Outer Joins in a Deductive Database System

Fernando Sáenz-Pérez [1] [2]

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

Abstract

Outer joins are extended relational algebra operations intended to deal with unknown information represented with null values. This work shows an approach to embed both null values and outer join operations in the deductive database system DES (Datalog Educational System), which uses Datalog as a query language. This system also supports SQL, where views and queries are compiled to Datalog programs. So, as SQL statements are ultimately solved by a Datalog engine, it became a need to integrate null-related operations into Datalog in order to support a wider set of SQL. Since DES implements a top-down-driven bottom-up stratified fixpoint computation based on tabling for solving Datalog queries, we show how to compute outer joins in such a context by means of source-to-source transformations applied to Datalog programs.

Keywords: Outer Joins, Deductive Databases, Datalog, SQL

1 Introduction

Deductive database systems include a form of the Datalog language, which has become the *de-facto* standard deductive database query language. There have been many versions of this language (pure Datalog, Datalog with negation, uninterpreted function symbols, disjunctive heads, constraints, ... [20]), and several deductive systems have emerged along time, mostly born from academic efforts. See, among others, DLV [17], XSB [26], bddbddb [16], LDL++ [2], DES [25], ConceptBase [15], and .QL [21].

This language has been extensively studied and is gaining a renowned interest thanks to their application to ontologies [13], semantic web [8], social

[1] 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)

[2] Email: fernan@sip.ucm.es

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

networks [22], policy languages [3], and even for optimization [14]. In addition, companies as LogicBlox (www.logicblox.com), Exeura (www.exeura.com), Semmle (www.semmle.com), and Lixto (www.lixto.com) embody Datalog-based deductive database technologies in the solutions they develop. The high-level expressivity of Datalog and its extensions has therefore been acknowledged as a powerful feature to deal with knowledge-based information. Compared to the widely-used relational database language SQL, Datalog adds two main advantages. First, its clean semantics allows to better reason about problem specifications. Its more neat formulations, notably when using recursive predicates, allow better understanding and program maintenance. Second, it provides more expressivity because the linear recursion limitation in SQL is not imposed.

However, in order to subsume SQL, a Datalog-based language has to include null values and its related outer join operations, among other features (aggregates, arithmetics, ...). In this work, we are therefore interested in obtaining the very same answer for a given Datalog query including outer joins as for its *equivalent* SQL counterpart, instead of embodying a true three-valued semantics for including undefinedness, as in other works [29]. This work was motivated by the use of Datalog Education System (DES), a deductive system including Datalog and SQL as query languages which is geared towards teaching. As in this system, SQL queries are compiled to Datalog programs, then providing support for outer joins became a need.

This paper shows how to represent null values and implement outer joins in this system, which enjoys tabling for solving queries, following a top-down-driven bottom-up stratified fixpoint computation. We describe its concrete implementation and explain how to adapt it for supporting those new constructors. In particular, we show how to compute outer joins by means of source-to-source transformations applied to Datalog programs. As our goal is to have the same answer for *equivalent* Datalog and SQL queries, we have mimicked the actual semantics of SQL null-related operators in the context of a tabled deductive system. The examples given throughout the paper, both Datalog and SQL, can be executed in the current version of DES [24] and their listings correspond to actual displays of the system.

To the best of our knowledge, outer joins in a Datalog context have been only dealt in the translation of SPARQL (a language for RDF in the context of Semantic Web) to Datalog rules [19]. That work uses a simplified version of ASP with so-called HEX-programs [12] as target instead of a tabling-based system we consider. Also, SPARQL is non-recursive and does not support nested-queries.

Organization of this paper is as follows: Section 2 introduces DES and its query languages: Datalog and SQL. Section 3 describes the tabling-based solving that DES implements which is used to compute outer joins. Section

4 introduces null values, their representation, how they are handled, and the outer join operations allowed in the system from a user viewpoint. Section 5 shows how it is possible to solve outer joins via program transformations and including native support for nulls. As well, points to drive the proposed program transformation technique to other systems are also given in Section 6. Finally, Section 7 draws some conclusions.

