Electronic Communications of the EASST
Volume 55 (2012)

Proceedings of the
XII Spanish Conference on Programming and Computer Languages
(PROLE 2012)

Tabling with Support for Relational Features in a Deductive Database

Fernando Sáenz-Pérez

16 pages

Guest Editors: María-del-Mar Gallardo
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Tabling with Support for Relational Features in a Deductive Database

Fernando Sáenz-Pérez[1][*]

Grupo de programación declarativa (GPD),
Dept. Ingeniería del Software e Inteligencia Artificial,
Universidad Complutense de Madrid, Spain[1]

Abstract: Tabling has been acknowledged as a useful technique in the logic programming arena for enhancing both performance and declarative properties of programs. As well, deductive database implementations benefit from this technique for implementing query solving engines. In this paper, we show how unusual operations in deductive systems can be integrated with tabling. Such operations come from relational database systems in the form of null-related (outer) joins, duplicate support and duplicate elimination. The proposal has been implemented as a proof of concept rather than an efficient system in the Datalog Educational System (DES) using Prolog as a development language and its dynamic database.

Keywords: Tabling, Outer Joins, Duplicates, Relational databases, Deductive databases, DES

1 Introduction

Tabling is a useful implementation technique embodied in several current logic programming systems, such as B-Prolog [ZS03], Ciao [GCH+08], Mercury [SS06], XSB [SW10], and Yap Prolog [RSC05], to name just a few. This technique walks by two orthogonal axes: performance and declarative properties of programs. Tabling enhances the former because repeated computations are avoided since previous results are stored and reused. The latter axis is improved because order of goals and clauses are not relevant for termination purposes. In fact, tabled computations in the context of finite predicates and bounded term depths are terminating, a property which is not ensured in top-down SLD computations.

Deductive database implementations with Datalog as query language have benefited from tabling [RU93, SSW94a, SP11] as an appropriate technique providing performance and a framework to implement query meaning. Terminating queries is a common requirement for database users. Also, the set oriented answer approach of a tabled system is preferred to the SLD one answer at a time.

However, "relational" database systems embed features which are not usually present altogether in deductive systems. These include duplicates, which were introduced to account for bags of data (multisets) instead of sets. Also, the need for representing absent or unknown information delivered the introduction of null values and outer join operators ranging over such

[*] This author 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)

values. Finally, aggregate functions allow to compute summarized data in terms of (multi)sets and considering null occurrences. So, these systems are not relational anymore as they depart from the original relational model, where these features are not considered. In addition, the introduction by major vendors of some of these features in databases are claimed as error sources and unnecessary [Dat09], but it is a fact that they are widely used by database practitioners. However, this is rather a debate out of the scope of this paper.

Thus, the aim of this paper is to show how such features can be supported altogether in a tabled deductive system with Datalog as a query language. To this end, we base our presentation on the grounds of DES (Datalog Educational System) [SP11], a system implemented in Prolog which runs on different platforms (both Prolog and OS's), which allows to easily test new engine implementations. This work extends [SP12b] by describing how tabling is implemented and introducing (tabled) duplicates in the context of a deductive system dealing with nulls. Supported Prolog platforms along time include Ciao, GNU Prolog, SICStus Prolog and SWI-Prolog. Because it was thought to be as platform independent as possible, tabling in particular was implemented as no supported platform provided it (Ciao recently added this feature). Also, proprietary systems as SICStus Prolog do not provide open sources to modify parts of the system, as it would be the case for implementing tabling.

The very first motivation for including such features in this system came for the need to support SQL as a query language in a deductive database. In DES, both Datalog and SQL are supported, and SQL statements are compiled to Datalog programs and eventually solved by the deductive inference engine. So, embodying nulls, outer joins and duplicates became a need. However, as the system was intended for educational purposes since its inception, it is not targeted at performance and also lacks features such as concurrency, security and others that a practical database system must enjoy.

Next section introduces DES whereas Section 3 describes its basic implementation of tabling stemmed from [Die87]. Section 4 explains the tabled support for nulls and outer join operations, and Section 5 do the same for duplicates. Section 6 gives some hints for performance improvement. Finally, Section 7 draws some conclusions and points out some future work.

2 Datalog Educational System

The Datalog Educational System (DES) [SP11] is a free, open-source, multiplatform, portable, in-memory, Prolog-based implementation of a deductive database system. DES 3.0 [SP12a] is the next shortcoming release, which enjoys Datalog and SQL query languages, full recursive evaluation with tabling, types, integrity constraints, stratified negation [Ull88], persistency, full-fledged arithmetic, ODBC connections and novel approaches to Datalog and SQL declarative debugging [CGS08, CGS11], test case generation for SQL views [CGS10], null value support, outer join and aggregate predicates and functions [SP11].

DES implements Datalog with stratified negation as described in [Ull88] with safety checks [Ull88, ZCF+97] and source-to-source program transformations for rule simplification, safety and compilation. Evaluation of queries is ensured to be terminating as long as no infinite predicates/operators are considered and since only atomic domains are supported (currently, only the infix operator "`is`" represents an infinite relation).

A reasonable set (for education purposes) of SQL following ISO standard SQL:1999 is supported (further revisions of the standard cope with issues such as XML, triggers, and cursors, which are outside of the scope of DES). SQL row-returning statements are compiled to and executed as Datalog programs (basics can be found in [Ull88]), and relational metadata for DDL statements are kept. Submitting such a query amounts to 1) parse it, 2) compile to a Datalog program including the relation `answer`/n with as many arguments as expected from the SQL statement, 3) assert this program, and 4) submit the Datalog query `answer(`$X_1,\ldots,X_n$`)`, where $X_i : i \in \{1,\ldots,n\}$ are n fresh variables. After its execution, this Datalog program is removed. On the contrary, if a data definition statement for a view is submitted, its translated program and metadata do persist. This allows Datalog programs to seamlessly use views created at the SQL side (also tables since predicates are used to implement them). The other way round is also possible if types are declared for predicates.

