Declarative Debugging of Wrong and Missing Answers for SQL Views

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

Technical Report SIC-10-12

[†]Dpto. de Sistemas Informáticos y Computación, UCM, Spain
[‡]Dpto. de Ingeniería del Software e Inteligencia Artificial, UCM, Spain
{rafa,fernan}@sip.ucm.es , ygarciar@fdi.ucm.es

April 2011
(Revised February 10, 2014)

Abstract. This report 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 logic 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 [20]) 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 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 [2,15] 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 [18], functional programming [14], functional-logic programming [6], and in deductive database languages [3]. The technique can be described as a general debugging schema [13] which starts when an *initial error symptom* is detected by the user. In the context of SQL views, the initial symptom corresponds to an unexpected result produced by a view. The debugger automatically builds a tree representing the erroneous computation that has produced the symptom. 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 query that produced the initial symptom. The children of each 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 be leveled at this proposal is that it can be difficult for the user to check the validity of the results. Indeed, the results returned by SQL 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.

```
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 A1.student from allInOneCourse A1  where A1.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

- 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 includes 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 SQL views 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 that have passed a basic level course (level 0). View *intensive* defines as intensive students those in the *allInOneCourse* table, together with the students that have completed the three initial levels. However, this view definition is erroneous: we have forgotten to check that the courses have been completed (flag *pass*). Finally, the main view *awards* selects those students who are registered 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 missing. 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 hence she is not an

intensive student. Thus, the result obtained by this query is nonvalid. A standard declarative debugger using for instance a top-down strategy [19], 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*; 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 report 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)?" The answer provided by the user can be either "Yes" or "No". In the case or "No" the user can point out a wrong tuple or a missing tuple. In the example, the user indicates that *Anna* is missing in *awards*. With this information, the debugger can:

- Reduce the number of questions directed at the user. Our technique considers only those relations producing/losing the wrong/missing tuple. In the example, the debugger checks that the result of *awards* depends on the content of *basic* and *intensive* and also that there is a tuple for *Anna* in both of them (*intensive* and *basic*). This means that either the view *awards* is erroneous or the tuple for *Anna* in *intensive* is wrong. Consequently the debugger disregards *basic* as a possible error source, 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 considering the validity of the rest of the tuples in this view.

Another novelty of our approach is that the computation tree is represented 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 [16].

The next Section presents some basic concepts used in the rest of the report. Section 3 introduces the debugging algorithm that constitutes the main contribution of this work. 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 Declarative Debugging in SQL: a first approach

In this Section, we summarize the main results of [4] by describing the basic concepts of declarative debugging applied to SQL views. This first approach to debug SQL views will be refine in Section 3.

2.1 Basic Concepts of Relational Databases

A *relation schema* $\mathcal{R}$ consists of a list of attributes $(A_1, \ldots, A_n)$. Each attribute A_i has an associated domain (*integer*, *string*, ...) denoted as $dom(A_i)$.

The domain of $\mathcal{R}$ is defined as $dom(\mathcal{R}) = dom(A_1) \times \cdots \times dom(A_n)$. A *relation* or *relation instance* R of relation schema $\mathcal{R}$ is a multiset of elements in $dom(\mathcal{R})$.

A *tuple* t of schema $\mathcal{R}$ is an element in $dom(\mathcal{R})$. Duplicate tuples are allowed in a relation. The multiplicity of a tuple t in the relation instance R is denoted as $|R|_t$. A tuple t is an element of relation R if its multiplicity is greater than zero. In this case, we say that $t \in R$. If the multiplicity of t in R is zero, we say that $t \notin R$.

Each tuple t in the relation instance R can be considered as a function such that $dom(t) = \{A_1, \ldots, A_n\}$, with $t(A_i)$ the value of the attribute A_i in t. The number of attributes of t is denoted as $length(t)$. In general, we will be interested in tuples that combine attributes from different relation instances, usually as result of cartesian products. In this case, we qualify the attributes names by relation names using the notation $t(R.A_i)$ instead of $t(A_i)$.

The concatenation of two tuples t, s with disjoint domain is defined as the union of both functions represented as $t \odot s$. Given a tuple t and an arithmetic expression e defined on the attributes in $dom(t)$, we use the notation $e(t)$ to represent the value obtained applying the substitution t to e. Given a secuence $l = (e_1, \ldots, e_m)$ of arithmetic expressions defined on the attributes in $dom(t)$ with $m > 1$, the projection $\pi_l(t)$ is defined as a new tuple of the form $(e_1(t), \ldots, e_m(t))$.

The notation t_i represents the i-th position in the tuple. In our setting, *partial* tuples are tuples that might contain the special symbol $\perp$ in some of their components and *total* in other case. The set of defined positions of a partial tuple t, *def(t)*, is defined by $p \in$ *def(t)* $\Leftrightarrow t_p \neq \perp$. We say that $t =_\perp t'$ if *length(t)* = *length(t')* and $t_p = t'_p$ for every $p \in$ (*def(t)* $\cap$ *def(t')*). Membership with partial tuples is defined as follows: if t is a partial tuple, and S a set of tuples with the same number of positions as t, we say that $t \in_\perp S$ if there is a tuple $t' \in S$ such that $t =_\perp t'$. Otherwise we say that $t \notin_\perp S$.

In SQL, all data are stored and accessed via relations of a particular relation schema. Relations that store data are called *tables*. Other relations do not store data, but are computed by applying relational operations to other relations. These relations are called *views* or *queries*. In implementations, views can be thought of as new tables created dynamically from existing ones by using a SQL queries. The general syntax of a SQL view is: create view V(A_1, ..., A_n) as Q, with Q a query and $A_1, \ldots, A_n$ the names of the view attributes.

Consider a *database schema* D as a tuple $(\mathcal{T}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of tables and $\mathcal{V}$ is a finite set of views. A *database instance* d of a database schema D is a set of table instances T, with T in $\mathcal{T}$. Sometimes, we use the notation $d(T)$ to refers to the relation instance T in d.

The syntax of SQL queries can be found in [20]. 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 queries Q_1 and Q_2 by means of set operators union [all], except [all][1] or intersect [all] (the keyword all indicates that the result is a multiset). We 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 [7]). For convenience, our debugger transforms basic queries into compound queries when necessary. For instance, the query defining view *awards* in the Figure 1 is transformed into the following query:

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

The semantics of SQL assumed in this report is given by the Extended Relational Algebra (ERA) [12], an operational semantics allowing aggregates, views, and most of the common features of SQL queries.

In ERA, each relation R of relation schema $\mathcal{R}$ is defined as a multiset of tuples in $dom(\mathcal{R})$ and it is considered as a relational expression. Similar to the notation for multiset relations, the multiplicity of a tuple t in a multiset expression R is denoted as $|R|_t$. Multisets can be denoted as a collection of individual tuples t, possibly containing duplicates, or as a set of pairs $(t, |R|_t)$ without duplicates. For convenience, next definitions refer to multisets denoted as a set of pairs $(t, |R|_t)$. ERA defines new relations by means multiset expressions. These expressions combine previously defined relations using set and multiset operators.

Next we introduce some of the operators needed to understand our work (see [10,12] for a formal definition of each operator). Let M_1 and M_2 be relational expressions of relation schema $\mathcal{R}$ and let M_3 be a relational expression of relation schema $\mathcal{R}'$. Then the following operations are relational expressions:

- The *union* $M_1 \cup_{\mathcal{M}} M_2$ collects the elements of M_1 and M_2 into a new multiset of tuples in $dom(\mathcal{R})$:

$$M_1 \cup_{\mathcal{M}} M_2 = \{(t, |M_1|_t + |M_2|_t) \mid t \in dom(\mathcal{R})\}$$

- The *intersection* $M_1 \cap_{\mathcal{M}} M_2$ produces a new multiset of tuples in $dom(\mathcal{R})$ consisting of the elements that are both in M_1 and M_2:

$$M_1 \cap_{\mathcal{M}} M_2 = \{(t, min(|M_1|_t, |M_2|_t) \mid t \in dom(\mathcal{R})\}$$

- The *difference* $M_1 \setminus_{\mathcal{M}} M_2$, "subtract" the contents of M_2 from the contents of M_1 into a new multiset of tuples in $dom(\mathcal{R})$:

$$M_1 \setminus_{\mathcal{M}} M_2 = \{(t, max(0, |M_1|_t - |M_2|_t) \mid t \in dom(\mathcal{R})\}$$

- The *cartesian product* $M_1 \times M_3$, forms the cartesian product of the elements of M_1 and M_3 producing the new multiset of tuples in $dom(\mathcal{R}) \odot dom(\mathcal{R}')$:

$$M_1 \times M_3 = \{(t \odot t', M_1(t) \cdot M_3(t')) \mid t \in dom(\mathcal{R}),\ t' \in dom(\mathcal{R}')\}$$

[1] The Oracle database systems uses MINUS instead of the SQL standard EXCEPT.

– The *selection* $\sigma_{\varphi}(M_1)$ selects from a multiset that meet a condition φ defined on individual tuples in $dom(\mathcal{R})$, producing the following new multiset of tuples in $dom(\mathcal{R})$:

$$\sigma_{\varphi}(M_1) = \{(t, |M_1|_t) \mid t \in dom(\mathcal{R}) \wedge \varphi(t)\} \cup \{(t, 0) \mid t \in dom(\mathcal{R}) \wedge \neg\varphi(t)\}$$

In this case, φ can be seen as a function from $dom(\mathcal{R})$ into the boolean domain.

– The *projection* $\Pi_l(M_1)$ projects a multiset M_1 on the elements in the sequence l, where l can contain attributes from M_1 as well as arithmetic expressions defined on the attributes from M_1. These arithmetic expressions can be seen as functions from $\mathcal{R}$ into a basic domain. The result is a new multiset with schema $\Pi_l(\mathcal{R})$:

$$\Pi_l(M_1) = \{(s, \Sigma_{\varphi(t')}(|M_1|_{t'})) \mid t \in dom(\mathcal{R})\}$$

where $s = \pi_l(t)$ and $\varphi(t') \equiv t' \in dom(\mathcal{R}) \wedge \pi_l(t') = \pi_l(t)$. The summation $\Sigma_{\varphi(y)}(|M|_y)$ is to be interpreted as the sum of $|M|_y$ for all y satisfying the condition $\varphi(y)$.

– The *renaming* expression $\rho_R(M_1)$ returns a new multiset R with schema $\mathcal{R}$:

$$R = \{(t, |M_1|_t) \mid t \in dom(\mathcal{R})\}$$

The expression $\rho_{R(B_1/A_1,\ldots,B_n/A_n)}(M_1)$ returns a new multiset R with schema $(B_1, \ldots, B_n)$:

$$R = \{(s_t, |M_1|_t) \mid t \in dom(\mathcal{R})\}$$

where $dom(s_t) = \{B_1, \ldots, B_n\}$, $dom(B_i) = dom(A_i)$ and $s_t(B_i) = t(A_i)$
The rename operator is used to give a name as well as a new schema to relational expressions.

In our setting we allow the set operators union, intersection and difference. However the result of these operations is considered as a multiset.

– The union $M_1 \cup M_2$ collects the elements of M_1 and M_2 into a new multiset with schema $\mathcal{R}$:

