Formalizing a Broader Recursion Coverage in SQL

Gabriel Aranda[1], Susana Nieva[1], Fernando Sáenz-Pérez[2], and Jaime Sánchez-Hernández[1] *

[1] Dept. Sistemas Informáticos y Computación, UCM, Spain
[2] Dept. Ingeniería del Software e Inteligencia Artificial, UCM, Spain
garanda@fdi.ucm.es, {nieva,fernan,jaime}@sip.ucm.es

Abstract. SQL is the *de facto* standard language for relational databases and has evolved by introducing new resources and expressive capabilities, such as recursive definitions in queries and views. Recursion was included in the SQL-99 standard, but this approach is limited as only linear recursion is allowed, mutual recursion is not supported, and negation cannot be combined with recursion. In this work, we propose a new approach, called R-SQL, aimed to overcome these limitations and others, allowing in particular cycles in recursive definitions of graphs and mutually recursive relation definitions. In order to combine recursion and negation, we import ideas from the deductive database field, such as stratified negation, based on the definition of a dependency graph between relations involved in the database. We develop a formal framework using a stratified fixpoint semantics and introduce a proof-of-concept implementation.

Keywords: Databases, SQL, Recursion, Fixpoint Semantics

1 Introduction

Codd's famous paper on relational model [2] sowed the seeds for current relational database management systems (RDBMS's), such as DB2, Oracle, MySQL, SQL Server and others. Formal query languages were proposed for the relational model: Relational algebra (RA) and relational calculus, which are syntactically different but semantically equivalent w.r.t. safe formulas [16]. Such RDBMS's rather rely on the SQL query language (current standard SQL:2008 [7]) that departs from the relational model and goes beyond. Its acknowledged success builds upon an elegant and yet simple formulation of a data model with relations which can be queried with a language including some basic RA-operators, which are all about relations. Original operators became a limitation for practical applications of the model, and others emerged to fill some gaps, including, for instance, aggregate operators for, e.g., computing running sums and averages.

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01 (FAST-STAMP), S2009/TIC-1465 (PROMETIDOS), and GPD-UCM-A-910502.

Other additions include representing absent or unknown information, which delivered the introduction of null values and outer join operators ranging over such values. Also, duplicates were introduced to account for bags (multisets) instead of sets. Finally, we mention the inclusion of recursion (Starburst [10] was the first non-commercial RDBMS to implement this whereas IBM DB2 was the first commercial one), a powerful feature to cope with queries that must be otherwise solved by the intermixing with a host language. However, as pointed out by many (see, e.g., [9],[13]), the relational model has several limitations. Thus, such current RDBMS's include that extended "relational" model, which is far from the original one and it is even heavily criticized [3] because of nulls and duplicates.

In this work, we focus on the inclusion of recursion in SQL as current RDBMS's lack both a formal support and suffer a narrow coverage of recursion. Regarding formalization, an extension of the RA is presented in [1], with a looping construct and assignment in order to deal with the integration of recursion and negation. [5] is the source of the original SQL-99 proposal for recursion, which is based on the research in the areas of logic programming and deductive databases [16], as explained in [4]. Another example of an approach built on an extension of RA with a fixpoint construct is in [6]. However, as far as we know, these formalizations do not lead to concrete implementations, while our proposal provides an operational mechanism allowing a straightforward implementation.

Regarding recursion coverage, there are several main drawbacks in current implementations of recursion: Linearity is required, so that relation definitions with calls to more than one recursive relation are not allowed. Some other features are not supported: Mutual recursion, and query solving involving an EXCEPT clause. In general, termination is manually controlled by limiting the number of iterations instead of detecting that there are no further opportunities to develop new tuples.

Here, we propose R-SQL, a subset of the SQL standard to cope with recursive definitions which are not restricted as current RDBMS's do, and also allowing neater formulations by allowing concise relation definitions (much following the assignment RA-operator) and avoiding extensive writings (cf. Section 2). For this language, first we develop a novel formalization based on stratified interpretations and a fixpoint operator to support theoretical results (cf. Section 3). And, second, we propose a proof-of-concept implementation which takes a set of database relation (in general, recursive) definitions and computes their meanings (cf. Section 4). This implementation uses the underlying host SQL system and Python to compute the outcome, and can be easily adapted to be integrated as a part of any state-of-the-art RDBMS. Section 5 concludes and presents some further work.

2 Introducing R-SQL

In this section, we present the language R-SQL by using a minimal syntax that allows to capture the core expressiveness of standard SQL. Namely, we consider

basic SQL constructs to cover relational algebra. Nevertheless, this language is conceived to be able to be extended in order to incorporate other usual features. R-SQL is focused on the incorporation of recursive relation definitions. The idea is simple and effective: A relation is defined with an assignment operation as a named query (view) that can contain a self reference, i.e., a relation `R` can be defined as `R sch :=` SELECT ... FROM ... `R` ..., where `sch` is the relation schema. Next, we introduce the formal grammar of this language, then we show by means of examples the benefits of R-SQL w.r.t. current RDBMS systems.

2.1 Syntax of R-SQL

The formal syntax of R-SQL is defined by the grammar in Figure 1. In this grammar, productions start with lowercase letters whereas terminals start with uppercase (SQL terminal symbols use small caps). Optional statements are delimited by square brackets and alternative sentences are separated by pipes. The grammar defines the following syntactic categories:

- A database `sql_db` is a (non-empty) sequence of relation definitions separated by semicolons (";"). A relation definition assigns a select statement to the relation, that is identified by its name `R` and its schema.
- A schema `sch` is a tuple of attribute names with their corresponding types.
- A select statement `sel_stm` is defined in the usual way. The clauses FROM and WHERE are optional. We also allow UNION and EXCEPT, but notice that the syntax for EXCEPT allows only a relation name instead of a select statement

```
sql_db  ::= R sch := sel_stm;
            ...
            R sch := sel_stm;

sch     ::= (A T,...,A T)

sel_stm ::= SELECT exp,...,exp [ FROM R,...,R [ WHERE wcond ] ]
            | sel_stm UNION sel_stm
            | sel_stm EXCEPT R

exp     ::= C | R.A | exp opm exp | - exp

wcond   ::= TRUE | FALSE | exp opc exp | NOT (wcond)
            | wcond [ AND | OR ] wcond

opm     ::= + | - | / | *

opc     ::= = | <> | < | > | >= | <=
```

`R` stands for relation names, `A` for attribute names, `T` for standard SQL types (as INTEGER, FLOAT, VARCHAR(N)), and `C` for constants belonging to a valid SQL type.

