A Logic Language for Database Integration*

Gianluigi Greco, Sergio Greco, Ester Zumpano
DEIS
Università della Calabria
87030 Rende, Italy
email: {ggreco,greco,zumpano}@si.deis.unical.it

Abstract

In this paper we present a logic language for the integration of different data sources. The language extends Datalog with several constructs such as nondeterministic choice, set constructors and aggregates. The use of an (extended) Datalog program for the integration of different data sources allows performing the merging process in a more flexible way with respect to the use of predefined operators, already defined in the literature. More specifically the extending Datalog language enables to perform the database integration by writing ad hoc operators reflecting the real user needs. Due to its specific features, the proposed extended language permits to easily implement most of the integration techniques already defined in the literature and can be profitable used to define more general techniques.

Keywords Database integration, Logic programming, Inconsistent databases, Heterogeneous Data Sources.

Contacting author

Ester Zumpano
DEIS
Università della Calabria
87030 Rende, Italy
email: zumpano@si.deis.unical.it
phone: +39 0984 494755
fax: +39 0984 494713

*Work partially supported by a MURST grants under the projects "Data-X" and "D2I". The second author is also supported by ICAR-CNR.

1 Introduction

Data integration is the activity of constructing a unified integrated (either virtual or materialized) view of data from heterogeneous information sources which were designed independently for autonomous applications and whose contents are strictly related. This activity plays a key role in several areas such as data warehousing, database integration, automated reasoning systems, active reactive databases and so it has been deeply investigated especially in the areas of databases and artificial intelligence. However, the database obtained from the integration of multiple autonomous information sources could contain inconsistent data, i.e. data which violate integrity constraints. The following example shows a typical case of inconsistency.

Example 1 Consider the database consisting of the single binary relation *Teaches(Course, Professor)* where the attribute *Course* is a key for the relation. Assume there are two different instances for the relations *Teaches*: $D_1 = \{Teaches(c_1, p_1), Teaches(c_2,\ p_2)\}$ and $D_2 = \{Teaches(c_1, p_1), Teaches(c_2,\ p_3)\}$.

The two instances satisfy the constraint that *Course* is a key, but from their union we derive a relation which does not satisfy the constraint since there are two distinct tuples with the same value for the attribute *Course*. □

Thus the integration of, possibly inconsistent, databases must consider the possibility of constructing an integrated consistent database by replacing inconsistent tuples. For instance, for the integrated relation of the above example, it is possible to obtain a consistent database by replacing the two inconsistent facts $Teaches(c_2, p_2)$ and $Teaches(c_2, p_3)$ with a tuple containing the certain information such as i) $Teaches(c_2, \perp)$, where the null value $\perp$ means that we don't know who teaches course c_2, or ii) the nested tuple $Teaches(c_2, \{p_2, p_3\})$, stating that course c_2 is taught by a professor in the set $\{p_2, p_3\}$ or iii) $Teaches(c_2, p_2)$ (resp. $Teaches(c_2, p_3)$), which means that in the integrated database we give credit to the information coming from D_1 (resp. D_2).

In this paper we propose a general framework for the integration of databases and next present the logic languages supporting the integration phase. More specifically, the integration of databases consists of two steps: in the first one the different values of each attribute of tuples, related to the same concept, are collected into a complex term, and in the second one the complex term is replaced by a simple term whose value is computed by means of some polynomial function applied to the complex term.

The integration framework can be easily implemented by means of a logic language obtained by extending Datalog with set constructors, aggregates, nondeterministic choice. Moreover, the use of an (extended) Datalog program for integrating databases makes the merging process more flexible with respect to the use of predefined operators defined in the literature since it gives us the possibility to write ad hoc operators, ("business rules"), reflecting the real user needs.

Example 2 Consider the database consisting of the single binary relation *Employee(Name, Age, Salary)* where the attribute *Name* is a key for the relation. Assume there are two different instances for the relations *Employee*: $D_1 = \{Employee(Mary, 28, 20000),\ Employee(Peter, 47,\ 50000)\}$ and $D_2 = \{Employee(Mary, 31, 30000),\ Employee(Peter, 47,\ 40000)\}$.

The use of a stratified logic program allows merging such relations, satisfying preference criteria for each attribute. For example, if we suppose certain the information over attribute *Age* provided by D_1, in case of conflicts we can give preference to values of *Age* supplied by

D_1. Anyhow, we can adopt a different strategy for the attribute *Salary* such as considering the average value of those provided by the two sources. In this case the result of the integration consists of the tuples $\{Employee(Mary, 28, 25000), Employee(Peter, 47, 45000)\}$. □

The rest of the paper is as follows. In Section 2, we present basic definition of logic programming and Datalog. In Section 3, we define the database integration problem considered in this paper. In section 4, we present an extension of Datalog with sets, bags and list constructors, aggregate functions and functions for the nondeterministic selection of an element from a complex term. In Section 5, we show that our language can be specialized to capture most of the integration techniques defined in the literature and also to express more general integration techniques.

2 Basic Notions

We assume the existence of finite domains of constants, variables, function symbols and predicate symbols. We also assume the existence of a particular constant $\perp$ denoting a null value. A term is either a variable, a constant or a structure of the form $f(t_1, ..., t_m)$ where f is a function symbol and $t_1, ..., t_m$ are terms. An atom is of the form $p(t_1, ..., t_n)$ where p is a predicate symbol and $t_1, ..., t_n$ are terms. A literal is either an atom A or its negation $\neg A$. A *logic program* (or, simply, a *program*) P is a finite set of rules. Each *rule* of P has the form $A \leftarrow A_1, ..., A_m$ where A is an atom (the *head* of the rule) and $A_1, ..., A_m$ are literals (the *body* of the rule). A rule with an empty body is called a *fact*.