$$M_1 \cup M_2 = \{(t, min(1, |M_1|_t + |M_2|_t)) \mid t \in dom(\mathcal{R})\}$$

– The intersection $M_1 \cup M_2$ produces a multiset with schema $\mathcal{R}$ consisting of the elements that are both in M_1 and M_2:

$$M_1 \cap M_2 = \{(t, min(1, |M_1|_t, |M_2|_t) \mid t \in dom(\mathcal{R})\}$$

– The *difference* $M_1 \setminus M_2$, "subtract" the contents of M_2 from the contents of M_1 into a new multi-set with schema $\mathcal{R}$:

$$\begin{aligned} M_1 \setminus M_2 = &\{(t, 1) \mid t \in dom(\mathcal{R}) \wedge |M_1|_t > 0 \wedge |M_2|_t = 0\} \cup \\ &\{(t, 0) \mid t \in dom(\mathcal{R}) \wedge |M_2|_t > 0\} \end{aligned}$$

In the rest of the report, we consider multisets denoted as a collection of individual tuples t, possibly containing duplicates.

We use the notation Φ_R for representing the ERA expression associated to a SQL relation R (table, query or view). For instance, in the case of SQL queries, the select, from and where sections correspond to the projection operation, cartesian product operation and selection condition of ERA respectively. The other sections such as group by and having corresponds to an aggregate expression in ERA as shown in [12]. The SQL set and multiset operators union [all], except [all] and intersect [all] correspond to the set and multiset operations in ERA. Tables are denoted by their names, that is, $\Phi_T = T$ if T is a table. SQL views are represented by means the rename operator of ERA.

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 relations $R_1, \ldots, R_n$.

Example 2. The ERA expressions associated with the views defined in Figure 1 are:

$$\Phi_{standard} = \rho_{standard(student/R.student,level/C.level,pass/R.pass)}(\\ \Pi_{R.student,C.level,R.pass}(\sigma_{C.id=R.course}(\rho_C(courses) \times \rho_R(registration))))$$

$$\Phi_{basic} = \rho_{basic(student/S.student)}(\ \Pi_{S.student}(\ \sigma_{S.level=0\ \wedge\ S.pass}(\ \rho_S(standard\ )\ )\ )\ )$$

$$\Phi_{intensive} = \rho_{intensive(student/A_1.student)}(\\ (\ \Pi_{A_1.student}(\ \sigma_{A_1.pass}(\ \rho_{A_1}(allInOneCourse\ )\ )\ )\ ) \\ \cup \\ (\ \Pi_{A_1.student}(\sigma_{Cond}(\rho_{A_1}(standard) \times \rho_{A_2}(standard) \times \rho_{A_3}(standard)))))$$

$$\text{where } Cond \equiv (\ A_1.student = A_2.student\ \wedge\ A_2.student = A_3.student \\ \wedge\ A_1.level = 1\ \wedge\ A_2.level = 2\ \wedge\ A_3.level = 3\ )$$

$$\Phi_{awards} = \rho_{awards}(\ \Pi_{student}(basic) \setminus\ \Pi_{student}(intensive)\ )$$

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, $\| \Phi_R \|_d = R$.*
- *If R is a database view or a query and R depends on the relations $R_1, \ldots, R_n$, then $\| \Phi_R \|_d = \Phi_R(M_1, \ldots, M_n)$, where $M_i = \| \Phi_{R_i} \|_d$ for $i = 1 \ldots n$.*

meaning that the computed answer of an expression Φ_R in the instance d is the result of evaluating the expression Φ_R after substituting each relation name R_i in Φ_R by its computed answer $\| \Phi_{R_i} \|_d$ for $i = 1 \ldots n$.

The parameter d indicating the database instance is omitted in the rest of the presentation whenever it 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 $\mathcal{SQL}(\mathsf{select\ *\ from}\ R)$. In particular, we assume in this report 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 report, D represents the database schema, d the current instance of D, and R a relation 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 relation* R *if* $\mathcal{I}(\mathrm{R}) \neq \mathcal{SQL}(\mathrm{R})$*.*

Observe that $\mathcal{I}(R) \neq \mathcal{SQL}(R)$ means that there is some tuple t such that $|\mathcal{I}(R)|_t \neq |\mathcal{SQL}(R)|_t$. The existence of an unexpected answer implies the existence of either a *wrong tuple* or a *missing tuple*.

Definition 5. *We say that* t *is a* wrong tuple *for a relation* R *if:*

$$|\mathcal{SQL}(R)|_{\mathrm{t}} > 0 \text{ and } |\mathcal{I}(\mathrm{R})|_{\mathrm{t}} < |\mathcal{SQL}(\mathrm{R})|_{\mathrm{t}}$$

Definition 6. *We say that* t *is a* missing tuple *for a relation* R *if:*

$$|\mathcal{I}(\mathrm{R})|_{\mathrm{t}} > 0 \text{ and } |\mathcal{I}(\mathrm{R})|_{\mathrm{t}} > |\mathcal{SQL}(\mathrm{R})|_{\mathrm{t}}$$

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

Definition 7. *Let* R *be either a query or a relation. The* expectable *answer for a relation* R *w.r.t. the instance* d*,* $\mathcal{E}(R, d)$*, is defined as:*

1. *If* R *is a table,* $\mathcal{E}(R, d) = R$*.*
2. *If* R *is a view, then* $\mathcal{E}(R, d) = \mathcal{E}(Q, d)$*, with* Q *the query defining* R*.*

3. *If R is a query and $R_1, \ldots, R_n$ the relations ocurring in R, then $\mathcal{E}(R, d) = \Phi_R(I_1, \ldots, I_n)$ where $I_i = \mathcal{I}(R_i)$ for $i = 1 \ldots n$, meaning that $\mathcal{E}(R, d)$ is the result of evaluating the expression Φ_R after substituting each relation name R_i in Φ_R by its intended answer $\mathcal{I}(R_i)$ for $i = 1 \ldots n$.*

In the rest of the report we use $\mathcal{E}(R)$ instead of $\mathcal{E}(R, d)$ if d is clear from the context. Thus, if R is a table, the expectable answer of R is the table instance R. In the case of a view V, the expectable answer corresponds to the expectable answer for the query Q defining V. In the case of a query Q, the expectable answer corresponds to the computed result that would be obtained assuming that all the relations R_i occurring in the definition of Q contain the intended answers.

A discrepancy between $\mathcal{I}(R)$ and $\mathcal{E}(R)$ indicates that R does not compute its intended answer, even assuming that all the relations it depends on correspond to their intended answers. Such relation is called *erroneous*.

Definition 8. *We say that a relation R is* erroneous *when $\mathcal{I}(R) \neq \mathcal{E}(R)$, and* correct *otherwise.*

In our running example, the intended answer for *awards* contains *Anna* once, which is represented as $|\mathcal{I}(\mathsf{awards})|_{(\text{'Anna'})} = 1$. As we said in the introduction, we know that *Anna* completed the basic course, that is the intended answer for *basic* contains *Anna* once, and that Anna is not an intensive student, that is the intended answer for *intensive* does not contain *Anna*. Then, following Definition 7, *Anna* is in $\mathcal{E}(awards)$ with multiplicity 1, which is represented as: $|\mathcal{E}(\mathsf{awards})|_{(\text{'Anna'})} = 1$. Then, following Definition 8, relation *awards* is correct. The real cause of the missing answer for the view *awards* is the erroneous definition of the view *intensive*.

Definition 8 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 $\mathcal{E}(V)$. Asking about $\mathcal{E}(V)$ to the users is unrealistic; we only can assume that they know $\mathcal{I}(V)$ but not $\mathcal{E}(V)$. Thus, the debugger requires from the user only to answer questions of the form *'Is the computed answer $\{| \ldots |\}$ the intended answer for view V?'*.

The following theorems relate the concept of erroneous relation and the concept of computed answer.

Theorem 1. *Let V be a database view and $R_1, \ldots, R_n$ the relations occurring in the query defining V such that $\mathcal{I}(R_i) = \mathcal{SQL}(R_i)$ for $i = 1 \ldots n$. Then, $\mathcal{SQL}(V)$ is unexpected iff V is erroneous.*

Proof. $\mathcal{SQL}(V)$ unexpected means that $\mathcal{I}(V) \neq \mathcal{SQL}(V)$. We prove that $\mathcal{E}(V) = \mathcal{SQL}(V)$ and the result $\mathcal{I}(V) \neq \mathcal{E}(V)$ holds. Let Q the query defining V. By Definitions 7 (2 and 3):

(1) $\qquad \mathcal{E}(V) = \Phi_V(I_1, \ldots, I_n)$ where $I_i = \mathcal{I}(R_i)$ for $i = 1 \ldots n$

By Definitions 2 and 1,

(2) $\mathcal{SQL}(V) =\| \Phi_V \| = \Phi_V(M_1, \ldots, M_n)$, where $M_i =\| \Phi_{R_i} \|$ for $i = 1 \ldots n$

By hypothesis and Definition 2

(3) $$\mathcal{I}(R_i) = \mathcal{SQL}(R_i) =\| \Phi_{R_i} \| = M_i \text{ for } i = 1 \ldots n$$

Then substituting the multiset I_i by the multiset M_i in (1) for $i = 1 \ldots n$, we obtain that:

(4) $$\mathcal{E}(V) = \Phi_V(M_1, \ldots, M_n)$$

By (2 and 4) we obtain that $\mathcal{E}(V) = \mathcal{SQL}(V)$ and the result $\mathcal{I}(V) \neq \mathcal{E}(V)$ holds. Then, by Definition 8, relation V is erroneous. □

Theorem 2. *Let T be a database table. Then, $\mathcal{SQL}(T)$ is unexpected iff T is erroneous.*

Proof. By Definition 4, $\mathcal{SQL}(T)$ unexpected means that:

(5) $$\mathcal{I}(T) \neq \mathcal{SQL}(T)$$

By Definitions 1 and 2,

(6) $$\mathcal{SQL}(T) = T$$

Combining (6) and Definition 7, item 1, $\mathcal{SQL}(T) = \mathcal{E}(T)$, and thus by (5), $\mathcal{E}(T) \neq \mathcal{I}(T)$. Then, by Definition 8, table T is erroneous. □

The debugging process is based on the results in Theorems 1 and 2. We emphasize the fact that the debugger requires from the user only to answer questions about the intended answer $\mathcal{I}(R)$, which is known by him, instead of the expectable answer $\mathcal{E}(R)$. In order to locate erroneous relations, the debugger compares the computed answer –obtained from the SQL system– and the intended answer –known by the user–.

2.3 Computation trees

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

Definition 9. *The computation tree $CT(R)$ associated with a relation R 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.*

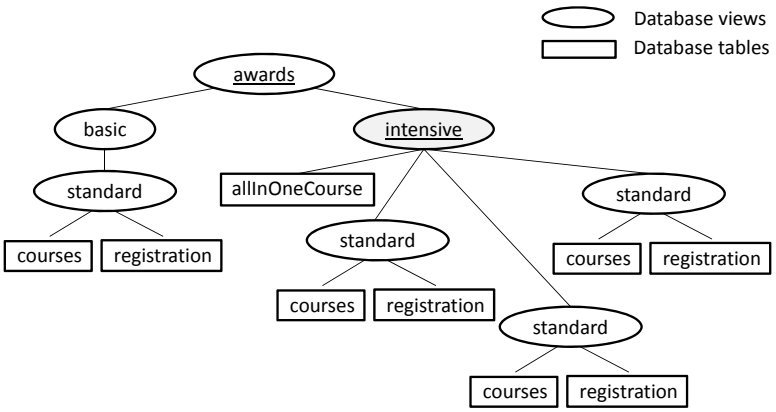

Fig. 2. Computation Tree associated with the view *awards*

- *If R' is a view, the children of N will correspond to the CTs for the relations occurring in the query associated with R'.*

Although Definition 9 includes the computed answer $\mathcal{SQL}(R)$ as part of nodes, this information is not relevant for the tree structure. In practice, this will lead to a non-efficient implementation in terms of memory usage. Instead, the computed answers are obtained from the SQL system by the debugger when needed. With this simplification, the computation tree corresponds to the dependency tree of the view R in the schema. Figure 2 shows the computation tree for our running example. After building the computation tree, the debugger will *navigate* the tree, asking the user about the validity of some nodes:

Definition 10. *Let $T = CT(R)$ be a computation tree, and $N = (R' \mapsto \mathcal{SQL}(R'))$ a node in T. We say that N is a* valid node *when $\mathcal{SQL}(R') = \mathcal{I}(R')$, a* nonvalid node *when $\mathcal{SQL}(R') \neq \mathcal{I}(R')$, and a* buggy node *when N is nonvalid and all its children in T are valid.*

The goal of the debugger is to locate buggy nodes. The next theorem shows that a computation tree with a nonvalid root always contains a buggy node, and that every buggy node corresponds to an erroneous relation.

Theorem 3. *Let V a database view and $CT(V)$ its associated computation tree. If the root of $CT(V)$ is nonvalid, then:*

- *Completeness. $CT(V)$ contains a buggy node.*
- *Soundness. Every buggy node in $CT(V)$ corresponds to an erroneous relation.*

Proof. The completeness is straightforward using induction on the number of nodes of $CT(V)$, see [13]. In order to prove the soundness, let $N = (R \mapsto \mathcal{SQL}(R))$ be a buggy node in $CT(V)$. We must prove R is erroneous. Since N is buggy, by definition 10,

$$\mathcal{SQL}(R) \neq \mathcal{I}(R) \tag{7}$$

– If R is a table, by (7) and Theorem 2, the result holds.
– If R is a view, we have that the children $N_1, \ldots, N_m$ of N correspond to the relations $R_1, \ldots, R_m$ occurring in the query associated with R. By definition 10, all of them are valid. Then, we have:

(8) $$\mathcal{I}(R_i) = \mathcal{SQL}(R_i) \text{ for } i = 1 \ldots m$$

By (7), (8) and Theorem 1, the result holds. □

2.4 Debugging session based on computation trees

Next we describe a simple debugging session based on computation trees and the result of the Theorem 3. A debugger technique following these ideas was presented in [4]. The debugger does not allow the user to point out wrong/missing tuples but it only allows the user to indicate the validity/nonvalidity of certain relations which correspond to nodes in the tree.

Suppose the user detects an unexpected result from view *awards* and he indicates to the debugger that the intended answers for table instances correspond to the actual database instance, thus trusting the instance. Then, he can type the command `/debug_sql` to debug the view *awards*, as follows:

