Declarative Debugging of Wrong and Missing Answers for SQL Views (extended version) *

Rafael Caballero[†], Yolanda García-Ruiz[†], and Fernando Sáenz-Pérez[‡]

[†]Departamento de Sistemas Informáticos y Computación
[‡]Dept. de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
{rafa,fernan}@sip.ucm.es and ygarciar@fdi.ucm.es

Abstract. This paper presents a debugging technique for diagnosing errors in SQL views. The debugger allows the user to specify the error type, indicating if there is either a missing answer (a tuple was expected but it is not in the result) or a wrong answer (the result contains an unexpected tuple). This information is employed for slicing the associated queries, keeping only those parts that might be the cause of the error. The validity of the results produced by sliced queries is easier to determine, thus facilitating the location of the error. Although based on the ideas of declarative debugging, the proposed technique does not use computation trees explicitly. Instead, the logical relations among the nodes of the trees are represented by logical clauses that also contain the information extracted from the specific questions provided by the user. The atoms in the body of the clauses correspond to questions that the user must answer in order to detect an incorrect relation. The resulting logic program is executed by selecting at each step the unsolved atom that yields the simplest question, repeating the process until an erroneous relation is detected. Soundness and completeness results are provided. The theoretical ideas have been implemented in a working prototype included in the Datalog system DES.

1 Introduction

SQL (Structured Query Language [16]) is a language employed by relational database management systems. In particular, the SQL select statement is used for querying data from databases. Realistic database applications often contain a large number of tables, and in many cases, queries become too complex to be coded by means of a single select statement. In these cases, SQL allows the user to define *views*. A SQL view can be considered as a virtual table, whose content is obtained executing its associated SQL select query. View queries can rely on previously defined views, as well as on database tables. Thus, complex

* Work partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502)

queries can be decomposed into sets of correlated views. As in other programming paradigms, views can have bugs. However, we cannot infer that a view is incorrectly defined when it computes an unexpected result, because it might be receiving erroneous input data from the other database tables or views. Given the high-abstraction level of SQL, usual techniques like trace debugging are difficult to apply. Some tools like [2, 11] allow the user to trace and analyze the stored SQL procedures and user defined functions, but they are of little help when debugging systems of correlated views. *Declarative Debugging*, also known as *algorithmic debugging*, is a technique applied successfully in (constraint) logic programming [14], functional programming [10], functional-logic programming [5], and in deductive database languages [3]. The technique can be described as a general debugging schema [9] which starts when an *initial error symptom* is detected by the user, which in our case corresponds to an unexpected result produced by a view. The debugger automatically builds a tree representing the erroneous computation. In SQL, each node in the tree contains information about both a relation, which is a table or a view, and its associated computed result. The root of the tree corresponds to the initial view. The children of a node correspond to the relations (tables or views) occurring in the definition of its associated query. After building the tree, it is *navigated* by the debugger, asking to the user about the validity of some nodes. When a node contains the expected result, it is marked as *valid*, and otherwise it is marked as *nonvalid*. The goal of the debugger is to locate a *buggy node*, which is a nonvalid node with valid children. It can be proved that each buggy node in the tree corresponds to either an erroneously defined view, or to a database table containing erroneous data. A debugger based on these ideas was presented in [4]. The main criticism that can leveled at this proposal is that it can be difficult for the user to check the validity of the results. Indeed, even very complex database queries usually are defined by a small number of views, but the results returned by these views can contain hundreds or thousands of tuples. The problem can be easily understood by considering the following example:

Example 1. The loyalty program of an academy awards an intensive course for students that satisfy the following constraints:
- The student has completed the basic level course (level = 0).
- The student has not completed an intensive course.
- To complete an intensive course, a student must either pass the *all in one* course, or the three initial level courses (levels 1, 2 and 3).

The database schema consists of three tables: *courses(id,level)* contains information about the standard courses, including their identifier and the course level; *registration(student,course,pass)* indicates that the *student* is in the *course*, with *pass* taking the value *true* if the course has been successfully completed; and the table *allInOneCourse(student,pass)* contains information about students registered in a special intensive course, with *pass* playing the same role as in *registration*. Figure 1 contains the views for selecting the award candidates. The first view is *standard*, which completes the information included in the table *Registration* with the course level. The view *basic* selects those *standard* students

```sql
create or replace view standard(student, level, pass) as
   select R.student, C.level, R.pass
   from courses C, registration R
   where C.id = R.course;

create or replace view basic(student) as
   select S.student
   from standard S
   where S.level = 0 and S.pass;

create or replace view intensive(student) as
   (select A.student from allInOneCourse A  where A.pass)
   union
   (select a1.student
    from standard A1, standard A2, standard A3
    where A1.student = A2.student and A2.student = A3.student
          and
          a1.level = 1  and a2.level = 2 and a3.level = 3);

create or replace view awards(student) as
  select student from  basic
  where student not in (select student from  intensive);
```

Fig. 1. Views for selecting award winner students

that have passed a basic level course (level 0). Next, view *intensive* defines as intensive students all the members of the *allInOneCourse* table together with the students that have completed the three initial levels. However, this view definition is erroneous: although it checks that the student is registered in the three levels we have forgotten to check that the courses have been completed (flag *pass*). Finally, the main view *awards* selects the students in the basic but not in the intensive courses. Suppose that we try the query `select * from awards;`, and that in the result we notice that the student *Anna* is missed. We know that *Anna* completed the basic course, and that although she registered in the three initial levels she did not complete one of them, and therefore she is not an intensive student. Hence, the result obtained by this query is nonvalid. A standard declarative debugger using for instance a top-down strategy [15], would ask first about the validity of the contents of *basic*, because it is the first child of *awards*. But suppose that *basic* contains hundreds of tuples, among them one tuple for *Anna*. The problem is that in order to answer that *basic* is valid, the user must check that *all* the tuples in the result are the expected ones, and that there is no missing tuple. Obviously, the question about the validity of *basic* becomes practically impossible to answer.

The main goal of this paper is to overcome or at least to reduce this drawback. This is done by asking for more specific information from the user. The questions

are now of the type "Is there a missing answer (that is, a tuple is expected but it is not there) or a wrong answer (an unexpected tuple is included in the result)?" With this information, the debugger can:
- Reduce the number of questions directed at the user. This is possible because we can deduce which relations among the children are producing/losing the wrong/missing tuple. In the previous example, the debugger would check that *Anna* is in *intensive*, and that therefore it cannot be in *awards*. Therefore, it skips any question about *basic* (the other child of *awards*), thus reducing the number of questions.
- The questions directed at the user about the validity in the children nodes can be simplified. For instance, the debugger only considers those tuples that are needed to produce the wrong or missing answer in the parent. In the example, the tool would ask if *Anna* was expected in *intensive*, without asking for the validity of the rest of the tuples in this view.

Another novelty of our approach is that we represent the computation tree using Horn clauses, which allows us to include the information obtained from the user during the session. This leads to a more flexible and powerful framework for declarative debugging that can now be combined with other diagnosis techniques. We have implemented these ideas in the system DES [12], which makes it possible for Datalog and SQL to coexist as query languages in the same database

The next section presents some basic concepts used in the rest of the paper. Section 3 introduces the debugging algorithm that constitutes the main contribution of our paper. Section 4 proves the theoretical results supporting our technique. The implementation is discussed in Section 5. Finally, Section 6 presents the conclusions and proposes future work.

2 Preliminaries

In this section, we summarize the main results of [4] by describing the basic concepts of declarative debugging applied to SQL views. We introduce an example that defines a particular database and allows us to show how the debugger works. This example is used in the rest of the paper.

2.1 Basic Concepts of Relational Databases

A *table schema* has the form $T(A_1, \ldots, A_n)$, with T being the table name and A_i the attribute names for $i = 1 \ldots n$. We refer to a particular attribute A by using the notation $T.A$. Each attribute A has an associated type. An *instance* of a table schema $T(A_1, \ldots, A_n)$ is determined by its particular *tuples*. Each tuple contains values of the correct type for each attribute in the table schema. The notation t_i represents the i-th element in the tuple. In our setting, *partial* tuples are tuples that might contain the special symbol $\perp$ in some of its components. The set of defined positions of a partial tuple s, *def(s)*, is defined by $p \in$ *def(s)* $\Leftrightarrow s_{\mathrm{p}} \neq \perp$. Tuples s with *def(s)* $= \emptyset$ are *total* tuples. Membership with partial tuples is defined as follows: if s is a partial tuple, and S a set of total tuples withthe same arity as s, we say that $s \in S$ if there is a tuple $u \in S$ such that $u_p = s_p$ for every $p \in (def(s) \cap def(u))$. Otherwise we say that $s \notin S$.

A *database schema* D is a tuple $(\mathcal{T}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of tables and $\mathcal{V}$ a finite set of views. *Views* can be thought of as new tables created dynamically from existing ones by using a SQL query. The general syntax of a SQL view is: create view $\mathsf{V}(A_1, \ldots, A_n)$ as Q, with Q a query and $V.A_1, \ldots V.A_n$ the names of the view attributes. A *database instance* d of a database schema is a set of table instances, one for each table in $\mathcal{T}$. The notation $d(T)$ represents the instance of a table T in d. The *dependency tree* of any view V in the schema is a tree with V labeling the root, and its children the dependency trees of the relations occurring in its query. In general, we will use the name *relation* to refer to either a table or a view. The syntax of SQL queries can be found in [16]. We distinguish between *basic queries* and *compound queries*. A basic query Q contains both select and from sections in its definition with the optional where, group by and having sections. For instance, the query associated to the view *standard* in the example of Figure 1 is a basic query. A compound query Q combines the results of two component queries Q_1 and Q_2 by means of set operators union [all], except [all] or intersect [all] (the keyword all indicates that the result is a multiset). For convenience, our debugger transforms basic queries into compound queries when necessary. This is the case of where sections defined in terms of and, or, ... We also assume that the queries defining views do not contain subqueries. Translating queries into equivalent definitions without subqueries is a well-known transformation (see for instance [6]). For instance, the query defining view *awards* in the Figure 1 is transformed into:

```
select student from basic
except
select student from intensive;
```

The semantics of SQL assumed in this paper is given by the Extended Relational Algebra (ERA) [8], an operational semantics allowing aggregates, views, and most of the common features of SQL queries. Each relation R is defined as a multiset of tuples. The notation $|R|_t$ refers to the number of occurrences of the tuple t in the relation R, and Φ_R represents the ERA expression associated to a SQL query or view R, as explained in [7]. A query/view usually depends on previously defined relations, and sometimes it will be useful to write $\Phi_R(R_1, \ldots, R_n)$ indicating that R depends on $R_1, \ldots, R_n$. Tables are denoted by their names, that is, $\Phi_T = T$ if T is a table.