Given a logic program P, the Herbrand universe for P, denoted H_P, is the set of all possible ground terms recursively constructed by considering constants and function symbols occurring in P. The Herbrand Base of P, denoted B_P, is the set of all possible ground atoms whose predicate symbols occur in P and whose arguments are elements from the Herbrand universe. A *ground instance* of a rule r in P is a rule obtained from r by replacing every variable X in r by a ground term in H_P. The set of ground instances of r is denoted by $ground(r)$; accordingly, $ground(P)$ denotes $\bigcup_{r \in P} ground(r)$. An interpretation I of P is a subset of B_P. A ground positive literal A (resp. negative literal $\neg A$) is true w.r.t. an interpretation I if $A \in I$ (resp. $A \notin I$). A conjunction of literals is true in an interpretation I if all literals are true in I. A ground rule is true in I if either the body conjunction is false or the head is true in I. A *(Herbrand) model* M of P is an interpretation that makes each ground instance of each rule in P *true*. A model M for P is minimal if there is no model N for P such that $N \subset M$.

Let I be an interpretation for a program P. The *immediate consequence operator* $T_P(I)$ is defined as the set containing the heads of each rule $r \in ground(P)$ s.t. the body of r is true in I. The semantics of a *positive* (i.e. negation-free) logic program P is given by the unique minimal model; this minimum model coincides with the least fixpoint $T_P^\infty(\emptyset)$ of T_P [19]. Generally, the semantics of logic programs with negation can be given in terms of total stable model semantics [10] which we now briefly recall.

Given a program P and an interpretation M, M is a (*total*) *stable model* of P if it is the minimum model of the positive program P^M defined as follows: P^M is obtained from $ground(P)$ by (i) deleting all rules which have some negative literal $\neg b$ in their body with $b \in M$, and (ii) removing all negative literals from the remaining rules. Logic programs may have zero, one or several stable models. Positive programs have a unique stable model which coincides with the

minimum model [10].

Given a program P and two predicate symbols p and q, we write $p \rightarrow q$ if there exists a rule where q occurs in the head and p in the body or there exists a predicate s such that $p \rightarrow s$ and $s \rightarrow q$. A program is *stratified* if there exists no rule where a predicate p occurs in a negative literal in the body, q occurs in the head and $q \rightarrow p$ i.e. there is no recursion through negation [1]. Stratified programs have a unique stable model which coincides with the *stratified model*, obtained by partitioning the program into an ordered number of suitable subprograms (called 'strata') and computing the fixpoints of every stratum from the lowest one up [1].

A Datalog program is a logic program without function symbols. The Herbrand universe and the Herbrand base of Datalog programs are finite. Datalog programs may have zero, one or several finite stable models. Generally, predicates are partitioned into extensional, defined by a set of ground facts, and intensional, defined by rules and we distinguish between database and program: the database consists of the set of facts defining extensional predicates whereas the program consists of the set of rules defining intensional predicates. In the following, given a database D and a Datalog program P, P_D denotes the Datalog program consisting of the rules in P plus the facts defining the database D.

3 The Database Integration Problem

The process of integrating data from different sources is carried out performing two main yet complementary steps: a first one which consists in the merging of the various relations and a second one in which the possibly inconsistent database is repaired by removing or inserting a minimal set of tuples so that it finally satisfies *integrity* constraints. Before formally introducing the database integration problem let us introduce some basic definitions and notations.

Let R be a relation name, then we denote by: i) *attr(R)* the set of attributes of R, ii) *key(R)* the set of attributes in the primary key of R, iii) $fd(R)$ the set of functional dependencies of R, and iv) *inst(R)* the instance of R (set of tuples). Given a tuple $t \in inst(R)$, $key(t)$ denotes the values of the key attributes of t whereas, for a given database D, $fd(D)$ denotes the set of functional dependencies of D and *inst(D)* denotes the database instance.

We assume that relations associated with the same class of objects have been homogenized with respect to a common ontology, so that attributes denoting the same concepts have the same name [25]. We say that two homogenized relations R and S, associated with the same concept, are *overlapping* if $key(R) = key(S)$. In the following we assume that relations associated with the same class of objects have the same primary key.

The database integration problem is as follows: given n databases $D_1 = \{R_{1,1}, ..., R_{1,k}\}, ..., D_n = \{R_{n,1}, ..., R_{n,k}\}$, computes a database $D = \{T_1, ..., T_k\}$, where each T_j is derived from $R_{1,j}, ...R_{n,j}$ and $R_{1,j}, ...R_{n,j}$ refer to the same class of objects. Thus, the database integration problem consists in the integration of n relations $R_{1,j}, ...R_{n,j}$ into a relation T_j by means of a merge (binary) operator $\diamond$, i.e. $T_j = R_{1,j} \diamond ... \diamond R_{n,j}$. In the following we assume that each database D_i is identified by a unique index i, with $1 \leq i \leq n$ and n denoting the number of databases to be merged. As usual, we use the symbol $\bowtie$ to denote the join operator.

