infinite, fixed database domain $\mathcal{U}$ whose elements can be referenced by constants $c_1,\ldots, c_n$ under the *unique name assumption*, i.e., different constants denote different real-word objects.

A *relational schema* (or simply *schema*) $\mathcal{RS}$ is a pair $\langle \Psi, \Sigma \rangle$, where:

- Ψ is a set of relations, each with an associated arity that indicates the number of its attributes. The attributes of a relation $r \in \Psi$ of arity n (denoted as r/n) are represented by the integers $1, \ldots, n$;
- Σ is a set of *integrity constraints* expressed on the relations in Ψ, i.e., assertions on the relations in Ψ that are intended to be satisfied by database instances.

A *database instance* (or simply *database*) $\mathcal{DB}$ for a schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ is a set of facts of the form $r(t)$ where r is a relation of arity n in Ψ and t is an n-tuple of constants of $\mathcal{U}$. We denote as $r^{\mathcal{DB}}$ the set $\{t \mid r(t) \in \mathcal{DB}\}$. A database $\mathcal{DB}$ for a schema $\mathcal{RS}$ is said to be *consistent* with $\mathcal{RS}$ if it satisfies all constraints expressed on $\mathcal{RS}$. The notion of satisfaction depends on the type of constraints defined over the schema.

The integrity constraints that we consider are *inclusion dependencies (IDs)*, *key dependencies (KDs)* and *exclusion dependencies (EDs)*. More specifically,

- an *inclusion dependency* is an assertion of the form $r_1[\mathbf{A}] \subseteq r_2[\mathbf{B}]$, where r_1, r_2 are relations in Ψ, $\mathbf{A} = A_1, \ldots, A_n$ $(n \geq 0)$ is a sequence of attributes of r_1, and $\mathbf{B} = B_1, \ldots, B_n$ is a sequence of distinct attributes of r_2. Therefore, we allow for repetition of attributes in the left-hand side of the inclusion[1]. A database $\mathcal{DB}$ for $\mathcal{RS}$ satisfies an inclusion dependency $r_1[\mathbf{A}] \subseteq r_2[\mathbf{B}]$ if for each tuple $t_1 \in r_1^{\mathcal{DB}}$ there exists a tuple $t_2 \in r_2^{\mathcal{DB}}$ such that $t_1[\mathbf{A}] = t_2[\mathbf{B}]$, where $t[\mathbf{A}]$ indicates the projection of the tuple t over $\mathbf{A}$;
- a *key dependency* is an assertion the form $key(r) = \mathbf{A}$, where r is a relation in Ψ, and $\mathbf{A} = A_1, \ldots, A_n$ is a sequence of distinct attributes of r. A database $\mathcal{DB}$ for $\mathcal{RS}$ satisfies a key dependency $key(r) = \mathbf{A}$ if for each $t_1, t_2 \in r^{\mathcal{DB}}$ with $t_1 \neq t_2$ we have $t_1[\mathbf{A}] \neq t_2[\mathbf{A}]$. We assume that at most one key dependency is specified for each relation;

[1] Repetitions in the right-hand side force equalities between attributes of the relation in left-hand side, hence they imply constraints that we do not consider in this report.

- an *exclusion dependency* is an assertion of the form $(r_1[\mathbf{A}] \cap r_2[\mathbf{B}]) = \emptyset$, where r_1 and r_2 are relations in Ψ, $\mathbf{A} = A_1, \ldots, A_n$ and $\mathbf{B} = B_1, \ldots, B_n$ are sequences of attributes of r_1 and r_2, respectively. A database $\mathcal{DB}$ for $\mathcal{RS}$ satisfies an exclusion dependency $(r_1[\mathbf{A}] \cap r_2[\mathbf{B}]) = \emptyset$ if there do not exist two tuples $t_1 \in r_1^{\mathcal{DB}}$ and $t_2 \in r_2^{\mathcal{DB}}$ such that $t_1[\mathbf{A}] = t_2[\mathbf{B}]$.

Sets of inclusion, key and exclusion dependencies expressed on the database schema are denoted by Σ_I, Σ_K, and Σ_E, respectively. Furthermore, we use $\mathcal{RS} = \langle \Psi, \Sigma_I, \Sigma_K, \Sigma_E \rangle$ as a shortcut for $\mathcal{RS} = \langle \Psi, \Sigma_I \cup \Sigma_K \cup \Sigma_E \rangle$. In the absence of some kind of dependencies, we disregard the corresponding symbol. When a database $\mathcal{DB}$ satisfies all dependencies in Σ (resp. Σ_I, Σ_K, or Σ_E), we say that $\mathcal{DB}$ is consistent with Σ (resp. Σ_I, Σ_K, or Σ_E).

A *relational query* (or simply *query*) over $\mathcal{RS}$ is a formula that specifies a set of data to be retrieved from a database. In this document, we consider the classes of conjunctive queries, union of conjunctive queries and DATALOG queries. A *conjunctive query* (CQ) q of arity n over the schema $\mathcal{RS}$ is written in the form

$$q(\vec{\mathbf{x}}) \leftarrow conj(\vec{\mathbf{x}}, \vec{\mathbf{y}})$$

where

- q belongs to a new set of symbols $\mathcal{Q}$ (the alphabet of queries, that is disjoint from both $\mathcal{U}$ and Ψ);
- $q(\vec{\mathbf{x}})$ is the *head* of the conjunctive query;
- $conj(\vec{\mathbf{x}}, \vec{\mathbf{y}})$ is the *body* of the conjunctive query, which is a conjunction of atoms that involve $\vec{\mathbf{x}} = X_1, \ldots, X_n$ and $\vec{\mathbf{y}} = Y_1, \ldots, Y_m$, where all X_i and Y_j are either variables or constants of $\mathcal{U}$;
- the predicate symbols of the atoms in $conj(\vec{\mathbf{x}}, \vec{\mathbf{y}})$ are in Ψ;
- the number of variables of $\vec{\mathbf{x}}$ is called the *arity* of q, and is the arity of the relation denoted by the query q.

