Applying Constraint Logic Programming to SQL Test Case Generation*

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
{rafa,fernan}@sip.ucm.es, ygarciar@fdi.ucm.es

Abstract. We present a general framework for generating SQL query test cases using Constraint Logic Programming. Given a database schema and a SQL view defined in terms of other views and schema tables, our technique generates automatically a set of finite domain constraints whose solutions constitute the test database instances. The soundness and correctness of the technique w.r.t. the semantics of Extended Relational Algebra is proved. Our setting has been implemented in an available tool covering a wide range of SQL queries, including views, subqueries, aggregates and set operations.

1 Introduction

Checking the correctness of a piece of software is generally a labor-intensive and time-consuming work. In the case of the declarative relational database language SQL [17] this task becomes especially painful due to the size of actual databases; it is usual to find select queries involving thousands of database rows, and reducing the size of the databases for testing is not a trivial task. The situation becomes worse when we consider correlated views. Thus, generating test database instances to show the possible presence of faults during unit testing has become an important task. Much effort has been devoted to studying and improving the different possible coverage criteria for SQL queries (see [21,1] for a general discussion, [3,18] for the particular case of SQL). However, the common situation of queries defined through correlated views had not yet been considered.

In this work we address the problem of generating test cases for checking correlated SQL queries. A set of related views is transformed into a constraint satisfiability problem whose solution provides an instance of the database which will constitute a test case. This technique is known as constraint-based test data generation [7], and has already been applied to SQL basic queries [20]. Other recent works [2] use RQP (Reverse Query Processing) to generate different database instances for a given query and a result of that query. In [6] the problem of generating database test cases in the context of Java programs interacting with relational databases, focusing on the relation between SQL queries and

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S-0505/TIC/0407, S2009TIC-1465 and UCM-BSCH-GR58/08-910502

M. Blume, N. Kobayashi, and G. Vidal (Eds.): FLOPS 2010, LNCS 6009, pp. 191–206, 2010.

program values. The contributions of our work w.r.t. previous related proposals are twofold:

First, as mentioned above, the previous works focus on a single SQL query instead of considering the more usual case of a set of correlated views. Observe that the problem of test case generation for views cannot be reduced to solving the problem for each query separately. For instance, consider the two simple views, that assume the existence of a table t with one integer attribute a:

```
create view v2(c) as select v1.b from v1 where v1.b>5;
create view v1(b) as select t.a  from t  where t.a>8;
```

A *positive* test case for $v2$ considering its query as a non-correlated query could consist of a single row for $v1$ containing for instance $v1.b = 6$, since $6 > 5$ and therefore this row fulfils the $v2$ condition. However, 6 is not a possible value for $v1.b$ because $v1.b$ can contain only numbers greater than 8. Therefore the connection between the two views must be taken into account (a valid positive test case would be for instance a single row in t with $t.a = 9$).

Second, we present a formal definition of the algorithm for defining the constraints. This definition allows us to prove the soundness and (weak) completeness of the technique with respect to the Extended Relational Algebra [9].

The next section presents the basis of our SQL setting. Section 3 introduces the concept of positive and negative test cases, while Section 4 defines the constraints whose solution will constitute our test cases. This section also introduces the main theoretical result, which is proven in Appendix A. Section 5 discusses the prototype implementation and Section 6 presents the conclusions.

2 Representing SQL Queries

The first formal semantics for relational databases were based on the concept of set (e.g. relational algebra, tuple calculus [4]). However these formalisms are 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 based on multisets [5], also known in this context as *bags*, have been proposed. In this paper we adopt the point of view of the *Extended Relational Algebra* [12,9]. We start by defining the concepts of database schemas and instances but with a Logic Programming flavor. In particular the database instance rows will be considered logic substitutions of attributes by values.

2.1 Relational Database Schemas and Instances

A *table schema* is of the form $T(A_1, \ldots, A_n)$, with T the table name and A_i 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*, ...) represented by $type(T.A)$. An *instance* of a table schema $T(A_1, \ldots, A_n)$ will be represented as a finite multiset of functions (called rows) $\{\!|\mu_1, \mu_2, ..., \mu_m|\!\}$

such that $dom(\mu_i) = \{T.A_1, \ldots, T.A_n\}$, and $\mu_i(T.A_j) \in type(T.A_j)$ for every $i = 1, \ldots, m$, $j = 1, \ldots, n$. Observe that we qualify the attribute names in the domain by table names. This is done because in general we will be interested in rows that combine attributes from different tables, usually as result of cartesian products. In the following, it will be useful to consider each attribute $T.A_i$ in $dom(\mu)$ as a logic variable, and μ as a logic substitution.

The concatenation of two rows μ_1, μ_2 with disjoint domain is defined as the union of both functions represented as $\mu_1 \odot \mu_2$. Given a row μ and an expression e we use the notation $e\mu$ to represent the value obtained applying the substitution μ to e. Analogously, let S be a multiset of rows $\{\!|\mu_1, \ldots, \mu_n|\!\}$ and let e be an expression. Then $(e)S$ represents the result of replacing each attribute $T.A$ occurring in an *aggregate subexpression* of e by the multiset $\{\!|\mu_1(T.A), \ldots, \mu_n(T.A)|\!\}$. The attributes $T.B$ not occurring in aggregate subexpressions of e must take the same value for every $\mu_i \in S$, and are replaced by such value. For instance, let $e = sum(T.A)+T.B$ and $S = \{\!|\mu_1, \mu_2, \mu_3|\!\}$ with $\mu_1 = \{T.A \mapsto 2, T.B \mapsto 5\}$, $\mu_2 = \{T.A \mapsto 3, T.B \mapsto 5\}$, $\mu_3 = \{T.A \mapsto 4, T.B \mapsto 5\}$. Then $(e)S = sum(\{\!|2, 3, 4|\!\}) + 5$. If $dom(\mu) = \{T.A_1, \ldots, T.A_n\}$ and $\nu = \{U.A_1 \mapsto T.A_1, \ldots, U.A_n \mapsto T.A_n\}$ (i.e., ν is a table renaming) we will use the notation μ^U to represent the substitution composition $\nu \circ \mu$. The previous concepts for substitutions can be extended to multisets of rows in a natural way. For instance, given the multiset of rows S and the row μ, $S\mu$ represents the application of μ to each member of the multiset.

