Datalog Educational System V3.3.1 User's Manual

Fernando Sáenz Pérez
Grupo de Programación Declarativa (GPD, Declarative Programming Group)
Universidad Complutense de Madrid (UCM)
July, 28th, 2013

Contents

1. Introduction

The Datalog Educational System (DES) is a free, open-source, multiplatform, portable, Prolog-based implementation of a deductive database system. DES 3.3.1 is the current implementation, which enjoys Datalog, Relational Algebra and SQL query languages, full recursive evaluation with memoization techniques, full-fledged arithmetic, stratified negation, duplicates and duplicate elimination, integrity constraints, ODBC connections to external relational database management systems (RDBMSs), Datalog and SQL tracers, a textual API for external applications, and novel approaches to hypothetical SQL queries, declarative debugging of Datalog queries and SQL views, test case generation for SQL views, modes, null values support, (tabled) outer join and aggregate predicates. The system is implemented on top of Prolog and it can be used from a Prolog interpreter running on any OS supported by such interpreter. Moreover, Windows, Linux and MacOSX executables are also provided.

We have developed DES aiming to have a simple, interactive, multiplatform, and affordable system (not necessarily efficient) for students, so that they can get the fundamental concepts behind a deductive database with Datalog, Relational Algrebra and SQL as query languages. SQL is supported with a reasonable coverage of the standard for teaching purposes. Supported (extended) relational algebra includes duplicates, outer joins and recursion. Other deductive systems are not fully suited to our needs due to the absence of some characteristics DES does offer for our educational purposes. This system is not targeted as a complete deductive database, so that it does not provide transactions, security, and other features present in current database systems.

The current release is mainly intended to fix a critical bug that makes some queries to be incomplete. This bug is due to the EDB optimization, which was enabled by default. However, there are some enhancements, as a preliminary and automatic inclusion of modes for unsafe predicates. These modes are understood more from a documentation point-of-view than from a constraint point-of-view, as mode assertions recall users about expected properties for the queries (in addition to the first message they got when compiling an unsafe rule). Also, the Datalog debugger has been enhanced a bit, and it does not require that the program to debug resides in a file, and in addition it enjoys less memory requirements. The complete list of enhancements, changes and fixed bugs are listed in Section 11.1.

A novel contribution implemented in this system is a declarative debugger of Datalog queries [CGS07,CGS08], which relies on program semantics rather than on the computation mechanism. The debugging process is usually started when the user detects an unexpected answer to a query. By asking questions about the intended semantics, the debugger looks for incorrect program relations. See Section 5.9 for details. Also, a similar declarative approach has been used to implement an SQL declarative debugger, following [CGS11b]. There, possible erroneous objects correspond to views, and the debugger looks for erroneous views asking the user whether the result of a given view is as expected. In addition, trusted views are supported to prune the number of questions. This was extended to also include user information about wrong and missing tuples [CGS12a]. See Section 5.10 for details. In addition, following the need for catching program errors when handling large amounts of data, we also include a test case generator for SQL correlated views [CGS10a]. Our tool can be used to generate positive, negative and both positive-negative test cases (cf. Section 5.11).

1.1 Deductive Databases

The intersection of databases, logic, and artificial intelligence delivered deductive databases. Deductive database systems are database management systems built around a logical model of data, and their query languages allow expressing logical queries. Relational database languages (where SQL is the *de-facto* standard) implement a limited form of logic whereas deductive database languages implement advanced forms of logic.

A deductive database is a system which includes procedures for defining deductive rules which can infer information (in the so-called intensional database) in addition to the facts loaded in the (so-called extensional) database. The logic model for deductive databases is closely related to the relational model and, in particular, with the domain relational calculus. Their query languages are related with the Prolog language and, mainly, with Datalog, a Prolog subset without constructed terms (in order to avoid infinite terms) and other non-declarative constructs such as the cut.

Origins of deductive databases can be found in automatic theorem proving and, later, in logic programming. Minker [Mink87] suggested that Green and Raphael [GR68] were the pioneers in discovering the relation between theorem proving and deduction in databases. They developed several question–answer systems using a version of the Robinson resolution principle [Robi65], showing that deduction can be systematically performed in a database environment. Other pioneer systems were MRPPS [MN82], DEDUCE–2 [Chan78] and DADM [KT81]. See Section 8 for references to other current deductive database systems.

2. Installation

2.1 Downloading DES

You can download the system from the DES web page via the URL:

`http://des.sourceforge.net/`

There, you can find source distributions for several Prolog interpreters and operating systems, and executable distributions for MS Windows, Linux and Mac OS.

2.1.1 Source Distribution

Under the source distribution, there are several versions depending on the Prolog interpreter you select to run DES: either SICStus Prolog [SICStus] or SWI Prolog [Wiele]. However, adapting the code in the file **`des_glue.pl`**, it could be ported to any other Prolog system. (See Section 5.17.3 for porting to unsupported systems.) We have tested DES under SICStus Prolog 4.2.3 and SWI–Prolog 6.4.1), and several operating systems (MS Windows XP/Vista/7, Ubuntu 10.04.1, Ubuntu 12.04, and MacOSX Snow Leopard).

The source distribution comes in a single archive file containing the following:

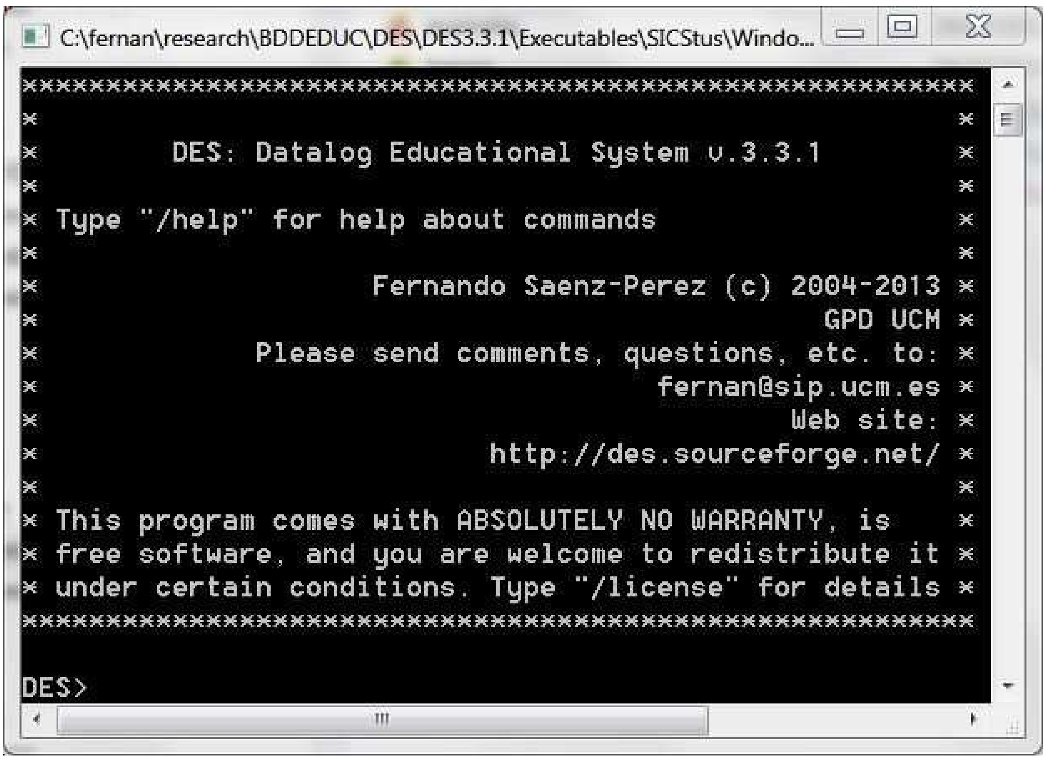
- **readmeDES<version>.txt**. A quick installation guide and file release contents
- **des.pl**. Core of DES, including Datalog processor
- **des_dcg.pl**. DCG expansion
- **des_sql.pl**. SQL processor
- **des_ra.pl**. RA processor
- **des_sql_debug.pl**. SQL declarative debugger
- **des_dl_debug.pl**. Datalog declarative debugger
- **des_types.pl**. Type inferrer for SQL, RA and Datalog
- **des_tc.pl**. Test case generator for SQL views
- **des_glue.pl**. Contains particular code for the selected host Prolog system
- **doc/manualDES<version>.pdf**. This manual
- **examples/*.dl** Example files which will be discussed in Section 6
- **license/license** A verbatim copy of the GNU Public License for this distribution

2.1.2 Executable Distribution

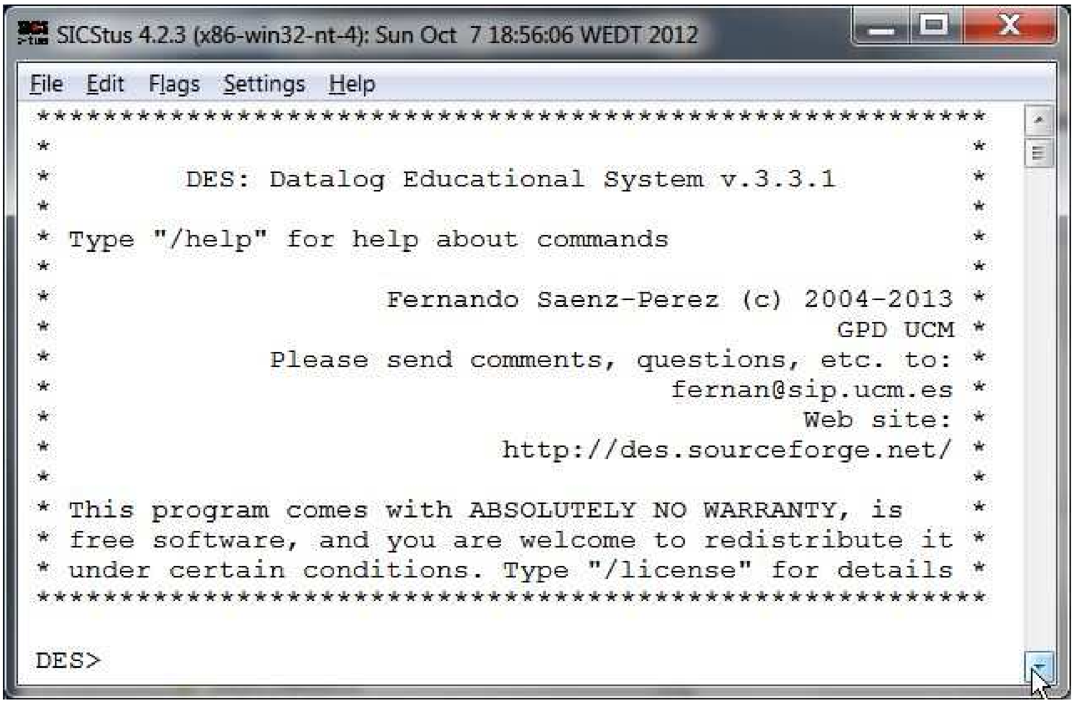
2.1.2.1 Windows

From the same URL above, you can download a Windows executable distribution in a single archive file containing the following:

- **readmeDES<version>.txt**. A quick installation guide and file release contents
- **des.exe**. Console executable file, intended to be started from a OS command shell, as depicted in the next figure:



- **deswin.exe**. Windows-application executable file, as depicted below:

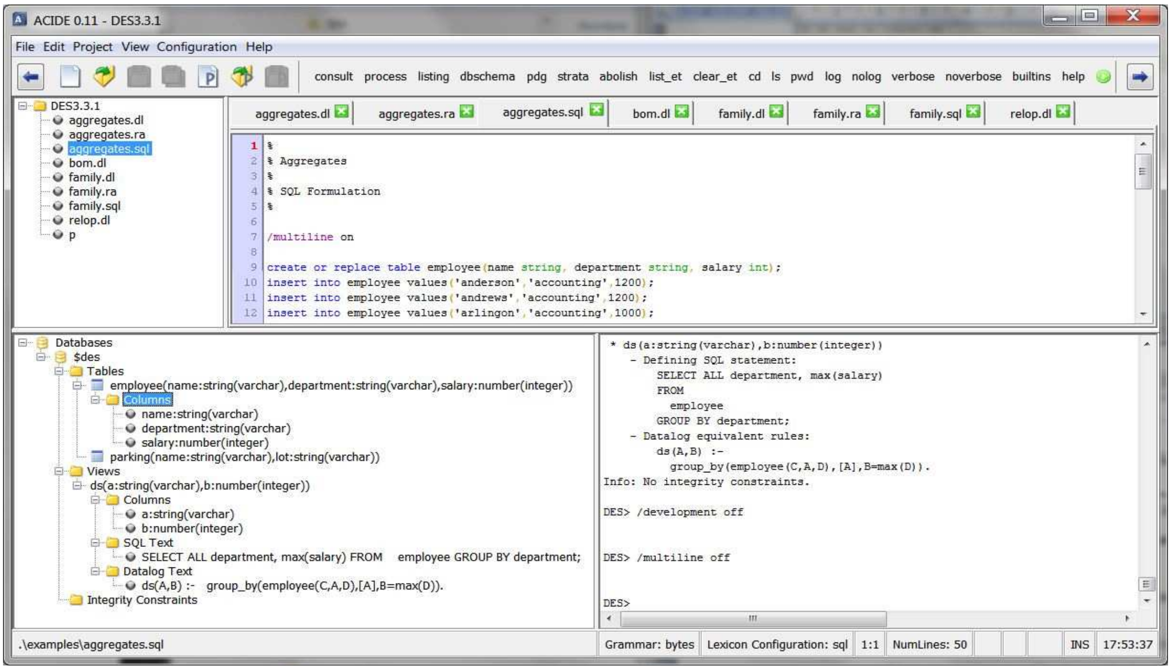


Please note that the menu bar above is inherited from the host Prolog system and all its settings apply to such system, not to DES.

- ***.dll**. DLL libraries for the runtime system
- **doc/manualDES<version>.pdf**. This manual
- **examples/*.dl** Example files which will be discussed in Section 6
- **license/license** A verbatim copy of the GNU Public License for this distribution

2.1.2.2 DES+ACIDE Bundle

From the same URL above, you can download a bundle including both DES and the integrated development environment ACIDE, preconfigured to work with DES. The following figure is a snapshot of the system:



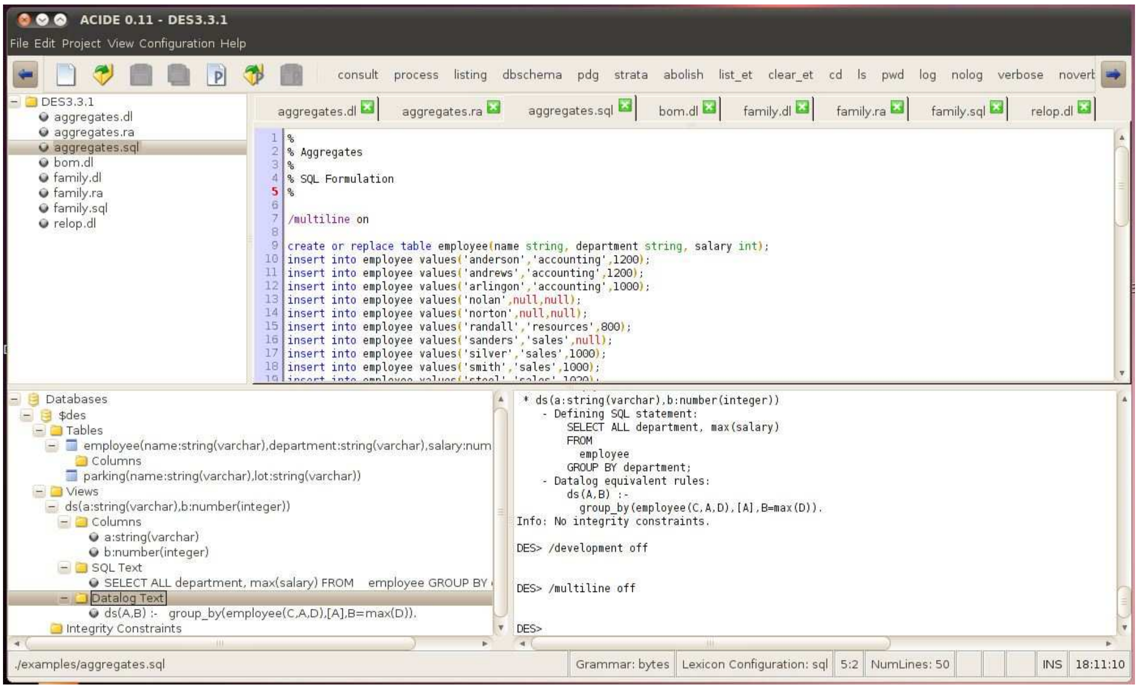
2.1.2.3 Linux

From the same URL above, you can download a Linux executable distribution in a single archive file containing the following:

- **readmeDES<version>**. A quick installation guide and file release contents
- **des**. Console executable file
- **doc/manualDES<version>.pdf**. This manual
- **examples/*.dl** Example files which will be discussed in Section 6
- **license/license** A verbatim copy of the GNU Public License for this distribution

The following screenshot has been taken in Ubuntu 10.04.1:

The same Windows ACIDE bundle can be downloaded for Linux. The following snapshot shows this running on Ubuntu 10.04:

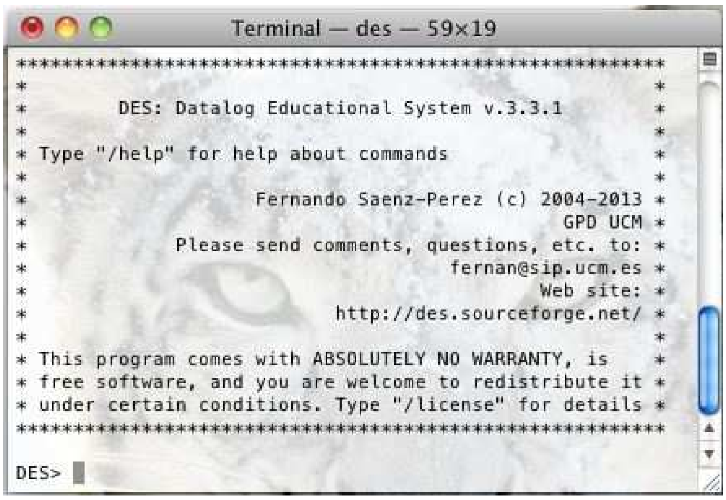


2.1.2.4 Mac OS X

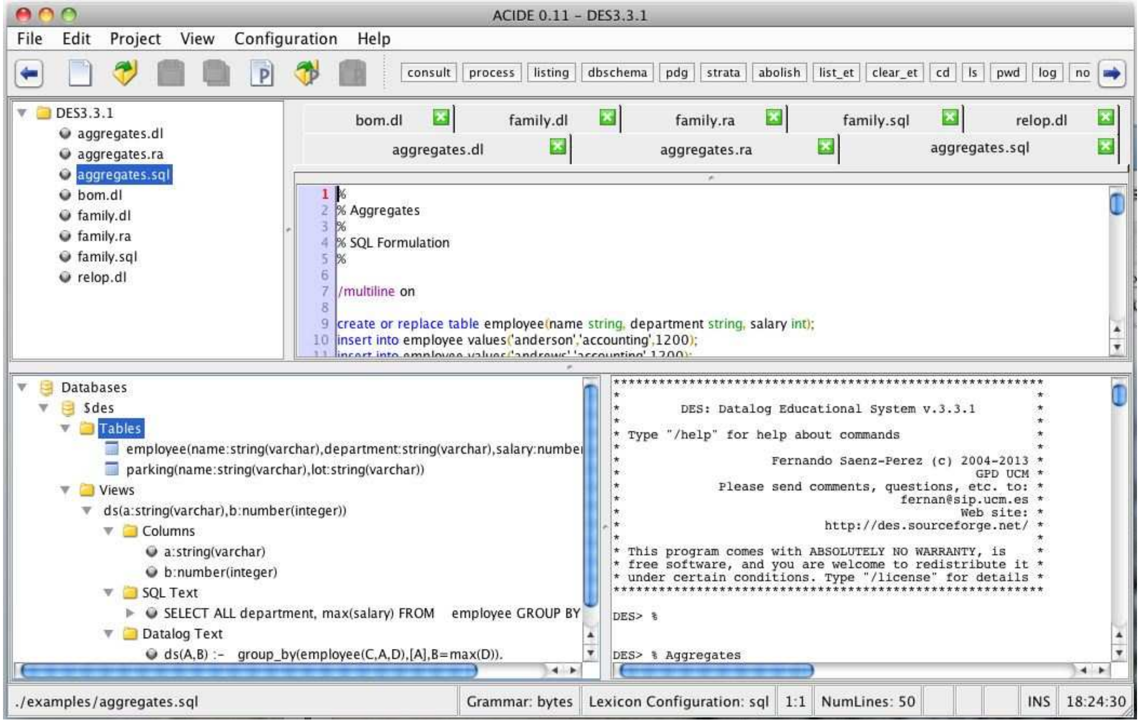
From the same URL above, you can download a Mac OS X executable distribution in a single archive file containing the following:

- **readmeDES<version>**. A quick installation guide and file release contents
- **des**. Console executable file
- **doc/manualDES<version>.pdf**. This manual
- **examples/*.dl** Example files which will be discussed in Section 6
- **license/license** A verbatim copy of the GNU Public License for this distribution

The following screenshot has been taken in Mac OS X Snow Leopard:



There is also an ACIDE bundle that can be downloaded for MacOSX. The following snapshot shows this running on MacOS Snow Leopard:



2.2 Installing and Executing DES

Unpack the distribution archive file into the directory you want to install DES, which will be referred to as the distribution directory from now on. This allows you to run the system, whether you have a Prolog interpreter or not (in this latter case, you have to run the system either on MS Windows, Linux or MacOS).

Although there is no need for further setup and you can go directly to Section 2.2.3, you can also configure a more user-friendly way for system start. In this way, you can follow two routes depending on the operating system.

2.2.1 MS Windows

2.2.1.1 Executable Distribution

Simply create a shortcut in the desktop for executing the executable of your choice: either **`des.exe`**, or **`deswin.exe`** or **`des_acide.jar`**. The former is a console-based executable, the second is a windows-based executable, and the latter is a Java application that includes a call to the binary **`des.exe`**. Executables have been generated with SICStus Prolog and SWI-Prolog, so that all notes relating these systems in the rest of this document also apply to these executables. In addition, since it is a portable application, it needs to be started from its distribution directory, which means that the start-up directory of the shortcut must be the distribution directory.

2.2.1.2 Source Distribution

Perform the following steps:

1. Create a shortcut in the desktop for running the Prolog interpreter of your choice.

2. Modify the start directory in the "Properties" dialog box of the shortcut to the installation directory for DES. This allows the system to consult the needed files at startup.
3. Append the following options to the Prolog executable path, depending on the Prolog interpreter you use:
   (a) SICStus Prolog: **`-l des.pl`**
   (b) SWI Prolog: **`-g "ensure_loaded(des)"`** (remove **`--win_app`** if present)

Another alternative is to write a batch file similar to the script file described in the next section.

2.2.2 Linux

2.2.2.1 Executable Distribution

You can create a script or an alias for executing the file **`des`** at the distribution root. This executable has been generated under SICStus Prolog, so that all SICStus notes in the rest of this document also apply to these executables. In addition, since it is a portable application, it needs to be started from its distribution directory.

2.2.2.2 Source Distribution

You can write a script for starting DES according to the selected Prolog interpreter, as follows:

(a) SICStus Prolog:

```
$SICSTUS –l des.pl
```

Provided that **`$SICSTUS`** is the variable which holds the absolute filename of the SICStus Prolog executable.

(b) SWI Prolog:

```
$SWI -g "ensure_loaded(des)"
```

Provided that **`$SWI`** is the variable which holds the absolute filename of the SWI Prolog executable.