2 DES, Datalog, and SQL

2.1 DES

The Datalog Educational System (DES) is a free, open-source, multiplatform, portable, Prolog-based implementation of a deductive database system. DES 2.3 [24] is the current implementation, which enjoys Datalog and SQL query languages, full recursive evaluation with tabling, full-fledged arithmetic, stratified negation [30], ODBC connections and novel approaches to Datalog and SQL declarative debugging [5,7], test case generation for SQL views [6], null values support, (tabled) outer join and aggregate predicates and functions [24].

DES has been developed to be used via an interactive command shell. Nonetheless, more appealing environments are available. On the one hand, DES has been plugged to the multi-platform, Java-based IDE ACIDE [23]. It features syntax colouring, project management, interactive console with edition and history, configurable buttons for commands, and shortcuts, among others. On the other hand, an Emacs environment has been developed by Markus Triska as a contribution to this project.

The system is implemented on top of Prolog and it can be used a state-of-the-art Prolog interpreter (currently, Ciao, GNU Prolog, SWI-Prolog and SICStus Prolog) running on any OS supported by such Prolog interpreter (i.e., almost any HW/SW platform). Portable executables has been also provided for Windows, Linux, and Mac OS X. They are portable as they do not need installation and can be run from any directory they are stored. This amounts to a straightforward startup procedure: Simply copy a folder to the desired target and run the application.

2.2 Datalog

The Datalog version considered in DES is as follows:

- A Datalog *program* consists of a set of rules.
- A *rule* has the form `head :- body`, or simply `head`, ending with a dot.
- A *head* is a positive atom including no built-in predicate symbols.
- A *body* contains conjunctions (denoted by "`,`") as well as disjunctions (de-

noted by ";") of literals (with usual associativity and priority for these operators).

- A *literal* is either an atom, or a negated atom (`not(Atom)`), or a built-in (literals are also referred to as goals).
- An *atom* is an atomic formula [1] restricted to have variables or constants as arguments.
- A *variable* is a program symbol starting with either an uppercase letter or an underline.
- A *constant* is a program symbol either starting with a lowercase letter or being a sequence of chars delimited by single quotes.
- A *query* is a literal (some built-ins are exceptions, as will be shown in Section 4, and include other atoms as arguments). In addition, temporary views can also be submitted as queries, as will be introduced in next subsection.

Compound terms are not allowed but as arithmetic expressions, which can occur in certain built-ins (for writing arithmetic expressions and conditions).

The answer to a query is the set of facts matching the query which are deduced in the context of the program, from both the extensional and intensional database. A query with different variables for all its arguments gives the whole set of facts (meaning) defining the queried relation. If a query contains a constant in an argument position, it means that the query processing will select the facts from the meaning of the relation such that the argument position matches the constant (i.e., analogous to a select relational operation with an equality condition). If a given variable occurs more than once in a query, the corresponding predicate arguments are required to match.

DES implements Datalog with stratified negation as described in [30] with safety checks [30,31]. Stratified negation broadly means that negation is not involved in a recursive computation path, although it can use recursive rules. The system can compute a query Q in the context of a program that is restricted to the dependency graph (which shows the computation dependencies among predicates) built for Q so that a stratification can be found. This means that, even when a program could be actually non-stratifiable, a query involving a subset of the program might be safely computable, provided that a suitable stratification can be found for its dependency subgraph.

2.3 SQL

DES covers a reasonable set of the SQL language following the ISO standard SQL:1999 (further revisions of the standard cope with issues as XML, triggers, and cursors which are outside of the scope of DES), including recursive queries (limitations can be found in [24]). There is provision for the DDL (data definition language – `CREATE TABLE`, ...), DML (data manipulation language

– `INSERT INTO`, ...) and DQL (data query language – `SELECT`, ...) parts of the language [3].