A *database schema* D is a tuple $(\mathcal{T}, \mathcal{C}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of tables, $\mathcal{C}$ a finite set of database constrains and $\mathcal{V}$ a finite set of views (defined below). In this paper we consider only *primary key* and *foreign key* constraints, defined as traditionally in relational databases (see Subsection 4.1). A *database instance* d of a database schema is a set of table instances, one for each table in $\mathcal{T}$ verifying $\mathcal{C}$ (thus we only consider *valid instances*). To represent the instance of a table T in d we will use the notation $d(T)$. A *symbolic database instance* d_s is a database instance whose rows can contain logical variables. We say that d_s is satisfied by a substitution μ when $(d_s\mu)$ is a database instance. μ must substitute all the logic variables in d_s by domain values.

2.2 Extended Relational Algebra and SQL Queries

Next we present the basics of Extended Relational Algebra (ERA from now on) [12,9] which will be used as semantics of our framework. There are other approaches for defining SQL semantics such as [14], but we have chosen ERA because it provides an *operational* semantics very suitable for proving the correctness of our technique. Let R and S be multisets. Let μ be any row occurring n times in R and m times in S. Then ERA consists of the following operations:

- Unions and intersections. The union of R and S, is a multiset $R \cup S$ in which the row μ occurs $n + m$ times. Analogously $R \cap S$, the intersection of R and S, is a multiset in which the row μ occurs $min(n, m)$ times.

- Projection. The expression $\pi_{e_1 \mapsto A_1, \ldots, e_n \mapsto A_n}(R)$ produces a new relation producing for each row $\mu \in R$ a new row $\{A_1 \mapsto e_1\mu, \ldots, A_n \mapsto e_n\mu\}$. The resulting multiset has the same number of rows as R.
- Selection. Denoted by $\sigma_C(R)$, where C is the condition that must be satisfied for all rows in the result. The selection operator on multisets applies the selection condition to each row occurring in the multiset independently.
- Cartesian products. Denoted as $R \times S$, each row in the first relation is paired with each row in the second relation.
- Renaming. The expression $\rho_S(R)$ changes the name of the relation R to S and the expression $\rho_{A/B}(R)$ changes the name of the attribute A of R to B.
- Aggregate operators. These operators are used to aggregate the values in one column of a relation. Here we consider sum, avg, min, max and count.
- Grouping operator. Denoted by γ, this operator allows us to consider the rows of a relation in groups, corresponding to the value of one or more attributes and to aggregate only within each group. This operation is denoted by $\gamma_L(R)$, where L is a list of elements, each one either a grouping attribute, that is, an attribute of the relation R to which the γ is applied, or an aggregate operator applied to an attribute of the relation. To provide a name for the attribute corresponding to this aggregate in the result, an arrow and a new name are appended to the aggregate. It is worth observing that $\gamma_L(R)$ will contain one row for each maximal group, i.e., for each group not strictly contained in a larger group.

A relational database can be consulted by using queries and views defined over other views and queries. Queries are select SQL statements. In our setting we allow three kind of queries:

- *Basic queries* of the form:

  $$\mathsf{Q} = \mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1\ , \ldots, R_m\ B_m\ \mathsf{where}\ C_w;$$

  with R_j tables or views for $j = 1 \ldots m$, e_i, $i = 1 \ldots n$ expressions involving constants, predefined functions and attributes of the form $B_j.A$, $1 \leq j \leq m$, and A an attribute of R_j. The meaning of any query Q in ERA is denoted $\langle Q \rangle$. In the case of basic queries is

  $$\langle Q \rangle = \Pi_{e_1 \to E_1, \ldots, e_n \to E_n}(\sigma_{C_w}(R))$$

  where $R = \rho_{B_1}(R_1) \times \ldots \times \rho_{B_m}(R_m)$.
- *Aggregate queries*, including group by and having clauses:

  $$\mathsf{Q} = \mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1\ , \ldots, R_m\ B_m\ \mathsf{where}\ C_w\ \mathsf{group\ by}\ A'_1, \ldots, A'_k\ \mathsf{having}\ C_h;$$

  In this case, the equivalent ERA expression is the following:

  $$\langle Q \rangle = \Pi_{e'_1 \to E_1, \ldots, e'_n \to E_n}(\sigma_{C'_h}(\gamma_L(\sigma_{C_w}(R))))$$

  where $L = \{A'_1, \ldots, A'_k, u_1 \mapsto U_1, \ldots, u_l \mapsto U_l\}$, R defined as in the previous case, u_i, $1 \leq i \leq l$ the aggregate expressions occurring either in the select or in the having clauses, U_i new attribute names, e'_j, $j = 1 \ldots n$ the result of replacing each occurrence of u_i by U_i in e_j and analogously for C'_h.

– *Set queries* of the form $Q = V_1$ {union, intersection} V_2;
with V_1, V_2 views (defined below) with the same attribute names. The meaning of set queries in ERA is represented by $\cup$ and $\cap$ multiset operators for union and intersection, respectively: $\langle Q \rangle = \langle V_1 \rangle \ \{\cup, \cap\} \ \langle V_2 \rangle$

In order to simplify our framework we assume queries such that:

– Without loss of generality we assume that the where and having clauses only contain existential subqueries of the form exists Q (or not exists Q). It has been shown that other subqueries of the form ... in Q, ... any Q or ... all Q can be translated into equivalent subqueries with exists and not exists (see for instance [10]). Analogously, subqueries occurring in arithmetic expressions can be transformed into exists subqueries.
– The from clause does not contain subqueries. This is not a limitation since all the subqueries in the from clause can be replaced by views.
– We also do not allow the use of the distinct operator in the select clause. It is well-known that queries using this operator can be replaced by equivalent aggregate queries without distinct. In the language of ERA, this means that the operator δ for eliminating duplicates –not used here– is a particular case of the aggregate operator γ (see [9]).
– Our setting does not allow: recursive queries, the minus operator, join operations, and null values. All these features, excepting the recursive queries, can be integrated in our setting, although they have not been considered here for simplicity.

We also need to consider the concept of *views*, which can be thought of as new tables created dynamically from existing ones by using a query and allowing the renaming of attributes.