Given a database $\mathcal{DB}$, the answer to q over $\mathcal{DB}$, denoted $q^{\mathcal{DB}}$, is the set of n-tuples of constants $\langle c_1, \ldots, c_n \rangle$, such that, when substituting each c_i for X_i, the formula

$$\exists \vec{\mathbf{y}}.conj(\vec{\mathbf{x}}, \vec{\mathbf{y}})$$

evaluates to true in $\mathcal{DB}$.

A set of conjunctive queries with the same head predicate is a *Union of Conjunctive Queries* (UCQ). More formally, a UCQ is written in the form

$$q(\vec{\mathbf{x}}) \leftarrow conj_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}_1) \vee \cdots \vee conj_m(\vec{\mathbf{x}}, \vec{\mathbf{y}}_m)$$

The answer to a UCQ q over a database $\mathcal{DB}$, as usually denoted $q^{\mathcal{DB}}$, is the set of n-tuples of constants $\langle c_1, \ldots, c_n \rangle$, such that, when substituting each c_i for X_i, the formula

$$\exists \vec{\mathbf{y}}_1.conj_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}_1) \vee \cdots \vee \exists \vec{\mathbf{y}}_m.conj_m(\vec{\mathbf{x}}, \vec{\mathbf{y}}_m)$$

evaluates to true in $\mathcal{DB}$.

Finally a DATALOG query is a collection of rules, each having the same form as a conjunctive query, except that predicate symbols in the body of the rules can be in $\mathcal{Q}$ as well. In a DATALOG query, each head predicate of the rules refers to an intermediate relation, disjoint from predicates referring to database relations. The intermediate predicates are called *Intensional DataBase* (IDB) predicates, whereas predicates referring to stored relations are called *Extensional DataBase* (EDB) predicates. Given a DATALOG query q and a database $\mathcal{DB}$, the answer $q^{\mathcal{DB}}$ of q over $\mathcal{DB}$ is the minimal fixpoint model of q and $\mathcal{DB}$ [2].

Given a CQ, UCQ, or DATALOG query q, we also say that $q^{\mathcal{DB}}$ denotes the set of tuples that *satisfy* q over $\mathcal{DB}$.

3 Formal Framework for Information Integration

Informally, a data integration system consists of a (virtual) global schema, which specifies the global (user) elements, a source schema, which describes the structure of the sources in the system, and a mapping, which specifies the relationship between the sources and the global schema. User queries are posed on the global schema, and the system provides the answers to such queries by exploiting the information supplied by the mapping and accessing the sources that contain relevant data. Thus, from the syntactic viewpoint, the specification of an integration system depends on the following parameters:

- The form of the global schema, i.e., the formalism used for expressing global elements and relationships between global elements. Several settings have been considered in the literature, where, for instance, the global schema can be relational, object-oriented [4], semi-structured [11], based on Description Logics [8, 5], etc.;
- The form of the source schema, i.e., the formalism used for expressing data at the sources and relationships between such data. In principle, formalisms commonly adopted for the source schema are the same mentioned for the global schema;
- The form of the mapping. Two basic approach have been proposed in the literature, called respectively *global-as-view* (GAV) and *local-as-view* (LAV) [10, 9]. The GAV approach requires that the global schema is defined in terms of the data sources: more precisely, every element of the global schema is associated with a view, i.e., a query, over the sources, so that its meaning is specified in terms of the data residing at the sources. Conversely, in the LAV approach, the meaning of the sources is specified in terms of the elements of the global schema: more exactly, the mapping between the sources and the global schema is provided in terms of a set of views over the global schema, one for each source element;
- The language of the mapping, i.e., the query language used to express views in the mapping;
- The language of the user queries, i.e., the query language adopted by users to issue queries on the global schema.

Let us now turn our attention on the semantics. According to [9], the semantics of a data integration system is given in terms of instances of the elements of the global schema (e.g., one set of tuples for each global relation if the global schema is relational, one set of objects for each global class if it is object-oriented, etc.). Such instances have to satisfy *(i)* the integrity constraints expressed between elements of the global schema, and *(ii)* the mapping specified between the global and the source schema.

Roughly speaking, the notion of satisfying the mapping depends on how the data that can be retrieved from the sources are interpreted with respect to the data that satisfy the corresponding portion of the global schema. Three different assumptions have been considered for such interpretation: the assumption of *sound* mapping, adopted when all data that can be retrieved at the sources satisfy the corresponding portion of the global schema but may result incomplete; the assumption of *complete* mapping, adopted when no data other than those retrieved at the sources satisfy the corresponding portion of global schema, i.e., they are complete but not all sound; the assumption of *exact* mapping, when data are both sound and complete.

In the following we formally define the data integration framework underlying the INFOMIX IIM. We consider here a relational setting, i.e., the global and the source schema are expressed in the relational model. However, our framework can be in principle generalized to different data models.