```
DES-SQL> /debug_sql awards
```

The debugger builds internally the computation tree of the view *awards* (see Figure 2), and starts the debugging session:

```
Info: Debugging view 'basic'.
{
  basic('Anna'),
  basic('Carla'),
    ....
  basic('James')
}
Input: Is this view valid? (y/n/a) [y]: y

Info: Debugging view 'intensive'.
{
  intensive('Alba'),
  intensive('Anna'),
    ....
  intensive('James')
}

Input: Is this view valid? (y/n/a) [y]: n

Info: Debugging view 'standard'.
{
  standard('Alba',1,0),
  standard('Alba',2,1),
```

```
  standard('Anna',0,1),
  standard('Anna',1,1),
  standard('Anna',2,1),
  standard('Anna',3,0),
  standard('Carla',0,1),
  standard('James',0,1),
  standard('James',1,1),
    ...
  standard('James',3,1)
}

Input: Is this view valid? (y/n/a) [y]: y
Info: Buggy view found: intensive/1.
```

The debugger marks the root of the tree as a nonvalid node (the computed answer of the view *awards* is an unexpected answer). The first question is about the validity of *basic*. Suppose the user checks the answer produced by the SQL system as valid. In this case, the children of *basic* are not considered anymore. The second question is about the validity of the view *intensive*. The computed answer of the view *intensive* produced by the SQL system contains Anna, and the user checks this answer as nonvalid. Next, the children of *intensive* are visited. The only child of *intensive* that is not a table is *standard*. As this is checked as valid, the debugger points out node *intensive* as buggy; node *intensive* is marked as nonvalid and all its children are marked as a valid nodes. In the tree of Figure 2, nonvalid nodes are underlined and the only buggy node (a shader node) corresponds to view *intensive*. The tool has asked three questions to the user. However, note that these questions can be difficult to answer when the computed answer contains hundreds or thousands of tuples.

In the next Section we improve the debugging technique presented in this Section by allowing the user to specify if an unexpected answer contains a wrong or a missing tuple.

3 Improved Debugging Algorithm

This Section refines the ideas for debugging SQL views presented so far. Although the process is based on the ideas of declarative debugging, this proposal does not use computation trees explicitly. Instead, our debugger represents the logic inferences defining buggy nodes in 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 $\mathsf{H} \leftarrow \mathsf{true}$.

This Section starts off describing some auxiliary functions that define the debugging algorithm. Next, we present the general schema of the algorithm. This Section ends discussing how the debugger process the user answer for detecting errors.

3.1 Auxiliary functions

We describe some auxiliary functions that define the debugging algorithm, although the code of some basic auxiliary functions is omitted. Some of them are the following:

Code 1 debug(V)

```
Input: V: view name
Output: A list of buggy views
 1: A := askOracle(all V)
 2: if A ≡ no or A ≡ missing(t) or A ≡ wrong(t) then
 3:    Valid := true
 4:    P := initialSetOfClauses(V, A)
 5:    while getBuggy(P)=[] do
 6:       LE := getUnsolvedEnquiries(P)
 7:       E := chooseEnquire(LE)
 8:       A := askOracle(E)
 9:       Valid := checkAnswer(A)
10:       if Valid then P := P ∪ processAnswer(E,A)
11:    end while
12:    L := getBuggy(P)
13: else
14:    L := []
15: end if
16: return L
```

- Functions *getSelect* y *getWhere* return the different sections of a SQL query.
- The result of the function *getFrom* is a sequence of elements of the form *R as R'* (assuming that every relation R in the from section of a SQL query has an alias 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 the relation 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.
  - If R is a view, then *getRelations(R)* = *getRelations(Q)*, with Q the query defining R.
- Function *generateUndefined(R)* generates a new tuple with length the number of attributes in the relation R containing only undefined values $(\perp, \ldots, \perp)$.
- Function *checkAnswer(A)* returns the boolean value *true* if the input parameter is of the form *yes*, *no*, *missing(t)* or *wrong(t)* and *false* in other case.

3.2 Debugging schema

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 *no*, 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.

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)). }
```

The atoms in the body of the clauses represent *enquiries* that might represent questions to the user. The initial list of clauses P is generated by the function *initialSetofClauses* (line 4). This function 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 5-11) 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. Each iteration of the loop represents the election of an enquiry in a body atom whose validity has not been established yet (lines 6-7). Then, in line 8 the debugger asks to the user about the result of the question associated to the chosen enquiry. Finally, the answer is processed (line 10). Next, we explain in detail each part of this main algorithm.

Code 2 corresponds to the initialization process called in line 4 of 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 the function *initialize* (line 2), a function that traverses recursively all the relations involved in the definition of the initial view V, calling *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 depends on must be valid.

Figure 3 shows a partial list of initial clauses for our example. The correlation between these clauses and the computation tree of Figure 2 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 by predicate *state*. The first parameter of *state* 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:

```
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).
    ...
buggy(courses)      :- state(all(courses),nonvalid).
buggy(registration) :- state(all(registration),nonvalid).
```

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

Definition 11. *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 with 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 ≡ (R' ⊆ 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 answers allowed are* yes *or* no. *In the case of* no*, the user can point out a wrong tuple* t ∈ S *and the answer is changed to* wrong(t). $\mathcal{C}(E) = |S|$ *as in the previous case.*
- *If* E ≡ (s ∈ R). *Tuple* s *can be a partial tuple. The question is* "Does the intended answer for R include a tuple matching the tuple s?" *The possible answers are* 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* answers, the user typically points to a tuple in the result R. In the case of *missing* answers, 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 the body of a clause in P implies an enquiry q. The enquiry q 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 enquiry q is called an *unsolved enquiry* otherwise. The function *getUnsolvedEnquiries* (see line 6 of Code 1) returns in a list all the unsolved enquiries occurring in body atoms of clauses in P. The function *chooseEnquiry* (line 7, Code 1) chooses one of these enquiries according to some predefined criteria. Our current implementation chooses 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 8) in order to ask to the user the associated question, returning the answer.