The general form of a view is: create view V($A_1, \ldots, A_n$) as Q, with Q a query and $V.A_1, \ldots V.A_n$ the name of the view attributes. Its meaning is defined as: $\langle V \rangle = \Pi_{E_1 \to V.A_1, \ldots, E_n \to V.A_n} \langle Q \rangle$, with $E_1, \ldots, E_n$ the attribute names of the select clause in Q. In general, we will use the name *relation* to refer to either a table or a view. The semantics of a table T in a given instance d is defined simply as its rows: $\langle T \rangle = d(T)$. A view query can depend on the tables of the schema and also on previously defined views (no recursion between views is allowed). Thus, 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 the from clause of its query. This concept can be easily extended to queries by assuming some arbitrary name labeling the root node, and to tables, where the dependency tree will be a single node labeled by the table name.

3 SQL Test Cases

In the previous section we have defined an operational semantics for SQL. Now we are ready for defining the concept of test case for SQL. We distinguish between positive and negative test cases:

Definition 1. *We say that a non-empty database instance d is a* positive test case *(PTC) for a view V when $\langle V \rangle \neq \emptyset$.*

Observe that our definition excludes implicitly the empty instances, which will be considered as neither positive nor negative test cases. We require that the (positive or negative) test case contains at least one row that will act as witness of the possible error in the view definition. The overall idea is that we consider d a PTC for a view when the corresponding query answer is not empty. In a basic query this means that at least one tuple in the query domain satisfies the where condition. In the case of aggregate queries, a PTC will require finding a valid aggregate verifying the having condition, which in turn implies that all its rows verify the where condition. If the query is a set query, then the ranges are handled according to the set operation involved.

The negative test cases (NTC) are defined by modifying the initial queries and then applying the concept of positive test case. With this purpose we use the notation Q_{C_w} and $Q_{(C_w,C_h)}$ to indicate that C_w is the where condition in Q and C_h is the having condition in Q (when Q is an aggregate query). If Q_{C_w} is of the form select e_1 E_1, ..., e_n E_n from R_1 B_1 , ..., R_m B_m where C_w; then the notation $Q_{not(C_w)}$ represents select e_1 E_1, ..., e_n E_n from R_1 B_1 , ..., R_m B_m where not(C_w); and analogously for $Q_{(C_w,C_h)}$ and $Q_{(not(C_w),C_h)}$, $Q_{(C_w,not(C_h))}$, and $Q_{(not(C_w),not(C_h))}$. For instance, in the case of basic query, we expect that a NTC will contain some row in the domain of the view not verifying the where condition:

Definition 2. *We say that a database instance d is a NTC for a view V with associated basic query Q_{C_w} when d is a PTC for $Q_{not(C_w)}$.*

In queries containing aggregate functions, the negative case corresponds either to a tuple that does not satisfy the where condition, or to an aggregate not satisfying the having condition:

Definition 3. *We say that a database instance d is a NTC for a view V with associated aggregate query $Q_{(C_w,C_h)}$ if it is a PTC for either $Q_{(not(C_w),C_h)}$, $Q_{(C_w,not(C_h))}$, or $Q_{(not(C_w),not(C_h))}$.*

Next is the definition of negative test cases for set queries:

Definition 4. *We say that a database instance d is a NTC for a view with query defined by:*

- *A query union of Q_1, Q_2, if d is a NTC for both Q_1 and Q_2.*
- *A query intersection of Q_1, Q_2, if d is a NTC for either Q_1 or Q_2.*

The main advantage of defining NTCs in terms of PTCs is that only a positive test case generator must be implemented. The previous definitions are somehow arbitrary depending on the coverage. For instance the NTCs for views with aggregate queries $Q_{(C_w,C_h)}$ could be defined simply as the PTCs for $Q_{(not(C_w),not(C_h))}$.

It is possible to obtain a test case which is both positive and negative at the same time thus achieving *predicate coverage* with respect to the where and having conditions (in the sense of [1]). We will call these tests PNTCs. For instance, for the query select A from T where A=5; with T a table with a single attribute A, the test case d s.t. $d(T) = \{|\mu_1, \mu_2|\}$ with $\mu_1 = \{T.A \mapsto 5\}$, $\mu_2 = \{T.A \mapsto X\}$, X any value different from 5, is a PNTC. However, this is not always possible. For instance, the query select R_1.A from T R_1 where R_1.A=5 and not exists (select R_2.A from T R_2 where R_2.A<>5); allows both PTCs and NTCs but no PNTC. Our tool will try to generate a PNTC for a view first, but if it is not possible it will try to obtain a PTC and a NTC separately.

4 Generating Constraints

The main goal of this paper is to use Constraint Logic Programming for generating test cases for SQL views. The process can be summarized as follows:

1. First, create a *symbolic database instance*. Each table will contain an arbitrary number of rows, and each attribute value in each row will correspond to a fresh logic variable with its associated domain integrity constraints.
2. Establish the constraints corresponding to the integrity of the database schema: primary and foreign keys.
3. Represent the problem of obtaining a test case as a constraint satisfaction problem.

Next, we explain in detail phases 2 and 3.

4.1 Primary and Foreign Keys

Given a relation R with primary key $pk(R) = \{A_1, \ldots, A_m\}$ and a symbolic instance d such that $d(R) = \{|\mu_1, \ldots, \mu_n|\}$, we check that d satisfies $pk(R)$ by establishing the following constraint:

$$\bigwedge_{i=1}^{n} \Big(\bigwedge_{j=i+1}^{n} \Big(\bigvee_{k=1}^{m} (\mu_i(R.A_k) \neq \mu_j(R.A_k)) \Big) \Big)$$

that is, different rows must contain different values for the primary key. Given two relations R_1, R_2 and an instance d such that $d(R_1) = \{|\mu_1, \ldots, \mu_{n_1}|\}$, $d(R_2) = \{|\nu_1, \ldots, \nu_{n_2}|\}$ a *foreign key* from R_1 referencing R_2, denoted by $fk(R_1, R_2) = \{(A_1, \ldots, A_m), (B_1, \ldots, B_m)\}$, indicates that for each row μ in R_1 there is a row ν in R_2 such that $(A_1\mu, \ldots, A_m\mu) = (B_1\nu, \ldots, B_m\nu)$. Foreign keys are represented with the following constraints:

$$\bigwedge_{i=1}^{n_1} \Big(\bigvee_{j=1}^{n_2} \Big(\bigwedge_{k=1}^{m} \big(\mu_i(R_1.A_k) = \nu_j(R_2.B_k) \big) \Big) \Big)$$

