Algorithmic Debugging of SQL Views [1]

R. Caballero and Y. García-Ruiz and F. Sáenz-Pérez

TECHNICAL REPORT SIC 03/11

Dep. Lenguajes y Sistemas Informáticos y Computación
Univ. Complutense de Madrid

May 4, 2011

[1]This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502)

Contents

Abstract

We present a general framework for debugging systems of correlated SQL views. The debugger locates an erroneous view by navigating a suitable computation tree. This tree contains the computed answer associated with every intermediate relation, asking the user whether this answer is expected or not. The correctness and completeness of the technique is proven formally, using a general definition of SQL operational semantics. The theoretical ideas have been implemented in an available tool which includes the possibility of employing trusted specifications for reducing the number of questions asked to the user.

Chapter 1

Introduction

SQL [15] is the *de facto* standard language for querying and updating relational databases. Its declarative nature and its high-abstraction level allows the user to easily define complex operations that could require hundreds of lines programmed in a general purpose language. In the case of relational queries, the language introduces the possibility of querying the database directly using a *select* statement. However, in realistic applications, queries can become too complex to be coded in a single statement and are generally defined using *views*. Views can be considered in essence as virtual tables. They are defined by a *select* statement that can rely on the database tables as well as in other previously defined views. Thus, views become the basic components of SQL queries.

As in other programming paradigms, views can have bugs. However, we cannot infer that a view is buggy only because it returns an unexpected result. Maybe it is correct but receives erroneous input data from the other views or tables it depends on. There are very few tools for helping the user to detect the cause of these errors; so, debugging becomes a labor-intensive and time-consuming task in the case of queries defined by means of several intermediate views. The main reason for this lack of tools is that the usual *trace* debuggers used in other paradigms are not available here due to the high abstraction level of the language. A *select* statement is internally translated into a sequence of low level operations that constitute the *execution plan* of the query. Relating these operations to the original query is very hard, and debugging the execution plan step by step will be of little help. In this report, we propose a theoretical framework for debugging SQL views based on *declarative debugging*, also known as *algorithmic debugging* [13]. This technique has been employed successfully in (constraint) logic programming [16], functional programming [10], functional-logic programming [3], and in deductive database languages [1]. The overall idea of declarative debugging [8] can be explained briefly as follows:

- The process starts with an initial error symptom, which in our case corresponds to the unexpected result of a user-defined view.

- The debugger automatically builds a tree representing the computation. Each node of the tree corresponds to an intermediate computation with its result. The children of a

node are those nodes obtained from the subcomputations needed for obtaining the parent result. In our case, nodes will represent the computation of a relation R together with its answer. Children correspond to the computation of views and tables occurring in R if it is a view.
- The tree is *navigated*. An external oracle, usually the user, compares the computed result in each node with the *intended* interpretation of the associated relation. When a node contains the expected result, it is marked as *valid*, otherwise it is marked as *nonvalid*.
- The navigation phase ends when a nonvalid node with valid children is found. Such node is called a *buggy node*, and corresponds to an incorrect piece of code. In our case, the debugger will end pointing out either an erroneously defined view, or a table containing a nonvalid instance.
Our goal is to present a declarative debugging framework for SQL views, showing that it can be implemented in a realistic, scalable debugging tool. We have implemented our debugging proposal in the Datalog Educational System (DES [12]), which makes it possible for Datalog and SQL to coexist as query languages for the same database. The current implementation of our proposal for debugging SQL views and instructions to use it can be downloaded from `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/Des`.

The next chapter introduces the basic terminology for representing SQL relations and the SQL semantics. Chapter 3 introduces the framework for SQL algorithmic debugging, proving its adequacy. A system implementing these ideas, together with a discussion about its usefulness, is introduced in Chapter 4. Finally, the report ends presenting the conclusions.

Chapter 2

SQL Semantics

The first formal semantics for relational databases based on the concept of set (e.g., relational algebra, tuple calculus [4]) were incomplete with respect to the treatment of non-relational features such as repeated rows and aggregates, which are part of practical languages such as SQL. Therefore, other semantics, most of them based on multisets [5], have been proposed. In our framework we will use the *Extended Relational Algebra* [7, 6]. We start by defining the concepts of database schemas and instances.

2.1 Relational Database Schemas and Instances

A *table schema* is of the form $T(A_1, \ldots, A_n)$, with T being the table name and A_i the attribute names for $i = 1 \ldots n$. We will refer to a particular attribute A by using the notation $T.A$. Each attribute A has an associated type (*integer*, *string*, ...).

An *instance* of a table schema $T(A_1, \ldots, A_n)$ is determined by its particular *rows*. Each row contains values of the correct type for each attribute in the table schema.