3.3 Processing user answers and detecting errors

Code 3 processAnswer(E,A)

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

```
1: if A ≡ yes then
2:    P := {state(E,valid).}
3: else if A ≡ no or A ≡ missing(t) or A ≡ wrong(t) then
4:    P := {state(E,nonvalid).}
5: end if
6: if E ≡ (s ∈ R) and (A ≡ yes) then
7:    P:= P ∪ processAnswer((all R),missing(s))
8: else if E ≡ (V ⊆ R) and (A ≡ wrong(s) or A ≡ no) then
9:    P:= P ∪ processAnswer((all R), A)
10: else if E ≡ (all V) with V a view and (A ≡ missing(t) or A ≡ wrong(t)) then
11:    Q := SQL query defining V
12:    P := P ∪ slice(V,Q,A)
13: end if
14: return P
```

Function *processAnswer* process the user answer distinguishing several cases depending on the form of its associated enquiry. The code of function *processAnswer* (called in line 10 of Code 1 and in line 3 of Code 2), 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, *debug(awards)* calls to *initialSetofClauses(awards, missing(('Anna')))* which calls to *processAnswer(all(awards), missing(('Anna')))* which adds the fact *state(all(awards), nonvalid)* 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 ∈ R)* with answer *yes* (line 6), then s is missing in the relation R. Notice, the enquiry *(s ∈ R)* is associated to a body atom of the form *state((s ∈ R), nonvalid)* of a clause added in P by the function *missingBasic* when the debugger checks that the tuple s is not in the computed answer of the view R (Code 5, line 8).

For enquiries of the form *(V ⊆ R)* and answer *wrong(s)*, it can be ensured that s is wrong in R (line 9). If the enquiry is of the form *(V ⊆ R)* with answer *no*, then R is a nonvalid relation. 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 12) 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 might allow the debugger to detect incorrect relations by asking simpler questions to the user. The implementation of *slice* can be found 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). Then 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 4 and 5). The case of subqueries combined by the operator intersect [all], with A

Code 4 slice(V,Q,A)

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

```
 1: P := ∅;    S= SQL(Q);
 2: if (A ≡ wrong(t) and Q ≡ Q1 union [all] Q2) or
       (A ≡ missing(t) and Q ≡ Q1 intersect [all] Q2) then
 3:    S1= SQL(Q1);    S2= SQL(Q2);
 4:    if |S1|t = |S|t then  P:= P ∪ slice(V, Q1, A)
 5:    if |S2|t = |S|t then  P:= P ∪ slice(V, Q2, A)
 6: else if A ≡ missing(t) and Q ≡ Q1 except [all] Q2  then
 7:    S1= SQL(Q1);    S2= SQL(Q2);
 8:    if |S1|t = |S|t then  P:= P ∪ slice(V, Q1, A)
 9:    if Q ≡ Q1 except Q2 and t ∈⊥ S2 then P :=P∪ slice(V,Q2,wrong(t))
10: else if basic(Q) and groupBy(Q)=[] then
11:    if A ≡ missing(t) then P := P ∪ missingBasic(V, Q, t)
12:    else if A ≡ wrong(t) then P := P ∪ wrongBasic(V, Q, t)
13: end if
14: return  P
```

≡ *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 6). 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 8). 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 9), 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 10), 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 11 of Code 4) when A is *missing(t)*. The input parameters are the view V, a query Q, and the missing 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 of Q, 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),

Code 5 missingBasic(V,Q,t)

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

```
 1: P := ∅;    S := SQL(SELECT getSelect(Q) FROM getFrom(Q) )
 2: if t ∉⊥ S then
 3:    for (R AS S) in (getFrom(Q)) do
 4:       s = generateUndefined(R)
 5:       for i=1 to length(getSelect(Q)) do
 6:          if t_i ≠⊥ and member(getSelect(Q),i) = S.A, A attrib., then s(R.A) = t_i
 7:       end for
 8:       if s ∉⊥ SQL(R) then
 9:          P := P ∪ { (buggy(V) ← state((s ∈ R), nonvalid).) }
10:       end if
11:    end for
12: end if
13: 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
```

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 correctness and the completeness of the proposal. In the rest of the Setion 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 $\mathsf{P} \vdash \mathsf{A}$ denotes that there is a SLD proof for A with respect to the program 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'.A_1, ..., R'.A_n from R_i as R')
        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
```

4.1 Correctness

We start checking the correctness of the framework.

Theorem 4. *Correctness.*
Let R be a relation and L the list returned by debug(R) (defined in Code 1). If the user answers correctly all the questions performed by the debugger, then every relation contained in L is erroneous (according to Definition 8).

Proof. We prove the correctness of our technique using some auxiliary results (see Figure 4).

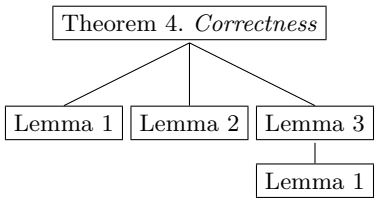

Fig. 4. The proof correctness structure

Lemma 1 proves properties of the new view created by function relevantTuples.
Lemma 2 establishes the relationship between enquiries and answers.
Lemma 3 indicates how state relates the answers obtained by the SQL system and the intended interpretation $\mathcal{I}$.

Let $\mathcal{P}$ be the logic program contained in the variable P of Code 1. Then the list L returned by debug(R) contains all the database relations A such that:

$$\mathcal{P} \vdash \mathit{buggy(A)} \tag{9}$$

Let A be any of the relations verifying (9). We prove that A is erroneous according to Definition 8.

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

1.- In Code 2 (function *createBuggyClause*, line 2). In this point, the added clause is:

– If A is a view:

$\mathsf{buggy(A)}\leftarrow \mathsf{state((all\ A),\ nonvalid)}$,$\mathsf{state((all}\ R_1\mathsf{),\ valid)}$,...,$\mathsf{state((all}\ R_n\mathsf{),\ valid)}$.

where $R_1, \ldots, R_n$ are all the relations employed in the definition of the view A. Then, by Lemma 3:

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

(11) $\quad \mathcal{P} \vdash \textit{state((all } A\textit{), nonvalid)} \Longrightarrow \mathcal{SQL}(A) \neq \mathcal{I}(A)$

By (11) and Definition 4, $\mathcal{SQL}(A)$ is unexpected, which means, by (10) and Theorem 1, that A is erroneous.

– If A is a table, the added clause is:

$$\mathsf{buggy(A)}\leftarrow \mathsf{state((all\ A),\ nonvalid)}.$$

Then, by Lemma 3:

(12) $\quad \mathcal{P} \vdash \textit{state((all } A\textit{), nonvalid)} \Longrightarrow \mathcal{SQL}(A) \neq \mathcal{I}(A)$

By (12) and Definition 4, $\mathcal{SQL}(A)$ is unexpected and the result is straightforward from Theorem 2.

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

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

Then $\mathcal{P} \vdash \textit{state((}\mathrm{s} \in \mathrm{R}\textit{), nonvalid))}$, and:

(13) By Code 5, line 8, $\quad s \notin_{\perp} \mathcal{SQL}(R)$

(14) By Lemma 3, $\quad s \notin_{\perp} \mathcal{I}(R)$

The input parameters of the function *missingBasic* defined in Code 5 are the view A, a query Q and a tuple t indicating that t is missing in A. Notice that the query Q is in general a component of the query defining the view A. The function *missingBasic* is called from function *slice* (Code 4) which ensures that Q is defined by a basic query without group by section. Next we prove that $|\mathcal{E}(Q)|_t \leq |\mathcal{SQL}(Q)|_t$ and by applying Lemma 2 we conclude that A is an incorrect view.

Examining the code of the function *missingBasic* it is clear that R AS S is in the from section and the *select* section contains $m \geq 1$ values of the form $S.A_1, \ldots, S.A_m$ with tuple s of the form: $s(R.A_i) = t(S.A_i) \neq \perp$ for $i = 1 \ldots m$ (otherwise the tuple s will be completely undefined and the condition $s \notin_{\perp} \mathcal{SQL}(R)$ could not hold).

Therefore, the ERA expression associated to the query Q can be written as

(15) $$\Phi_Q = \prod\nolimits_{(\ldots S.A_1,\ldots,S.A_m \ldots)}(\sigma_C(\cdots \times \rho_S(R) \times \ldots))$$

for some condition C in the where section of Q. Then, by Definitions 1 and 2:

(16) $$\mathcal{SQL}(Q) = \| \Phi_Q \| = \prod\nolimits_{(\ldots S.A_1,\ldots,S.A_m \ldots)}(\sigma_C(\cdots \times \rho_S(M) \times \ldots))$$

with $M = \mathcal{SQL}(R)$. By (13), $s \notin_{\perp} M$, that is there is no tuple $u \in M$ such that $s =_{\perp} u$. Therefore, there is no tuple $u \in M$ such that $u(R.A_i) = t(S.A_i)$ for $i = 1 \ldots m$. Then:

$$t \notin_{\perp} \mathcal{SQL}(Q) \tag{17}$$

Applying Definition 7 to (15) we obtain:

$$\mathcal{E}(Q) = \Pi_{(\ldots S.A_1,\ldots,S.A_m \ldots)}(\sigma_C(\cdots \times \rho_S(I) \times \ldots)) \tag{18}$$

with $I = \mathcal{I}(R)$. By (14), $s \notin_{\perp} I$, that is there is no tuple $u' \in I$ such that $s =_{\perp} u'$. Therefore, there is no tuple $u' \in I$ such that $u'(R.A_i) = t(S.A_i)$ for $i = 1 \ldots m$. Then:

$$t \notin_{\perp} \mathcal{E}(Q) \tag{19}$$

By (17) and (19):

$$|\mathcal{E}(Q)|_t = |\mathcal{SQL}(Q)|_t = 0 \tag{20}$$

The idea is that if the relation R does not contain a tuple matching s in all its defined attributes, then it is not possible to obtain the tuple t in $\mathcal{SQL}(Q)$ from R. Taking into account that the function *missingBasic* has been called from *slice* (Code 4, line 11), with the same input parameters A, Q, and a missing tuple t, then Lemma 2 can be applied to the call *slice(A,Q,missing(t))*, and (20) implies that A is an incorrect relation.

3.- In Code 6, line 8, which introduces the clause:

$$(\mathsf{buggy(A)} \leftarrow \mathsf{state}((\mathsf{V}_1 \subseteq \mathsf{R}_1), \mathsf{valid}), \ldots, \mathsf{state}((\mathsf{V}_n \subseteq \mathsf{R}_n), \mathsf{valid}))$$

where V_i is a new database view returned by function *relevantTuples* (called in Code 6, line 6). Then $\mathcal{P} \vdash$ *state((*$\mathrm{V}_i \subseteq \mathrm{R}_i$*), valid))*, for i=1 ... n.

By Lemma 3:

$$\mathcal{SQL}(V_i) \subseteq \mathcal{I}(R_i) \qquad \text{for i=1 ... n} \tag{21}$$

Analogously to the previous case, the function *wrongBasic* is called from function *slice* (Code 4, line 12). The input parameters of the function *wrongBasic* defined in Code 6 are the view A, a query component Q of the query defining the view A (Q is defined by a basic query without group by section) and a wrong tuple t. The ERA expression of Q is of the form:

$$\Phi_Q = \Pi_{(S)}(\sigma_C(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_n}(R_n))) \tag{22}$$

where S is the list of expressions in the select section, C is the condition in the where section and R_1 AS S_1, ..., R_n AS S_n is the sequence of elements in the from section of the query Q. Using Definitions 1 and 2 we obtain the SQL computed answer

$$\mathcal{SQL}(Q) = \| \Phi_Q \| = \Pi_{(S)}(\sigma_C(\rho_{S_1}(M_1)) \times \cdots \times \rho_{S_n}(M_n))))$$

with $M_i = \mathcal{SQL}(R_i)$ for $i = 1 \ldots n$, and in particular:

$$|\mathcal{SQL}(Q)|_t = |\Pi_{(S)}(\sigma_C(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_n}(M_n))))|_t \tag{23}$$

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

$$|\mathcal{SQL}(Q)|_t = |\Pi_{(S)}(\sigma_C(\rho_{S_1}(M'_1)) \times \cdots \times \rho_{S_i}(M'_i) \times \cdots \times \rho_{S_n}(M'_n)))|_t \tag{24}$$

with $M'_i = \mathcal{SQL}(V_i)$.

By Definition 7, item 3, we obtain:

$$\mathcal{E}(Q) = \Pi_{(S)}(\sigma_C(\rho_{S_1}(I_1)) \times \cdots \times \rho_{S_n}(I_n))))$$

with $I_i = \mathcal{I}(R_i)$ for $i = 1 \ldots n$. And in particular:

$$|\mathcal{E}(Q)|_t = |\Pi_{(S)}(\sigma_C(\rho_{S_1}(I_1)) \times \cdots \times \rho_{S_n}(I_n))))|_t \tag{25}$$

It is easy to check that in an expression like (24), replacing a multiset M'_i in the cartesian product by other multiset W such that $M'_i \subseteq W$ implies at least the same tuples in the result (and possibly more, new tuples). Therefore, applying (21) to (24) and (25):

$$|\mathcal{SQL}(Q)|_t \leq |\mathcal{E}(Q)|_t \tag{26}$$

Taking into account that the function *wrongBasic* has been called from *slice* (Code 4, line 12), with the same input parameters A, Q, and a wrong tuple t, then Lemma 2 can be applied to the call *slice(A,Q,wrong(t))*, and (26) implies that A is an incorrect relation. □

Next we prove some auxiliary Lemmata. First we prove a property of the new view created by function *relevantTuples*.

Lemma 1. *After a call of the form* relevantTuples(R_i,S_i,V,Q,t), *V is a new view such that*

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

Proof.

Function *relevantTuples* is called from function *wrongBasic*. The input parameters are a query Q, a (partial or total) tuple t of the form $(t_1, \ldots, t_k)$, a relation R_i occurring in the from section of the query Q and its associated alias S_i. The output is a new auxiliary view V in the database schema containing only those tuples from R_i that contribute to produce the tuple t in the result of the query Q. The definition of V can be found in Code 7. Suppose that:

- *getFrom(Q)* = R_1 *as* S_1, ..., R_m *as* S_m,
- *getWhere(Q)* = C,
- *getSelect(Q)* = e_1, ..., e_k
- C' is the SQL condition representing the logical expression: $\bigwedge_{\substack{1 \leq j \leq k \\ t_j \neq \perp}} (e_j = t_j)$

Then, the definition of V can be represented as:

(27)