Definition 1. *The* computed answer *of an ERA expression Φ_R with respect to some schema instance d is denoted by $\| \Phi_R \|_d$, where:*

- *If R is a database table, $\| \Phi_R \|_d = d(R)$.*
- *If R is a database view or a query and $R_1, \ldots, R_n$ the relations defined in R, then $\| \Phi_R \|_d = \Phi_R(\| \Phi_{R_1} \|_d, \ldots, \| \Phi_{R_n} \|_d)$.*

The parameter d indicating the database instance is omitted in the rest of the presentation whenever is clear from the context.

Queries are executed by SQL systems. The answer for a query Q in an implementation is represented by $\mathcal{SQL}(Q)$. The notation $\mathcal{SQL}(R)$ abbreviates $\mathcal{SQL}(\textsf{select * from } R)$. In particular, we assume in this paper the existence of *correct* SQL implementations.

Definition 2. *A* correct *SQL implementation verifies that $\mathcal{SQL}(Q) = \parallel \Phi_Q \parallel$ for every query Q.*

2.2 Declarative Debugging Framework

In the rest of the paper, D represents the database schema, d the current instance of D, and R a relation defined in D. We assume that the user can check if the computed answer for a relation matches its intended answer.

Definition 3. *The* intended answer *for a relation R w.r.t. d, is a multiset denoted as $\mathcal{I}(R)$ containing the answer that the user expects for the query* select * from R *in the instance d.*

This concept corresponds to the idea of *intended interpretations* employed usually in algorithmic debugging.

Definition 4. *We say that $\mathcal{SQL}(R)$ is an* unexpected answer *for a query* R *if* $\mathcal{I}(\text{R}) \neq \mathcal{SQL}(\text{R})$. *An unexpected answer can contain either a* wrong tuple, *when there is some tuple* t *in $\mathcal{SQL}(R)$ s.t.* $|\mathcal{I}(\text{R})|_\text{t} < |\mathcal{SQL}(\text{R})|_\text{t}$, *or a* missing tuple, *when there is some tuple* t *in* $\mathcal{I}(\text{R})$ *s.t.* $|\mathcal{I}(\text{R})|_\text{t} > |\mathcal{SQL}(\text{R})|_\text{t}$.

For instance, the intended answer for *awards* contains *Anna* once, which is represented as $|\mathcal{I}(\textsf{awards})|_{(\textsf{Anna})} = 1$. However, the computed answer does not include this tuple: $|\mathcal{SQL}(\textsf{awards})|_{(\textsf{Anna})} = 0$. Thus, *('Anna')* is a missing tuple for *awards*. In order to define the key concept of erroneous relation we need the following auxiliary concept.

Definition 5. *Let R be either a query or a relation. The* expectable *answer for R w.r.t. d, $\mathcal{E}(R)$, is defined as:*

1. *If R is a table, $\mathcal{E}(R) = d(R)$, with d the database schema instance.*
2. *If R is a view, then $\mathcal{E}(R) = \mathcal{E}(Q)$, with Q the query defining R.*
3. *If R is a query $\mathcal{E}(R) = \Phi_R(\mathcal{I}(R_1), \ldots, \mathcal{I}(R_n))$ with $R_1, \ldots, R_n$ the relations occurring in R.*

Thus, in the case of a table, the expectable answer is its instance. In the case of a view V, the expectable answer corresponds to the computed result that would be obtained assuming that all the relations R_i occurring in the definition of V contain the intended answers. Then, $\mathcal{I}(R) \neq \mathcal{E}(R)$ indicates that R does not compute its intended answer, even assuming that all the relations it depends on contain their intended answers. Such relation is called *erroneous*.

Definition 6. *We say that a relation R is* erroneous *hen $\mathcal{I}(R) \neq \mathcal{E}(R)$, and* correct *otherwise.*

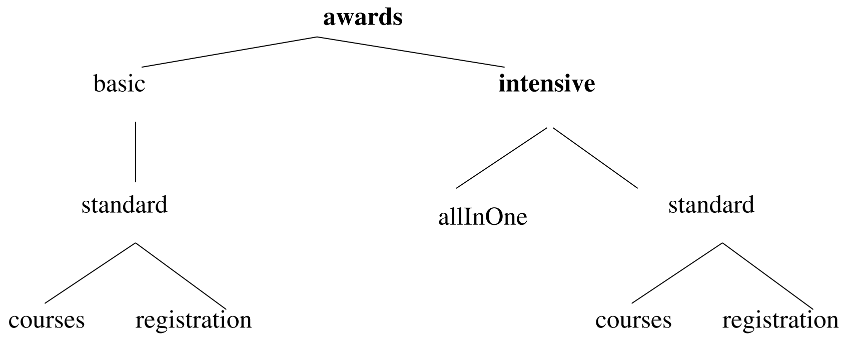

Fig. 2. Example of Computation Tree

In our running example, the real cause of the missing answer for the view *awards* is the erroneous definition of the view *intensive*.

Definition 6 clarifies the fundamental concept of erroneous relation. However, it cannot be used directly for defining a practical debugging tool, because in order to point out a view V as erroneous, it would require comparing $\mathcal{I}(V)$ and $\mathcal{E}(V)$. Instead, we require from the user only to answer questions of the form *'Is the computed answer (...) the intended answer for view V?'* Thus, the declarative debugger will compare the computed answer –obtained from the SQL system– and the intended answer –known by the user–.

The debugging process starts when the user finds a view R returning an unexpected result. In a first phase, the debugger builds a *computation tree* for this view R. The definition of this structure is the following:

Definition 7. Computation Trees
The computation tree $CT(R)$ associated with R w.r.t. the instance d is defined as follows:

- *The root of $CT(R)$ is $(R \mapsto \mathcal{SQL}(R))$.*
- *For any node $N = (R' \mapsto \mathcal{SQL}(R'))$ in $CT(R)$:*
  - *If R' is a table, then N has no children.*
  - *If R' is a view, the children of N will correspond to the CTs for the relations occurring in the query associated with R'.*

After building the computation tree, the debugger will *navigate* the tree, asking the user about the validity of some nodes:

Definition 8. Valid, Nonvalid and Buggy Nodes
Let $T = CT(R)$ be a computation tree, and $N = (R' \mapsto \mathcal{SQL}(R'))$ a node in T. We say that N is valid *when $\mathcal{SQL}(R') = \mathcal{I}(R')$,* nonvalid *when $\mathcal{SQL}(R') \neq \mathcal{I}(R')$, and* buggy *when N is nonvalid and all its children in T are valid.*

The goal of the debugger will be to locate buggy nodes. In [4] we prove that a computation tree with a nonvalid root always contains a buggy node, and that every buggy node corresponds to an erroneous relation.

Next we describe a particular debugging session based on the ideas in [4]. Figure 2 shows the computation tree of the view *awards* after removing the repeated children. The debugger marks the root of the tree as a nonvalid node (the computed answer of the view *awards* is an unexpected answer). Next, the debugger navigates the tree asking the user about the validity of some nodes. Notice these questions can be difficult to answer when the computed answer contains hundreds or thousands of tuples. The first question is about the validity of *Grants*. Suppose the user checks the answer produced by the SQL system as valid. In this case, children of *Grants* are not considered anymore. The second question is about of the validity of the view *Candidates*. The computed answer of the view *Candidates* produced by the SQL system contains Anna and the user checks this answer as nonvalid. Next, the first child of *Candidates* is visited. The user again checks the computed answer of the view *BasicLevelStudents* as nonvalid. The only child of *BasicLevelStudents* is *SalsaStudent*. As this is checked as valid, the debugger points out node *BasicLevelStudents* as buggy; node *BasicLevelStudents* is marked as nonvalid and its only child is marked as a valid node. In the tree, Nonvalid nodes are in bold face and the only buggy node (a circled node) corresponds to view *BasicLevelStudents*.

In the next Section we propose to improve the debugging technique presented in [4] by allowing the user to specify the error type, that is wrong tuple or missing tuple.