A *database schema* D is a tuple $(\mathcal{T}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of tables and $\mathcal{V}$ a finite set of views. *Views* can be thought of as new tables created dynamically from existing ones by using a SQL query. The general syntax of a SQL view is: create view V(A_1, ..., A_n) as Q, with Q a query and $V.A_1, \ldots V.A_n$ the names of the view attributes. A *database instance* d of a database schema is a set of table instances, one for each table in $\mathcal{T}$. To represent the instance of a table T in d we will use the notation $d(T)$.

The syntax of SQL queries can be found in [15]. The *dependency tree* of any view V in the schema is a tree with V labeling the root, and its children the dependency trees of the relations occurring in its query. In general, we will use the name *relation* to refer to either a table or a view. The next example defines a particular database schema that will be used in the rest of the report as a running example.

Example 2.1.1 *The* Dog and Cat Club *annual dinner is going to take place in a few weeks, and the organizing committee is preparing the guest list. Each year they browse*

Owner

id	name
1	Mark Costas
2	Helen Kaye
3	Robin Scott
4	Tom Cohen

Pet

code	name	species
100	Wilma	dog
101	Kitty	cat
102	Wilma	cat
103	Lucky	dog
104	Rocky	dog
105	Oreo	cat
106	Cecile	turtle
107	Chelsea	dog

PetOwner

id	code
1	100
1	101
2	102
2	103
3	104
3	105
4	106
4	107

Figure 2.1: All Pets Club database instance

the database of the All Pets Club *looking for people that own at least one cat and one dog. Owners come to the dinner with all their cats and dogs. However, two additional constraints have been introduced this year:*

- *People owning more than 5 animals are not allowed (the dinner would become too noisy).*
- *No animals sharing the same name are allowed at the party. This means that if two different people have a cat or dog sharing the same name neither of them will be invited. This severe restriction follows after last year's incident, when someone cried* Tiger *and dozens of pets started running without control.*

Figure 2.1 shows the All Pets Club *database instance. It consists of three tables:* Owner, Pet, *and* PetOwner *which relates each owner with its pets. Primary keys are shown underlined. Figure 2.2 contains the views for selecting the dinner guests. The first view is* AnimalOwner, *which obtains all the tuples (*id,aname,species*) such that* id *is the owner of an animal of name* aname *of species* species. LessThan6 *returns the identifiers of the owners with less than six cats and dogs.* CatsAndDogsOwner *returns pairs (*id,aname*) where* id *is the identifier of the owner of either a cat or a dog with name* aname, *such that* id *owns both cats and dogs.* NoCommonName *is defined by removing owners sharing pet names from the total list of cats and dog owners. Finally, the main view is* Guest, *which selects those owners that share no pet name with another owner (view* NoCommonName*) and that have less than six cats and dogs (view* LessThan6*). However, these views contain a bug that will become apparent in the next sections.*

2.2 Extended Relational Algebra

The Extended Relational Algebra (ERA from now on) [7] is an operational SQL Semantics allowing aggregates, views and most of the common features of SQL queries. The main characteristics of ERA are:

```
create or replace view AnimalOwner(id,aname,species) as
 select O.id, P.name, P.species
 from Owner O, Pet P, PetOwner PO
 where O.id = PO.id and P.code = PO.code;

create or replace view LessThan6(id) as
 select id from AnimalOwner
 where species='cat' or species='dog'
 group by id having count(*)<6;

create or replace view CatsAndDogsOwner(id,aname) as
 select AO1.id,AO1.aname
 from AnimalOwner AO1, AnimalOwner AO2
 where AO1.id = AO2.id and AO1.species='dog'
       and AO2.species='cat';

create or replace view NoCommonName(id) as
 select id from CatsAndDogsOwner
 except
 select B.id from CatsAndDogsOwner A, CatsAndDogsOwner B
                  where A.id <> B.id
                  and A.aname = B.aname;

create or replace view Guest(id,name) as
 select id, name
 from Owner natural inner join NoCommonName
            natural inner join LessThan6;
```

Figure 2.2: Views for selecting dinner guests

- The table instances and the result of evaluating queries/views are multisets, (it is also possible to consider *lists* instead of multisets if we consider relevant the order among rows in a query result).
- ERA expressions define new relations by combining previously defined relations using multiset operators $\cup, \cap, \setminus, \times$, projections π, selections σ, the renaming operator ρ, the grouping operator γ, and the distinct operator δ (see [6] for a formal definition of each operator).
- We use Φ_R to represent a SQL query or view R as an ERA expression, as explained in [6]. Since a query/view depends on previously defined relations, sometimes it will be useful to write $\Phi_R(R_1, \ldots, R_n)$ indicating that R depends on $R_1, \ldots, R_n$. If $M_1, \ldots, M_n$ are multisets we use the notation $\Phi_R(M_1, \ldots, M_n)$ to indicate that the

expression Φ_R is evaluated after substituting $R_1, \ldots, R_n$ by $M_1, \ldots, M_n$.

