R-SQL: An SQL Database System with Extended Recursion*

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@sip.ucm.es, fernan@sip.ucm.es, jaime@sip.ucm.es

Abstract:

The relational database language SQL:1999 standard supports recursion, but this approach is limited to the linear case. Moreover, mutual recursion is not supported, and negation cannot be combined with recursion. The language R-SQL was designed to overcome these limitations. In addition, it improves termination properties in recursive definitions. We have developed an implementation of R-SQL, that can be integrated into existing commercial SQL database systems, extending such systems with the aforementioned benefits of R-SQL. In this paper we describe a concrete instance, implemented using Python as host language and PostgreSQL as database system.

Keywords: Databases, SQL, Recursion, Fixpoint Semantics

1 Introduction

Recursion is a powerful tool nowadays included in almost all programming systems. However, for current implementations of the declarative programming language SQL, this tool is heavily compromised or even not supported at all (MySQL, MS Access, ...) Those systems including recursion suffer from several drawbacks. Linearity is required, so that relation definitions with calls to more than one recursive relation are not allowed. Mutual recursion, and query solving involving an EXCEPT clause are not supported. 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. Duplicate discarding is not supported and, so, queries that are actually terminating are not detected as such.

Starburst [MP94] was the first non-commercial RDBMS to implement recursion whereas IBM DB2 was the first commercial one. ANSI/ISO Standard SQL:1999 included for the first time recursion in SQL. Today, we can find recursion in several systems: IBM DB2, Oracle, MS SQL Server, HyperSQL and others with the aforementioned limitations. In [ANSS13] we proposed 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 applied ideas from the deductive database field, such as stratified negation, based on the definition of a dependency graph between the relations involved

* 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.

```
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 and C for constants belonging to a valid SQL type.

Figure 1: A Grammar for the R-SQL Language

in the database [Ull95]. We developed a formal framework following the original relational data model [Cod70], so avoiding both duplicates and nulls (as encouraged by [Dat09]). We used a stratified fixpoint semantics and introduced a proof-of-concept implementation.

In this work, we present the R-SQL database system, a prototype implementing such formal framework, in further detail. Also, we recall the syntax, and the meaning of database definitions. The system can be downloaded from https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQL.

Related academic approaches include DLV^{DB} [TLLP08], LDL++ [AOT+03] (now abandoned and replaced by DeAL, which does not refer to SQL queries up to now), and DES [SP13]. The first one, resulting of a spin-off at Calabria University, is the closer to our work as it produces SQL code to be executed in the external database with a semi-naïve strategy, but lacks formal support for its proposal, and it does not describe non-linear recursion. Last two ones also allow connecting to external databases, but processing of recursive SQL queries are in-memory.

2 Introducing R-SQL

In this section, we present an overview of the language R-SQL, which 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.

2.1 The Definition Language 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 language R-SQL overcomes some limitations present in current RDBMS's following

SQL:1999. These languages use NOT IN and EXCEPT clauses to deal with negation, and WITH RECURSIVE to engage recursion. As it is pointed out in [GUW09], SQL:1999 does not allow an arbitrary collection of mutually recursive relations to be written in the WITH RECURSIVE clause.

A bundle of R-SQL database examples can be found with the system distribution. Next, we present some of them, to show the expressiveness of the definition language.

Mutual Recursion Although any mutual recursion can be converted to direct recursion by inlining [KRP93], our proposal allows to explicitly define mutual recursive relations, which is an advantage in terms of program readability and maintenance. For instance, the following R-SQL database defines the relations `even` and `odd`, as the classical specification of even and odd numbers up to a bound (100 in the example):

```
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;
```

Nonlinear Recursion The standard SQL restricts the number of allowed recursive calls to be only one. Here we show how to specify Fibonacci numbers in R-SQL[1]:

```
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;
```

Duplicates and Termination Non termination is another problem that arises associated to recursion when coupled with duplicates. For instance, the following standard SQL query (that considers a finite relation `t`) makes current systems 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, as DB2):

