Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 271 (2011) 63–78
www.elsevier.com/locate/entcs

DES: A Deductive Database System

Fernando Sáenz-Pérez[1]

Declarative Programming Group (GPD)
Dept. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain

Abstract

This work describes a novel implementation of a deductive database system which fills some gaps other systems do not. In fact, this system was born to this end and since its inception, many new features have been added (null values, outer joins, aggregates, ...). In particular, it embodies both Datalog and SQL query languages, where the same database can be queried. It enjoys an actual interactive environment for any platform (Windows, Linux, Macintosh, ...) and it has been plugged to a Java GUI IDE for easing user interaction (syntax highlighting, projects, ...). The system is distributed under GPL license, hosted by sourceforge, and heavily used all around the world.

Keywords: Deductive Databases, Datalog, SQL, Aggregates.

1 Introduction

The intersection of relational databases, logic, and artificial intelligence was the melting pot of deductive databases. A deductive database system includes procedures for inferring information from the so-called intensional database (deductive rules) in addition to the so-called extensional database (deductive rules without body, i.e., *facts* following the logic programming nomenclature). Deductive database languages are related to the Prolog language, and Datalog has become the *de-facto* standard deductive database query language. Datalog allows to write queries as normal logic programs (without function symbols), and they are ensured to be terminating upon some conditions (e.g., avoiding infinite relations as arithmetical predicates).

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

doi:10.1016/j.entcs.2011.02.011

This language has been extensively studied and is gaining a renowned interest thanks to their application to ontologies [10], semantic web [6], social networks [16], policy languages [2], and even for optimization [11]. In addition, companies as LogicBlox, Exeura, Semmle, and Lixto 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 compact 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. In fact, multiple recursive calls can be found in a deductive rule body.

Several deductive systems have emerged along time, mostly born from academic efforts. See, for instance DLV [14], XSB [19], bddbddb [13], LDL++ [1], ConceptBase [12], and .QL [15]. Translating these outcomes to experiment with and to widen the dissemination of state-of-the-art features of such deductive systems is hard since no one meets the following desired properties: multi-platform, interactiveness, multi-language support, freeness, and open-sourcing, among others.

In this paper, we list the main features of DES (Datalog Educational System) [18], highlighting some of the ones that make this tool different from any (existing, available) other.

Organization of this paper is as follows. Section 2 summarizes the main features of the current version of the system. Section 3 describes the Datalog and SQL query languages as they can be used from DES. Section 4 explains our proposal to the management of null values and outer join operators in Datalog, which in turn are used in the compilation of SQL statements to Datalog programs. Also, a novel approach to aggregates is described in Section 4, including both aggregate functions and predicates. Finally, Section 6 draws some conclusions.

2 Main Features

This section lists a brief summary of the main features DES enjoys:

Full Recursion

Datalog rules can be recursive, mutually recursive, include negation in bodies, and contain as many recursive calls as needed, as opposed to recursive SQL.

Stratified Negation

DES ensures that negative information can be gathered from a program with negated goals provided that a restricted form of negation is used: stratified negation [23]. This broadly means that negation is not involved in a recursive computation path, although it can use recursive rules. The system can correctly 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. The user can ask the system for displaying the predicate dependency graph as well as for the stratification via commands.

Built-ins

There are available some usual comparison operators (`=`, `\=`, `>`, ...). 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 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 are explained in Sections 4 and 5.

Full-Fledged Arithmetics

In contrast to other deductive systems (as, notably, DLV), arithmetical expressions can be as complex as needed, using almost the complete set of functions and operators in state-of-the-art Prolog systems, following the standard ISO Prolog.

Safety and Computability

Built-in predicates are appealing, but they come at a cost: The domain of their arguments is infinite, in contrast to the finite domain of each argument of any user-defined predicate. Since it is neither reasonable nor possible to (extensionally) give an infinite answer, when a subgoal involving a built-in is going to be computed, its arguments need to be range restricted, i.e., the arguments have to take values provided by other subgoals. DES provides a preprocessor for source-to-source translations that allows deciding whether a rule is safe (an extension of conditions in [23,25] for safe rules) and, if so, to translate it by reordering its goals in order to make it computable.