Fig. 1. A Grammar for the R-SQL Language

as usual in SQL. This is done in order to keep simple the syntax and does not imply expressivity losses, because a relation name can be identified with the select statement that defines it.
- An expression `exp` can be either a constant value `C`, an attribute of a relation (denoted by `R.A`), or an arithmetic expression.
- A Boolean condition `wcond` in the WHERE clause of a select statement is built up in the usual way, using also the standard comparison operators.

Below, we show a syntactic transformation $[\_]_{\mathcal{RA}}$ that maps every select statement to an equivalent RA-expression in the usual way[3].

- $[\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_m \text{ WHERE } \mathtt{wcond}]_{\mathcal{RA}} = \pi_{\mathtt{exp}_1,\ldots,\mathtt{exp}_k}(\sigma_{\mathtt{wcond}}(\mathtt{R}_1 \times \ldots \times \mathtt{R}_m))$
- $[\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2]_{\mathcal{RA}} = [\mathtt{sel\_stm}_1]_{\mathcal{RA}} \bigcup [\mathtt{sel\_stm}_2]_{\mathcal{RA}}$
- $[\mathtt{sel\_stm} \text{ EXCEPT } \mathtt{R}]_{\mathcal{RA}} = [\mathtt{sel\_stm}]_{\mathcal{RA}} - \mathtt{R}$

The formal meaning of every `sel_stm` w.r.t. an interpretation I, stated in Definition 5 (Section 3), evinces the idea that the expected interpretation of a select statement $[\![\mathtt{sel\_stm}]\!]^I$ should be the set of tuples associated to the corresponding equivalent RA-expression $[\mathtt{sel\_stm}]_{\mathcal{RA}}$.

2.2 Expressiveness of R-SQL

Next, we illustrate that R-SQL overcomes some limitations present in current RDBMS's following SQL-99. These languages use NOT EXITS and EXCEPT clauses to deal with negation, and WITH RECURSIVE to engage recursion. As it is pointed out in [5], SQL-99 does not allow an arbitrary collection of mutually recursive relations to be written in the WITH RECURSIVE clause. Although any mutual recursion can be converted to direct recursion by inlining [8], our proposal allows to explicitly define mutual recursive relations, which is an advantage in terms of program readability and maintenance. For instance, using R-SQL, it is easy to write the classical example for computing even and odd numbers up to a bound (100 in the example) as follows:

```
even(x float) := SELECT 0 UNION
  SELECT odd.x+1 FROM odd WHERE odd.x<100;

odd(x float) := SELECT even.x+1 FROM even WHERE even.x<100;
```

Further, linear recursion in standard SQL restricts the number of allowed recursive calls to be only one, i.e., Fibonacci numbers cannot be specified as follows[4]:

[3] Notice that arithmetic expressions are allowed as arguments in *projection* (π) and *select* (σ) operations.

[4] The relations `fib1` and `fib2` simply represent two aliases for `fib`, which are necessary because, for simplicity, we have not added support for renamings in R-SQL FROM clauses.

```
fib1(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib2(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib(n float, f float) := SELECT 0,1 UNION SELECT 1,1 UNION
  SELECT fib1.n+1,fib1.f+fib2.f FROM fib1,fib2
  WHERE fib1.n=fib2.n+1 AND fib1.n<10;
```

This means that several graph algorithms specified using non-linear recursion cannot be directly expressed in current recursive SQL systems [17].

Non-termination is another problem that arises associated to recursion. For instance, the basic transitive closure over a graph that includes a cycle makes current SQL systems (such as PostgreSQL and MySQL) either to reject the query or to go into an infinite loop (some systems allow to impose a maximum number of iterations as a simple termination condition). Nevertheless, the fix-point computation used by R-SQL guarantees termination when dealing with finite relations. The following example written in R-SQL defines the relations `arc` (a graph with a cycle) and `path` (its transitive closure). The computation is terminating since both relations are finite.

```
arc(ori varchar(1), des varchar(1)) :=
  SELECT a,b UNION SELECT b,c UNION SELECT c,a;

path(ori varchar(1), des varchar(1)) :=
  SELECT arc.ori, arc.des FROM arc UNION
  SELECT arc.ori, path.des FROM arc,path WHERE arc.des=path.ori
```

The following running example contains a concrete relation defined using the classical transitive closure technique mentioned above.

Example 1. A database for flights. As usual, the information about direct flights can be composed of the city of origin, the city of destination, and the length of the flight. Cities (Lisboa, Madrid, Paris, London, New York) will be represented with constants (`lis`, `mad`, `par`, `lon`, `ny`, resp.)

```
flight(frm varchar(10), to varchar(10), time float) :=
  SELECT 'lis','mad',1.0 UNION SELECT 'mad','par',1.5 UNION
  SELECT 'par','lon',2.0 UNION SELECT 'lon','ny',7.0  UNION
  SELECT 'par','ny',8.0;
```

The relation `reachable` consists of all the possible trips between the cities of the database, maybe concatenating more than one flight.

```
reachable(frm varchar(10), to varchar(10)) :=
  SELECT flight.frm, flight.to FROM flight UNION
  SELECT reachable.frm, flight.to FROM reachable,flight
  WHERE reachable.to = flight.frm;
```

The relation `travel` also gives time information about alternative trips.

```
travel(frm varchar(10), to varchar(10), time float) :=
  SELECT flight.frm, flight.to, flight.time
  FROM flight UNION
  SELECT flight.frm, travel.to, flight.time+travel.time
  FROM flight, travel WHERE flight.to = travel.frm;
```

Both `reachable` and `travel` represent transitive closures of the relation `flight`. Notice that if `flight` has a cycle, then the relation `travel` that includes times for each trip is infinite, while `reachable` is not. As pointed before, `reachable` can be finitely computed in our system. But, as `travel` would produce an infinite set of different tuples, some computation limitation would have to be imposed (as the maximum time for a `travel`, for example). However, this is not a drawback of our approach, but an issue due to using infinite relations (built with arithmetic expressions).