3 Debugging Algorithm

In this section we present the algorithm that defines our debugging technique, describing the purpose of each function. Although the process is based on the ideas of declarative debugging, this proposal does not use computation trees explicitly. Instead, our debugger represents computation trees by means of Horn clauses, denoted as $\mathsf{H} \leftarrow \mathsf{C}_1, \ldots, \mathsf{C}_n$, where the comma represents the conjunction, and $\mathsf{H}, \mathsf{C}_1, \ldots, \mathsf{C}_n$ are positive atoms. As usual, a *fact* H stands for the clause $\mathsf{H} \leftarrow \mathsf{true}$. Next, we describe the functions that define the algorithm, although the code of some basic auxiliary functions is omitted for the sake of space. This is the case of *getSelect*, *getFrom*, *getWhere*, and *getGroupBy* which return the different sections of a SQL query. In the case of *getFrom*, it is assumed that all the relations have an alias and the result is a sequence of elements of the form *R* as *R'*. A Boolean expression like *getGroupBy(Q)*=[] is satisfied if the query Q has no group by section. Function *getRelations(R)* returns the set of relations involved in R. It can be applied to queries, tables and views: if R is a table, then *getRelations(R)* $= \{R\}$, if R is a query, then *getRelations(R)* is the set of relations occurring in the definition of the query, and if R is a view, then *getRelations(R)* $=$ *getRelations(Q)*, with Q the query defining R. The function *generateUndefined(R)* generates a tuple whose arity is the number of attributes

Code 1 debug(V)

Input: V: view name
Output: A list of buggy views

```
1: A := askOracle(all V)
2: P := initialSetOfClauses(V, A)
3: while getBuggy(P)=[ ] do
4:     LE := getUnsolvedEnquiries(P)
5:     E := chooseEnquire(LE)
6:     A := askOracle(E)
7:     P := P ∪ processAnswer(E,A)
8: end while
9: return (getBuggy(P))
```

in R containing only undefined values $(\perp, \ldots, \perp)$. Thus, *generateUndefined(R)* $\notin S$ for every S.

The general schema of the algorithm is summarized in the code of function *debug* (Code 1). The debugger is started by the user when an unexpected answer is obtained as computed answer for some SQL view V. In our running example, the debugger is started with the call *debug(awards)*. Then, the algorithm asks the user about the type of error (line 1). The answer A can be simply *valid*, *nonvalid*, or a more detailed explanation of the error, like *wrong(t)* or *missing(t)*, indicating that t is a wrong or missing tuple respectively. In our example, A takes the initial value *missing(('Anna'))*. During the debugging process, variable P keeps a list of Horn clauses representing a logic program. The initial list of clauses P is generated by the function *initialSetofClauses* (line 2). The initialization (line 2) introduces the clauses that correspond to the computation tree rooted by V (which are listed partially in Figure 3 for the running example). The purpose of the main loop (lines 3-8) is to add information to the program P, until a buggy view can be inferred. The function *getBuggy* returns the list of all the relations R such that *buggy(R)* can be proven w.r.t. the logic program P. The clauses in P contains enquiries that might imply questions to the user. Each iteration of the loop represents the election of an enquiry in a body atom whose validity has not been established yet (lines 4-5). Then, an enquiry about the result of the query is asked to the user (line 6). Finally, the answer is processed (line 7). Next, we explain in detail each part of this main algorithm.

Code 2 corresponds to the initialization process of line 2 from Code 1. The function *initialSetofClauses* gets as first input parameter the initial view V. This view has returned an unexpected answer, and the input parameter A contains the explanation. The output of this function is a set of clauses representing the logic relations that define possible buggy relations with predicate *buggy*. Initially it creates the empty set of clauses and then it calls to *initialize* (line 2), a function that traverses recursively all the relations involved in the definition of the initial view V, calling to *createBuggyClause* with V as input parameter. *createBuggyClause* adds a new clause indicating the enquiries that must hold in order to consider V as incorrect: it must be nonvalid, and all the relations it

Code 2 initialSetofClauses(V, A)

Input: V: view name, A: answer
Output: A set of clauses

```
1: P := ∅
2: P := initialize(V)
3: P := P ∪ processAnswer((all V), A)
4: return P
```

initialize(R)

Input: R: relation
Output: A set of clauses

```
1: P := createBuggyClause(R)
2: for each R_i in getRelations(R) do
3:     P := P ∪ initialize(R_i)
4: end for
5: return P
```

createBuggyClause(V)

Input: V: view name
Output: A Horn clause

```
1: [R_1, ..., R_n] := getRelations(V)
2: return { buggy(V)← state((all V), nonvalid),
                state((all R_1), valid), ..., state((all R_n), valid)). }
```

```
buggy(awards)    :- state(all(awards),nonvalid),
                    state(all(basic),valid), state(all(intensive),valid).
buggy(basic)     :- state(all(basic),nonvalid), state(all(standard),valid).
buggy(intensive) :- state(all(intensive),nonvalid),
                    state(all(allInOneCourse),valid), state(all(standard),valid).
       ...
```

Fig. 3. Initial set of clauses for the running example

depends on must be valid. Figure 3 shows a partial list of initial clauses for our example.

The correlation between these clauses and the dependency tree is straightforward. Finally, in line 3, function *processAnswer* incorporates the information that can be extracted from A into the program P. The information about the validity/nonvalidity of the results associated to enquiries is represented in our setting with predicate *state*. The first parameter is an enquiry E, and the second one can be either *valid* or *nonvalid*. The next definition determines the possible enquiries, their associated questions and answers, and a measure $\mathcal{C}$ of the complexity of the questions:

Definition 9. *Enquiries can be of any of the following forms:* (all R)*,* (s ∈ R)*, or* (R' ⊆ R) *with* R, R' *relations, and* s *a tuple with the same schema as relation* R*. Each enquiry* E *corresponds to a specific question to the user, and it has a possible set of answers and an associated complexity* $\mathcal{C}(E)$*:*

- If E ≡ (all R)*. Let* $S = \mathcal{SQL}(R)$*. The associated question asked to the user is* "Is S the intended answer for R?" *The answer can be either* yes *or* no*. In the case of* no*, the user is asked about the type of the error,* missing *or* wrong*, giving the possibility of providing a witness tuple* t*. If the user provides this information,*

the answer is changed to missing(t) *or* wrong(t)*, depending on the type of the error. We define* $\mathcal{C}(E) = |S|$*, with* $|S|$ *the number of tuples in* S*.*

-If E $\equiv$ (R' $\subseteq$ R)*. Let* S = $\mathcal{SQL}$(R')*. Then the associated question is* "Is S included in the intended answer for R?" *As in the previous case the answer allowed can be* yes *or* no*. In the case of* no*, the user can point out a wrong tuple* t $\in S$ *and the answer is changed to* wrong(t)*.* $\mathcal{C}(E) = |S|$ *as in the previous case.*

- If E $\equiv$ (s $\in$ R)*. The question is* "Does the intended answer for R include a tuple s?" *The possible answer can be* yes *or* no*. No further information is required from the user. In this case* $\mathcal{C}(E) = 1$*, because only one tuple must be considered.*

In the case of *wrong*, the user typically points to a tuple in the result R. In the case of *missing*, the tuple must be provided by the user, and in this case *partial* tuples, i.e., tuples including some undefined attributes are allowed. The answer *yes* corresponds to the state *valid*, while the answer *no* corresponds to *nonvalid*. An atom *state(q,s)* occurring in a clause body, is a *solved enquiry* if the logic program P contains at least one fact of the form *state(q, valid)* or *state(q, nonvalid)*, that is, if the enquiry has been already solved. The atom is called an *unsolved enquiry* otherwise. The function *getUnsolvedEnquiries* (see line 4 of Code 1) returns in a list all the unsolved enquiries occurring in P. The function *chooseEnquiry* (line 5, Code 1) chooses one of these enquiries according to some criteria. In our case we choose the enquiry E that implies the smaller complexity value $\mathcal{C}(E)$, although other more elaborated criteria could be defined without affecting the theoretical results supporting the technique. Once the enquiry has been chosen, Code 1 uses the function *askOracle* (line 6) in order to ask to the associated question, returning the answer of the user. We omit the definitions of these simple functions for the sake of space.

The code of function *processAnswer* (called in line 7 of Code 1), can be found in Code 3. The first lines (1-5) introduce a new logic fact in the program with the state that corresponds to the answer A obtained for the enquiry E. In our running example, the fact *state(all(awards), nonvalid)* is added to the program. The rest of the code distinguishes several cases depending on the form of the enquiry and its associated answer. If the enquiry is of the form $(s \in R)$ with answer is *no* (meaning $s \notin \mathcal{I}(R)$), and the debugger checks that the tuple s is in the computed answer of the view R (line 7), then s is wrong in the relation R. In this case, the function *processAnswer* is called recursively with the enquiry *(all R)* and *wrong(s)* (line 8). If the answer is *yes* and the debugger checks that s does not belong to the computed answer of R (line 10), then s is missing in the relation R. For enquiries of the form $(V \subseteq R)$ and answer *wrong(s)*, it can be ensured that s is wrong in R (line 13). If the enquiry is *(all V)* for some view V, and with an answer including either a wrong or a missing tuple, the function *slice* (line 16) is called. This function exploits the information contained in the parameter A (*missing(t)* or *wrong(t)*) for slicing the query Q in order to produce, if possible, new clauses which will allow the debugger to detect incorrect relations by asking simpler questions to the user. The implementation of *slice* can be found

Code 3 processAnswer(E,A)

Input: E: enquiry, A: answer obtained for the enquiry
Output: A set of new clauses

```
if A ≡ yes then
   P := {state(E,valid).}
else if A ≡ no or A ≡ missing(t) or A ≡ wrong(t) then
   P := {state(E,nonvalid).}
end if
if E ≡ (s ∈ R)  then
   if (s ∈ SQL(R) and A ≡ no) then
      P:= P ∪ processAnswer((all R),wrong(s))
   else if (s∉ SQL(R) and A ≡ yes) then
      P:= P ∪ processAnswer((all R),missing(s))
   end if
else if E ≡ (V ⊆ R) and (A ≡ wrong(s)) then
   P:= P ∪ processAnswer((all R), A)
else if E ≡ (all V) with V a view and (A ≡ missing(t) or A ≡ wrong(t)) then
   Q := SQL query defining V
   P := P ∪ slice(V,Q,A)
end if
return  P
```

in Code 4. The function receives the view V, a subquery Q, and an answer A as parameters. Initially, Q is the query defining V, and A the user answer, but this situation can change in the recursive calls. The function distinguishes several particular cases:

- The query Q combines the results of Q_1 and Q_2 by means of either the operator union or union all, and A is *wrong(t)* (first part of line 2). This means that the query Q produces too many copies of t. Then, if any Q_i produces as many copies of t as Q, we can blame Q_i as the source of the excessive number of t's in the answer for V (lines 3 and 4). The case of subqueries combined by the operator intersect [all], with $A \equiv$ *missing(t)* is analogous, but now detecting that a subquery is the cause of the scanty number of copies of t in $\mathcal{SQL}(V)$.