SQL DQL statements are translated into and executed as Datalog programs (basics can be found in [30]), and relational metadata for DDL statements are kept. Submitting a DQL 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`($\overline{X}$), where $\overline{X}$ are N fresh variables. After its execution, this Datalog program is removed. On the contrary, if a DDL statement defining a view is submitted, its translated program and metadata do persist. A DML statement including either a `WHERE` condition for filtering or a `SELECT` data source is translated into a Datalog query and program, so that results obtained from executing this query are used to modify base relations, depending on the statement (`DELETE`, `UPDATE`, or `INSERT`).

3 Tabling-based Solving

The computational model of DES follows a top-down-driven bottom-up fixpoint computation with tabling. Tabling is implemented with the ideas found in [11,28], which deal with termination and performance issues of Prolog programs. In its current form, it can be seen as an extension of the work in [11] in the sense that, in addition, it deals with negation, undefined (although incomplete) information, nulls and aggregates, also providing a more efficient tabling mechanism (however, this system does not pretend to be competitive with current implementations but a system capable of showing the nice aspects of the more powerful form of logic we can find in Datalog systems w.r.t. relational database systems).

3.1 Tabling

DES uses an extension table which stores answers to goals previously computed, as well as their calls. For the ease of the introduction, we assume an answer table and a call table to store answers and calls, respectively. Answers may be positive or negative, that is, if a call to a positive goal `p` succeeds, then, the fact `p` is added as an answer to the answer table; if a negated goal `not(p)` succeeds, then the fact `not(p)` is added. 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.

[3] Note that we distinguish, as opposed to common use, DQL and DML. Usually, DQL, as we understand it, is rather included in DML.

The algorithm which implements this idea can be sketched as follows: First, test whether there is a previous call that subsumes the current call, where calls occur in the usual innermost left-to-right order of Prolog implementations. There are two possibilities: 1) there is such a previous call: then, use the result in the answer table, if any. 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 knowing that there is neither a call nor an answer to this call in the extension table. So, firstly store the current call and then, solve the goal with the program rules (recursively applying this algorithm). Once the goal has been solved (if succeeded), store 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). This is known as a memoization process, which is implemented with a Prolog predicate, and will also be referred to as a memo function.

Negative facts are produced when a negative goal is proved by means of negation as failure (closed world assumption (CWA) [30]). In this situation, a goal as `not(p)` which succeeds produces the fact `not(p)` which is added to the answer table, just the same way as proving a positive goal. However, both positive and negative facts cannot occur in a stratifiable program [30]. Before executing any query, the extension table is empty; after executing a query, at least the call is not empty. Also, the extension table is emptied after the execution of a temporary view. The extension table contains the calls made during the last fixpoint iteration (see next section for details); the calls are cleared before each iteration whereas the answers are kept.

3.2 Fixpoint Computation

The tabling mechanism is insufficient in itself for computing all possible answers to a query. The rationale behind this comes from the fact that the computed information is not complete when solving a given goal, because it can use incomplete information from the goals in its defining rules (these goals can be mutually recursive). Therefore, it is needed to ensure that all the possible information is deduced by finding a fixpoint of the memo function. First, the call table is emptied in order to allow the system to try to obtain new answers for a given call, preserving the previous computed answers. Then, the memo function is applied, possibly providing new answers. If the answer 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 we find a stable answer table (with no changes in the answer table). The answer table contains the meaning of the query (plus perhaps other meanings for the relations used in the computation of the given query).

The fixpoint is found in finite time because the memo function is monotonic

in the sense that new entries are only added each time it is called, while keeping the old ones. Repeatedly applying the memo function to the answer table delivers a finite answer table since the number of new facts that can be derived from a Datalog program without (built-in) infinite relations is finite (recall that user domains are finite since function symbols are not allowed and no compound terms are possible). On the one hand, the number of positive facts which can be inferred are finite because there is a finite number of ground facts which can be used in a given proof, and proofs have finite depth provided that tabling prevents recomputations of older nodes in the proof tree. On the other hand, the number of negative facts which can be inferred is also finite because they are proved using negation as failure. (Failures are always finite because they are proved trying to get a success.) Finally, there are facts that cannot be proved to be true or false because of recursion. These cases are detected by the tabling mechanism which prevent infinite recursion such as in `p :- p`.