The relation `madAirport` contains travels departing or arriving in Madrid, while `avoidMad` contains possible travels that neither begin, nor end in Madrid.

```
madAirport(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable
  WHERE (reachable.frm = 'mad' OR reachable.to = 'mad');

avoidMad(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable
  EXCEPT madAirport;
```

This definition includes negation together with recursive relations. This combination can not be expressed in SQL-99 as it is shown in [4].

3 A Stratified Fixpoint Semantics for R-SQL

It is well-known that the combination of negation and recursion in database languages is a difficult task [1]. This problem has been tackled with stratified fixpoint semantics in several works [12, 11, 14], and we have found that these techniques can be also applied to our proposal to obtain an operational semantics for R-SQL. In this section we present a novel formalization of recursive SQL relations by means of a stratified fixpoint interpretation that formalizes the meaning of R-SQL-databases, and we show how to compute such fixpoint.

Next, we introduce the notions of dependency graph and stratification that provide the basis for the stratified negation formalization we are looking for. Then, we define the concept of stratified interpretations, and prove the existence of the fixpoint of a continuous operator as the required interpretation of a database. The obtained semantics will be the basis of the implementation of a concrete R-SQL database system.

3.1 Dependency Graph and Stratification

Stratification is based on the definition of a *dependency graph* for a database. In the following, we consider a database `sql_db` defined as $\texttt{R}_1\texttt{sch}_1 := \texttt{sel\_stm}_1 ; \ldots ;$

$\texttt{R}_n\texttt{sch}_n := \texttt{sel\_stm}_n$. We denote by RN the set $\{\texttt{R}_1, \ldots, \texttt{R}_n\}$ of relation names of sql_db. We assume that relations are well defined, in the sense that the relation names used inside $\texttt{sel\_stm}_1 \ldots \texttt{sel\_stm}_n$ are in RN. The dependency graph associated to sql_db, denoted by $DG_{\texttt{sql\_db}}$, is a directed graph whose nodes are the elements of RN, and the edges, that can be *negatively labelled*, are determined by the dependencies between the database relations, that are defined as follows. A relation definition of the form `R sch := sel_stm` produces edges in the graph from every relation name inside sel_stm to R. Those edges produced by the relation name that is just to the right of an EXCEPT are negatively labelled.

Definition 1. For every two relations $\texttt{R}_1$, $\texttt{R}_2 \in \texttt{RN}$, we say:

- $\texttt{R}_2$ *depends* on $\texttt{R}_1$ if there is a path from $\texttt{R}_1$ to $\texttt{R}_2$ in $DG_{\texttt{sql\_db}}$.
- $\texttt{R}_2$ *negatively depends* on $\texttt{R}_1$ if there is a path from $\texttt{R}_1$ to $\texttt{R}_2$ in $DG_{\texttt{sql\_db}}$ with at least one negatively labelled edge.

Example 2. Consider the database of Example 1. Its corresponding set of relation names is $\texttt{RN} = \{\texttt{flight}, \texttt{reachable}, \texttt{travel}, \texttt{madAirport}, \texttt{avoidMad}\}$. Its dependency graph is depicted in Figure 2, where negatively labelled edges are annotated with $\neg$.

Definition 2. A *stratification* of sql_db is a mapping $str : \texttt{RN} \to \{1, \ldots, n\}$, such that:

- $str(\texttt{R}_i) \leq str(\texttt{R}_j)$, if $\texttt{R}_j$ depends on $\texttt{R}_i$,
- $str(\texttt{R}_i) < str(\texttt{R}_j)$ if $\texttt{R}_j$ negatively depends on $\texttt{R}_i$.

sql_db is *stratifiable* if there exists a stratification for it. In this case, for every $\texttt{R} \in \texttt{RN}$, we say that $str(\texttt{R})$ is the *stratum* of R. We denote by *numstr* the maximum stratum of the elements of RN. And $str(\texttt{sel\_stm})$ represents the maximum stratum of the relations included in sel_stm.

Intuitively, a relation name preceded by an EXCEPT plays the role of a negated predicate (relation) in the deductive database field. A stratification-based solving procedure ensures that when a relation that contains an EXCEPT in its definition is going to be calculated, the meaning of the inner negated relation has been completely evaluated, avoiding nonmonotonicity, as it is widely studied in Datalog [16]. The novelty lies on introducing these ideas into the field of the relational model.

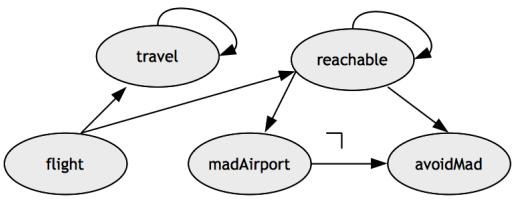

Fig. 2. $DG_{\texttt{sql\_db}}$ of Example 1

3.2 Stratified Interpretations and Fixpoint Operator

From now on, we consider a stratifiable `sql_db`, and that *str* is a stratification for it. In the previous section, we established that in a relation definition for `R sch`, the schema `sch` is a sequence of type declarations for the attributes of `R`. In order to give meaning to this relation, we assume that every type `T` included in `sch` denotes a domain D. In previous examples we have used two types: `varchar`, denoting the domain of strings, and `float`, denoting a numeric domain. We will consider a *universal domain* $\mathcal{D}$ which is the union of the family of the considered domains. Relations of arity k will denote a set of k-tuples included in $\mathcal{D}^k$. In general, every relation denotes a subset of $\mathcal{T} = \bigcup_{n\geq 1} \mathcal{D}^n$.

Interpretations are functions that associate an element of $\mathcal{P}(\mathcal{T})$ to each element of `RN`. So, considering the usual relational model terminology of schema and instance of a relation, the interpretation of a relation in our model can be seen as the relationship between the schema and the instance of the relation. Interpretations are classified by strata. An interpretation of a stratum i gives meaning to the relations of strata less or qual to i. Next, we formalize the concept of interpretation over a stratum.