There are available some usual built-in comparison operators (=, \=, >, . . .). When being solved, all these operators demand ground (variable-free) arguments (i.e., no constraints are allowed up to now) but equality, which performs unification. In addition, arithmetic expressions are allowed via the infix operator `is`, which relates a variable/number with an arithmetic expression. The result of evaluating this expression is assigned/compared to the variable. The predicate `not/1` implements stratified negation. Other built-ins include outer joins and aggregates.

3 Tabling-based Query Solving

The computational model of DES follows a top-down-driven bottom-up fixpoint computation with tabling, which follows the ideas found in [SD91, Die87, TS86]. In its current form, it can be seen as an extension of the work in [Die87] in the sense that, in addition, it deals with a modified algorithm for negation, undefined (although incomplete) information, nulls and aggregates. Also, instead of translating each tabled predicate for including fixpoint and memoization management as in [Die87], Datalog rules are stored as dynamic predicates and other predicates explicitly deal with fixpoint computation as shown next.

3.1 Tabling

DES uses an extension table (following the nomenclature in [Die87], but also known as extension table) which stores answers to goals previously computed, as well as their calls. For the ease of the introduction, we assume an extension table (`ET` implemented with predicate `et`/1) and a call table (`CT`, implemented with predicate `called`/1) to extensionally store answers and calls, respectively. Also, annotations for completed computations and its handling, which prevents some unnecessary computations, are omitted. Answers may be positive or negative, that is, if a call to a positive goal G succeeds, then, the fact G is added as an answer to the extension table; if a negated goal `not(`G`)` succeeds, then the fact `not(`G`)` is added. Negative facts are deduced when a negative goal is proven by means of negation as failure (closed world assumption (CWA) [Ull88]). Both positive and negative facts cannot occur in a stratifiable program [Ull88]. Calls are also added to the call table whenever they are solved. This allows to detect whether a call has been previously solved and the computed results in the extension table (if any) can be reused.

So, repeated answers are not kept in the extension table.

First occurrence during computation of a tabled goal is known as a *generator* whilst further occurrences of subsumed goals are known as *consumers*. A generator is responsible of building all the *different* answers which will be used by its consumers eventually.

The algorithm implementing this idea (following *ET* algorithm in [Die87]) is depicted next:

```
% Already called. Call table with an entry for the current call
memo(G) :-
  build(G,Q),     % Build in Q the same call with fresh variables
  called(Q),      % Look for a unifiable call in CT for the current call
  subsumes(Q,G),  % Test whether CT call subsumes the current call
  !,              %
  et_lookup(G).   % If so, use the results in extension table (ET)
% New call. Call table without an entry for the current call
memo(G) :-
  assertz(called(G)),      % Assert the current call to CT
 ( (et_lookup(G))          % First call returns all previous answers in ET
  ;
   (solve_goal(G),         % Solve the current call using applicable rules
    build(G,Q),            % Build in Q the same call with fresh variables
    no_subsumed_by_et(Q),  % Test whether there is no entry in ET for Q
    et_assert(G),          % If so, assert the current result in ET
    et_changed)).          % Flag the change
```

First clause tests whether there is a previous call that subsumes the current call. For this, `build` constructs a term which is a copy up to variable renaming (i.e., implemented with `copy_term`; more on this later when dealing with nulls). Predicate `subsumes`/2 on the left of Figure 1 implements term subsumption, where a general term subsumes a specific term `st` if grounding of `st` variables (as, e.g., via `numbervars`) makes them unifiable. There are two possibilities: 1) There is such a previous call subsuming the current one: then, use the result in the extension table, if any. To this end, predicate `et_lookup`/1 is implemented as a simple call to the predicate `et`/1 (`et_lookup(G) :- et(G).`) It is possible that there is no such a result (for instance, when computing the goal `p` in the program `p :- p`) and no more tuples can be deduced. 2) Otherwise, process the new call. Second clause of `memo` stores the new call in `CT` and then, returns all previous answers in `ET` (from a previous fixpoint iteration). Next, it solves the goal with the program rules (recursively applying this algorithm). Once the goal has been solved (if succeeded), it stores the computed answer if there is no any previous answer subsuming the current one (note that, via recursion, we can deliver new answers for the same call). Subsumption is now checked with predicate `no_subsumed_by_et`/1, shown on the right of Figure 1. The whole process is known as a memoization process and will also be referred to as the *memo* function.

```
subsumes(General,Specific) :-          no_subsumed_by_et(Q,G) :-
  \+ \+ (make_ground(Specific),           \+ ((et_lookup(Q),
         General=Specific).                     subsumes(Q,G))).
```

Figure 1: Predicates `subsumes` and `no_subsumed_by_et`

3.2 Fixpoint Computation

The memo function is insufficient in itself for computing all possible answers to a goal since incomplete information is used from the goals in its defining rules (as these goals can be mutually recursive). Therefore, it is needed to ensure that all the possible information is deduced by finding a fixpoint of this function, which is implemented as shown next (following ET^* [Die87]):

```
solve_star(Q,St) :-
  repeat,
  (remove_calls,    % Clear CT
   et_not_changed,  % Flag ET as not changed
   solve(Q,St),     % Solve the call to Q using memoization at stratum St
   fail             % Request all alternatives
  ;
   no_change,       % If no more alternatives, start a new iteration
   !, fail).        % Otherwise, fail and exit
```

First, the call table is emptied to try to obtain new answers for a given call, preserving the previous computed answers. Then, the memo function is applied (via predicate `solve`/2), possibly providing new answers. If the extension table remains the same as before after this last memo function application, we are done. Otherwise, the memo function is reapplied as many times as needed until no changes are found in the extension table. Upon exiting, the extension table contains the meaning of the query (plus perhaps other meanings for the relations used in the computation of the given query).