- Tables are denoted by their names, that is, $\Phi_T = T$ if T is a table.
- The computed answer of Φ_R with respect to some schema instance d will be denoted by $\| \Phi_R \|_d$, where
  - If R is a database table, $\| \Phi_R \|_d = d(R)$.
  - If R is a database view or a query and $R_1, \ldots, R_n$ the relations defined in R then $\| \Phi_R \|_d = \Phi_R(\| \Phi_{R_1} \|_d, \ldots, \| \Phi_{R_n} \|_d)$.

Observe that $\| \Phi_R \|_d$ is well defined since mutually recursive view definitions are not allowed[1]. We assume that $\| \Phi_R \|_d$ actually corresponds to the answer obtained by a correct SQL implementation, i.e., that the available SQL systems implement ERA.

Example 2.2.1 *The ERA expression associated with the views* AnimalOwner *and* CatsAndDogsOwner *are:*

$$\Phi_{AnimalOwner} = \rho_{AnimalOwner(id,aname,species)}(\Pi_{O.id,P.name,P.species}(\sigma_{O.id=PO.id \ \wedge \ P.code=PO.code}(O \times P \times PO)))$$

$$\Phi_{CatsAndDogsOwner} = \rho_{CatsAndDogsOwner(id,aname)}(\Pi_{A.id,A.aname}(\sigma_{A.id=B.id \ \wedge \ A.species='dog' \ \wedge \ B.species='cat'}(\rho_A(AnimalOwner) \times \rho_B(AnimalOwner))))$$

where O, P, and PO abbreviate respectively Owner, Pet, *and* PetOwner. *Assuming the schema instance d of Figure 2.1 we have:*

$$\| AnimalOwner \|_d = \Phi_{AnimalOwner}(d(Owner), d(Pet), d(PetOwner)) = \{\!| (1, Wilma, dog), (1, Kitty, cat), (2, Wilma, cat), (2, Lucky, dog), (3, Rocky, dog), (3, Oreo, cat), (4, Cecile, turtle), (4, Chelsea, dog) |\!\}$$

In turn, this answer can be used to obtain $\| CatsAndDogsOwner \|_d$*:*

$$\| CatsAndDogsOwner \|_d = \Phi_{CatsAndDogsOwner}(\| AnimalOwner \|_d) = \{\!| (1, Wilma), (2, Lucky), (3, Rocky) |\!\}$$

The two SQL answers correspond to the computed answer in any SQL system for these views.

[1] Recursive views are allowed in the SQL:1999 standard but they are not supported in all the systems, and they are not considered here.

Chapter 3

Declarative Debugging Framework

In this section, we assume a set of SQL views $\mathcal{V} = \{V_1, \ldots, V_n\}$ such that for some $1 \leq i \leq n$, and for some database instance d, V_i has produced an unexpected result in some SQL system. We also assume that this SQL system implements the ERA operational semantics of previous section. First, we introduce the concept of erroneous SQL relations, and then our proposal to debug with computation trees.

3.1 Erroneous SQL Relations

Our debugging technique will be based on the comparison between the answers by a SQL system implementing the ERA semantics, and the oracle intended answers. Next, we define the concept of intended answer for schema relations.

Definition 3.1.1 Intended Answers for Schema Relations
Let D be a database schema, d an instance of D, and R a relation defined in D. The intended answer for R w.r.t. d, is a multiset denoted as $\mathcal{I}(R, d)$ containing the answer that the user expects for the query select * from R; *in the instance d.*

The intended answer depends not only on the view semantics but also on the contents of the tables in the instance d. This concept corresponds to the idea of *intended interpretations* employed usually in algorithmic debugging. Figure 3.1 contains the intended answer for each view defined in Figure 2.2. For instance, it is expected that *AnimalOwner* will identify each owner *id* with the names and species of his pets. It is also expected than *LessThan6* will contain the *id* of all four owners, since all of them have less than six cats and dogs. The intended answer for view *CatsAndDogsOwner* contains the *id* and *name* attributes of those entries in *AnimalOwner* corresponding to owners with at least one dog and one cat, and removing pets different from cats and dogs. View *NoCommonName* is expected to contain only one row for owner with identifier 3. The reason is that both owners 1 and 2 share a pet name (*Wilma*). Finally, the only expected *Guest* will be the owner with identifier 3, *Robin Scott*. If now we try the query select

AnimalOwner

id	aname	species
1	Wilma	dog
1	Kitty	cat
2	Wilma	cat
2	Lucky	dog
3	Rocky	dog
3	Oreo	cat
4	Cecile	turtle
4	Chelsea	dog