Definition 3. An *interpretation* I for `sql_db`, over the stratum i, $1 \leq i \leq numstr$ is a function from `RN` to $\mathcal{P}(\mathcal{T})$, such that, for each `R` $\in$ `RN`:

- If `R` has schema $(\mathtt{A}_1\mathtt{T}_1, \ldots, \mathtt{A}_r\mathtt{T}_r)$, and $D_1, \ldots, D_r$ are, respectively, the domains denoted by $\mathtt{T}_1, \ldots, \mathtt{T}_r$, then $I(\mathtt{R}) \subseteq D_1 \times \ldots \times D_r$.
- $I(\mathtt{R}) = \emptyset$, if $str(\mathtt{R}) > i$.

The set of interpretations for `sql_db` over the stratum i, $1 \leq i \leq numstr$ is denoted by $\mathcal{I}_i^{\mathtt{sql\_db}}$. The following definition provides an order on $\mathcal{I}_i^{\mathtt{sql\_db}}$.

Definition 4. Let $i \geq 1$, and $I_1, I_2 \in \mathcal{I}_i^{\mathtt{sql\_db}}$. I_1 *is less or equal than* I_2 *at stratum* i, denoted by $I_1 \sqsubseteq_i I_2$, if the following conditions are satisfied for every `R` $\in$ `RN`:

- $I_1(\mathtt{R}) = I_2(\mathtt{R})$, if $str(\mathtt{R}) < i$.
- $I_1(\mathtt{R}) \subseteq I_2(\mathtt{R})$, if $str(\mathtt{R}) = i$.

It is straightforward to check that for any i, $1 \leq i \leq numstr$, $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$ is a poset. The main question is that when an interpretation over a stratum i increases, the set of tuples associated to the relations whose stratum is i can increase, but the sets associated to relations of smaller strata remain invariable. In addition, this poset is a cpo, as it is proved in the following lemma.

Lemma 1. For any $i \geq 1$, the pair $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$ is a complete partially ordered set. Moreover, if $\{I_n\}_{n\geq 0}$ is a chain of interpretations in $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$, then $\hat{I}$, defined as $\hat{I}(\mathtt{R}) = \bigcup_{n\geq 0} I_n(\mathtt{R})$, is the least upper bound of $\{I_n\}_{n\geq 0}$.

Proof. It is easy to prove that $\hat{I} \in \mathcal{I}_i^{\mathtt{sql\_db}}$, and that it is an upper bound. In addition, if I is another upper bound, that implies: If $str(\mathtt{R}) < i$, $I(\mathtt{R}) = I_n(\mathtt{R})$ for every $n \geq 0$, and hence $\hat{I}(\mathtt{R}) = I(\mathtt{R})$. If $str(\mathtt{R}) = i$, $I_n(\mathtt{R}) \subseteq I(\mathtt{R})$ for every $n \geq 0$, then $\bigcup_{n\geq 0} I_n(\mathtt{R}) \subseteq I(\mathtt{R})$. Therefore $\hat{I} \sqsubseteq_i I$, by the definition of $\sqsubseteq_i$. □

The following definition formalizes the meaning of a select statement sel_stm in the context of a concrete interpretation I, both associated to a concrete sql_db database. As we pointed out before, the interpretation of a sel_stm will be the set of tuples associated to its corresponding RA-expression, $[\![\texttt{sel\_stm}]\!]_{\mathcal{RA}}$, when the value of the involved relation names is given by I.

Definition 5. Let $i \geq 1$, and $I \in \mathcal{I}_i^{\texttt{sql\_db}}$. Let sel_stm be a select statement including only relation names of RN, such that $str(\texttt{sel\_stm}) \leq i$. We recursively define the *interpretation of* sel_stm *w.r.t.* I, denoted by $[\![\texttt{sel\_stm}]\!]^I$, as:

- $[\![\texttt{sel\_stm}_1 \text{ UNION } \texttt{sel\_stm}_2]\!]^I = [\![\texttt{sel\_stm}_1]\!]^I \bigcup [\![\texttt{sel\_stm}_2]\!]^I$, where $\bigcup$ stands for the set union.
- $[\![\texttt{sel\_stm} \text{ EXCEPT } \texttt{R}]\!]^I = [\![\texttt{sel\_stm}]\!]^I \setminus I(\texttt{R})$, where $\setminus$ represents set difference.
- $[\![\text{SELECT } \texttt{exp}_1, \ldots, \texttt{exp}_k]\!]^I = \{(exp_1, \ldots, exp_k)\}$, where exp_i is the mathematical evaluation of $\texttt{exp}_i$.
- $[\![\text{SELECT } \texttt{exp}_1, \ldots, \texttt{exp}_k \text{ FROM } \texttt{R}_1, \ldots, \texttt{R}_m \text{ WHERE } \texttt{wcond}]\!]^I =$ $\{(exp_1[\overline{a}/\overline{\texttt{A}}], \ldots, exp_k[\overline{a}/\overline{\texttt{A}}]) | \overline{a} \in I(\texttt{R}_1) \times \ldots \times I(\texttt{R}_m)$ and $wcond[\overline{a}/\overline{\texttt{A}}]$ is satisfied$\}$.

$\overline{\texttt{A}}$ is a sequence of attributes labelled with their corresponding relation names. More precisely, if $\texttt{A}_1^j, \ldots, \texttt{A}_{r_j}^j$ are the attributes of $\texttt{R}_j$, $1 \leq j \leq m$, then $\overline{\texttt{A}}$ represents the complete sequence $\texttt{R}_1.\texttt{A}_1^1, \ldots, \texttt{R}_1.\texttt{A}_{r_1}^1, \ldots, \texttt{R}_m.\texttt{A}_1^m \ldots \texttt{R}_m.\texttt{A}_{r_m}^m$. $exp_j[\overline{a}/\overline{\texttt{A}}]$, $1 \leq j \leq k$, is the mathematical evaluation of $\texttt{exp}_j$, after replacing the tuple $\overline{a}$ by $\overline{\texttt{A}}$. And $wcond[\overline{a}/\overline{\texttt{A}}]$ is the evaluation of the Boolean expression wcond, with the previous substitution.