Predicate `solve` is defined as a straight call to the memo function but for built-ins (that are left apart from the memoization process as would otherwise be a resource waste) and conjunctive goals (which recursively calls itself).

3.3 Dependency Graph and Stratification

Each time a database changes, a predicate dependency graph (PDG) is built [ZCF+97]. This graph shows the dependencies between predicates in the program. Each node in this graph is a program predicate symbol and there are as many nodes as such symbols. Arcs come from each predicate in a rule body (antecedent) to its rule predicate. Arcs are labeled as either negative, if the antecedent node occurs negated, or positive otherwise. This dependency graph is used to looking for a stratification for the program [ZCF+97]. A stratification collects predicates into numbered strata $(1 \ldots N)$ so that, given the function $strata(\mathtt{p})$ which assigns a strata number to predicate p, then for a positive arc $\mathtt{p} \leftarrow \mathtt{q}$, $strata(\mathtt{p}) \leq strata(\mathtt{q})$, and for a negative arc $\mathtt{p} \overset{\neg}{\leftarrow} \mathtt{q}$, $strata(\mathtt{p}) < strata(\mathtt{q})$. A cycle in this graph containing a negative arc amounts to a non-stratifiable program.

A naïve bottom-up computation would solve all of the predicates in stratum 1, then 2, and so on, until the meaning of the whole program is found. However, the implementation of DES only resort to compute by stratum when a negative dependency occurs in the predicate dependency graph, restricted to the query, as shown next:

```
solve_stratified(Query) :-
  sub_pdg(Query,(_Nodes,Arcs)),
  (neg_dependencies(Arcs) -> solve_star(Query,1)
   ;
```

```
    strata(St), sort_by_strata(St,Arcs,Preds),
    build_queries(Preds,Query,Queries), solve_star_list(Queries)).
```

Here, predicate `sub_pdg`/2 gets the current PDG restricted to the query. Predicate `neg_dependencies`/1 tests whether there are negative dependencies in the subgraph. Predicate `strata`/1 gets the current stratification. Predicates in the sub-PDG are sorted w.r.t. this stratification with `sort_by_strata`/2. Then `build_queries` build a list of queries for sorted predicates (an atom with fresh variables for each predicate) appended to the input query. The call to `solve_star_list`/1 solves each of these queries in order by successively calling `solve_star`/2 with each query and its corresponding stratum number.

4 Nulls and Outer Joins

Unknownness has been handled in relational databases long time ago because its ubiquitous presence in real-world applications. Despite its claimed dangers due to unclean semantics (see, e.g., the discussion in [Dat09]), null values to represent unknowns have been widely used. Also, interest in including nulls in logic programming has been stated some time ago [TG94].

Supporting nulls conducts to also provide built-ins to handle them, as outer join operations. DES includes the common outer join operations in relational databases, providing the very same semantics for outer join operators ranging over null values, which are described next.

4.1 Null Semantics

A null value represents unknown data. To include such values into relational database systems (RDBMS's), a new logical value is added for unknown results, leading to a three-valued logic (3VL, for `true`, `false` and `unknown`). Any comparison operator (=, <, ...) relating at least a null value should return the `unknown` logic value [Dat09]. Although a 3VL is assumed for RDBMS's (Oracle, DB2, SQL Server, MySQL, ...), the fact is that the implemented logic does not account for the unknown logic value as it is represented by the null value [Dat09].

However, as we are interested in allowing outer join operations and we rely on a logic engine with 2VL (two-valued logic), we restrict to this, so that any comparison relating at least a null value returns `false` instead of `unknown`. Truth tables for usual logical operators (`not`, `and` and `or`) remain thus as for 2VL. Regarding comparison operators, two (*distinct*) null values are not (known to be) equal, and are (not known to be) distinct. Thus, neither `null = null` (syntactic equality) nor `null \= null` (syntactic disequality) hold. Further, for the same null value, the equality should succeed, as in the conjunctive query `X=null,X=X`. Evaluation of a given expression including at least one null value always returns the same concrete null value. Thus, two expressions are considered equal if they are syntactically equal. This covers, for instance, that the following query succeeds: `X=null,X+1=X+1`.

4.2 Null Representation

Nulls are internally represented with the term `'$NULL'(Id)`, where *Id* is a unique integer which does not occur in any other null. This representation is similar to that also suggested in other systems [SWSJ09], but, as a difference, DES considers null as a first class citizen and its internal representation is hidden from the user. Therefore, asserting or consulting a rule as `p(null)` is directly allowed. Since the null value in this rule receives a unique identifier, the conjunctive query `p(X),X=X` succeeds, since `X` stands for the same unknown value (note that this is in contrast to the flaw in SQL, where `SELECT * FROM p WHERE x=x` discards tuples with a null in `x`).