- The query Q is of the form Q_1 except [all] Q_2, with $A \equiv$ *missing(t)* (line 5). If the number of occurrences of t in both Q and Q_1 is the same, then t is also missing in the query Q_1 (line 6). Additionally, if query Q is of the particular form Q_1 except Q_2, which means that we are using the difference operator on sets (line 7), then if t is in the result of Q_2 it is possible to claim that the tuple t is wrong in Q_2. Observe that in this case the recursive call changes the answer from *missing(t)* to *wrong(t)*.

- If Q is defined as a basic query without group by section (line 8), then either function *missingBasic* or *wrongBasic* is called depending on the form of A.

Both *missingBasic* and *wrongBasic* can add new clauses that allow the system to infer buggy relations by posing questions which are easier to answer. Function *missingBasic*, defined in Code 5, is called (line 9 of Code 4) when A is *missing(t)*. The input parameters are the view V, a query Q, and the missing

Code 4 slice(V,Q,A)

Input: V: view name, Q: query, A: answer
Output: A set of new clauses

```
P := ∅;   S= SQL(Q);   S1= SQL(Q1);   S2= SQL(Q2)
if (A ≡ wrong(t) and Q ≡ Q1 union [all] Q2) or
      (A ≡ missing(t) and Q ≡ Q1 intersect [all] Q2) then
   if |S1|t = |S|t then  P:= P ∪ slice(V, Q1, A)
   if |S2|t = |S|t then  P:= P ∪ slice(V, Q2, A)
else if A ≡ missing(t) and Q ≡ Q1 except [all] Q2  then
   if |S1|t = |S|t then  P:= P ∪ slice(V, Q1, A)
   if Q ≡ Q1 except Q2 and t ∈ S2 then P :=P∪ slice(V,Q2,wrong(t))
else if basic(Q) and groupBy(Q)=[] then
   if A ≡ missing(t) then P := P ∪ missingBasic(V, Q, t)
   else if A ≡ wrong(t) then P := P ∪ wrongBasic(V, Q, t)
end if
return  P
```

Code 5 missingBasic(V,Q,t)

Input: V: view name, Q: query, t: tuple
Output: A new list of Horn clauses

```
P := ∅;   S := SQL(SELECT getSelect(Q) FROM getFrom(Q) )
if t ∉ S then
   for (R AS S) in (getFrom(Q)) do
      s = generateUndefined(R)
      for i=1 to length(getSelect(Q)) do
         if ti ≠⊥and member(getSelect(Q),i) = S.A, A attrib., then s.A = ti
      end for
      if s ∉ SQL(R) then
         P := P ∪ { (buggy(V) ← state((s ∈ R), nonvalid).) }
      end if
   end for
end if
return  P
```

Code 6 wrongBasic(V,Q,t)

```
Input: V: view name, Q: query, t: tuple
Output: A set of clauses
1: P := ∅
2: F := getFrom(Q)
3: N := length(F)
4: for i=1 to N do
5:    R_i as S_i := member(F,i)
6:    relevantTuples(R_i,S_i,V_i, Q, t)
7: end for
8: P := P ∪ { (buggy(V) ← state((V_1 ⊆ R_1), valid), ..., state((V_n ⊆ R_n), valid).) }
9: return P
```

Code 7 relevantTuples(R_i,R',V,Q,t)

```
Input: R_i: relation, R': alias,
   V: new view name, Q: Query, t: tuple
Output: A new view in the database schema
1: Let A_1, ..., A_n be the attributes defining R_i
2: SQL(create view V as
     (select R_i.A_1, ..., R_i.A_n from R_i)
        intersect all
     (select R'.A_1, ..., R'.A_n from getFrom(Q)
     where getWhere(Q) and eqTups(t,getSelect(Q))))
```

eqTups(t,s)

```
Input: t,s : tuples
Output: SQL condition
1: C := true
2: for i=1 to length(t) do
3:    if t_i ≠ ⊥ then
4:       C:= C AND t_i = s_i
5: end for
6: return C
```

tuple t. Notice that Q is in general a component of the query defining V. For each relation R with alias S occurring in the from section, the function checks if R contains some tuple that might produce the attributes of the form $S.A$ occurring in the tuple t. This is done by constructing a tuple s undefined in all its components (line 4) except in those corresponding to the select attributes of the form $S.A$, which are defined in t (lines 5 - 7). If R does not contain a tuple matching s in all its defined attributes (line 8), then it is not possible to obtain the tuple t in V from R. In this case, a buggy clause is added to the program P (line 9) meaning that if the answer to the question *"Does the intended answer for R include a tuple s?"* is *no*, then V is an incorrect relation.

The implementation of *wrongBasic* can be found in Code 6. The input parameters are again the view V, a query Q, and a tuple t. In line 1, this function creates an empty set of clauses. In line 2, variable F stands for the set containing all the relations in the from section of the query Q. Next, for each relation $R_i \in F$ (lines 4 - 7), a new view V_i is created in the database schema after calling the function *relevantTuples* (line 6), which is defined in Code 7. This auxiliary view contains only those tuples in relation R_i that contribute to produce the wrong tuple t in V. Finally, a new buggy clause for the view V is added to the program P (line 8) explaining that the relation V is buggy if the answer to the question associated to each enquiry of the form $V_i \subseteq \mathrm{R}_i$ is *yes* for $i \in \{1 \ldots n\}$.

4 Theoretical Results

In the previous section we have introduced the debugging algorithm, explaining the intuitive ideas supporting the technique. Now we establish formally the soundness of the proposal. In the rest of the section we assume that the debugging algorithm uses a SLD-based logic system for checking the atoms that are entailed by the program contained in the variable P of Code 1. The notation P $\vdash$ A denotes that there is a SLD proof for A with respect to the program P.

We start checking the correctness of the framework.

Theorem 1. *Correctness.*
Let R be a relation such that debug(R) *(defined in Code 1) returns a list L. Then all relation names contained in L are erroneous relations.*

Proof. The logic program contained in the variable P of Code 1. Then L contains all the atoms R such that

$$P \vdash \textit{buggy(R)} \tag{1}$$

Let R be any of the relations verifying (1). We prove that R is an erroneous relation. According to Definition 6 this means that we must check $\mathcal{I}(R) \neq \mathcal{E}(R)$.

The SLD inference proving *buggy(R)* must start using a clause with head *buggy(R)* (R is a relation name, and therefore *buggy(R)* is a ground atom). The algorithm code introduces clauses for predicate *buggy* at three points:

1.- In Code 2 (function *createBuggyClause*, line 2). The clause is:

buggy(R)← state((all R), nonvalid),state((all R_1), valid),. . . ,state((all R_n), valid)

where $R_1, \ldots, R_n$ are all the relations employed in the definition of R. Then, by Lemma 4:

$$P \vdash \textit{state((all } R_i\textit{), valid)} \Longrightarrow \mathcal{SQL}(R_i) = \mathcal{I}(R_i) \text{ for } i = 1 \ldots n \tag{2}$$

$$P \vdash \textit{state((all R), nonvalid)} \Longrightarrow \mathcal{SQL}(R) \neq \mathcal{I}(R) \tag{3}$$

Let d the current database instance. If R is a table:

$$\text{By Definitions 1 and 2,} \qquad \mathcal{SQL}(R) = d(R) \tag{4}$$

$$\text{By Definition 5, item 1,} \qquad \mathcal{E}(R) = d(R) \tag{5}$$

Combining (4) and (5), $\mathcal{SQL}(R) = \mathcal{E}(R)$, and considering (3) the result $\mathcal{E}(R) \neq \mathcal{I}(R)$ is obtained. If R is a view:

$$\text{By Definitions 1 and 2,} \qquad \mathcal{SQL}(R) = \Phi_R(\mathcal{SQL}(R_1), \ldots, \mathcal{SQL}(R_n)) \tag{6}$$

$$\text{Applying (2) to (6),} \qquad \mathcal{SQL}(R) = \Phi_R(\mathcal{I}(R_1), \ldots, \mathcal{I}(R_n)) \tag{7}$$

Combining (7) and Definition 5, items 2-3, $\mathcal{SQL}(R) = \mathcal{E}(R)$, and thus by (3), $\mathcal{E}(R) \neq \mathcal{I}(R)$.