2.2.3 Starting DES from a Prolog interpreter

Besides the methods just described, you can start DES from a Prolog interpreter, disregarding the OS and platform, first changing to the distribution directory, and then submitting:

```
?- [des].
```

Or better, if the system does support it:

```
?- ensure_loaded(des).
```

If the system does not start by itself, then type:

```
?- start.
```

3. Getting Started

Whichever method you use to start DES (a script, batch file, or shortcut, as described in Section 2.2), you get the following:

```
*********************************************************
*                                                       *
*        DES: Datalog Educational System v.3.1          *
*                                                       *
* Type "/help" for help about commands                  *
*                                                       *
*                    Fernando Saenz-Perez (c) 2004-2013 *
*                                               GPD UCM *
*             Please send comments, questions, etc. to: *
*                                     fernan@sip.ucm.es *
*                                             Web site: *
*                           http://des.sourceforge.net/ *
*                                                       *
* This program comes with ABSOLUTELY NO WARRANTY, is    *
* free software, and you are welcome to redistribute it *
* under certain conditions. Type "/license" for details *
*********************************************************

DES>
```

This last line (**DES>**) is the DES system prompt, which allows you to write Datalog, SQL and Relational Algebra (RA) queries, commands, temporary views and conjunctive queries (see next sections). If an error leads to an exit from DES and you have started from a Prolog interpreter, then you can write "**des.**" (*without* the double quotes and *with* the dot) at the Prolog prompt to continue.

Although a query in any of the languages above can be submitted from such prompt, there are currently four modes available which enable to use a concrete query interpreter for Datalog, SQL, Relational Algebra and Prolog. The first one is the default. A mode can be switched via the commands **/datalog**, **/sql**, **/ra** and **/prolog**, respectively. Note that commands always start with a slash (**/**). Anyway, if you are in a given mode, you can submit queries or goals to other interpreters simply writing the query or goal after any of the previous commands. Also, if you are in Datalog mode, you can directly submit both SQL and RA queries.

Data are stored in a deductive database, including facts and rules. All queries and goals, irrespective of the language, refer to this database. When an external database is opened (see Section 5.1), their tables and views are available and can be queried from Datalog, Prolog and SQL.

In contrast with other interpreters, default input mode is single-line, which means that the input will be processed after hitting the Intro key, which allows to omit the terminating character. Nonetheless, this mode can be switched to multi-line as described in Section 5.6 with the command **/multiline on**.

3.1 Datalog Mode

In this mode, a query is sent to the Datalog processor. If it does not follow Datalog syntax, then it is sent, first, to the SQL processor (see Section 0) and, second, to the RA processor (see Section 4.3) should such query is written in any of these other query languages (See caveats in Section 3.5). Commands (see Section 5.14) are sent to the command processor. Commands can end with an optional dot. In single-line mode, Datalog inputs can also end with an optional dot, but the dot is required in multi-line mode. Datalog mode is the default and can be anyway enabled via the command **/datalog**.

The typical way of using the system is to write Datalog program files (with default extension **.dl**) and consulting them before submitting queries. Another alternative is to assert program rules from the system prompt.

Following the first alternative, you write the program in a text file, and then change to the path where the file is located by using the command **/cd *Path***, where ***Path*** is the new directory (relative or absolute). Next, the command **/consult *FileName*** is used to consult the file ***FileName***.

Provided there are a number or example files in the directory **examples** at the distribution directory, and assuming that the current path is the distribution directory (as by default), one can use the following commands to consult the example file **relop.dl**:[1]

```
DES> /cd examples

DES> /consult relop.dl
Info: 18 rules consulted.
```

(where the default extension **.dl** can be omitted). Note that rules in files must end with a dot, in contrast to command prompt inputs, where the dot is optional in single-line input. Rules in a consulted file may span on multiple lines.

Then, one can examine the contents of the database (see Section 6.1 for an explanation of the consulted program) via the command:

```
DES> /listing

a(a1).
a(a2).
a(a3).
b(a1).
b(b1).
b(b2).
c(a1,a1).
c(a1,b2).
c(a2,b2).
cartesian(X,Y) :-
  a(X),
  b(Y).
difference(X) :-
  a(X),
  not(b(X)).
full_join(X,Y) :-
  fj(a(X),b(Y),X = Y).
```

[1] See section 5 for more details about commands.

```
inner_join(X) :-
  a(X),
  b(X).
left_join(X,Y) :-
  lj(a(X),b(Y),X = Y).
projection(X) :-
  c(X,Y).
right_join(X,Y) :-
  rj(a(X),b(Y),X = Y).
selection(X) :-
  a(X),
  X = a2.
union(X) :-
  a(X)
  ;
  b(X).

Info: 18 rules listed.
```

Submitting a query is pretty easy:

```
DES> a(X)
{
  a(a1),
  a(a2),
  a(a3)
}
Info: 3 tuples computed.
```

You can interactively add new rules with the command **`/assert`**, as in:

```
DES> /assert a(a4)
DES> a(X)
{
  a(a1),
  a(a2),
  a(a3),
  a(a4)
}
Info: 4 tuples computed.
```

Saving the current database, which may include such interactively added (or deleted) tuples, is allowed with the command **`/save_ddb`** ***`Filename`***, which saves in a plain file the Datalog rules in memory. Later, they can be restored with **`/restore_ddb`** ***`Filename`*** (this command is only an alias for **`/consult`**.) In the following session, the current database is stored, abolished (cleared), and finally restored. All the data, including the ones interactively added have been recovered:

```
DES> /save_ddb db.dl
DES> /abolish
DES> /restore_ddb db.dl
Info: 19 rules consulted.
DES> a(X)
{
  a(a1),
```

```
  a(a2),
  a(a3),
  a(a4)
}
Info: 4 tuples computed.
```

Another useful command is **/list_et**, which lists, in particular, the answers already computed. Following the last series of queries and commands above, we submit:

```
Answers:
{
  a(a1),
  a(a2),
  a(a3),
  a(a4)
}
Info: 4 tuples in the answer table.
Calls:
{
  a(A)
}
Info: 1 tuple in the call table.
```

Here, we can see that the computed meaning of the queried relation is stored in an extension table, as well as the last call (cf. sections 5.17.1 and 5.17.2). Unless either the database is changed (e.g., via **/assert** or **/retract** commands) or a temporary view (see Section 4.1.6) executed or the command **/clear_et** is submitted, the extension table keeps computed results, otherwise it is cleared.

3.2 SQL Mode

In this mode, queries are sent to the SQL processor, whereas commands (cf. Section 5.14) are sent to the command processor. SQL queries can end with an optional semicolon in single-line mode. Multi-line mode requires the ending semicolon. SQL mode is enabled via the command **/sql**. Datalog and RA queries cannot be handled by this mode.

If we want to develop an analogous SQL example session to the Datalog example in the last section, we can submit the first inputs (also available in the file **examples/relop.sql**) listed below (the example is augmented to provide a first glance of SQL). Now, answer relations to SQL queries are denoted by the relation name **answer**. Also note that lines starting by **%** are simply remarks. If you wish to automatically reproduce the following interactive session of inputs, you can type **/process examples/relop.sql** (notice that you must omit **examples/** if you are in this directory already):

```
Info: Processing file 'relop.sql' ...
DES> % Switch to SQL interpreter
DES> /sql
DES-SQL> % Creating tables
DES-SQL> create or replace table a(a string);
DES-SQL> create or replace table b(b string);
DES-SQL> create or replace table c(a string,b string);
```

```
DES-SQL> % Listing the database schema
DES-SQL> /dbschema
Info: Table(s):
 * a(a:string(varchar))
 * b(b:string(varchar))
 * c(a:string(varchar),b:string(varchar))
Info: No views.
Info: No integrity constraints.
DES-SQL> % Inserting values into tables
DES-SQL> insert into a values ('a1');
Info: 1 tuple inserted.
DES-SQL> insert into a values ('a2');
Info: 1 tuple inserted.
DES-SQL> insert into a values ('a3');
Info: 1 tuple inserted.
DES-SQL> insert into b values ('b1');
Info: 1 tuple inserted.
DES-SQL> insert into b values ('b2');
Info: 1 tuple inserted.
DES-SQL> insert into b values ('a1');
Info: 1 tuple inserted.
DES-SQL> insert into c values ('a1','b2');
Info: 1 tuple inserted.
DES-SQL> insert into c values ('a1','a1');
Info: 1 tuple inserted.
DES-SQL> insert into c values ('a2','b2');
Info: 1 tuple inserted.
DES-SQL> % Testing the just inserted values
DES-SQL> select * from a;
answer(a.a) ->
{
  answer(a1),
  answer(a2),
  answer(a3)
}
Info: 3 tuples computed.
DES-SQL> select * from b;
answer(b.b) ->
{
  answer(a1),
  answer(b1),
  answer(b2)
}
Info: 3 tuples computed.
DES-SQL> select * from c;
answer(c.a, c.b) ->
{
  answer(a1,a1),
  answer(a1,b2),
  answer(a2,b2)
}
Info: 3 tuples computed.
DES-SQL> % Projection
DES-SQL> select a from c;
```

```
answer(c.a) ->
{
  answer(a1),
  answer(a2)
}
Info: 2 tuples computed.
DES-SQL> % Selection
DES-SQL> select a from a where a='a2';
answer(a.a) ->
{
  answer(a2)
}
Info: 1 tuple computed.
DES-SQL> % Cartesian product
DES-SQL> select * from a,b;
answer(a.a, b.b) ->
{
  answer(a1,a1),
  answer(a1,b1),
  answer(a1,b2),
  answer(a2,a1),
  answer(a2,b1),
  answer(a2,b2),
  answer(a3,a1),
  answer(a3,b1),
  answer(a3,b2)
}
Info: 9 tuples computed.
DES-SQL> % Inner Join
DES-SQL> select a from a inner join b on a.a=b.b;
answer(a) ->
{
  answer(a1)
}
Info: 1 tuple computed.
DES-SQL> % Left Join
DES-SQL> select * from a left join b on a.a=b.b;
answer(a.a, b.b) ->
{
  answer(a1,a1),
  answer(a2,null),
  answer(a3,null)
}
Info: 3 tuples computed.
DES-SQL> % Right Join
DES-SQL> select * from a right join b on a.a=b.b;
answer(a.a, b.b) ->
{
  answer(a1,a1),
  answer(null,b1),
  answer(null,b2)
}
Info: 3 tuples computed.
DES-SQL> % Full Join
```

```
DES-SQL> select * from a full join b on a.a=b.b;
answer(a.a, b.b) ->
{
  answer(a1,a1),
  answer(a1,null),
  answer(a2,null),
  answer(a3,null),
  answer(null,a1),
  answer(null,b1),
  answer(null,b2)
}
Info: 7 tuples computed.
DES-SQL> % Union
DES-SQL> select * from a union select * from b;
answer(a.a) ->
{
  answer(a1),
  answer(a2),
  answer(a3),
  answer(b1),
  answer(b2)
}
Info: 5 tuples computed.
DES-SQL> % Difference
DES-SQL> select * from a except select * from b;
answer(a.a) ->
{
  answer(a2),
  answer(a3)
}
Info: 2 tuples computed.
Info: Batch file processed.
```

Duplicates are disabled by default, i.e., answers are set-oriented. But they can be enabled as well, which is useful in Datalog, SQL and RA queries (see Section 4.1.9). For instance:

```
DES-Prolog> /duplicates on
Info: Duplicates are on.

DES-Prolog> /datalog projection(X)
{
  projection(a1),
  projection(a1),
  projection(a2)
}
Info: 3 tuples computed.
```

3.3 Relational Algebra Mode

In this mode, queries are sent to the Relational Algebra (RA) processor, whereas commands (cf. Section 5.14) are sent to the command processor. RA queries can end with an optional semicolon in single-line mode. Multi-line mode requires the ending semicolon. RA mode is enabled via the command **/ra**. Datalog and SQL queries cannot be handled by this mode.

If we want to develop an analogous RA example session to the former examples, we can submit the first inputs (also available in the file **examples/relop.ra**) listed below. Now, answer relations to RA queries are denoted by the relation name **answer**. As before, lines starting by either **%** or **--** are simply remarks. If you wish to automatically reproduce the following interactive session of inputs, you can type **/process examples/relop.ra** (notice that you must omit **examples/** if you are in this directory already):

```
DES-RA> % Testing the just inserted values
DES-RA> select true (a);
answer(a.a:string(varchar)) ->
{
  answer(a1),
  answer(a2),
  answer(a3)
}
Info: 3 tuples computed.
DES-RA> select true (b);
answer(b.b:string(varchar)) ->
{
  answer(a1),
  answer(b1),
  answer(b2)
}
Info: 3 tuples computed.
DES-RA> select true (c);
answer(c.a:string(varchar),c.b:string(varchar)) ->
{
  answer(a1,a1),
  answer(a1,b2),
  answer(a2,b2)
}
Info: 3 tuples computed.
DES-RA> % Projection
DES-RA> project a (c);
answer(c.a:string(varchar)) ->
{
  answer(a1),
  answer(a2)
}
Info: 2 tuples computed.
DES-RA> % Selection
DES-RA> select a='a2' (a);
answer(a.a:string(varchar)) ->
{
  answer(a2)
}
Info: 1 tuple computed.
DES-RA> % Cartesian product
DES-RA> a product b;
answer(a.a:string(varchar),b.b:string(varchar)) ->
```

```
{
  answer(a1,a1),
  answer(a1,b1),
  answer(a1,b2),
  answer(a2,a1),
  answer(a2,b1),
  answer(a2,b2),
  answer(a3,a1),
  answer(a3,b1),
  answer(a3,b2)
}
Info: 9 tuples computed.
DES-RA> % Theta Join
DES-RA> select a.a=b.b (a product b);
answer(a.a:string(varchar),b.b:string(varchar)) ->
{
  answer(a1,a1)
}
Info: 1 tuple computed.
DES-RA> a zjoin a.a=b.b b;
answer(a.a:string(varchar),b.b:string(varchar)) ->
{
  answer(a1,a1)
}
Info: 1 tuple computed.
DES-RA> % Natural Inner Join
DES-RA> a njoin c;
answer(a.a:string(varchar),c.b:string(varchar)) ->
{
  answer(a1,a1),
  answer(a1,b2),
  answer(a2,b2)
}
Info: 3 tuples computed.
DES-RA> % Left Outer Join
DES-RA> a ljoin a.a=b.b b;
answer(a.a:string(varchar),b.b:string(varchar)) ->
{
  answer(a1,a1),
  answer(a2,null),
  answer(a3,null)
}
Info: 3 tuples computed.
DES-RA> % Right Outer Join
DES-RA> a rjoin a.a=b.b b;
answer(a.a:string(varchar),b.b:string(varchar)) ->
{
  answer(a1,a1),
  answer(null,b1),
  answer(null,b2)
}
Info: 3 tuples computed.
DES-RA> % Full Outer Join
DES-RA> a fjoin a.a=b.b b;
```

```
answer(a.a:string(varchar),b.b:string(varchar)) ->
{
  answer(a1,a1),
  answer(a2,null),
  answer(a3,null),
  answer(null,b1),
  answer(null,b2)
}
Info: 5 tuples computed.
DES-RA> % Union
DES-RA> a union b;
answer(a.a:string(varchar)) ->
{
  answer(a1),
  answer(a2),
  answer(a3),
  answer(b1),
  answer(b2)
}
Info: 5 tuples computed.
DES-RA> % Difference
DES-RA> a difference b;
answer(a.a:string(varchar)) ->
{
  answer(a2),
  answer(a3)
}
Info: 2 tuples computed.
DES-RA> % Intersection
DES-RA> a intersect b;
answer(a.a:string(varchar)) ->
{
  answer(a1)
}
Info: 1 tuple computed.
DES-RA> % Grouping
DES-RA> group_by a a,count(*) true (c);
answer(c.a:string(varchar),$a3:number(integer)) ->
{
  answer(a1,2),
  answer(a2,1)
}
Info: 2 tuples computed.
DES-RA> % Renaming
DES-RA> select a1.a<a2.a ((rename a1(a) (a)) product (rename
a2(a) (a)));
answer(a1.a:string(varchar),a2.a:string(varchar)) ->
{
  answer(a1,a2),
  answer(a1,a3),
  answer(a2,a3)
}
Info: 3 tuples computed.
DES-RA> % Duplicate elimination
```

```
DES-RA> /duplicates off
Info: Duplicates are already disabled.
DES-RA> project a (c);
answer(c.a:string(varchar)) ->
{
  answer(a1),
  answer(a2)
}
Info: 2 tuples computed.
DES-RA> /duplicates on
DES-RA> project a (c);
answer(c.a:string(varchar)) ->
{
  answer(a1),
  answer(a1),
  answer(a2)
}
Info: 3 tuples computed.
DES-RA> distinct (project a (c));
answer(c.a:string(varchar)) ->
{
  answer(a1),
  answer(a2)
}
Info: 2 tuples computed.
```

3.4 Prolog Mode

This mode is enabled via the command **`/prolog`** and goals are sent to the Prolog processor. Assuming that the file **`relop.dl`** has been already consulted, let's consider the following example:

```
DES-Prolog> projection(X)
projection(a1)
? (type ; for more solutions, <Intro> to continue) ;
projection(a1)
? (type ; for more solutions, <Intro> to continue) ;
projection(a2)
? (type ; for more solutions, <Intro> to continue) ;
no

DES-Prolog> /datalog projection(X)
{
  projection(a1),
  projection(a2)
}
Info: 2 tuples computed.
```

The execution of this goal allows to noting the basic differences between Prolog and Datalog engines. First, the former searches for solutions, one-by-one, that satisfy the goal **`projection(X)`**. The latter gives the whole meaning[2] of the user-defined relation **`projection`** with the query **`projection(X)`** at a time. And, second, note the default set-oriented behaviour of the Datalog engine, which discards duplicates in the answer.

3.5 Caveats

Since the Datalog mode prompt accepts Datalog, SQL and RA queries, a given query can be interpreted in more than one language. Let's consider the following system session, in which a table is created and an RA query is submitted:

```
DES> create table t(a int)
DES> distinct (t)
Info: Processing:
  answer :-
    distinct(t).
Warning: Undefined predicate(s): [t/0]
{
}
Info: 0 tuples computed.
```

Here, we get an unexpected output coming from the Datalog interpreter, as such input could be interpreted both as a Datalog query and an RA query. To overcome such situations, simply precede the query by the language selection command, as follows:

```
DES> /ra distinct (t)
answer(t.a:number(integer)) ->
{
}
Info: 0 tuples computed.
```

Alternatively, switch to the other query processor:

```
DES> /ra
DES-RA> distinct (t)
```

3.6 Getting Help

You can get useful information with the following commands:

- **`/help.`** Shows the list of available commands, which are explained in Section 5.14.
- **`/help`** ***`Keyword`*****`.`** To request help on a given keyword (command or built-in).
- **`/builtins.`** Shows the list of built-ins, which are explained in Section 4.5.

Also, visit the URL for last information:

`http://des.sourceforge.net/`

Finally, you can contact the author via the e-mail address:

[2] The meaning of a relation is the set of facts inferred both extensionally and intensionally from the program.

fernan@sip.ucm.es

4. Query Languages

DES has evolved from a quite simple Datalog interpreter to its current state, which relies on a deductive database engine which can be queried with either Datalog, SQL or RA languages. In addition, a Prolog interface is also provided in order to highlight the differences between Datalog and Prolog systems. Since DES is intended to students, it has no full-blown features of either state-of-the-art Prolog, Datalog or SQL-based systems. However, it has many features that make it appealing as an educational tool, along with the novel implementations of declarative debugging (sections 5.9 and 5.10) and the test case generator (Section 5.11). In this section, we describe its four query languages: Datalog, SQL, RA, and Prolog.

The database is shared by all the query languages, so that queries or goals can refer to any object defined using any language. However, there are some dependent issues that must be taken into account. For instance, once a Datalog fact is loaded into the database, the relation it defines can be queried in Datalog. But, if one wants to access this relation from either SQL or RA, two alternatives are provided: 1) Define the same relation in SQL via a **`create table`** statement (Section 4.2.4.1), and 2) Declare types for the table (Section 4.1.15.1). This particular issue comes from the fact that Datalog relations have unnamed attributes, and a positional reference is used for accessing those relations. In turn, SQL and RA use a notational syntax, giving names to relation arguments. To illustrate the first alternative, let's consider the following session:

```
DES> /assert t(1)
DES> t(X)
{
  t(1)
}
Info: 1 tuple computed.
DES> select * from t
Error: Unknown table or view "t"
DES> create table t(a int);
DES> select * from t;
answer(t.a:number(integer)) ->
{
  answer(1)
}
Info: 1 tuple computed.
```

The error above reflects that **`t`** is not a known object in the database schema.

Following the second alternative to access a Datalog relation from SQL:

```
DES> /assert t(1)
DES> :-type(t,[a:int])
DES> select * from t
answer(t.a:number(integer)) ->
{
  answer(1)
```

```
}
Info: 1 tuple computed.
```

4.1 Datalog

Since Datalog stems from Prolog, we have adopted almost all the Prolog syntax conventions for writing Datalog programs (the reader is assumed to have basic knowledge about Prolog). Syntax follows Prolog ISO standard [ISO00] (considering its syntax as a subset of Prolog). We allow (recursive) Datalog programs with stratified negation [Ullm95], i.e., normal logic programs without function symbols. Stratification is imposed to ensure a clear semantics when negation is involved, and function symbols are not allowed in order to guarantee termination of queries, a natural requirement with respect to a (relational) database user who is not able to deal with compound data.

Commands are somewhat different for Prolog programmers as they are accustomed to (see Section 5.14). Also, exceptions are noted when necessary.

4.1.1 Syntax

Definitions for Datalog mainly come from the field of Logic Programming., following [Lloyd87], referring the reader to this book for a more general presentation of Logic Programming. Next, some definitions for understanding the syntax of programs, queries and views are introduced.

- Numbers. Integers and float numbers are allowed. A number is a float whenever the number contains a dot (`.`) between two digits. The range depends on the Prolog platform being used. Negative numbers are identified by a preceding minus (`-`), as usual.

  Scientific notation is supported as: **`aEb`**, where **`a`** is a fractional number (always including a dot), and **`b`** is an integer, which may start with **`+`** or **`–`** (but it is not required).

  Examples of numbers are **`1`**, **`1.1`**, **`-1.0`**, **`1.2E34`**, **`1.2E+34`**, and **`1.2E-34`**.

  Note that **`-1.`**, **`+1`**, **`.1`**, **`1.E23`**, and **`1E23`** are not valid numbers. A plus sign is not part of a positive number; however, both a plus and a minus sign can be used as a prefix unary operator in arithmetical expressions (cf. Section 4.5.4.1) and also following the symbol **`E`** in scientific notation, as already seen.

- Constants. A constant can be:
  - A number (integer or float).
  - Any sequence of alphanumeric characters (including the underscore _), starting with a lowercase letter
  - Any sequence of characters delimited by single quotes.

  Examples of alphanumeric constants are **`foo`**, **`foo_foo`**, **`'foo foo'`**, **`'2*3'`**, and **`'X'`**.

- Variables. Variables are written with alphanumeric characters, and alternatively start with either an uppercase or with an underscore (_). Anonymous variables are also allowed, which are denoted with a single underscore. Each occurrence of an anonymous variable is considered different from any other anonymous variable. For instance, in the rule **`a :- b(_),c(_).`** both goals do not share variables. Any variable starting with an underscore (either anonymous or not) is removed from a computed query (cf. Section 4.1.7).

Examples of variables are: **`X`**, **`_X`**, **`_var`**, and **`_`**.

- Unknowns. Unknowns are represented as null values and are written alternatively as both **`null`** and **`'$NULL'(ID)`**, where ***`ID`*** is a unique identifier. The first form is used for normal users, whilst the second one is intended for development uses (cf. **`development`** command in Section 5.14.7).
- Terms. Terms can be:
  - Noncompound. Variables or constants.
  - Compound. As in Prolog, they have the form **`t(t1, ..., tn)`**, where **`t`** is a function symbol (functor), and **`ti`** (**1** ≤ **i** ≤ **n**) are terms.