4.2 SQL Test Cases as a Constraint Satisfaction Problem

Now we are ready for describing the technique supporting our implementation. First we introduce the two following auxiliary operations over multisets:

Definition 5. *Let* $A = \{\!|(a_1, b_1), \ldots, (a_n, b_n)|\!\}$. *Then we define the operations* Π_1 *and* Π_2 *as follows:* $\Pi_1(A) = \{\!|a_1, \ldots, a_n|\!\}$, $\Pi_2(A) = \{\!|b_1, \ldots, b_n|\!\}$.

The following definition will associate a first order formula to every possible row of a relation. The idea is that the row will be in the relation instance iff the formula is satisfied.

Definition 6. *Let* D *be a database schema and* d *a database instance. We define* $\theta(R)$ *for every relation* R *in* D *as a multiset of pairs* (ψ, u) *with* ψ *a first order formula, and* u *a row. This multiset is defined as follows:*

1. *For every table* T *in* D *such that* $d(T) = \{\!|\mu_1, \ldots, \mu_n|\!\}$*:*

$$\theta(T) = \{\!|(true, \mu_1), \ldots, (true, \mu_n)|\!\}$$

2. *For every view* $V =$ create view $V(A_1, \ldots, A_n)$ as Q,

$$\theta(V) = \theta(Q)\{V.A_1 \mapsto E_1, \ldots, V.A_n \mapsto E_n\}$$

   with $E_1, \ldots, E_n$ *the attribute names in the* select *clause of* Q.
3. *If* Q *is a basic query of the form:*

$$\text{select } e_1\ E_1, \ldots, e_n\ E_n \text{ from } R_1\ B_1, \ldots, R_m\ B_m \text{ where } C_w;$$

   Then:

$$\theta(Q) = \{\!|(\psi_1 \wedge \ldots \wedge \psi_m \wedge \varphi(C_w, \mu),\ s_Q(\mu)) \mid (\psi_1, \nu_1) \in \theta(R_1), \ldots, (\psi_m, \nu_m) \in \theta(R_m), \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}|\!\}$$

   with $s_Q(\mu) = \{E_1 \mapsto (e_1\mu), \ldots, E_n \mapsto (e_n\mu)\}$, *and the first order formula* $\varphi(C, \mu)$ *is defined as*

   - *if* C *does not contain subqueries,* $\varphi(C, \mu) = C'\mu$, *with* C' *obtained from* C *by replacing every occurrence of* `and` *by* $\wedge$, `or` *by* $\vee$, *and* `not` *by* $\neg$.
   - *if* C *does contain subqueries, let* Q= (exists Q_E) *be an outermost existential subquery in* C, *with* $\theta(Q_E) = \{\!|(\psi_1, \mu_1), \ldots (\psi_n, \mu_n)|\!\}$. *Let* C' *be the result of replacing* Q *by* true *in* C. *Then* $\varphi(C, \mu) = (\vee_{i=1}^{n} \psi_i) \wedge \varphi(C', \mu)$.

4. *For set queries:*
   - $\theta(V_1$ union $V_2) = \theta(V_1) \cup \theta(V_2)$ *with* $\cup$ *the multiset union.*
   - $(\psi, \mu) \in \theta(V_1$ intersection $V_2)$ *with cardinality* k *iff* $(\psi_1, \mu) \in \theta(V_1)$ *with cardinality* k_1, $(\psi_2, \mu) \in \theta(V_2)$ *with cardinality* k_2, $k = min(k_1, k_2)$ *and* $\psi = \psi_1 \wedge \psi_2$.

5. *If Q includes aggregates, then it is of the form:*

$$\begin{array}{l}\mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1, \ldots, R_m\ B_m\\ \mathsf{where}\ C_w\ \mathsf{group\ by}\ e'_1, \ldots, e'_k\ \mathsf{having}\ C_h\end{array}$$

Then we define:

$$\begin{array}{rl} P = & \{\!|(\psi, \mu)\ \mid\ (\psi_1, \nu_1), \ldots, (\psi_m, \nu_m) \in (\theta(R_1) \times \ldots \times \theta(R_m)) \\ & \qquad \psi = \psi_1 \wedge \ldots \wedge \psi_m,\ \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}\ |\!\} \\ \theta(Q) = & \{\!|(\bigwedge(\Pi_1(A)) \wedge aggregate(Q, A),\ s_Q(\Pi_2(A)))\ |\ A \subseteq P\ |\!\}\end{array}$$

$$\begin{array}{l} aggregate(Q, A) = group(Q,\Pi_2(A)) \wedge maximal(Q,A) \wedge \varphi(C_h, \Pi_2(A)) \\ group(Q, S) = (\bigwedge\{\!|\varphi(C_w, \mu)\ \mid \mu \in S|\!\}\ ) \wedge \\ \qquad\qquad (\bigwedge\{\!|\ ((e'_1)\nu_1 = (e'_1)\nu_2 \wedge \ldots \wedge (e'_k)\nu_1 = (e'_k)\nu_2)\ \mid\ \nu_1, \nu_2 \in S|\!\}\ ) \\ maximal(Q, A) = \bigwedge\{\!|\ (\neg\psi \vee \neg group(Q, \Pi_2(A) \cup \{\!|\mu|\!\})\ \mid\ (\psi, \mu) \in (P - A)\ |\!\}\end{array}$$

Observe that the notation $s_Q(x)$ with Q a query is a shorthand for the row μ with domain $\{E_1, \ldots, E_n\}$ such that $(E_i)x = (e_i)x$, with $i = 1 \ldots n$, with select $e_1\ E_1,\ \ldots,\ e_n\ E_n$ the select clause of Q. If E_i's are omitted in the query, it is assumed that $E_i = e_i$.