```
WITH RECURSIVE v(a) AS  SELECT * FROM t UNION ALL SELECT * FROM v
  SELECT * FROM v
```

Nevertheless, the fixpoint computation for the corresponding R-SQL relation:

```
v(a float) := SELECT * FROM t UNION SELECT * FROM v;
```

guarantees termination because duplicates are discarded[2] and `v` does not grow unbounded. The very same termination problem also happens in current RDBMS's with the basic transitive closure over graphs including cycles, but not in R-SQL which ensures termination for finite graphs.

[1] 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.

[2] Note that UNION does not require ALL, as current RDBMS's do.

2.2 The meaning of an R-SQL database definition

In [ANSS13] we formalized an operational semantics for the language R-SQL based on stratified negation and fixpoint theory, here we summarize the main ideas.

Stratification is based on the definition of a *dependency graph* $DG_{\texttt{db}}$ for an R-SQL database db that is a directed graph whose nodes are the relation names defined in db, and the edges, that can be *negatively labelled*, are determined as follows. A relation definition of the form R sch := sel_stm in db 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.

If there are n relations defined in db, and we denote by RN the set of the relation names defined in db, a *stratification* of db is a mapping $str : \texttt{RN} \to \{1,\ldots,n\}$, such that for every two relations $\texttt{R}_1, \texttt{R}_2 \in \texttt{RN}$ it satisfies:

- $str(\texttt{R}_1) \leq str(\texttt{R}_2)$, if there is a path from $\texttt{R}_1$ to $\texttt{R}_2$ in $DG_{\texttt{db}}$,
- $str(\texttt{R}_1) < str(\texttt{R}_2)$ if there is a path from $\texttt{R}_1$ to $\texttt{R}_2$ in $DG_{\texttt{db}}$ with at least one negatively labelled edge.

An R-SQL database db is *stratifiable* if there exists a stratification for it. We denote by *numStr* the maximum stratum of the elements of RN. And, for every i, $1 \leq i \leq numStr$, $\texttt{RN}_i$ denotes the set of relation names $\texttt{R} \in \texttt{RN}$, such that $str(\texttt{R}) = i$.

Intuitively, a relation name preceded by an EXCEPT operator 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 [Ull95].

We say that an interpretation I is the relationship between every relation R name and its instance $I(\texttt{R})$. Interpretations are classified by strata; an interpretation belonging to a stratum i gives meaning to the relations of strata less or equal to i. The meaning of every sel_stm w.r.t. an interpretation I can be understood as the set of tuples (in the current instance represented by I) associated to the corresponding equivalent *RA*-expression, denoted by $[\texttt{sel\_stm}]^I$. This *RA*-expression is defined as follows: [3]

- $[\text{SELECT } \texttt{exp}_1,\ldots,\texttt{exp}_k \text{ FROM } \texttt{R}_1,\ldots,\texttt{R}_m \text{ WHERE } \texttt{wcond}]^I = \pi_{\texttt{exp}_1,\ldots,\texttt{exp}_k}(\sigma_{\texttt{wcond}}(I(\texttt{R}_1) \times \ldots \times I(\texttt{R}_m)))$
- $[\texttt{sel\_stm}_1 \text{ UNION } \texttt{sel\_stm}_2]^I = [\texttt{sel\_stm}_1]^I \cup [\texttt{sel\_stm}_2]^I$
- $[\texttt{sel\_stm} \text{ EXCEPT } \texttt{R}]^I = [\texttt{sel\_stm}]^I - I(\texttt{R})$

Example 1 Consider the definitions of the relations odd and even of Section 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:

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

$[\texttt{SELECT even.x} + 1 \texttt{ FROM even WHERE even.x} < 100]^I =$
$\{(\texttt{even.x}+1)[a/\texttt{even.x}] \mid (a) \in I(\texttt{even}), (\texttt{even.x} < 100)[a/\texttt{even.x}] \text{ is satisfied}\} = \{(1),(3)\}.$

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

