A Deductive Database with Datalog and SQL Query Languages

Fernando Sáenz-Pérez†, Rafael Caballero‡, and Yolanda García-Ruiz‡*

Grupo de Programación Declarativa (GPD),
†Dept. Ingeniería del Software e Inteligencia Artificial,
‡Dept. Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Spain

Abstract. This paper introduces Datalog Educational System (DES), a deductive database which supports both Datalog and SQL as query languages. Since its inception, this system is targeted to educational purposes rather to develop an efficient, competitive system with respect to other existing systems. As distinguishing features, it is free, open-source, multiplatform, interactive, portable, GUI-enabled, implemented following ISO-Prolog and supports extensions to pure Datalog in the form of stratified negation, strong constraints, types, metapredicates, and duplicates. Also, test case generation for SQL views and declarative debugging for Datalog programs and SQL views are supported. SQL statements, following ISO standard, are compiled to Datalog programs and solved by its inference engine. Nonetheless, ODBC connections are also supported, which enables access to external DBMSs and benefit from their solving performance, persistency and scalability.

The original publication is available at www.springerlink.com [19].

Keywords: Deductive Databases, Relational Databases, Datalog, SQL, DES

1 Introduction

We have witnessed recently the advent of new interest on deductive databases and emerging companies promoting deductive technologies. Datalog, as a deductive query language, has been extensively studied and is gaining a renowned interest thanks to their application to ontologies [5], semantic web [7], social networks [16], policy languages [2], and even for optimization [9]. In addition, current companies as LogicBlox, Exeura, Semmle, and Lixto embody Datalog-based deductive database technologies in the solutions they develop.

* These authors have been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCG-GR35/10-A-910502)
This paper presents the Datalog Educational System (DES), which born from the need to teach deductive concepts to postgraduate students. As by that time there was no open-source, free, multiplatform, and interactive system, we decided to start this project. It was first released in 2004 and since then, many releases have been published including features as:

- Tabling-based deductive engine implementing stratified negation.
- Datalog and SQL query languages sharing the same database.
- Nulls and outer join operations.
- Duplicates and duplicate elimination.
- Text-based interactive system with commands for changing and examining its state, logging, batch execution, and many more.
- Source-level tracers and declarative debuggers for both Datalog and SQL.
- Strong constraints including types, primary and foreign keys, functional dependencies, and user-defined constraints.
- ODBC connections to seamlessly access external databases.

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 [17]. It features syntax colouring, project management, interactive console with edition and history, configurable buttons for commands, and more. On the other hand, Markus Triska contributed with an Emacs environment.

The system is implemented on top of Prolog and it can be run from a state-of-the-art Prolog interpreter (currently, last versions of Ciao, GNU Prolog, SWI-Prolog and SICStus Prolog) on any OS supported by such Prolog interpreter (i.e., almost any HW/SW platform). Portable executables (i.e., they do not need installation and can be run from any directory they are stored) has been also provided for Windows, Linux, and Mac OS X.

Datalog as supported by DES mainly follows Prolog ISO standard [10], whilst SQL follows SQL:2008 ISO standard [11]. DES provides several metapredicates as well, some of them are included to add support for SQL operations:

- **Negation.** `not(Goal)` computes the negation of `Goal` by means of negation as failure (closed world assumption (CWA) [21]). `Goal` is located at a higher strata than the predicate it occurs [21]).

- **Aggregates.** `group_by(Relation, Grouping_Variables, Condition)` creates groups from `Relation` w.r.t. `Grouping_Variables` and compute aggregate data with `Condition`, which can include aggregate functions as `sum(Variable)`, which returns the running sum of `Relation` w.r.t. `Variable`. If no grouping is needed or it is left to be done automatically, aggregate predicates are also available, as `sum(Relation, Variable, Result)`.

- **Duplicate elimination.** Metapredicate `distinct(Relation)` computes distinct tuples of `Relation` when duplicates are enabled (with the command `/duplicates on`). Also, `distinct(Projecting_Variables, Relation)` performs the same but finding distinct tuples for its first argument, which is a subset of variables of the second.
– **Outer join operations.** Nulls and null-related operations coming from the database community are allowed, as, e.g.: \( lj(L, R, C) \), which computes the left outer join of relations \( L \) and \( R \) w.r.t. the join condition \( C \) [18].