Example 1. Let V_1 , V_2 , V_3 and V_4 be four SQL views defined as:

```
create view V1(A1, A2) as
        select T1'.A E1, T1'.B E2
        from T1 T1'
        where T1'.A ≥ 10

create view V2(A) as
        select T2'.C E1
        from V1 V1', T2 T2'
        where V1'.A1 + T2'.C = 0

create view V3(A) as
        select(V1'.A1) E
        from V1 V1'
        where exists
              (select T2'.C E1
               from T2 T2'
               where T2'.C = V1'.A1)

create view V4(A) as
        select V1'.A2 E
        from V1 V1'
        where V1'.A2 = "a"
        group by V1'.A2
        having sum(V1'.A1) > 100;
```

Suppose table T_1 has the attributes A, B while table T_2 has only one attribute C. Consider the following symbolic database instances $d(T_1) = \{\!|\mu_1, \mu_2|\!\}$ and $d(T_2) = \{\!|\mu_3, \mu_4|\!\}$ with: $\mu_1 = \{T_1.A \mapsto x_1, T_1.B \mapsto y_1\}$, $\mu_2 = \{T_1.A \mapsto x_2, T_1.B \mapsto y_2\}$ and $\mu_3 = \{T_2.C \mapsto z_1\}$, $\mu_4 = \{T_2.C \mapsto z_2\}$. Then:

$$\theta(T_1) = \{\!|(true, \mu_1), (true, \mu_2)|\!\}, \quad \theta(T_2) = \{\!|(true, \mu_3), (true, \mu_4)|\!\}$$

$$\begin{array}{rl} \theta(V_1) = & \{\!|\ (x_1 \geq 10, \{V_1.A_1 \mapsto x_1, V_1.A_2 \mapsto y_1\})\ , \\ & \quad (x_2 \geq 10, \{V_1.A_1 \mapsto x_2, V_1.A_2 \mapsto y_2\})\ \ |\!\} \\ \theta(V_2) = & \{\!|\ (x_1 \geq 10 \wedge x_1 + z_1 = 0, \{V_2.A \mapsto z_1\}), \\ & \quad (x_1 \geq 10 \wedge x_1 + z_2 = 0, \{V_2.A \mapsto z_2\}), \\ & \quad (x_2 \geq 10 \wedge x_2 + z_1 = 0, \{V_2.A \mapsto z_1\}), \\ & \quad (x_2 \geq 10 \wedge x_2 + z_2 = 0, \{V_2.A \mapsto z_2\})\ \ |\!\}\end{array}$$

$$\theta(V_3) = \{\!|\ (x_1 \geq 10 \wedge ((z_1 = x_1) \vee (z_2 = x_1)), \{V_3.A \mapsto x_1\}),\\ (x_2 \geq 10 \wedge ((z_1 = x_2) \vee (z_2 = x_2)), \{V_3.A \mapsto x_2\})\ |\!\}$$
$$\theta(V_4) = \{\!|(\psi_1, \{V_4.A \mapsto y_1\}), (\psi_2, \{V_4.A \mapsto y_1\}), (\psi_3, \{V_4.A \mapsto y_2\})|\!\}$$
$$\psi_1 = (x_1 \geq 10 \wedge x_2 \geq 10) \wedge (y_1 = \text{“}a\text{”} \ \wedge\ y_2 = \text{“}a\text{”} \ \wedge\ y_1 = y_2) \wedge\\ (x_1 + x_2 > 100)$$
$$\psi_2 = (x_1 \geq 10) \wedge (y_1 = \text{“}a\text{”}) \wedge\\ (\neg(x_2 \geq 10) \vee \neg(y_1 = \text{“}a\text{”} \ \wedge\ y_2 = \text{“}a\text{”} \ \wedge\ y_1 = y_2)) \wedge (x_1 > 100)$$
$$\psi_3 = (x_2 \geq 10) \wedge (y_2 > \text{“}a\text{”}) \wedge\\ (\neg(x_1 \geq 10) \vee \neg(y_1 = \text{“}a\text{”} \ \wedge\ y_2 = \text{“}a\text{”} \ \wedge\ y_1 = y_2)) \wedge (x_2 > 100)$$

For instance observe that V_4 has an aggregate query with a *group by* over V_1. Since $\theta(V_1)$ contains 2 tuples, $\theta(V_4)$ contains three possible tuples, one for each possible group in V_1: the first group containing the two rows in V_1, the second corresponding only to the first row, and the third possibility a group containing only the second row in V_1.

The following result and its corollary represent the main result of this paper, stating the soundness and completeness of our proposal:

Theorem 1. *Let D be a database schema and d a database instance. Assume that the views and queries in D do not include subqueries. Let R be a relation in D. Then $\mu \in \langle R \rangle$ with cardinality k iff $(\mu, true) \in \theta(R)$ with cardinality k.*

Proof. See Appendix A.

The restriction to queries without subqueries is due to the limitations of ERA. The following corollary contains the idea for generating constraints that will yield the PTCs:

Corollary 1. *Let D be a database schema and d_s a symbolic database instance. Assume that the views and queries in D do not include subqueries. Let R be a relation in D such that $\theta(R) = \{\!|(\psi_1, \mu_1), \ldots (\psi_n, \mu_n)|\!\}$, and η a substitution satisfying d_s. Then $d_s\eta$ is a PTC for R iff $(\bigvee_{i=1}^{n} \psi_i)\eta = true$.*

Proof. Straightforward from Theorem 1: $(\bigvee_{i=1}^{n} \psi_i)\eta = true$ iff there is some ψ_i with $1 \leq i \leq n$ such that $\psi_i\eta = true$ iff $(\mu_i\eta) \in \langle R \rangle$ iff $\langle R \rangle \neq \emptyset$.

5 Implementation and Prototype

In this section, we comment on some aspects of our implementation and show a system session with actual results of the test case generator.

Our test case generator is bundled as a component of the Datalog deductive database system DES [15]. The input of the tool consists of:

- A database schema D defined by means of SQL language, i.e., a finite set of tables $\mathcal{T}$, constraints $\mathcal{C}$ and views $\mathcal{V}$, as well as the integrity constraints for the columns (primary and foreign keys).
- A SQL view V for which the test case is to be generated.