3.1 Syntax

A data integration system $\mathcal{I}$ is a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where:

- $\mathcal{G}$ is the *global schema* expressed in the relational model with inclusion, key and exclusion dependencies, i.e., $\mathcal{G} = \langle \Psi, \Sigma \rangle$, where $\Sigma = \Sigma_I \cup \Sigma_K \cup \Sigma_E$;
- $\mathcal{S}$ is the *source schema* expressed in the relational model without integrity constraints, i.e., $\mathcal{S} = \langle \Psi_{\mathcal{S}}, \emptyset \rangle$. Dealing with only relational sources is not restrictive, since we can always assume that suitable wrappers present sources in the relational format. Furthermore, assuming that no integrity constraint is specified on $\mathcal{S}$ is equivalent to assuming that data satisfy constraints expressed on the sources in which they are stored. This is a common assumption in data integration, because sources are in general autonomous and external to the integration system, and satisfaction of constraints at the sources should be guaranteed by local data management systems;
- $\mathcal{M}$ is the *mapping* between $\mathcal{G}$ and $\mathcal{S}$. In our framework we consider both the GAV and the LAV mapping. More precisely,
  - the GAV mapping is a set of assertions of the form $\langle r_{\mathcal{G}}, q_{\mathcal{S}} \rangle$, where $r_{\mathcal{G}}$ is a global relation and $q_{\mathcal{S}}$ is the associated query over the source schema $\mathcal{S}$. In this document, we study the setting in which the language used to express queries in the GAV mapping is non-recursive Datalog;
  - the LAV mapping is a set of assertions of the form $\langle r_{\mathcal{S}}, q_{\mathcal{G}} \rangle$, where $r_{\mathcal{S}}$ is a source relation and $q_{\mathcal{G}}$ is the associated query over the global schema $\mathcal{G}$. We assume that queries in the LAV mapping are Conjunctive Queries.

Finally, a *query over* $\mathcal{I}$ (also called *user query* in the following) is a formula that specifies which data to extract from the integration system. Each user query is issued over the global schema $\mathcal{G}$, and we assume that the language used to specify user queries is Union of Conjunctive Queries.

3.2 Semantics

Intuitively, to define the semantics of a data integration system, we have to start with a set of data at the sources, and we have to specify which are the data that satisfy the global schema with respect to such data at the sources, according to the assumption adopted for the mapping. Thus, in order to assign the semantics to a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, we start by considering a *source database for* $\mathcal{I}$, i.e., a database $\mathcal{D}$ for the source schema $\mathcal{S}$. Then, we consider the assumption on $\mathcal{M}$, and denote it with $as(\mathcal{M})$, and pose $as(\mathcal{M}) = s$, c, or e, for the sound, complete, and exact assumption, respectively.

Based on $\mathcal{D}$ and $as(\mathcal{M})$, we now specify which is the information content of the global schema $\mathcal{G}$. We call any database $\mathcal{B}$ for $\mathcal{G}$ a *global database* for $\mathcal{I}$. Formally, the semantics of $\mathcal{I}$ w.r.t. $\mathcal{D}$ and $as(\mathcal{M})$, is the set of global databases $\mathcal{B}$ for $\mathcal{I}$ such that:

(i) $\mathcal{B}$ is consistent with $\mathcal{G}$;

(ii) $\mathcal{B}$ satisfies $as(\mathcal{M})$ with respect to $\mathcal{D}$, i.e.,

- if $\mathcal{M}$ is GAV, $\mathcal{B}$ satisfies $as(\mathcal{M})$ if for each assertion $\langle r_{\mathcal{G}}, q_{\mathcal{S}} \rangle \in \mathcal{M}$ we have that
  - (a) $r_{\mathcal{G}}^{\mathcal{B}} \supseteq q_{\mathcal{S}}^{\mathcal{D}}$ if $as(\mathcal{M}) = s$;
  - (b) $r_{\mathcal{G}}^{\mathcal{B}} \subseteq q_{\mathcal{S}}^{\mathcal{D}}$ if $as(\mathcal{M}) = c$;
  - (c) $r_{\mathcal{G}}^{\mathcal{B}} = q_{\mathcal{S}}^{\mathcal{D}}$ if $as(\mathcal{M}) = e$;
- if $\mathcal{M}$ is LAV, $\mathcal{B}$ satisfies $as(\mathcal{M})$ if for each assertion $\langle r_{\mathcal{S}}, q_{\mathcal{G}} \rangle \in \mathcal{M}$ we have that
  - (a) $r_{\mathcal{S}}^{\mathcal{D}} \subseteq q_{\mathcal{G}}^{\mathcal{B}}$ if $as(\mathcal{M}) = s$;
  - (b) $r_{\mathcal{S}}^{\mathcal{D}} \supseteq q_{\mathcal{G}}^{\mathcal{B}}$ if $as(\mathcal{M}) = c$;
  - (c) $r_{\mathcal{S}}^{\mathcal{D}} = q_{\mathcal{G}}^{\mathcal{B}}$ if $as(\mathcal{M}) = e$;

Intuitively, in GAV (and in LAV) the three different assumptions allow us to model different situation in which queries over $\mathcal{S}$ (resp. relations in $\mathcal{S}$) provide (a) any subset of tuples that satisfy the corresponding relation in $\mathcal{G}$ (resp. query over $\mathcal{G}$), (b) any superset of such tuples, or (c) exactly such tuples.

The semantics of $\mathcal{I}$ w.r.t. $\mathcal{D}$ and $as(\mathcal{M})$, is denoted with $sem_{as(\mathcal{M})}(\mathcal{I}, \mathcal{D})$, where $as(\mathcal{M}) = s, c, e$ respectively for the sound, complete, or exact assumption. Obviously, $sem_{as(\mathcal{M})}(\mathcal{I}, \mathcal{D})$ contains in general several global databases for $\mathcal{I}$.

Finally, we give the semantics of queries. Formally, given a source database $\mathcal{D}$ for $\mathcal{I}$ and an assumption $as(\mathcal{M})$ on $\mathcal{M}$, we call *certain answers* (or simply *answers*) to a query q of arity n with respect to $\mathcal{I}$, $\mathcal{D}$ and $as(\mathcal{M})$, the set

$$ans_{as(\mathcal{M})}(q, \mathcal{I}, \mathcal{D}) = \{ \langle c_1, \ldots, c_n \rangle \mid \langle c_1, \ldots, c_n \rangle \in q^{\mathcal{B}} \text{ for each } \mathcal{B} \in sem_{as(\mathcal{M})}(\mathcal{I}, \mathcal{D}) \}$$

The problem of *query answering* is the problem of computing the set $ans_{as(\mathcal{M})}(q, \mathcal{I}, \mathcal{D})$. It should be easy to see that query answering in data integration systems is essentially a form of reasoning in the presence of incomplete information [14].