As a running example, we consider a flight database, which can be defined alternatively from either SQL or Datalog:

\% SQL:
CREATE TABLE flight(origin STRING, destination STRING, duration INT);
INSERT INTO flight VALUES ('Madrid', 'Paris', 90);
INSERT INTO flight VALUES ('Paris', 'Oslo', 100);
INSERT INTO flight VALUES ('Madrid', 'London', 110);

\% Datalog:
:-type(flight(origin:string, destination:string, duration:int)).
flight('Madrid', 'Paris', 90).
flight('Paris', 'Oslo', 100).
flight('Madrid', 'London', 110).

Each SQL statement above can be interactively introduced at the system prompt or stored in a file and processed with the command `process FileName`. The Datalog program can be also stored in a file and consulted with the command `consult FileName`. For inserting tuples (or rules, in general) in Datalog, the command `assert Rule` is provided. Whilst types are mandatory in table definition, in Datalog they are optionally declared with the assertion `:-type(Relation, [ColumnType])`, where `[ColumnType]` is a list of column names and its types `(ColumnName:Type)`. Once one of the programs above has been consulted, queries can be submitted from the system prompt, as the following:

DES> SELECT destination FROM flight WHERE origin='Madrid'
answer(flight.destination) -> { answer('London'), answer('Paris') }

DES> flight('Madrid',Destination,Duration)
{ flight('Madrid','London',110), flight('Madrid','Paris',90) }

However, while the first query returns the tuples of `flight` projected by the argument `destination`, the second does not. To get a similar output relation, temporary Datalog views are provided, which allow defining both the projection of columns and renaming of relations:

DES> dest(Destination) :- flight('Madrid',Destination,Duration)
{ dest('London'), dest('Paris') }

For the consulted program, a predicate dependency graph (PDG) [21] is built, which relates each predicate in the program with all the predicates used to compute its meaning. From this graph, a stratification [21] is computed, if it exists, so that negation and aggregate predicates are not involved in cycles. For solving a query, a subgraph restricted to the predicates occurring in the query is computed, so that only the meaning of relevant predicates are computed, following a top-down driven approach rather than a bottom-up. Even when a given PDG is non-stratififiable, it is possible that the subgraph for a given query could be as long as this subgraph does not involve such offending cycles.
2 System Architecture

Figure 1 barely depicts the system architecture of DES. Datalog programs are stored in an in-memory Prolog database. Datalog queries are solved by the deductive engine relying on a cache to store results from fixpoint computations. These are computed by using a tabling technique [6], which finds its roots in dynamic programming. The data structure holding these results is called extension table (ET). Displaying the results for a Datalog query amounts to inspect ET for entries matching the query after its solving. In turn, SQL views and tables are stored in two alternative repositories.

First alternative for dealing with SQL statements is to use the very same Prolog DB. For this, views are translated into Datalog programs and tables into predicates consisting only of facts. SQL row-returning statements are compiled to and executed as Datalog programs (basics can be found in [21]), and relational metadata for DDL statements are kept. For solving such a query, it is compiled to a Datalog program including the relation $answer/n$ with as many arguments as expected from the SQL statement. Then, this program is stored in the Prolog DB, and the Datalog query $answer(X_1, \ldots, X_n)$, where $X_i : i \in \{1, \ldots, n\}$ are $n$ fresh variables, is submitted. Results are cached in ET and displayed eventually from this table. 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 (further work may include an automatic type assertion via type inferencing). In order to maintain consistency, the cache is cleared whenever the database is updated via asserting Datalog rules, creating SQL views or modifying base tables via INSERT, DELETE or UPDATE SQL statements.