Any explicit null occurring in either a program or a query is replaced by its internal representation during parsing. Internal representations are also allowed to be written for implementors purposes, but irrespective of the user-provided identifier (which can also be a variable), it is replaced by a unique identifier. Also, when building a new fresh call in predicate `build`/2 (cf. Section 3.1), not only variables have to be fresh, but also any occurrence of a null value. Therefore, this predicate also includes a *null provider* for such occurrences, where concrete null identifiers are replaced by variable identifiers, as `$NULL(V))`, where `V` is a variable. A null provider argument in a rule means that each tuple generated by that rule (and therefore added to the extension table) gets a unique null for that argument eventually. However, along fixpoint iterations, the non-ground null representation is the one to be stored in the extension table. Only once the fixpoint has been reached, nulls are grounded for the answer to be shown to the user. This is in contrast to asserting or consulting a rule containing a null argument, as `p(null)`, where the rule is stored as `p($NULL(N))`, where `N` is a concrete number. Users are precluded from using null generators, which are only available as a result of preprocessing, but it will be needed along the tabled computations of outer joins.

4.3 Outer Join Built-ins

Three outer join operations are provided, following well-known relational database query languages (SQL and extended relational algebra): left (`lj`/3), right (`rj`/3) and full (`fj`/3) outer joins. A left outer join `lj(L,R,C)` computes the cross-product of two relations `L` and `R` that satisfy a third relation `C`, extended with some special tuples including nulls as explained next. Tuples in `L` which have no counterpart in `R` w.r.t. `C` are included in the result, so that the values corresponding to columns of `R` are set to `null`. The right outer join `rj(L,R,C)` is equivalent to `lj(R,L,C)` (it is only provided as a syntactic sugar), and the full outer join `fj(L,R,C)` is equivalent to `lj(L,R,C)` ∪ `rj(L,R,C)`. In addition, both `L` and `R` can take the form of such constructions in order to allow more neat, nested applications of outer joins.

Outer joins are non-monotonic metapredicates which may deliver incorrect results in a bottom-up computation. Let's consider an iterative, bottom-up computation of the fixpoint for the left outer join `lj(L,R,C)`. In a given cycle of the fixpoint computation for this outer join, a tuple l in `L` might not find a matching tuple in `R` because this relation might not be completely computed, so that a tuple t with nulls in place of the arguments of `R` would be added to the current interpretation for the result of the left outer join. However, a further cycle may develop new tuples for `R` that do match with l. Therefore, the incorrectly added tuple t should be removed (non-monotonicity). So, in order to prevent speculative computations and removing entries from the extension table which are not longer true due to new entries added along fixpoint iterations, the meaning of involved relations in an outer join are required to be computed already before computing the meaning of the outer join. This can be achieved by taking advantage of the stratification idea: relations in an outer join are collected into a lower stratum as if they were negative atoms. That way, before computing an outer join built-in, its arguments are already computed because they belong to a lower stratum (cf. Section 3.3).

4.4 Outer Join Transformations

This section introduces a source-to-source transformation (in a preprocessing phase) for solving the left outer join (other outer joins are analogous), rather than resorting to write (Prolog-)specific code for this. As it is well-known, a single left or right outer join suffices to express others.

A new predicate $p*i* is introduced as an argument of the built-in, void predicate lj/1, which does nothing, but is handy to specify a predicate classification in strata. So, the predicate $p*i* is to be set in a deeper strata than the predicate of the rule in which it occurs, say of predicate p, because the negative arc $p*i*$\overset{\neg}{\leftarrow}$p is added to the dependency graph. The call lj($p*i*) is solved by predicate solve/2 as a built-in, with a straight call to $p*i* (no entries are added to the extension table for lj/1). Next, predicate $p*i* is defined to compute the outer join. All of the facts in the meaning of $p*i* come from two sources: the facts in L joined with those of R that meet O, and the facts in L joined with nulls that *do not* meet O. Next example shows how these data are collected for solving the outer join v(X,Y) :- lj(s(X,U),t(V,Y),U>V):

```
v(X,Y)                           :- lj('$p0'(X,U,V,Y)).
'$p0'(A,B,'$NULL'(C),'$NULL'(D)) :- s(A,B), not('$p1'(A,B,E,F)).
'$p0'(A,B,C,D)                   :- '$p1'(A,B,C,D).
'$p1'(A,B,C,D)                   :- s(A,B), t(C,D), B > C.
```

Predicate $p0 is source of the facts either provided by the positive case (a *straight* call to $p1 from the second rule of $p0), or by the negative case (a *negated* call to $p1 in the first rule of $p0). This negated call oughts $p1 to be in a lower strata than $p0. Therefore, before computing $p0, the meaning of $p1 is completely available. Predicate $p1 contains the (possible) hard stuff to be computed since it contains the Cartesian product of two relations, followed by the condition. Despite its arrangement, which may yield to think of a bad computational behavior (compute all tuples from s, then all from t, and finally filter results), the top-down driven computation looks for a tuple from s, then a tuple from t, and only adds a new tuple to the extension table of '$p1' if the condition B > C holds. Indeed, this is quite similar to the RDB implementations of join operations (modulo indexing). The first rule for '$p0' builds the null values for the arguments of the right relation R for which no tuples are found meeting condition O, i.e., So, it is a null provider, as it contains specifications with the form $NULL(V), where V is a variable. If there are more than one tuple in L that does not match with R, each one is therefore joined with a non-ground null tuple. If the null ground representation was instead considered, then the same null tuples will be appended to the result, breaking the assessment that null values should all be unique.

Notice that this transformation includes floundering [BD98] in the first rule for '$p0': the call to not('$p1'(A,B,E,F)), where variables E and F are not range restricted. However, floundering in this concrete case poses no problem as the call to $p1 is completely computed *before* it is used by any other call and no other negated call occurs in the program. Note that the other call in the program to $p1 is for the positive case where all of its arguments become ground. In particular, the negated call will use those results and the corresponding negative entries will be added to the extension table. Such negated entries are not be reused by any other (negated) calls in the program because they belong to system-generated predicates. Safety checking takes such floundering into account, avoiding error messages. Other works treat the floundering problem in a more general use of negation (see, e.g., constructive negation [LAC99] and also tabled query evaluation [Dam96]), where non-ground negated calls are possibly involved in recursive calls, which we do not consider in our setting.