```
create view V as
  (select Si.A1, ..., Si.An from Ri as Si )
      intersect all
  (select Si.A1, ..., Si.An
   from R1 as S1, ..., Ri as Si, ..., Rm as Sm
    where C and C')
```

1. Taking into account that $R_i.A_1, \ldots, R_i.A_n$ are all the attributes of R_i, the definition of V from (27) can be represented by the following ERA expression as intersection of multisets:

   (28) $$\rho_V(\rho_{S_i}(R_i) \cap_{\mathcal{M}} \Pi_{S_i.A_1,\ldots,S_i.A_n}(\sigma_{C \wedge C'}(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_m}(R_m))))$$

   and therefore $\mathcal{SQL}(V) \subseteq \mathcal{SQL}(R_i)$.

2. The function *relevantTuples* is called from function *wrongBasic* (line 6, Code 6), which is called from function *slice* (Code 4, line 12). The *if* sentence in *slice* ensures that Q is a basic query without group by section. Therefore, Q must be of the form:

   ```
   select e1, ..., ek
   from R1 as S1, ..., Ri as Si, ..., Rm as Sm
   where C
   ```

   which can be represented in ERA as:

   (29) $$\Phi_Q = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_i}(R_i) \times \cdots \times \rho_{S_m}(R_m)))$$

   Let $\Phi_{Q'}$ be the ERA expression obtained by replacing R_i by V in (29):

   (30) $$\Phi_{Q'} = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_i}(V) \times \cdots \times \rho_{S_m}(R_m)))$$

   Observe that only the from section needs to be modified, because the rest of the query does not include R_i but his alias S_i, and aliases are kept unaltered in $\Phi_{Q'}$. Then:
   Using Definitions 1 and 2 we obtain the SQL computed answer of Q and Q':

   (31) $$\| \Phi_Q \| = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M_i) \times \cdots \times \rho_{S_m}(M_m)))$$

   (32) $$\| \Phi_{Q'} \| = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M) \times \cdots \times \rho_{S_m}(M_m)))$$

   with $M_j = \mathcal{SQL}(R_j)$ for $j = 1 \ldots m$, and $M = \mathcal{SQL}(V)$.
   Then we must prove that $|\mathcal{SQL}(Q)|_t = |\mathcal{SQL}(Q')|_t$. In other words, by Definition 2, we must prove

   $$| \| \Phi_Q \| |_t = | \| \Phi_{Q'} \| |_t$$

   Let U be the minimum multiset such that

   (33) $$U \subseteq \rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M_i) \times \ldots \rho_{S_m}(M_m))$$

   (34) $$| \| \Phi_Q \| |_t = |\Pi_{e_1,\ldots,e_k}(\sigma_C(U))|_t$$

(minimum here means that is if we remove from U any occurrence of any of its tuples, (34) becomes false). Since the difference between (31) and (32) is only the replacement of M_i by M, we can concentrate on this part of the tuples of U. Observe that $M_i = \mathcal{SQL}(R_i)$. Hence we define:

$$U_{R_i} = \{(u, |M_i|_u) \mid u = \pi_{S_i.A_1,\ldots,S_i.A_n}(w), w \in U\}$$

By construction of U_{R_i}, we obtain:

$$U_{R_i} \subseteq M_i = \mathcal{SQL}(R_i) \tag{35}$$

The proof is complete is we check that $\rho_V(U_{R_i}) = M$.
By construction of U:

$$U = \sigma_{C \wedge C'}(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M_i) \times \cdots \times \rho_{S_m}(M_m))$$

and from this and the construction of U_{R_i}:

$$U_{R_i} = \Pi_{S_i.A_1,\ldots,S_i.A_n}(\sigma_{C \wedge C'}(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M_i) \times \cdots \times \rho_{S_m}(M_m)))$$

which considering that:

$$M = \rho_V(\rho_{S_i}(M_i) \cap_{\mathcal{M}} \Pi_{S_i.A_1,\ldots,S_i.A_n}(\sigma_{C \wedge C'}(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_m}(M_m))))$$

yields:

$$M = \rho_V(\rho_{S_i}(M_i) \cap_{\mathcal{M}} U_{R_i})$$

and by (35) we obtain $M = \rho_V(U_{R_i})$. □

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

Lemma 2. *Let* slice(V,Q,A) *be any call to Code 4 that occurs during the execution of the debugger. Then:*

- *If* A ≡ *wrong(t) and* $|\mathcal{E}(Q)|_t \geq |\mathcal{SQL}(Q)|_t$, *then* V *is an incorrect view.*
- *If* A ≡ *missing(t) and* $|\mathcal{E}(Q)|_t \leq |\mathcal{SQL}(Q)|_t$, *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 12. This call ensures that V is a view, Q is 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 Definition 7, $\mathcal{E}(\mathsf{V}) = \mathcal{E}(\mathsf{Q})$, we have that $\mathcal{E}(\mathsf{V}) \neq \mathcal{I}(\mathsf{V})$ and according to the Definition 8, the view V is erroneous.

If $n > 0$ we suppose that the result holds for the n-th call slice(V,Q,A), and we want to check that it is also valid for the n+1-th call slice(V,Q',A'). Observe that all the recursive calls occur in Code 4 and verify that they do not change the first parameter 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 of the $n-1$ call, slice(V,Q,A). Now we check that the result can be applied also to V, Q' and A', distinguishing cases depending on the particular call:

- slice(V,Q,A) calls to slice(V,Q',A') in Code 4, Line 4. In this case, considering Q' as Q_1, we have:

  $$|\mathcal{SQL}(\mathsf{Q}_1)|_t = |\mathcal{SQL}(\mathsf{Q})|_t \tag{36}$$

  Then, one of the following conditions hold:
  - A ≡ missing(t) and Q ≡ Q_1 intersect Q_2.
    Let $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \cap \Phi_{\mathsf{Q}_2}$ be the ERA expression associated to the SQL query Q. By Definition 7, we have $\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \cap \mathcal{E}(\mathsf{Q}_2)$. Therefore,

    $$|\mathcal{E}(\mathsf{Q}_1)|_t \geq |\mathcal{E}(\mathsf{Q})|_t \tag{37}$$

    If $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, by (36) we obtain that $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$ and by (37), $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$. Then, by induction hypothesis, V is incorrect.
  - A ≡ missing(t) and Q ≡ 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.
  - A ≡ wrong(t) and Q ≡ Q_1 union Q_2.
    From $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \cup \Phi_{\mathsf{Q}_2}$, and by Definition 7 we have $\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \cup \mathcal{E}(\mathsf{Q}_2)$. Therefore,

    $$|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{E}(\mathsf{Q})|_t \tag{38}$$

    If $|\mathcal{E}(\mathsf{Q}_1)|_t \geq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, by (36) we obtain that $|\mathcal{E}(\mathsf{Q}_1)|_t \geq |\mathcal{SQL}(\mathsf{Q})|_t$ and by (38) $|\mathcal{E}(\mathsf{Q})|_t \geq |\mathcal{SQL}(\mathsf{Q})|_t$. By induction hypothesis we conclude that V is an incorrect view.
  - A ≡ wrong(t) and Q ≡ Q_1 union all Q_2. Analogous to the previous point. Observe that replacing the set operator $\cup$ by $\cup_{\mathcal{M}}$ does not affect to the result.
- slice(V,Q,A) calls to slice(V,Q',A') in Code 4, Line 5. Considering Q' as Q_2. This case is analogous to the previous case changing Q_1 by Q_2.
- slice(V,Q,A) calls to slice(V,Q',A') in Code 4, Line 8. Considering Q' as Q_1 . In this case we have:

  $$|\mathcal{SQL}(\mathsf{Q}_1)|_t = |\mathcal{SQL}(\mathsf{Q})|_t \tag{39}$$

  and A ≡ missing(t) and Q ≡ Q_1 except [all] Q_2. Then, from $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \setminus \Phi_{\mathsf{Q}_2}$ (changing $\setminus$ by $\setminus_{\mathcal{M}}$ in the case of all), and Definition 7 we have $\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \setminus \mathcal{E}(\mathsf{Q}_2)$. Therefore

  $$|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{E}(\mathsf{Q}_1)|_t \tag{40}$$

  If $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, by (39) we obtain that $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$ and by (40), $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$. Then, by induction hypothesis, V is incorrect.

– slice(V,Q,A) calls to slice(V,Q',A') in Code 4, Line 9. Considering Q' as Q_2 . In this case A ≡ missing(t), A' ≡ wrong(t), Q ≡ Q_1 EXCEPT Q_2, t $\in_\perp \mathcal{SQL}(\mathsf{Q}_2)$.

From $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \setminus \Phi_{\mathsf{Q}_2}$, and by Definition 7 we have

$$\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \setminus \mathcal{E}(\mathsf{Q}_2) \tag{41}$$

and by Definition 1,

$$\mathcal{SQL}(Q) = \| \Phi_Q \| = \| \Phi_{Q_1} \| \setminus \| \Phi_{Q_2} \| = \mathcal{SQL}(Q_1) \setminus \mathcal{SQL}(Q_2) \tag{42}$$

From t $\in_\perp \mathcal{SQL}(Q_2)$, we have that $|\mathcal{SQL}(Q)|_t = 0$, which means that the induction hypothesis $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$ can be rewritten as:

$$\text{If } \mathsf{t} \notin \mathcal{E}(\mathsf{Q}), \text{ then V is incorrect} \tag{43}$$

Now observe that in this case the call to slice is slice(V,Q_2,wrong(t)). Therefore we must prove that if $|\mathcal{E}(\mathsf{Q}_2)|_t \geq |\mathcal{SQL}(\mathsf{Q}_2)|_t$, then V is an incorrect view.
$|\mathcal{E}(\mathsf{Q}_2)|_t \geq |\mathcal{SQL}(\mathsf{Q}_2)|_t$ implies in particular that t $\in_\perp \mathcal{E}(\mathsf{Q}_2)$, which by (41) means that t $\notin \mathcal{E}(\mathsf{Q})$. Then, (43) holds. □

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

Lemma 3. *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:*

(P.1) $\mathcal{P} \vdash$ state((all R), valid) $\Rightarrow \mathcal{SQL}(\mathrm{R}) = \mathcal{I}(\mathrm{R})$
(P.2) $\mathcal{P} \vdash$ state((all R), nonvalid) $\Rightarrow \mathcal{SQL}(\mathrm{R}) \neq \mathcal{I}(\mathrm{R})$
(P.3) $\mathcal{P} \vdash$ state((t ∈ R), valid) $\Rightarrow$ t $\in_\perp \mathcal{I}(\mathrm{R})$
(P.4) $\mathcal{P} \vdash$ state((t ∈ R), nonvalid) $\Rightarrow$ t $\notin_\perp \mathcal{I}(\mathrm{R})$
(P.5) $\mathcal{P} \vdash$ state(($\mathrm{R}_1 \subseteq$ R), valid) $\Rightarrow \mathcal{SQL}(\mathrm{R}_1) \subseteq \mathcal{I}(\mathrm{R})$
(P.6) $\mathcal{P} \vdash$ state(($\mathrm{R}_1 \subseteq$ R), nonvalid) $\Rightarrow \mathcal{SQL}(\mathrm{R}_1) \not\subseteq \mathcal{I}(\mathrm{R})$

Proof. Proving $\mathcal{P} \vdash$ *state(E,S)* implies that there is a fact *state(E,S)* $\in \mathcal{P}$, because *state* is defined only by facts introduced by *processAnswer(E,A)* (Code 3, lines 1-5). We distinguish cases depending on the form of the input parameters E and A received by *processAnswer*.

– $E \equiv (R_1 \subseteq R)$. Then the function *processAnswer* has been called after asking the user about the validity of the enquire E, obtaining answer A. This happens in Code 1, line 10, and corresponds to the question *"Is S included in the intended answer for R?"*, with $S = \mathcal{SQL}(R_1)$ (see Definition 11). 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 $\mathcal{SQL}(R_1) \subseteq \mathcal{I}(R)$ that proves the implication *(P.5)*. The function *processAnswer* introduces the fact *state*($(R_1 \subseteq R)$,*nonvalid)* when the answer of the user is either *no* or *wrong(s)*, meaning that $\mathcal{SQL}(R_1) \not\subseteq \mathcal{I}(R)$ which proves the implication *(P.6)*.