2.- In Code 5, line 9, which introduces the clause:

$$(\mathsf{buggy(V)} \leftarrow \mathsf{state((s \in R),\ nonvalid)})$$

Then $P \vdash$ *state((*s $\in$ R*), nonvalid))*, and:

$$\text{By Code 5, line 8,} \qquad s \notin \mathcal{SQL}(R) \tag{8}$$

$$\text{By Lemma 4,} \qquad s \notin \mathcal{I}(R) \tag{9}$$

Observe that function *missingBasic* in Code 5 is called from function *slice* (Code 4) which ensures that Q is defined by a basic query without group by section. Moreover, examining code of function *missingBasic* it is clear that the *select* clause contains $m > 1$ values of the form $S.A_1, \ldots, S.A_m$ with S *AS* R a relation in the from clause (otherwise the tuple s will be completely undefined and the condition $s \notin \mathcal{SQL}(R)$ could not hold). Therefore, the ERA expression of Q can be written as

$$\Phi_Q = \Pi_{(\ldots S.A_1,\ldots,S.A_m\ldots)}(\sigma_C(\cdots \times \rho_S(R) \times \ldots)) \tag{10}$$

for some condition C. Then, by Definitions 1 and 2:

$$\mathcal{SQL}(Q) = \parallel \Phi_Q \parallel = \Pi_{(\ldots S.A_1,\ldots,S.A_m\ldots)}(\sigma_C(\cdots \times \rho_S(\mathcal{SQL}(R)) \times \ldots)) \tag{11}$$

Applying Definition 5 to (10) we obtain:

$$\mathcal{E}(Q) = \Pi_{(\ldots S.A_1,\ldots,S.A_m\ldots)}(\sigma_C(\cdots \times \rho_S(\mathcal{I}(R)) \times \ldots)) \tag{12}$$

Observing that s is a subtuple (in the defined positions) of the missing tuple t by construction of s, it is straightforward to prove that:

$$\text{By (8) and (11),} \qquad t \notin \mathcal{SQL}(Q)$$

$$\text{By (9) and (12),} \qquad t \notin \mathcal{E}(Q)$$

which means that

$$|\mathcal{E}(Q)|_t = |\mathcal{SQL}(Q)|_t = 0 \tag{13}$$

Observe finally that *slice* is called from *processAnswer*, and that therefore by Lemma 3 V is an incorrect view.

3.- In Code 6, which introduces the clause:

$$(\mathsf{buggy(V)} \leftarrow \mathsf{state((V_1 \subseteq R_1),\ valid)}, \ldots, \mathsf{state((V_n \subseteq R_n),\ valid)})$$

By Lemma 4:

$$\text{For i=1} \ldots \text{n,} \qquad \mathcal{SQL}(V_i) \subseteq \mathcal{I}(R_i) \tag{14}$$

Examing the call to *wrongBasic* in *slice* (Code 4, line 10), it is obvious that Q has no group by section, and therefore its ERA expression is of the form

$$\Phi_Q = \Pi_{(S)}(\sigma_C(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_n}(R_n))) \tag{15}$$

for some list of expressions S and condition C. Using Definitions 1 and 2 we obtain the SQL computed answer

$$\mathcal{SQL}(Q) = \parallel \Phi_Q \parallel = \Pi_{(S)}(\sigma_C(\rho_{S_1}(\mathcal{SQL}(R_1)) \times \cdots \times \rho_{S_n}(\mathcal{SQL}(R_n))))$$

and in particular:

$$|\mathcal{SQL}(Q)|_t = |\Pi_{(S)}(\sigma_C(\rho_{S_1}(\mathcal{SQL}(R_1)) \times \cdots \times \rho_{S_n}(\mathcal{SQL}(R_n))))|_t \qquad (16)$$

By Lemma 1, replacing each R_i by its corresponding R_i does not affect to the number of copies of t obtained, that is:

$$|\mathcal{SQL}(Q)|_t = |\Pi_{(S)}(\sigma_C(\rho_{S_1}(\mathcal{SQL}(V_1)) \times \cdots \times \rho_{S_n}(\mathcal{SQL}(V_n))))|_t \qquad (17)$$

It is easy to check that in an expression like (17), replacing a relation V_i in the cartesian product by other relation S such that $V_i \subseteq S$ implies at least the same tuples in the result (and possibly more, new tuples). Therefore, applying (14) to (17):

$$|\mathcal{SQL}(Q)|_t \leq |\Pi_{(S)}(\sigma_C(\rho_{S_1}(\mathcal{I}(R_1)) \times \cdots \times \rho_{S_n}(\mathcal{I}(R_n))))|_t \qquad (18)$$

and applying the definition of expectable answer (Definition 5) to the right-hand side of this inequality:

$$|\mathcal{SQL}(Q)|_t \leq |\mathcal{E}(Q)|_t \qquad (19)$$

Taking into account that the function *wrongBasic* has been called from *slice* (Code 4, line 10), with the same input parameters *(*V), Q, and with $A \equiv$ *wrong(t)*, then Lemma 3 can be applied to the call *slice(V,Q,A)*, and (19) implies that V is an incorrect relation.

■

This result uses various auxiliary Lemmata, which are proved below. First we prove a property of the new view created by function *relevantTuples*.

Lemma 1. *After a call of the form* relevantTuples(R_i,R',V,Q,t), V *is a new view such that*

1. $\mathcal{SQL}(V) \subseteq \mathcal{SQL}(\mathrm{R}_i)$
2. *Let* Q' *the result of replacing* R_i *by* V *in* Q. *Then* $|\mathcal{SQL}(Q)|_t = |\mathcal{SQL}(Q')|_t$

Proof.

1. The definition of V can be found in Code 7. Suposse that *getFrom(Q)* = R_1 *as* R'_1, ..., R_i *as* R'_i, ..., R_m *as* R'_m, *getWhere(Q)* = C, and that *getSelect(Q)* = e_1, ..., e_k, the definition can represented as:

```
create view V as
  (select Ri.A1, ..., Ri.An from Ri)
     intersect all
  (select R'.A1, ..., R'.An
   from R1 as R'1, ..., Ri as R'i, ..., Rm as R'm
    where C and e1=t1 and ... and ek=tk)
```
(20)

which, taking into account that $R_i.A_1, \ldots, R_i.A_n$ are all the attributes of R_i means that (20) can be represented ERA

$$V \leftarrow R_i \cap_{\mathcal{M}} \Pi_{R'.A_1,\ldots,R'.A_n}(\sigma_{C\wedge e_1=t_1\wedge\cdots\wedge e_k=t_k}(\rho_{R'_1}(R_1) \times \cdots \times \rho_{R'_m}(R_m)) \tag{21}$$

and therefore V is a subset of R_i.

2. The function *relevantTuples* is called from function *wrongBasic* (line 6, Code 6), which is called from function *slice* (Code 4, line 10). The *if* sentence in *slice* ensures that Q is a basic query without group by clause. Therefore, Q must be of the form:
```
select e1, ..., ek
from R1 as R'1, ..., Ri as R'i, ..., Rm as R'm
where C
```
which can be represented in ERA as:

$$\Phi_Q = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{R'_1}(R_1) \times \ldots \rho_{R'_i}(R_i) \cdots \times \rho_{R'_m}(R_m))) \tag{22}$$

We call $\Phi_{Q'}$ to the result of replacing R'_i by V in (22):

$$\Phi_{Q'} = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{R'_1}(R_1) \times \ldots \rho_{R'_i}(V) \cdots \times \rho_{R'_m}(R_m))) \tag{23}$$

observe that only the from section needed to be modified, because the rest of the query does not include R_i but his alias R'_i, and alias is kept unaltered in $\Phi_{Q'}$. The we must prove that $\parallel \Phi_Q \parallel_t = \parallel \Phi_{Q'} \parallel_t$. Taking into account that neither (22) nor (23) contain set differences (the negation in ERA), and from $\mathcal{SQL}(V) \subseteq \mathcal{SQL}(R_i)$, we have $\parallel \Phi_Q \parallel_t \geq \parallel \Phi_{Q'} \parallel_t$. Then it is is enought to prove $\parallel \Phi_Q \parallel_t \leq \parallel \Phi_{Q'} \parallel_t$ to complete the result, that is we must check that if t occurs with arity n in $\parallel \Phi_Q \parallel$ then it occurs with the same arity in $\parallel \Phi_{Q'} \parallel$.
Observing (22) it is obvious that the n copies of t in $\parallel \Phi_Q \parallel$ must come from n tuples $s_i \in \rho_{R'_1}(R_1) \times \ldots \rho_{R'_i}(R_i) \cdots \times \rho_{R'_m}(R_m))$, $i = 1 \ldots p$. Each tuple s_i satisfies the condition C and projected over $e_1, \ldots, e_k$ produces the value t. Therefore

$$s_i \in (\sigma_{C\wedge e_1=t_1\wedge\cdots\wedge e_k=t_k}(\rho_{R'_1}(R_1) \times \cdots \times \rho_{R'_m}(R_m)) \tag{24}$$

Then, by the structure of (23), it is enough to check that $s_i \in \rho_{R'_1}(R_1) \times \ldots \rho_{R'_i}(V) \cdots \times \rho_{R'_m}(R_m))$, $i = 1 \ldots p$. If we call u_i, $i = 1 \ldots p$ to the restriction of each s_i to the attributes of R_i, then the proof is complete if we check that $u_i \in V$ for $i = 1 \ldots p$. That is, from (24):

$$u_i \in \Pi_{R'.A_1,\ldots,R'.A_n}(\sigma_{C \wedge e_1=t_1 \wedge \cdots \wedge e_k=t_k}(\rho_{R'_1}(R_1) \times \cdots \times \rho_{R'_m}(R_m)) \quad (25)$$

with $A_1, \ldots, A_n$ the attributes of R_i. Then $u_i \in R_i$, and therefore

$$u_i \in (R_i \cap_{\mathcal{M}} \Pi_{R'.A_1,\ldots,R'.A_n}(\sigma_{C \wedge e_1=t_1 \wedge \cdots \wedge e_k=t_k}(\rho_{R'_1}(R_1) \times \cdots \times \rho_{R'_m}(R_m)))) \quad (26)$$

which by (21) implies $u_i \in V$, which completes the proof. ■

Next we prove an auxiliary result that establishes the relationship between enquiries and answers:

Lemma 2. *Let* processAnswer(E,A) *be any call to Code 3 that occur during the execution of the debugger. Then:*

1. *If* E ≡ (s ∈ R)*, then:*
   (a) *If* A ≡ yes*, then* s $\in \mathcal{I}(R)$.
   (b) *If* A ≡ no*, then* s $\notin \mathcal{I}(R)$.
2. *If* E ≡ (V ⊆ R)*, then:*
   (a) *If* A ≡ yes*, then* $\mathcal{SQL}(\mathrm{V}) \subseteq \mathcal{I}(\mathrm{R})$.
   (b) *If* A ≡ no*, then* $\mathcal{SQL}(\mathrm{V}) \nsubseteq \mathcal{I}(\mathrm{R})$.
   (c) *If* A ≡ wrong(t)*, then* $|\mathcal{I}(\mathrm{R})|_{\mathrm{t}} < |\mathcal{SQL}(\mathrm{V})|_{\mathrm{t}}$.
3. *If* E ≡ (all R)*, then:*
   (a) *If* A ≡ yes*, then* $\mathcal{SQL}(\mathrm{R}) = \mathcal{I}(\mathrm{R})$
   (b) *If* A ≡ no*, then* $\mathcal{SQL}(\mathrm{R}) \neq \mathcal{I}(\mathrm{R})$
   (c) *If* A ≡ wrong(t)*, then* $|\mathcal{I}(\mathrm{R})|_{\mathrm{t}} < |\mathcal{SQL}(\mathrm{R})|_{\mathrm{t}}$
   (d) *If* A ≡ missing(t)*, then* $|\mathcal{I}(\mathrm{R})|_{\mathrm{t}} > |\mathcal{SQL}(\mathrm{R})|_{\mathrm{t}}$

Proof. We distinguish cases depending on the form of the input parameters E, A.