LessThan6

id
1
2
3
4

CatsAndDogsOwner

id	aname
1	Wilma
1	Kitty
2	Wilma
2	Lucky
3	Oreo
3	Rocky

NoCommonName

id
3

Guest

id	name
3	Robin Scott

Figure 3.1: Intended answer for the views in Example 2.1.1

* from Guest; in a SQL system, we obtain a computed answer representing the multiset $\{\!|(1, Mark\ Costas),\ (2, Helen\ Kaye),\ (3, Robin\ Scott)|\!\}$. This computed answer is different from the intended answer for *Guest*, and indicates that there is some error. However, we cannot ensure that the error is in the query for *Guest*, because the error can come from any of the relations in its *from* clause. And also from the relations used by these relations, and so on. In order to define the key concept of erroneous relation it will be useful to define the auxiliary concept of *inferred answer*.

Definition 3.1.2 Inferred Answers
Let D be a database schema, d an instance of D, and R a relation in D. The inferred answer for R, with respect to d, $\mathcal{E}(R, d)$, is defined as

1. *If R is a table, $\mathcal{E}(R, d) = d(R)$.*
2. *If R is a view, $\mathcal{E}(R, d) = \Phi_R(\mathcal{I}(R_1, d), \ldots, \mathcal{I}(R_n, d))$ with $R_1, \ldots, R_n$ the relations occurring in R.*

Thus, in the case of tables, the inferred answer is just its table instance. In the case of a view V, the inferred answer corresponds to the computed result that would be obtained assuming that all the relations R_i occurring in the definition of V contain the intended answers. For instance, consider Example 2.1.1 and the instance d of Figure 2.1. Assume that all the tables contain the intended answers, i.e., for every table T, $\mathcal{I}(T, d) = d(T)$. Then the inferred answer for view *CatsAndDogsOwner* is the same as its computed answer $\| CatsAndDogsOwner \|_d$:

$$\mathcal{E}(CatsAndDogsOwner, d) = \Phi_{CatsAndDogsOwner}(\mathcal{I}(AnimalOwner, d)) = \{\!|\ (1, Wilma), (2, Lucky), (3, Rocky)|\!\}$$

However, this result is different from the intended answer for this view (Fig. 3.1). A discrepancy between $\mathcal{I}(R,d)$ and $\mathcal{E}(R,d)$ shows that R does not compute its intended answer, even assuming that all the relations it depends on contain their intended answers. Such relation is erroneous:

Definition 3.1.3 Erroneous Relation
Let D be a database schema, d an instance of D, an R a relation defined in D. We say that R is an erroneous relation when $\mathcal{I}(R,d) \neq \mathcal{E}(R,d)$.

For instance, *CatsAndDogsOwner* is erroneous; its correct definition is:

```
create or replace view CatsAndDogsOwner(id,aname) as
  select AO1.id,AO1.aname
  from animalOwner AO1, animalOwner AO2
  where AO1.id = AO2.id and
        ((AO1.species='dog' and AO2.species='cat') or
         (AO2.species='dog' and AO1.species='cat'));
```

instead of the code listed in Figure 2.2. This error was the cause of the unexpected answer for *Guest*. Observe that in Figure 3.1 we have not included the intended answers for table instances, assuming implicitly that they correspond to the actual table instances, thus *trusting* the instance d. This is the usual case, although our setting allows also the detection of wrong table instances. The oracle can choose whether the table instances can be trusted or not during a debugging session (see Section 4).

3.2 Debugging with Computation Trees

Definition 3.1.3 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 $\mathcal{I}(V,d)$ and $\mathcal{E}(V,d)$. By Definition 3.1.2, to obtain $\mathcal{E}(V,d)$, the tool will need the intended answer $\mathcal{I}(R,d)$ for every R occurring in the query defining V. But $\mathcal{I}(R,d)$ is only known by the user, who should provide this information during the debugging process. Obviously, a technique requiring such amount of information would be rejected by most of the users. Instead, we will require from the oracle only to answer questions of the form *'Is the computed answer (...) the intended answer for view V?'* Thus, the declarative debugger will compare the computed answer –obtained from the SQL system– and the intended answer –known by the oracle– In a first phase, the debugger builds a *computation tree* for the main view. The definition of this structure is the following:

Definition 3.2.1 Computation Trees
Let D be a database schema with views $\mathcal{V}$, d an instance of D, and R a relation defined in D. The computation tree $CT(R,d)$ associated with R w.r.t. d is defined as follows:

- *The root of $CT(R,d)$ is $(R \mapsto\| \Phi_R \|_d)$.*

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

In practice, the nodes in the computation tree correspond to the syntactic dependency tree of the main SQL view, with the children at each node corresponding to the relations occurring in the definition of the corresponding view. After building the computation tree, the debugger will *navigate* the tree, asking the oracle about the validity of some nodes:

Definition 3.2.2 Valid, Nonvalid and Buggy Nodes
Let $T = CT(R, d)$ *be a computation tree, and* $N = (R' \mapsto\| \Phi_{R'} \|_d)$ *a node in* T*. We say that* N *is* valid *when* $\| \Phi_{R'} \|_d = \mathcal{I}(R', d)$*,* nonvalid *when* $\| \Phi_{R'} \|_d \neq \mathcal{I}(R', d)$*, and* buggy *when* N *is nonvalid and all its children in* T *are valid.*

The goal of the debugger will be to locate buggy nodes. 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 3.2.3 *Let* d *be an instance of a database schema* D*,* V *a view defined in* D*, and* T *a computation tree for* V *w.r.t.* d*. If the root of* T *is nonvalid then:*

- *Completeness.* T *contains a buggy node.*
- *Soundness. Every buggy node in* T *corresponds to an erroneous relation.*

Proof. 3.2.4 *The completeness is straightforward using induction on the number of nodes of* T*, see [8]. In order to prove the soundness, let* $N = (R \mapsto\| \Phi_R \|_d)$ *be a buggy node in* T*. We must prove* R *is erroneous i.e., that* $\mathcal{I}(R, d) \neq \mathcal{E}(R, d)$*. Since* N *is buggy,* $\| \Phi_R \|_d \neq \mathcal{I}(R, d)$*. Then it is enough to prove that* $\| \Phi_R \|_d = \mathcal{E}(R, d)$*. If* R *is a table, then* N *is a leaf and* $\| \Phi_R \|_d = d(R) = \mathcal{E}(R, d)$*. If* R *is a view, we have that the children* $N_1, \ldots, N_m$ *of* N *will correspond to the relations* $R_1, \ldots, R_m$ *occurring in* R*. Since all of them are valid, we have* $\| \Phi_{R_i} \|_d = \mathcal{I}(R_i, d)$ *for* $i = 1 \ldots m$*. Then:*

$$\mathcal{E}(R, d) = \Phi_R(\mathcal{I}(R_1, d), \ldots, \mathcal{I}(R_n, d)) = \Phi_R(\| \Phi_{R_1} \|_d, \ldots, \| \Phi_{R_n} \|_d) = \| \Phi_R \|_d \quad \square$$

In our setting, the debugging process starts when the user finds a view V returning an unexpected result. The debugger builds the computation tree for V, which therefore will have a nonvalid root as required by the theorem. Figure 3.2 shows the computation tree for our running example after removing the repeated children for the sake of space. Nonvalid nodes are underlined, and the only buggy node (in bold face) corresponds to view *CatsAndDogsOwner*.

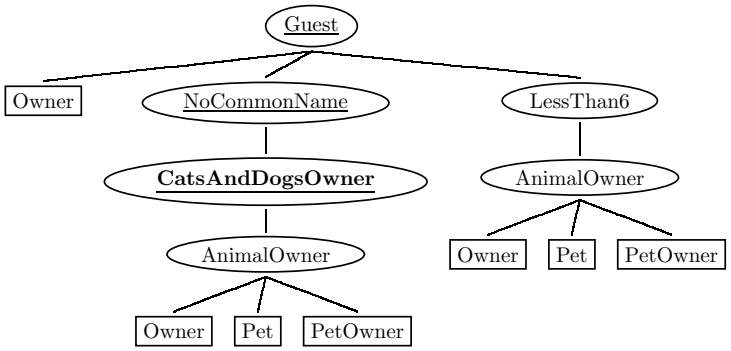

Figure 3.2: Computation tree for view Guest

Lemma 3.2.5 *Let $T = CT(R, d)$ be a computation tree with nonvalid root and let N be a valid node in T. Let T' be defined as $T' = T - N$, when the operation $T - N$ corresponds to:*

- *Removing N from T if it is a leaf.*
- *If N is an internal node and M is the parent of N, then the children of N become children of M after removing N.*

Then, T' contains a buggy node which is also buggy in T.

Proof. 3.2.6 *Let B be a buggy node in T'. Then B is nonvalid. This means that $B \neq N$, and thus $B \in T$. We distinguish two cases:*
- If N is not a child of B in T, then B have the same children in T' and in T and the result follows.
- Suppose that N is a child of B in T. Then B in T has the same children as in T' with the exception of N, which replaces some of them. Since N is valid all the children of B in T are valid, and B is buggy in T. □

The second lemma proves that in the case of identical children subtrees (very common in SQL), one of the children can be safely removed.