Definition 1 Given two relations R and S such that $attr(R) \subseteq attr(S)$ and two tuples $t_1 \in inst(R)$ and $t_2 \in inst(S)$, we say that t_1 is *less informative* than t_2 ($t_1 \ll t_2$) if for each attribute

a in $attr(R)$, $t_1[A] = t_2[A]$ or $t_1[A] = \perp$, where $\perp$ denotes the null value. Moreover, given two relations R and S, we say that $R \ll S$ if $\forall t_1 \in inst(R)\ \exists t_2 \in inst(S)$ s.t. $t_1 \ll t_2$. □

Definition 2 Let R and S be two relations, a binary operator $\diamond$ such that:

1. $attr(R \diamond S) = attr(R) \cup attr(S)$;
2. $R \bowtie S \ll R \diamond S$;
3. $R \diamond S = S \diamond R \quad (commutativity)$;
4. $(R \diamond S) \diamond R = (R \diamond S) \diamond S = R \diamond S \quad (idempotency)$.

is called *merge operator*. Moreover, a merge operator $\diamond$ is said to be

- *lossless* if, for all R and S, $R \ll (R \diamond S)$ and $S \ll (R \diamond S)$;
- *dependency preserving* if, for all R and S, is $(R \diamond S) \models (fd(R) \cap fd(S))$;
- *associative* if, $(R \diamond S) \diamond T = R \diamond (S \diamond T)$. □

Note that integrating more than two relations by means of a not associative merge operator, may give different results, if we apply the merge operators in different orders. Thus, the associative property is desirable, but several operators defined in the literature do not satisfy such a property.

In order to compare different merge operators we introduce the following definition.

Definition 3 Given two lossless merge operators $\diamond_1$ and $\diamond_2$, we say that $\diamond_1$ is

- *content preferable* to $\diamond_2$ ($\diamond_1 \prec_C \diamond_2$) if, for all R and S, $|R \diamond_1 S| < |R \diamond_2 S|$, and
- *dependency preferable* to $\diamond_2$ ($\diamond_1 \prec_{FD} \diamond_2$) if, for all R and S, the number of tuples in $(R \diamond_1 S)$ which violate the $fd(R) \cap fd(S)$ are less than the number of tuples in $(R \diamond_2 S)$ which violate the $fd(R) \cap fd(S)$. □

The idea through which performing the database integration consists in organizing the collection of tuples with the same value for the key attributes into a 'nested' tuple. In particular, given a set of n conflicting tuples $(t_{1,1}, ..., t_{1,m}), ..., (t_{n,1}, ..., t_{n,m})$, the technique we propose, first replaces the n tuples having arity m, with a tuple $(f_1(\{t_{1,1}, ..., t_{n,1}\}), ..., f_m(\{t_{1,m}, ..., t_{n,m}\}))$, where f_i is a polynomial function which is applied to a set of n elements and returns a ('complex') term for each set; and next uses a polynomial function, g, to combine the complex terms into the output standard tuples.

It is worth nothing that before collecting tuples with the same value for the key attributes, the relations to be merged must be first *reconciled* so that they have the same schema and the same set of key values.

Definition 4 Let $S_1, ..., S_n$ be a set of overlapping relations, K be the key of the relations, then the set of *reconciled* relations $S'_1, ..., S'_n$ is such that the generic S'_i is defined as follows:

- the schema contains all attributes of all homogenized relations, i.e. $attr(S'_i) = \bigcup_{j=1}^{n} attr(S_j)$,
- the instance is constructed as follows:

- it contains all tuples $t \in S_i$ completed with $\perp$ for all attributes belonging to $attr(S'_i) - attr(S_i)$;
- $\forall t' \in S_j$ with $j \neq i$ such that there is no tuple $t'' \in S_i$ with $t'[K] = t''[K]$, it contains a tuple t consisting of $t[K]$ completed with $\perp$ for all attributes. belonging to $attr(S'_i) - K$ □

Observe that $\pi_K(S'_i) = \pi_K(S'_j)$; moreover, if $S_1, ..., S_n$ are consistent, then $|S'_i| = |S'_j| = |\pi_K(S'_i)| = |\pi_K(S'_j)|$, for all $i \neq j$.

Example 3 Consider the following three overlapping relations S_1, S_2 and S_3:

Name	*Dept*	*Salary*
Greg	*Admin*	10000
Jones	*Sales*	20000
Smith	*Sales*	$\perp$

S_1

Name	*City*	*Salary*
Greg	*NY*	25000
Jones	*WA*	20000
Taylor	*WA*	30000
Smith	*WA*	25000

S_2

Name	*Dept*	*Salary*
Greg	*Sales*	20000
Jones	*Sales*	30000

S_3

The homogenized relations S'_1, S'_2 and S'_3 associated to S_1, S_2 and S_3 are:

Name	*Dept*	*City*	*Salary*
Greg	*Admin*	$\perp$	10000
Jones	*Sales*	$\perp$	20000
Smith	*Sales*	$\perp$	$\perp$
Taylor	$\perp$	$\perp$	$\perp$

S'_1

Name	*Dept*	*City*	*Salary*
Greg	$\perp$	*NY*	25000
Jones	$\perp$	*WA*	20000
Smith	$\perp$	*WA*	25000
Taylor	$\perp$	*WA*	30000

S'_2

Name	*Dept*	*City*	*Salary*
Greg	*Sales*	$\perp$	20000
Jones	*Sales*	$\perp$	30000
Smith	$\perp$	$\perp$	$\perp$
Taylor	$\perp$	$\perp$	$\perp$

S'_3

In the following, for the sake of simplicity, we assume that source relations are consistent, although the extension for not consistent relations is trivial as we shall see in Example 5.