Example 3. Consider the definitions of the relations odd and even of Section 2.2. Let us assume a concrete interpretation I such that $I(\texttt{even}) = \{(0), (2)\}$ and $I(\texttt{odd}) = \emptyset$. Hence, the interpretation of the select statement that defines the relation odd w.r.t. I is:
$[\![\texttt{SELECT even.x+1 FROM even WHERE even.x<100}]\!]^I = \{(\texttt{even.x+1})[a/\texttt{even.x}]$ $|(a) \in I(\texttt{even})$ and $(\texttt{even.x<100})$ $[a/\texttt{even.x}]$is satisfied$\} = \{(1), (3)\}$.

The case of the relation even is analogous:
$[\![\texttt{SELECT 0 UNION SELECT odd.x+1 FROM odd WHERE odd.x<100}]\!]^I =$
$[\![\texttt{SELECT 0}]\!]^I \bigcup [\![\texttt{SELECT odd.x+1 FROM odd WHERE odd.x<100 }]\!]^I = \{(\texttt{0})\} \bigcup$
$\{(\texttt{odd.x+1})[a/\texttt{odd.x}]$ $|(a) \in I(\texttt{odd}), (\texttt{odd.x<100})[a/\texttt{odd.x}]$ is satisfied$\} = \{(\texttt{0})\}$.

Notice that the interpretation $\hat{I}$ defined by $\hat{I}(\texttt{even}) = \{(0), (2), \ldots, (100)\}$ and $\hat{I}(\texttt{odd}) = \{(1), (3), \ldots, (99)\}$ satisfies:
$\hat{I}(\texttt{even}) = [\![\texttt{SELECT 0 UNION SELECT odd.x+1 FROM odd WHERE odd.x<100}]\!]^{\hat{I}}$.
$\hat{I}(\texttt{odd}) = [\![\texttt{SELECT even.x+1 FROM even WHERE even.x<100}]\!]^{\hat{I}}$.

The semantics of sql_db will be formalized by means of an interpretation I over *numstr*, such that for every $\texttt{R} \in \texttt{RN}$, if R sch := sel_stm is the definition of R in sql_db, then I maps the set $[\![\texttt{sel\_stm}]\!]^I$ to R, as the interpretation $\hat{I}$ of Example 3 does. For every stratum i, the appropriate interpretation that gives the complete meaning to each relation of stratum i is the least fixpoint of a continuous operator over the set of interpretations of stratum i. These fixpoint interpretations are constructed sequentially from stratum 1 to *numstr*.

The fixpoint of the last stratum *numstr* provides the semantics for the whole database. Some technical lemmas are shown in order to ensure the existence of such fixpoint interpretations.

The following lemma states that the sets of tuples denoted by a select statement of a stratum i, w.r.t. two ordered interpretations, satisfy an inclusion relation that is in accordance with the order $\sqsubseteq_i$ between the two interpretations.

Lemma 2. Let $i \geq 1$, $\texttt{R} \in \texttt{RN}$, with $str(\texttt{R}) \leq i$, and I_1, $I_2 \in \mathcal{I}_i^{\texttt{sql\_db}}$, such that $I_1 \sqsubseteq_i I_2$. Then, every `sel_stm` included in the select statement that defines `R` holds:

- If $str(\texttt{sel\_stm}) < i$, then $[\![\texttt{sel\_stm}]\!]^{I_1} = [\![\texttt{sel\_stm}]\!]^{I_2}$.
- If $str(\texttt{sel\_stm}) = i$, then $[\![\texttt{sel\_stm}]\!]^{I_1} \subseteq [\![\texttt{sel\_stm}]\!]^{I_2}$.

Proof. The proof is inductive on the structure of `sel_stm`. Here, we only show the most critical case. The others are similar.

$[\![\texttt{sel\_stm EXCEPT R}']\!]^{I_1} = [\![\texttt{sel\_stm}]\!]^{I_1} \setminus I_1(\texttt{R}')$. According to the definition of stratification, $str(\texttt{R}') < i$, because we are assuming that `sel_stm EXCEPT R'` occurs in the definition of `R` and $str(\texttt{R}) \leq i$. Hence $I_1(\texttt{R}') = I_2(\texttt{R}')$. Now, if $str(\texttt{sel\_stm EXCEPT R}') \leq i$, then $[\![\texttt{sel\_stm}]\!]^{I_1} \subseteq [\![\texttt{sel\_stm}]\!]^{I_2}$, applying the induction hypothesis. Therefore $[\![\texttt{sel\_stm EXCEPT R}']\!]^{I_1} \subseteq [\![\texttt{sel\_stm EXCEPT R}']\!]^{I_2}$, with equality for the case $str(\texttt{sel\_stm EXCEPT R}') < i$. □

The following lemma underlies the proof of the continuity of the operator whose fixpoint provides the semantics of a database (it can be proved by induction on the structure of `sel_stm`).

Lemma 3. Let $i \geq 1$, $\texttt{R} \in \texttt{RN}$, with $str(\texttt{R}) \leq i$, and $\{I_n\}_{n\geq 0}$ be a chain in $\mathcal{I}_i^{\texttt{sql\_db}}$. Then, for every `sel_stm` included in the definitions of `R`, if $\hat{I} = \bigsqcup_{n\geq 0} I_n$, there exists $n \geq 0$, such that $[\![\texttt{sel\_stm}]\!]^{\hat{I}} = [\![\texttt{sel\_stm}]\!]^{I_n}$.

Next, for every i, a continuous operator T_i over the set $\mathcal{I}_i^{\texttt{sql\_db}}$ of interpretations of stratum i is defined. Analogously to the theoretical foundations that support Datalog [16], we choose the least fixpoint of T_i, as the interpretation over i that will give meaning to the relations of stratum i. In accordance with the Knaster-Tarski theorem, this fixpoint can be obtained as the least upper bound of the chain of interpretations resulting by successively applying this operator to a minimal interpretation.

Definition 6. Let $1 \leq i \leq numstr$. The *operator* $T_i : \mathcal{I}_i^{\texttt{sql\_db}} \longrightarrow \mathcal{I}_i^{\texttt{sql\_db}}$ transforms interpretations over i as follows. For every $I \in \mathcal{I}_i^{\texttt{sql\_db}}$, $\texttt{R} \in \texttt{RN}$:

- $T_i(I)(\texttt{R}) = I(\texttt{R})$, if $str(\texttt{R}) < i$.
- $T_i(I)(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^I$, if $str(\texttt{R}) = i$ and `R sch := sel_stm` is the definition of `R` in `sql_db`.
- $T_i(I)(\texttt{R}) = \emptyset$, if $str(\texttt{R}) > i$.