Second alternative is to use the ODBC bridge to access external databases therefore taking advantage from their solving performance, persistency and scalability. Submitting a CREATE VIEW SQL statement amounts to forward it to the external database through the ODBC connection. This statement is processed by the external database and operation success is returned to DES, which does not use the Prolog DB for SQL statements anymore. A SQL row-returning statement is also submitted through the bridge, which returns result tuples that are cached by DES. Datalog programs and queries can refer to SQL data because the bridge provides a view to external data sources as Datalog predicates. However, in this second alternative, the other way round is not possible yet, as the external data engine is not aware of the deductive data.
3 Source-to-source Program Transformations

Asserting a Datalog rule involves several steps. First, parsing builds a syntactic tree for valid rules. If errors are found, an exception is raised with error location and source data. Otherwise, a preprocessing stage is performed, consisting of:

- Simplify successive applications of \texttt{not(Goal)} to avoid more strata than strictly needed. As well, applications of this predicate to comparison built-ins (as \texttt{=}, \texttt{<}, \ldots) are translated to the complemented versions (\texttt{\neg =}, \texttt{\geq}, \ldots, resp.)

Finally, a compound goal is also allowed and it is defined as the body of a rule for a brand new predicate where its arguments are the relevant variables in \texttt{Goal}. The single argument of \texttt{not} is replaced by a straight call to this predicate (e.g., the goal \texttt{not((p(X),q(X)))} is translated into \texttt{not('$p1'(X))} and the rule \texttt{'$p1'(X):-p(X),q(X)} is added to the program).

- Aggregate predicates can include compound goals as aggregate relations. These relations are computed before the aggregation itself (sum, average, etc.) following a similar, stratified approach as for computing negation. Therefore, compound goals are also translated as for negation.

- Since disjunctive bodies are allowed, a rule containing a disjunction is transformed to as many rules as needed (e.g., \texttt{p(X):-q(X);r(X)} is translated into \texttt{p(X):-'$p1'(X)} and the rules \texttt{'$p1'(X):-q(X)} and \texttt{'$p1'(X):-r(X)} are added to the program).

- Program simplification can be enabled with the command \texttt{/simplify on}, which amounts to remove true goals, unify variables, simplify Boolean conditions, and evaluate arithmetic expressions.

- Program transformation for safe rules can also be enabled with the command \texttt{/safe on}, which reorder goals in a rule if this rule is unsafe (Section 4).

- Finally, outer joins are translated, first, in order to be solved without re-sorting to metapredicates as described in [18] and, second, so that relations to be joined are not compound by adding as many predicates as needed (e.g., \texttt{lj(rj(p(X),q(Y),X>Y),r(Z),Z\geq X)} would be translated into \texttt{lj('$p1'(X,Y)),r(Z),Z\geq X)} and rule \texttt{'$p1'(X,Y):-rj(p(X),q(Y),X>Y)} added to the database).

4 Compile-time Analyses

DES conducts compile-time analysis to detect unsafe rules. Classic safety [21, 23] refers to built-in predicates that can be source of infinite relations, in contrast to user-defined predicates, which are always finite. For instance, the rule \texttt{less(X,Y):-X<Y} is unsafe since the built-in \texttt{<} can be source of infinite data (its meaning must include any pair such that its first argument is less than its second one). Negation requires its argument to have no unsafe variables, i.e., those which are not bound by a former data provider (as a call to a user-defined predicate). Built-in \texttt{X is Expr} evaluates the arithmetical expression \texttt{Expr}, so that \texttt{Expr} is also demanded to be ground and, thus, all its variables must be safe.
Another source of unsafety, departing from the classical notion, resides in
metapredicates as distinct/2 and aggregates. A set variable is any variable
occurring in a metapredicate such that it is not bound by the metapredicate.
For instance, \( Y \) in the goal distinct([X], t(X,Y)) is a set variable, as well as in
group_by(t(X,Y), [X], C=count). Because computing a goal follows SLD order,
if a set variable is used after the metapredicate, as in distinct([X], t(X,Y)),
p(Y), then this is an unsafe goal as in the call to distinct, \( Y \) is not bound,
and all tuples in \( t/2 \) are considered for computing its outcome. Swapping both
subgoals yields a safe goal. So data providers for set variables are only allowed
before their use in such metapredicates.

Along program transformation, unsafe rules can be automatically generated,
as in the translations of outer joins. However, they are safe because of their
use: unsafe arguments of such rules are always given as input in goals. So, mode
information for predicates is handled throughout program compilations to detect
truly unsafe rules, avoiding to raise warnings about system generated rules.