Let $S_1, ..., S_n$ be a set of consistent reconciled relations with key attributes K and let $T = S_1 \cup ... \cup S_n$. Let k be a key value in $\pi_K(T)$, then $T^k = [t_1, ..., t_n]$ denotes the list of tuples in T with key k such that $t_i \in S_i$ and is called *cluster* with key value k.

Definition 5 A merge function f is a polynomial function operating on a set of reconciled relations $S_1, ..., S_n$ producing a new set of tuples $R = f(S_1, ..., S_n)$ such that

1. $attr(R) = \cup_i attr(S_i)$;
2. $S_1 \bowtie \cdots \bowtie S_n \ll f(S_1, ..., S_n)$;
3. $S_i \ll f(S_1, ..., S_n)$ for all i;
4. $\pi_K(R) = \pi_K(S_1 \cup ... \cup S_n)$ where K is the key of $S_1, ..., S_n$.

Moreover, we say that f is *decomposable* if it can be applied to the different clusters of $S_1 \cup ... \cup S_n = T$, i.e. $R = f(T^{k_1}) \cup ... \cup f(T^{k_m})$, where $\{k_1, ..., k_m\} = \pi_K(T)$ is the set of keys in T. □

In the following we only consider decomposable functions so that guaranteeing that tuples in different clusters are not 'combined' to produce new tuples.

Definition 6 Let $S_1, ..., S_n$ be a set of consistent reconciled relations with schema $(K, A_1, ..., A_m)$ and key K. Then, every decomposable integrating function f operating on a cluster T^k of $T = S_1 \cup ... \cup S_n$, can be decomposed into $m+1$ polynomial functions $f_1, ..., f_m, g$ such that

- $f(T^k) = g(\{(k, f_1(L_1), ..., f_m(L_m))\})$ with $L_i = [S_1^k, ..., S_n^k] = [\pi_{A_i}(S_1), ..., \pi_{A_i}(S_n)],\ \forall i$,
- $|f_i(L_i)| \leq |L_i|,\ \forall i$, and
- g operates on nested tuples and gives, as result, a set of standard tuples (integrated relation), such that $|g(T^k)| \leq |L_1 \times ... \times L_n|$. □

Moreover, a generic f_i is said to be *canonical* if the following properties hold:

- $f([\perp, ..., \perp]) = \perp$,
- $f(L) = z$ if z is the unique not null value in L.

Example 4 Consider the reconcilied relations S_1', S_2' and S_3' introduced in the Example 3. Applying a generic merge function, the general template of the integrated relation is of the following type:

Name :	*Dept*	*City*	*Salary*
Greg	f_1([Admin,⊥,Sales])	f_2([⊥,NY,⊥])	f_3([10000,25000,20000])
Jones	f_1([Sales,⊥,Sales])	f_2([⊥,WA,⊥])	f_3([20000,20000,30000])
Smith	f_1([Sales,⊥,⊥])	f_2([⊥,WA,⊥])	f_3([⊥,25000,⊥])
Taylor	f_1([⊥,⊥,⊥])	f_2([⊥,WA,⊥])	f_3([⊥,30000,⊥])

T

Assuming that each f_i is a canonical function the template can be simplified as follows:

Name :	*Dept*	*City*	*Salary*
Greg	f_1([Admin,⊥,Sales])	NY	f_3([10000,25000,20000])
Jones	Sales	WA	f_3([20000,20000,30000])
Smith	Sales	WA	25000
Taylor	⊥	WA	30000

T

In the following example we informally show that our framework can be extended to consider inconsistent source relations.

Example 5 Consider the following two overlapping relations S_1 and S_2:

Name :	*Dept*	*Salary*
Greg	*Admin*	10000
Greg	*Sales*	20000
Smith	*Sales*	⊥

S_1

Name :	*City*	*Salary*
Greg	*NY*	25000
Smith	*WA*	20000
Smith	*WA*	25000

S_2

After fixing an order on the tuples of the two input relation we get the following template relation:

Name :	*Dept*	*City*	*Salary*
Greg	f_1([Admin,Sales,⊥])	NY	f_3([10000,20000,25000])
Smith	Sales	WA	f_3([⊥,20000,25000])

T

Several merge operators have been proposed in the literature [5, 18, 25]. All these operators can be defined as special cases of the above framework through the instantiation of the functions g and f_i as we'll be shown next.

4 A Logic Language for Database Integration

In this section we show that the integration of relations can be easily implemented by means of a logic language obtained by extending Datalog with several constructs such as nondeterministic choice, set constructors and aggregates. In particular, we firstly show that the extended Datalog language, due to its specific features, allows expressing most of the relevant merging techniques proposed in the literature and next show that, logical rules define a powerful mechanism to implement more general integration techniques.

In order to manage complex terms we extend the Datalog language by introducing complex terms such as sets, bags and lists; moreover, we also introduce standard SQL functions which are applied to complex terms and a nondeterministic function which selects (nondeterministically) one element from a complex term.

Bags and sets

A (ground) *bag term* S is of the form $\{s_1, ..., s_n\}$, where s_j $(1 \leq j \leq n)$ is a constant and the sequence in which the elements are listed is immaterial. Moreover, a bag term $\{s_1, ..., s_n\}$ is called *set term* if the number of occurrences of every s_j, for $1 \leq j \leq n$, is immaterial. Thus, the three sets $\{a, b\}$, $\{b, a\}$ and $\{b, a, b\}$ coincide, while the two bags $\{a, b\}$ and $\{b, a, b\}$ are different.