This operator is proved to be monotone (it is a consequence of Lemma 2) and continuous for every i.

Lemma 4. [Monotonicity of T_i] Let $i \geq 1$ and $I_1, I_2 \in \mathcal{I}_i^{\texttt{sql\_db}}$, such that $I_1 \sqsubseteq_i I_2$. Then, $T_i(I_1) \sqsubseteq_i T_i(I_2)$.

Proposition 1. [Continuity of T_i] Let $i \geq 1$ and $\{I_n\}_{n\geq 0}$ be a chain of interpretations in $\mathcal{I}_i^{\texttt{sql\_db}}$ ($I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$). Then, $T_i(\bigsqcup_{n\geq 0} I_n) =_i \bigsqcup_{n\geq 0} T_i(I_n)$.

Proof. The proof of $\bigsqcup_{n\geq 0} T_i(I_n) \sqsubseteq_i T_i(\bigsqcup_{n\geq 0} I_n)$ is a direct consequence of the monotonicity of T_i (Lemma 4). Let us prove $T_i(\bigsqcup_{n\geq 0} I_n) \sqsubseteq_i \bigsqcup_{n\geq 0} T_i(I_n)$:

- If $str(\texttt{R}) < i$, then $T_i(\bigsqcup_{n\geq 0} I_n)(\texttt{R}) = \bigsqcup_{n\geq 0} I_n(\texttt{R})$, by the definition of T_i. Now, for every $n \geq 0$, $I_n(\texttt{R}) = T_i(I_n)(\texttt{R})$, also by definition of T_i. Therefore, $(T_i(\bigsqcup_{n\geq 0} I_n))(\texttt{R}) = (\bigsqcup_{n\geq 0} T_i(I_n))(\texttt{R})$.
- If $str(\texttt{R}) = i$, then $T_i(\bigsqcup_{n\geq 0} I_n)(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{\bigsqcup_{n\geq 0} I_n}$, by definition of T_i. And, in accordance with Lemma 3, for some $n \geq 0$: $[\![\texttt{sel\_stm}]\!]^{\bigsqcup_{n\geq 0} I_n} \subseteq [\![\texttt{sel\_stm}]\!]^{I_n}$. Now $[\![\texttt{sel\_stm}]\!]^{I_n} = T_i(I_n)(\texttt{R})$, by definition of T_i, and obviously $T_i(I_n)(\texttt{R}) \subseteq \bigcup_{n\geq 0} T_i(I_n)(\texttt{R})$, but $\bigcup_{n\geq 0} T_i(I_n)(\texttt{R}) = (\bigsqcup_{n\geq 0} T_i(I_n))(\texttt{R})$, by Lemma 1. Hence, we conclude $T_i(\bigsqcup_{n\geq 0} I_n)(\texttt{R}) \subseteq (\bigsqcup_{n\geq 0} T_i(I_n))(\texttt{R})$. □

Next, the expected result corresponding to the existence of least fixpoint stratum by stratum is shown.

Lemma 5. The operator T_1 has a least fixpoint, which is $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$, where $\emptyset : \texttt{RN} \to \mathcal{P}(\mathcal{T})$ is the interpretation such that $\emptyset(\texttt{R}) = \emptyset$ for every $\texttt{R} \in \texttt{RN}$.

Proof. By the Knaster-Tarski fixpoint theorem [15], using Proposition 1. □

We will denote $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$ by fix_1, i.e., fix_1 represents the least fixpoint at stratum 1. Using Example 1, Figure 3 shows the tuples corresponding to the successive applications of the operator T_1 until $fix_1(\texttt{travel})$ is obtained.

Consider now the sequence $\{T_2^n(fix_1)\}_{n\geq 0}$ of interpretations in $(\mathcal{I}_2^{\texttt{sql\_db}}, \sqsubseteq_2)$ greater than fix_1. Using the definition of T_i and the fact that $fix_1(\texttt{R}) = \emptyset$ for every R such that $str(\texttt{R}) \geq 2$, it is easy to prove, by induction on $n \geq 0$, that this sequence is a chain:

$$fix_1 \sqsubseteq_2 T_2(fix_1) \sqsubseteq_2 T_2(T_2(fix_1)) \sqsubseteq_2 \ldots, \sqsubseteq_2 T_2^n(fix_1), \ldots$$

$T_1^n(\emptyset)(\texttt{travel})$	Set of tuples
$T_1^1(\emptyset)(\texttt{travel})$	{(lon,ny,7.0), (par,lon,2.0), (par,ny,8.0), (mad,par,1.5), (lis,mad,1.0)}
$T_1^2(\emptyset)(\texttt{travel})$	{(lis,par,2.5), (par,ny,9.0), (mad,ny,9.5), (mad,lon,3.5)}
$T_1^3(\emptyset)(\texttt{travel})$	{(lis,ny,10.5), (lis,lon,4.5), (mad,ny,10.5)}
$T_1^4(\emptyset)(\texttt{travel})$	{(lis,lon,4.5), (mad,ny,10.5), (lis,ny,11.5)}

Fig. 3. Obtaining $fix_1(\texttt{travel})$

As before, in accordance with Proposition 1, $\{T_2^n(fix_1)\}_{n\geq 0}$ has a least upper bound, $\bigsqcup_{n\geq 0} T_2^n(fix_1)$, in $(\mathcal{I}_2^{\texttt{sql\_db}}, \sqsubseteq_2)$ that is the least fixpoint of T_2 containing fix_1. We denote this interpretation by fix_2.

By proceeding successively, for every i, $1 < i \leq numstr$, a chain:

$$fix_{i-1} \sqsubseteq_i T_i(fix_{i-1}) \sqsubseteq_i T_i(T_i(fix_{i-1})) \sqsubseteq_i \ldots \sqsubseteq_i T_i^n(fix_{i-1}) \ldots$$

can be defined, and a fixpoint of T_i, $fix_i = \bigsqcup_{n\geq 0} T_i^n(fix_{i-1})$, can be found.