Other deductive systems, such as DLV [LPF^{+}06], might benefit from including outer joins as well. In this case, floundering programs are not allowed, but for true negation (CWA is not assumed; instead, negative data are explicitly declared). Fortunately, as pointed out in [Ull88], programs as above can be transformed into non-floundering programs, where all calls to negated goals are ensured to be ground. Next program shows this transformation, where non-relevant variables are dropped and unfolding is applied:

```
v(X,Y)                             :- '$p0'(X,U,V,Y).
'$p0'(A,B,'$NULL'(C),'$NULL'(D))  :- s(A,B), not('$p1'(B)).
'$p0'(A,B,C,D)                     :- s(A,B), t(C,D), B > C.
'$p1'(B)                           :- s(A,B), t(C,D), B > C.
```

However, comparing this version to the example, even when the number of relations does not increase, extra computation has to be done in the second clause of `$p0`. So, although it seems possible to compute outer joins in DLV with this technique, nulls should be natively supported; otherwise it couldn't be applied because there is no provision to get unique identifiers for null values in this system (DLV does not feature a general-purpose programming language, but a deductive language).

XSB [SSW94b] is another system which supports non-ground semantics allowing floundering programs with the use of the special negation `sk_not`/1, which automatically produces a similar translation as explained before [SWSJ09]. To write outer joins in this system, in particular it is needed to generate unique identifier integer numbers for the null values and declare as tabled the predicates involved in the computation of the outer join. The following program implements the outer join example in XSB:

```
:- table('$p0'/4), table('$p1'/4), table(s/2), table(t/2).
main(Vs) :- findall(v(X,Y),v(X,Y),Vs).
v(X,Y)   :- '$p0'(X,U,V,Y).
'$p0'(A,B,'$NULL'(C),'$NULL'(D)) :-
    get_id(C), get_id(D), s(A,B), sk_not('$p1'(A,B,E,F)).
'$p0'(A,B,C,D) :- '$p1'(A,B,C,D).
'$p1'(A,B,C,D) :- s(A,B), t(C,D), B > C.
:- dynamic id/1.
id(0).
get_id(X) :- id(X), retractall(id(X)), Y is X+1, assertz(id(Y)).
```

Here, the main entry point (predicate `main`/1) returns a list of deduced facts via the metapredicate `findall`, which collects all answers to the goal `v(X,Y)`. Predicate `get_id` returns a new integer each time it is called, therefore allowing to uniquely identify nulls.

5 Duplicates

Allowing tables to contain duplicate rows and queries returning also duplicates is a common feature in current RDBMS's. However, the introduction of duplicates is claimed to suffer some other issues: Duplicates in a table are repeated rows in a relation and, from a logical viewpoint, have no sense because repeated rows mean the same[1]. Another issue with duplicates is that equivalent-intended statements can deliver a different number of duplicate rows [Dat09]. As well, they preclude query optimizations and make optimizers much more complicated than if were if no duplicates were allowed. Nonetheless, duplicates are useful in a number of situations, for instance, when considering aggregates.

Noticeably, whilst in relational databases they are assumed, they are not usual in deductive databases (mainly because of the claimed issues), where they are removed by default. However, there are deductive systems supporting duplicates as LDL++, but duplicates are removed from recursive rules. As a main difference, DES also allows recursive rules to be generators of duplicates in a similar way as in SQL recursive statements. Since duplicates are not removed from derived relations, each rule is understood as a possible, distinct duplicate generator. When duplicates are disabled, they are discarded along computation, i.e., subsumed answers are not added to the extension table. Next subsection shows how this behavior is supported in a tabled system.

5.1 Duplicates and Tabling

An alternative for supporting duplicates in extensional predicates it to distinguish each rule in the program, so that two repeated rules are not considered to be equivalent w.r.t. subsumption. To this end, we can add a unique *rule identifier* to each rule in the program, as in the program `1:p(a), 2:p(a), 3:p(b)`. Then, an entry in the extension table will contain tuples of the form (*Atom*, *RuleId*), where *Atom* is the answer and *RuleId* the rule identifier that generated that answer. In this example, the entries obtained for the call `p(X)` are: {(`p(a)`,1), (`p(a)`,2), (`p(b)`,3)}. So, the first and second answers do not subsume each other and the usual behavior for answer subsumption can be kept. From a user viewpoint, rule identification is of no use (as in relational databases), so that they are hidden when displaying answers and listing rules.

As an example of a recursive predicate, let's consider the rules {`p(a),p(a),p(X):-p(X)`} (two facts and a recursive rule, respectively). Its intended meaning is the multiset containing the tuple `p(a)` four times, where two tuples correspond to the extensional rules for `p` and the other two to the single intensional rule for `p`. This intensional rule generates one tuple for each extensional rule. By adding the rule `p(X) :- p(X)` once more, the meaning of `p` would contain `p(a)` ten times, i.e., it contains: two tuples from the two facts, and four tuples for each recursive rule. The first recursive rule is source of four tuples because of the two facts and the two tuples from the second recursive rule (analogously for the second recursive rule). In fact, this mimics SLD resolution by collecting all possible answers coming from *different* sources, therefore pruning infinite computation paths, i.e., all possible computation paths are considered, stopping when a (recursive) node already used in the computation is reached. Figure 2 shows the tabling tree for the query `p(X)` and the annotated program `1.p(a), 2.p(a), 3.p(X) :- p(X)`, and `4.p(X) :- p(X)`. Infinite computations are elided in this case because the rule with identifier `3` is neither reused for solving its body nor the body of rule `4` as this rule is a descendant of `3` (analogously for rule `4`).