We point out that the enumeration of the elements of a set term can be given either directly or by specifying the conditions for collecting their elements (*grouping variables*). Grouping variables may occur in the head of clauses with the following format

$$p(x_1, ..., x_h, <y_1>, ..., <y_k>, \ll y_{k+1} \gg, ..., \ll y_m \gg) \leftarrow B_1, ..., B_n$$

where $B_1, ..., B_n$ are the goals of the rules, p is the head predicate symbol with arity $h + m$, y_i for $1 \leq i \leq m$, is a grouping variable, and $x_1, ..., x_h$ are the other arguments (terms or other grouping variables). To the grouping variable $\ll Y \gg$ (resp. $<Y>$) will be assigned the bag (resp. set) $\{Y\theta \mid \theta$ is a substitution for r such that $B_1\theta, ..., B_n\theta$ are true$\}$. A grouping variable is similar to the construct *GROUP BY* of SQL.

Example 6 Consider the database D consisting of the following facts

$$\mathtt{q(a, 2, x).} \quad \mathtt{q(a, 3, y).} \quad \mathtt{q(b, 4, x).} \quad \mathtt{q(b, 7, y).} \quad \mathtt{q(b, 4, z).}$$

and the program P consisting of the rule:

$$\mathtt{p(X, \ll Y \gg)} \leftarrow \mathtt{q(X, Y, Z)}$$

The program P_D has only one minimal model: $M = D \cup \{\mathtt{p(a, \{2, 3\}), p(b, \{4, 7, 4\})}\}$. Moreover, by replacing the bag constructor with the set constructor we get the rule:

$$\mathtt{p(X, <Y>)} \leftarrow \mathtt{q(X, Y)}$$

The new program has only one minimal model: $M = D \cup \{\mathtt{p(a, \{2, 3\}), p(b, \{4, 7\})}\}$. □

In the following we assume that programs with sets and bags constructors do not appear inside recursive predicates.

Aggregate functions

Beside sets and bags, we also consider built-in aggregate functions, such as *min, max, count, sum* and *avg* which are applied to sets and bags.

Definition 7 An aggregate term is of the form $f(S)$ where S is a grouping variable and $f \in \{min, max, count, sum, avg\}$ is an aggregate function. □

Note that since grouping variables may only occur in the head of rules, aggregate terms only occur in the head of rules too. Observe that $min(\ll S \gg) = min(<S>)$ and $max(\ll S \gg) = max(<S>)$.

Example 7 Consider the database D of the previous example and the program P consisting of the following rules

$$
\begin{array}{lcl}
\mathtt{p_1(X, <Y>)} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_2(X, \ll Y \gg))} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_3(X, min(<Y>))} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_4(X, max(<Y>))} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_5(X, count(<Y>))} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_6(X, count(\ll Y \gg))} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_7(X, sum(<Y>))} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_8(X, sum(\ll Y \gg))} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_9(X, avg(<Y>))} & \leftarrow & \mathtt{q(X, Y, Z)} \\
\mathtt{p_{10}(X, avg(\ll Y \gg))} & \leftarrow & \mathtt{q(X, Y, Z)}
\end{array}
$$

The evaluation of the above rules gives the following facts:

$$
\begin{array}{ll}
\mathtt{p_1(a, \{2, 3\})}, & \mathtt{p_1(\{4, 7\})} \\
\mathtt{p_2(a, \{2, 3\})}, & \mathtt{p_2(\{4, 7, 4\})}, \\
\mathtt{p_3(a, 2)}, & \mathtt{p_3(b, 4)}, \\
\mathtt{p_4(a, 3)}, & \mathtt{p_4(b, 7)}, \\
\mathtt{p_5(a, 2)}, & \mathtt{p_5(b, 2)}, \\
\mathtt{p_6(a, 2)}, & \mathtt{p_6(b, 3)}, \\
\mathtt{p_7(a, 5)}, & \mathtt{p_7(b, 11)}, \\
\mathtt{p_8(a, 5)}, & \mathtt{p_8(b, 15)}, \\
\mathtt{p_9(a, 2.5)}, & \mathtt{p_9(b, 5.5)}, \\
\mathtt{p_{10}(a, 2.5)}, & \mathtt{p_{10}(b, 5)}.
\end{array}
$$

□

Nondeterministic predicates: list constructor and choice

A (*bounded*) *list term* L is a term of the form $[s_1, ..., s_n]$, where s_j $(1 \leq j \leq n)$ is a constant. We shall use the standard notation and the standard *cons* operator so that a not empty list can be denoted by $[X|L]$ where X is the head of the list and L is the tail; the empty list is denoted by $[]$. We also assume the existence of a list constructor which may occur in the head of clauses. Basically a list constructor is of the form $\ll Y \gg_I$ and its semantics is that the elements in the bag $\ll Y \gg$ are ordered with respect to the values of I. Clearly, the variable I must take values from a linearly ordered domain and the result may be nondeterministic since there could be more than one possible orderings.

Example 8 For the database of the previous example, the rule

$$\mathtt{p}(\mathtt{X}, \ll \mathtt{Z} \gg_{\mathtt{Y}}) \leftarrow \mathtt{q}(\mathtt{X}, \mathtt{Y}, \mathtt{Z})$$

computes two facts: $p(a, [x, y])$ and either $p(b, [x, z, y])$ or $p(b, [z, x, y])$. The rule

$$\mathtt{p}(\mathtt{X}, \ll \mathtt{Y} \gg_{\mathtt{Z}}) \leftarrow \mathtt{q}(\mathtt{X}, \mathtt{Y}, \mathtt{Z})$$

computes two facts: $p(a, [2, 3])$ and $p(b, [4, 7, 4])$. □

The list constructor is similar to the built-in predicate *bagof* of PROLOG. As for sets and bags, we assume that the list constructors do not appear inside recursive rules and that aggregates functions can also be applied to lists.[1]

Other than classical aggregate operators, we also consider a nondeterministic function, called *choice*, which selects nondeterministically one element from a set, bag or list. A choice term is of the form $choice(S)$ where S is a grouping variable. Clearly, $choice(<S>) = choice(\ll S \gg) = choice(\ll S \gg_I)$.