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:
$\hat{I}(\texttt{even}) = [\texttt{SELECT 0 UNION SELECT odd.x} + 1 \texttt{ FROM odd WHERE odd.x} < 100]^{\hat{I}}.$
$\hat{I}(\texttt{odd}) = [\texttt{SELECT even.x} + 1 \texttt{ FROM even WHERE even.x} < 100]^{\hat{I}}.$

So, to give meaning to a database definition, we are interested in an interpretation, called fix, such that for every $\texttt{R} \in \texttt{RN}$, if sel_stm is the definition of R, then $fix(\texttt{R}) = [\texttt{sel\_stm}]^{fix}$. In the previous example fix will be $\hat{I}$. Since R can occur inside its definition, for every stratum i, the appropriate interpretation fix_i that gives the complete meaning to each relation of stratum i is the least fixpoint of a continuous operator. These fixpoint interpretations are sequentially constructed from stratum 1 to $numStr$. fix represents the fixpoint of the last stratum and provides the semantics for the whole database.

For every i, $1 \leq i \leq numStr$, we define the continuous operator T_i that transforms interpretations belonging to a stratum i as follows:

- $T_i(I)(\texttt{R}) = I(\texttt{R})$, 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 db.
- $T_i(I)(\texttt{R}) = \emptyset$, if $str(\texttt{R}) > i$.

The operator T_1 has a least fixpoint, which is $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$, where $\emptyset(\texttt{R}) = \emptyset$ for every $\texttt{R} \in \texttt{RN}$. 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.

Consider now the sequence $\{T_2^n(fix_1)\}_{n\geq 0}$ of interpretations of stratum 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:

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

$\{T_2^n(fix_1)\}_{n\geq 0}$ is a chain that has as least upper bound, $\bigsqcup_{n\geq 0} T_2^n(fix_1)$, which is the least fixpoint of T_2 containing fix_1. We denote this interpretation by fix_2. By proceeding successively in the same way it is possible to find fix_{numStr}. In [ANSS13] we have proved that fix_{numStr} is the interpretation fix we are looking for, that associates the set of tuples denoted by its definition to every relation of the database .

3 The R-SQL System

In this section we introduce the R-SQL system, based on the fixpoint construction of the previous section. Once the system is loaded in SWI Prolog (see the `readme` file of the distribution for details) a database definition `dbDef.sql` can be processed with the command `?- process(dbDef).` First, the system parses the input R-SQL database definition, then it builds the dependency graph and the stratification if exists (it aborts, otherwise); finally, it pro-

duces a Python script that will create the SQL database in a RDBMS. After this process, the user can connect to PostgreSQL in order to query or modify the database. Although we are referring to PostgreSQL in this paper, the implementation can be straightforwardly applied to other systems. In fact, the distribution also includes a MySQL implementation. The main difference when adapting to a particular system is the concrete SQL dialect of such a system. For instance, we use EXCEPT in PostgreSQL and NOT IN in MySQL, as it lacks a set difference operator.

Next we present a database for flights to illustrate the process and also will be the working example for the rest of the section. 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 (Lisbon, Madrid, Paris, London, New York) will be represented with constants (`lis`, `mad`, `par`, `lon`, `ny`, resp.). The relation `reachable` consists of all the possible trips between the cities of the database, maybe concatenating more than one flight. The relation `travel` is analogous but also gives time information about alternative trips.