Temporary Views

Temporary views allow to write compound queries on the fly (as, e.g., conjunctions and disjunctions). A temporary view is a rule which is added to

the database, and its head is submitted as a query and executed. Afterwards, the rule is removed. For instance, given the relations `a`/1 and `b`/1 defined by facts, the view `d(X) :- a(X), not(b(X))` computes the set difference between the instance relations (sets) `a` and `b`. Note that the view is evaluated in the context of the program; so, if there are more rules already defined with the same name and arity of the rule's head, the evaluation of the view will return its answer set considering the program already loaded. For instance, the view `a(X) :- b(X)` computes the union of `a` and `b`.

Automatic Temporary Views

Automatic temporary views, *autoviews* for short, are temporary views which do not need a head. When submitting a conjunctive query, a new temporary relation, named `answer`, is built with as many arguments as relevant variables occur in the conjunctive query. `answer` is a reserved word and cannot be used for defining other relations. The conjunctive query `a(X), b(Y)` is an example of an autoview, which computes the Cartesian product of `a` and `b`.

Two Query Languages. One Deductive Database

Both Datalog and SQL languages are provided and query the same database. Moreover, Datalog programs can seamlessly refer to objects created in the SQL side, as tables and views [2] . Whereas the so-called extensional (deductive) database (EDB) is composed of Datalog facts and tuples in tables, the so-called intensional database (IDB) is composed of Datalog rules and relational views. The system includes a parser and preprocessor for Datalog, and a parser and a compiler from SQL to Datalog. SQL queries are processed with the deductive engine.

Datalog Declarative Debugger

In [4], an approach to debug Datalog programs anchored to the semantic level instead of the computation level is proposed. This approach has been implemented in DES as a novel way of applying declarative debugging, also called algorithmic debugging, to Datalog programs. It is possible to debug queries and diagnose missing answers (an expected tuple is not computed) as well as wrong answers (a given computed tuple should not be computed). The system uses a question-answering procedure which the user starts when

[2] Note that SQL views cannot refer to Datalog relations because SQL tables and views have attached relational metadata regarding column names and types, whereas Datalog rules do not.

he detects an unexpected answer for some query. Then, if possible, it points to the program fragment responsible of the incorrectness.

Test Case Generator

DES implements a novel test case generator for SQL views following [5]. Test case generation provides tuples for the involved input tables that can be matched to the intended interpretation of a view and therefore be used to catch posible design errors in the view. Both positive (PTC) and negative (NTC) test cases are generated. Executing a view for a PTC should return, at least, one tuple. This tuple can be used by the user to catch errors in the view, if any. This way, if the user detects that this tuple should not be part of the answer, it is definitely a witness of the error in the design of the view. On the contrary, the execution of the view for a negative test case should return at least one tuple which should not be in the result set of the query. Again, if no such a tuple can be found, this tuple is a witness of the error in the design.

Commands

The system console accepts several commands which are isolated from the database signature, i.e., name clash is avoided even when a relation takes the same name than a command. This is possible because submitting a command implies to precede it with the symbol "`/`". Commands are catalogued as: 1) Rule database commands, for inserting, deleting and listing both programs and single rules. 2) Operating system commands, for dealing with the operating system file system and external commands. 3) Extension table commands, for information about the memoization result. 4) Log commands, for logging system output to files. 5) Informative commands, for showing the predicate dependency graph, stratification, system status, help and others. 6) Miscellanea commands, as for quitting the session and invoking Prolog.

Batch Processing

The command `process` *`filename`* allows to process the file *`filename`* as a batch of user inputs. In addition, if the file `des.ini` is located at the installation directory, its contents are interpreted as input prompts and executed before giving control to the user. Therefore, automation is possible as, for instance, to set the DES application as a component of more complex systems of as a delegate for tasks sent from other systems. In this case, inter-process communication is via files. Batch files can contain remarks because prompt input lines starting with the symbol `%` are interpreted as such, which imply no computations. In addition, the command `/log` allows to write the system output to a file, which can be used by another application.

System Status

The way the system behaves can be modified by setting system flags. Several configurations are allowed: 1) Simplification mode, where automatic simplification for rules is allowed in order to enhance performance. 2) Program transformations, for trying to find safe formulations (cf. "Safety and Computability" above). 3) Development mode, for detailed listings that show compiled rules and internal representations. 4) Datalog and SQL pretty-print listings. 5) Verbose output, which lists informative reports about the execution. 6) Selection of algorithms to compute negation. 7) Elapsed time displays, whether basic or detailed.