For non-stratifiable programs (cf. next subsection), it is also possible to infer both a positive and a negative fact for a given call. Then, an undefined fact replaces the contradictory information. The implementation simply removes the contradictory facts and informs about the undefinedness. However, the current algorithm for determining undefinedness is incomplete as this feature is only kept for teaching purposes on rather small examples (for instance, XSB includes a complete implementation of the well-founded semantics which deals with undefined facts).

3.3 Dependency Graphs and Stratification

Each time a program is consulted or modified (i.e., via submitting a query or changing the database), a predicate dependency graph is built [31]. This graph shows the dependencies, through positive and negative atoms, 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 antecedent in a rule (i.e., each predicate in a rule body) to its consequent (i.e., rule head). Arcs are labeled as either negative, if the antecedent node occurs negated, or positive otherwise. This dependency graph is used for looking for a stratification for the program [31]. 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 $\mathtt{p} \overset{\neg}{\leftarrow} \mathtt{q}$, $strata(\mathtt{p}) < strata(\mathtt{q})$. Then, it follows that 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, query solving in DES is restricted to the predicates that can be reached from the query in the dependency graph. And it only resorts to compute by stratum when a negative dependency occurs in this subgraph. Nevertheless, each predicate that is actually needed is solved by means of the extension table mechanism described in the previous section. As a consequence, many computations are avoided w.r.t. a naïve bottom-up implementation.

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 [10]), null values to represent unknowns have been widely used. Including nulls in a Datalog system 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 [10]. Although a 3VL is assumed for RDBMS's[4] (Oracle, DB2, SQL Server, MySQL, ...), the fact is that the implemented logic does not account for the unknown logic value. Instead, it is incorrectly represented by the null value [10].

Moreover, the following example shows that the expected outcome from a "real world" viewpoint is not got. Let's consider the SQL query `SELECT COUNT(*) FROM t WHERE a = a`, assuming a table `t` with a single column `a`. If the instance of `t` contains a tuple with a null value, the answer to this query is 0, but it should be otherwise 1, as `a` is trivially equal to itself for any given tuple.

To illustrate the incorrect implementation of 3VL, we can follow the same example assuming that `t` contains *two* tuples with a null value, and the query `SELECT * FROM t t1, t t2 WHERE t1.a = t2.a`. Which would we expect from comparing the first to the second tuple in `t` in this query? (Let's denote the two null values in `t` as $\texttt{null}_1$ and $\texttt{null}_2$, respectively.) Following 3VL in current RDBMS's, as $\texttt{null}_1 = \texttt{null}_2$ admittedly delivers `unknown`, then it is

[4] Although prepended by that R, recall that relational model is about a two-valued logic so that such systems are nonrelational indeed.

neither `true` nor `false` anymore. So, what bag is the tuple <$\texttt{null}_1$,$\texttt{null}_2$> thrown at? Incorrectly, at the `false` bag following a CWA approach. One can argue that such tuple was thrown at an `unknown` bag hidden from the user. But then, the answer is incomplete: the user should be warned that there are tuples that cannot be classified as either belonging to the meaning of the query or not.

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 disquality) hold. However, notice that a semantic flaw emerges in `not(null = null)` (negation), which succeeds![5] This does not follow SQL: The conditions `a<>b` and `not(a=b)`, where `a` and `b` are columns, yield the same logical outcome when considering nulls. So, comparing null values are discouraged in DES.

Nonetheless, to test whether a value is null, two built-in predicates are provided:

- `is_null`/1: Test whether its argument is a null value. Analogous operation to the SQL clause `IS NULL`.
- `is_not_null`/1: Test whether its argument is *not* a null value. Analogous operation to the SQL clause `IS NOT NULL`.