– $E \equiv (t \in R)$. Analogously, the function *processAnswer* has been called from Code 1, line 7 after asking the user about the validity of the enquire E, obtaining answer A. In this case the answer provided by the user to the question *"Does the intended answer for R include a tuple matching the tuple t?"* can be *yes* or *no*. If the function *processAnswer* introduces the fact *state*($(t \in R)$,*valid)*, this implies that the

answer of the user was *yes*, meaning that $t \in_\perp \mathcal{I}(R)$ proving the implication *(P.3)*. The function *processAnswer* introduces the fact *state((t ∈ R),nonvalid)* when the answer of the user is *no*, meaning that $t \notin_\perp \mathcal{I}(R)$. Thus the implication *(P.4)* holds.

– $E \equiv$ *(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 the question *"Is S the intended answer for R?"*, with $S = \mathcal{SQL}(R)$. This corresponds to a call to processAnswer either from Code 1 line 10 or from Code 2, line 3.
   - If the call is from Code 1 line 10 then the fact *state((all R),valid)* is introduced as a consequence of an answer *yes*, meaning that $\mathcal{SQL}(R) = \mathcal{I}(R)$. Thus the implication *(P.1)* holds.
   - If the call is from Code 2 line 3 then the fact *state((all R),nonvalid)* is introduced by *processAnswer* as a consequence of an answer of the form *no*, *missing(t)* or *wrong(t)*. All these cases mean that $\mathcal{SQL}(R) \neq \mathcal{I}(R)$ (see Definition 4), and the implication *(P.2)* holds.
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 and all of them correspond to the implication *(P.2)*:
   - Line 7. The initial call must be *processAnswer((s ∈ R),yes)*, and this call has introduced a fact *state((s ∈ R),valid)*, which means:

     (44) $$s \in_\perp \mathcal{I}(R)$$

     Enquiries of the form *(s ∈ R)* are associated to clauses of the form:
     $$\mathsf{buggy(V)} \leftarrow \mathsf{state((s \in R),\ nonvalid)}$$
     which are added in P by the function *missingBasic* (Code 5) only when the debugger checks that:

     (45) $$s \notin_\perp \mathcal{SQL}(R)$$

     The recursive call is *processAnswer((all R),missing(s))*. This call introduces the fact *state((all R),nonvalid))*. Then, from (44) and (45), we obtain the result $\mathcal{SQL}(R) \neq \mathcal{I}(R)$ and the implication *(P.2)* holds.
   - Line 9. The first call must be *processAnswer((V ⊆ R), A)* with $A \equiv$ *no* or$A \equiv$ *wrong(s)*. This is one of the cases already analyzed, where A is the answer provided by the user for the enquiry *(V ⊆ R)*. Now, observe that this enquiry must correspond to the election of an atom *state((V ⊆ R), . . .)* already ocurring in the program. Such atoms are introduced in line 8 of Code 6. In this function, the parameter V corresponds to a new view created by function *relevantTuples* (Code 7), and by Lemma 1:

     (46) $$\mathcal{SQL}(V) \subseteq \mathcal{SQL}(R)$$

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

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

which, combined with (46) means:

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

□

4.2 Completeness

Next we study the completeness of the technique.

Theorem 5. *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* no, wrong(t) *or* missing(t)*, then the call* debug(R) *(defined in Code 1) returns a list L containing at least one relation.*

Proof. We prove the Completeness of our technique using some auxiliary results (see Figure 5).

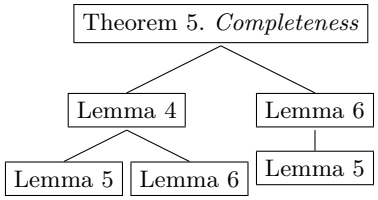

Fig. 5. The proof completeness structure

Lemma 4 proves that the program $\mathcal{P}$ contained in the variable P of Code 1 is finite.
Lemma 5 proves that the call to the function *slice* returns a finite number of clauses.
Lemma 6 proves that the while loop in Code 1 terminates in a finite number of iterations.

Let $\mathcal{P}$ be the logic program contained in the variable P of Code 1. By the structure of Code 1, the while loop only stops when there is at least one relation S such that $\mathcal{P} \vdash$ *buggy(S)*, and when this happens at least the relation S will be in the list L. Therefore, we only need to prove that the algorithm terminates in the conditions of the premises.
First we prove the termination of all calls to functions occurring in Code 1:

1. The call to *askOracle* in Code 1, lines 1 and 8 ends returning a valid value, that is a value of the form *yes*, *no*, *wrong(t)* or *missing(t)*.
2. The call to *initialize* in Code 2, line 3 ends because it traverses the computation tree top-down, and we are assuming finite computation trees (the database schema is finite and mutually recursive views are not allowed).
3. Functions *wrongBasic* and *missingBasic* always end. In both cases the body of the *for* loop is executed a finite number times because both the set *getFrom(Q)* and the set *getSelect(Q)* are finite for every query Q.

4. Function *slice* calls itself recursively traversing the structure of a query Q. Since the query definition must be finite, the recursion always ends reaching a *basic* query, and this case does not include recursive calls. This function calls to *wrongBasic* and *missingBasic* in lines 12 and 11 that by 3 always end. Then *slice* always ends.
5. Function *processAnswer* (Code 3) calls itself recursively at three points, but all these calls include a first parameter *(all R)* and observing the code we can check that these calls generate no further recursion. This function calls to *slice* in line 12, that by 4, always ends. Then *processAnswer* always ends.
6. The function *initialSetOfClauses* is called in Code 1 line 4. The input parameters are the view V to debug and a valid answer returned by the function *askOracle(all V)*. Then, the call to *initialSetOfClauses* ends by items 2 and 5.
7. The function *getBuggy* is called in Code 1 line 5 returning the list of all the relations R such that the goal *buggy(R)* can be proven w.r.t. the logic program P. This function always ends because the goal *buggy(R)* in *getBuggy(P)* is terminating. Lema 4 proves that the program P is finite. Thus, there is a maximum number of clauses *buggy* in P. And a finite, non-recursive (mutually recursive views are not allowed) and ground logic program is always terminating for any goal. This means that the goal *buggy(R)* in *getBuggy(P)* is terminating.
8. The function *getUnsolvedEnquiries* is called in Code 1 line 6. This function collects in a list *LE* all the unsolved enquiries e occurring in body atoms of the form *state(e,a)* of *buggy* clauses in P such that the logic program P does not contain neither a fact of the form *state(e,valid)* nor a fact of the form *state(e,nonvalid)*. By Lemma 4 there is a finite number of clauses *buggy* in P. For every atom of the form *state(e,a)* throw the goal *state(e,_)*, where the anonymous variable indicates that we do not care about the second parameter value. The goal *state(e,_)* is always terminating because predicate *state* is defined only by facts.
   Notice the returned list *LE* is never empty. In that case, the previous call to the function *getBuggy* would have returned a not empty list and the while loop would have stoped. As in the previous case, this function is always terminating.
9. The function *chooseEnquire* is called in Code 1 line 7 returning one of the enquiries in the list returned by the function *getUnsolvedEnquiries* according to some predefined terminating criterium.
10. Function *slice* calls itself recursively traversing the structure of a query Q. Since the query definition must be finite, the recursion always ends reaching a *basic* query, and this case does not include recursive calls.

In order to complete the proof we must check that the while loop in Code1, line 5 always terminates. By item 7, function *getBuggy(P)* ends and the goal *buggy(R)* in *getBuggy(P)* is terminating. By Lemma 6 the while loop terminates in a finite number of iterations.

□

Lemma 4. *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 computation tree of the view to debug. During the execution of the algorithm new facts for predicate *state* are added, and also new clauses with head *buggy* are included. Let n be the number of nodes in the computation tree rooted by R. Notice that n is finite number, because views cannot be mutually recursive and we are assuming a finite database schema. Facts and clauses are added in two points in Code *debug*:

a) In Line 4. Function *initialSetOfClauses(R, A)* adds $k_1 \leq 2 \times n$ buggy clauses and one fact for the predicate *state*.

- The function *initialize(R)* (Code 2) traverses recursively the computation tree for R adding exactly n buggy clauses, one buggy clause for each node in the computation tree $CT(R)$.
- The function *processAnswer((all R), A)* (called in Code 2, line 3), adds one fact for the predicate *state* and calls to function *slice(R, Q, A)* with Q the query defining R. Let m be the number of relations occurring in the *from* clauses of the basic components occurring in the query Q. Then, by Lemma 5, the call to *slice(R,Q,A)* returns k_2 clauses, with $k_2 \leq m$. By Definition 9, $m = n - 1$.
  Then $k_1 = n + k_2 \leq 2 \times n$.
  Consider the program P after the call to *initialSetOfClauses(R, A)* (Code 1, line 4). Let L be the list of all the unsolved enquiries in P and let k_3 the number of elements in L. Notice that the cardinality of each node in the computation tree $CT(R)$ is at most m, where the cardinality of a node N is defined as the number of children of N. Then, in this point, the number of elements in L is less or equal than $k_1 \times m$ (L contains one unsolved enquiry for each atom of the form *state(q,s)* occurring in body clauses in the current program P). Therefore, $k_3 \leq k_1 \times m$.

b) In Line 10. Function *processAnswer(E, A)* adds k_4 clauses and at most two facts for the predicate *state* in each iteration of the while loop.
- Function *processAnswer(E, A)* adds at most two facts for the predicate *state* because only one recursive call can be occurs.
- Function *processAnswer((all V), A)* calls to function *slice(V, Q, A)* with Q the query defining V. Let $CT(V)$ the computation tree of V with n' nodes. $CT(V)$ is a subtree of $CT(R)$ and therefore $n' \leq n$. Let p be the number of relations occurring in the *from* clauses of the basic components occurring in the query Q. Then, by Lemma 5, the call to *slice(V,Q,A)* returns k_4 clauses, with $k_4 \leq p$. By definition 9, $p = n' - 1$. Then $k_4 \leq n' \leq n$.
  By Lemma 6 the while loop stops before k_3 iterations. After k_3 iterations the list L is empty and the function *getBuggy* returns a not empty list. Then, after $k3$ iterations, the number of facts added to the program is less than $k_3 \times 2$ and the number of buggy clauses added to the program is less than $k_3 \times k_4$.

In summary, the number of facts added to the program is less than $p_1 = 1 + (k_3 \times 2)$ and the number of buggy clauses added to the program is less than $p_2 = k_1 + (k_3 \times k_4)$.