Example 9 Consider the database D of Example 6 and the program P consisting of the following rule:

$$\mathtt{p}(\mathtt{X}, \mathtt{choice}(<\mathtt{Y}>)) \leftarrow \mathtt{q}(\mathtt{X}, \mathtt{Y}, \mathtt{Z})$$

The program has four alternative minimal models: $M_1 = \{p(a, 2), p(b, 4)\}$, $M_2 = \{p(a, 2), p(b, 7)\}$, $M_3 = \{p(a, 3), p(b, 4)\}$ and $M_4 = \{p(a, 3), p(b, 7)\}$. □

We also assume the existence of the standard predicates $member(X, L)$ where X is a variable or constant and L is a ground set, bag or list [26]; the predicates assigns to X the elements in the ground term L.

5 Database Integration

The use of an (extended) Datalog program for database integration allows performing the merging process in a more flexible way with respect to the use of predefined operators, already defined in the literature [25]. In fact, due to its specific features, the extending Datalog language here proposed enables to perform the database integration by writing ad hoc operators reflecting the real user needs.

The specialization of the functions f_i and g permits us to express most of the integrating techniques and operators defined in literature. For instance, the merging by majority technique [17] is obtained by specializing all f_i to select the element which occurs a maximum number of times in the set. In other case the function f_i computes a value such as the maximum, minimum, average, etc. Thus, we assume here that the function g returns the set of tuples which can be constructed by combining the values supplied by f_i functions in all possible ways.

For the following, we assume that the input relations $S_1, ..., S_n$ are stored by means of a global set of facts with schema $(i, k, e_1, ..., e_m)$, where i denotes the input relation, k is the set of attributes corresponding to a key and $e_1, ...e_m$ are the remaining attributes.

[1] Lists are basically ordered bags.

The following program computes a relation f integrating a set of relations $S_1, ..., S_n$.

$\texttt{allAtt}(\texttt{K}, \ll\texttt{E}_1\gg_\texttt{I}, ..., \ll\texttt{E}_\texttt{m}\gg_\texttt{I}) \leftarrow \texttt{s}(\texttt{I}, \texttt{K}, \texttt{E}_1, ..., \texttt{E}_\texttt{m}).$
$\texttt{f}(\texttt{K}, \texttt{E}_1, ..., \texttt{E}_\texttt{m}) \leftarrow \texttt{allAtt}(\texttt{K}, \texttt{L}_1, ..., \texttt{L}_\texttt{m}), \texttt{f}_1(\texttt{L}_1, \texttt{E}_1), ..., \texttt{f}_\texttt{m}(\texttt{L}_\texttt{m}, \texttt{E}_\texttt{m})$.

Here the predicate f_i (for all $1 \leq i \leq m$) receives in input a list of elements L_i and returns an element E_i which is used to build output tuples.

Example 10 The content of the predicate *allAttr* for the homogenized relations S'_1, S'_2 and S'_3 of Example 3 is the following

Name :	*Dept*	*City*	*Salary*
Greg	[Admin,⊥,Sales]	[⊥,NY,⊥]	[10000,25000,20000]
Jones	[Sales,⊥,Sales]	[⊥,WA,⊥]	[20000,20000,30000]
Smith	[Sales,⊥,⊥]	[⊥,WA,⊥]	[⊥,25000,⊥]
Taylor	[⊥,⊥,⊥]	[⊥,WA,⊥]	[⊥,30000,⊥]

The body of the second rule implements the different functions f_i. The predicates f_i returns a value; the combination of the different values is used to construct the output tuple. □

The specialization of the predicates $f_1, ..., f_m$ allows to easily implement different merge operators, such as the "Merging by majority", the "merge" and the "prioritized merge" operator. Moreover, since all predicates have the same behavior, we shall use a unique predicate called *mergeAttr*.

Merging by majority

In [18], the aim of obtaining a new relation which is consistent with the integrity constraint is carried out by using an an approach that takes into account the majority view of the knowledge bases if conflicts arises. In order to solve conflicts arising during the merge task, an operator is defined, which takes the majority view for conflicting values.

The "merging by majority" technique, proposed by Lin and Mendelson, tries to remove the conflicts maintaining the (not null) value which is present in the majority of the knowledge bases. However, this approach could fail when, for a field there is not the majority of databases agreeing on a value and, therefore, it is not possible to choose among different alternative values. In this case, for each of this field, a set of possible values is associated in the integrated databases. The formalization as logic program is the following:

$\texttt{mergeAttr}(\texttt{L}, \bot) \leftarrow \texttt{null}(\texttt{L}).$
$\texttt{mergeAttr}(\texttt{L}, \texttt{X}) \leftarrow \texttt{countOccurrences}(\texttt{L}, \texttt{X}, \texttt{N}), \neg\texttt{moreFrequent}(\texttt{L}, \texttt{X}, \texttt{N}).$