We will assume, in the following, to work with the sound mapping assumption. As better explained in deliverables D1.2 and D3.1, this semantics is the most significant in our context. Anyway, the Infomix Query Language well fits the loosely-sound and loosely-exact semantics, since restrictions we are going to introduce are the same in these latter cases.

4 Query Languages and Interaction with ICs

In this section we analyze the basic query language for the INFOMIX system, i.e. Datalog with stratified negation and aggregates.

Throughout the description, in order to provide some explicative examples we shall refer to a global schema $\mathcal{G} = \langle \Psi, \Sigma_I \rangle$, where Ψ contains the following relations:

$$employee(Ename, Salary, Dep, Boss)$$
$$department(Code, Director)$$

storing information about the employees of the departments of a given company. Specifically, each employee has associated a *Boss* which is an employee in its turn (often with a much higher salary).

4.1 Non-recursive Datalog Programs

Syntax: Following Prolog's convention, strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants. A *term* is either a variable or a constant. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A Datalog *rule* r is a formula

$$a \ :\!-\ b_1, \cdots, b_k.$$

where $a, b_1, \cdots, b_k$ are atoms and $k \geq 0$. The atom a will be called the *head* of r, while the conjunction $b_1, ..., b_k$ is the *body* of r. Let $H(r)$ and $B(r)$ be sets denoting the predicates in the head and in the body of r, respectively.

If $B(r) = \emptyset$, the rule is called a fact, and we usually omit the " :- " sign.

A datalog program $\mathcal{P}$ is a finite set of datalog rules. Given a Datalog program $\mathcal{P}$, the dependency graph $\mathcal{G}(\mathcal{P})$ is a directed graph whose nodes are the predicate symbols in $\mathcal{P}$ and such that there exists an edge from the predicate b to the predicate a if and only if $\mathcal{P}$ contains a rule r where $a \in H(r)$ and $b \in B(r)$.

An interesting syntactic restriction on Datalog programs is in disallowing recursion. This can be formalized by constraining the dependency graph to be acyclic.

Example 4.1 The program

$$a \ :\!-\ b,\ c.$$
$$b \ :\!-\ d.$$
$$d \ :\!-\ a.$$

is recursive, since in the corresponding dependency graph we have a cycle constituted by the arcs (a, b), (b, d) and (d, a). □

Semantics: The *Herbrand Universe* $U_{\mathcal{P}}$ of a program $\mathcal{P}$ is the set of all constants appearing in $\mathcal{P}$, and its *Herbrand Base* $B_{\mathcal{P}}$ is the set of all ground atoms constructed from the predicates appearing in $\mathcal{P}$ and the constants from $U_{\mathcal{P}}$. A term (resp. an atom, a rule or a program) is *ground* if no variables occur in it. A rule r' is a *ground instance* of a rule r, if r' is obtained from r by replacing every variable in r with some constant in $U_{\mathcal{P}}$. We denote by $ground(\mathcal{P})$ the set of all ground instances of the rules in $\mathcal{P}$.

An interpretation of $\mathcal{P}$ is any subset of $B_{\mathcal{P}}$. The value of a ground atom L w.r.t. an interpretation I, $value_I(L)$, is $true$ if $L \in I$ and $false$ otherwise. The truth value of a conjunction of ground literals $C = L_1, \ldots, L_n$ is the minimum over the values of the L_i, i.e. $value_I(C) = min(\{value_I(L_i) \mid 1 \leq i \leq n\})$; if $n = 0$, then $value_I(C) = true$. A ground rule r is *satisfied* by I if $value_I(Head(r)) \geq value_I(Body(r))$. Thus, a rule r with empty body is satisfied by I if $value_I(Head(r)) = true$. An interpretation M for $\mathcal{P}$ is a model of $\mathcal{P}$ if M satisfies all rules in $ground(\mathcal{P})$.

Non-recursive Datalog programs are very popular as query languages because they can be specified as the union of conjunctive queries, where each conjunctive query is expressible in the select-project-join fragment of SQL.

Query: Without loss of generality, we reformulate the notion of DATALOG query as a mapping from a database to a given goal relation. A query is a pair $\langle G, \mathcal{P} \rangle$ where G is a predicate symbol, called *goal*, and $\mathcal{P}$ is a datalog program. The query $\mathcal{Q}$ is non-recursive iff $\mathcal{P}$ is non-recursive. Given a database D, a tuple t of a relation r, can be seen as a ground fact of the form $r(t)$. Thus, given D, we we denote by $\mathcal{L}(D)$ the set of ground facts derived from each tuple in D.

The result of a query $\mathcal{Q} = \langle G, \mathcal{P} \rangle$ on an input database D is defined in terms of the models of $\mathcal{P}_D = \mathcal{P} \cup \mathcal{L}(D)$. Specifically, the result, denoted by $\mathcal{Q}(D)$, is the set $\{t \mid \exists M \in \mathcal{P}_D \wedge G(t) \in M\}$. For instance the query $\langle q_0, \mathcal{P}_0 \rangle$, where $\mathcal{P}_0$ is as follows

$$q_0(Ename) \text{ :- } employee(Ename, 100.000, Dep, Boss), department(Dep, leone).$$

returns all the employees working at the department whose chief is *leone* having a salary of *100.000* euros for year.

This fragment of SQL has been widely investigated in the past reports of the INFOMIX project. Specifically, in the deliverable D3.1 it has been evidenced that it is a fully decidable query language, if the set of ICs the user is allowed to specify are: KDs, EDs, and NKCIDs.