  Up to the current version, compound terms can only occur in arithmetic expressions. Their function symbols can be any of the built-in arithmetic operators and functions (cf. Section 4.5.2). These operators can be:
  - Infix, as addition (e.g., **`1+2`**)
  - Prefix, as bitwise negation (e.g., **`\1`**)

  Examples of terms are: **`r(p)`**, and **`p(X,Y)`**, and **`X > Y`**.
- Atoms. An atom has the form **`a(t1, ..., tn)`**, where **`a`** is a predicate (relation) symbol, and **`ti`** (**0** ≤ ***i*** ≤ ***n***) are terms. If **`i`** is **`0`**, then the atom is simply written as **`a`**.

  Positive, ground atoms are used to build the Herbrand universe.

  There are several built-in predicates: **`is`** (for evaluating arithmetical expressions), arithmetic functions, (infix and prefix) operators and constants, and comparison operators. Comparison operators are infix, as "less-than". For example, **`1 < 2`** is a positive atom built from an infix built-in comparison operator (see Section 4.5.1).

  Examples of atoms are: **`p`**, **`r(a,X)`**, **`1 < 2`**, and **`X is 1+2`**.

  Note that **`p(1+2)`** and **`p(t(a))`** are not valid atoms.
- Conditions. A condition is a Boolean expression containing conjunctions (**`,/2`**), disjunctions (**`;/2`**), built-in comparison operators, constants and variables.

  Four examples of conditions are: **`X>1`**, **`X=Y`**, **`(X>Y,Y>Z)`**, **`(X=<Y;Z<0)`**.

  Note that **`X>Y+Z`** is now supported; it can be solved whenever the rule where it occurs is safe (cf. Section 5.3).
- Relation functions. A function has the form **`f(a1, …, an)`**, where **`f`** is a function name, **`ai`** are its arguments, and maps to a relation. Only built-in functions are allowed. The current provision of built-in functions includes, among others:
  - **`not(a)`**. Intended for computing the negation of its single argument **`a`**.
  - **`lj(a1,a2,a3)`**. Intended for computing the *left* outer join of the relations **`a1`** (left relation) and **`a2`** (right relation), committing the condition (Boolean expression) **`a3`** (join condition).

- o **rj(a1,a2,a3)**. Intended for computing the *right* outer join of the relations **a1** (left relation) and **a2** (right relation), committing the condition (Boolean expression) **a3** (join condition).
- o **fj(a1,a2,a3)**. Intended for computing the *full* outer join of the relations **a1** (left relation) and **a2** (right relation), committing the condition (Boolean expression) **a3** (join condition).

Note that outer join functions can be nested.

- Literals. Literals can be:
  - o Positive. An atom.
  - o Negative. A negated body of the form **not(*Body*)**, where ***Body*** is a body (cf. next section). Negative literals are used to express the negation of a relation (either as a query or as a part of a rule body).
  - o Disjunctive. A disjunctive literal is of the form **l;r** , where **l** and **r** are literals.
  - o Divided. A divided literal is of the form **l** division **r**, where **l** and **r** are literals.

Examples of literals are:

- **p**
- **r(a,X)**
- **not(q(X,b))**
- **not(a;b)**
- **r(a,X);not(q(X,b))**,
- **1 < 2**
- **t(X,Y) division s(Y)**
- **X is 1+2**

Shorthands for compound goals as **not(a;b)** are allowed as well, which stands for **not((a;b))**.

A literal can occur in rule bodies, queries, and view bodies.

Syntax of built-ins are explained in their corresponding forthcoming sections.

4.1.2 Rules

Datalog rules have the form **head :- body**, or simply **head**. Both end with a dot. A Datalog head is a positive atom that uses no built-in predicate symbol. A Datalog body contains a comma-separated sequence of literals which may contain built-in symbols as listed in Section 4.5, as well as disjunctions (**;**/2) and divisions (**division**/2).

4.1.3 Programs

DES programs consist of a multiset of rules. Programs may contain remarks. A single-line remark starts with the symbol **%**, and ends at the end of line. Consulted programs can also contain multi-line remarks, enclosed between **/*** and ***/**, which can be nested.

4.1.4 Queries

A (positive) query is the name of a relation with as many arguments as the arity of the relation (a positive literal). Each one of these arguments can be a variable or a constant; a compound term is not allowed but as an arithmetic expression. Built-in relations may require relations and conditions as arguments. A negative query is written as **not(Query)**.

Queries are typed at the DES system prompt. The answer to a query is the (multi)set of atoms 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 the arguments of the queried relation gives the whole set of deduced facts (meaning) defining the relation, as the query **a(X)** in the example of Section 3. 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 with the constant (i.e., analogous to a select relational operation). This is the case of the query **a(a3)** in the same example.

You can also write conjunctive queries on the fly, such as **a(X), b(X)** (see Section 4.1.6). Built-in comparison operators (listed in Section 4.5.1) can be safely used in queries whenever their arguments are ground at evaluation time (excepting equality, which performs unification). Disjunctive queries are also allowed, too, such as **a(X); b(X)**. Concluding, a query follows the same syntax as rule bodies.

If only a limited number of tuples in the answer are required, one can submit the query as **top(*N*,*Query*)**, where ***N*** is the maximum number of tuples to be returned.

Also, query answers can be sorted with **order_by(Query, [Expr1, …, ExprN], [Ord1, …, OrdN])** or simply **order_by(Query, [Expr1, …, ExprN])**, where **Expr*i*** is an expression and **Ord*i*** can be either **a** (for ascending order) or **d** (for descending order). For an explicit ordering to take effect, default answer ordering must be disabled with **/order_answer off** (answer ordering is enabled by default).

```
DES> /assert t(3,1)
DES> /assert t(2,2)
DES> /assert t(1,3)
DES> /assert t(2,1)
DES> t(X,Y)
{
  t(1,3),
  t(2,1),
  t(2,2),
  t(3,1)
}
Info: 4 tuples computed.
DES> /order_answer off
DES> t(X,Y)
{
  t(3,1),
```

```
  t(2,2),
  t(1,3),
  t(2,1)
}
Info: 4 tuples computed.
DES> order_by(t(X,Y),[X],[d])
Info: Processing:
  answer(X,Y) :-
    order_by(t(X,Y),[X],[d]).
{
  answer(3,1),
  answer(2,2),
  answer(2,1),
  answer(1,3)
}
Info: 4 tuples computed.
DES> order_by(t(X,Y),[X,Y],[d,a])
Info: Processing:
  answer(X,Y) :-
    order_by(t(X,Y),[X,Y],[d,a]).
{
  answer(3,1),
  answer(2,1),
  answer(2,2),
  answer(1,3)
}
Info: 4 tuples computed.
```

Note, however, that ordering affects the result of a computation. The next example shows how, depending on the order criteria, the answer is different:

```
DES> top(1,order_by(t(X,Y),[X],[a]))
Info: Processing:
  answer(X,Y)
in the program context of the exploded query:
  answer(X,Y) :-
    top(1,'$p0'(Y,X)).
  '$p0'(Y,X) :-
    order_by(t(X,Y),[X],[a]).
{
  answer(1,3)
}
Info: 1 tuple computed.
DES> top(1,order_by(t(X,Y),[X],[d]))
Info: Processing:
  answer(X,Y)
in the program context of the exploded query:
  answer(X,Y) :-
    top(1,'$p0'(Y,X)).
  '$p0'(Y,X) :-
    order_by(t(X,Y),[X],[d]).
{
  answer(3,1)
}
Info: 1 tuple computed.
```

4.1.5 Temporary Views

Temporary views allow you to write conjunctive queries on the fly. A temporary view is a rule which is added to the database; its head is considered as a query and executed. Afterwards, the rule is deleted. Temporary views are useful for quickly submitting conjunctive queries. For instance, the view:

```
DES> d(X) :- a(X), not(b(X))
```

computes the set difference between the sets **a** and **b**, provided they have been already defined.

Note that the view is evaluated in the context of the program; so, if you have more rules already defined with the same name and arity of the rule's head, the evaluation of the view will return its meaning under the whole set of rules matching the query. For instance:

```
DES> a(X) :- b(X)
```

computes the set union of the sets **a** and **b**, provided they have been already defined.

4.1.6 Automatic Temporary Views

Automatic temporary views, shortly autoviews, are temporary views which do not need a head and allows you to write conjunctive queries on the fly. When you write a conjunctive query, a new temporary relation, named **answer**, is built with as many arguments as variables occur in the conjunctive query. **answer** is a reserved word and cannot be used for defining any other relation. As an example of an autoview, let’s consider:

```
DES> a(X),b(Y)

Info: Processing:
  answer(X,Y) :-
    a(X),
    b(Y).
{
  answer(a1,a1),
  answer(a1,b1),
  answer(a1,b2),
  answer(a2,a1),
  answer(a2,b1),
  answer(a2,b2),
  answer(a3,a1),
  answer(a3,b1),
  answer(a3,b2)
}
Info: 9 tuples computed.
```

which computes the Cartesian product of the relations **a** and **b**, provided they have been already defined as:

```
  a(a1).
  a(a2).
  a(a3).
```

```
  b(b1).
  b(b2).
  b(a1).
```

4.1.7 Underscored Variables

An underscored variable (a variable starting with the underscore symbol '_') is handled similar to Prolog. It is assumed to be of no interest for the answer, so that they are discarded from the answer should they occur in the body of a query, view or autoview (even in its head). For instance, computing the projection of a relation **t** with respect to its first argument can be simply done as follows:

```
DES> /assert t(1,2)
DES> /assert t(2,3)
DES> t(X,_)
Info: Processing:
  answer(X) :-
    t(X,_).
{
  answer(1),
  answer(2)
}
Info: 2 tuples computed.
```

instead of having to resort to an autoview such as:

```
DES> p(X):-t(X,Y)
Info: Processing:
  p(X) :-
    t(X,Y).
{
  p(1),
  p(2)
}
Info: 2 tuples computed.
```

Also, let's consider other situation, as follows:

```
DES> /duplicates off
DES> t(X,Y)
{
  t(1,1),
  t(1,2),
  t(3,3)
}
Info: 3 tuples computed.
DES> t(X,X)
{
  t(1,1),
  t(3,3)
}
Info: 2 tuples computed.
```

If you use instead underscored variables, you get one answer tuple:

```
DES> t(_X,_X)
Info: Processing:
  answer :-
    t(_X,_X).
{
  answer
}
Info: 1 tuple computed.
```

However, if duplicates are enabled, you get two answer tuples, although the concrete values for the arguments of **t** are not visible:

```
DES> /duplicates on
DES> t(_X,_X)
Info: Processing:
  answer :-
    t(_X,_X).
{
  answer,
  answer
}
Info: 2 tuples computed.
```

4.1.8 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 [Ullm95]. This broadly means that negation is not involved in a recursive computation path, although it can use recursive rules. The following program[3] illustrates this point:

```
a :- not(b).
b :- c,d.
c :- b.
c.
```

The query **a** succeeds with the meaning **{a}**. Observe also that **not(a)** does not succeed, i.e., its meaning is the empty set.

DES provides two different algorithms for computing negation: **strata** (a default algorithm following a bottom-up top-down-guided stratum saturation) and **et_not** (taken from [SD91]), which are selected via the command **/negation *Algorithm***. (cf. Section 5.14.10).

If you are interested in how programs with negation are solved for the algorithm **strata**, you can find useful the following commands (cf. Section 5.14.7):

```
DES> /pdg

Nodes: [d/0,a/0,b/0,c/0]
Arcs : [a/0-b/0,c/0+b/0,b/0+d/0,b/0+c/0]
```

[3] In file **negation.dl**, located at the **examples** distribution directory. Adapted from [RSSWF97].

```
DES> /strata

[(d/0,1),(a/0,2),(b/0,1),(c/0,1)]
```

The first command shows the predicate dependency graph (see, e.g., [ZCF+97]) for the loaded program. First, nodes in the graph are shown in a list whose elements P are predicates with their arities with the form predicate/arity. Next, arcs in the graph are shown in a list whose elementes are either P+Q or P-Q, where P and Q are nodes in the graph. An arc P+Q means that there exists a rule such that P is the predicate for its head, and Q is the predicate for one of its literals. If the literal is negated, the arc is negative, which is expressed as P-Q. The graph for this program can be depicted as in Figure 3.

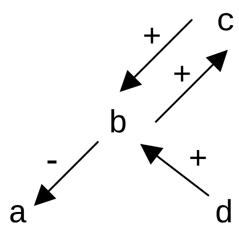


Figure 3. Predicate Dependency Graph for **`negation.dl`**

The second command shows the stratum assigned to each predicate. This assignment is computed by following an algorithm based on [Ullm95], but modified for taking advantage of the predicate dependency graph. Strata are shown as a list of pairs (P,S), where P is a predicate and S is its assigned stratum. In this example, all of the program predicates are in stratum 1 but **a**, which is assigned to stratum 2. This means that if the meaning of **a** is to be computed, then the meanings of predicates in lower strata (and only those predicates **a** depends on) have to be firstly computed.

Since the algorithm **`strata`** does not follow a naïve bottom-up solving, only the meanings of required predicates are computed. To illustrate this, consider the query **b** for the same program. DES computes the predicate dependency subgraph for **b**, i.e., all of the predicates which are reachable from **b**, and, then, a stratification is computed. Notice the different information given by the system for solving the queries **a** and **b** (here, verbose output is currently enabled with the command **`/verbose on`**):

```
DES> a
Info: Computing by stratum of [b].
{
  a
}
Info: 1 tuple computed.
DES> b
{
}
Info: 0 tuples computed.
```

For the goal **a**, the system informs that **b** is previously computed (nevertheless taking advantage of the extension table mechanism), whereas for the goal **b** there is no need of resorting to the stratum-by-stratum solving.

Finally, consult also Section 5.3 for limitations in the use of negation.

4.1.9 Duplicates

Duplicates in answers are removed by default. However, it is also possible to enable them with the command **`/duplicates on`**. This allows to generate answers as multisets instead of as the typical set-oriented deductive systems behave. Computing the meaning of a relation containing duplicates in the extensional database (i.e., its facts) will include all of them in the answer, as in:

```
DES> /duplicates on
DES> /assert t(1)
DES> /assert t(1)
DES> t(X)
{
  t(1),
  t(1)
}
Info: 2 tuples computed.
```

Rules can also be source of duplicates, as in:

```
DES> /assert s(X):-t(X)
DES> s(X)
{
  s(1),
  s(1)
}
Info: 2 tuples computed.
```

In addition, recursive rules are duplicate sources, as in:

```
DES> /assert t(X):-t(X)
DES> t(X)
{
  t(1),
  t(1),
  t(1),
  t(1)
}
Info: 4 tuples computed.
```

where two tuples directly come from the two facts for **`t/1`**, and the other two from the single recursive rule. Again, adding the same recursive rule yields:

```
DES> /assert t(X):-t(X)
DES> t(X)
{
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
```

```
  t(1)
}
Info: 10 tuples computed.
```

where this answer contains the outcome due to: two tuples directly from the two facts, and four tuples for each recursive rule. The first recursive rule is source of four tuples because of the two facts and the two tuples from the second recursive rule. Analogously, the second recursive rule is source of another four tuples: two facts and the two tuples from the first recursive rule.

The rule of thumb to understand duplicates in recursive rules is to consider all possible computation paths in the dependency graph, stopping when a (recursive) node already used in the computation is reached.

It is also possible to discard duplicates for an atom with the metapredicate **`distinct/1`**. For instance, let's consider the following with the same example above:

```
DES> distinct(t(X))
Info: Processing:
  answer(X) :-
    distinct(t(X)).
{
  answer(1)
}
Info: 1 tuple computed.
```

Such query is equivalent to the following SQL statement, provided that metadata is available for the relation **`t`**:

```
DES> :-type(t(a:int))
DES> select distinct * from t
answer(t.a) ->
{
  answer(1)
}
Info: 1 tuple computed.
```

As it would be expected, duplicates are only discarded for the call **`distinct(`*`Atom`*`)`**, but not for other occurrences of ***`Atom`*** during query solving. Thus:

```
DES> t(X),distinct(t(X))
Info: Processing:
  answer(X) :-
    t(X),
    distinct(t(X)).
{
  answer(1),
  answer(1),
  answer(1),
  answer(1),
  answer(1),
  answer(1),
  answer(1),
  answer(1),
  answer(1),
  answer(1)
```

```
}
Info: 10 tuples computed.
```

Compare this to the call:

```
DES> t(X),t(X)
Info: Processing:
  answer(X) :-
    t(X),
    t(X).
{
  answer(1),
  ...
  answer(1)
}
Info: 100 tuples computed.
```

A subset of arguments in an atom can be selected for discarding duplicates. To this end, the metapredicate **distinct/2** is provided. Its first argument is the list of variables for which duplicates are not required, i.e., each concrete assignment of values to all variables in the list must be different. So, let's consider the following session:

```
DES> /listing
t(1,1).
t(1,2).
t(2,1).
Info: 3 rules listed.
DES> distinct([X],t(X,Y))
Info: Processing:
  answer(X) :-
    distinct([X],t(X,Y)).
{
  answer(1),
  answer(2)
}
Info: 2 tuples computed.
```

In addition, discarding duplicates can be performed in the context of aggregates:

```
DES> count(distinct(t(X)),C)
Info: Processing:
  answer(C)
in the program context of the exploded query:
  answer(C) :-
    count('$p0'(X),[],C).
  '$p0'(A) :-
    distinct(t(A)).
{
  answer(1)
}
Info: 1 tuple computed.
```

See also Section 4.1.12 for discarding duplicates in aggregates.

4.1.10 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. The following illustrates this expected behaviour:

```
DES> null=null
{
}
Info: 0 tuples computed.

DES> null\=null
{
}
Info: 0 tuples computed.
```

However, for the same null value, the equality should succeed, as in the conjunctive query: **X=null,X=X**.

A null value is internally represented as **'$NULL'(*ID*)**, where ***ID*** is a unique identifier (an integer). Development listings (enabled via the command **/development on**) allow to inspect these identifiers, such as in:

```
DES> /development on
DES> p(X,Y):-X=null,Y=null,X=Y
Info: Processing:
  p(X,Y) :-
    X = '$NULL'(14),
    Y = '$NULL'(15),
    X = Y.
{
}
Info: 0 tuples computed.
DES> p(X,Y):-X=null,Y=null,X\=Y
Info: Processing:
  p(X,Y) :-
    X = '$NULL'(16),
    Y = '$NULL'(17),
    X \= Y.
{
}
Info: 0 tuples computed.
```

The built-in predicate **is_null/1** tests whether its single argument is a null value:

```
DES> is_null(null)
{
  is_null(null)
}
Info: 1 tuple computed.

DES> X=null,is_null(X)
```

```
Info: Processing:
  answer(X) :-
    X = null,
    is_null(X).
{
  answer(null)
}
Info: 1 tuple computed.
```

Its counterpart predicate is also provided: **`is_not_null/1`**, which is true if its argument is not a null value.

Note that from a system implementor viewpoint, nulls can never unify because they are represented by different ground terms. On the other hand, disequality is explicitly handled in order to fail when comparing nulls.

Evaluation of a given expression including at least one null value always returns the same concrete null value. Thus, two expressions including null values are considered equivalent if they are *syntactically* equal (w.r.t. ground instantiations for null values in particular). For instance, **`X=null,X+1=X+1`** succeeds, whereas **`X=null,Y=null,X+1=Y+1`** and **`X=null,X+1=1+X`** do not.

4.1.11 Outer Joins

Three outer join operations are provided (cf. Section 4.5.6), following relational database query languages (SQL, extended relational algebra): left, right and full outer join. Having loaded the example program **`relop.dl`**, we can submit the following queries:

```
DES> /c relop
DES> /listing a
a(a1).
a(a2).
a(a3).
DES> /listing b
b(a1).
b(b1).
b(b2).
DES> lj(a(X),b(Y),X=Y)
Info: Processing:
  answer(X,Y) :-
    lj(a(X),b(Y),X = Y).
{
  answer(a1,a1),
  answer(a2,null),
  answer(a3,null)
}
Info: 3 tuples computed.
DES> rj(a(X),b(Y),X=Y)
Info: Processing:
  answer(X,Y) :-
    rj(a(X),b(Y),X = Y).
{
  answer(a1,a1),
```

```
  answer(null,b1),
  answer(null,b2)
}
Info: 3 tuples computed.
DES> fj(a(X),b(Y),X=Y)
Info: Processing:
  answer(X,Y) :-
    fj(a(X),b(Y),X = Y).
{
  answer(a1,a1),
  answer(a1,null),
  answer(a2,null),
  answer(a3,null),
  answer(null,a1),
  answer(null,b1),
  answer(null,b2)
}
Info: 7 tuples computed.
```

Note that the third parameter is the join condition. Be aware and do not miss a where condition with a join condition. Let´s consider the above query **lj(a(X),b(Y),X=Y)**. Do not expect the same result as above for the following query:

```
DES> lj(a(X),b(X),true)
Info: Processing:
  answer(X) :-
    lj(a(X),b(X),true).
{
  answer(a1)
}
Info: 1 tuple computed.
```

Here, the same variable **X** for the relations **a** and **b** means that tuples from **a** and **b** with the same value are to be joined, as in the next equivalent query:

```
DES> lj(a(X),b(Y),true),X=Y
Info: Processing:
  answer(X,Y) :-
    lj(a(X),b(Y),true),
    X = Y.
{
  answer(a1,a1)
}
Info: 1 tuple computed.
```

Outer join relations can be nested as well:

```
DES> lj(a(X),rj(b(Y),c(U,V),Y=U),X=Y)
Info: Processing:
  answer(X,Y,U,V) :-
    lj(a(X),rj(b(Y),c(U,V),Y = U),X = Y).
{
  answer(a1,a1,a1,a1),
  answer(a1,a1,a1,b2),
  answer(a2,null,null,null),
```

```
  answer(a3,null,null,null)
}
Info: 4 tuples computed.
```

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

```
DES> lj(a(X),c(U,V),(X>U;X>V))
Info: Processing:
  answer(X,U,V)
in the program context of the exploded query:
  answer(X,U,V) :-
    lj(a(X),c(U,V),(X > U;X > V)).
{
  answer(a1,null,null),
  answer(a2,a1,a1),
  answer(a2,a1,b2),
  answer(a3,a1,a1),
  answer(a3,a1,b2),
  answer(a3,a2,b2)
}
Info: 6 tuples computed.
```

4.1.12 Aggregates

Aggregates refer to functions and predicates that compute values with respect to a collection of values instead of a single value. Aggregates are provided by means of five usual computations: **sum** (cumulative sum), **count** (element count), **avg** (average), **min** (minimum element), and **max** (maximum element). In addition, the less usual **times** (cumulative product) is also provided. They behave close to most SQL implementations, i.e., ignoring nulls. Duplicate-free counterparts are also provided: **sum_distinct**, **count_distinct**, **avg_distinct**, and **times_distinct**. Note that for minimum and maximum, no counterparts are provided since they would compute the same results.

4.1.12.1 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 occur.

4.1.12.2 Group_by Predicate

A **group_by** predicate encloses a query for which a given list of variables builds answer sets (groups) for all possible values of these variables. Let's consider the following excerpt from the file **aggregates.dl**:

```
% employee(Name,Department,Salary)
employee(anderson,accounting,1200).
employee(andrews,accounting,1200).
employee(arlingon,accounting,1000).
employee(nolan,null,null).
employee(norton,null,null).
employee(randall,resources,800).
employee(sanders,sales,null).
```

```
employee(silver,sales,1000).
employee(smith,sales,1000).
employee(steel,sales,1020).
employee(sullivan,sales,null).
```

We can count the number of employees for each department with the following query:

```
DES> group_by(employee(N,D,S),[D],R=count)
Info: Processing:
  answer(D,R) :-
    group_by(employee(N,D,S),[D],R = count).
{
  answer(accounting,3),
  answer(null,2),
  answer(resources,1),
  answer(sales,5)
}
Info: 4 tuples computed.
```

Note that two employees are not assigned to any department yet (**nolan** and **norton**). This query behaves as an SQL user would expect, though nulls do not have to represent the same data value (in spite of this, such tuples are collected in the same bag).