$\texttt{moreFrequent}(\texttt{L}, \texttt{X}, \texttt{N}) \leftarrow \texttt{countOccurrences}(\texttt{L}, \texttt{X}, \texttt{N}), \texttt{countOccurrences}(\texttt{L}, \texttt{X}, \texttt{N}_1), \texttt{N}_1 > \texttt{N}.$
$\texttt{countOccurrences}(\texttt{L}, \texttt{X}, \texttt{count}(\ll\texttt{X}\gg)) \leftarrow \texttt{member}(\texttt{X}, \texttt{L}),\ \texttt{X} \neq \bot$.
$\texttt{countOccurrences}(\texttt{L}, \texttt{X}, \texttt{N}) \leftarrow \texttt{countOccurrences}(\texttt{L}, \texttt{X}, \texttt{N}), \texttt{countOccurrences}(\texttt{L}, \texttt{X}, \texttt{N}_1), \texttt{N}_1 > \texttt{N}.$

where the predicate $countOccurrences(L, X, N)$ assigns to N the number of occurrences of X in L.

The Merge Operator

Given a set of homogenized relations $S_1, S_2, ...S_n$, the *merge operator*, introduced in [11], integrates the information provided by each source relation by performing the full outer join and then "extends" each tuple coming from each relation S_i by replacing the possible null values appearing in it with values appearing in some correlated (i.e. with the same key) tuple of all other relations S_j, with $j \neq i$. Let S_1 and S_2 be two homogenized relations over the key K, the merge operation $R = f(S_1, S_2)$ is defined by the following logic program:

$$\begin{array}{lll} \mathtt{r(K, A_1, ..., A_m)} & \leftarrow & \mathtt{s_1(K, B_1, ..., B_m), s_2(K, C_1, ..., C_m), max(B_1, C_1, A_1), ..., max(B_m, C_m, A_m).} \\ \mathtt{r(K, A_1, ..., A_m)} & \leftarrow & \mathtt{s_2(K, B_1, ..., B_m), s_1(K, C_1, ..., C_m), max(B_1, C_1, A_1), ..., max(B_m, C_m, A_m).} \end{array}$$

$$\begin{array}{lll} \mathtt{max(\perp, C, C).} & & \\ \mathtt{max(B, C, B)} & \leftarrow & \mathtt{B} \neq \perp . \end{array}$$

The merging of n homogenized relations can be implemented by using the associative property of the operator; thus, the merging of three relations $f(S_1, S_2, S_3)$ can be defined as applying iteratively the above rules since $f(S_1, S_2, S_3) = f(f(S_1, S_2), S_3)$.

Prioritized merge operator

In order to satisfy preference constraints, we introduce an an asymmetric binary operator, called *prioritized merge operator* $\triangleleft$, which in the case of conflicting relations eliminates the inconsistencies by giving preference to tuples coming from the preferred relation.

The prioritized merge operator can be easily implemented directly. Let S_1 and S_2 be two homogenized relations over the key K, the prioritized merge operation $R = S_1 \triangleleft S_2$ is given by the following logic program:

$$\mathtt{r(K, A_1, ..., A_m)} \leftarrow \mathtt{s_1(K, B_1, ..., B_m), s_2(K, C_1, ..., C_m), max(B_1, C_1, A_1), ..., max(B_m, C_m, A_m).}$$

where *max* has been previously defined. Note that if a new relation, say S_3, has to be involved in the integration process, the program can be reused using the tuples of r, obtained by the merging of S_1 and S_2, as the left relation and S_3 as the right relation.

More general techniques

As previously shown the use of a (stratified) logic program for integrating databases makes the merging process more flexible with respect to the use of predefined operators as it permits to easily implement most of the integration techniques already defined in the literature and allows writing ad hoc operators reflecting the real user needs. In this section we show how, thanks to its specific features, the proposed extended language can be profitable used to define more general integration techniques.

Example 11 Consider the three homogenized relations of Example 3. Assume that we want integrate in the database the information of employees whose global salary is greater than 50 000.

$$\begin{array}{lll} \mathtt{salary(Name, sum(\ll Sal \gg))} & \leftarrow & \mathtt{s(I, Name, Dept, City, Sal).} \\ \mathtt{r(Name, Dept, City, S)} & \leftarrow & \mathtt{salary(Name, Sal),\ Sal > 50000,\ s(I, Name, Dept, City, S).} \end{array}$$

Note that in this case we are assuming that information coming from the different relations are correct and that in the integrated relation the attribute $Name$: is not a key. □

Example 12 Considering Example 3, we want to merge relations S_1, S_2 and S_3 in a way more flexible respect to the result obtained with the traditional merge operators. In particular, having two conflicting tuples, if the value of $Dept$ is the same we want to obtain the average value of the $Salary$, while if an employee is registered in two different department we take the sum of all values of such attribute. Moreover, we assume to give no importance to conflicts on the attribute $city$.

$$\begin{array}{lcl} \texttt{salary(Name, Dept, avg(}\ll\texttt{Sal}\gg\texttt{))} & \leftarrow & \texttt{s(I, Name, Dept, City, Sal).} \\ \texttt{r(Name, }\ll\texttt{Dept}\gg\texttt{, sum(}\ll\texttt{Sal}\gg\texttt{))} & \leftarrow & \texttt{salary(Name, Dept, Sal), s(I, Name, Dept, City, S).} \end{array}$$

□

Example 13 Consider the two relations R and S denoting oriented weighted graphs whose schemata are $(From, To, Length)$ where the pair of attributes $(From, To)$ is a key. The following program P computes first the union of the two relations and next selects arcs (a, b, c) such that there is no path from a to b with length $c' < c$.