Nulls are internally represented with the term `'$NULL'(`*`IdNumber`*`)`, where *`IdNumber`* is a unique integer which does not occur in any other null. This representation is similar to that also suggested in other systems [27], but, as a difference, DES considers null as a first class citizen and its internal representation is hidden from the user. Therefore, asserting or consulting a rule as `p(null)` is directly allowed. Back to the first SQL statement in this section above, if executed in DES, it delivers the "expected" answer, since the comparison `a = a` is instanced as `'$NULL'(`*`Id`*`) = '$NULL'(`*`Id`*`)` which has given a concrete integer for *`Id`*, and therefore syntactic unification succeeds.

4.2 Outer Join Built-ins

Three outer join operations are provided, following relational database query languages (SQL, extended relational algebra): left, right and full outer joins. An outer join computes the cross-product of two relations that satisfy a third relation, extended with some special tuples including nulls as explained next. In an outer join, tuples in one of the first two relations which have no counter-

[5] The negation of the equality *should* behave as disequality.

part in the other relation (w.r.t. the third relation) are included in the result (the values corresponding to the relation with no corresponding tuple are set to `null`). If this is true for relation A in the cross-product $A \times B$ then it is a left outer join; if it is true for B then it is a right outer join; if it is true for both then it is a full outer join. In DES, the left (resp. right, and full) outer join corresponds to the construction `lj(A,B,C)` (resp. `rj(A,B,C)`, and `fj(A,B,C)`), with `A`, `B`, and `C` relations. In addition, both `A` and `B` can take the form of such constructions in order to allow more neat, nested applications of outer joins (cf. Subsection 5).

A *join* condition must not be confused with a *where* condition. Let's consider the query `lj(a(X),b(Y),X=Y)`, which is not equivalent to `lj(a(X),b(X),true)` [6]. Assuming that `x` and `y` are columns of tables `a` and `b`, resp., these queries could be respectively written in SQL as follows:

```
SELECT * FROM a LEFT JOIN b ON x=y;
SELECT * FROM a LEFT JOIN b WHERE x=y;
```

Outer join relations can be nested as well, as in:

```
lj(a(X),rj(b(Y),c(U,V),Y=U),X=Y)
```

which is equivalent to the following SQL statement:

```
SELECT * FROM a LEFT JOIN
(b RIGHT JOIN c ON y=u) ON x=y;
```

Note that compound conditions must be enclosed between parentheses, as in:

```
lj(a(X),c(U,V),(X>U;X>V))
```

5 Source-to-Source Transformations

This section explains outer join-related, source-to-source transformations, which are conducted during the preprocessing phase in DES. Other transformations include simplifications, unfolding, outer joins and aggregate predicates (which are out of the scope of this paper). Transformations are guided by several needs: enhancing performance, ensuring termination and resorting to Datalog tabled computation for solving the outer join primitives (rather than writing (Prolog-)specific code for that).

As stated in Section 4, a left join operation `lj(A,B,C)` requires to build tuples for each tuple in `A` which does not match with any other of `B` w.r.t. `C`. In a given cycle of the fixpoint computation, a tuple t_A might not find a matching tuple in B, but a further cycle may develop new tuples for `B` that do. In order to prevent speculative computations and removing entries from the extension table, we restrict to have completed the computation of the involved

[6] Notice that the variable `X` is shared by relations `a` and `b`.

relations. This can be achieved by taking advantage of the stratification idea: relations in outer joins are collected into strata as if they were negative atoms in order to have their answer sets completely defined before an outer join over those relations.

To this end, a left join `lj(A,B,C)` is transformed into an equivalent set of rules (w.r.t. semantics) for reusing the dependency graph and stratification approach. 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 `$p`$\overset{\neg}{\leftarrow}$`p` is added to the dependency graph. Its arity is $|vars(\texttt{A}) \cup vars(\texttt{B}) \cup vars(\texttt{C})|$.