Theorem 1. There is a fixpoint interpretation $fix : \texttt{RN} \longrightarrow \mathcal{P}(\mathcal{T})$, such that for every $\texttt{R} \in \texttt{RN}$, if `sel_stm` is the definition of R, then $fix(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix}$.

Proof. The interpretation fix we are looking for is fix_{numstr}, the least fixpoint of the operator T_{numstr}, applied to $fix_{numstr-1}$. As it has been pointed out, this fixpoint exists and verifies $fix_1 \sqsubseteq_{numstr} fix_2 \sqsubseteq_{numstr} \ldots \sqsubseteq_{numstr} fix_{numstr}$. Moreover, if $str(\texttt{R}) = i$, $1 \leq i \leq numstr$, and it is defined by the statement `sel_stm`, then $fix(\texttt{R}) = fix_i(\texttt{R}) = T_i(fix_i)(\texttt{R})$, because fix_i is the fixpoint of T_i. Now, $T_i(fix_i)(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix_i}$, by definition of T_i. We can conclude $fix(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix}$, trivially if $i = numstr$, or using Lemma 2, if $i < numstr$, because $fix_i \sqsubseteq_{numstr} fix$. □

Therefore, the interpretation fix defines the fixpoint semantics of `sql_db`. This semantics is the support of the database system prototype we have implemented, which is described next.

4 Implementing R-SQL

In this section we introduce a working proof-of-concept implementation for the R-SQL language that takes a set of relation definitions and outputs their meanings if a stratification can be found. More specifically, taking a stratifiable database definition in the R-SQL syntax as input, we get a SQL database (for a concrete SQL database system), that corresponds to the fixpoint semantics of the input database. If the database is not stratifiable, the system throws an error message and stops.

4.1 An Algorithm to Compute the Database Fixpoint

Let `sql_db` be the definition of a R-SQL database. In order to create the corresponding SQL database we have to generate the appropriate SQL sentences for building the expected relations, that will be eventually processed by a RDBMS. The algorithm takes `sql_db` as input, i.e., a sequence of relation definitions, $\texttt{R}_1\texttt{sch}_1$:= $\texttt{sel\_stm}_1$; ...; $\texttt{R}_n\texttt{sch}_n$:= $\texttt{sel\_stm}_n$. The computation builds the dependency graph for `sql_db`, as shown in Section 3.1, then calculates a stratification for it obtaining the sets $\mathrm{R}_1, \ldots, \mathrm{R}_{numstr}$, where R_i is the set of relations of stratum i, and finally the fixpoint is computed with the following algorithm:

```
(1)   str:=1
(2)   while str ≤ numstr do
(3)       for each Rᵢ ∈ R_str do CREATE TABLE Rᵢ schᵢ
(4)       change := true
(5)       while change do
(6)           size := rel_size_sum(R_str)
(7)           for each Rᵢ ∈ R_str do
(8)               INSERT INTO Rᵢ SELECT * FROM sel_stmᵢ
(9)                   EXCEPT SELECT * FROM Rᵢ;
(10)          change = (size ≠ rel_size_sum(R_str))
(11)      end while
(12)      str:=str+1
(13)  end while
```

This algorithm generates for each $\mathtt{R}_i$ of $\mathtt{sql\_db}$ a SQL table with the elements of $fix(\mathtt{R}_i)$. Each iteration of the external *while* at line 2 corresponds to a stratum *str*, and builds the tables of the relations of this stratum, by calculating fix_{str}. To do that, first of all an empty table with the corresponding attributes is created for every relation in the stratum *str* (line 3). Then, the iteration n of the innermost *while* at line 5 computes $T^n_{str}(fix_{str-1})$, as we will explain. For every relation $\mathtt{R}_i$ of *str*, it submits the INSERT statement at line 8. The sentence SELECT * FROM $\mathtt{sel\_stm}_i$ selects all tuples as defined by the relation $\mathtt{R}_i$ (notice that $\mathtt{sel\_stm}_i$ is a valid SQL statement). Assuming that the current database instance coincides with the value of the interpretation $T^{n-1}_{str}(fix_{str-1})$, then in accordance with Definition 5, the set of tuples that satisfy that SQL statement coincides with $[\![\mathtt{sel\_stm}_i]\!]^{T^{n-1}_{str}(fix_{str-1})}$. And this is $T^n_{str}(fix_{str-1})(\mathtt{R}_i)$, by Definition 6. The tuples already present in the table are excluded to avoid repetitions (with the EXCEPT clause at line 9). In this way, $T^n_{str}(fix_{str-1})(\mathtt{R}_i)$ is obtained for every $\mathtt{R}_i$ of stratum *str*. The expression $rel\_size\_sum(\mathrm{R}_{str})$ at line 10 is equal to $\sum_{\mathtt{R}\in R_{str}} |\mathtt{R}|$, where $|\mathtt{R}|$ is the current number of tuples of the table corresponding to R. Therefore, the variable *change* controls changes on the table sizes in order to stop the process, since *change* = *false* means that $T^n_{str}(fix_{str-1}) = T^{n-1}_{str}(fix_{str-1})$, so that fix_{str} has been reached. Then, the last iteration of the external *while* calculates fix_{numstr}, the fixpoint of $\mathtt{sql\_db}$.

4.2 A Concrete Implementation

The concrete implementation of this algorithm can be done in a number of ways. We have developed a Prolog program that processes the R-SQL input file, builds the dependency graph and the stratification (if exists), and finally produces a Python module with the code of the previous section. In fact, the external *while* at line 2 is expanded according to the number of strata, writing explicitly the corresponding code for each stratum. The *for* loop at line 7 is also expanded as we will see in Example 4. We have chosen Python as the host language mainly because is multiplatform and it provides easy connections with different database

systems such as PostgreSQL, MySQL, or even via ODBC, which allows connectivity to almost any RDBMS. The additional features required for the host language are basic: Loops, assignment and basic arithmetic.