$$\begin{array}{lcl} \texttt{rs(From, To, Length)} & \leftarrow & \texttt{r(From, To, Length).} \\ \texttt{rs(From, To, Length)} & \leftarrow & \texttt{s(From, To, Length).} \end{array}$$

$$\begin{array}{lcl} \texttt{closure(From, To, Length)} & \leftarrow & \texttt{rs(From, To, Length).} \\ \texttt{closure(From, To, Length)} & \leftarrow & \texttt{rs(From, X, L1)), closure(X, To, L2), Length = L1 + L2.} \end{array}$$

$$\begin{array}{lcl} \texttt{tc}'\texttt{(From, To, Length)} & \leftarrow & \texttt{rs(From, To, Length), closure(From, To, L), L < Length.} \\ \texttt{tc(From, To, Length)} & \leftarrow & \texttt{rs(From, To, Length), } \neg\texttt{tc}'\texttt{(From, To, Length).} \end{array}$$

□

6 Conclusions

In this paper we have proposed a logic programming language for the integration of different data sources. In particular, the merging of two data sources, say D_1 and D_2, is provided by a stratified Datalog program extended with set constructors, aggregates and nondeterministic choice. It has been proved that the use of a (stratified) logic program for integrating databases makes the merging process more flexible with respect to the use predefined operators, as it enables writing ad hoc operators, ("business rules"), reflecting the real user needs. Moreover we have shown that, due to its specific features, the proposed integration framework permits to easily express many of the merge operators already proposed in literature and that it can be profitable used to define more general techniques. [25].

References

[1] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases.* Addison-Wesley, 1994.

[2] Agarwal, S., Keller, A. M., Wiederhold, G., and Saraswat, K., Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases. *Proc. Int. Conf. on Data Engineering*, 495–504, 1995.

[3] Arenas, M., Bertossi, L., and Chomicki, J., Consistent Query Answers in Inconsistent Databases. *PODS Conf.*, 68–79, 1999.

[4] Arenas, M., Bertossi, L., and Chomicki, J., Specifying and Querying Database repairs using Logic Programs with Exceptions. *FQAS* Conf., 27–41, 2000.

[5] Baral, C., Kraus, S., and Minker, J., Combining Multiple Knowledge Bases. *IEEE-TKDE*, Vol. 3(2), 208–220, 1991.

[6] Baral, C., Kraus, S., Minker, J., and Subrahmanian, V. S., Combining Knowledge Bases Consisting of First Order Theories. *Proc. ISMIS Conference*, pp. 92–101, 1991.

[7] Bry, F., Query Answering in Information System with Integrity Constraints, *IFIP 11.5 Working Conf.*, 113–130, 1997.

[8] M. Denecker, N. Pelov, and M. Bruynooghe, Well-founded and stable semantics for logic programs with aggregates, *ICLP 2001* 212–226,2001.

[9] Dung, P. M., Integrating Data from Possibly Inconsistent Databases. *CoopIS Conf.*, 58–65, 1996.

[10] Gelfond, M., and Lifschitz, V. The Stable Model Semantics for Logic Programming, *ICLP Conf.* 1070–1080, 1988.

[11] Greco, S., and Zumpano E., Querying Inconsistent Database *LPAR Conf.*, 308–325, 2000.

[12] Grant, J., and Subrahmanian, V. S., Reasoning in Inconsistent Knowledge Bases. *IEEE-TKDE*, Vol. 7(1), 177–189, 1995

[13] Kanellakis, P. C., Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. 2, J. van Leewen (ed.), North-Holland, 1991.

[14] Kowalski, R. A., and Sadri, F., Logic programs with exceptions. *NGC J.*, 9(No. 3/4), 387–400, 1991.

[15] Lin, J., A Semantics for Reasoning Consistently in the Presence of Inconsistency. *Artificial Intelligence*, Vol. 86(1), 75–95, 1996.

[16] Lin, J., Integration of Weighted Knowledge Bases. *Artificial Intelligence*, Vol. 83(2), 363–378, 1996.

[17] Lin, J., and Mendelzon, A. O., Merging Databases Under Constraints. *Int. Journal of Cooperative Information Systems* Vol. 7(1),5-5-76, 1998.

[18] Lin, J., and Mendelzon, A. O., Knowledge Base Merging by Majority, in *Dynamic Worlds: From the Frame Problem to Knowledge Management*, R. Pareschi and B. Fronhoefer (eds.), Kluwer, 1999.

[19] Lloyd, J., *Foundation of Logic Programming.* Spinger-Verlag, 1987.

[20] Minker, J. On Indefinite Data Bases and the Closed World Assumption, *6-th Conf. on Automated Deduction*, 292–308, 1982.

[21] Pradhan, S., Minker, J., and Subrahmanian, V. S., Combining Databases with Prioritized Information. JIIS 4(3): 231-260, 1995.

[22] Monotonic aggregation in deductive databases, *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database Systems* 114–126,1992.

[23] Subrahmanian, V. S., Amalgamating Knowledge Bases. *ACM-TODS*, Vol. 19(2), pp. 291-331, 1994.

[24] Ullman, J. K., *Principles of database and knowledge-base systems*, Vol. 1, Comp. Science Press, 1988.

[25] Yan, L.L., and Ozsu, and M.T., Conflict Tolerant Queries in Aurora. In *CoopIS Conf.*, 279–290, 1999.

[26] Zaniolo, C., Arni, N., and Ong, Q., The LDL++ system *TKDE*, 1992.