Next, the 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 `A` joined with those of `B` which meet `C`, and the facts in `A` joined with nulls which *do not* meet `C`. So, a source-to-source transformation in this case can be seen with the following example. Let's consider the rule `v(X,Y) :- lj(s(X,U),t(V,Y),U>V)`, which is transformed into:

```
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.
```

Note that the predicate `$p0` is used to compute both sources of facts, whether provided by the positive case (a *straight* call to `$p1` from the second rule of `$p0`) or the negative one (a *negated* call to `$p1` in the first rule of `$p0`). This negative call oughts `$p1` to be in a lower strata than `$p0`. Therefore, before computing `$p0`, the meaning of `$p1` is completely available. Also note that the first rule builds the null values for the arguments of the right relation `B` for which no tuples are found meeting `C`. Observe terms `'$NULL'(C)` and `'$NULL'(D)`, containing free (unsafe) variables. They are the internal representations for null values, which a normal user would see simply as `null`. Before adding a tuple to the extension table, each null value is assigned with a unique integer should it is not assigned already. This allows a single rule to become source of different null values for each tuple it may deliver as an element of its meaning. Finally, the predicate `$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 answer table of `'$p1'` if the condition `B > C` holds. Indeed, this is quite similar to the

RDBMS implementations of join operations (modulo indexing).

Whilst transformation of the right join is a reflection of the left join, the full join is a bit different, where the source of tuples comes from three sources: the same two as for the left join, plus the facts in `B` joined with those of `A` which do not meet `C`.

Notice that the transformed program above include floundering [4]: there exists a call `not('$p1'(A,B,E,F))` in the first rule for `'$p0'`, 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 answer table. Such negated entries should not be reused by any other (negated) calls in the program since they are not ground.

Although a safety check is performed whenever a rule is going to be asserted, this is previous to the program transformation so that no errors are reported to the user in this automatic translation. Other works treat the floundering problem in a general use of negation (see, e.g., constructive negation [18] and also tabled query evaluation [9]), where non-ground negated calls are possibly involved in recursive calls, which we do not consider in our setting.

As well, nested outer join calls are allowed, as in the next view, where relations `s` and `t` are now assumed to take only one argument for the sake of brevity: `v(X,Y,U) :- lj(lj(s(X),t(Y),X < Y),s(U),Y = U).`

6 Transfers to Other Systems

Other deductive systems, such as DLV [17], 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 [30], above programs can be transformed into non-floundering programs, where all calls to negated goals are ensured to be ground. Let's consider for instance how to transform the running example: Non-relevant variables should be dropped in the translation (here, `A` in predicates `$p1` and `$p2`). Also, unfolding can be applied and get:

```
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 running 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 an adaptation of our proposal, 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 [26] 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 [27]. 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, which is now possible as we will see. Next, we show the complete program (but the facts for `s` and `t`) for the running example involving only one outer join:

```
:- 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.

7 Conclusions

We have presented a novel approach to embed outer joins as Datalog built-ins in an existing deductive database system, which is used in many universities (des.sourceforge.net/des_facts). Its statistics (des.sourceforge.net/statistics) show a notable increasing number of downloads, up to more than 1,500 downloads a month, and more than 34,000 downloads since 2004. New releases are expected each two or three months, therefore revealing it as a live project.

Those outer join built-ins are not only used in the Datalog query language but are also used in the evaluation of SQL DQL statements, as they are compiled to Datalog programs and executed by the deductive engine. The program transformation technique has been found to be useful in both ensuring terminating computations and also computing outer join results as these built-ins are expressed as Datalog programs, rather than writing specialized Prolog code to compute them.

Though this work was originally focused to provide outer joins in an educational deductive system, we have also shown that this technique can also be applied to other systems such as XSB which features non-ground semantics, and highlighted the need for native support of nulls in DLV as it is not a general-purpose programming system as XSB is. Compared to DES, an explicit management of nulls is needed in the former system, and, in the latter, it is also needed to use the (slightly costly) alternative translation to non-floundering programs.