4.2 Recursion

It is well known that there are certain (polynomial computable) queries that cannot be expressed in non-recursive Datalog programs [1].

Example 4.2 Consider a situation in which we need to know whether the employee e_1 is the boss of the employee e either directly or even by means of a number of employee $e_2, .., e_n$ such that e_1 is the boss of e_2, e_2 is the boss of e_3,..., e_n is the boss of e.

Thus, by this query we are interested in knowing whether the level in the company of an employee is greater than that of another. Intuitively, as this query is based on a recursive definition, it cannot be expressed in non-recursive Datalog. Conversely, by allowing recursion we obtain the query $\langle q, \mathcal{P}_1 \rangle$, where $\mathcal{P}_1$ is

$$\begin{array}{l} q(E_1, E_2) \ \text{:-} \ employee(E_1, Salary, Dep, E_2). \\ q(E_1, E_2) \ \text{:-} \ q(E_1, E_3), \ employee(E_3, Salary, Dep, E_2). \end{array}$$

and we can simply check whether the tuple (e_1, e) is in the result. □

Unfortunately, recursive programs cannot be treated like UCQs. In fact, the recent work of Calvanese and Rosati [6] proves that, in the presence of either inclusion dependencies or key and foreign key dependencies, answers to recursive queries over finite databases are in general different from the answers obtained over unrestricted databases. More important, it proves that, both for unrestricted and for finite databases, recursive query answering is undecidable in the presence of inclusion dependencies or in the presence of key and foreign key dependencies.

The above results imply that, under the sound semantics, the presence of even very simple forms of integrity constraints makes the problem of answering recursive queries (and the problem of deciding query containment of a conjunctive query w.r.t. a recursive query) undecidable.

4.3 Negation

An interesting extension of Datalog is the introduction of negation. A literal l is either an atom a or its negation not a. Specifically, a *generalized rule* r is a formula

$$a \ \text{:-} \ b_1, \cdots, b_k, \ \text{not} \ b_{k+1}, \cdots, \ \text{not} \ b_m.$$

where a is an atom and $b_1, \cdots, b_m$ are literals and $m \geq k \geq 0$. Moreover, $B(r) = B^+(r) \cup B^-(r)$ denotes the set of the body literals, where $B^+(r)$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$.

Datalog with negation (Datalog$^\neg$) is strictly more expressive than Datalog. Unfortunately, an undisciplined use of negation may lead to programs whose meaning is difficult to be understood from the end user. We believe that free negation is not well suited for being part of a query language devoted to a broad range of users. Thus, in the following we provide some restrictions on the use of negation, which lead to more expressive languages w.r.t. recursive programs, still close to SQL standards.

We first provide some preliminary definitions.

Definition 4.1 Functions $|| \ || : B_\mathcal{P} \rightarrow \{0, 1, \ldots\}$ from the ground (classical) literals of the Herbrand Literal Base $B_\mathcal{P}$ to finite ordinals are called *level mappings* of $\mathcal{P}$. □

Level mappings give us a useful technique for describing various classes of programs.

Definition 4.2 A Datalog program $\mathcal{P}$ is called *(locally) stratified* ([3, 13]), if there is a level mapping $|| \ ||_s$ of $\mathcal{P}$ such that, for every rule $r \in ground(\mathcal{P})$,

1. For any $l \in B^+(r)$, and for any $l' \in H(r)$, $||l||_s \leq ||l'||_s$;
2. For any $l \in B^-(r)$, and for any $l' \in H(r)$, $||l||_s < ||l'||_s$. □

Example 4.3 Consider the following two programs.

$$\begin{array}{lll} \mathcal{P}_1: & p(a) \text{ :- not } q(a). & \\ & p(b) \text{ :- not } q(b). & \end{array} \qquad \begin{array}{lll} \mathcal{P}_2: & p(a) \text{ :- not } q(b). & \\ & q(b) \text{ :- not } p(a). & \end{array}$$

It is easy to see that program $\mathcal{P}_1$ is stratified, while program $\mathcal{P}_2$ is not. A suitable level mapping for $\mathcal{P}_1$ is the following:

$$||p(a)||_s = 2 \quad ||p(b)||_s = 2 \quad ||q(a)||_s = 1 \quad ||q(b)||_s = 1$$

As for $\mathcal{P}_2$, an admissible level mapping would need to satisfy $||p(a)||_s < ||q(b)||_s$ and $||q(b)||_s < ||p(a)||_s$, which is impossible. □

Within the INFOMIX setting we enable the query to have stratified negation, denoted by Datalog$^{\neg s}$, but in this case IDs cannot be specified over the global schema. A datalog query with stratified negation is is a pair $\langle G, \mathcal{P} \rangle$ where G is a predicate symbol, called *goal*, and $\mathcal{P}$ is a Datalog$^{\neg s}$ program. The semantics is the same of the case of recursive queries, provided the fact that, given an interpretation I, the value of a ground negated literal not L is true if and only if $value_I(L)$ is false.

Example 4.4 The following query computes (using the goal *topEmployee*) the employees which have no other boss than the director.

$$\begin{array}{lll} topEmployee(E) & \text{:-} & employee(E, Salary, Dep, Boss), \\ & & \quad department(Dep, Boss), \text{not } otherBoss(E, Boss). \\ otherBoss(E, Boss) & \text{:-} & employee(E, Salary, Dep, Boss'), Boss' \neq Boss. \end{array}$$

□

In the absence of integrity constraints, in the presence of key dependencies only, or in the presence of key dependencies and exclusion dependencies, query answering is decidable for queries with negation as well. Actually, in the case of IDs issued over the global schema, there is no systematic result up to the present time. Thus, we decided to not allow the user to express negation in such complex setting.