The analysis allows deciding whether a rule is safe and, if so, it is transformed
by reordering the goals in order to make it computable. An error is raised when
a rule or query is actually unsafe (e.g., the rule \( p(X) :- X < Y \) is unsafe because
of \( Y \)), whereas a warning is issued if the rule might be safely computed (e.g.,
less(1,2) can be safely computed since its arguments are ground).

5 Strong Constraints

Consistency constraints over data are known as strong integrity constraints in the
deductive database area. Examples of such integrity constraints in the relational
field are primary keys and foreign keys, to name a few. As well, constraints
and well-founded model semantics [22], respectively, are otherwise understood
as model filters. In these cases, since a database can have several models, only
those fulfilling constraints are included in the answer, therefore discarding unfeasible
models from the answer. In DES, instead, we focus on integrity constraints
as understood in the relational field in order to provide a means to detect in-
consistent data with respect to user requirements, including types, primary and
foreign keys, functional dependencies, and user-defined constraints.

As an example of constraint, in addition to the type constraint in Section 1,
let's consider :-pk(flight,[origin,destination]), which defines the column
pair in the list to be a primary key for flight. Also, assuming:

\[
\text{connected}(\text{O},\text{D},\text{T}) \ :- \ \text{flight}(\text{O},\text{D},\text{T}) .
\]
\[
\text{connected}(\text{O},\text{D},\text{T}) \ :- \ \text{flight}(\text{O},\text{A},\text{T1}), \text{connected}(\text{A},\text{D},\text{T2}), \text{T} \text{ is } \text{T1+T2} .
\]

Then, :-group_by(connected(0,D,T), [0,D], S=sum(T)), S>=300 is a user
constraint limiting the duration from an origin to a destination to be less than
300 minutes. Notice that, as usual in the deductive field and contrary to the norm
in SQL, an integrity constraint specifies unfeasible values rather than feasible.
6 Tracing and Debugging

In contrast to imperative programming languages, deductive and relational database query languages feature solving procedures which are far from the query languages itself. Whilst one can trace an imperative program by following each statement as it is executed, along with the program state, this is not feasible in declarative (high abstraction) languages as Datalog and SQL.

Similarly, SQL represents a true declarative language which is even farthest from its computation procedure than Prolog. Indeed, the execution plan for a query include transformations considering data statistics to enhance performance. These query plans are composed of primitive relational operations (such as Cartesian product) and specialized operations for which efficient algorithms have been developed, containing in general references to index usage.

Therefore, instead of following a more imperative approach to tracing, here we focus on a (naïve) declarative approach which only take into account the outcomes at some program points. This way, the user can inspect each point and decide whether its outcome is correct or not. This approach will allow examining the syntactical graph of a query, which possibly depends on other views or predicates (SQL or Datalog, resp.) In the case of Datalog queries, this graph contains the nodes and edges in the dependency graph restricted to the query, ignoring other nodes which do not take part in its computation. In the case of SQL, the graph shows the dependencies between a view and its data sources (in the FROM clause). Available commands for tracing Datalog and SQL are /trace_datalog Goal and /trace_sql View, respectively.

Algorithmic debugging is also applied to both Datalog and SQL, following [3] and [4], respectively. Similar to how tracing traverses the dependency graph, the debugger in addition prunes paths in the graph by asking the user about validity of its nodes. Available commands for enabling this kind of debugging are /debug_datalog Goal and /debug_sql View.

7 Conclusions

This paper has presented DES, a deductive system used in many universities (http://des.sourceforge.net/des_facts) for which its downloading statistics (http://des.sourceforge.net/statistics) reveal 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 up to more than 1,500 downloads a month, more than 35,000 downloads since 2004.

Features that, as a whole, distinguish DES from other existing systems as DLV [14], XSB [20], bddbddb [13], LDL++ [1], ConceptBase [12], and .QL [15] include null support and outer join operations, duplicates, strong constraints, full-fledged arithmetic, multi-platform, interactivity, multi-language support, freeness, and open-sourcing, among others. In particular, no one supports outer join operations and full support for duplicates (not only for base relations as LDL++ but also for any rule).
References