References

[1] Krzysztof R. Apt. Introduction to logic programming. Technical report, University of Texas at Austin, Austin, TX, USA, 1988.

[2] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The Deductive Database System LDL++. *Theory and Practice of Logic Programming*, 3(1):61–94, 2003.

[3] Moritz Becker, Cedric Fournet, and Andrew Gordon. Design and Semantics of a Decentralized Authorization Language. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 3–15, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Gerhard Brewka and Jrgen Dix. Knowledge Representation with Logic Programming. In J. Dix, L. Moniz Pereira, and T.C. Przymusinski, editors, *Proceedings of LPKR'97*, volume 1471 of *LNAI*, pages 1–51. Springer-Verlag, 1998.

[5] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. A Theoretical Framework for the Declarative Debugging of Datalog Programs. In *International Workshop on Semantics in Data and Knowledge Bases (SDKB)*, volume 4925 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2008.

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

[7] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Algorithmic Debugging of SQL Views. In *Ershov Informatics Conference (PSI'11)*, Lecture Notes in Computer Science. Springer, 2011. In Press.

[8] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog±: a unified approach to ontologies and integrity constraints. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 14–30, New York, NY, USA, 2009. ACM.

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

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

[11] Suzanne W. Dietrich. Extension tables: Memo relations in logic programming. In *IEEE Symp. on Logic Programming*, pages 264–272, 1987.

[12] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05*, pages 90–96. Professional Book, 2005.

[13] Richard Fikes, Patrick J. Hayes, and Ian Horrocks. OWL-QL - a language for deductive query answering on the Semantic Web. *J. Web Sem.*, 2(1):19–29, 2004.

[14] Sergio Greco, Irina Trubitsyna, and Ester Zumpano. NP Datalog: A Logic Language for NP Search and Optimization Queries. *Database Engineering and Applications Symposium, International*, 0:344–353, 2005.

[15] Matthias Jarke, Manfred A. Jeusfeld, and Christoph Quix (Eds.). ConceptBase V7.1 User Manual. Technical report, RWTH Aachen, April 2008.

[16] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In Chen Li, editor, *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–12. ACM, 2005.

[17] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Tran. on Computational Logic*, 7(3):499–562, 2006.

[18] Julie Yuchih Liu, Leroy Adams, and Weidong Chen. Constructive negation under the well-founded semantics. *The Journal of Logic Programming*, 38(3):295–330, 1999.

[19] Axel Polleres. From SPARQL to rules (and back). In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 787–796, New York, NY, USA, 2007. ACM.

[20] R. Ramakrishnan and J.D. Ullman. A survey of research on Deductive Databases. *The Journal of Logic Programming*, 23(2):125–149, 1993.

[21] G. Ramalingam and Eelco Visser, editors. *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation*. ACM, 2007.

[22] Royi Ronen and Oded Shmueli. Evaluating very large Datalog queries on social networks. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 577–587, New York, NY, USA, 2009. ACM.

[23] Fernando Sáenz-Pérez. ACIDE: An Integrated Development Environment Configurable for LaTeX. *The PracTeX Journal*, 2007(3), August 2007. Available at `acide.sourceforge.net`.

[24] Fernando Sáenz-Pérez. Datalog Educational System V2.3, May 2011. `des.sourceforge.net/`.

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

[26] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In *SIGMOD'94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 442–453, New York, NY, USA, 1994. ACM.

[27] T. Swift, D.S. Warren, K. Sagonas, and J. Freire et al. The XSB System Version 3.2. Volume 2: Libraries, Interfaces and Packages, 2009. `http://xsb.sourceforge.net/`.

[28] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, 1986.

[29] Bonnie Traylor and Michael Gelfond. Representing Null Values in Logic Programming. In *LFCS '94: Proceedings of the Third International Symposium on Logical Foundations of Computer Science*, pages 341–352, London, UK, 1994. Springer-Verlag.

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

[31] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roberto Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.