4.4 Aggregate Functions

We conclude the overview on the additional features for the query language by considering aggregate functions.

A *symbolic set* is a pair $\{Vars : Conj\}$, where *Vars* is a list of variables and *Conj* is a conjunction of standard literals.[2] A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$,

[2]Intuitively, a symbolic set $\{X : a(X,Y), p(Y)\}$ stands for the set of X-values making $a(X,Y), p(Y)$ true, i.e., $\{X : \exists Y s.t.\ a(X,Y), p(Y)\ is\ true\}$. Note that also negative literals may occur in the conjunction *Conj* of a symbolic set.

where $\bar{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a set, and f is a *function name* among #`count`, #`min`, #`max`, #`sum`. Let I be an interpretation for a program P. The informal semantics of the aggregate functions are as follows:

- #**count**$\{Vars : Conj\}$. **#count** returns the number of ground instances of $Vars$ which satisfy the conjunction $Conj$ w.r.t. the interpretation I.
- #**sum**$\{Vars : Conj\}$. **#sum** computes the sum of all the values of the first variable in *Vars* in the ground instances satisfying the conjunction $Conj$ w.r.t. the interpretation I.
- #**min**$\{Vars : Conj\}$. Let X be the first variable of *Vars*. Then, the function **#min** returns the minimum value of the variable X among those taken in the ground instances of *Vars* making the conjunction $Conj$ true w.r.t. the interpretation I.
- #**max**$\{Vars : Conj\}$. Let X be the first variable of *Vars*. Then, the function **#max** returns the maximum value of the variable X among those taken in the ground instances of *Vars* making the conjunction $Conj$ true w.r.t. the interpretation I.

An *aggregate atom* is $Lg \prec_1 f(S) \prec_2 Rg$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, and Lg and Rg (called *left guard*, and *right guard*, respectively) are terms. One of "$Lg \prec_1$" and "$\prec_2 Rg$" can be omitted. Given an interpretation I, the aggregate literal is true if and only if the value taken by the aggregate function $f(\mathcal{S})$ belongs to the range specified by the guard values Lg and Rg.

A Datalog (resp. Datalog$^{\neg s}$) program with aggregates is a Datalog (resp. Datalog$^{\neg s}$) program whose atom can be also aggregate atoms.

Example 4.5 The following program computes the employee having the best salary:

$$bestSalary(E, S) \text{ :- } Employee(E, S, _\,, _\,), S = \#\mathbf{max}\{Sal : Employee(_\,, Sal, _\,, _\,)\}.$$

□

The decidability of query answering, when the language is enriched with aggregate operators, is much more intricate than the other cases, and much more efforts must be spent for studying the interaction of aggregate data with complex ICs. We will allow aggregate queries only in the presence of KDs and in the presence of EDs, the only cases where decidability of query answering is known.

5 IQL grammar

We provide in the following a complete specification of the language grammar, given in EBNF notation. Definition of terminal entities like `variable`, `constant`, are not given.

```
program           = goal, {rule};
goal              = "GOAL ", predicatename, ";";
rule              = head, "." | head, ":-", body, ".";
body              = literal, {"," literal }, ".";
head              = atom;
literal           = ["not"], atom | ["not"], aggregateatom | builtin;
atom              = predicatename, [ "(" term {,term} ")" ];
aggregateatom     = guard, operator, aggregatefunction, [operator, guard] |
                    aggregatefunction, operator, guard;
guard             = variable | number;
term              = variable | constant | number | stringconstant;
operator          = "=" | "<" | "<=" | ">" | ">=";
alloperators      = operator | "<>";
builtin           = term, alloperators, term;
aggregatefunction = ( "#count","#min","#max","#sum" ),
                    "{", variable, ["," variable] ":" atom, ["," atom] "}";
```

An IQL program P is valid iff the following conditions hold:

- P is stratified and nonrecursive;
- aggregated atoms are safe in each rule.

Stratification

From the practical point of view, we prefer to introduce the concept of usual stratification, instead of the, more precise, notion of local stratification. The choice is motivated by the consideration that local stratification strictly depends on data which a query is applied on. Thus, it may be confusing for an end-user: a given IQL program may turn out to be or not to be locally stratified (and thus it can be evaluated) depending on the input data. We extend the notion of stratification taking into account aggregate atoms.

Given an IQL program $\mathcal{P}$, let us define a order relation between the predicates of $\mathcal{P}$ as follows. Given two predicates A and B,

- $B \prec A$ if there exists a rule r in $\mathcal{P}$ such that A appears in the head of r, and B appears in an aggregate atom of the body of r.
- $B \preceq A$ if there exists a rule r in $\mathcal{P}$ such that A appears in the head of r, and B occurs in the body of r (as the predicate of a standard atom).

Example 5.1 Let $\mathcal{P}$ be the program

$r_1 : q(X) \text{ :- } s(X).$

$r_2 : q(X) \text{ :- } p(X), \#\mathbf{count}\{Y : a(Y, X), \text{not } b(X)\} \leq 2.$

in which in the body of the rule r_2 appears an aggregate atom of the type *count*. From the rule r_1 it follows that

$$s \preceq q$$

and from r_2 it follows that

$$p \preceq q$$

$$a \prec q$$

$$b \prec q$$

A *stratification* for $\mathcal{P}$ is a partition $\langle S_1, ..., S_n \rangle$ of the set of the predicates of $\mathcal{P}$ that respects the relation " $\prec$ " above defined. Given two predicates A and B belonging to S_i and S_j, respectively, $A \prec B$ implies $i \prec j$, and $A \preceq B$ implies $i \preceq j$.
A program $\mathcal{P}$ is stratified if there exists a stratification for $\mathcal{P}$.