– $E \equiv (V \subseteq R)$. Then the function has been called after asking the user about the validity of the enquire E, obtaining answer A. This happens in Code 1, line 7, and also in Code 2, line 3. By Definition 9, this case corresponds to the question *"Is S included in the intended answer for R?"*, with $S = \mathcal{SQL}$*(select * from* R_1*)* $= \mathcal{SQL}(R_1)$. If the answer of the user was *yes*, this means that $\mathcal{SQL}(V) \subseteq \mathcal{I}(R)$, while the answer *no*, meansthat $\mathcal{SQL}(V) \nsubseteq \mathcal{I}(R)$. If the user indicates an answer *wrong(t)*, this means that V contains more copies of t than expected in R. Therefore $|\mathcal{I}(R)|_t < |\mathcal{SQL}(V)|_t$.
– $E \equiv (t \in R)$. A is then the answer provided by the user to the question *"Does R include a tuple of the form t?"*, and the result is analogous.

– $E \equiv$ *(all R)*. This input parameter corresponds to calls obtained in two different situations:

1. As in the previous cases, when the debugger obtains the user answer to a question, in this case *"Is S the intended answer for R?"*, with $S = \mathcal{SQL}(select\ *\ from\ \ R) = \mathcal{SQL}(R)$. Then the result is analogous. For instance if the user answers *yes*, the user is indicating that $\mathcal{SQL}(R) = \mathcal{I}(R)$.
2. In a recursive call produced by *processAnswer*. It is easy to check that only one recursive call can occur, due to the change in the first parameter to *(all R)* (which avoids further recursive calls). That is, a first call occurs containing the answer provided by the user, and the execution of this call starts a recursive call, which does not call *processAnswer* recursively. The recursive calls are located in three points of Code 3:
   - Line 8. Then, the initial call is *processAnswer((s* $\in$ *R),no)*As explained above this implies

$$s \notin \mathcal{I}(R) \tag{27}$$

     Also, the condition of the *if* statement preceding the recursive call ensures that

$$s \in \mathcal{SQL}(R) \tag{28}$$

     The recursive call is *processAnswer((all R), wrong(s))*. Then we must prove that $\mathcal{SQL}(R) \neq \mathcal{I}(R)$, and this is straightforward from (27) and (28).
   - Line 10. The initial call must be *processAnswer((s* $\in$ *R),yes)*, which means:

$$s \in \mathcal{I}(R) \tag{29}$$

     The condition of the *if* statement indicates that

$$s \notin \mathcal{SQL}(R) \tag{30}$$

     The recursive call is *processAnswer((all R),missing(s))*, and the expected result $\mathcal{SQL}(R) \neq \mathcal{I}(R)$ follows from (29) and (30).
   - Line 13. The first call must be *processAnswer((V* $\subseteq$ *R), wrong(s))*. This is one of the cases already analyzed, where *wrong(s)* is the answer provided by the user for the enquiry *(V* $\subseteq$ *R)*. Now, observe that this enquiry must correspond to the election of an atom *state((V* $\subseteq$ *R), ...)* already ocurring in the program. Such atoms are introduced in line 8 of Code 6. The view V has been created by function *relevantTuples* (Code 7), and by Lemma 1:

$$\mathcal{SQL}(V) \subseteq \mathcal{SQL}(R) \tag{31}$$

This first call *processAnswer((V ⊆ R), wrong(s))* has introduced a fact *state((V ⊆ R),nonvalid)*, and we have already prove that this implies:

$$\mathcal{SQL}(V) \not\subseteq \mathcal{I}(R) \quad (32)$$

which, combined with 31 means:

$$\mathcal{SQL}(R) \not\subseteq \mathcal{I}(R) \quad (33)$$

The recursive call is *processAnswer((all R), wrong(s))*, and we must prove that $\mathcal{SQL}(R) \neq \mathcal{I}(R)$, which is a direct consequence of (33).

■

Lemma 3. *Let* slice(V,Q,A) *be any call to Code 4 that occur during the execution of the debugger. Then:*

- *If* $\mathsf{A} \equiv \mathsf{wrong(t)}$ *and* $|\mathcal{E}(\mathsf{Q})|_t \geq |\mathcal{SQL}(\mathsf{Q})|_t$*, then* V *is an incorrect view.*
- *If* $\mathsf{A} \equiv \mathsf{missing(t)}$ *and* $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$*, then* V *is an incorrect view.*

Proof. We prove the results by induction on the number n of recursive calls to slice occurred before the current call.

If $n = 0$, then the initial call for slice corresponds to processAnswer, Code 3, line 16. This call ensures that V is a view, Q is initially the query defining V, and A is either missing(t) or wrong(t), where t has been pointed out as missing (respectively wrong) by the user. By definition 4, and taking into account that $\mathcal{SQL}(\mathsf{V}) = \mathcal{SQL}(\mathsf{Q})$, we have that in this first call:

- If A is wrong(t), then $|\mathcal{I}(V)|_t < |\mathcal{SQL}(Q)|_t$. Therefore, $|\mathcal{E}(\mathsf{Q})|_t \geq |\mathcal{SQL}(\mathsf{Q})|_t$ implies $|\mathcal{E}(\mathsf{Q})|_t > |\mathcal{I}(V)|_t$.
- If A is missing(t), then $|\mathcal{I}(V)|_t > |\mathcal{SQL}(Q)|_t$. Then, $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$ implies $|\mathcal{E}(\mathsf{Q})|_t < |\mathcal{I}(V)|_t$.

In both cases, and considering that from Def. 5 $\mathcal{E}(\mathsf{V}) = \mathcal{E}(\mathsf{Q})$, we have that $|\mathcal{E}(\mathsf{V}) \neq \mathcal{I}(\mathsf{V})|$, that is V is erroneous (Def. 6).

If $n > 0$ we are considering a recursive call slice(V,Q',A'). All the recursive calls occur of Code 4 verify that they do not change the V, which is hence the same as in the initial call. The values Q' and A', might have changed with respect to the input values Q and A. By inductive hypothesis we have that this Lemma can be applied to the input values V, Q, and A. Now we check that the result can be applied also to V, Q' and A', distinguising cases depending on the particular call:

- Code 4, Line 3. In this case

  $$(1) \quad |\mathcal{SQL}(\mathsf{Q}_1)| = |\mathcal{SQL}(\mathsf{Q})|_t$$

  and one of the following conditions hold:

- $\mathsf{A} \equiv \mathsf{missing(t)}$ AND $\mathsf{Q} \equiv \mathsf{Q}_1$ INTERSECT Q_2. Then
  (2) If $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$, then V is incorrect (induction hypothesis).
  (3) If $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, then V is incorrect (by (1) and (2)).
  From $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \cap \Phi_{\mathsf{Q}_2}$, and Definition 5 we have $\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \cap \mathcal{E}(\mathsf{Q}_2)$. Thefore
  $$(4) \quad |\mathcal{E}(\mathsf{Q}_1)|_t \geq |\mathcal{E}(\mathsf{Q})|_t$$
  and finally combining (3) and (4) we have the expected result:
  (5) If $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, then by (3) $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, and by (4) this means that V is incorrect.
- $\mathsf{A} \equiv \mathsf{missing(t)}$ AND $\mathsf{Q} \equiv \mathsf{Q}_1$ INTERSECT ALL Q_2. Analogous to the previous point. Observe that replacing the set operator $\cap$ by $\cap_{\mathcal{M}}$ does not affect to the result.
- Either ($\mathsf{A} \equiv \mathsf{wrong(t)}$ AND $\mathsf{Q} \equiv \mathsf{Q}_1$ UNION Q_2), or ($\mathsf{A} \equiv \mathsf{wrong(t)}$ AND $\mathsf{Q} \equiv \mathsf{Q}_1$ UNION ALL Q_2). Very similar to the corresponding cases of the intersection and of the multiset intersection (in the case of ALL), replacing $\cap$ by $\cup$ ($\cap_{\mathcal{M}}$ by $\cup_{\mathcal{M}}$ in the case of ALL), $\geq$ by $\leq$, and $\leq$ by $\geq$.