DES [15] is implemented in Prolog and includes a SQL parser for queries and views, and a type inference system for SQL views. In this way we benefit from the DES facilities for dealing with SQL and at the same time we can exploit the constraint solving features available in current Prolog implementations. As a first step, we have chosen SICStus Prolog as a suitable platform for our development (although others will be handled in a near future).

As explained in Section 4, we do need constraints that include a mix of conjunctions and disjunctions. We use *reification* to achieve an efficient implementation of these connectives. Thus, we reify every atomic constraint and transform conjunctions and disjunctions of constraints into finite domain constraints of the form $B_1 * \ldots * B_k \geq B_0$, and $B_1 + \ldots + B_k \geq B_0$, respectively. B_0 allows a compact form to state the truth or falsity of these constraints.

Apart from the constraints indicated in Section 4 we also need to consider *domain integrity constraints*, the constraints that restrict the given set of values a table attribute can take. These values are represented by a built-in datatype, e.g., string, integer, and float. On the one hand, types in SQL are *declared* in create table statements. In addition, further domain constraints can be declared, which can be seen as subtype declarations, as the *column constraint* $A > 0$, where A is a table attribute with numeric type. On the other hand, types are *inferred* for views.

Up to now, we support integer and string datatypes by using the finite domain ($\mathcal{FD}$) constraint system available in SICStus Prolog. Although with a few changes this can also be easily mapped to Ciao Prolog, GNU Prolog and SWI-Prolog. Posting our constraints over integers to the underlying $\mathcal{FD}$ constraint solver is straightforward. In the case of string constraints we map each different string constant in the SQL statements to a unique integer, allowing equality and disequality ($\mathcal{FD}$) constraints. This mapping is stored in a dictionary before posting constraints for generating the test cases. Then, string constants are replaced by integer keys in the involved views. Generation and solving of constraints describing the test cases in the integer domain follows. Before displaying the instanced result involving only integers, the string constants are recovered back by looking for these integer keys in the dictionary. If some key values are not in the dictionary they must correspond to new strings. The tool generates new string constants for these values. Our treatment is only valid for equalities and disequalities, and it does not cover other common string operations such as the concatenation or the *LIKE* operator which will require a string constraint solver (see [8] for a discussion on solving string constraints involving the *LIKE* operator).

Our tool allows the user to choose the type of test case to be generated, either PTC, or NTC or both PNTC for any view V previously defined in D. The output is a database instance d of a database schema D such that d is a test case for the given view V with as few entries as possible.

For instance, consider the following session:

```
DES-SQL> CREATE OR REPLACE TABLE t(a INT PRIMARY KEY, b INT);
DES-SQL> CREATE OR REPLACE VIEW u(a1, a2) AS SELECT a, b
         FROM t WHERE a >= 10;
DES-SQL> CREATE OR REPLACE VIEW v(a) AS SELECT a2 FROM u
         WHERE a2 = 88 GROUP BY a2 HAVING SUM(a1) > 0;
```

Then, test cases (both positive and negative) for the view `v` can be obtained via the following command:

```
DES-SQL> /test_case v
Info: Test Case over integers:
[t([[1000,88],[999,1000]])]
```

Here, we get the PNTC `[t([[1000,88],[999,1000]])]`. If it is not possible to find a PNTC, the tool would try to generate a PTC and a NTC separately.

Observe that in practice our system cannot reach completeness, but only weak completeness module the size of the tables of the instance. That is, our system will find a PTC if it is possible to construct it with all the tables containing a number of rows less than an arbitrary number. By default the system starts trying to define PTCs with the number of rows limited to 2. If it is not possible, the number of rows is increased. The process is repeated stopping either when a PTC is found or when an upper bound is reached (by default 10). Both the lower and the upper limits are user configurable.

6 Conclusions and Future Work

We have presented a novel technique for generating finite domain constraints whose solutions correspond to test cases for SQL relations. Similar ideas have been suggested in other works but, to the best of our knowledge, not for views, which corresponds to more realistic applications. We have formally defined the algorithm for producing the constraints, and have proved the soundness and weak completeness of the approach with respect to the operational semantics of Extended Relational Algebra. Another novelty of our approach is that we allow the use of string values in the query definitions. Although constraint systems over other domains, as reals or rationals, are available, we have not used them in our current work. However, they can be straightforwardly implemented. In addition, enumerated types (available in object-oriented SQL extensions) could also be included, following a similar approach to the one taken for strings.

Our setting includes primary and foreign keys, existential subqueries, unions, intersections, and aggregate queries, and can be extended to cover other SQL features not included in this paper. For instance, null values can be considered by defining an extra *null table* T_{null} containing the logic variables that are null, and taking into account this table when evaluating expressions. For instance, a condition $T.A = T'.B$ will be translated into $(T.A = T'.B) \wedge (T.A \notin T_{null}) \wedge (T'.B \notin T_{null})$.

Dealing with recursive queries is more involved. One possibility could be translating the SQL views into a logic language like Prolog, and then use a technique for generating test cases for this language [11]. However, aggregate queries are not easily transformed into Prolog queries, and thus this approach will only be useful for non-aggregate queries.

It is well-known that the problem of finding complete sets of test cases is in general undecidable [1]. Different coverage criteria have been defined (see [1] for a survey) in order to define test cases that are complete at least w.r.t. some desired property. In this work, we have considered a simple criterion for SQL queries, namely the *predicate coverage criterium*. However, it has been shown [19] that other coverage criteria can be reduced to predicate coverage by using suitable query transformations. For instance, if we look for a set of test cases covering every atomic condition in the where clause of a query Q, we could apply our tool to a set of queries, each one containing a where clause containing only one of the atomic conditions occurring in Q.

A SICStus Prolog prototype implementing these ideas has been reported in this paper, which can be downloaded and tested (binaries provided for both Windows and Linux OSs) from http://gpd.sip.ucm.es/yolanda/research.htm. To allow performance comparisons and make the sources for different Prolog platforms available, an immediate work is the port to Ciao, GNU Prolog and SWI-Prolog.