If we rather want to count *active* employees (those with assigned salaries), we pose the following query:

```
DES> group_by(employee(N,D,S),[D],R=count(S))
Info: Processing:
  answer(D,R) :-
    group_by(employee(N,D,S),[D],R = count(S)).
{
  answer(accounting,3),
  answer(null,0),
  answer(resources,1),
  answer(sales,3)
}
Info: 4 tuples computed.
```

Note that null departments have no employee with assigned salary.

Counting the number of departments from the relation **employee** needs to discard duplicates, as in:

```
DES> count_distinct(employee(N,D,S),D,T).
Info: Processing:
  answer(T) :-
    count_distinct(employee(N,D,S),D,[],T).
{
  answer(3)
}
Info: 1 tuple computed.
```

Conditions including aggregates on groups can be stated as well (cf. having conditions in SQL). For instance, the following query counts the active employees of departments with more than one employee.

```
DES> group_by(employee(N,D,S),[D],count(S)>1)
Info: Processing:
  answer(D) :-
    group_by(employee(N,D,S),[D],(A = count(S),A > 1)).
{
  answer(accounting),
  answer(sales)
}
Info: 2 tuples computed.
```

Note that the number of employees can also be returned, as follows:

```
DES> group_by(employee(N,D,S),[D],(R=count(S),R>1))
Info: Processing:
  answer(D,R) :-
    group_by(employee(N,D,S),[D],(R = count(S),R > 1)).
{
  answer(accounting,3),
  answer(sales,3)
}
Info: 2 tuples computed.
```

Conditions including no aggregates on tuples of the input relation (cf. SQL **FROM** clause) can also be used (cf. **WHERE** conditions in SQL). For instance, the following query computes the number of employees whose salary is greater than 1,000.

```
DES> group_by((employee(N,D,S),S>1000),[D],R=count(S))
Info: Processing:
  answer(D,R)
in the program context of the exploded query:
  answer(D,R) :-
    group_by('$p2'(S,D,N),[D],R = count(S)).
  '$p2'(S,D,N) :-
    employee(N,D,S),
    S > 1000.
{
  answer(accounting,2),
  answer(sales,1)
}
Info: 2 tuples computed.
```

Note that the following query is not equivalent to the former, since variables in the input relation are not bound after a grouping computation. The following query illustrates this situation, which generates a syntax error.

```
DES> group_by(employee(N,D,S),[D],R=count(S)), S>1000
Error: Incorrect use of shared set variables in metapredicate:
  [N,S]
```

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

```
DES> /assert p(1,1)
DES> /assert p(2,2)
DES> /assert q(X,C):-group_by(p(X,Y),[X],(C=count;C=sum(Y)))
DES> q(X,C)
Info: Computing by stratum of [p(A,B)].
{
  q(1,1),
  q(2,1),
  q(2,2)
}
Info: 3 tuples computed.
```

An analogous SQL session follows:

```
DES-SQL> create table p(X int, Y int)
DES-SQL> 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)
DES-SQL> select * from q
answer(q.X, q.C) ->
{
  answer(1,1),
  answer(2,1),
  answer(2,2)
}
Info: 3 tuples computed.
```

4.1.12.3 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. Again, with the same file, the following queries are allowed:

```
DES> count(employee(N,D,S),S,T)
Info: Processing:
  answer(T) :-
    count(employee(N,D,S),S,[],T).
{
  answer(7)
}
Info: 1 tuple computed.
```

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:

```
DES> c(D,C):-count(employee(N,D,S),S,C)
Info: Processing:
  c(D,C) :-
    count(employee(N,D,S),S,[D],C).
{
  c(accounting,3),
  c(null,0),
```

```
  c(resources,1),
  c(sales,3)
}
Info: 4 tuples computed.
```

Note that the system adds to the aggregate predicate an argument with the list of grouping variables, which are the ones occurring in the first argument of the aggregate predicate that also occur in the head. This code translation is required for the aggregate predicate to be compute, although such form has not been made available to the user.

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:

```
DES> count((employee(N,D,S),avg(employee(N1,D1,S1),S1,A),S>A),C)
Info: Processing:
  answer(C)
in the program context of the exploded query:
  answer(C) :-
    count('$p2'(A,S,D,N),[],C).
  '$p2'(A,S,D,N) :-
    employee(N,D,S),
    avg(employee(N1,D1,S1),S1,[],A),
    S > A.
{
  answer(2)
}
Info: 1 tuple computed.
```

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:

```
DES> count((employee(N,D,S),avg(employee(N1,D,S1),S1,A),S>A),C)
Info: Processing:
  answer(C)
in the program context of the exploded query:
  answer(C) :-
    count('$p2'(A,S,N),[],C).
  '$p2'(A,S,N) :-
    employee(N,D,S),
    avg(employee(N1,D,S1),S1,[],A),
    S > A.
{
  answer(3)
}
Info: 1 tuple computed.
```

Also, as a restriction of the current implementation, keep in mind that having conditions including aggregates (as the one including the average computations above) can only occur in the first argument of an aggregate. The following query, which should be equivalent to the last one, would generate a run-time exception:

```
DES> v(D):-
avg(employee(N1,D,S1),S1,A),count((employee(N,D,S),S>A),C)
Error: S > A will raise a computing exception at run-time.
Warning: This view is unsafe because of variable(s):
  [A]
```

Finally, recall that expressions including aggregate functions are not allowed in conjunction with aggregate predicates, but only in the context of a **group_by** predicate.

4.1.13 Disjunctive Bodies

As introduced in Section 4.1.1, rule bodies can contain disjunctions, such as the one contained in the program **family.dl**:

```
parent(X,Y) :-
  father(X,Y)
  ;
  mother(X,Y).
```

This clause is equivalent to:

```
parent(X,Y) :-
  father(X,Y).
parent(X,Y) :-
  mother(X,Y).
```

If you list the database contents via the command **/listing** you will get the first form when development listings are off (via the command **/development off**). Otherwise, you get the second one (command **/development on**).

Datalog views and autoviews containing disjunctive bodies are allowed, and the system informs about the program transformation needed to compute them. For instance, you can directly submit the rule above as a view at the DES prompt:

```
DES> parent(X,Y) :- father(X,Y) ; mother(X,Y)
Info: Processing:
  parent(X,Y)
in the program context of the exploded query:
  parent(X,Y) :-
    father(X,Y).
  parent(X,Y) :-
    mother(X,Y).
{
  parent(amy,fred),
  parent(carolI,carolII),
  parent(carolII,carolIII),
  parent(fred,carolIII),
  parent(grace,amy),
  parent(jack,fred),
  parent(tom,amy),
  parent(tony,carolII)
```

```
}
Info: 8 tuples computed.
```

4.1.14 Relational Division in Datalog

The provided relational division operation for Datalog follows the original proposal of Codd [Codd72] but, instead of comparing schemas based on column names, we compare schemas based on variable names. Given a left operand L and a right operand R in a division operator, the result is a relation with as many arguments as variables are in vars(L)-vars(R), where vars(R)⊂vars(L) and vars(T) returns the variables in a term T.

For example, given the database:

```
t(1,1).
t(1,2).
t(2,1).
s(1).
s(2).
```

Then, the query:

```
t(X,Y) division s(Y)
```

returns:

```
{answer(1)}
```

Now, let's consider that the relations to be divided contain other arguments that are not relevant for the division operator. For instance, let's consider the relation **`work(employee, project, hours)`**, under an intuitive meaning. If we want to know the name and department of each employee who is working on each project on which employee **`smith`** is working, we have to project the division operands for the appropriate arguments. For instance:

```
DES> /assert np_work(N,P) :- work(N,P,_)
DES> np_work(N,P) division np_work(smith,P)
```

However, by using anonymous variables, it is possible to define the relevant variables for the division operator: All non-relevant variables can be discarded by using anonymous variables instead. Following the same example, the same query can be submitted as simply as:

```
DES> work(N,P,_) division work(smith,P,_)
```

4.1.15 Integrity Constraints

Integrity constraints allow to specify valid values for tuples in relations. DES provides several predefined constraints stemmed from SQL: type, primary key and foreign key. In addition, a predefined functional integrity constraint is also provided. Users can also define its own integrity constraints, which are called user-defined integrity constraints from now on. All of them can be declared and the system monitors their fulfilment, which is the default behaviour. However, the command **`/check off`** allows to disable constraint checking. All predefined integrity constraints apply to facts, but type constraints, which also apply to rules. Also, user-defined constraints apply to facts and rules.

A comma-separated sequence of predefined integrity constraints is allowed to specify multiple constraints in a single input.

4.1.15.1 Type

A type constraint specifies the values in a domain a predicate argument (table column in relational jargon) may take. An example of type constraint declaration at the command prompt is as follows:

```
DES> :- type(p,[int,string])
```

This is equivalent to the following alternative syntax:

```
DES> :- type(p(int,string))
```

Allowed types include the following (where each row in the first column contains type synonyms):

`varchar` `string`	String of unbounded length
`char(N)` `varchar(N)`	String with length up to **N**
`char`	String with length 1
`integer` `int`	Integer number
`float` `real`	Real number

Precision and range depend on the underlying Prolog system.

Subsequent type declarations are allowed for the same predicate and arity; the last declaration is the one to persist, overriding previous type declarations for such predicate. The following session is possible, and thus the second declaration persists:

```
DES> :- type(p,[string,string])
DES> :- type(p,[int,int])
```

As well, columns can be given names:

```
DES> :- type(p,[a:int,b:string])
```

which is equivalent to the following alternative syntax:

```
DES> :- type(p(a:int,b:string))
```

However, a type declaration for a relation already typed with a different arity is not allowed. As will be seen in further sections, SQL statements can refer to Datalog relations, and SQL does not allow relations of the same name and different arities.

```
DES> :- type(p,[a:int])
Error: Cannot add types to a relation with several arities.
       Relation: p
```

A Datalog type declaration is analogous to the creation of an SQL table, with the same outcome (defining metadata for a relation: relation name, column names and types).

```
DES> /dbschema p
Info: Table:
 * p(a:number(integer),b:string(varchar))

DES> drop table p

DES> /dbschema p
Info: No table or view found with that name.
DES> create table p(a int, b string)

DES> /dbschema p
Info: Table:
 * p(a:number(integer),b:string(varchar))
```

It is also possible to omit column names. In this case, they are automatically provided (with names '**$1**','**$2**', and so on).

```
DES> :- type(p,[int,string])

DES> /dbschema p
Info: Table:
 * p($1:number(integer),$2:string(varchar))
```

Let's consider the following session, where it can be seen that the system monitors type constraints in both Datalog and SQL queries:

```
DES> :-type(p,[int,string])
DES> /assert p(a,b)
Error: Type mismatch p.$1:number(integer) vs.
string(char(_6372)).
       p($1:number(integer),$2:string(varchar))
DES> /assert p(1,a)
DES> p(X,Y)
{
  p(1,a)
}
Info: 1 tuple computed.
DES> select * from p
answer(p.$1, p.$2) ->
{
  answer(1,a)
}
Info: 1 tuple computed.
DES> insert into p values('a','b')
Error: Type mismatch p.$1:number(integer) vs.
string(char(_6937)).
       p($1:number(integer),$2:string(varchar))
```

```
Info: 0 tuples inserted.
```

Note that columns with automatically given names can be accessed from an SQL statement, but enclosed as special user identifiers. ISO delimiters (double quotes "", supported by Oracle and SQL Server) are supported as well as other vendor-specific delimiters: MS Access (square brackets []) and MySQL (back quotes ``). Otherwise, an error is raised:

```
DES>  select $1 from p
Error: Input processing error.

DES> select "$1" from p
answer(p.$1) ->
{
  answer(1)
}
Info: 1 tuple computed.
```

A relation already defined is checked for consistency when trying to assert a new type constraint:

```
DES> /assert t(1)
DES> /assert t(a)
DES> :-type(t,[int])
Error: No type tuple covers all the loaded rules for t/1:
       t(1).
       t(a).
Info: 2 rules listed.
```

Should any other constraint remains asserted (other than a type constraint), a type constraint cannot be changed:

```
DES> :-type(p,[a:int,b:string])
Error: Cannot change type assertion while other constraints
remain.
```

4.1.15.1.1 Types on Intensional Database

Types can also be declared for predicates of the intensional database, i.e., those predicates defined at least with rules, not only with facts. So, asserting a new type constraint over an intensional relation will trigger type checking, inferring types along the predicate dependency graph restricted to the typed predicate. Let's consider the following situation as an example:

```
DES> /listing
s(a).
t(1).
t(X) :-
  s(X).
Info: 3 rules listed.

DES> :-type(t,[int])
Error: No type tuple covers all the loaded rules for t/1:
       t(1).
       t(X) :-
         s(X).
```

```
Info: 2 rules listed.
```

4.1.15.1.2 Types on Propositional Relations

Finally, propositional relations are also subject of beign typed, of course with an empty list of arguments:

```
DES> :-type(a,[])
DES> /dbschema a
Info: Table:
 * a
```

The alternative syntax becomes shorter in this case indeed:

```
DES> :-type(a)
```

4.1.15.2 Nullability (Existency Constraint)

Columns can be imposed to contain a concrete value rather than a null. The next system session shows an example:

```
DES> :-type(p,[a:int,b:string])
DES> :-nn(p,[a])
```

The list of column names specifies the columns for which null values are not allowed. Thus, trying to assert a tuple such as the following, will raise an error:

```
DES> /assert p(null,'')
Error: Not null violation p.[a]
```

Subsequent existency constraints are allowed for the same predicate and arity; the last declaration is the one to persist, overriding previous declarations for such predicate.

4.1.15.3 Primary Key

A primary key constraint specifies that no two tuples have the same values for a given set of columns. Next, a system session illustrates the use of a primary key assertion:

```
DES> :-type(p,[a:int,b:string])
DES> :-pk(p,[a])
```

Primary key constraints are trivially satisfied when duplicates are disabled, as relations are considered as sets, irrespective of the current database instance, that may contain duplicates for the arguments in the primary key.

Several primary key declarations are allowed for the same predicate and arity; the last declaration is the one to persist, overriding previous type declarations for such predicate:

```
DES> :-pk(p,[a])
DES> :-pk(p,[c])
Error: Unknown column c.
DES> :-pk(p,[a,a])
```

A relation already defined with facts or rules is checked for consistency when trying to assert a new primary key constraint:

```
DES> :-type(q,[a:int,b:int])
DES> /assert q(1,1)
DES> /assert q(2,2)
DES> /assert q(1,2)
DES> :-pk(q,[a])
Error: Primary key violation q.[a]
       Offending values in database: [pk(1)]
Info: Constraint has not been asserted.
```

4.1.15.4 Candidate Key (Uniqueness Constraint)

As a primary key, a candidate key constraint specifies that no two tuples have the same values for a given set of columns. Next, a system session illustrates the use of a candidate key assertion:

```
DES> :-type(p,[a:int,b:string])
DES> :-ck(p,[a])
```

Candidate key constraints are trivially satisfied when duplicates are disabled, as relations are considered as sets, irrespective of the current database instance, that may contain duplicates for the arguments in the candidate key.

Several candidate key declarations are allowed for the same predicate and arity. By contrast to primary keys, several candidate key constraints are allowed for the same predicate:

```
DES> :-ck(p,[b])
DES> :-ck(p,[a,b])
DES> /dbschema p
Info: Table:
 * p(a:number(integer),b:string(varchar))
    - NN: [a]
    - CK: [a]
    - CK: [b]
    - CK: [a,b]
```

4.1.15.5 Foreign Key

A foreign key constraint specifies that the values in a given set of columns of a relation must exist already in the columns declared in the primary key constraint of another relation. Next, an example of a foreign key assertion is shown:

```
DES> :-type(p(a:int)),type(q(b:int)),pk(q,[b])
DES> :-fk(p,[a],q,[b])
```

However, if the relations do not exist, an error is raised:

```
DES> :-fk(p,[a],q,[b])
Error: Relation p has not been typed yet.
DES> :-type(p,[a:int]), type(q,[b:int])
```

Trying to impose a foreign key with a referenced table which does not have a primary key for matching columns raises an error:

```
DES> :-fk(p,[a],q,[b])
Error: Referenced column list q.[b] is not a primary key.
DES> :-pk(q,[b])
DES> :-fk(p,[a],q,[b])
```

The same constraint cannot be reasserted:

```
DES> :-fk(p,[a],q,[b])
Error: Trying to reassert an existing constraint.
DES> /dbschema
Info: Table(s):
 * p(a:number(integer))
    - FK: p.[a] -> q.[b]
 * q(b:number(integer))
    - PK: [b]
Info: No views.
DES> /assert p(1)
Error: Foreign key violation p.[a]->q.[b]
       when trying to insert: p(1)
DES> /assert q(1)
DES> /assert p(1)
DES> /listing
p(1).
q(1).
Info: 2 rules listed.
```

Several foreign keys may exist for the same relation:

```
DES> :-type(p,[a:int])
DES> :-type(q,[b:int])
DES> :-type(r,[a:int,b:int,c:string])
DES> :-pk(p,[a]), pk(q,[b])
DES> :-fk(r,[a],p,[a]), fk(r,[b],q,[b])
DES> /dbschema r
Info: Table:
 * r(a:number(integer),b:number(integer),c:string(varchar))
    - FK: r.[a] -> p.[a]
    - FK: r.[b] -> q.[b]
```

Referenced columns have to match the types of foreign key columns, otherwise an error is raised:

```
DES> :-fk(r,[c],q,[b])
Error: Type mismatch r.c:string(varchar) <> q.b:number(integer)
```

A relation already defined with facts or rules is checked for consistency when trying to assert a new foreign key constraint:

```
DES> :-type(p,[a:int])
DES> :-type(q,[a:int])
DES> /assert p(1)
DES> :-pk(q,[a])
DES> :-fk(p,[a],q,[a])

Error: Foreign key violation p.[a]->q.[a]
       Offending values in database: [fk(1)]
```

```
Info: Constraint has not been asserted.
```

4.1.15.6 Functional Dependency

A functional dependency constraint specifies that, given a set of attributes A_1 of a relation R, they functionally determine another set A_2, i.e., each tuple of values of A_1 in R is associated with precisely one tuple of values A_2 in the same tuple of R.

```
DES> :-fd(p,[a],[c])
Error: Relation p has not been typed yet.
DES> :-type(p,[a:int,b:int])
DES> :-fd(p,[a],[c])
Error: Unknown column c.
DES> :-fd(p,[a],[b])
DES> /dbschema p
Info: Table:
 * p(a:number(integer),b:number(integer))
    - FD: [a] -> [b]
```

By asserting the fact **p(1,2)**, it must hold that any other tuple with 1 in its first attribute must have the value 2 in its second attribute.

```
DES> /assert p(1,2)
DES> /assert p(1,3)
Error: Functional dependency violation p.[a]->p.[b]
       in table p(a,b)
       when trying to insert: p(1,3)
       Witness tuple        : p(1,2)
```

Several functional dependency constraints can be imposed on a given relation. They can be deleted either with the command **drop_ic** or when an SQL **DROP TABLE** or **DROP DATABASE** statements are issued.

Trivial functional dependencies are rejected:

```
DES> :-fd(p,[a],[a])
Warning: Trivial functional dependency. Not asserted.
```

A relation already defined with facts or rules is checked for consistency when trying to assert a new functional dependency constraint:

```
DES> :-type(p,[a:int,b:int,c:int])
DES> /assert p(1,1,1)
DES> /assert p(1,2,3)
DES> :-fd(p,[a],[c])
Error: Functional dependency violation p.[a]->p.[c]
       Offending values in database: [fd(1,1,1),fd(1,2,3)]
Info: Constraint has not been asserted.
```

4.1.15.7 User-defined Integrity Constraints

Users can also define their own integrity constraints. A user-defined integrity constraint is represented with a rule without head. The rule body is an assertion that specifies inconsistent data, i.e., should this body can be proved, an inconsistency is detected and reported to the user.

Declaring such integrity constraints implies to change your mind w.r.t. usual consistency constraints as domain constraints in SQL. For instance, to specify that a column **c** of a table **t** can take values between two integers one can use the SQL clause **CHECK** in the creation of the table as follows[4]:

```
CREATE TABLE t(c INT CHECK (c BETWEEN 0 AND 10));
```

In contrast, in Datalog you can submit the following constraints:

```
DES> :-type(t,[c:int])
DES> :-t(X),(X<0;X>10)
```

Notice that the rule body succeeds for values in **t** out of the interval [**0**,**10**]. So, an integrity constraint specifies *unfeasible* values rather than feasible. Also note that whilst several predefined constraints are allowed in a constraint, only one user-defined integrity constraint is allowed. A couple of assertions to show the behaviour of the above example follow:

```
DES> /assert t(0)
DES> /assert t(11)
Error: Integrity constraint violation.
       ic(X) :-
         t(X),
         X < 0
         ;
         X > 10.
       Offending values in database: [ic(11)]
```

Note that to be able to interpret that offending values, the integrity constraint is shown as a rule defining a new predicate **ic**, where the rule's head has as many variables as relevant variables in the constraint. Then, offending values are encapsulated in the meaning of the constraint relation **ic**.

A rule body of a constraint is any valid rule body, i.e., goals in constrainsts can refer to other user-defined or built-in predicates as well, including negation, aggregates, etc. Let's consider the following session, in which we are interested in specifying a directed tree (a connected graph with no cycles):

```
DES> /verbose on
Info: Verbose output is on.
DES> /consult paths
Info: Consulting paths...
  edge(a,b).
  edge(a,c).
  edge(b,a).
  edge(b,d).
  path(X,Y) :-
    path(X,Z),
    edge(Z,Y).
  path(X,Y) :-
    edge(X,Y).
```

[4] This **CHECK** SQL clause is not yet supported by DES.

```
  end_of_file.
Info: 6 rules consulted.
Info: Computing predicate dependency graph...
Info: Computing strata...
DES> :-path(X,X)
Info: Parsing query...
Info: Constraint successfully parsed.
Info: Checking user-defined integrity constraint over database.
       :-
         path(X,X).
Info: Computing predicate dependency graph...
Info: Computing strata...
Error: Integrity constraint violation.
       ic(X) :-
         path(X,X).
       Offending values in database: [ic(b),ic(a)]
Info: Constraint has not been asserted.
```

The constraint **`:-path(X,X)`** specifies that a path from a node to itself is not allowed. As the consulted program contains a cycle involving nodes **`a`** and **`b`**, the constraint is violated and therefore it is not asserted. Offending values are listed (in this case, all the values involved in any cycle; you can try out other edges and see the outcome).