The result holds by considering $k = p_1 + p_2$. □

Lemma 5. *Let* slice(V,Q,A) *be any call to Code 4 that occurs during the execution of the debugger. Let $Q_1, \ldots, Q_q$ be the basic queries occurring in the query* Q*, and let n_i be the number of relations occurring in the* from *clause of the basic query* Q_i*, $i = 1, \ldots, q$. Then:*

1. *The call to* slice(V,Q,A) *returns k clauses, with $k \leq n_1 + \cdots + n_q$.*
2. *All the clauses returned by the call to* slice(V,Q,A) *are any of the following forms:*
   - *buggy(V) ← state((s ∈ R), nonvalid) with R a relation occurring in the* from *clause of any basic query* Q_i*, $i = 1, \ldots, q$.*

– *buggy(V)* ← *state((*$V_1 \subseteq R_1$*), valid), . . . , state((*$V_n \subseteq R_n$*), valid) with* R_j *relations occurring in the* from *clause of any basic query* Q_i*,* $i = 1, \ldots, q$*.*

Proof. We prove the results by structural induction on the form of the query Q.

Basis: If Q is a basic query. We distinguish two cases:

- If A = missing(t), the set of clauses returned by the call *slice(V,Q,A)* is the set of clauses returned by the call to *missingBasic(V, Q, t)* (line 11 of code 4). Let R_1 AS $S_1, \ldots, R_n$ AS S_n be the list of elements returned by the function *getFrom(Q)*. Then the call *missingBasic(V, Q, t)* returns k buggy clauses of the form:
$$\mathsf{buggy(V)} \leftarrow \mathsf{state((s} \in \mathsf{R}_i\mathsf{),\ nonvalid)}$$
with $0 \leq k \leq n$. Then, the results 1 and 2 hold with $k \leq n$.
- If A = wrong(t), the set of clauses returned by the call *slice(V,Q,A)* is the set of clauses returned by the call to *wrongBasic(V, Q, t)* (line 12 of code 4). Let R_1 AS $S_1, \ldots, R_n$ AS S_n be the list of elements returned by the function *getFrom(Q)*. The call *wrongBasic(V, Q, t)* returns only one buggy clause (Code 6, line 8) of the form:
$$\mathsf{(buggy(V)} \leftarrow \mathsf{state((V}_1 \subseteq \mathsf{R}_1\mathsf{),\ valid),\ \ldots,\ state((V}_n \subseteq \mathsf{R}_n\mathsf{),\ valid))}$$
Then, the results 1 and 2 hold with $k = 1$.

Notice that in both cases, *slice(V,Q,A)* adds new clauses that do not imply enquiries of the form *(all R)*.

Inductive step: If Q is a compound query. Let Q_1, Q_2 its query components. In this case $q = q_1 + q_2$ where q_i is the total number of basic queries occurring in the query Q_i, $i = 1, 2$. By induction hypothesis, the call to the function *slice(V,*Q_i*,A)* returns a finite set of clauses $k_i \leq n_i$, where n_i is the total number of relations occurring in the *from* clauses of the basic queries Q_{i1}, . . . , Q_{iq_i} occurring in the query Q_i, $i = 1, 2$.
Additionally, all the clauses returned by the call to *slice(V,*Q_i*,A)* are any of the following forms:

- $\mathsf{buggy(V)} \leftarrow \mathsf{state((s} \in \mathsf{R}\mathsf{),\ nonvalid)}$ with R a relation occurring in the *from* clause of any basic query Q_{i1}, . . . , Q_{iq_i} occurring in the query Q_i, $i = 1, 2$.
- $\mathsf{buggy(V)} \leftarrow \mathsf{state((V}_1 \subseteq \mathsf{R}_1\mathsf{),\ valid),\ \ldots,\ state((V}_n \subseteq \mathsf{R}_n\mathsf{),\ valid)}$ with R_j the relations occurring in the *from* clause of any basic query Q_{i1}, . . . , Q_{iq_i} occurring in the query Q_i, $i = 1, 2$.

Following the Code 4, lines 2 to 9, the number of clauses k returned by the call *slice(V,Q,A)* is less than the number of clauses returned by the call to *slice(V,*Q_1*,_)* and *slice(V,*Q_2*,_)*. Then, $k \leq k_1 + k_2 < n_1 + n_2$.

□

Lemma 6. *Given a call* debug(R)*, the* while *loop in Code 1, lines 5-11 terminates in a finite number of iterations.*

Proof. A common tool for proving the termination of programs is the well-founded set, a set ordered in such a way as to admit no infinite descending sequences [9]. We use a well-founded multiset order $\succeq$ for proving the termination of the while loop in Code 1. The idea is to define an ordering $\succeq$ on finite multisets of enquiries that is induced by the ordering $\succ$ on the enquiries. We consider the following well-founded partial-ordering $\succ$ on the enquiries:

- *(all V)* $\succ$ *(s ∈ R)* for all R in $CT(V)$
- *(all V)* $\succ$ *(V' ⊆ R)* for all R in $CT(V)$
- *(V ⊆ R)* $\succ$ *(s ∈ R')* for all R' in $CT(R)$
- *(V ⊆ R)* $\succ$ *(V' ⊆ R')* for all R' in $CT(R)$ and $R \neq R'$
- *(s ∈ V)* $\succ$ *(t ∈ R)* for all R in $CT(V)$ and $R \neq V$
- *(s ∈ V)* $\succ$ *(V' ⊆ R)* for all R in $CT(V)$ and $R \neq V$

It is easy to check that there is no infinite descending chain using the ordering $\succ$ on the enquiries. The ordering $\succ$ on the enquiries induces an ordering $\succeq$ on multisets of enquiries which is defined as follows:

Let L_1 and L_2 be two multisets of enquiries. $L_1 \preceq L_2$ if for some multisets of enquiries X and Y, where X es not empty and $X \subseteq L_1$, $L_2 = (L_1 \setminus_{\mathcal{M}} X) \cup_{\mathcal{M}} Y$ and for all $y \in Y$, there exist $x \in X$ such that $x \succ y$.

In this ordering, $L_1 \preceq L_2$ if L_2 can be obtained from L_1 by replacing one or more enquiries in L_1 by any finite number of enquiries, each of which is smaller than one of the replaced enquiry. In particular, a multiset of enquiries is reduced by replacing an enquiry with zero enquiries, i.e. by deleting it.

Consider de program P after the call to *initialSetOfClauses* (Code 1, line 4). Let LE be the list of all the unsolved enquiries in P returned by the call to the function *getUnsolvedEnquiries* (Code 1, line 6). Next we prove that after each loop iteration either the while condition becomes true (a buggy node is found) and the while loop terminates or the multiset LE of the unsolved enquiries in P is reduced. This means that eventually LE will be empty, but this means that the initial set of enquiries LE are solved and a buggy node has been found.

At each iteration of the loop (Code 1, lines (5 - 11)), new clauses and facts returned by the function *processAnswer* (line 10) are added in P.

The new clauses returned by the function *processAnswer* are the clauses returned by the function *slice* called in Code 3, line 12. By Lemma 5, function *slice* returns a finite number of new clauses, and the returned clauses do not imply enquiries of the form *(all V)*.

One unsolved enquiry E in LE is selected in line 7, and the answer A provided by the user is processed calling to the function *processAnswer(E, A)* (line 10 in Code 1). Each call to the function *processAnswer* solves one or two unsolved enquiries in LE (Code 3, lines 1-5). We distinguish cases depending of the form of the enquiry E and the answer A:

- $E \equiv$ *(t ∈ V)*.
  - $A \equiv$ *no*. In this case, function *processAnswer* returns the fact *state((t ∈ V),nonvalid)* and the enquiry E is solved. Notice no more iterations are needed because the enquiry $(t \in V)$ is associated to a buggy clause in P of the form:

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

    and therefore, the goal *buggy(R)* can be proved w.r.t. the program P and the while condition becomes true.
  - $A \equiv$ *yes*. In this case, function *processAnswer* returns two facts. The first one is *state((t ∈ V),valid)* and the second one is *state((all V), nonvalid)*. The last one is returned by the recursive call to the function *processAnswer* with the enquiry *(all V)* and missing(t) as parameters (line 7). Therefore at least the enquiry E is solved.
    The recursive call returns a set of clauses returned by the call to the function *slice(V, Q, missing(t))* with Q the query defining V. By Lemma 5, item 2, the

returned clauses imply a finite set of enquiries $E_1, \ldots E_n$ such that $E \succ E_i$, $0 \leq i \leq n$. If the while condition becomes true, the while loop terminates. Otherwise, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by replacing the enquiry *(t ∈ V)* (and possibly the enquiry *(all V)*) by the set of enquiries $E_1, \ldots E_n$ and therefore $LE \succeq LE'$.

- $E \equiv$ *(V ⊆ R)*.
  - $A \equiv no$ or $A \equiv wrong(t)$. Function *processAnswer* returns a fact of the form *state((V ⊆ R),nonvalid)* and the enquiry E is solved. The function *processAnswer* is called recursively with the enquiry *(all R)* and A as parameters (line 10). The recursive call returns a set of clauses returned by the call to the function *slice(R, Q, A)* with Q the query defining R. By Lemma 5, item 2, the returned clauses imply a finite set of enquiries $E_1, \ldots E_n$ such that $E \succ E_i$, $0 \leq i \leq n$. If the the goal *buggy(R)* can not be proved w.r.t. the program P, the loop body is executed again, in which case the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by replacing the enquiry *(V ⊆ R)* (and possibly the enquiry *(all R)*) by the set of enquiries $E_1, \ldots E_n$. Therefore, $LE \succeq LE'$.
  - $A \equiv yes$. Function *processAnswer* returns a fact of the form *state((V ⊆ R),valid)* and the enquiry E is solved. If the while loop does not terminates, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by deleting the enquiry *(V ⊆ R)* and therefore $LE \succeq LE'$.
- $E \equiv$ *(all V)*.
  - $A \equiv yes$. Function *processAnswer* returns a fact of the form *state((all V), valid)* and the enquiry E is solved. If the while condition becomes true, the while loop terminates. Otherwise, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by deleting the enquiry *(all V)* and therefore $LE \succeq LE'$.
  - $A \equiv no$. In this case, function *processAnswer* returns a fact of the form *state((all V),nonvalid)* and the enquiry E is solved. As in the previous case, if the while condition becomes true, the while loop terminates. Otherwise, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by by deleting the enquiry *(all V)* and therefore $LE \succeq LE'$.
  - $A \equiv missing(t)$ or $A \equiv wrong(t)$. Function *processAnswer* returns a fact of the form *state((all V),nonvalid)* (and the enquiry E is solved) and a set of clauses returned by the call to the function *slice(V, Q, A)* with Q the query defining V. By Lemma 5, item 2, the returned clauses imply a finite set of enquiries $E_1, \ldots E_n$ such that $E \succ E_i$, $0 \leq i \leq n$. If the while condition becomes true, the while loop terminates. Otherwise, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by deleting the enquiry *(all V)* and therefore $LE \succeq LE'$. □

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 [16,17]), 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 12 of Code 3). The debugger checks that Q_2 produces *('Anna')* (line 9 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 12 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 clause is added to the program P (line 8, Code 6) where subset(intensive_slice,intensive) represents the enquiry $E \equiv$*(intensive_slice* $\subseteq$ *intensive*) :