```
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;

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;

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 the 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:1999 as it is shown in [FMMP96]. The dependency graph of this database is depicted in Figure 2, where negatively labelled edges are annotated with $\neg$. Then, the system assigns stratum 1 to `flight`, `reachable`, `travel`, `madAirport`, and stratum 2 to `avoidMad`.

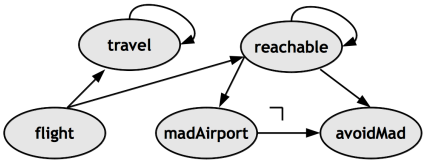

Figure 2: $DG_{\texttt{db}}$ of the working example.

3.1 The Generic Algorithm to Compute the Database Fixpoint

Let db be a stratifiable database definition $\texttt{R}_1$ $\texttt{sch}_1 := \texttt{sel\_stm}_1$; ...; $\texttt{R}_n$ $\texttt{sch}_n := \texttt{sel\_stm}_n$. In order to create the corresponding SQL database we have to generate the appropriate SQL statements for building the expected relations (corresponding to the fixpoint of db) when processed by a RDBMS. The algorithm in Figure 3 produces such SQL statements. This algorithm com-

```
1  str:=1
2  while str ≤ numStr do
3      for all R ∈ RN_str do
4          CREATE TABLE R sch;
5      end for
6      change := true
7      while change do
8          size := rel_size_sum(RN_str)
9          for all R ∈ RN_str do
10             INSERT INTO R SELECT * FROM sel_stm_R
11                                 EXCEPT SELECT * FROM R;
12         end for
13         change = (size ≠ rel_size_sum(RN_str))
14     end while
15     str:=str+1
16 end while
```

Figure 3: Algorithm to compute the database fixpoint

putes $fix(\texttt{R})$ for each relation R of the database and stores the corresponding tuples in a SQL table with the same name R. Relations are computed by strata, from stratum 1 to the last one *numStr*, by the *while* loop at lines 2-16. Each iteration obtains fix_{str} for the current stratum *str*. Within this loop, first of all, the *for* at lines 3-4 creates an empty table with the corresponding attributes for each relation R in stratum *str*. Then the *while* loop at lines 7-14 at the *n*-iteration computes $T^n_{str}(fix_{str-1})$. This is done by performing repeatedly the INSERT statements at lines 10-11 for each relation R in the current stratum, that add the tuples resulting from the statement $\texttt{sel\_stm}_\texttt{R}$ that defines such relation. Assuming that the current database instance coincides with the value of the interpretation $T^{n-1}_{str}(fix_{str-1})$, the set of tuples that satisfy that SQL statement

coincides with $[\texttt{sel\_stm}]^{T_{str}^{n-1}(fix_{str-1})}$. And this is $T_{str}^{n}(fix_{str-1})(\texttt{R}_i)$. The tuples already present in the table are excluded to avoid duplicates by means of the EXCEPT clause at line 11. In this way, $T_{str}^{n}(fix_{str-1})(\texttt{R})$ is obtained for every R of stratum *str*. Finally, the expression $rel\_size\_sum(\texttt{RN}_{str})$ at lines 8 and 13 denotes $\sum_{\texttt{R}\in\texttt{RN}_{str}}|\texttt{R}|$, where $|\texttt{R}|$ stands for the 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 when the fixpoint for a stratum is reached.

3.2 Python Code Generation

The concrete implementation of this algorithm can be done in a number of ways. 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.