Another use is to first specify the constraint and then a graph. However, don't be tempted to submit the constraint and consult the program: the constraint will be removed since consulting a program amounts to erase the existing database, including user-defined integrity constraints. Instead, use the **`reconsult`** command:

```
DES> /verbose on
Info: Verbose output is on.
DES> /cd examples
Info: Current directory is:
  c:/fernan/research/bddeduc/des/des3.3.1/examples/
DES> :-path(X,X)
Info: Parsing query...
Info: Constraint successfully parsed.
Info: Checking user-defined integrity constraint over database.
       :-
         path(X,X).
Info: Computing predicate dependency graph...
Warning: Undefined predicate(s): [path/2]
Info: Computing strata...
DES> /reconsult paths
Info: Consulting paths...
  edge(a,b).
  edge(a,c).
  edge(b,a).
  edge(b,d).
Info: Checking user-defined integrity constraint over database.
       :-
         path(X,X).
Info: Computing predicate dependency graph...
Info: Computing strata...
```

```
  path(X,Y) :-
    path(X,Z),
    edge(Z,Y).
Info: Checking user-defined integrity constraint over database.
       :-
         path(X,X).
Info: Computing predicate dependency graph...
Info: Computing strata...
Error: Integrity constraint violation.
       ic(X) :-
         path(X,X).
       Offending values in database: [ic(b),ic(a)]
  path(X,Y) :-
    edge(X,Y).
           File :
c:/fernan/research/bddeduc/des/des3.3.1/examples/paths.dl
           Lines: 10,10
  end_of_file.
Info: 5 rules consulted.
Info: Computing predicate dependency graph...
Info: Computing strata...
```

Note that the first rule for **path** is not rejected since in the already consulted program it is still consistent w.r.t. to the constraint. However, trying to add the second rule for **path** makes it infeasible, so that it is rejected. Now, only 5 rules have been asserted. If the file was not included the third fact for **edge**, then it would be accepted as a valid tree. Again, trying to insert such a tuple, after such a program is consulted, raises an error:

```
DES> /assert edge(d,a)
Info: Checking user-defined integrity constraint over database.
       :-
         path(X,X).
Info: Computing predicate dependency graph...
Info: Computing strata...
Error: Integrity constraint violation.
       ic(X) :-
         path(X,X).
       Offending values in database: [ic(a),ic(b),ic(d)]
```

Observe that since the **path** relation is now complete, all the nodes in the cycle are displayed (**a**, **b**, and **c**).

The considered constraint is not yet enough to ensure a directed tree defined by **edge** facts. Two conditions remain: First, a given node cannot have more than one incoming edge, and, second, a tree must be a connected graph. If the first condition is imposed, it suffices for the second to check that the number of nodes is the number of edges plus 1. So:

```
DES> /assert node(N):-edge(N,A);edge(A,N)
Info: Computing predicate dependency graph...
Info: Computing strata...
Info: Rule asserted.
DES> :-count(edge(A,B),Es), count(node(N),Ns), D is Ns-Es, D\=1.
Info: Parsing query...
```

```
Info: Constraint successfully parsed.
Info: Computing predicate dependency graph...
Info: Computing strata...
Info: Checking user-defined integrity constraint over database.
       :-
         count(edge(A,B),Es),
         count(node(N),Ns),
         D is Ns - Es,
         D \= 1.
Info: Computing by stratum of [edge(A,B),node(A)].
Info: Computing predicate dependency graph...
Info: Computing strata...
DES> /assert edge(e,f) % An unconnected component
Info: Checking user-defined integrity constraint over database.
       :-
         count(edge(A,B),Es),
         count(node(N),Ns),
         D is Ns - Es,
         D \= 1.
Info: Computing by stratum of [edge(A,B),node(A)].
Info: Computing predicate dependency graph...
Info: Computing strata...
Error: Integrity constraint violation.
       ic(Es,Ns,D) :-
         count(edge(A,B),Es),
         count(node(N),Ns),
         D is Ns - Es,
         D \= 1.
       Offending values in database: [ic(4,6,2)]
```

User-defined integrity constraints are dropped when abolishing the database or consulting a file.

4.1.15.8 Dropping Constraints

Any predefined or user-defined integrity constraint can be dropped with the command **`/drop_ic`** (see Section 5.14.1) followed by the constraint to be dropped with the same syntax as its declaration.

4.1.15.9 Caveats

Either by consulting a program, or by dropping the current database, or by abolishing the database, all integrity constraints are removed, including SQL table and view definitions.

As rules are not checked for predefined constraints, a situations like the following may occur:

```
DES> create table t(a int primary key)
DES> insert into t values (1)
Info: 1 tuple inserted.
DES> /assert t(X):-X=1
DES> /duplicates on
DES> t(X)
{
  t(1),
```

```
  t(1)
}
Info: 2 tuples computed.
```

Nonetheless, if you also want to monitor rules, you can otherwise use a user-defined constraint such as:

```
DES> create table t(a int)
DES> insert into t values (1)
Info: 1 tuple inserted.
DES> :-group_by(t(X),[X],C=count(X),C>1),C>1
DES> /assert t(X):-X=1
Error: Integrity constraint violation.
       ic(X,C) :-
         group_by(t(X),[X],(C = count(X),C > 1)),
         C > 1.
       Offending values in database: [ic(1,2)]
Error: Asserting rules due to integrity constraint violation.
```

4.1.16 Hypothetical Queries

Hypothetical queries are a common need in several scenarios, related mainly with business intelligence applications and the like. They are also known as "what-if" queries and help managers to take decisions on scenarios which are somewhat changed with respect to a current state. Such queries are used, for instance, for deciding which resources must be added, changed or removed to optimize some criterium (cost function - also well related to optimization technologies). Hypothetical queries in the database arena are typically used for assumptions w.r.t. a current database instance.

DES includes one form of hypothetical Datalog queries which may serve to answer several questions. The syntax of an hypothetical query is as follows:

```
Rule1 /\ ... /\ RuleN => Goal
```

which means that, assuming that the current database is augmented with the rules **`Rule1`**, ..., **`RuleN`**, then **`Goal`** is computed with respect to the current database which is augmented with these rules, which must be safe (see Section 5.3). Such query is also understand as a literal in the context of a rule, so that any rule can contain hypothetical goals, as in **`a :- b => c`**. In turn, any **`Rule`*`i`*** can contain hypothetical goals. Variables in **`Rule`*`i`*** are local to **`Rule`*`i`*** (i.e., they are neither shared with other rules nor the goal). Moreover, a hypothetical literal does neither share variables with other literals nor the head of the rule in which it occurs.

Borrowing an example from [Bon90][5], we consider an extended and adapted rule-based system for describing university policy: **`student(S)`** means that **`S`** is a student, **`course(C)`** that **`C`** is a course, **`take(S,C)`** that student **`S`** takes course **`C`**, and **`grad(S)`** that **`S`** is eligible for graduation. The extensional database can contain facts as:

[5] However, note that our approach differs from [Bon90] in at least the following: we allow for rules in the assumption (not only facts), an assumed fact should not be unsafe, and we do not allow assuming negative information (yet!)

```
student(adam).
student(bob).
student(pete).
student(scott).
student(tony).

course(eng).
course(his).
course(lp).

take(adam,eng).
take(pete,his).
take(pete,eng).
take(scott,his).
take(scott,lp).
take(tony,his).
```

The intensional database can contain rules as:

```
grad(S) :- take(S,his), take(S,eng).
```

A regular query for students that would be eligible to graduate is:

```
DES> grad(S)
{
  grad(pete)
}
Info: 1 tuple computed.
```

A first hypothetical query for this database asks "If Tony took **eng**, would he be eligible to graduate?", which can be queried with:

```
DES> take(tony,eng) => grad(tony)
Info: Processing:
  answer :-
    take(tony,eng)=>grad(tony).
{
  answer
}
Info: 1 tuple computed.
```

More than one assumption can be simultaneously stated, as in: "If Tony took **eng**, and Adam took **his**, what are the students that are eligible to graduate?"

```
DES> take(tony,eng) /\ take(adam,his) => grad(S)
Info: Processing:
  answer(S) :-
    take(tony,eng)/\take(adam,his)=>grad(S).
{
  answer(adam),
  answer(pete),
  answer(tony)
}
Info: 3 tuples computed.
```

Another query is "Which are the students which would be eligible to graduate if **his** and **lp** were enough to get it?":

```
DES> (grad(S) :- take(S,his), take(S,lp)) => grad(S)
Info: Processing:
  answer(S) :-
    (grad(S):-take(S,his),take(S,lp))=>grad(S).
{
  answer(pete),
  answer(scott)
}
Info: 2 tuples computed.
```

Note that, although **S** occurs in both the antecedent and the consequent, they are not actually shared, and they simply act as different variables.

Considering also information about course prerequisites as:

```
pre(eng,lp).
pre(hist,eng).
pre(Pre,Post) :-
  pre(Pre,X),
  pre(X,Post).
```

One might wonder whether adding a new prerequisite implies a cycle (so that students cannot fulfil prerequisites at all for the courses in a cycle):

```
DES> pre(lp,hist)=>pre(X,X)
Info: Processing:
  answer(X) :-
    pre(lp,hist)=>pre(X,X).
{
  answer(eng),
  answer(hist),
  answer(lp)
}
Info: 3 tuples computed.
```

The answer includes those nodes in the graph that are in a cycle.

4.1.16.1 Hypothetical Queries and Integrity Constraints

Assumptions can be used in combination with any of the features of DES; in particular, integrity constraints. Following the previous example, you can even express it with the aid of integrity constraints. Avoiding cycles can be forced by:

```
DES> :-pre(X,X)
```

Then, if you want to list prerequisites assuming **pre(lp,hist)** as before:

```
DES> pre(lp,hist)=>pre(X,Y)
Info: Processing:
  answer(X,Y) :-
    pre(lp,hist)=>pre(X,Y).
Error: Integrity constraint violation.
       ic(X) :-
         pre(X,X).
```

```
       Offending values in database: [ic(lp),ic(eng),ic(hist)]
Info: The following rule cannot be assumed:
  pre(lp,hist).
{
  answer(eng,lp),
  answer(hist,eng),
  answer(hist,lp)
}
Info: 3 tuples computed.
```

So, the system informs that there is an inconsistency when trying to assert such offending fact (**pre(lp,hist)**), which makes prerequisites to form a cycle (as shown in the offending value list **[ic(lp),ic(eng),ic(hist)]**). The system informs about the rules that cannot be assumed but continues its processing. This is also useful to know the result for the admissible assumptions. Note that, in general, offending facts can be a subset of the meaning of an assumed rule in the context of the current database. To illustrate this, let's consider the following program for throwing a coin:

```
% Tails win:
:- win, heads.

win :- heads ; tails.
```

Predicate **win** states that one wins if either heads or tails are got, and the constraint states that you have to get tails to win. Then, the following hypothetical goal states whether assuming heads or tails leads to win.

```
DES> heads /\ tails => win
Info: Processing:
  answer :-
    heads/\tails=>win.
Error: Integrity constraint violation.
       ic :-
         win,
         heads.
Info: The following rule cannot be assumed:
  heads.
{
  answer
}
Info: 1 tuple computed.
```

As informed, **heads** cannot be assumed in order to win.

4.1.16.2 Hypothetical Queries and Duplicates

Duplicates can also be used along computations involving assumptions. Let's consider a variation of the classical Nim game, known as the subtraction game. Here, there is only one heap from which a player can take one or two tokens in his turn. A player wins if there is only one token in other player's turn (*misère* game). This can be formulated with the next program:

```
win_nim :-
  take       => one_left.
win_nim :-
```

```
  take/\take => one_left.
win_nim :-
  take       => enough, win_nim.
win_nim :-
  take/\take => enough, win_nim.

one_left :-
  total(N),
  count(take,C),
  N-C=1.

enough :-
  total(N),
  count(take,C),
  C>0.

total(4).
```

The predicate **win_nim** states that I win if I take one or two tokens and there is one left for you. Otherwise, if there are enough tokens (after taking one or two) to continue playing, then let's see if I can win.

Each occurrence of **take** in the left hand side of **=>** is an assumed fact that can be counted if duplicates are enabled (otherwise, the counting will be 0 - if there is no one - or 1 - if there is one or more, as duplicates are discarded). So, the predicate **one_left** determines whether there is exactly one token left, and **enough** determines if there is one token left at least. The predicate **total** states the total number of tokens which are available for a game.

For instance, if we had 4 tokens and was my turn, I cannot ensure to win because the other player can take only one token and, then, in my next turn, should I take either one or two, I'll lose.

```
DES> win_nim
{
}
Info: 0 tuples computed.
```

4.1.16.3 Hypothetical Queries and Negation

Implication can also be used in conjunction with negation. Let's consider the following example, which states flight links (**flight**/2 for origin and destination) between airports (airport}), and where flight travels (**flight_travel**/2 also for origin and destination) are possible if involved airports are not closed:

```
flight_travel(X,Y) :-
  flight(X,Y),
  not(closed(X)),
  not(closed(Y)).
flight_travel(X,Y) :-
  flight_travel(X,Z),
  flight_travel(Z,Y).

flight(a,b).
flight(b,c).
```

```
flight(c,d).
```

A regular query for consulting possible travels is:

```
DES> flight_travel(X,Y)
{
  flight_travel(a,b),
  flight_travel(a,c),
  flight_travel(a,d),
  flight_travel(b,c),
  flight_travel(b,d),
  flight_travel(c,d)
}
Info: 6 tuples computed.
```

Assuming that airport **b** is closed, we ask for the possible travels with this assumption:

```
DES> closed(b) => flight_travel(X,Y)
Info: Processing:
  answer(X,Y) :-
    closed(b)=>flight_travel(X,Y).
{
  answer(c,d)
}
Info: 1 tuple computed.
```

where negated calls to **closed**/1 occur in the first rule of **flight_travel**/2.

We can also ask for the opposite: Which are the flight travels which are not possible for that assumption:

```
DES> flight_travel(X,Y),(closed(b)=>not(flight_travel(X,Y)))
Info: Processing:
  answer(X,Y) :-
    flight_travel(X,Y),
    closed(b)=>not(flight_travel(X,Y)).
{
  answer(a,b),
  answer(a,c),
  answer(a,d),
  answer(b,c),
  answer(b,d)
}
Info: 5 tuples computed.
```

Note that, first, we ask for all the possible flights (first goal **flight_travel(X,Y)**) and, then, we restrict to those flights which are not possible under the assumption. The first goal is needed for the query to be safe. Recall that Datalog with negation is not constructive (variables in the negated goal are not instantiated unless their values are already provided by a positive goal), and answers must be ground. Note, also, that the meaning of the first occurrence of goal **flight_travel(X,Y)** in this last query is the very same as the meaning of the first query. However, the meaning of the second occurrence of that goal restricts the answer to those flights for which involved airports are not closed because of the assumption.

4.2 SQL

The syntax recognized by the interpreter is borrowed from the SQL standard. This section describes the main limitations, features, and decisions taken in designing SQL, which coexists with Datalog. Also, we describe the four parts of the supported subset of the SQL language: DDL (Data Definition Language, for defining the database schema), DQL (Data Query Language, for listing contents of the database) and DML (Data Manipulation Language, for inserting and deleting tuples), and ISL (Information Schema Language). Section 4.2.8 resumes the SQL grammar. As ODBC connections are allowed, some DBMS specific features have been added, as well as non-standard features in ISL.

4.2.1 Main Limitations

- The projection list consists of column references (***`column`***, ***`table.column`***, ***`alias.column`***), wildcards (**`*`**, ***`table.*`***, ***`alias.*`***), alias references, arithmetic expressions and SQL statements. Other expressions might be supported in further releases.
- A limited coverage of database integrity constraints.
- Strong typing. Different numeric type values cannot be compared (e.g., real and integer). Also, there is no provision for automatic type casting
- No provision for ordering results (**`order by`** clause).
- No insertions/deletions/updates into views.
- Limited syntax error reports. The parser does not inform about all the possible syntax error causes, but for table, view and column misspelled names. However, syntax errors from ODBC connections are displayed.

4.2.2 Main Features

As main features, we highlight:

- Data query, data definition, and data manipulation language parts provided.
- Subqueries (nested queries without depth limits).
- Correlated queries (tables and relations in nested subqueries can be referenced by the host query). For example: **`SELECT * FROM t,(SELECT a FROM s) WHERE t.a=s.a`**.
- Subqueries in comparisons, as **`SELECT a FROM t WHERE t.a > (SELECT a FROM s)`**.
- Table, relation, and expression aliases with full scope.
- Support for duplicates and duplicate elimination
- Non-linear recursive queries.
- Recursive queries are not restricted w.r.t. aggregates or nested computations as usual RDBMS's are (IBM DB2, MS SQL Server, SUN Oracle, MySQL, ...)

- Simplified recursive queries are allowed: Although supported, there is no need for using a **`WITH`** clause
- Hypothetical queries, which are a novel proposal out of the standard
- Set operators build relations, which can be used wherever a data source is expected (**`FROM`** clause).
- Null values are supported, along with outer joins (full, left and right).
- Aggregate functions allowed in expressions at the projection list and **`HAVING`** conditions. **`GROUP BY`** clauses are also allowed.
- View support. Any relation built with an SQL query can be defined as a view (even recursive queries).
- Supported database integrity constraints include type constraints, existency (nullability), primary keys, candidate keys, and referential integrity constraints.
- Parentheses can be used elsewhere they are needed and also for easing the reading of statements.
- Suggestions are provided for misspelled table, view and column names when similar entries are found

4.2.3 Datalog vs. SQL

With respect to Datalog, some decisions have been taken:

- As in Datalog, user identifiers are case-sensitive (table and attribute names, ...). This is not the normal behaviour of current relational database systems.
- In contrast to Datalog, built-in identifiers are not case-sensitive. This conforms to the normal behaviour of current relational database systems.

4.2.4 Data Definition Language

This part of the language deals with creating (or replacing), and dropping tables and views. There is no provision for updating the schema, which can be consulted with the command **`/dbschema`**.

4.2.4.1 Creating Tables

The first form of this statement is as follows:

`CREATE [OR REPLACE] TABLE` ***`TableName`*** **`(`** ***`Column1 Type1`*** **`[`** ***`ColumnConstraint1`*** **`], ...,`** ***`ColumnN TypeN`*** **`[`** ***`ColumnConstraintN`*** **`] [,`** ***`TableConstraints`*** **`])`**

This statement defines the table schema with name ***`TableName`*** and column names ***`Column1`***, ..., ***`ColumnN`***., with types ***`Type1`***, ..., ***`TypeN`***, respectively. If the optional clause **`OR REPLACE`** is used, the table is dropped if existed already, deleting all of its tuples.

A second form of this statement allows to create a table with the same schema of an existing table, following SQL standard optional feature T171:

`CREATE TABLE` ***`TableName`*** **`( LIKE`** ***`ExistingTableName`*** **`)`**

Parentheses are not mandatory, though. This version copies the complete schema, including all integrity constraints (both predefined and user-defined).

There is provision for several column constraints:

- **`NOT NULL`**. Existency constraint forbiding null values
- **`PRIMARY KEY`**. Primary key constraint for only one column
- **`UNIQUE`**. Uniqueness constraint for only one column (Also allowed the alternative syntax: **`CANDIDATE KEY`**)
- **`REFERENCES`** ***`TableName`*****`[(`*****`Column`*****`)]`**. Referential integrity constraint for only one column
- **`DETERMINED BY`** ***`Column`***. Functional dependency. If this constraint is applied to the column ***`Column1`***, then: ***`Column`*** → ***`Column1`***

Check constraints are not supported in this syntax up to now. However, they can be imposed via Datalog user-defined constraints as explained in Section 4.1.15.7.

Also, there is provision for several table constraints:

- **`PRIMARY KEY (`*****`Column`*****`,..., `*****`Column`*****`)`** . Primary key constraint for one or more columns
- **`UNIQUE (`*****`Column`*****`,..., `*****`Column`*****`)`** . Uniqueness constraint for one or more columns (Also allowed the non-standard alternative syntax: **`CANDIDATE KEY (`*****`Column`*****`,..., `*****`Column`*****`)`**)
- **`FOREIGN KEY (`*****`Column1`*****`,..., `*****`ColumnN`*****`) REFERENCES`** ***`TableName`*****`[(`*****`Column1`*****`,..., `*****`ColumnN`*****`)] )]`**. Referential integrity constraint for one or more columns
- **`CHECK (`*****`CheckConstraint`*****`)`**. Check constraint, as listed next

Check constraints:

- ***`Condition`***. As in a **`WHERE`** clause
- **`(`*****`ColumnR1`*****`,..., `*****`ColumnRN`*****`) DETERMINED BY (`*****`ColumnL1`*****`,..., `*****`ColumnLN`*****`)`**. Functional dependency: ***`ColumnL1,...,ColumnLN`*** → ***`ColumnR1,...,ColumnRN`***

Allowed types include:

- **`CHAR`**. Fixed-length string of 1
- **`CHAR(`*****`n`*****`)`**. Fixed-length string of *n* characters
- **`VARCHAR(`*****`n`*****`)`**. Variable-length string of up to n characters
- **`VARCHAR`** (or **`STRING`**). Variable-length string of up to the maximum length of the underlying Prolog atom
- **`INTEGER`** (or **`INT`**) . Integer number
- **`REAL`**. Real number

Examples:

```
CREATE TABLE t(a INT PRIMARY KEY, b STRING)

CREATE OR REPLACE TABLE s(a INT, b INT REFERENCES t(a), PRIMARY
KEY (a,b))
```

Note in this last example that if the column name in the referential integrity constraint is missing, the referred column of table **t** is assumed to have the same name that the column of **s** where the constraint applies (i.e., **b**). So, an error is thrown because columns **s.b** and **t.b** have different types:

```
DES-SQL> CREATE OR REPLACE TABLE s(a INT, b INT REFERENCES t,
PRIMARY KEY (a,b))

Error: Type mismatch s.b:number(int) <> t.b:string(varchar).
Error: Imposing constraints.
```

A declared primary key or foreign key constraint is checked whenever a new tuple is added to a table, following relational databases. Note that assertion of rules from the Datalog side are allowed but not checked. A Datalog rule should be viewed as a component of the intensional database. RDBs avoid to define a view with the same name as a table and, therefore, there is no way of unexpected behaviours such as the illustrated below:

```
DES-SQL> create or replace table t(a int, b int, c int, d int,
primary key (a,c))

DES-SQL> insert into t values(1,2,3,4)
Info: 1 tuple inserted.

DES-SQL> % The following is expected to raise an error:

DES-SQL> insert into t values(1,1,3,4)
Error: Primary key violation when trying to insert: t(1,1,3,4)
Info: 0 tuples inserted.

DES-SQL> % However, the following is allowed:

DES-SQL> /assert t(X,Y,Z,U) :- X=1,Y=2,Z=3,U=4.

DES-SQL> /listing
t(1,2,3,4).
t(X,Y,Z,U) :-
  X = 1,
  Y = 2,
  Z = 3,
  U = 4.
```

Production rules (i.e., those defining the intensional database) are not checked for primary key and foreign key constraints.

Next, a very simple example is reproduced to illustrate basic constraint handling:

```
DES-SQL> create or replace table u(b int primary key,c int)

DES-SQL> create or replace table s(a int,b int, primary key
(a,b))
```

```
DES-SQL> create or replace table t(a int,b int,c int,d int,
primary key (a,c), foreign key (b,d) references s(a,b), foreign
key(b) references u(b))

DES-SQL> insert into t values(1,2,3,4)
Error: Foreign key violation t.[b,d]->s.[a,b] when trying to
insert: t(1,2,3,4)
Info: 0 tuples inserted.

DES-SQL> insert into s values(2,4)
Info: 1 tuple inserted.

DES-SQL> insert into t values(1,2,3,4)
Error: Foreign key violation t.[b]->u.[b] when trying to insert:
t(1,2,3,4)
Info: 0 tuples inserted.

DES-SQL> insert into u values(2,2)
Info: 1 tuple inserted.

DES-SQL> insert into t values(1,2,3,4)
Info: 1 tuple inserted.

DES-SQL> /listing
s(2,4).
t(1,2,3,4).
u(2,2).
```

4.2.4.2 Creating Views

CREATE [OR REPLACE] VIEW *ViewName*(*Column1*, ..., *ColumnN*)
AS *SQLStatement*

This statement defines the view schema in a similar way as defining tables. If the optional clause **OR REPLACE** is used, the view is dropped if existed already. Other tuples or rules asserted (with the command **/assert**) are not deleted. The view is created with the SQL statement ***SQLStatement*** as its definition.

Note that column names are mandatory.