```
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_slice_1(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');

create view standard_slice_2(student,level,pass) as
   ( select R.student, R.level, R.pass from standard as R)
       intersect all
   (select  A2.student, A2.level, A2.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');

create view standard_slice_3(student,level,pass) as
   ( select R.student, R.level, R.pass from standard as R)
       intersect all
   (select  A3.student, A3.level, A3.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');
```

Finally, the clause:

```
buggy(intensive) :- state(subset(standard_slice_1,standard),valid),
                    state(subset(standard_slice_2,standard),valid),
                    state(subset(standard_slice_3,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 pointing out *intensive* as a source of the error.

6 Conclusions

We have presented a new technique for debugging systems of SQL views. Our proposal present a declarative debugging technique and then it refines the technique 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 declarative debugging when applied directly to SQL views, namely the huge number of tuples that the user must consider in order to determine the validity of the result produced by a relation. Previous works deal with the problem of tracking provenance information for query results [11,8], but to the best of our knowledge, none of them treat the case of missing tuples, which is important in our setting. This report

extends two previous papers [4,5] which present an abbreviated version and without proofs of all the results of our setting.

The proposed algorithm 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 report 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 *SDKB 2008*, volume 4925 of *LNCS*, 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 of *Lecture Notes in Computer Science*, pages 77–85. Springer LNCS, 2012.
5. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Declarative Debugging of Wrong and Missing Answers for SQL Views. In *Eleventh International Symposium on Functional and Logic Programming (FLOPS 2012)*, volume 7294 of *LNCS*, pages 73–87. Springer-Verlag, 2012.
6. 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.
7. 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.
8. Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25:179–227, June 2000.
9. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, Aug. 1979.
10. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.
11. B. Glavic and G. Alonso. Provenance for nested subqueries. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 982–993, New York, NY, USA, 2009. ACM.
12. P. W. Grefen and R. A. de By. A multi-set extended relational algebra: a formal approach to a practical issue. In *ICDE'94*, pages 80–88. IEEE, 1994.

13. L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 3, 1997.
14. 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.
15. Rapid SQL Developer Debugger, 2011. `http://docs.embarcadero.com/products/rapid_sql/`.
16. F. Sáenz-Pérez. Datalog Educational System v2.6, October 2011. `http://des.sourceforge.net/`.
17. F. Sáenz-Pérez. DES: A Deductive Database System. *Elec. Notes on Theor. Comp. Science*, 271, 2011.
18. E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.
19. J. Silva. A survey on algorithmic debugging strategies. *Advances in Engineering Software*, 42(11):976–991, 2011.
20. SQL, ISO/IEC 9075:1992, third edition, 1992.