Example 5.2 The program in the example 5.1 is stratified. In fact

$$\langle \{a,\ b,\ \ s,\ p\}, \{q\} \rangle$$

is a stratification for this program. On the contrary, the following is not stratified:

$r_1 : q(X) \text{ :- } p(X).$
$r_2 : p(X) \text{ :- } \#\mathbf{count}\{Y : q(Y)\} \leq 2.$

In fact it is easy to see that no partition of the predicates of $\mathcal{P}$ can satisfy the stratification conditions.

Remark All predicates appearing in an aggregate function must belong to a component of the positive dependency graph, which is strictly lower than the component of the head of the rule.

To check this condition, we will enrich the positive dependency graph with all arcs of the form $h \leftarrow p$, where h is a head predicate and p is a predicate appearing in the aggregate function. Then, we will check that h and p do not belong to the same component of the positive dependency graph.

Safety

Regular head and body variables must respect the standard safety conditions. Suppose that the aggregate atom $\mathcal{F}$, defined in the following, appears in the body of a rule r:

$$Lg \leq f\{Vars : Conj\} \leq Ug$$

then, the following conditions have to be verified:

1. Each guard of $\mathcal{F}$, Lg and Ug, must be either a constant (actually, a non-negative integer) or it must appear also in the set of variables occuring in the body of the rule r. If the guard does not appear in this set, the aggregate literal must be an *assignment*, as shown in example 5.4.

2. Each variable appearing in a negative literal in *Conj* must also appear either among the variables of the body of r or in a positive literal of *Conj*.

Remark Note that variables appearing in the body of r can be saved by an assigning aggregate.

Example 5.3 Let $\mathcal{P}$ be an IQL program made by the following rules:

$r_1 : p(X) \text{ :- } q(X,Y), Y \leq \#\mathbf{count}\{V : a(V)\} \leq Z.$
$r_2 : p(X) \text{ :- } q(X,Y), Y \leq \#\mathbf{sum}\{V : a(V), \text{not } b(Z)\} \leq 5.$
$r_3 : p(X) \text{ :- } q(X,Y), p(Z), Z \leq \#\mathbf{count}\{V : a(V)\} \leq Y.$

The rule r_1 is an example of an unsafe rule. Indeed, the variable Z, representing the upper guard on the aggregate literal *count*, doesn't appear in any other literal of the body of the rule.

Also the rule r_2 violates the safety conditions. In fact, note that the variable Z, occurring in a negative literal of the conjunction inside *sum*, doesn't appear in any other positive literal of the conjunction itself or in the body of the rule.

The rule r_3 on the contrary respects all safety conditions.

Example 5.4 The following is an example of assignment.

$$h(X) \text{ :- } X = \#\mathbf{count}\{V : a(V)\}.$$

All the ground instances of a(V) are counted up and the value of count is assigned to X.

6 IQL Semantics

Let $\mathcal{P}$ be an IQL program, and let $\langle S_1, ..., S_n \rangle$ be a stratification for $\mathcal{P}$. Moreover let r be a rule of $\mathcal{P}$.

A *substitution* for r is a function β mapping the set of variables of r to the set of constants of the Herbrand Universe of $\mathcal{P}$, $U_{\mathcal{P}}$.

Given a conjunction $Conj$, we will denote with $\beta(Conj)$ the conjunction obtained from $Conj$, substituting each variable X appearing in $Conj$, with its image through β, $\beta(X)$.

Definition 6.1 (Local and Global Variables) A *global* variable of r is a variable appearing in some standard atom of r.
A *local* variable of r is a variable X in a set $Vars : Conj$ of an aggregate function of the body of r.

Example 6.1 In the rule r defined as

$$p(X) \text{ :- } q(X,Y), 1 \leq \#\mathbf{sum}\{V : a(V,W), \text{not } b(W,Y)\} \leq X.$$

The variables X and Y are global, while the variables V and W are local.

Definition 6.2 (Instances and preinstances of rules) A *preinstance* of r is a rule $\sigma(r)$ obtained substituting each variable X of r with a constant $\sigma(X)$, where σ is a substitution from the set of the global variables of r in the Herbrand Universe $\mathcal{U}_\mathcal{P}$ of $\mathcal{P}$ (i.e., in the set of all the constants appearing in $\mathcal{P}$). An *instance* r'' of r is obtained from a preinstance r' of r, substituting each symbolic set $Vars : Conj$ occurring in (a function of) r' with the set $\mathcal{Q}$ of the pairs $< \gamma(Vars), \gamma(Conj) >$ where γ is a substitution from the set of the local variables in $U_\mathcal{P}$.

Example 6.2 Let us refer to the rule r defined in the example 6.1. An example of preinstance r' of r in which the constant 2 is associated to the global variable X and the value 1 to the global variable Y, is given by

$$p(2) \text{ :- } q(2,1), 1 \leq \#\mathbf{sum}\{V : a(V,W), \text{not } b(W,1)\} \leq 2.$$

Supposing that the values in $U_\mathcal{P}$ that can be associated to the local variables V and W are 1 and 2, the instance r'' of r, corresponding to the preinstance r' is given by

$$\begin{aligned} p(2) \text{ :- } q(2,1), 1 \leq \#\mathbf{sum}\{&< 1 : a(1,1), \text{not } b(1,1) >, \\ &< 1 : a(1,2), \text{not } b(2,1) >, \\ &< 2 : a(2,1), \text{not } b(1,1) >, \\ &< 2 : a(2,2), \text{not } b(2,1) >\} \leq 2. \end{aligned}$$

The set $\mathcal{Q}$ represents therefore the instantiation of the symbolic set $Vars$:$Conj$. The instantiation ground($\mathcal{P}$) of $\mathcal{P}$ is the set of all the possible instances of all the rules of $\mathcal{P}$. In general an interpretation is used to associate a meaning to an instantiated program ground($\mathcal{P}$). We describe this by transforming our syntactic constructions in a different domain, therefore we have to define an association between constants and objects in the different domain and an association between predicate symbols and functions which associate true or false to a given n-tuple.