Although test case generation is a time consuming problem, the efficiency of our prototype is reasonable, finding in a few seconds TCs for views with dependence trees of about ten nodes and with a number of rows limited to seven for every table. The main efficiency problem comes from aggregate queries, where the combinatorial problem of selecting the aggregates can be too complex for the solver. To improve this point, even when efficiency of the SICStus constraint solver is acknowledged, there are more powerful solvers in the market. In particular, we plan to test the industrial, more efficient $\mathcal{FD}$ and $\mathcal{R}$ IBM ILOG solvers [13], which allow to handle bigger problems at a faster rate than SICStus solvers. Also, another striking state-of-the-art, free, and open-source $\mathcal{FD}$ solver library to be tested is Gecode [16].

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2008)
2. Binnig, C., Kossmann, D., Lo, E.: Towards automatic test database generation. IEEE Data Eng. Bull. 31(1), 28–35 (2008)
3. Cabal, M.J.S., Tuya, J.: Using an SQL coverage measurement for testing database applications. In: Taylor, R.N., Dwyer, M.B. (eds.) SIGSOFT FSE, pp. 253–262. ACM, New York (2004)
4. Codd, E.: Relational Completeness of Data Base Sublanguages. In: Rustin, R. (ed.) Data base Systems. Courant Computer Science Symposia Series 6, Prentice-Hall, Englewood Cliffs (1972)

5. Dayal, U., Goodman, N., Katz, R.H.: An extended relational algebra with control over duplicate elimination. In: PODS 1982: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems, pp. 117–123. ACM, New York (1982)
6. Degrave, F., Schrijvers, T., Vanhoof, W.: Automatic generation of test inputs for mercury, pp. 71–86 (2009)
7. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), 900–910 (1991)
8. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: ISSTA 2007: Proceedings of the 2007 international symposium on Software testing and analysis, pp. 151–162. ACM, New York (2007)
9. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall PTR, Upper Saddle River (2008)
10. Gogolla, M.: A note on the translation of SQL to tuple calculus. SIGMOD Record 19(1), 18–22 (1990)
11. Gómez-Zamalloa, M., Albert, E., Puebla, G.: On the generation of test data for prolog by partial evaluation. CoRR, abs/0903.2199 (2009)
12. Grefen, P.W.P.J., de By, R.A.: A multi-set extended relational algebra: a formal approach to a practical issue. In: 10th International Conference on Data Engineering, pp. 80–88. IEEE, Los Alamitos (1994)
13. ILOG CP 1.4, `http://www.ilog.com/products/cp/`
14. Negri, M., Pelagatti, G., Sbattella, L.: Formal semantics of SQL queries. ACM Trans. Database Syst. 16(3), 513–534 (1991)
15. Sáenz-Pérez, F.: Datalog educational system. user's manual version 1.7.0. Technical report, Faculty of Computer Science, UCM (November 2009), `http://des.sourceforge.net/`
16. Schulte, C., Lagerkvist, M.Z., Tack, G.: Gecode, `http://www.gecode.org/`
17. SQL, ISO/IEC 9075:1992, third edn. (1992)
18. Suárez-Cabal, M., Tuya, J.: Structural coverage criteria for testing SQL queries. Journal of Universal Computer Science 15(3), 584–619 (2009)
19. Tuya, J., Suárez-Cabal, M.J., de la Riva, C.: Full predicate coverage for testing SQL database queries. Software Testing, Verification and Reliability (2009) (to be published)
20. Zhang, J., Xu, C., Cheung, S.C.: Automatic generation of database instances for white-box testing. In: COMPSAC, pp. 161–165. IEEE Computer Society, Los Alamitos (2001)
21. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys 29, 366–427 (1997)

A Proof of Theorem 1

In this Appendix we include the proof of our main theoretical result. The theorem establishes a bijective mapping between the rows obtained by applying the ERA semantics to a relation R defined on an schema with instance d and the tuples $(true, \sigma)$ in $\theta(R)$ (see Definition 6). Before proving the result we introduce an auxiliary Lemma:

Lemma 1. *Let D be a database schema and d a database instance, $R_1, \ldots, R_m$ relations verifying Theorem 1, $B_1, \ldots, B_m$ attribute names, and R an expression*

in ERA defined as $R = \rho_{B_1}(R_1) \times \ldots \times \rho_{B_m}(R_m)$. *Let* P *be a multiset defined as*

$$P = \{\!|(\psi, \mu) \mid (\psi_1, \nu_1), \ldots, (\psi_m, \nu_m) \in (\theta(R_1) \times \ldots \times \theta(R_m)) \\ \psi = \psi_1 \wedge \ldots \wedge \psi_m,\ \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m} |\!\}$$

Then $\mu \in R$ *with cardinality* k *iff* $(true, \mu) \in P$ *with cardinality* k.

Proof. $(true, \mu) \in P$ with cardinality k iff there are pairs $(\psi_i, \nu_i) \in \theta(R_i)$ with cardinality c_i for $i = 1 \ldots m$ such that $k = c_1 \times \ldots \times c_m$ and $\mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}$. From the conditions of P we have that $\psi = true$ iff $\psi_i = true$ for $i = 1 \ldots m$. By hypothesis $(true, \nu_i) \in \theta(R_i)$ with cardinality c_i iff $\nu_i \in R_i$ with cardinality c_i for $i = 1 \ldots m$, iff $({\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}) \in (\rho_{B_1}(R_1) \times \ldots \times \rho_{B_m}(R_m))$ with cardinality $c_1 \times \ldots \times c_m$, i.e., $\mu \in R$ with cardinality k.