– Code 4, Line 4. Analogous to the previous case changing Q_1 by Q_2.
– Code 4, Line 6. Then $\mathsf{A} \equiv \mathsf{missing(t)}$ AND $\mathsf{Q} \equiv \mathsf{Q}_1$ EXCEPT [ALL] Q_2, and (1), (2), (3) hold. Then $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \setminus \Phi_{\mathsf{Q}_2}$ (changing $\setminus$ by $\setminus_{\mathcal{M}}$ in the case of ALL), and by Definition 5 (4) holds as well. Then by the same reasoning as in the first case, the result (5) holds for the call slice(V, Q_1, A).
– Code 4, Line 7: Then $\mathsf{A} \equiv \mathsf{missing(t)}$, $\mathsf{Q} \equiv \mathsf{Q}_1$ EXCEPT Q_2, $\mathsf{t} \in \mathcal{SQL}(\mathsf{Q}_2)$, and (2) holds. Then $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \setminus \Phi_{\mathsf{Q}_2}$, which means by Definition 5
$$(6) \qquad \mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \setminus \mathcal{E}(\mathsf{Q}_2)$$
and by Definition 1,
$$(7) \qquad \mathcal{SQL}(Q) = \parallel \Phi_Q \parallel = \parallel \Phi_{Q_1} \parallel \setminus \parallel \Phi_{Q_2} \parallel = \mathcal{SQL}(Q_1) \setminus \mathcal{SQL}(Q_2)$$
From $\mathsf{t} \in \mathcal{SQL}(\mathsf{Q}_2)$, we have that $|\mathcal{SQL}(Q)|_t = 0$, which means that the induction hypothesis (2) can be rewwriten as:
$$(8) \qquad \text{If } \mathsf{t} \notin \mathcal{E}(\mathsf{Q}), \text{ then V is incorrect}$$
Now observe that in this case the call to slice is slice(V,Q_2,wrong(t)). Therefore we must prove that
$$(9) \qquad \text{If } |\mathcal{E}(\mathsf{Q}_2)|_t \geq |\mathcal{SQL}(\mathsf{Q}_2)|_t, \text{ then V is an incorrect view}$$
Assume that $|\mathcal{E}(\mathsf{Q}_2)|_t \geq |\mathcal{SQL}(\mathsf{Q}_2)|_t$ in (9) holds. This implies in particular that $\mathsf{t} \in \mathcal{E}(\mathsf{Q}_2)$, which by (6) means that $\mathsf{t} \notin \mathcal{E}(\mathsf{Q}_2)$, which combined with (8) implies that V is incorrect, and thus (9) holds. ■

The next lemma indicate how state relates the answers obtained by the SQL system and the intended interpretation $\mathcal{I}$:

Lemma 4. *Let* R *be a relation,* $\mathcal{I}$(R) *its intended answer w.r.t. the current instance, and let* $\mathcal{P}$ *be the logic program contained in the variable* P *of Code 1. Then, the following implications hold at any moment of the execution of the algorithm:*

$$
\begin{array}{ll}
\mathcal{P} \vdash \text{state}((\text{all R}), \text{valid}) & \Rightarrow \mathcal{SQL}(\text{R}) = \mathcal{I}(\text{R}) \\
\mathcal{P} \vdash \text{state}((\text{all R}), \text{nonvalid}) & \Rightarrow \mathcal{SQL}(\text{R}) \neq \mathcal{I}(\text{R}) \\
\mathcal{P} \vdash \text{state}((\text{t} \in \text{R}), \text{valid}) & \Rightarrow \text{t} \in \mathcal{I}(\text{R}) \\
\mathcal{P} \vdash \text{state}((\text{t} \in \text{R}), \text{nonvalid}) & \Rightarrow \text{t} \notin \mathcal{I}(\text{R}) \\
\mathcal{P} \vdash \text{state}((\text{R}_1 \subseteq \text{R}), \text{valid}) & \Rightarrow \mathcal{SQL}(\text{R}_1) \subseteq \mathcal{I}(\text{R}) \\
\mathcal{P} \vdash \text{state}((\text{R}_1 \subseteq \text{R}), \text{nonvalid}) & \Rightarrow \mathcal{SQL}(\text{R}_1) \not\subseteq \mathcal{I}(\text{R})
\end{array}
$$

Proof. Proving $\mathcal{P} \vdash$ *state(E,S)* implies that there is a fact *state(E,S)* $\in \mathcal{P}$, because the algorithm only introduces facts for this predicate. And only function *processAnswer(E,A)* (Code 3) introduces facts of this form in the program (lines 1-5). We distinguish cases depending on the form of the input parameters *E*, *A*.

- $E \equiv (R_1 \subseteq R)$. Then the function has been called after asking the user about the validity of the enquire *E*, obtaining answer A. This happens in Code 1, line 7, and also in Code 2, line 3. This case corresponds to the question *"Is S included in the intended answer for R?"*, with $S = \mathcal{SQL}$*(select * from* R_1*)* $= \mathcal{SQL}(R_1)$. If the function *processAnswer* introduces the fact *state(*$(R_1 \subseteq R)$*,valid)*, this implies that the answer of the user was *yes*, meaning that $\mathcal{SQL}(R_1) \subseteq \mathcal{I}(R)$. The function *processAnswer* introduces the fact *state(*$(R_1 \subseteq R)$*,nonvalid)* when the answer of the user is *no*, meaning that $\mathcal{SQL}(R_1) \not\subseteq \mathcal{I}(R)$.
- $E \equiv (t \in R)$. As in the previous case, *A* is then the answer provided by the user to the question *"Does R include a tuple of the form t?"*. If the function *processAnswer* introduces the fact *state(*$(t \in R)$*,valid)*, this implies that the answer of the user was *yes*, meaning that $t \in \mathcal{I}(R)$. The function *processAnswer* introduces the fact *state(*$(t \in R)$*,nonvalid)* when the answer of the user is *no*, meaning that $t \notin \mathcal{I}(R)$.
- $E \equiv$ *(all R)*. This input parameter corresponds to calls obtained in two different situations:
  1. As in the previous cases, when the debugger obtains the user answer to a question, in this case *"Is S the intended answer for R?"*, with *S* $= \mathcal{SQL}$*(select * from R)* $= \mathcal{SQL}(R)$. Then if *processAnswer* introduces the fact *state((all R),valid)* this implies that the answer of the user was *yes*, meaning that $\mathcal{SQL}(R) = \mathcal{I}(R)$. Analogously, a fact *state((all R),nonvalid)* introduced by *processAnswer* implies that the answer was either *no*, *missing(t)* or *wrong(t)*. All these cases mean that $\mathcal{SQL}(R) \neq \mathcal{I}(R)$ (see Definition 4).
  2. In a recursive call produced by *processAnswer*. It is easy to check that only one recursive call can occur, due to the change in the first parameter to *(all R)* (which avoids further recursive calls). That is, a first call occurs containing the answer provided by the user, and the execution of this call starts a recursive call, which does not call *processAnswer* recursively. The recursive calls are located in three points of Code 3:

- Line 8. Then, the initial call is *processAnswer((s $\in$ R),no)*, which has produced a fact *state((s $\in$ R),nonvalid)*. As explained above this implies

$$s \notin \mathcal{I}(R) \tag{34}$$

Also, the condition of the *if* statement preceding the recursive call ensures that

$$s \in \mathcal{SQL}(R) \tag{35}$$

The recursive call is *processAnswer((all R), wrong(s))*, and the *state* fact considered is *state((all R),nonvalid)$\leftarrow$ true)*. Then we must prove that $\mathcal{SQL}(R) \neq \mathcal{I}(R)$, and this is straightforward from (34) and (35).
- Line 10. The initial call must be *processAnswer((s $\in$ R),yes)*, and this call has introduced a fact *state((s $\in$ R),valid)*, which means:

$$s \in \mathcal{I}(R) \tag{36}$$

The condition of the *if* statement indicates that

$$s \notin \mathcal{SQL}(R) \tag{37}$$

The recursive call is *processAnswer((all R),missing(s))*, and the fact we are analyzing is hence *state((all R),nonvalid)$\leftarrow$ true)* or simply *state((all R),nonvalid))*. Then the expected result $\mathcal{SQL}(R) \neq \mathcal{I}(R)$ follows from (36) and (37).
- Line 13. The first call must be *processAnswer((V $\subseteq$ R), wrong(s))*. This is one of the cases already analyzed, where *wrong(s)* is the answer provided by the user for the enquiry *(V $\subseteq$ R)*. Now, observe that this enquiry must correspond to the election of an atom *state((V $\subseteq$ R), ...)* already ocurring in the program. Such atoms are introduced in line 8 of Code 6. The view V has been created by function *relevantTuples* (Code 7), and by Lemma 1:

$$\mathcal{SQL}(V) \subseteq \mathcal{SQL}(R) \tag{38}$$

This first call *processAnswer((V $\subseteq$ R), wrong(s))* has introduced a fact *state((V $\subseteq$ R),nonvalid)*, and we have already prove that this implies:

$$\mathcal{SQL}(V) \nsubseteq \mathcal{I}(R) \tag{39}$$

which, combined with 38 means:

$$\mathcal{SQL}(R) \nsubseteq \mathcal{I}(R) \tag{40}$$

The recursive call is *processAnswer((all R), wrong(s))*, which introduces the fact *state((all R),nonvalid))*. We must prove that $\mathcal{SQL}(R) \neq \mathcal{I}(R)$, which is a direct consequence of (40).

■

Next we study the completeness of the technique.

Theorem 2. *Completeness.*
Let R be a relation, and A the answer obtained after the call to askOracle(all R) *in line 1 of Code 1. If A is of the form* nonvalid*,* wrong(t) *or* missing(t)*, then the call* debug(R) *(defined in Code 1) returns a list L containing at least one relation.*

Proof. The while loop in Code 1 stops when an atom *buggy(R')* can be inferred from the program in variable *P*. When this happens, the return in line 9 collects all the *buggy* atoms, and thus the result contains at least *R'*).