Definition 6.3 (Interpretation) An *interpretation* for the program $\mathcal{P}$ is a subset I of ground atoms belonging to the Herbrand Base $B_\mathcal{P}$.

Let $\overline{r}$ be an instance of a rule r of $\mathcal{P}$ and let $f(\mathcal{Q})$ be a function occurring in the body of $\overline{r}$. To define the value of the function $f(\mathcal{Q})$ w.r.t. I, it is necessary to specify the meaning assumed by $\mathcal{Q}$ in I. In fact, $\mathcal{Q}$ becomes a real set when it is valued w.r.t. an interpretation I. The value $I(\mathcal{Q})$ of $\mathcal{Q}$ w.r.t. I is the set

$$\{a : (<a, Conj> \in Q) \wedge (Conj \text{ is true w.r.t. } I)\}.$$

The value $I(f(\mathcal{Q}))$ of $f(\mathcal{Q})$ w.r.t. I is the result of the application of the function f to $I(\mathcal{Q})$.

Example 6.3 Let $f(\mathcal{S})$ be the function

$$\#\textbf{count}\{X : a(X,Y), b(X)\}$$

One of its instantiations $f(\mathcal{Q})$ is given by the application of the function *count* to the set of pairs $\mathcal{Q}$ obtained by the generation of all the possible ground instances of $\mathcal{S}$. In relation with the symbolic set $\mathcal{S}$ of this example, $\mathcal{Q}$ will be given by

$$< 1, (a(1,1), b(1)) >$$

$$< 1, (a(1,2), b(1)) >$$

$$< 2, (a(2,1), b(2)) >$$

$$< 2, (a(2,2), b(2)) >$$

Let us suppose now that it is defined for the program $\mathcal{P}$ in which $f(\mathcal{Q})$ appears, the interpretation I

$$I = \{a(1,1), a(2,1), b(1), b(2)\}$$

The value $I(f(\mathcal{Q}))$ of $f(\mathcal{Q})$ w.r.t. I will be given by the application of *count* to the only instances of $\mathcal{S}$ for which the conjunction of $\mathcal{S}$ is true w.r.t. I, i.e. by

$$\#\textbf{count}\{< 1 : a(1,1), b(1) >, < 2 : a(2,1), b(2) >\}.$$

An aggregate atom

$$A = Lg \leq f(\mathcal{Q}) \leq Ug$$

is true w.r.t. I if both the relations

$$Lg \leq I(f(\mathcal{Q}))$$

$$I(f(\mathcal{Q})) \leq Ug$$

are true w.r.t I.

Example 6.4 Let us consider the function $f(\mathcal{S})$

$$\#\textbf{count}\{X : a(X,Y), b(X)\}$$

and the interpretation I

$$I = \{a(1,1), a(2,1), b(1), b(2)\}$$

already used in the example 6.3 and suppose that $f(\mathcal{S})$ is defined inside the aggregate atom A . Let $\mathcal{A}$ be

$$0 \leq \#\textbf{count}\{f(\mathcal{S})\} \leq 10$$

The instance obtained for the symbolic set $f(\mathcal{S})$ in correspondence of the interpretation I

$$\#\textbf{count}\{< 1 : a(1,1), b(1) >, < 2 : a(2,1), b(2) >\}.$$

returns the value 2. Being this value inside the range of values defined by the two guards of the aggregate atom $\mathcal{A}$ it will be true w.r.t. the interpretation I .

A positive aggregate literal L is true w.r.t. I if the corresponding aggregate atom $\mathcal{A} \in I$; otherwise L is false w.r.t. I. A negative aggregate literal not L is true w.r.t. I if the corresponding aggregate atom $\mathcal{A}$ is false w.r.t. I; otherwise not L is false w.r.t. I.
The notions of satisfaction of the rules, minimal model and stable model for programs with functions, are immediately extended from DLP, through the application of the definitions reported above (the reader may refer to deliverable D1.2, and to [7]). The output of a IQL query will be the extension of the predicate specified within the GOAL statement inside the (unique) stable model of an IQL program.

7 Conclusions

We introduced in this deliverable the Infomix Query Language, which will be used as the query language for the end-user within the Infomix platform. The language is powerful enough to accommodate most common user needs, since IQL features select-project-join capabilities, stratified negation, and aggregates. Some of these capabilities cannot be employed in presence of certain kinds of integrity constraints on the global schema. Anyway, it is specified how to tailor the language implementation in order to identify and filter out such cases.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publ. Co., Reading, Massachussetts, 1995.

[3] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[4] D. Beneventano, S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, and M. Vincini. Information integration: the MOMIS project demonstration. 2000.

[5] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Answering queries using views over description logics knowledge bases. pages 386–391, 2000.

[6] Diego Calvanese and Riccardo Rosati. Answering recursive queries under keys and foreign keys is undecidable. In *In Proc. of the 10th Int. Workshop on Knowledge Representation meets Databases (KRDB 2003)*, pages 3–14, 2003.

[7] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[8] Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, and Divesh Srivastava. The Information Manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, pages 85–91, 1995.

[9] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proc. of the 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2002)*, pages 233–246, 2002.

[10] Alon Y. Levy. Logic-based techniques in data integration. In Jack Minker, editor, *Logic Based Artificial Intelligence*. 2000.

[11] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries on heterogeneous data sources. pages 241–250, 2001.

[12] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[13] Theodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., 1988.

[14] Ron van der Meyden. Logical approaches to incomplete information. In Jan Chomicki and Günter Saake, editors, *Logics for Databases and Information Systems*, pages 307–356. 1998.