The idea above about uniquely identifying each rule can be applied to recursive predicates

[1] Citing Codd: "If something is true, saying it twice doesn't make it any more true."

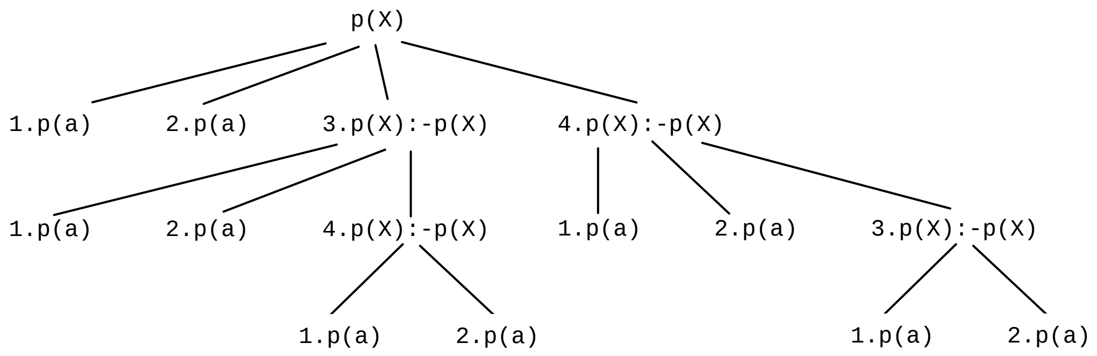

Figure 2: Tabling tree for the query `p(X)`

as well. One possible solution is to keep track of the computation path by annotating the rules which have been used to deduce a given atom. So, each entry in the extension table can be identified by a chain of such rule identifiers. Each deduced atom is associated with a pair (*Id*, *IdChain*), where *Id* is the identifier of the rule which is the generator of the atom, and *IdChain* is the list of as many pairs as goals in rule *Id*. So, all rule identifiers in the computation path up to the tabling tree leaf are stored. Referring to Figure 2, the following are the entries stored in the extension table after computation, from left to right: (`p(a)`, (1,[])), (`p(a)`, (2,[])), (`p(a)`, (3,[(1,[])])), . . . , (`p(a)`, (4,[(3,[(2,[])])])).

To implement this, two additional parameters to predicate `memo`/1 are added: `D` for selecting whether or not duplicate answers for a relation are requested (`all`, `distinct`, resp.) or eliminate duplicates for a given set of arguments (`distinct(Vs,Ps)`), and `Id` as the identifier pair as described above. Its new prototype is `memo(+G,+D,-Id)`. These two parameters are passed to `et_lookup` (which is further explained in Subsection 5.3) in the first clause of this predicate and also to `no_subsumed_by_et(Q,D,(G,Id))`, which is modified as follows:

```
% no_subsumed_by_et(+Query,+Distinct,+(Goal,IdGoal))
no_subsumed_by_et(Q,all,(_G,_IdG)) :- % No entry matching Q
  \+ et_lookup(Q,all,_IdQ), !.
no_subsumed_by_et(Q,all,(G,IdG)) :-   % Existing entries matching Q
  nr_id(IdG),  % Check that IdG is not cyclic
  \+ (et_lookup(Q,all,IdQ), my_subsumes((Q,IdQ),(G,IdG))).
no_subsumed_by_et(Q,_D,(G,_IdG)) :-   % For distinct answers
  \+ (et_lookup(Q,all,_IdQ), my_subsumes(Q,G)).
```

5.2 Duplicates and Nulls

When duplicates are disabled, a null value is considered as a single constant in answers, so that repeated entries are removed. For example, if an answer contains two occurrences of `p(null)`, only one is shown. This follows the SQL criterium when applying the clause `DISTINCT` in a `SELECT` statement. If duplicates are enabled, each null is considered as a different constant identified by its internal numeric representation. Thus, in this example, the user would obtain two tuples. However, tabling computes both tuples in both cases. By enabling development listings, which in particular shows null internal representations, in the former case both tuples are listed.

5.3 Duplicate Elimination

When duplicates are enabled, duplicate elimination is provided with the built-in `distinct/1`, which applies to a positive atom. In this case, duplicate elimination policy for an atom `A` consists of discarding all duplicates for `A` and all of its descendants in the tabling tree. To implement this policy, function `memo` is added with a parameter indicating whether duplicates are discarded (`distinct`) or not (`all`). When a `distinct(`*G*`)` call is to be solved with predicate `solve`, a call to this function is called with this parameter, which is passed to subsequent descendant calls to the memo function in the same subtree. Because there can be other calls to `A` not involved in a duplicate elimination path, all its answers are computed by the generator, and consumers involved in duplicate elimination are responsible of using non-repeated entries. So, each consumer uses the modified predicate `et_lookup`, as shown next, including a new second parameter selecting whether or not all duplicate answers are requested:

```
% et_lookup(+Goal,+Distinct,-Id)
et_lookup(G,all,IdG)       :- et(G,IdG).
et_lookup(G,distinct,IdG) :-
  findall((G,IdG),et(G,IdG),GIdGs), setof(G,IdG^member((G,IdG),GIdGs),Gs),
  member(G,Gs), once(member((G,IdG),GIdGs)).
```

This predicate is called from predicate `memo`/3 in two scenarios: First, for consumers that get all entries in the extension table for a subsumed call (first clause of `memo`) or from a previous fixpoint iteration (first call in second clause of `memo`), both with the value `distinct` for the second argument of `et_lookup`. In this first case and when distinct answers are required, second clause of `et_lookup` is applied. Second case is for generators (second call in second clause of `memo`), where all answers are required and this predicate is called with second argument with `all`. For providing distinct entries in `ET`, first all entries matching the input goal are collected with `findall`, along with their identifiers. Next, `setof` filters duplicated goals. Each possible answer `G` is selected with `member` via backtracking. One class representant is finally selected with the only one solution call to `member` in last line. Its identifier of this representant is the one returned.