Termination

Evaluation of queries is ensured to be terminating as long as no infinite predicates/operators are considered (data are constants so that terms with unlimited depth are not allowed). Currently, only the infix operator `is` represents an infinite relation and can deliver unlimited pairs. For instance, let's consider the rules `p(0).` and `p(X) :- p(Y), Y is X+1.` Then, the query `p(X)` is not terminating since its meaning is infinite (`{p(0), p(1), ...}`). However, terminating programs involving this operator are also possible by explicitly limiting its domain (cf. Section 3).

Implementation

The system has been implemented following ISO Prolog, its binaries uses an efficient Prolog engine (SICStus Prolog) and moreover implements memoization techniques [21,9] for upgraded efficiency. The computation is guided by the query, instead of following a bottom-up approach. However, focus was not set on performance, but on rapid prototyping of useful features, so that this system cannot be seen as a practical deductive database for large amounts of data since it is developed from an in-memory database point of view. In addition, neither indexing is provided nor concurrent accesses are allowed.

Free and Open-Source

DES is free, open-source and distributed under GPL license.

Impact

Many universities and researchers have used it (`http://des.sourceforge.net/des_facts`) and its downloading statistics (`http://des.sourceforge.net/statistics`) reveals it as a live project (a new release is expected every two or three months). Statistical numbers

show a notable increasing number of downloads, amounting to more than 1,500 downloads a month (vs. 300 for XSB) during last months, more than 27,000 downloads since 2004, and more than 32,000 entries in Google. Figure 1 shows the number of downloads a month since its first release (scale in thousands displayed on the left axis and numbers as blue bars) and bandwidth (scale in GB displayed on the right axis and numbers as yellow surface). Also, as a matter of impact, if the word "Datalog" is queried in any web search engine, DES occurs at the very first positions.

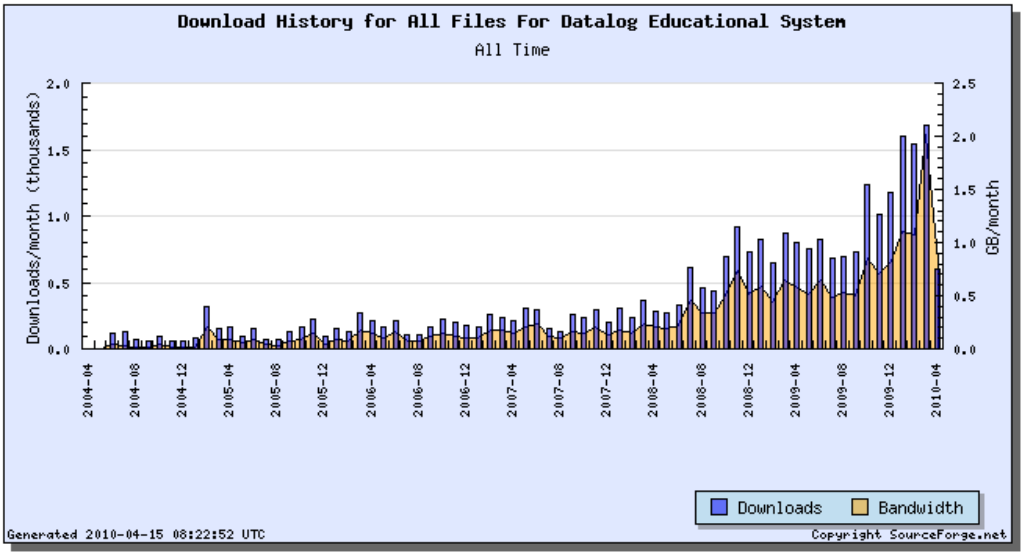

Fig. 1. Statistics since 2004

Interactive Shell

DES has been developed to be used via an interactive command shell (see "Portable System" below what applications are provided). Other systems, as DLV, do not provide an interactive shell, which we find quite useful for learning and quickly experiment with the system.

IDEs

Nonetheless, more appealing environments are available. On the one hand, DES has been plugged to the multi-platform, Java-based IDE ACIDE [17]. It features syntax colouring, project management, interactive console with edition and history, configurable buttons for DES commands, shortcuts, and much more (see Figure 2). On the other hand, an Emacs environment has been distributed by Markus Triska as a contribution to this project. It includes

DES in one buffer and allows to edit and run Datalog programs with keyboard shortcuts.