Examples:

```
DES> /dbschema
Info: Table(s):
 * s(a:number(integer),b:number(integer))
    - PK: [a,b]
 * u(b:number(integer),c:number(integer))
    - PK: [b]
 *
t(a:number(integer),b:number(integer),c:number(integer),d:number
(integer))
    - PK: [a,c]
    - FK: t.[b,d] -> s.[a,b]
    - FK: t.[b] -> u.[b]
Info: View(s):
```

```
 * v(a:number(integer),b:number(integer),c:number(integer),
d:number(integer))
    - Defining SQL Statement:
        SELECT ALL *
        FROM
          t
        WHERE a > 1;
    - Datalog equivalent rules:
        v(A,B,C,D) :-
          t(A,B,C,D),
          A > 1.
 * w(a:number(integer),b:number(integer))
    - Defining SQL Statement:
        SELECT ALL t.a, s.b
        FROM
          t,
          s
        WHERE t.a > s.a;
    - Datalog equivalent rules:
        w(A,B) :-
          t(A,C,D,E),
          s(F,B),
          A > F.
Info: No integrity constraints.
```

Note that primary key constraints follow the table schema, and inferred types are in the view schema.

4.2.4.3 Dropping Tables

DROP TABLE [IF EXISTS] ***TableName*****,...,*****TableName***

This statement drops the table schema corresponding to each one of the provided names (***TableName***), deleting all of its tuples (whether they were inserted with **INSERT** or with the command **/assert**) and rules (which might have been added via **/assert**). If the optional clause **IF EXISTS** is included, dropping an inexistent table does not raise an error.

Example:

```
DROP TABLE t;
```

4.2.4.4 Dropping Views

DROP VIEW ***ViewName***

This statement drops the view with name ***ViewName***, deleting all of its tuples (whether they were inserted with INSERT or with the command **/assert**) and rules (which might have been added via **/assert**). Other tuples or rules asserted (with the command **/assert**) are not deleted.

Example:

```
DROP VIEW v;
```

4.2.4.5 Renaming Tables

RENAME TABLE *TableName* TO *NewTableName*

This non standard statement (following IBM DB2) allows to change the name of table ***TableName*** to ***NewTableName***. Foreign keys referring to this table are modified accordingly. Also, views including referenes to this table are modified to refer to the new name.

4.2.4.6 Renaming Views

RENAME VIEW *ViewName* TO *NewViewName*

This non standard statement (following IBM DB2) allows to change the name of view ***ViewName*** to ***NewViewName***. Also, views including references to this view are modified to refer to the new name.

4.2.4.7 Dropping Databases

DROP DATABASE

This statement drops the current database, dropping all tables, views, and rules (this includes Datalog rules and constraints that may have been asserted or consulted). It behaves exactly as the command **/abolish**.

Example:

```
DROP DATABASE;
```

4.2.5 Data Manipulation Language

This part of the language deals with inserting and deleting tuples from tables. There is no provision for updating tuples.

4.2.5.1 Inserting Tuples

INSERT INTO *TableName*[(*Col1*,…,*ColN*)] VALUES (*Cte1*,...,*CteN*) [, ..., (*Cte1*,...,*CteN*)]

This statement inserts into the table ***TableName*** as many tuples as those built with each tuple of values ***Cte1***, ..., ***CteN***. ***Col1*** to ***ColN*** are non-repeated column names of the table. If no column names are given, ***N*** is expected to be the number of columns of the table. If column names are given, each value ***Ctei*** corresponds to column name ***Coli***. For those column names which are not provided in a column name sequence, nulls are inserted.

The next example inserts a single tuple:

```
CREATE TABLE t(a int, b int)
INSERT INTO t VALUES (1,1)
```

The next one inserts a single tuple into the same table with a null for column **a**:

```
INSERT INTO t(b) VALUES (2)
```

Which is equivalent to:

```
INSERT INTO t(b,a) VALUES (2,null)
```

and represents the tuple **(null,2)**. (Note that the order of provided column names are reversed with respect to the table definition.)

For inserting several tuples at a time:

```
INSERT INTO t VALUES (1,1),(null,2)
```

Another form of the **INSERT** statement allows to inserting tuples which are the result set from a **SELECT** statement:

INSERT INTO *TableName*[(*Col1*,…,*ColN*)] *SQLStatement*

This statement inserts into the table ***TableName*** as many tuples as returned by the SQL statement ***SQLStatement***. This statement has to return as many columns as either the columns of ***TableName***, if no column names are given, or the number of provided column names (***N***), otherwise.

Examples:

```
INSERT INTO t SELECT * FROM s
```

You can also insert tuples into a table coming directly (or indirectly) from the table itself for duplicating rows, as in:

```
INSERT INTO t SELECT * FROM t
```

Note that there is no recursion in this query as the source table **t** is not changed during solving the **SELECT** statement.

For testing the new (duplicated) contents of **t**, you have to use **/listing t**, instead of a **SELECT**, since this statement always returns a set (no duplicates) when duplicates are disabled (cf. Section 4.1.9).

You can specify columns of the target table as in:

```
INSERT INTO t(b) SELECT a FROM t
```

which inserts as many rows in t as it had before insertion, and for each row, a new tuple is built with the value of the source column **a** in the target column **b**, and null in the target column **a**.

4.2.5.2 Deleting Tuples

DELETE FROM *TableName*

This statement deletes all the tuples of the table ***TableName***. It does not delete production rules asserted via **/assert**.

Example:

```
DELETE FROM t
```

Another form of the **DELETE** statement allows to deleting tuples which fulfil a given condition:

DELETE FROM *TableName* WHERE *Condition*

This statement deletes from the table ***TableName*** all of its tuples matching the condition ***Condition***. It does not delete production rules asserted via **/assert**.

Example:

```
DELETE FROM t WHERE a NOT IN (SELECT a FROM s)
```

4.2.6 Data Query Language

There are three main types of SQL query statements: **SELECT** statements, set statements (**UNION**, **INTERSECT**, and **EXCEPT**), and **WITH** statements (for building recursive queries).

4.2.6.1 Basic SQL Queries

The syntax of the basic SQL query statement is:

```
SELECT [DISTINCT|ALL] ProjectionList
[FROM Relations
[WHERE Condition]
[ORDER BY OrdExpressions] ]
```

Where:

- Square brackets indicate that the enclosed text is optional. Also, the vertical bar is used to denote alternatives.
- ***ProjectionList*** is a list of comma-separated columns or arithmetic expressions that will be returned as a tuple result. Wildcards are allowed, as ***** (for referring to all the columns in the data source) and ***Relation.**** (for referring to all the columns in the relation ***Relation***). The name ***Relation*** can be the name of a table or an alias (for a table or subquery). Clause **DISTINCT** discards duplicates whereas clause **ALL** does not (this is only noticeable when duplicates are enabled with the command **/duplicates on**).
- ***Condition*** is a logical condition built from comparison operators (**=**, **<>**, **<**, **>**, **>=**, and **<=**), Boolean operators (**AND**, **OR**, and **NOT**), Boolean constants (**TRUE**, **FALSE**), the existence operator (**EXISTS**) and the inclusion operator (**IN**). See the grammar description in Section 4.2.8 for details. Subqueries are allowed with no limitations.
- ***Relations*** is a list of comma-separated relation definitions. A relation can be either a table name, or a view name, or a subquery, or a join relation. They can be renamed via aliases. If no **FROM** clause is provided, the built-in **DUAL** relation is used as a data source (cf. Section 4.2.6.1.2).
- ***OrdExpressions*** is a list of comma-separated ordering expressions. An ordering expression can be either simply an expression or an expression followed by the ordering criterium (**ASC** -or **ASCENDING**- for ascending, and **DESC** -or **DESCENDING**- for descending).

Examples:

Given the tables:

```
CREATE TABLE s(a int, b int);
CREATE TABLE t(a int, b int);
CREATE TABLE v(a int, b int);
```

We can submit the following queries:

```
SELECT distinct a
FROM t

SELECT t.*, s.b
FROM t,s,v
WHERE t.a=s.a AND v.b=t.b

SELECT t.a, s.b, t.a+s.b
FROM t,s
WHERE t.a=s.a

SELECT *
FROM (SELECT * from t) as r1,
     (SELECT * from s) as r2
WHERE r1.a=r2.b;

SELECT *
FROM s
WHERE s.a NOT IN SELECT a FROM t;

SELECT *
FROM s
WHERE EXISTS
  SELECT a
  FROM t
  WHERE t.a=s.a;

SELECT *
FROM s
WHERE s.a > (SELECT a FROM t);

SELECT 1, a1+a2, a+1 AS a1, a+2 AS a2
FROM t;

SELECT 1;
```

Notes:

- SQL arithmetic expressions follow the same syntax as Datalog.
- An SQL arithmetic expression can be renamed and used in other expressions.
- Circular definitions will yield exceptions at run-time, as in **`a+a3 AS a3`**

A join relation is either of the form:

`Relation` **`NATURAL`** ***`JoinOp Relation`***

or:

`Relation JoinOp Relation` **`[`*****`JoinCondition`*****`]`**

Where ***`Relation`*** is as before (without any limitation), JoinOP is any join operator (including **`INNER JOIN`**, **`LEFT OUTER JOIN`**, **`RIGHT OUTER JOIN`**, and **`FULL OUTER JOIN`**), and ***`JoinCondition`*** can be either:

`ON` ***`Condition`***

or:

```
USING (Column1,...,ColumnN)
```

Where ***Condition*** is as described in a **WHERE** clause, and ***Column1***, ..., ***ColumnN*** are common column names of the joined relations.

Examples:

Given the tables:

```
CREATE TABLE s(a int, b int);
CREATE TABLE t(a int, b int);
CREATE TABLE v(a int, b int);
```

We can submit the following queries:

```
SELECT *
FROM t INNER JOIN s ON t.a=s.a AND t.b=s.b;

SELECT *
FROM t NATURAL INNER JOIN s;

SELECT *
FROM t INNER JOIN s USING (a,b);

SELECT * FROM t INNER JOIN s USING (a);

SELECT *
FROM t INNER JOIN s USING (b);

SELECT *
FROM (t INNER JOIN s ON t.a=s.a) AS s, v
WHERE s.a=v.a;

SELECT *
FROM (t LEFT JOIN s ON t.a=s.a) RIGHT JOIN v ON t.a=v.a;

SELECT * FROM t FULL JOIN s ON t.a=s.a;
```

Note:

The default keyword **ALL** following **SELECT** retains duplicates whenever duplicates are enabled (command /duplicates on). In turn, **DISTINCT** discards duplicates. But note that if duplicates are disabled, both **ALL** and **DISTINCT** behave the same (i.e., discarding duplicates).

4.2.6.1.1 Top-N Queries

The number of computed tuples for a select statements can be limited with the so-called Top-N queries. ISO 2008 includes this as a final clause in the select statement:

```
SELECT [DISTINCT|ALL] ProjectionList
FROM Rels
…
```

```
FETCH FIRST Integer ROWS ONLY
```

However, DES also provides another non-standard, but common form in other RDBMS's of such queries:

```
SELECT [TOP Integer] [DISTINCT|ALL] ProjectionList
…
```

You can switch the order of the top and distinct clauses, and even specify both forms of Top-N queries in the same statement, as long as they express the same limit.

4.2.6.1.2 The `dual` table

The **`dual`** table is a special one-row, one-column table present by default in all Oracle database installations. It is suitable for use in selecting a pseudocolumn with no data source. As propositional relations are also allowed in DES, **`dual`** does not need a column at all, and it is therefore defined as a single fact without arguments. This table can be used to compute arithmetics as, e.g.:

```
DES-SQL> select 1+1 from dual
answer($a0) ->
{
  answer(2)
}
Info: 1 tuple computed.
```

As in MySQL, DES also allows to omit the **`FROM`** clause in theses cases (the compilation from SQL to Datalog adds the **`dual`** table as data source):

```
DES-SQL> select 1+1
answer($a0) ->
{
  answer(2)
}
Info: 1 tuple computed.
```

Although this table is not displayed with the command **`/dbschema`**, it can be nevertheless dropped with a **`DROP TABLE`** SQL statement. If it is deleted, the just described behaviour is no longer possible. In addition, it cannot be redeclared with a **`CREATE TABLE`** SQL statement, but with a type declaration, as **`:-type(dual)`**. Both **`DROP DATABASE`** statement and **`/abolish`** command restore this table.

4.2.6.2 Relational Division in SQL

The division operation was originally introduced as a relational operation in Codd's paper about relational model. Although it seems to be a practical operation, it is not included in current DBMS's. However, DES includes a **`DIVISION`** operator that can be used in the **`FROM`** clause of a **`SELECT`** statement. The next system session illustrates its use:

```
DES> create table t(a int, b int)
DES> create table s(a int)
DES> insert into t values (1,1)
Info: 1 tuple inserted.
DES> insert into t values (1,2)
Info: 1 tuple inserted.
```

```
DES> insert into t values (2,1)
Info: 1 tuple inserted.
DES> insert into s values (1)
Info: 1 tuple inserted.
DES> insert into s values (2)
Info: 1 tuple inserted.
DES> select * from t division s
answer(t.b:number(integer)) ->
{
  answer(1)
}
Info: 1 tuple computed.
```

4.2.6.3 Set SQL Queries

The three set operators defined in the standard are available: **`UNION`**, **`EXCEPT`**, and **`INTERSECT`**. (Also, Oracle's **`MINUS`** is allowed as a synonymous for **`EXCEPT`**.) The first one also admits the form **`UNION ALL`** for retaining duplicates. The syntax of a set SQL query is:

```
SQLStatement
SetOperator
SQLStatement
```

Where ***`SQLStatement`*** is any SQL statement described in the data query part (without any limitation). ***`SetOperator`*** is any of the abovementioned set operators.

Examples:

```
(SELECT * FROM s) UNION     (SELECT * FROM t);

(SELECT * FROM s) UNION ALL (SELECT * FROM t);

(SELECT * FROM s) INTERSECT (SELECT * FROM t);

(SELECT * FROM s) EXCEPT    (SELECT * FROM t);
```

Note that parentheses are not mandatory in these cases and are only used for readability.

4.2.6.4 `WITH` SQL Queries

The WITH clause, as introduced in the SQL:1999 standard and available in several RDBMS as DB2, Oracle and SQL Server, is intended in particular to define recursive queries. Its syntax is:

```
WITH LocalViewDefinition1,
     ...,
     LocalViewDefinitionN
SQLStatement
```

Where ***`SQLStatement`*** is any SQL statement, and

`LocalViewDefinition1`, ..., ***`LocalViewDefinition1`*** are (local) view definitions that can only be used inside ***`SQLStatement`***. These local views are not stored in the database and are rather computed when executing ***`SQLStatement`***. Although they are local, they must have different names from existing objects (tables or views). The syntax of a local view definition is as follows:

```
[RECURSIVE] ViewName(Column1, ..., ColumnN) AS SQLStatement
```

Here, the keyword **RECURSIVE** for defining recursive views is not mandatory (the parser simply ignores it).

Examples[6]:

```
CREATE TABLE flights(airline,frm,to,departs,arrives);

WITH
  RECURSIVE reaches(frm,to) AS
    (SELECT frm,to FROM flights)
    UNION
    (SELECT r1.frm,r2.to
     FROM reaches AS r1, reaches AS r2
     WHERE r1.to=r2.frm)
SELECT * FROM reaches;

WITH
  Triples(airline,frm,to) AS
    SELECT airline,frm,to
    FROM flights,
  RECURSIVE Reaches(airline,frm,to) AS
    (SELECT * FROM Triples)
    UNION
    (SELECT Triples.airline,Triples.frm,Reaches.to
     FROM Triples,Reaches
     WHERE Triples.to = Reaches.frm AND
           Triples.airline=Reaches.airline)
(SELECT frm,to FROM Reaches WHERE airline = 'UA')
 EXCEPT
(SELECT frm,to FROM Reaches WHERE airline = 'AA');
```

In addition, shorter definitions for recursive views are allowed in DES. The next view delivers the same result set as the first example above:

```
CREATE VIEW reaches(frm,to) AS
    (SELECT frm,to FROM flights)
    UNION
    (SELECT r1.frm,r2.to
     FROM reaches AS r1, reaches AS r2
     WHERE r1.to=r2.frm);
```

4.2.6.5 Hypothetical SQL Queries

A novel addition to SQL in DES includes hypothetical queries. Such queries are useful, for instance, in decision support systems as they allow to submit a query by assuming some knowledge which is not in the database.

[6] Adapted from [GUW02].

Syntax of hypothetical queries is proposed as:

```
ASSUME
 LocalAssumption1,
 ...,
 LocalAssumptionN
SQLStatement
```

Where ***SQLStatement*** is any SQL DQL statement, and ***LocalAssumption1***, ..., ***LocalAssumptionN*** are of the form:

```
DQLStatement IN ExistingRelation
```

And ***LocalAssumptionN*** are added as unions to existing relations (either tables or views). Syntax of these local view definitions are as in **WITH** statements.

As an example, let's consider a flight database defined by the following:

```
CREATE TABLE flight(origin string, destination string, time
real);

INSERT INTO flight VALUES('lon','ny',9.0);
INSERT INTO flight VALUES('mad','par',1.5);
INSERT INTO flight VALUES('par','ny',10.0);

CREATE OR REPLACE VIEW travel(origin,destination,time) AS WITH
connected(origin,destination,time) AS
    SELECT * FROM flight
  UNION
    SELECT flight.origin,connected.destination,
           flight.time+connected.time
    FROM flight,connected
    WHERE flight.destination = connected.origin
SELECT * FROM connected;
```

Here, relation **flight** represents possible direct flights between locations, and **travel** represents possible connections by using one or more direct flights. Both include flight time. By querying the relation travel, we get:

```
DES> select * from travel
answer(travel.origin:string(varchar),travel.destination:string(v
archar),travel.time:number(float)) ->
{
  answer(lon,ny,9.0),
  answer(mad,ny,11.5),
  answer(mad,par,1.5),
  answer(par,ny,10.0)
}
Info: 4 tuples computed.
```

Now, if we assume there is a tuple **flight('mad','lon',2.0)**, we can query the database with this assumption with the following query (with multi-line input enabled):

```
DES> ASSUME
       SELECT 'mad','lon',2.0
```

```
     IN
       flight(origin,destination,time)
     SELECT * FROM travel;

answer(travel.origin:string(varchar),travel.destination:string(v
archar),travel.time:number(float)) ->
{
  answer(lon,ny,9.0),
  answer(mad,lon,2.0),
  answer(mad,ny,11.0),
  answer(mad,ny,11.5),
  answer(mad,par,1.5),
  answer(par,ny,10.0)
}
Info: 6 tuples computed.
```

Note that the **SELECT** statement following the keyword **ASSUME** simply stands for the construction of a single tuple for table **flight** (such statement can be otherwise stated as **SELECT 'mad','lon',2.0 FROM dual**, where **dual** is the built-in table described in Section 4.2.6.1.2).

In addition, not only tuples can be extensionally assumed, but any SQL DQL statement, i.e., tuples intensionally assumed. As an example, let's suppose that the relation **flight** is as previously defined, and a view **connect** that displays locations connected by direct flights:

```
DES> CREATE VIEW connect(origin,destination) AS
     SELECT origin,destination FROM flight;

DES> SELECT * FROM connect;
answer(connect.origin:string(varchar),connect.destination:string
(varchar)) ->
{
  answer(lon,ny),
  answer(mad,par),
  answer(par,ny)
}
Info: 3 tuples computed.
```

Then, if we assume that connections are allowed with transits, we can submit the following hypothetical query (note that the assumed SQL statement is recursive):

```
DES> ASSUME
       (SELECT flight.origin,connect.destination
        FROM flight,connect
        WHERE flight.destination = connect.origin)
     IN
       connect(origin,destination)
     SELECT * FROM connect;

answer(connect.origin:string(varchar),connect.destination:string
(varchar)) ->
{
  answer(lon,ny),
  answer(mad,ny),
```

```
  answer(mad,par),
  answer(par,ny)
}
Info: 4 tuples computed.
```

In addition to this, one can use a WITH statement instead of an ASSUME statement, by simply stating an existing relation in the definition of the local view. For instance, for the last example, we can write:

```
DES> WITH
       connect(origin,destination) AS
         (SELECT flight.origin,connect.destination
          FROM flight,connect
          WHERE flight.destination = connect.origin)
     SELECT *
     FROM connect;

answer(connect.origin:string(varchar),connect.destination:string
(varchar)) ->
{
  answer(lon,ny),
  answer(mad,ny),
  answer(mad,par),
  answer(par,ny)
}
Info: 4 tuples computed.
```

One can use several assumptions in the same query, but only one for a given relation. If needed, you can assume several rules by using **UNION**. For example:

```
WITH
flight(origin,destination,time) AS
  SELECT 'mad','lon',2.0
  UNION
  SELECT 'ny','par',10.0
SELECT *
FROM travel;
```

which is equivalent to:

```
ASSUME
  SELECT 'mad','lon',2.0
  UNION
  SELECT 'ny','par',10.0
IN
  flight(origin,destination,time)
SELECT *
FROM travel;
```

Note:

SQL queries are only allowed as such, i.e., they cannot be used as part of any view declaration. Further versions might allow this.