5.4 Duplicates and Projection

Aforementioned handling of duplicate elimination is not enough to deal with correlated SQL queries. Compiling such a SQL query to Datalog may involve a distinct operator for a projection (a subset) of the arguments in a relation as, for instance:

```
        CREATE TABLE t(a int, b int);  CREATE TABLE s(a int, b int);
               CREATE VIEW v(a) AS SELECT a FROM t WHERE a IN
                 (SELECT DISTINCT a FROM s WHERE t.b<s.b);
```

A possible Datalog program for this view follows:

```
v(A)      :- t(A,B), distinct(v_1(A,B)).
v_1(A,B) :- s(A,C), B < C.
```

But this is not a correct equivalent formulation because `distinct/1` applies to different tuples from `v_1` instead of different values for `a`, and `b` must be passed to `v_1` to filter results. So, a new built-in `distinct(`*`Arguments`*`,`*`Relation`*`)` is needed, which computes different results for the list of relation arguments for the relation. The first Datalog rule would be rewritten as `v(A) :- t(A,B), distinct([A],v_1(A,B)).`

Solving this new built-in is via the call `memo(G,distinct(Vs,Ps),Id)`, where `G` is to be computed for returning only distinct tuples of variables `Vs`, which correspond to argument positions `Ps`. These positions are kept to account projection positions as any variable in `Vs` might become ground before solving this call. `no_subsumed_by_et/3` is modified accordingly by adding the next clause, which deals with duplicate elimination when duplicates are enabled.

```
no_subsumed_by_et(Q,distinct(_Vs,Ps),(G,_IdG)) :-
  \+ (functor(Q,F,A), functor(FQ,F,A),
       get_ith_arg_list(Ps,Q,QAs), get_ith_arg_list(Ps,FQ,QAs),
       et(FQ,_Id),get_ith_arg_list(Ps,G,GAs),my_subsumes(QAs,GAs)).
```

As well, predicate `et_lookup/3` is added with the following clause:

```
et_lookup(G,distinct(Vs,_SG),IdG) :-
  findall((G,IdG),et(G,IdG),GIdGs), term_variables(G,GVs),
  set_diff(GVs,Vs,EVs),
  build_ex_quantifier(EVs,my_member((G,IdG),GIdGs),QG),
  setof(Vs,IdG^QG,Vss), my_member(Vs,Vss), once(my_member((G,IdG),GIdGs)).
```

Although similar to the second clause of this predicate in previous section, this clause adds handling of unprojected variables by building existential quantifiers over them, so that `setof/3` returns distinct tuples for projected variables.

5.5 Duplicates in Aggregates

Aggregates are supported in DES in several flavors [SP11], both as functions and predicates, but for the sake of this paper, we restrict the presentation to a simplified aggregate predicates. An aggregate predicate returns its result in its last argument position, as in `sum(p(X),X,R)`, which binds `R` to the cumulative sum of values for `X`, provided by relation `p`. Duplicate elimination versions are also available, such as `sum_distinct/3`.

Solving rules involving aggregates via successive fixpoint iterations might lead to incorrect entries to be added to the extension table because in a given iteration it is not ensured that all the meaning of the aggregated relation is computed. A straightforward approach to solve them is analogous to negation: new negative arcs are added to the dependency graph in order to place such relation in a stratum lower than the predicate of the rule in which it occurs. So, the rule `s(R) :- sum(p(X),X,R)` implies an arc $\mathtt{s} \overset{\neg}{\leftarrow} \mathtt{p}$, which in turn also implies the constraint $strata(\mathtt{p}) < strata(\mathtt{s})$ for the stratification.

Then, when solving an aggregate in a given fixpoint iteration, all the tuples of its aggregated relation are known, so that duplicate elimination is simply performed by applying `setof` instead of `bagof` to groups from entries in the extension table. A simplified version implementing this is shown next:

```
compute_distinct_aggregate_pred(Aggr) :-
  Aggr =.. [F,R,V,O], R =.. [_P|Args], get_arg_position(V,Args,I),
  nf_setof(N, CR^Ids^(et(CR,Ids),\+ \+ (CR=R),
                      arg(I,CR,N),N\='$NULL'(_Id)), Ns),
  compute_aggregate(F,Ns,O).
```

Here, `nf_setof` is the non-failing version of `setof` (it returns an empty list instead of failure), and collects all values for the argument to which the aggregate is applied (`X` in the example above). As in relational databases, null arguments are omitted from this set.

6 Enhancing Performance

Several performance penalties can be found in the current implementation of the deductive database system DES. Next, we list them and propose some ways for overcoming:

- Prolog-based implementation. Instead of resorting to a compiler generating native code, we have used a Prolog system because it is much more adequate for rapid prototyping than low-level languages as, e.g., C++. Indeed, the higher abstraction level of Prolog w.r.t. third generation languages is a definitive advantage to quickly get results. However, one can find speed-ups in such languages which are orders of magnitude better. Even Java programs compiled to byte code, reveal better performance, as, e.g., the classic queens benchmark if compared to a CLP(FD) implementation.