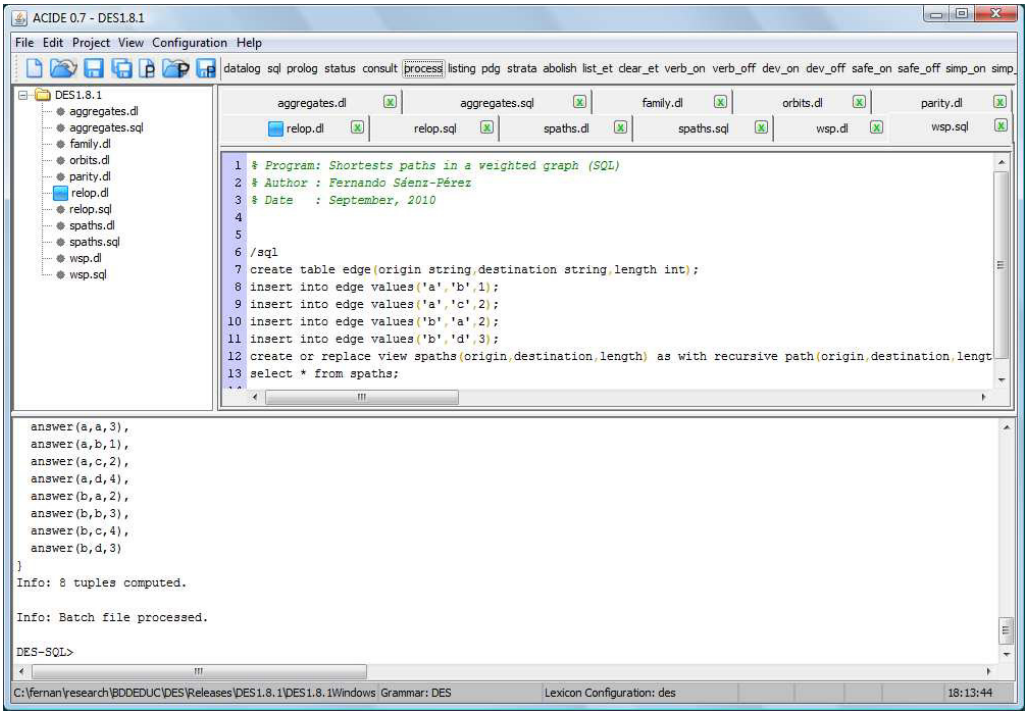

Fig. 2. ACIDE Integrated Development Environment configured for DES

Multi-platform

Since DES is ported to several Prolog systems (including Ciao Prolog [3], GNU Prolog [8], SICStus Prolog [20] and SWI-Prolog [24]), it can be used from any of these environments running on any platform they support (e.g., Windows, Linux glibc 2.x, Mac OS X 10.x, Solaris 10, AIX 5.1L, ...).

Portable System

Portable applications 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. DES has been compiled to two portable applications for Windows and Linux OS command shells. In addition, a portable windows application is also provided for the former OS (featuring menus with configurations mainly targeted to the underlying Prolog engine and OS).

3 Datalog and SQL

This section deals with the supported query languages in DES. Nonetheless, Prolog goals can also be submitted to the deductive database both query languages do share.

3.1 Datalog

- A DES *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 a conjunctions (denoted by "`,`") and disjunctions (denoted by "`;`") of literals (with usual associativity and priority for these operators).
- A *literal* is either an atom, or a negated atom or a built-in.
- A *query* is a literal and its arguments can be variables or constants (some built-ins are exceptions, as will be shown in Section 4, and include other atoms as arguments). In addition, recall that temporary views can also be submitted as queries, as introduced in Section 2.

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

Datalog programs are typically consulted via the system command `/consult` *`filename`*, and queries are typed at the DES system prompt. 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 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).

3.2 SQL

DES covers a wide set of the SQL language following the ISO standard, including recursive queries. There is provision for the DDL (data definition language), DML (data manipulation language) and DQL (data query language) parts of the language. Database updates are allowed in DML via `INSERT`, `DELETE`, and `UPDATE` statements. Also, integrity constraints as primary key and foreign key can be specified in DDL `CREATE TABLE` statements and also monitored by the system. A type system has been implemented to check

and infer types of views and queries. DQL part includes `SELECT` and `WITH` statements.

As well, SQL statements can be submitted at the system prompt. However, in contrast to Datalog programs, they can not be consulted but batch processed via the system command `/process` *`filename`*.

SQL DQL statements are translated into and executed as Datalog programs, and relational metadata for DDL statements are kept and can be consulted with the command `/dbschema` *`object`*, where the optional *`object`* can be either a table or a view. Submitting a DQL query amounts to parse it, compile to a Datalog program which includes the relation `answer`$/N$ with as many arguments as expected from the statement, assert this program and submit the Datalog query `answer`$(\overline{X})$, where $\overline{X}$ are N fresh variables. After its execution, this program is removed. On the contrary, if a DDL statement defining a view is submitted, its translated program and metadata do persist.