Lemma 3.2.7 *Let $T = CT(R, d)$ be a computation tree and N a node with two children subtrees T_1, T_2 such that $T_1 = T_2$. Let T' be the result of removing either T_1 or T_2 from T. Then T contains a buggy node associated with a relation R' iff T' contains a buggy node associated with the same relation R'.*

Proof. 3.2.8 *Straightforward.* □

Chapter 4

An Actual Declarative Debugger

We have implemented our debugging proposal in the Datalog Educational System (DES [11, 12]), which makes it possible for Datalog and SQL to coexist as query languages for the same database. The current implementation of our proposal for debugging SQL views and instructions to use it can be downloaded from

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/Des`

This section introduces the system we have selected for implementing our proposal to SQL view debugging. Next, it displays a debugging session with the running example. In addition, we introduce trusted specifications as an aid to eagerly catch errors when some views are known to be valid.

4.1 Donor System

We have implemented our debugging proposal in the Datalog Educational System (DES [11, 12]), which makes it possible for Datalog and SQL to coexist as query languages for the same database. Both SQL views and Datalog predicates are seen as relations so that Datalog programs can seamlessly refer to SQL views[1]. This system has been used primarily for teaching purposes, as students can get the fundamental concepts behind a deductive database experimenting the expressiveness of both query languages. Therefore, in a single environment is possible, for instance, to experiment with the power of recursion in both SQL and Datalog, and their differences in expressing equivalent queries. [2]. Up to version 1.8.1, its deductive engine is responsible of query solving, both for Datalog and SQL queries. The latter is possible because each SQL query is compiled to a Datalog

[1]The other way round is not possible since this Datalog system in not typed and SQL has no schema information about Datalog predicates. However, a next release could handle this via either type inference or allowing the user to specify types for Datalog programs.

[2]For instance, whereas SQL views are restricted to linear recursion, Datalog clauses are not. Also, more concise Datalog predicates w.r.t. the equivalent SQL views can got in some situations [12].

program and its equivalent Datalog query is submitted instead. The last version also allows to open ODBC connections to relational database management systems (RDBMS) so that tables and views can reside in an external RDBMS. In this setting, SQL views are processed in the external system and their results cached in the DES system, retaining the possibility of submitting Datalog queries referring to SQL relations.
As an interactive system, DES seems adequate for our proposal in the interaction with the user (oracle) testing the debugging process. In addition, since it is implemented in Prolog, implementation effort of our debugging proposal is relieved. Selecting a mature, widely-used system (cf. its statistic numbers and facts) might better spread declarative debugging, both in our current proposal as in a former one [1] targeted to debug Datalog queries. So, benefits of declarative debugging can be compared, for instance, to a more "imperative" tracing of Datalog and SQL queries, as also supported in DES.

4.2 A Debugging Session

Figure 3.2 represents the computation tree for our running example, after removing repeated children following Lemma 3.2.7. n the figure, nonvalid nodes are underlined. The only buggy node (in bold face) corresponds to view *CatsAndDogsOwner*.
Although Definition 3.2.1 includes the computed answers as part of nodes, 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. In our example the associated computed answers are:

AnimalOwner

id	aname	species
1	Wilma	dog
1	Kitty	cat
2	Wilma	cat
2	Lucky	dog
3	Rocky	dog
3	Oreo	cat
4	Cecile	turtle
4	Chelsea	dog

LessThan6

id
1
2
3
4

NoCommonName

id
1
2
3

CatsAndDogsOwner

id	aname
1	Wilma
2	Lucky
3	Rocky

Guest

id	aname
1	Wilma
2	Lucky
3	Rocky

The user can type the command `/debug_sql` to debug a given view, as follows:

```
DES-SQL> /debug_sql Guest
```

The debugger builds internally the computation tree of Figure 3.2, and starts the debugging session:

```
Info: Outcome of view 'LessThan6': { 1, 2, 3, 4 }
```

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

Info: Outcome of view 'NoCommonName': { 1, 2, 3 }
Input: Is this view valid? (y/n/a) [y]: n

Info: Outcome of view 'CatsAndDogsOwner':
{ (1,'Wilma'), (2,'Lucky'), (3,'Rocky') }
Input: Is this view valid? (y/n/a) [y]: n