Next we prove the Theorem by induction on the number of nodes of the dependence tree for R. If this number is 1 (basis) then R is a table T, $\langle T \rangle = d(T)$, and the result is an easy consequence of Definition 6 item 1. If the dependence tree contains at least two nodes (inductive case) R cannot be a table. We distinguish cases depending on the form of R:

- R *aggregate query.* Then Q is of the form

$$\mathsf{Q} = \mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1\ , \ldots, R_m\ B_m \\ \mathsf{where}\ C_w\ \mathsf{group\ by}\ A'_1, \ldots, A'_k\ \mathsf{having}\ C_h$$

Then $\langle Q \rangle = \Pi_{e'_1 \to E_1, \ldots, e'_n \to E_n}(\sigma_{C'_h}(\gamma_L(\sigma_{C_w}(R))))$, with $R = \rho_{B_1}(R_1) \times \ldots \times \rho_{B_m}(R_m)$, $L = \{A'_1, \ldots, A'_k, u_1 \mapsto U_1, \ldots, u_l \mapsto U_l\}$, u_i the aggregate expressions occurring either in the select or in the having clauses for $i = 1 \ldots l$, U_i new attribute names for $i = 1 \ldots l$, e'_j the result of replacing each occurrence of u_i in e_j, $1 \leq j \leq n$ by U_i and analogously for C'_h. From Definition 6, item 5 we have

$$\theta(Q) = \{\!|(\bigwedge(\Pi_1(A)) \wedge aggregate(Q, A),\ s_Q(\Pi_2(A))) \mid A \subseteq P |\!\}$$

Let $\mu \in \langle Q \rangle$ with cardinality k. Then there are rows $\nu_1, \ldots, \nu_r$ such that μ is of the form $\mu = (\nu_i)\{E_1 \mapsto e'_1, \ldots, E_n \mapsto e'_n\}$, $1 \leq i \leq r$ with $\nu_i \in (\sigma_{C'_h}(\gamma_L(\sigma_{C_w}(R))))$, with cardinality c_i for $i = 1 \ldots r$ and $k = c_1 + \ldots + c_r$. From the definition of γ we have that the c_i occurrences of ν_i for $i = 1 \ldots r$ correspond to the existence of c_i maximal aggregates $S_i^j \subseteq \sigma_{C_w}(R)$, $j = 1 \ldots c_i$.

Observe for every $\eta \in S_i^j$ we have that the cardinality of η in S_i^j and in R is the same because S_i^j is maximal. Then from Lemma 1 we have that the set $A_i^j = \{\!|(true, \eta) \mid \eta \in S_i^j |\!\}$ verifies $A_i^j \subseteq P$ for $i = 1 \ldots r$, $j = 1 \ldots c_i$. Then it is immediate that $\bigwedge(\Pi_1(A_i^j)) = true$ and that $s_Q(\Pi_2(A_i^j)) = s_Q(S_i) = \{E_1 \mapsto ((e_1)S_i^j), \ldots, E_n \mapsto ((e_n)S_i^j)\} = (\nu_i)\{E_1 \mapsto e'_1, \ldots, E_n \mapsto e'_n\} = \mu$. Then we have that $(true \wedge aggregate(Q, A_i^j),\ \mu) \in \theta(Q)$ for $i = 1 \ldots r$, $j = 1 \ldots c_i$. It remains to check that $aggregate(Q, A_i^j) = true$, i.e., that

- $group(Q, \Pi_2(A_i^j)) = true$. $\Pi_2(A_i^j) = S_i^j$ and the definition of group requires that all the rows in S_i^j verify the where condition and that every row takes

the same values for the grouping attributes. The first requirement is a consequence of $S_i^j \subseteq \sigma_{C_w}(R)$, while the second one holds because we are assuming that the multiset S_i^j was selected has a valid group by the operator γ.
- $maximal(Q, A_i^j) = true$. The auxiliary definition *maximal* indicates that no other element of the form $(true, \mu')$ from P can be included in S_i^j verifying that we still have the same values for the grouping attributes and μ' verifying the where condition. This is true because if there were such $(true, \mu') \in P - A_i^j$, then by Lemma 1 μ' will be in R and S_i^j will not be maximal in $\sigma_{C_w}(R)$ as required by γ.
- $\varphi(C_h, \Pi_2(A_i^j)) = true$. Observe that $\varphi(C_h, \Pi_2(A_i^j)) = \varphi(C_h, S_i^j)$, and that in the absence of subqueries φ only checks that the S_i^j verify the having condition C_h, which is true because ν_i verifies C_h'.

Then we have $(true, \mu) \in \theta(Q)$ for $i = 1 \ldots r$, $j = 1 \ldots c_i$ and thus $(true, \mu) \in \theta(Q)$ with cardinality k .
The converse result, i.e., assuming $(true, \mu) \in \theta(Q)$ with cardinality k and proving that then $\mu \in\in \langle Q \rangle$ with cardinality k, is analogous.

- R *basic query.* Similar to the previous case.
- $R = V_1$ *union* V_2. Then $\langle R \rangle = \langle V_1 \rangle \cup \langle V_2 \rangle$, $\theta(R) = \theta(V_1) \cup \theta(V_2)$ and the result follows by induction hypothesis since V_1, V_2 are children of R in its dependence tree.
- $R = V_1$ *intersection* V_2 . Then

$$\begin{aligned} \langle R \rangle &= \langle V_1 \rangle \cap \langle V_2 \rangle \\ \theta(R) &= \{\!|(\psi_1 \wedge \psi_2 \wedge \nu_1 = \nu_2,\, \nu_1) \mid \ (\psi_1, \nu_1) \in \theta(V_1), (\psi_2, \nu_2) \in \theta(V_2)|\!\} \end{aligned}$$

Then $\mu \in \langle R \rangle$ with cardinality k iff $\mu \in \langle V_1 \rangle$ and $\mu \in \langle V_2 \rangle$ with cardinalities k_1, k_2 respectively and $k = min(k_1, k_2)$. By the induction hypothesis $(true, \mu) \in \theta(V_1)$ with cardinality k_1, $(true, \mu) \in \theta(V_2)$ with cardinality k_2 and this happens iff $(true, \mu) \in \theta(R)$.
- R *is a view* V *with associated query* Q. Then $\langle V \rangle = \Pi_{E_1 \to V.A_1, \ldots, E_n \to V.A_n} \langle Q \rangle$ and $\theta(V) = \theta(Q)\{V.A_1 \mapsto E_1, \ldots, V.A_n \mapsto E_n\}$ with $E_1, \ldots, E_n$ the attribute names of the select clause in Q. We have proved above that $\mu \in \langle Q \rangle$ iff $(true, \mu) \in \theta(Q)$ with the same cardinality. Now observe that for every $\mu \in \langle Q \rangle$ applying the projection $\Pi_{E_1 \to A_1, \ldots, E_n \to A_n}$ produces a renaming of its domain $E_1, \ldots, E_n$ to $A_1, \ldots A_n$, and that this is the same as $\mu\{V.A_1 \mapsto E_1, \ldots, V.A_n \mapsto E_n\}$.