3.3 Datalog vs. SQL

Datalog subsumes SQL since the former is based on first order logic, whereas the latter is based on an extended relational algebra. In addition, Datalog is more readable and concise (cf. also Subsection 5.2). Let's consider the following problem: Given a graph defined by the relation `edge(Origin,Destination,Length)`, find the minimum path between all possible pairs. A possible recursive SQL formulation follows:

```
% View:
CREATE OR REPLACE VIEW
shortest_paths(Origin,Destination,Length)
AS WITH RECURSIVE
path(Origin,Destination,Length) AS
(SELECT edge.*,1 FROM edge)
UNION
(SELECT path.Origin,edge.Destination,
        path.Length+1
 FROM path,edge
 WHERE path.Destination=edge.Origin
 AND path.Length < (SELECT COUNT(*)
                    FROM Edge) )
SELECT Origin,Destination,MIN(Length)
FROM path
GROUP BY Origin,Destination;
% Query:
```

```
SELECT * FROM shortest_paths;
```

The same problem can be formulated in Datalog as:

```
% Program:
  path(X,Y,1) :-
    edge(X,Y).
  path(X,Y,L) :-
    path(X,Z,L0),
    edge(Z,Y),
    count(edge(A,B),Max),
    L0<Max,
    L is L0+1.
% Query:
  shortest_paths(X,Y,L) :-
    min(path(X,Y,Z),Z,L).
```

4 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, for instance, the discussion in [7]), null values to represent unknowns have been widely used. Including nulls in a Datalog system implies to also provide built-ins to handle them, as the outer join operations. DES includes the common outer join operations in relational databases, providing the very same semantics for operators ranging over nulls.

4.1 Null Values

The null value is included in each program signature for denoting unknowns, in a similar way it is an inherent part of current relational database systems. Comparing null values in Datalog opens a new scenario: Two null values are not (known to be) equal, and are (not known to be) distinct. So, neither `null = null` nor `null \= null` hold. However, a semantic flaw emerges in `not(null = null)`, which succeeds![3] The very same situation occurs in SQL: The conditions `A<>B` and `not(A=B)`, where `A` and `B` are columns, do not yield the same logical outcome when considering nulls. So, the user has to be conscious of this behavior. Therefore, instead of using comparison operators over variables that may take null values, two built-in predicates are provided:

- `is_null`/1: Test whether its single argument is a null value.

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

- `is_not_null`/1: Test whether its single argument is *not* a null value.

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 counterpart 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 then 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.

A *join* condition has not to be missed 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)` [4]. 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 Aggregates

Aggregates refer to functions and predicates that compute values with respect to a collection of values instead of a single value. We provide five usual aggregates: `sum` (cumulative sum), `count` (element count), `avg` (average), `min` (minimum element), and `max` (maximum element). In addition, the less usual

[4] Notice that the variable `X` is shared for relations `a` and `b`.

`times` (cumulative product) is also provided. They behave close to most SQL implementations, i.e., ignoring nulls.

5.1 Aggregate Functions

An aggregate function can occur in expressions and returns a value, as in `R=1+sum(X)`, where `sum` is expected to compute the cumulative sum of possible values for `X`, and `X` has to be bound in the context of a `group_by` predicate (cf. next section), wherein the expression also occurs.

5.2 Predicate `group_by`

This predicate encloses a query for which a given list of variables builds answer sets (groups) for all possible values of these variables. If we consider the relation `employee(`*`Name, Department, Salary`*`)`, the number of employees for each department can be counted with the query `group_by(employee(N,D,S),[D],R=count)`. If employees are not yet assigned to a department (i.e., a null value in *`Department`*), then this query behaves as a SQL user would expect: excluding those employees from the count outcome. If we rather want to count active employees (those with assigned salaries), we can use the query `group_by(employee(N,D,S),[D],R=count(S))`.

Conditions including aggregates on groups (cf. `HAVING` conditions in SQL) can be stated as well. For instance, for counting the active employees of departments with more than one employee, one can query:

```
group_by(employee(N,D,S),[D],count(S)>1)
```

Conditions including no aggregates on tuples (cf. `WHERE` conditions in SQL) of the input relation (cf. SQL `FROM` clause) can also be used. For instance, the following query computes the number of employees by department whose salary is greater than 1,000: `group_by((employee(N,D,S), S>1000), [D], R=count(S))`. Note that the following query is not equivalent to the last one, since variables in the input relation are not expected to be bound after a grouping computation, and it raises a run-time exception upon execution: `group_by(employee(N,D,S), [D], R=count(S)), S>1000`.