Info: Outcome of view 'AnimalOwner':
{
  (1,'Kitty',cat), (1,'Wilma',dog),(2,'Wilma',cat), (2,'Lucky',dog),
  (3,'Rocky',dog), (3,'Oreo',cat), (4,'Cecile',turtle), (4,'Chelsea',dog)
}
Input: Is this view valid? (y/n/a) [y]: y
Info: Buggy view found: CatsAndDogsOwner/2.
```

As can be seen, the computation tree is navigated in a top-down fashion. First, since the user checks view *LessThan6* as valid, its children are not considered anymore. Then, as the user checks view *NoCommonName* as nonvalid, its first child is visited. The user again checks this view as nonvalid, which involves the tool to visit the last possible node: *AnimalOwner*. As this is checked as valid, its closer nonvalid ascendant with all its descendants checked as valid is detected, so the tool points out node *CatsAndDogsOwner* as the buggy view. The tool has asked four questions to the user.
In the declarative debugging literature several strategies have been proposed for browsing the computation tree in order to locate the buggy nodes (see [14] for a comparison). One of the strategies which provides better efficiency in the sense of minimizing the number of questions asked to the oracle is the *Divide & Query* strategy [13], which has been incorporated into our debugger. This strategy divides the computation tree T in two trees: a subtree ST of T, and the subtree T' resulting of removing ST from T. The subtree ST is chosen such that ST and T' contain a similar number of nodes. If the root node of ST is valid (resp. non valid), ST (resp. T') is discarded, and the debugging process continues recursively with T' (resp. ST). In general, this strategy leads to a smaller number of queries to the user, noticeably with large trees. Applying this strategy to our example yields three questions instead of four as were needed in the previous case:

```
DES-SQL> /debug_sql Guest order(dq)
Info: Outcome of view 'NoCommonName': { 1, 2, 3 }
Input: Is this view valid? (y/n/a) [y]: n

Info: Outcome of view 'AnimalOwner':
{
  (1,'Kitty',cat),(1,'Wilma',dog), (2,'Lucky',dog), (2,'Wilma',cat),
  (3,'Oreo',cat), (3,'Rocky',dog), (4,'Cecile',turtle), (4,'Chelsea',dog)
}
```

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

Info: Outcome of view 'CatsAndDogsOwner':
{ (1,'Wilma'), (2,'Lucky'), (3,'Rocky') }
Input: Is this view valid? (y/n/a) [y]: n
Info: Buggy view found: CatsAndDogsOwner/2.
```

In this example, tables have been trusted, but it is also possible to ask the user for the validity of the involved tables in the debugging process via the command `/debug_sql Guest trust_tables(no)`. However, even when tables are not trusted, in this example it is not needed to ask the user for any one.

4.3 Trusted Specifications

In SQL the following scenario is very common: a set of correct views is updated to improve its efficiency or readability. The new set of views includes both new views and improved versions of some old views, which keep their names and intended answers. Sometimes the new, usually more involved system, no longer produces the expected results. We propose to use the first, reliable version, which we call a *trusted specification* during the subsequent debugging session. We have incorporated this feature into our tool. For instance, let us consider that the user has corrected our running example, which is now working properly. Now, imagine that in order to improve its efficiency, the set of views is changed by removing *AnimalOwner*, adding instead a new view *CatOrDogOwner*, and modifying *LessThan6* and *CatsAndDogsOwner*, which now make use of *CatOrDogOwner*. Figure 4.1 lists the modified and new views (*Guest* and *NoCommonName* are the same as in Figure 2.2). The intended answer of the views with the same name is kept. In the case of *CatOrDogOwner*, its intended answer is the multiset of owners with their pet names and species, but limited to cats and dogs. The very same tree as in Figure 3.2 results after replacing literals *AnimalOwner* by *CatOrDogOwner*. However, the new version is erroneous, since the *where* condition *A.species=B.species* of *CatsAndDogsOwner* should be $A.species \neq B.species$, in order to ensure that the owner has at least one dog and one cat.
Now, the user again detects an unexpected result from view *Guest* since its outcome incorrectly includes the owner with identifier 4, *Tom Cohen*. A new debugging session starts, but now the old version of the views (in the file `pets_trust`) can be used as a trusted specification as follows:

```
DES-SQL> /debug_sql Guest trust_file(pets_trust)
Info: view 'LessThan6' is valid w.r.t. the trusted file.
Info: view 'NoCommonName' is nonvalid w.r.t. the trusted file.
Info: view 'CatsAndDogsOwner' is nonvalid w.r.t. the trusted file.
Info: Outcome of view 'CatOrDogsOwner':
{
```

```
create or replace view CatsOrDogsOwner(id,aname,species) as
 select O.id, P.name, P.species
 from Owner O, Pet P, PetOwner PO
 where O.id = PO.id and P.code = PO.code
       and (species='cat' or species='dog');

create or replace view CatsAndDogsOwner(id,aname) as
 select A.id, A.aname
 from CatsOrDogsOwner A, CatsOrDogsOwner B
 where A.id=B.id and A.species=B.species;

create or replace view LessThan6(id) as
 select id from CatsOrDogsOwner
 group by id having count(*)<6;
```

Figure 4.1: Modified views for selecting dinner guests

```
  (1,'Kitty',cat),  (1,'Wilma',dog), (2,'Lucky',dog),  (2,'Wilma',cat),
  (3,'Oreo',cat),   (3,'Rocky',dog), (4,'Chelsea',dog)
}
Input: Is this view valid? (y/n/a) [y]: y
Info: Buggy view found: CatsAndDogsOwner/2.
```

Here, the tool traverses the computation tree as before, but the user is not asked for views in the set of trusted views, and the erroneous view is catched with only one check (compared to the four checks that would be needed otherwise). The debugger detects that the new version of *CatsAndDogsOwner* is erroneous.