- Interpreted vs. compiled Prolog code. We assert Datalog rules in a dynamic Prolog database. Most Prolog systems distinguish between asserted and compiled (consulted) programs. Whereas the former use the dynamic database, the latter compile to some sort of WAM code including first-argument indexing. This makes consulted programs perform better than asserted programs. So, following [Die87], we could implement query solving in the definition of Datalog predicates and queries. Then, predicates would be compiled instead of asserted and therefore more efficient.

- Indexing. Despite the last issue and its improvement, there remains dynamic data in the call and extension tables. On the one hand, such dynamic data suffers from the same issue as above, and, on the other hand, accessing such data structures with a sequential index is not enough for large amounts of data (assuming that we could automatically do it as the Prolog compiler does with compiled predicates). Better index structures should be taken into account, such as balanced trees or, better, *tries*, as implemented in the XSB system.

- Query solving. Identifying particular query solving cases and applying specifically-suited code can provide better performance. For instance, there is no need to find a fixpoint for extensional database predicates. It suffices with a single linear fetching to deduce the meaning, as it is common to do in fixpoint computations (e.g., abstract-interpretation-based static analysis). This can be extended to non-recursive predicates, where only one iteration should be needed. Also, computations which are known to be complete during fixpoint can be annotated as such, so that next calls to them can be directly resolved with the extension table.

7 Conclusions

This paper has shown how some widespread-used features in relational databases, namely nulls, outer joins and duplicates, can be included altogether into a tabled deductive database system. Since this system compiles SQL statements to Datalog programs, it was a need to embody such features coming from relational database systems into the concrete deductive inference engine DES. Because this system is not geared towards performance, this implementation should be seen as a proof of concept. Some hints have been provided for the porting to other deductive systems and future work may include to use other external efficient engines such as XSB.

Bibliography

[BD98] G. Brewka, J. Dix. Knowledge Representation with Logic Programming. In Dix et al. (eds.), *Proceedings of LPKR'97*. LNAI 1471, pp. 1–51. Springer-Verlag, 1998.

[CGS08] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. A Theoretical Framework for the Declarative Debugging of Datalog Programs. In *International Workshop on Semantics in Data and Knowledge Bases*. LNCS 4925, pp. 143–159. Springer, 2008.

[CGS10] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. Applying Constraint Logic Programming to SQL Test Case Generation. In *Proc. International Symposium on Functional and Logic Programming (FLOPS'10)*. LNCS 6009. 2010.

[CGS11] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. Algorithmic Debugging of SQL Views. In *Ershov Informatics Conference (PSI'11)*. LNCS. Springer, 2011. In Press.

[Dam96] C. Damásio. *Paraconsistent Extended Logic Programming with Constraints,*. PhD thesis, Dept. de Informâtica, Universidade Nova de Lisboa, 1996.

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

[Die87] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *IEEE Symp. on Logic Programming*. Pp. 264–272. 1987.

[GCH$^+$08] P. C. de Guzmán, M. Carro, M. V. Hermenegildo, C. Silva, R. Rocha. An improved continuation call-based implementation of tabling. PADL'08, pp. 197–213, 2008.

[LAC99] J. Y. Liu, L. Adams, W. Chen. Constructive negation under the well-founded semantics. *JLP* 38(3):295–330, 1999.

[LPF$^+$06] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Tran. on Computational Logic* 7(3):499–562, 2006.

[RSC05] R. Rocha, F. M. A. Silva, V. S. Costa. Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In Gabbrielli and Gupta (eds.), *ICLP*. LNCS 3668, pp. 250–264. Springer, 2005.

[RU93] R. Ramakrishnan, J. Ullman. A survey of research on Deductive Databases. *JLP* 23(2):125–149, 1993.

[SP11] F. Sáenz-Pérez. DES: A Deductive Database System. *Electronic Notes on Theoretical Computer Science* 271:63–78, March 2011.

[SP12a] F. Sáenz-Pérez. Datalog Educational System. May 2012. http://des.sourceforge.net/.

[SP12b] F. Sáenz-Pérez. Outer Joins in a Deductive Database System. *Electronic Notes in Theoretical Computer Science* 282:73 – 88, 2012.

[SD91] C. Shih, S. Dietrich. Extension Table Evaluation of Datalog Programs with Negation. In *Proc. of the IEEE International Phoenix Conference on Computers and Communications*. Volume AZ, pp. 792–798. Scottsdale, March 1991.

[SS06] Z. Somogyi, K. Sagonas. Tabling in mercury: Design and implementation. In *In Proceedings of Practical Aspects of Declarative Programming (PADL'06)*. Pp. 150–167. Springer-Verlag, 2006.

[SSW94a] K. Sagonas, T. Swift, D. S. Warren. XSB as an Efficient Deductive Database Engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*. Pp. 442–453. ACM Press, 1994.

[SSW94b] K. Sagonas, T. Swift, D. S. Warren. XSB as an efficient deductive database engine. In *SIGMOD'94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. Pp. 442–453. ACM, New York, NY, USA, 1994.

[SW10] T. Swift, D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *CoRR* abs/1012.5123, 2010. Submitted to TPLP.

[SWSJ09] T. Swift, D. Warren, K. Sagonas, J. Freire et al. The XSB System Version 3.2. Volume 2: Libraries, Interfaces and Packages. 2009.

[TG94] B. Traylor, M. Gelfond. Representing Null Values in Logic Programming. In *LFCS'94*. Pp. 341–352. 1994.

[TS86] H. Tamaki, T. Sato. OLDT Resolution with Tabulation. In *Third International Conference on Logic Programming*. Pp. 84–98. 1986.

[Ull88] J. D. Ullman. *Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1988.

[ZCF$^+$97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.

[ZS03] N.-F. Zhou, T. Sato. Efficient fixpoint computation in linear tabling. PPDP'03, pp. 275–283. ACM, New York, NY, USA, 2003.