Example 4. Below, we show the result of executing our proposed algorithm for the `sql_db` of Example 1. The system assigns stratum 1 to `flight, reachable, travel, madAirport`, and stratum 2 to `avoidMad`. Next, we detail some parts of the code generated stratum by stratum. Firstly, for stratum 1, we have:

while change **do**
size := rel_size_sum(R_{str})
```
INSERT INTO flight SELECT 'lis','mad',1 UNION SELECT 'mad','par',1
UNION SELECT 'par','lon',2 UNION SELECT 'lon','ny',7
UNION SELECT 'par','ny',8 EXCEPT SELECT * FROM flight;

INSERT INTO reachable SELECT flight.frm, flight.to
FROM flight UNION SELECT reachable.frm, flight.to
WHERE reachable.to = flight.frm
EXCEPT SELECT * FROM reachable;

INSERT INTO travel SELECT * FROM flight UNION
SELECT flight.frm, travel.to, flight.time+travel.time
FROM flight, travel WHERE flight.to = travel.frm
EXCEPT SELECT * FROM travel;

INSERT INTO madAirport SELECT travel.frm,travel.to
FROM travel EXCEPT SELECT * FROM madAirport;
```
change = (size $\neq$ rel_size_sum(R_{str}))
end while

In the first iteration of this loop, we obtain all the tuples for `flight` and `madAirport` relations. But the recursive definitions for `reachable` and `travel` need more iterations. As mentioned before, those iterations correspond to the successive applications of T_1. The tuples added for `travel` at each iteration are shown in Figure 3 (Section 3.2). After five iterations, the loop stops and the first stratum is completed. In the second stratum we consider the `avoidMad` relation:

```
INSERT INTO avoidMad SELECT travel.frm,travel.to FROM travel
EXCEPT SELECT * FROM madAirport EXCEPT SELECT * FROM avoidMad;
```

This second loop ends after two iterations. This completes fix_2 for our `sql_db`, i.e., it obtains the semantics of the working example database.

4.3 Integrating R-SQL into a RDBMS

Our proposal establishes the core for introducing a novel approach for recursion in SQL. The current implementation of R-SQL has been conceived as a proof-of-concept of the theoretical foundations of the language. As we have stated, this leads to compute the semantics of the whole database from scratch. Nevertheless, the main goal of the proposal is not to introduce a new database language, but

to allow less-restricted recursive relation definitions in existing SQL engines. In that sense, our proposal can be understood as the foundation of an existing SQL RDBMS that supports extended forms of recursion, allowing users to define recursive relations as regular views using the R-SQL techniques, developed in this work. Once an R-SQL database definition has been processed, the tables obtained can be stored as a database instance in a concrete RDBMS. On the one hand, the user can formulate queries that will be solved using those tables (without performing any further fixpoint computation). On the other hand, as we pointed out before, the user can define new recursive relations using views. Those views can be readily used in conjunction with other regular views, and they can be either computed on demand or can be materialized.

In order to compute the answer of new recursive relations, the current (relation) instance can be considered as a stratified R-SQL database. It is correct to assign higher strata to the new relations, because none of the existing relations depend on the new ones, and a relation definition does not introduce dependencies between the relations that appear in its select statement. Then, their tuples can be obtained by executing the algorithm in Section 4.1 to compute the fixpoint of their corresponding strata, therefore saving recalculating the previous ones. Moreover, it is straightforward to modify the algorithm to get a lazy evaluation of such relations, performing iterations only when new values are demanded. To seamlessly integrate this into a RDBMS, we can profit from the fourth-generation languages (e.g., SQL PL in IBM DB2 and PL/SQL in Oracle).

5 Conclusions

In this paper, we have introduced the R-SQL language as a new approach for incorporating recursion in SQL. This is not a trivial task, and it was not addressed in the initial proposals of SQL. It was firstly introduced in the 1999 standard, allowing only a limited form of recursion, namely linear recursion, which does not allow neither multiple recursive calls nor mutually recursive definitions. The difficulties increase when recursion is combined with negation.

We have developed a theoretical framework and a suitable implementation for R-SQL, inspired on the stratification techniques and fixpoint computations used for instance in Datalog. The stratification mechanism implies to impose some syntactic conditions on the database definitions, that guarantee that the fixpoint for such a database can be computed in a finite number of steps. This condition is less restrictive than the linearity conditions required by the standard SQL. This means that our approach is more expressive than the one adopted in SQL; in addition our language is supported by a solid computational semantics. We have presented a proof-of-concept implementation of the R-SQL database definition language based on this semantics. This implementation produces as output a set of standard SQL statements embedded in a Python program that builds the relational tables corresponding to the fixpoint of the input database definition. This implementation has been tested with PostgreSQL, but the architecture can

be easily ported to any RDBMS. The system is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQL`.

As already suggested, our approach can be integrated into a state-of-the-art RDBMS. This can be dealt by resorting to database function definitions, which allow cursor-returning functions. In addition for this integration to be practical, performance improvements play a key role as, e.g., indexing of temporary relations during fixpoint computations and identifying tuple seeds in relation definitions that do not need to be recomputed.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. E. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13(6):377–390, June 1970.
3. C. J. Date. *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA, 2009.
4. S. J. Finkelstein, N. Mattos, I. S. Mumick, and H. Pirahesh. Expressing recursive queries in SQL. Technical report, ISO, 1996.
5. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.
6. M. A. W. Houtsma and P. M. G. Apers. Algebraic optimization of recursive queries. *Data Knowl. Eng.*, 7:299–325, 1991.
7. ISO/IEC. SQL:2008 ISO/IEC 9075(1-4,9-11,13,14):2008 Standard, 2008.
8. O. Kaser, C. R. Ramakrishnan, and S. Pawagi. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.*, 2(1-4):151–164, Mar. 1993.
9. R. A. Kowalski. Logic for data description. In *Logic and Data Bases*, pages 77–103, 1977.
10. I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. *SIGMOD Rec.*, 23:103–114, May 1994.
11. S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation. In *FLOPS'08*, volume 4989 of *LNCS*, pages 289–304. Springer-Verlag, 2008.
12. K. Ramamohanarao and J. Harland. An introduction to deductive database languages and systems. *The VLDB Journal*, 3(2):107–122, 1994.
13. R. Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling (Intervale)*, pages 191–233, 1982.
14. J. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Kaufmann, Los Altos, CA, 1988.
15. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
16. J. Ullman. *Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1995.
17. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers Inc., 1997.