The predicate `group_by` admits a more compact representation than its SQL counterpart. Let's consider the following Datalog view:

```
q(X,C) :-
 group_by(p(X,Y),[X],(C=count;C=sum(Y)))
```

which is equivalent to:

```
CREATE VIEW q(X,C) AS
  (SELECT X,COUNT(Y) AS C
   FROM p GROUP BY X)
UNION
  (SELECT X,SUM(Y) AS C
   FROM p GROUP BY X);
```

5.3 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 the input relation `p`. These aggregate predicates simply allow another way of expressing aggregates, in addition to the way explained just above. For instance, the following query is allowed: `count(employee(N,D,S),S,T)`.

A `group_by` operation is simply specified by including the grouping variable(s) in the head of a clause, as in the following view, which computes the number of active employees by department: `c(D,C) :- count(employee(N,D,S),S,C)`. *Having* conditions are also allowed, including them as another goal of the first argument of the aggregate predicate as, for instance, in the following view, which computes the number of employees that earn more than the average: `count((employee(N,D,S),avg(employee(N1, D1,S1),S1,A),S>A),C)`. Note that this query uses different variables in the same argument positions for the two occurrences of the relation `employee`. Compare this to the following query, which computes the number of employees so that each one of them earns more than the average salary of his corresponding department. Here, the same variable name D has been used to refer to the department for which the counting and average are computed: `count((employee(N,D,S),avg(employee(N1, D,S1),S1,A),S>A),C)`.

6 Conclusions

This paper has listed the main features of the deductive database educational system DES and described some of the most relevant ones that distinguish it as a unique system. Following such features, intended users who can benefit from this system include students, teachers, practitioners and researchers, since it can be used to, first, learn and teach both SQL and Datalog languages in a single, database-shared environment. Second, to experiment with its features since it is free, open-source and furthermore is completely implemented with Prolog, a high-abstraction-level programming language. On the one hand, this allows to change its behaviour and add new features much more easily than either using a lower-abstraction-level language or using several languages.

On the other hand, it allows to test own proposals in the logic domain (as, for instance, language and database extensions, ontologies, and semantic web applications). And, third, to be used as a deductive component of other systems as, for instance, ontology semantic resources needing knowledge-based reasoning.

We think that the key features making DES a success are: easy-to-use/install interactive system, robust, multi-platform and a design which has been guided by demand in teaching. However, since this system is an ongoing project, many more improvements can (and most likely will) be included, as connections to existing relational DBMSs, enhanced performance, precise syntax errors reports, multiline input, multisets, and so on.

Acknowledgement

This work has been supported by projects TIN2008-06622-C03-01, S-0505/TIC/0407, S2009TIC-1465, and UCM-BSCH-GR58/08-910502. Also thanks to Jan Wielemaker for providing SWI-Prolog [24], Markus Triska for its SWI-Prolog FD library [22], Daniel Diaz for GNU-Prolog [8], and the CLIP group for Ciao Prolog [3]. Finally, to Rafael Caballero and Yolanda García-Ruiz for their contributions in making possible both the Datalog declarative debugger and test case generator.

References

[1] 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.

[2] 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.

[3] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog system. Reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from `http://www.clip.dia.fi.upm.es/`.

[4] 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 2008*, volume 4925 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2008.

[5] 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.

[6] 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.

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

[8] Daniel Diaz. GNU Prolog 1.3.1. A Native Prolog Compiler with Constraint Solving over Finite Domains, 2009. Available from http://www.gprolog.org/.

[9] Suzanne W. Dietrich. Extension tables: Memo relations in logic programming. In *SLP*, pages 264–272, 1987.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] G. Ramalingam and Eelco Visser, editors. *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*. ACM, 2007.

[16] 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.

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

[18] Fernando Sáenz-Pérez. Datalog Educational System V1.8.1, March 2010. http://des.sourceforge.net/.

[19] 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.

[20] SICStus Prolog, 2010. http://www.sics.se/isl/sicstus.

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

[22] Markus Triska. Generalising constraint solving over finite domains. In *International Conference on Logic Programming (ICLP)*, pages 820–821, 2008.

[23] Jeffrey D. Ullman. *Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1995.

[24] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.

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