Chapter 5

Conclusions

In this report, we propose using algorithmic debugging for finding errors in systems involving several SQL views. To the best of our knowledge, it is the first time that a debugging tool of these characteristics has been proposed. The debugger is based on the navigation of a suitable computation tree corresponding to some view returning some unexpected result. The validity of the nodes in the tree is determined by an external oracle, which can be either the user, or a trusted specification containing a correct version of part of the views in the system. The debugger ends when a buggy node, i.e., a nonvalid node with valid children, is found. We prove formally that every buggy node corresponds to an erroneous relation, and that every computation tree with a nonvalid root contains some buggy node. Although the results are established in the context of the Extended Relational Algebra, they can be easily extended to other possible SQL semantics, such as the Extended Three Valued Predicate Calculus [9].

The technique is easy to implement, obtaining an efficient, platform-independent and scalable debugger without much effort. The tool is very intuitive, because it automates what they usually do manually when an unexpected answer is found in a system with several views: check the relations in the *from* clause, and if some of them return an unexpected answer, repeat the process. Automating this process is of great help, especially when the tool includes additional features as advanced navigation strategies, or the possibility of using trusted specifications. We have successfully implemented our proposal in the existing, widely-used DES system.

The main criticisms that can be leveled at this proposal are:

- *The technique points out a view as incorrect. However, it does not indicate which part of the associated query is incorrect.* This is true, and in fact the technique would be useless in the case of a single view. Nevertheless, we think that finding an erroneous view in the context of complex systems with dozens of intermediate views is of great help. It is also worth noticing that, like in any programming paradigm, it is convenient to divide the problem in several tasks, which means in SQL defining views as simple as possible and using auxiliary views for computing intermediate results. If this is the case, finding the

bug in an erroneous view, or simply replacing the incorrect view by a new correct version, is usually an easy task.

- *The debugger can ask too many questions.* This can be a major drawback in contexts such as imperative programming, where computation trees for typical computations can contain thousands of nodes. However, this is not the case of SQL view systems. The Divide & Query strategy requires in average of $\log_2 n$ questions [14], with n the number of nodes in the computation tree. Thus, a tree involving 100 views (which corresponds to a very complex SQL system) will require an average of seven questions before the erroneous view is found.

- *Some of the questions can be very difficult to answer for the user.* This is an important point, specially considering the large number of rows in real database instances. In order to face this inconvenience, in [2] we have proposed a test-case generator for SQL views. This tool, which is also part of the system DES, generates small database instances that allow the user to check views in systems with several correlated views. If some error is found when checking the test cases, the user can start the debugging tool to locate the error.

As future work, we plan the development of a graphical interface, which can be very helpful for inspecting the computation tree providing information about the validity of the nodes.

Bibliography

[1] 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.

[2] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Applying Constraint Logic Programming to SQL Test Case Generation. In *Proc. International Symposium on Functional and Logic Programming (FLOPS'10)*, volume 6009/2010 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2010.

[3] 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.

[4] E. Codd. Relational Completeness of Data Base Sublanguages. In Rustin, editor, *Data base Systems*, Courant Computer Science Symposia Series 6. Englewood Cliffs, N.J. Prentice-Hall, 1972.

[5] U. Dayal, N. Goodman, and R. H. Katz. An extended relational algebra with control over duplicate elimination. In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 117–123, New York, NY, USA, 1982. ACM.

[6] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[7] P. W. Grefen and R. A. B. de. A multi-set extended relational algebra: a formal approach to a practical issue. In *10th International Conference on Data Engineering*, pages 80–88. IEEE, 1994.

[8] L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 3, 1997.

[9] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3):513–534, 1991.

[10] H. Nilsson. How to look busy while being lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

[11] F. Sáenz-Pérez. Datalog Educational System v2.0, August 2010. `http://des.sourceforge.net/`.

[12] F. Sáenz-Pérez. DES: A Deductive Database System. *Electronic Notes on Theoretical Computer Science*, 271:63–78, March 2011.

[13] E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.

[14] J. Silva. A Comparative Study of Algorithmic Debugging Strategies. In *Proc. of International Symposium on Logic-based Program Synthesis and Transformation LOPSTR 2006*, pages 134–140, 2007.

[15] SQL, ISO/IEC 9075:1992, third edition, 1992.

[16] A. Tessier and G. Ferrand. Declarative Diagnosis in the CLP Scheme. In P. Deransart, M. Hermenegildo, and J. Małuszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, chapter 5, pages 151–174. Springer, 2000.