Below, we show the result of executing our proposed algorithm for the working example of flights. We detail some parts of the code generated stratum by stratum. For stratum 1, we have:

```
change := True
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 != 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 4. 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 `db`, i.e., it obtains the semantics of the working example database.

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

Figure 4: Obtaining $fix_1(\texttt{travel})$

The values for `flight`, `madAirport` and `avoidMad` tables are illustrated in the graph in Figure 5. Direct flights will be represented in blue color and labeled with their corresponding time. Paths for `madAirport` relation will be represented in red color and path for `avoidMad` relation will be represented in black color. The whole database resulting tables are included in Appendix A.

3.3 The System in Action and Future Improvements

Once an R-SQL database definition has been processed, the tables obtained are stored as a database instance in PostgreSQL. Then, the user can formulate queries that will be solved using those tables (without performing any further fixpoint computation).

Notice that the modification of a relation of the database from the PostgreSQL system can cause inconsistencies since the tables are not recomputed. For instance, after processing the database for flights, if the user adds or deletes a tuple for the relation `flight`, then the relation `travel` will become inconsistent according to its R-SQL definition. But this is the common RDBMS behavior, when dealing with materialized views. R-SQL has been designed to compute the meaning of a database definition and then to query this database. A future direction in order to fully integrate R-SQL into a RDBMS is to have the possibility of restoring the consistence of the database (using triggers for instance), as well as to define additional (possibly recursive) views. This restoring involves the recomputation of the database fixpoint. But, using the dependency graph, it is easy to determine the subset of relations that must be calculated, instead of computing the whole fixpoint for the database. Moreover, those relations may not need to be

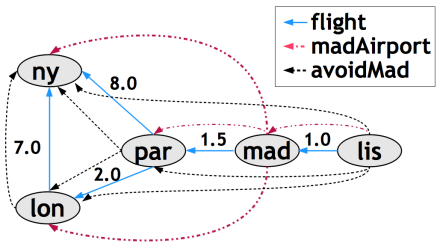

Figure 5: Graphical representation of resulting values of the working example.

recomputed from scratch. In addition, it is straightforward to modify the algorithm introduced in Section 3.1 to get a lazy evaluation of such relations, performing iterations only when new values are demanded. Efficiency can also be improved by indexing (e.g., tries [SW12] and BDD's [WACL05]) temporary relations during fixpoint computations, and identifying tuple seeds in relation definitions that do not need to be recomputed. 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). The current implementation of R-SQL must be understood as a prototype, which we plan to enhance as just noted and then compare it with other systems.

Bibliography

[ANSS13] G. Aranda, S. Nieva, F. Sáenz-Pérez, J. Sánchez-Hernández. Formalizing a Broader Recursion Coverage in SQL. In *PADL'13*. LNCS 7752. 2013. In Press.

[AOT+03] F. Arni, K. Ong, S. Tsur, H. Wang, C. Zaniolo. The Deductive Database System LDL++. *TPLP* 3(1):61–94, 2003.

[Cod70] E. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM* 13(6):377–390, June 1970.

[Dat09] C. J. Date. *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA, 2009.

[FMMP96] S. J. Finkelstein, N. Mattos, I. S. Mumick, H. Pirahesh. Expressing Recursive Queries in SQL. Technical report, ISO, 1996.

[GUW09] H. Garcia-Molina, J. D. Ullman, J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[KRP93] O. Kaser, C. R. Ramakrishnan, S. Pawagi. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.* 2(1-4):151–164, Mar. 1993.

[MP94] I. S. Mumick, H. Pirahesh. Implementation of magic-sets in a relational database system. *SIGMOD Rec.* 23:103–114, May 1994.

[SP13] F. Sáenz-Pérez. Towards Bridging the Expressiveness Gap Between Relational and Deductive Databases. In *PROLE2013*. 2013.

[SW12] T. Swift, D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP* 12(1-2):157–187, 2012.

[TLLP08] G. Terracina, N. Leone, V. Lio, C. Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP* 8(2):129–165, 2008.

[Ull95] J. Ullman. *Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1995.

[WACL05] J. Whaley, D. Avots, M. Carbin, M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In *In APLAS'05*. 2005.

flight

Origin	Destination	Time
lis	mad	1.0
mad	par	1.5
par	lon	2.0
lon	ny	7.0
par	ny	8.0

avoidMad

Origin	Destination
lis	ny
lis	lon
par	ny
lis	par
lon	ny
par	lon

travel

Origin	Destination	Time
lon	ny	7.0
par	lon	2.0
mad	par	1.5
par	ny	8.0
lis	mad	1.0
lis	par	2.5
par	ny	9.0
mad	ny	9.5
mad	lon	3.5
lis	ny	10.5
lis	lon	4.5
mad	ny	10.5
lis	ny	11.5

reachable

Origin	Destination
par	lon
lon	ny
par	ny
mad	par
lis	mad
mad	lon
mad	ny
lis	par
lis	lon
lis	ny

madAirport

Origin	Destination
lis	mad
mad	par
mad	lon
mad	ny

Figure 6: Resulting tables of the working example.

A Appendix: Results for the Working Example

This appendix includes the full Python code generated by the R-SQL system for the working example of the paper. The actual implementation of R-SQL introduces some improvements in the code. For example, the code generated for `flight` table differs from the one explained in the article, in order to save data from the cursor. In addition, the final values for all the tables are illustrated in Figure 6.