In order to complete the proof we must check that the while loop always terminates. The result is a consequence of the following auxiliary result:

"Given a call *debug(R)* there is constant *k* such that the program P always have less than *k* clauses"

The reason is that this means that:

- The program *P* is finite, there is a maximum number of clauses *buggy* that can be introduced. And a finite, non-recursive, ground logic program is always terminating for any goal. This means that the goal *buggy(R)* in *getBuggy(P)* is terminating.
- The initial set of clauses represents the computation tree introduced in [4]. In particular a node for a relation *R* in the tree is buggy if and only if it corresponding *buggy* clause can be satisfied in *P*. The first user answer means that the root of the tree is nonvalid, and due to the general completeness result this means that the computation tree has a buggy node. Since *P* is finite, its number of enquiries is also finite, and therefore after some questions to the user the buggy clause corresponding to the buggy node will be found. Observe that the new clauses added during the process are shortcuts for finding the error with less questions, but that the original set of clauses is kept during the process.

■

Lemma 5. *Given a call* debug(R)*, there is a constant k such that the program* P *in Code 1 always have less than k clauses.*

Proof. Originally the number of clauses with head *buggy* is the number of nodes in the dependency tree. During the execution of the algorithm new facts for predicate *state* are added, and also new clauses with head *buggy* are included. The maximum number of possible *state* facts correspond to the number of *state* atoms in the body of clauses for *buggy*. Therefore it is enough to prove that there is a maximum number k' of clauses for *buggy*.

Now, observe that the new clauses are introduced by functions *missingBasic* and *wrongBasic*. These functions are called by *slice*. The maximum number of calls correspond to the number of basic query components in the set of relations.

Moreover, it can be checked that both *missingBasic* and *wrongBasic* can generate a new clause for each relation in the from section of the input query Q.

Therefore:

- Let n be the number of nodes in the dependency tree rooted by R. Notice that n is finite number, because views cannot be mutually recursive and we are assuming a finite database schema.
- Let q be the total number of basic components occurring in the queries of views that appear in the dependency tree of R.
- Let p be the maximum number of relations occurring in the *from* clause of any basic components mentioned in the previous items.

Then we can take k' as $n+(q\times p)$, that is the original number of *buggy* clauses plus the number of clauses generated by the basic components. It is worth observing that either *missingBasic* or *wrongBasic* might generate a clause for a given relation in the from section of a basic component, but never both of them.

Finally, if we consider m the maximum number of relations defining any view in the dependency tree, we can take $k = k' + m \times k'$, that is a maximum of k' clauses for *buggy* plus a maximum of $m \times k'$ facts for *state*. ■

Thus, the algorithm always stops pointing to some user view (completeness) which is incorrectly defined (correctness).

5 Implementation

The algorithm presented in Section 3 has been implemented in the Datalog Educational System (DES [12, 13]), which makes it possible for Datalog and SQL to coexist as query languages for the same database. The debugger is started when the user detects that *Anna* is not among the (large) list of student names produced by view *awards*. The command `/debug_sql` starts the session:

```
1: DES-SQL> /debug_sql awards
2: Info: Debugging view 'awards': {   1 - awards('Carla'), ... }
3: Is this the expected answer for view 'awards'?  m'Anna'
4: Does the intended answer for 'intensive' include  ('Anna')        ? n
5: Does the intended answer for 'standard'  include  ('Anna',1,true) ? y
6: Does the intended answer for 'standard'  include  ('Anna',2,true) ? y
7: Does the intended answer for 'standard'  include  ('Anna',3,false)? y
8: Info: Buggy relation found: intensive
```

The user answer *m'Anna'* in line 3 indicates that *('Anna')* is missing in the view *awards*. In line 4 the user indicates that view *intensive* should not include *('Anna')*. In lines 5, 6, and 7, the debugger asks three simple questions involving the view *standard*. After checking the information for *Anna*, the user indicates that the listed tuples are correct. Then, the tool points out *intensive* as the buggy view, after only five simple questions. Observe that intermediate views can contain hundreds of thousands of tuples, but the slicing mechanism helps to focus only on the source of the error. Next, we describe briefly how these questions have been produced by the debugger.

After the user indicates that *('Anna')* is missing, the debugger executes a call *processAnswer(all(awards),missing((Anna)))*. This implies a call to *slice(awards,* Q_1 except Q_2*, missing(('Anna')))* (line 16 of Code 3). The debugger checks that Q_2 produces *('Anna')* (line 7 of Code 4), and proceeds with the recursive call *slice(awards,* Q_2*, wrong(('Anna')))* with $Q_2 \equiv$ select student from intensive. Query Q_2 is basic, and then the debugger calls *wrongBasic(awards,* Q_2*, ('Anna'))* (line 10 of Code 4)). Function *wrongBasic* creates a view that selects only those tuples from *intensive* producing the wrong tuple *('Anna')* (function *relevantTuples* in Code 7):

```
create view intensive_slice(student) as
(select * from intensive)
intersect all
(select * from intensive I where I.student = 'Anna');
```

Finally the following buggy clause is added to the program P (line 8, Code 6):

```
buggy(awards) :- state(subset(intensive_slice,intensive),valid).
```

By enabling development listings with the command `/development on`, the logic program is also listed during debugging. The debugger chooses the only body atom in this clause as next unsolved enquiry, because it only contains one tuple. The call to *askOracle* returns *wrong(('Anna'))* (the user answers *'no'* in line 4). Then *processAnswer(subset(intensive_slice,intensive), wrong(('Anna')))* is called, which in turn calls to *processAnswer(all(intensive),wrong(('Anna')))* recursively. Next call is *slice(intensive, Q, wrong(('Anna')))*, with $Q \equiv Q_3$ union Q_4 the query definition of *intensive* (see Figure 1). The debugger checks that only Q_4 produces *('Anna')* and calls to *slice(intensive,* Q_4*, wrong(('Anna')))*. Query Q_4 is basic, which implies a call to *wrongBasic(intensive,* Q_4*, ('Anna'))*. Then *relevantTuples* is called three times, one for each occurrence of the view *standard* in the from section of Q_4, creating new views:

```
create view standard_slicei(student,level,pass) as
   ( select R.student, R.level, R.pass from standard as R)
       intersect all
   (select  A1.student, A1.level, A1.pass
    from standard as A1,  standard as A2, standard as A3
    where (A1.student = A2.student and A2.student = A3.student
       and A1.level = 1 and A2.level = 2 and A3.level = 3)
       and A1.student = 'Anna');
```

for $i = 1 \dots 3$. Finally, the clause:

```
buggy(intensive) :- state(subset(standard_slice1,standard),valid),
                    state(subset(standard_slice2,standard),valid),
                    state(subset(standard_slice3,standard),valid).
```

is added to P (line 8, Code 6). Next, the tool selects the unsolved question with less complexity that correspond to the questions of lines 5, 6, and 7, for which the user answer *yes*. Therefore, the clause for buggy(intensive) succeeds and the algorithm finishes.

The current implementation of our proposal, including instructions about how to use it, can be downloaded from
`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/DesSQL`.

6 Conclusions

We have presented a new technique for debugging systems of SQL views. Our proposal refines the initial idea presented in [4] by taking into account information about *wrong* and *missing* answers provided by the user. Using a technique similar to dynamic slicing [1], we concentrate only in those tuples produced by the intermediate relations that are relevant for the error. This minimizes the main problem of the technique presented in [4], which was the huge number of tuples that the user must consider in order to determine the validity of the result produced by a relation. The algorithm proposed looks for particular but common error sources, like tuples missed in the from section or in *and* conditions (that is, *intersect* components in our representation). If such shortcuts are not available, or if the user only answers *yes* and *no*, then the tools works as a pure declarative debugger.

A more general contribution of the paper is the idea of representing a declarative debugging computation tree by means of a set of logic clauses. In fact, the algorithm in Code 1 can be considered a general debugging schema, because it is independent of the underlying programming paradigm. The main advantage of this representation is that it allows combining declarative debugging with other diagnosis techniques that can be also represented as logic programs. In our case, declarative debugging and slicing cooperate for locating an erroneous relation. It would be interesting to research the combination with other techniques such as the use of assertions.

References

1. H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25:246–256, June 1990.
2. ApexSQL Debug , 2011. `http://www.apexsql.com/sql_tools_debug.aspx/`.
3. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. A theoretical framework for the declarative debugging of datalog programs. In *International Workshop on Semantics in Data and Knowledge Bases SDKB 2008*, volume 4925 of *Lecture Notes in Computer Science.*, pages 143–159. Springer, 2008.
4. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Algorithmic Debugging of SQL Views. In *Proceedings of the 8th International Andrei Ershov Memorial Conference, PSI 2011*, volume 7162. Springer LNCS, 2012.
5. R. Caballero, F. López-Fraguas, and M. Rodríguez-Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Proc. FLOPS'01*, number 2024 in LNCS, pages 170–184. Springer, 2001.

6. S. Ceri and G. Gottlob. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Trans. Softw. Eng.*, 11:324–345, April 1985.
7. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.
8. P. W. Grefen and R. A. de By. A multi-set extended relational algebra: a formal approach to a practical issue. In *10th International Conference on Data Engineering*, pages 80–88. IEEE, 1994.
9. L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 3, 1997.
10. H. Nilsson. How to look busy while being lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.
11. Rapid SQL Developer Debugger, 2011. `http://docs.embarcadero.com/products/rapid_sql/`.
12. F. Sáenz-Pérez. Datalog Educational System v2.6, October 2011. `http://des.sourceforge.net/`.
13. F. Sáenz-Pérez. DES: A Deductive Database System. *Elec. Notes on Theor. Comp. Science*, 271, 2011.
14. E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.
15. J. Silva. A survey on algorithmic debugging strategies. *Advances in Engineering Software*, 42(11):976–991, 2011.
16. SQL, ISO/IEC 9075:1992, third edition, 1992.