```
import psycopg2

try:
    conn = psycopg2.connect("dbname='test' user='postgres'
           host='localhost' password='123456'")

except:
    print "I am unable to connect to the database"

cursor=conn.cursor()


# Code generated for Stratum 1
cursor.execute("drop table if exists flight;")
cursor.execute("create table flight
              (ori varchar(10),des varchar(10),time float);")
cursor.execute("drop table if exists reachable;")
cursor.execute("create table reachable
              (ori varchar(10),des varchar(10));")
cursor.execute("drop table if exists travel;")
```

```
cursor.execute("create table travel
              (ori varchar(10),des varchar(10),time float);")
cursor.execute("drop table if exists madAirport;")
cursor.execute("create table madAirport
              (ori varchar(10),des varchar(10));")


ch = True
while ch:
     size1 = 0
     size2 = 0
     ch = False
     cursor.execute("select * from flight;")
     res=cursor.fetchall()
     size1= size1 + len(res)
     cursor.execute("insert into flight select 'lis','mad',1
                    except select * from flight;")
     cursor.execute("insert into flight select 'mad','par',1.5
                    except select * from flight;")
     cursor.execute("insert into flight select 'par','lon',2
                    except select * from flight;")
     cursor.execute("insert into flight select 'lon','ny',7
                    except  select * from flight;")
     cursor.execute("insert into flight select 'par','ny',8
                    except select * from flight;")

     cursor.execute("select * from flight;")
     res=cursor.fetchall()
     size2= size2 + len(res)

     cursor.execute("select * from reachable;")
     res=cursor.fetchall()
     size1= size1 + len(res)
     cursor.execute("insert into reachable
                   select flight.ori,flight.des from flight
                   union (select reachable.ori,flight.des
                   from reachable,flight
                   where reachable.des = flight.ori
                   except select * from reachable;")
     cursor.execute("select * from reachable;")
     res=cursor.fetchall()
     size2= size2 + len(res)

     cursor.execute("select * from travel;")
     res=cursor.fetchall()
     size1= size1 + len(res)
     cursor.execute("insert into travel
                   select * from flight
                   union select flight.ori,travel.des,
                   flight.time+travel.time
                   from flight,travel where flight.des = travel.ori
                   except select * from travel;")
     cursor.execute("select * from travel;")
     res=cursor.fetchall()
```

```
     size2= size2 + len(res)

     cursor.execute("select * from madAirport;")
     res=cursor.fetchall()
     size1= size1 + len(res)
     cursor.execute("insert into madAirport
                  select reachable.ori,reachable.des
                  from reachable
                  where reachable.ori='mad' or reachable.des='mad')
                  except select * from madAirport;")
     cursor.execute("select * from madAirport;")
     res=cursor.fetchall()
     size2= size2 + len(res)

     ch = (size1 != size2)

#  Code generated for Stratum 2
cursor.execute("drop table if exists avoidMad;")
cursor.execute("create table avoidMad
              (ori varchar(10),des varchar(10));")

ch = True
while ch:
     size1 = 0
     size2 = 0
     ch = False
     cursor.execute("select * from avoidMad;")
     res=cursor.fetchall()
     size1= size1 + len(res)
     cursor.execute("insert into avoidMad
                  select reachable.ori,reachable.des
                  from reachable)
                  except select * from madAirport where true
                  except select * from avoidMad;")
     cursor.execute("select * from avoidMad;")
     res=cursor.fetchall()
     size2= size2 + len(res)

     ch = (size1 != size2)
```