4.2.7 Information Schema Language (ISL)

Several non-standard statements are provided to display schema information:

- **`SHOW TABLES;`** List table names. *TAPI enabled*
- **`SHOW VIEWS;`** List view names. *TAPI enabled*
- **`SHOW DATABASES;`** List database names. *TAPI enabled*
- **`DESCRIBE`** ***`Relation`*****`;`** Display schema for Relation, as **`/dbschema`**

4.2.8 SQL Grammar

Here, terminal symbols are: Parentheses, commas, semicolons, single dots, asterisks, and apostrophes. Other terminal symbols are completely written in capitals, as **`SELECT`**. Percentage symbols (**`%`**) start comments. User identifiers must start with a letter and consist of letters and numbers; otherwise, a user identifier can be enclosed between quotation marks (both square brackets and double quotes are supported) and contain any characters. Next, **`SQLstmt`** stands for a valid SQL statement.

```
SQLstmt ::=
  DDLstmt[;]
  |
  DMLstmt[;]
  |
  DQLstmt[;]
  |
  ISLstmt[;]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DDL (Data Definition Language) statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DDLstmt ::=
  CREATE [OR REPLACE] TABLE CompleteConstrainedSchema
  |
  CREATE [OR REPLACE] TABLE TableName LIKE TableName
  |
  CREATE [OR REPLACE] VIEW ViewSchema AS DQLstmt
  |
  RENAME TABLE TableName TO TableName
  |
  RENAME VIEW ViewName TO ViewName
  |
  DROP TABLE [IF EXISTS] TableName,...,TableName % Extended
syntax following MySQL 5.6
  |
  DROP VIEW ViewName
  |
  DROP DATABASE

Schema ::=
  RelationName
  |
  RelationName(Att,...,Att)
```

```
CompleteConstrainedSchema ::=
  RelationName(Att Type [ColumnConstraint ...
ColumnConstraint],...,Att Type [ColumnConstraint ...
ColumnConstraint] [, TableConstraints])

CompleteSchema ::=
  RelationName(Att Type,...,Att Type)

Type ::=
  CHAR(n)  % fixed-length string of n characters
  |
%   CHARACTER(n)  % equivalent to the former
%   |
  CHAR  % fixed-length string of 1 character
  |
  VARCHAR(n)  % variable-length string of up to n characters
  |
  VARCHAR2(n)  % Oracle's variable-length string of up to n
characters
  |
  VARCHAR  % variable-length string of up to the maximum length
of the underlying Prolog atom
  |
  STRING  % As VARCHAR
  |
%   CHARACTER VARYING(n)  % equivalent to the former
%   |
  INT
  |
  INTEGER  % equivalent to the former
  |
%   SMALLINT
%   |
%   NUMERIC(p,d) % a total of p digits, where d of those are in
the decimal place
%   |
  REAL
  |
%   DOUBLE PRECISION  % equivalent to the former
%   |
%   FLOAT(n)  % with precision of at least n digits
%   |
%   DATE % four digit year, month and day
%   |
%   TIME % hours, minutes and seconds
%   |
%   TIMESTAMP % combination of date and time


ColumnConstraint ::=
  NOT NULL
  |
  PRIMARY KEY
```

```
  |
  UNIQUE
  |
  CANDIDATE KEY
  |
  REFERENCES TableName[(Att)]
  |
  CHECK (CheckConstraint)

TableConstraints ::=
  TableConstraint,...,TableConstraint

TableConstraint ::=
  UNIQUE (Att,...,Att)
  |
  CANDIDATE KEY (Att,...,Att)
  |
  PRIMARY KEY (Att,...,Att)
  |
  FOREIGN KEY (Att,...,Att) REFERENCES TableName[(Att,...,Att)]
  |
  CHECK (CheckConstraint)

CheckConstraint ::=
  WhereCondition
  |
  (Att,...,Att) DETERMINED BY (Att,...,Att)

RelationName is a user identifier for naming tables, views and
aliases
TableName is a user identifier for naming tables
ViewName is a user identifier for naming views
Att is a user identifier for naming relation attributes

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DML (Data Manipulation Language) statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DMLstmt ::=
  INSERT INTO TableName[(Att,...,Att)]
    VALUES (Cte,...,Cte) [, ..., (Cte,...,Cte)]
  |
  INSERT INTO TableName[(Att,...,Att)] DQLstmt
  |
  DELETE FROM TableName
  |
  DELETE FROM TableName WHERE Condition
  |
  UPDATE TableName SET Att1=Expr1,...,Attn=Exprn WHERE Condition

Cte is a constant

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DQL (Data Query Language) statements:
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DQLstmt ::=
  (DQLstmt)
  |
  UBSQL

UBSQL ::=
  SELECTstmt
  |
  DQLstmt UNION [ALL] DQLstmt
  |
  DQLstmt EXCEPT DQLstmt
  |
  DQLstmt MINUS DQLstmt
  |
  DQLstmt INTERSECT DQLstmt
  |
  WITH LocalViewDefinition,...,LocalViewDefinition DQLstmt
  |
  ASSUME LocalAssumption,...,LocalAssumption DQLstmt

LocalViewDefinition ::=
  [RECURSIVE] CompleteSchema AS DQLstmt

LocalAssumption ::=
  DQLstmt IN CompleteSchema

SELECTstmt ::=
  SELECT [TOP Integer] [[ALL|DISTINCT]] SelectExpressionList
  [FROM Rels
   [WHERE WhereCondition]
   [GROUP BY Atts]
   [HAVING HavingCondition]
   [ORDER BY OrderDescription]
   [FETCH FIRST Integer ROWS ONLY]]

Atts ::=
  Att,...,Att

OrderDescription ::=
  Att [[ASC|DESC]],...,Att [[ASC|DESC]]

SelectExpressionList ::=
  *
  |
  SelectExpression,...,SelectExpression

SelectExpression ::=
  UnrenamedSelectExpression
  |
  RenamedExpression

UnrenamedSelectExpression ::=
```

```
  Att
  |
  RelationName.Att
  |
  RelationName.*
  |
  ArithmeticExpression
  |
  DQLstmt

RenamedExpression ::=
  UnrenamedExpression [AS] Identifier

ArithmeticExpression ::=
  Op1 ArithmeticExpression
  |
  ArithmeticExpression Op2 ArithmeticExpression
  |
  ArithmeticFunction(ArithmeticExpression,...,
                     ArithmeticExpression)
  |
  Number
  |
  Att
  |
  RelationName.Att
  |
  ArithmeticConstant
  |
  DQLstmt

Op1 ::=
  - | \

Op2 ::=
  ^ | ** | * | / | // | rem | \/ | xor | + | - | /\ | << | >>

ArithmeticFunction ::=
    sqrt/1 | ln/1 | log/1 | log/2 | sin/1 | cos/1 | tan/1 |
cot/1
  | asin/1 | acos/1 | atan/1 | acot/1 | abs/1 | float/1
  | integer/1 | sign/1 | gcd/2 | min/2 | max/2 | truncate/1
  | float_integer_part/1 | float_fractional_part/1
  | round/1 | floor/1 | ceiling/1

Aggregate Functions:
The argument may include a prefix "distinct" for all but "min"
and "max":
  avg/1 | count/1 | count/0 | max/1 | min/1 | sum/1 | times/1

ArithmeticConstant ::=
  pi | e

Rels ::=
```

```
  Rel,...,Rel

Rel ::=
  UnrenamedRel
  |
  RenamedRel

UnrenamedRel ::=
  TableName
  |
  ViewName
  |
  DQLstmt
  |
  JoinRel
  |
  DivRel

RenamedRel ::=
  UnrenamedRel [AS] Identifier

JoinRel ::=
  Rel [NATURAL] JoinOp Rel [JoinCondition]

JoinOp ::=
  INNER JOIN
  |
  LEFT [OUTER] JOIN
  |
  RIGHT [OUTER] JOIN
  |
  FULL [OUTER] JOIN

JoinCondition ::=
  ON WhereCondition
  |
  USING (Atts)

DivRel ::=
  Rel DIVISION Rel

WhereCondition ::=
  BWhereCondition
  |
  UBWhereCondition

HavingCondition
  As WhereCondition, but including aggregate functions

BWhereCondition ::=
  (WhereCondition)

UBWhereCondition ::=
  TRUE
```

```
  |
  FALSE
  |
  EXISTS DQLstmt
  |
  NOT (WhereCondition)
  |
  (AttOrCte,...,AttOrCte) [NOT] IN DQLstmt
  |
  WhereExpression IS [NOT] NULL
  |
  WhereExpression [NOT] IN DQLstmt
  |
  WhereExpression Operator [[ALL|ANY]] WhereExpression
  |
  WhereCondition [AND|OR] WhereCondition

WhereExpression ::=
  Att
  |
  Cte
  |
  ArithmeticExpression
  |
  DQLstmt

AggrArithmeticExpression ::=
  [AVG|MIN|MAX|SUM]([DISTINCT] Att)
  |
  COUNT([*|[DISTINCT] Att])

AttOrCte ::=
  Att
  |
  Cte

Operator ::=
  = | <> | < | > | >= | <=

Cte ::=
  Number
  |
  'String'
  |
  NULL
```

`Number` is an integer or floating-point number

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ISL (Information Schema Language) statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ISLstmt ::=
  SHOW TABLES
```

```
|
SHOW VIEWS
|
SHOW DATABASES
|
DESCRIBE [TableName|ViewName]
```

4.3 (Extended) Relational Algebra

Following the seminal proposal [Codd70] there have been some extensions to the basic and additional operators in the original proposal. Here, we include all the original and extended operators for dealing with outer joins, duplicate elimination, recursion, and grouping with aggregates.

With respect to textual syntax, we follow [Diet01], where arguments of functions are enclosed between parentheses (as relations), and subscripts and superscripts are delimited between blanks. Arguments in infix operators are not enclosed between any delimiters, also parentheses can be used to enhance reading. Conditions and expressions are built with the same syntax as in SQL. Examples below refer to the database defined in **examples/relop.ra**.

4.3.1 Operators

This section includes descriptions for basic, additional and extended operators.

4.3.1.1 Basic operators

- Selection $\sigma_\theta(R)$. Select tuples in relation R matching condition θ.

  Concrete syntax:

  select ***Condition*** **(*Relation*)**

  Example:

  ```
  select a<>'a1' (c);
  ```

- Projection $\pi_{A1,...,An}(R)$. Return all tuples in R only with columns $A_1,...,An$.

  Concrete syntax:

  project ***A1,...,An*** **(*Relation*)**

  Example:

  ```
  project b (c);
  ```

- Set union $R_1 \cup R_2$.

  Concrete syntax:

  Relation1 **union** ***Relation2***

  Example:

  ```
  a union b;
  ```

- Set difference R_1 - R_2.

Concrete syntax:

```
Relation1 difference Relation2
```

Example:

```
a difference b;
```

- Cartesian product $R_1 \times R_2$.

  Concrete syntax:

  ```
  Relation1 product Relation2
  ```

  Example:

  ```
  a product b;
  ```

- Renaming $\rho_{R2(A1,...,An)}(R_1)$. Rename R_1 to R_2, and also arguments of R_1 to $A1,...,An$.

  Concrete syntax:

  ```
  rename Schema (Relation)
  ```

  Example:

  ```
  project v.a (rename v(a) (select true (a)));
  ```

- Assignment $R_1(A1,...,An) \leftarrow R_2$. Create a new relation R_1 with argument names $A1,...,An$ as a copy of R_2. It allows to define new views.

  Concrete syntax:

  ```
  Relation1 := Relation2
  ```

  Example:

  ```
  v(c) := select true (a);
  ```

4.3.1.2 Additional operators

These operators can be expressed in terms of basic operators, and include:

- Set intersection $R_1 \cap R_2$.

  Concrete syntax:

  ```
  Relation1 intersect Relation2
  ```

  Example:

  ```
  a intersect b;
  ```

- Theta join $R_1 \bowtie_\theta R_2$. Equivalent to $\sigma_\theta(R_1 \times R_2)$.

  Concrete syntax:

  ```
  Relation1 zjoin Condition Relation2
  ```

  Example:

  ```
  a zjoin a.a<b.b b;
  ```

- Natural (inner) join $R_1 \bowtie R_2$. Return tuples of R_1 joined with R_2 such that common attributes are pair wise equal and occur only once in output relation.

  Concrete syntax:

  ```
  Relation1 njoin Relation2
  ```

  Example:

  ```
  a njoin c;
  ```

- Division $R_1 \div R_2$. Return restrictions of tuples in R_1 to the attribute names of R_1 which are not in the schema of R_2, for which it holds that all their combinations with tuples in R_2 are present in R_1. The attributes in R_2 form a proper subset of attributes in R_1.

  Concrete syntax:

  ```
  Relation1 division Relation2
  ```

  Example:

  ```
  a division c;
  ```

4.3.1.3 Extended operators

These operators *can not* be expressed in terms of former operators, and include:

- Extended projection $\pi_{E1,...,En}(R)$. Return tuples of R with columns E_1,...,E_n where each E_i is an expression built from constants and attributes of R.

  Concrete syntax:

  ```
  project E1,...,En (Relation)
  ```

  Example:

  ```
  :-type(d(a:string,b:int)).
  project b+1 (d);
  ```

- Duplicate elimination $\delta(R)$. Return tuples in R, discarding duplicates.

  Concrete syntax:

  ```
  distinct (Relation)
  ```

  Example:

  ```
  distinct (project a (c));
  ```

  Note: As **distinct** is also a Datalog (meta)predicate, the query **distinct (c)** from the Datalog prompt would be solved as a Datalog query, instead of a RA one. Then, if you have to ensure your query will be evaluated by the RA processor, you can either switch to RA with **/ra**, or prepend the query with **/ra**, as follows:

  ```
  DES> % Either switch to RA:
  DES>/ra
  DES-RA> distinct (project a (c));
  DES> /datalog
  ```

```
DES> % Or simply add /ra
DES>/ra distinct (project a (c));
```

- Left outer join $R_1 ⟕_{\theta} R_2$. Includes all tuples of R_1 joined with matching tuples of R_2 w.r.t. condition θ. Those tuples of R_1 which do not have matching tuples of R_2 are also included in the result, and columns corresponding to R_2 are filled with null values.

  Concrete syntax:

  Relation1* ljoin *Condition Relation2

  Example:

  ```
  a ljoin a=b b;
  ```

- Right outer join $R_1 ⟖_{\theta} R_2$. Equivalent to $R_2 ⟕_{\theta} R_1$. $R_1 ⟕_{\theta} R_2$

  Concrete syntax:

  Relation1* rjoin *Condition Relation2

  Example:

  ```
  a rjoin a=b b;
  ```

- Full outer join $R_1 ⟗_{\theta} R_2$. Equivalent to $R_1 ⟕_{\theta} R_2 \cup R_1 ⟖_{\theta} R_2$.

  Concrete syntax:

  Relation1* fjoin *Condition Relation2

  Example:

  ```
  a fjoin a=b b;
  ```

- Grouping with aggregations ${}_{G_1,\ldots,G_n}\varsigma^{E_1,\ldots,E_n}{}_{\theta}(R)$. Build groups of tuples in R so that: first, each tuple in the group have the same values for attributes $G_1,\ldots,G_n$, second, matches condition θ (possibly including aggregate functions) and, third, is projected by expressions $E_1,\ldots,E_n$ (also possibly including aggregate functions). An empty list of grouping attributes $G_1,\ldots,G_n$ is denoted by an opening and a closing bracket (**[]**).

  Concrete syntax:

  group_by *GroupingAtts ProjectingExprs HavingCond* (*Relation*)

  Examples:

  ```
  % Number of employees
  group_by [] count(*) true (employee);
  % Employees with a salary greater than average salary,
  % grouped by department
  group_by dept id salary > avg(salary) (employee);
  ```

4.3.2 Recursion in RA

Recursion in RA expressions can be specified by simply including the name of the view which is being defined in its definition body. Solving recursion in RA has been proposed as the application of a fixpoint operator to an RA expression (see, for instance, [Agra88, HA92]). DES compiles RA expressions to Datalog programs and uses the (fixpoint-based) deductive engine to solve them.

As an example of recursion in RA, let's consider the following classic program for finding paths in a graph:

```
create table edge(origin string, destination string);

paths(origin, destination) :=
   select true (edge)
  union
   project paths.origin, edge.destination
    (select paths.destination=edge.origin (edge product paths));

select true (paths);
```

4.3.3 RA Grammar

Here, terminal symbols are: Parentheses, commas, semicolons, single dots, asterisks, and apostrophes. Other terminal symbols are completely written in capitals, as **SELECT**. However, they are recognized by the parser in any letter case. Percentage symbols (**%**) start comments. User identifiers must start with a letter and consist of letters and numbers; otherwise, a user identifier can be enclosed between quotation marks (both square brackets and double quotes are supported) and contain any characters. Next, **RAstmt** stands for a valid RA statement.

This grammar is built following [Diet01], so that RA files read in WinRDBI (a tool described in that book) are also read in DES. DES grammar extends WinRDBI grammar in providing support also for: Theta join operator, outer join operators, duplicate elimination (distinct operator), grouping (**group_by** operator), recursive queries, and renaming operator (this avoids to resort to building new relations with the assignment operator **:=**, although it is supported, too).

```
RAstmt ::=
  SELECT WhereCondition (RArel)       % Selection (sigma)
  |
  PROJECT SelectExpressionList (RArel) % Projection (pi)
  |
  RENAME Schema (RArel)               % Renaming (rho)
  |
  DISTINCT (RArel)                    % Duplicate elimination
  |
  RArel PRODUCT RArel                 % Cartesian Product
  |
  RArel DIVISION RArel                % Division
  |
  RArel UNION RArel                   % Set union
  |
  RArel DIFFERENCE RArel              % Set difference
```

```
  |
  RArel INTERSECT RArel               % Set intersection
  |
  RArel NJOIN RArel                   % Natural join
  |
  RArel ZJOIN WhereCondition RArel    % Zeta join
  |
  RArel LJOIN WhereCondition RArel    % Left outer join
  |
  RArel RJOIN WhereCondition RArel    % Right outer join
  |
  RArel FJOIN WhereCondition RArel    % Full outer join
  |
  GROUP_BY GAtts SelectExpressionList HavingCondition (RArel)
                                      % Grouping


RArel ::=
  RAstmt
  |
  Relation

View definition (assignment statement):
RAview ::=
  Schema := [RAstmt | Relation]

Schema ::=
  ViewName
  |
  ViewName(ColName,...,ColName)

GAtts :=
  []
  |
  Atts
```

Where **Atts, Condition**, **SelectExpressionList** and **HavingCondition** are as in SQL grammar.

4.4 Prolog

Syntax of Prolog programs and goals is the same as for Datalog, including all built-in operators (cf. next Section) but aggregates. Notice that negation is written as **not(*Goal*)**, instead of the usual **\+ *Goal***.

When a goal is solved, instead of displaying the variable substitution for the answer, the goal is displayed with the substitution applied, as in:

```
DES-Prolog> t(X)
t(1)
? (type ; for more solutions, <Intro> to continue) ;
t(2)
? (type ; for more solutions, <Intro> to continue) ;
no
```

4.5 Built-ins

Most built-ins are shared by the four languages. For instance, w.r.t. comparison operators, the only difference is the less or equal (**=<**) operator used in Datalog and Prolog. This operator is different from the used in SQL and RA, which is written as **<=**. The former is written that way since in Prolog and Datalog, it is distinguished from the implication to the left operator (**<=**). SQL does not provide implications; so, the SQL syntax seems to be more appealing since the order of the two symbols matches the order of words.

Arithmetic expressions are constructed with the same built-ins in the three languages. However, in Datalog and Prolog, you need to use the infix **is** (cf. Section 4.5.2).

The built-in predicates **is_null/1** and **is_not_null/1** belong to the Datalog language.

Also, consult Section 5.3 for limitations regarding safety in the use of built-ins in Datalog.

4.5.1 Comparison Operators

All comparison operators are infix and apply to terms. For the inequality and disequality operators (greater than, less than, etc.), numbers are compared in terms of their arithmetical value; other terms are compared in Prolog standard order.

If a compound term is involved in a comparison operator, it is evaluated as an arithmetic expression and its result is then compared (for all operators by equality) or unified (for equality).

All comparison operators, but equality, demand ground arguments since they are not constraints, but test operators, and argument domains are infinite. If a ground argument is demanded and a variable is received, an exception is raised.

Next, we list the available comparison operators, where **X** and **Y** are terms (variables, constants or arithmetic expressions).

- **X = Y** (Syntactic equality)
Tests syntactic equality between **X** and **Y**. It also performs unification when variables are involved. This is the only comparison operator that does not demand ground arguments.
- **X \= Y** (Syntactic disequality)
Tests syntactic disequality between **X** and **Y**.
- **X > Y** (Greater than)
Tests whether **X** is greater than **Y**.
- **X >= Y** (Greater than or equal to)
Tests whether **X** is greater than or equal to than **Y**.
- **X < Y** (Less than)
Tests whether **X** is less than **Y**.
- **X =< Y** (Less than or equal to)
Tests whether **X** is less than or equal to **Y**.

4.5.2 Datalog and Prolog Arithmetic

Borrowed from most Prolog implementations, arithmetic is allowed by using the infix operator **`is`**, which is used to construct a query with two arguments, as follows:

`X is Expression`

where ***`X`*** is a variable or a number, and ***`Expression`*** is an arithmetic expression built from numbers, variables, built-in arithmetic operators, constants and functions, mainly following ISO for Prolog (they are labelled, if so, in the listings below). Availability of arithmetic built-ins mainly depend on the underlying Prolog system (binary distributions cope with all the listed built-ins).

At evaluation time, the expression must be ground (i.e., its variables must be bound to numbers or constants); otherwise, problems as stated in the previous section may arise. Evaluating the above query amounts to evaluate the arithmetic expression according to the usual arithmetic rules, which yields a number (integer or float), and ***`X`*** is bound to this number if it is a variable or tested its equivalence if it is a number. Precision depends on the underlying Prolog system.

Arithmetic built-ins have meaning only in the second argument of **`is`**; they cannot be used elsewhere. For example:

```
DES> X is sqrt(2)

{
  1.4142135623730951 is sqrt(2)
}
Info: 1 tuple computed.
```

Here, **`sqrt(2)`** is an arithmetic expression that uses the built-in function **`sqrt`** (square root). But:

```
DES> sqrt(2) is sqrt(2)
```

raises an input error because an arithmetic expression can only occur as the right argument of **`is`**. Another example is:

```
DES> X is e

{
  2.718281828459045 is exp(1)
}
Info: 1 tuple computed.

DES> e is e

{
}
Info: 0 tuples computed.
```

This means that the built-in arithmetic constant **`e`** cannot be used outside of an arithmetic expression, and it is otherwise understood as a user defined relation. Here, an input error is not raised since **e** could be a user defined relation. In fact, this should raise a type error, but they are not currently controlled.

In addition, note that arithmetic expressions are compound terms which are translated into an internal equivalent representation. The last example shows this since the constant **e** is translated to **exp(1)**.

Concluding, the infix (infinite) relation **is** is understood as the set of pairs **<V, E>** such that **V** is the equivalent value to the evaluation of the arithmetical expression **E**. Note that, since this relation is infinite, we may reach non-termination: Let's consider the following program (**loop.dl** in the distribution directory) with the query **loop(X)**:

```
loop(0).
loop(X) :-
  loop(Y),
  X is Y + 1.
```

Evaluating that query results in a non-terminating cycle because unlimited tuples **is(*N*,*N*+1)** become computed. To show it, try the query, press Ctrl-C, and type **listing(et)** at the Prolog prompt (only when DES has been started from a Prolog interpreter).

4.5.3 SQL Arithmetic

Arithmetic expressions are constructed with the arithmetic operators listed in the next section. They are used in projection lists and conditions.

4.5.4 Arithmetic Built-ins

This section contains the listings for the supported arithmetic operators, constants, and functions.

4.5.4.1 Arithmetic Operators

The following operators are the only ones allowed in arithmetic expressions, where **X** and **Y** stand also for arithmetic expressions.

- **\X** (Bitwise negation) *ISO*
  Bitwise negation of the integer **X**.
- **-X** (Negative value) *ISO*
  Negative value of its single argument **X**.
- **X ** Y** (Power) *ISO*
  X raised to the power of **Y**.
- **X ^ Y** (Power)
  Synonym for **X ** Y**.
- **X * Y** (Multiplication) *ISO*
  X multiplied by **Y**.
- **X / Y** (Real division) *ISO*
  Float quotient of **X** and **Y**.
- **X + Y** (Addition) *ISO*
  Sum of **X** and **Y**.
- **X - Y** (Subtraction) *ISO*
  Difference of **X** and **Y**.

- `X // Y` (Integer quotient) *ISO*

Integer quotient of `X` and `Y`. The result is always truncated towards zero.

- `X rem Y` (Integer remainder) *ISO*

The value is the integer remainder after dividing `X` by `Y`, i.e., `integer(X)-integer(Y)*(X//Y)`. The sign of a nonzero remainder will thus be the same as that of the dividend.

- `X \/ Y` (Bitwise disjunction) *ISO*

Bitwise disjunction of the integers `X` and `Y`.

- `X /\ Y` (Bitwise conjunction) *ISO*

Bitwise disjunction of the integers `X` and `Y`.

- `X xor Y` (Bitwise exclusive or) *ISO*

Bitwise exclusive or of the integers `X` and `Y`.

- `X << Y` (Shift left) *ISO*

`X` shifted left `Y` places.

- `X >> Y` (Shift right) *ISO*

`X` shifted right `Y` places.

4.5.4.2 Arithmetic Constants

- `pi` (π)

Archimedes' constant.

- `e` (Neperian number)

Neperian number.

4.5.4.3 Arithmetic Functions

- `sqrt(X)` (Square root) *ISO*

Square root of `X`.

- `log(X)` (Natural logarithm) *ISO*

Logarithm of `X` in the base of the Neperian number (`e`).

- `ln(X)` (Natural logarithm)

Synonym for `log(X)`.

- `log(X,Y)` (Logarithm)

Logarithm of `Y` in the base of `X`.

- `sin(X)` (Sine) *ISO*

Sine of `X`.

- `cos(X)` (Cosine) *ISO*

Cosine of `X`.

- `tan(X)` (Tangent) *ISO*

Tangent of `X`.

- `cot(X)` (Cotangent)

Cotangent of `X`.

- `asin(X)` (Arc sine)

Arc sine of `X`.

- `acos(X)` (Arc cosine)

Arc cosine of `X`.

- `atan(X)` (Arc tangent) *ISO*

Arc tangent of `X`.

- `acot(X)` (Arc cotangent)

Arc cotangent of `X`.

- `abs(X)` (Absolute value) *ISO*

Absolute value of **X**.

- **float(X)** (Float value) *ISO*

Float equivalent of **X**, if **X** is an integer; otherwise, **X** itself.

- **integer(X)** (Integer value)

Closest integer between **X** and 0, if **X** is a float; otherwise, **X** itself.

- **sign(X)** (Sign) *ISO*

Sign of **X**, i.e., -1, if **X** is negative, 0, if **X** is zero, and 1, if **X** is positive, coerced into the same type as **X** (i.e., the result is an integer, iff **X** is an integer).

- **gcd(X,Y)** (Greatest common divisor)

Greatest common divisor of the two integers **X** and **Y**.

- **min(X,Y)** (Minimum)

Least value of **X** and **Y**.

- **max(X,Y)** (Maximum)

Greatest value of **X** and **Y**.

- **truncate(X)** (Truncate) *ISO*

Closest integer between **X** and 0.

- **float_integer_part(X)** (Integer part as a float) *ISO*

The same as **float(integer(X))**.

- **float_fractional_part(X)** (Fractional part as a float) *ISO*

Fractional part of **X**, i.e., **X - float_integer_part(X)**.

- **round(X)** (Closest integer) *ISO*

Closest integer to **X**. **X** has to be a float. If **X** is exactly half-way between two integers, it is rounded up (i.e., the value is the least integer greater than **X**).

- **floor(X)** (Floor) *ISO*

Greatest integer less or equal to **X**. **X** has to be a float.

- **ceiling(X)** (Ceiling) *ISO*

Least integer greater or equal to **X**. **X** has to be a float.

4.5.5 Negation

- **not(*Query*)** (Stratified negation)

It stands for the complement of the relation ***Query*** w.r.t. the meaning of the program (i.e., closed world assumption). See Sections 4.1.8 and 5.17.3. If ***Query*** is not an atom, a new predicate defined by a head ***Head*** with relevant variables in ***Query*** is built, and defined by the single rule ***Head* :- *Query***. Then, **not(*Head*)** replaces **not(*Query*)**.

4.5.6 Datalog Outer Joins

- **lj(*LeftRelation*,*RightRelation*,*JoinCondition*)** (Left join)

It stands for the left outer join of the relations ***LeftRelation*** and relations ***RightRelation***, under the condition ***JoinCondition*** (expressed as literals, cf. Section 4.1.1), as understood in extended relational algebra (***LeftRelation*** ⟕ ***JoinCondition*** ***RightRelation***).

- **rj(*LeftRelation*,*RightRelation*,*JoinCondition*)** (Right join)

It stands for the right outer join of the relations ***LeftRelation*** and relations ***RightRelation***, under the condition ***JoinCondition*** (expressed as literals, cf. Section 4.1.1), as understood in extended relational algebra (***LeftRelation*** ⟖ ***JoinCondition*** ***RightRelation***).

- **fj(*LeftRelation*,*RightRelation*,*JoinCondition*)** (Full join)

It stands for the full outer join of the relations ***`LeftRelation`*** and relations ***`RightRelation`***, under the condition ***`JoinCondition`*** (expressed as literals, cf. Section 4.1.1), as understood in extended relational algebra (***`LeftRelation`*** ⟗ ***`JoinCondition`*** ***`RightRelation`***).

4.5.7 Datalog Aggregates

4.5.7.1 Aggregate Functions

Aggregate functions can only occur in the context of a **`group_by`** aggregate predicate (see next section) and apply to the result set for its input relation.

- **`count(`*`Variable`*`)`**
  Return the number of tuples so that the value for **`Variable`** is not null.
- **`count`**
  Return the number of tuples of the result set.
- **`sum(`*`Variable`*`)`**
  Return the sum of possible values for **`Variable`**, ignoring nulls.
- **`times(`*`Variable`*`)`**
  Return the product of possible values for **`Variable`**, ignoring nulls.
- **`avg(`*`Variable`*`)`**
  Return the average of possible values for **`Variable`**, ignoring nulls.
- **`min(`*`Variable`*`)`**
  Return the minimum value for **`Variable`**, ignoring nulls.
- **`max(`*`Variable`*`)`**
  Return the maximum value for **`Variable`**, ignoring nulls.

4.5.7.2 Group_by Predicate

- **`group_by(`*`Query,Variables,GroupConditions`*`)`**
  Solve ***`GroupConditions`*** in the context of ***`Query`***, building groups w.r.t. the possible values the variables in the list ***`Variables`***. This list is specified as a Prolog list, i.e., a sequence of comma-separated values enclosed between brackets. If this list is empty, there is only one group: the answer set for ***`Query`***. The goal ***`GroupConditions`*** may contain expressions including aggregate functions.

4.5.7.3 Aggregate Predicates

- **`count(`*`Query,Variable,Result`*`)`**
  Count in ***`Result`*** the number of tuples in the result set for the query ***`Query`*** so that ***`Variable`*** is a variable of ***`Query`*** (an attribute of the result relation set) and this attribute is not null. It returns 0 if no tuples are found in the result set.
- **`count(`*`Query,Result`*`)`**
  Count in ***`Result`*** the total number of tuples in the result set for the query ***`Query`***, disregarding whether they contain nulls or not. It returns 0 if no tuples are found in the result set.
- **`sum(`*`Query,Variable,Result`*`)`**
  Sum in ***`Result`*** the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored.
- **`times(`*`Query,Variable,Result`*`)`**
  Compute in **`Result`** the product of all the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored.
- **`avg(`*`Query,Variable,Result`*`)`**

Compute in **`Result`** the average of the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored.

- **`min(`*`Query,Variable,Result`*`)`**

Compute in ***`Result`*** the minimum of the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored. If there are no such numbers, it returns **`null`**.

- **`max(`*`Query,Variable,Result`*`)`**

Compute in ***`Result`*** the maximum of the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored. If there are no such numbers, it returns **`null`**.

4.5.8 Datalog Null-related Predicates

- **`is_null(`*`Term`*`)`**

Succeed if ***`Term`*** is bound to a null value. It raises an exception if ***`Term`*** is a variable.

- **`is_not_null(`*`Term`*`)`**

Succeed if ***`Term`*** is not bound to a null value. It raises an exception if ***`Term`*** is a variable.

4.5.9 Duplicates

The following built-ins take effect when duplicates are enabled via the command **`/duplicates on`**.

- **`distinct(`*`Query`*`)`**

Succeed as many times as different ground answers are computed for ***`Query`***.

- **`distinct([`*`Variables`*`], `*`Query`*`)`**

Succeed as many times as different ground tuples (built with ***`Variables`***) are computed for ***`Query`***.

4.5.10 Top-N Queries

- **`top(`*`N`*`,`*`Query`*`)`**

Succeed at most ***`N`*** times for ***`Query`***.

5. System Description

This section includes descriptions about the connection to relational database systems via ODBC connections, persistency, safety and computability issues, source-to-source transformations, the declarative debuggers and tracers, the batch processing, system messages, and finally lists all the available commands.

5.1 RDBMS connections via ODBC

DES provides support for connections to (relational) database management systems (RDBMSs) in order to provide data sources for relations. This means that a relation defined in a RDBMS as a view or table is allowed as any other relation defined via a predicate in the deductive database. Then, computing a query can involve computations both in the deductive inference engine and in the external RDBMS SQL engine. Such relations become first-class citizens in the deductive database and, therefore, can be queried in Datalog and RA. If the relation is a view, it will be processed by the SQL engine. When an ODBC connection is opened, all SQL statements are redirected to such connection, so DES does not longer process such statements. This means that all the SQL features of the connected RDBMS are available.

Almost any relational database (RDB) can be accessed from DES using an ODBC connection. Relational database management system (RDBMS) manufacturers provide ODBC implementations which run on many operating systems (Microsoft Windows, Linux, Mac OS X, ...) RDBMSs include enterprise RDBMS (as Oracle, MySQL, DB2, ...) and desktop RDBMS (as MS Access and FileMaker).

ODBC drivers are usually bundled with OS platforms, as Windows OSs (ODBC implementation), Linux OS distributions as Ubuntu, Red Hat and Mandriva (UnixODBC implementation), and Mac OSs 10*x* (iODBC implementation). However, additional drivers for specific databases are needed to be installed.

Since each RDBMS provides an ODBC driver and each OS an ODBC implementation, details on how to configure such connections are out of the scope of this manual. However, to configure such a connection, typically, the ODBC driver is looked for and installed in the OS. Then, following the manufacturer recommendations, it is configured. You can find many web pages with advice on this. Here, we assume that there are ODBC connections already available.

5.1.1 Opening an ODBC Connection

To access a RDB in DES, first open the connection with the following command, where **`test`** is the name of a previously created ODBC connection to a database:

```
DES-SQL> /open_db test
```

You can also provide a user name and password (if needed) as in:

```
DES-SQL> /open_db test user(smith) password(my_pwd)
```

If ODBC connector returns an error, then you have to enclose these between apostrophes (`'`) as in:

```
DES-SQL> /open_db test user('smith') password('my_pwd')
```

Note that if you have previously created some database objects (tables, views, ...) in DES without an ODBC connection, they are still available and can be queried too (for more information see Section 5.1.7).

5.1.2 Using a Connection

Assuming that the connection links to an empty database, let's start creating some database objects:

(Note that, depending on the installed MySQL ODBC driver version, annoying messages might be displayed.)

```
DES-SQL> create table t(a varchar(20) primary key)
DES-SQL> create table s(a varchar(20) primary key)
DES-SQL> create view v(a,b) as select * from t,s
DES-SQL> insert into t values(1)
Info: 1 tuple inserted.
DES-SQL> insert into s select * from t
```

```
Info: 1 tuple inserted.
DES-SQL> insert into s values(2)
Info: 1 tuple inserted.
```

Next, one can ask for the database schema (metadata) with the command:

```
DES-SQL> /dbschema
Info: Table(s):
 * s(a:varchar)
 * t(a:varchar)
Info: View(s):
 * v(a:varchar,b:varchar)
```

All of these tables and views can be accessed from DES, as if they were local:

```
DES-SQL> select * from s;
answer(a:varchar) ->
{
  answer('1'),
  answer('2')
}
Info: 2 tuples computed.

DES-SQL> select * from t;
answer(a:varchar) ->
{
  answer('1')
}
Info: 1 tuple computed.

DES-SQL> select * from v;
answer(a:varchar,b:varchar) ->
{
  answer('1','1'),
  answer('1','2')
}
Info: 2 tuples computed.

DES-SQL> insert into t values('1')
Exception: error(odbc(23000,1062,[MySQL][ODBC 3.51
Driver][mysqld-5.0.41-community-nt]Duplicate entry '1' for key
1),_G3)
```

In this example, as table **t** has its single column defined as its primary key, trying to insert a duplicate entry results in an exception from the ODBC driver. Integrity constraints are handled by the RDBMS connected, instead of DES (notice that the exception message is different from the one generated by DES).

Moreover, you can submit SQL statements that are not supported by DES but otherwise by the connected RDBMS, as:

```
DES-SQL> alter table t drop primary key;
```

Then, you can insert again and see the result (including duplicates):

```
DES-SQL> insert into t values('1')
```

```
Info: 1 tuple inserted.

DES-SQL> select * from v;
answer(a:varchar,b:varchar) ->
{
  answer('1','1'),
  answer('1','1'),
  answer('1','2'),
  answer('1','2')
}
Info: 4 tuples computed.
```

Also, duplicate removing is also possible by the external RDBMS:

```
DES-SQL> select distinct * from v;
answer(a:varchar,b:varchar) ->
{
  answer('1','1'),
  answer('1','2')
}
Info: 2 tuples computed.
```

Nonetheless, these external objects can be accessed from Datalog as well (please remember to enable duplicates to get the expected result):

```
DES-SQL> /datalog
DES> /duplicates on
Info: Duplicates are on.
DES> s(X),t(X)
Info: Processing:
  answer(X) :-
    s(X),
    t(X).
{
  answer('1'),
  answer('1')
}
Info: 2 tuples computed.
```

This is equivalent to the following SQL statement:

```
DES> select s.a from s,t where s.a=t.a
answer(a:varchar) ->
{
  answer('1'),
  answer('1')
}
Info: 2 tuples computed.
```

However, whilst the former has been processed by the Datalog engine, the latter has been processed by the external RDBMS. So, some complex SQL statements might be more efficiently processed by the external RDBMS.

Duplicates are relevant in a number of situations. For instance, consider the following, where duplicates are initially disabled:

```
DES> group_by(v(X,Y),[X,Y],C=count)
Info: Processing:
  answer(X,Y,C) :-
    group_by(v(X,Y),[X,Y],C = count).
{
  answer('1','1',1),
  answer('1','2',1)
}
Info: 2 tuples computed.
```

Although there are a couple of tuples for each group (see the table contents above), only one is returned in the count because they are indistinguishable in a set. Now, if duplicates are allowed, we get the expected result:

```
DES> /duplicates on
Info: Duplicates are on.

DES> group_by(v(X,Y),[X,Y],C=count)
Info: Processing:
  answer(X,Y,C) :-
    group_by(v(X,Y),[X,Y],C = count).
{
  answer('1','1',2),
  answer('1','2',2)
}
Info: 2 tuples computed.
```

Note that, even when you can access SQL objects from Datalog, the contrary is not allowed because there is nor Datalog metadata information for the external SQL engine, neither access to Datalog data. The data bridge is only opened from DES to the external DBMS, not the other way round. This is in contrast to the SQL database internally provided by DES, which allows a bidirectional communication since type information is supported for Datalog predicates.

5.1.3 Opening Several Connections

From release 3.0 on, several OCBC connections can be opened simultaneously. Each time a new connection is opened, it becomes the new current connection, and all query processing is related to it by default. For instance, to inspect (a rather limited set of) metadata, one can submit the following command:

```
DES> /open_db mysql
DES> /dbschema
Info: Database 'mysql'
Info: Table(s):
 * s(a:varchar(20))
 * t(a:integer(4))
 * w(a:varchar(20))
Info: View(s):
 * v(a:varbinary(20))
Info: No integrity constraints.
```

To list all the opened connections, use the command:

```
DES> /show_dbs
```

```
$des
access
csv
db2
excel
mysql
oracle
postgresql
sqlserver
```

where you can see the list of opened connections, starting with **$des**, which is the default database (DES deductive engine). You can close all connections but the default one. As the names suggest, you can open a wide range of data sources, not only from database management systems as DB2, Oracle, SQL Server but also from other sources as datasheets (Excel) and text files (CSV (comma-separated values) files). For defining a "table" in MS Excel, you should use Insert -> Name -> Define, where you specify the name of the table and the cell range it covers (where first row can be used as field names, optionally). Types are inferred by the Excel system. Similarly, when defining a connection to a text file, field names can be those in the first line of explicitly given. Again, types are inferred. In both cases, you can inspect the "database" schema and query them with either SQL statements, or Datalog queries or RA expressions.

Note that some data sources do neither creating views nor constraints, such as datasheets and text files.

A warning for newbies: You have to define connection names following ODBC installation; do not expect the ones listed above are provided by default, you need both the ODBC connection and the data provider (database server or whatever) already installed and configured.

5.1.4 Current Connection

To find out the current opened ODBC database, use the command:

```
DES-SQL> /current_db
```

5.1.5 Making a Connection the Current One

Making a given connection the current one is simply done with:

```
DES-SQL> /use_db access
```

where **access** is an example of an already opened connection name.

5.1.6 Closing a Connection

Closing the current connection is simply done with:

```
DES-SQL> /close_db
```

You can also specify to close a given connection, as in:

```
DES-SQL> /close_db access
```

5.1.7 Schema and Data Visibility

Any submitted query or command refer to the current connection if not otherwise specified as an argument of a command. When opening a connection (and automatically making it the current one), their data and schema are visible, but not the data and schema of other already opened connections. In contrast, data from the default deductive database are visible for Datalog and RA queries, although its schema does not. Recall that you can create tables and views in the default database, which will be handled by DES but not projected to any external database (unless you persist a predicate; see Section 5.2). Anyway, data from the default deductive database (**`$des`**) are *not* visible for SQL statements for a current connection other than **`$des`**, as they are submitted for processing to the external database.

In the following system session, one creates a table in the default database of DES (DDB), inserts a value, opens a connection, and realize that the table schema is not visible, but its data do. This comes from the fact that, first, SQL data is translated by DES to Datalog data and, second, Datalog data can be seamlessly combined with external databases (EDB).

```
DES> create table t(a int)                 % Create table t in DDB
DES> insert into t values(1)               % Insert t(1) in DDB
Info: 1 tuple inserted.
DES> select * from t                       % Select data from DDB
answer(t.a:number(integer)) ->
{
  answer(1)
}
Info: 1 tuple computed.
DES> /open_db mysql                        % Open an EDB
DES> select * from t                       % Select data from EDB
Error: ODBC Code (1146):                   % As t is not defined in EDB,
[MySQL][ODBC 5.1 Driver][mysqld-           % then, error
5.5.9]Table 'test.t' doesn't exist
DES> t(X)                                  % Predicate t is known to DDB and
{                                          % can be queried from Datalog
  t(1)
}
Info: 1 tuple computed.
```

In this way, you can also combine data from DES and the external data source. Next system session example shows this by creating a new table in the external database and combining above predicate **`t/1`**, defined in DDB, with a new table **`s`** created in EDB:

```
DES> create table s(a int)                 % Create table s in EDB
DES> insert into s values(2)               % Insert s(2) in EDB
Info: 1 tuple inserted.
```

```
DES> select * from s                    % Select data from EDB.
answer(a:integer(4)) ->                 % Note the different type w.r.t. DDB
{
  answer(2)
}
Info: 1 tuple computed.
DES> t(X),s(Y)                          % Join t/1 (DDB) with s/1 (EDB)
Info: Processing:
  answer(X,Y) :-
    t(X),
    s(Y).
{
  answer(1,2)
}
Info: 1 tuple computed.
```

5.1.8 Solving Engine and ODBC connections

When the current database is an open ODBC connection, any statement is submitted to the external database for its solving by default. However, this behavior can be changed by forcing DES to solve SQL DQL queries submitted to an external database. This allows to experiment with more expressive forms of SQL queries as allowed by the local deductive engine, as hypothetical queries, non-linear and mutually recursive queries.

To force a single SQL DQL query to be processed by DES, simply use the command **`/des`** followed by the query. Note however that DML and DDL queries are still sent to the external DBMS. Let's consider MySQL, which does not support recursive queries up to its current version 5.5. If we had available the table **`edge(a int, b int)`**, we can compute its transitive closure as follows:

```
DES> /open_db mysql
DES> select * from edge
answer(a:integer(4),b:integer(4)) ->
{
  answer(1,2),
  answer(2,3),
  answer(3,4)
}
Info: 3 tuples computed.
DES> /des assume select e1.a,e2.b from edge e1, edge e2 where
e1.b=e2.a in edge(a,b) select * from edge
answer(edge.a:number(integer),edge.b:number(integer)) ->
{
  answer(1,2),
  answer(1,3),
  answer(1,4),
  answer(2,3),
  answer(2,4),
  answer(3,4)
}
Info: 6 tuples computed.
```

5.1.9 Integrity Constraints, ODBC Connections, and Persistency

Integrity constraints as described in Section 4.1.15 are monitored by DES for the local deductive database. This means that inserting values directly into external tables (either by submitting an **INSERT INTO** statement from the opened connection or by inserting values out of DES) is not monitored for constraint consistency. However, as constraint consistency checking considers all visible data, when asserting into the local database, data from the current opened connection is also taken into account. The following system session shows a possible scenario illustrating these situations:

```
DES> /use_db $des
DES> create or replace table t(a int primary key)
DES> /dbschema
Info: Database '$des'
Info: Table(s):
 * t(a:number(integer))
    - PK: [a]
Info: No views.
Info: No integrity constraints.
DES> /open_db mysql
```

Table '**t**' is also an external table in connection '**mysql**':

```
DES> /dbschema t
Info: Database 'mysql'
Info: Table:
 * t(a:integer(4))
```

Retrieve tuples from external table '**t**':

```
DES> select * from t
answer(a:integer(4)) ->
{
}
Info: 0 tuples computed.
```

The following is inserted in external table '**t**'. Recall that SQL statements under an opened connection are submitted directly to the external RDBMS:

```
DES> insert into t values (1)
Info: 1 tuple inserted.
DES> insert into t values (1) % Not rejected as it is not
monitored by DES
Info: 1 tuple inserted.
```

DES does monitor the following assertion as it is directed to the local database:

```
DES> /assert t(1)
Error: Primary key violation t.[a]
       when trying to insert: t(1)
Error: Asserting rules due to integrity constraint violation.
DES> /use_db $des
```

When the current database is the local database ('**$des**'), the external table '**t**' is not visible. So, the following fact is asserted in the local database:

```
DES> insert into t values (1)
Info: 1 tuple inserted.
```

Any other attempt to assert the same fact **t(1)** is rejected

```
DES> /assert t(1)
Error: Primary key violation t.[a]
       when trying to insert: t(1)
Error: Asserting rules due to integrity constraint violation.
```

The following would also go to the local database:

```
DES> insert into t values (1)
Error: Primary key violation t.[a]
       when trying to insert: t(1)
Error: Asserting rules due to integrity constraint violation.
Info: 0 tuples inserted.
```

Finally, any persisted predicate (see Section 5.2) which has attached constraints is checked for its consistency, irrespective of the external database it is stored. Also, any of the supported constraints can be attached to persistent predicates, therefore providing a high expressivity and declarative consistency level.

5.1.10 Caveats and Limitations

This section lists some caveats and limitations of the current implementation of ODBC connections to external data sources.

5.1.10.1 Caching

Data in relational tables are cached in the memo table during Datalog computations, and it is not requested anymore until this cache is cleared (either explicitly with the command **/clear_et** or because a command or statement invalidating its contents, as an SQL update query). Therefore, it could be possible to access outdated data from a Datalog query. Let's consider:

```
DES-SQL> /datalog t(X)
{
  t('1')
}
Info: 1 tuple computed.
```

Then, from the MySQL client:

```
mysql> insert into t values('2');
Query OK, 1 row affected (0.06 sec)
```

And, after, in DES, the new tuple is not listed via a Datalog query:

```
DES-SQL> /datalog t(X)
{
  t('1')
}
Info: 1 tuple computed.
```

However, an SQL statement returns the correct answer:

```
DES-SQL> select * from t;
answer(a:varchar) ->
{
  answer('1'),
  answer('2')
}
Info: 2 tuples computed.
```

In addition, it is not recommended to mix Datalog and SQL data. It is possible to assert tuples with the same name and arity as existing RDBMS's tables and/or views. Let's consider the same table t as above with the same data (two tuples **`t('1')`** and **`t('2')`**) and assert a tuple **`t('3')`** as follows:

```
DES-SQL> /assert t('3')

DES-SQL> /datalog t(X)
{
  t('1'),
  t('2'),
  t('3')
}
Info: 3 tuples computed.

DES-SQL> select * from t
answer(a:varchar) ->
{
  answer('1'),
  answer('2')
}
Info: 2 tuples computed.
```

This reveals that, although on the DES side, Datalog data are known, it is not on the RDBMS side. This is in contrast to the DES management of data: if no ODBC connection is opened, the DES engine is aware of any changes to data, both from Datalog and SQL sides.

Concluding, those updates that are external to DES might not be noticed by the DES engine. And, also, an ODBC connection should be seen as a source of external data that should not be mixed with Datalog data. However, you can safely use the more powerful Datalog language to query external data (and to be sure the current data is retrieved, clear the cache with **`/clear_et`**).

5.1.10.2 ODBC Metadata

When computing the predicate dependency graph and stratification, metadata from the external DBMS is retrieved, which can be a costly operation if the number of tables and views is large. This is the default case when opening connections to DBMSs as SQL Server or Oracle, where many views are defined for an empty database. Also, ODBC connections to Oracle seem to be slow.

Listing the database schema can suffer this situation as well, by issuing the command **`/dbschema`**. Instead, it is better to focus on the required object to display, as either **`/dbschema`** ***`relname`*** or **`/dbschema`** ***`connection`*****`:`*****`relname`***.

5.1.10.3 ODBC Limitations

As predicate dependency graphs are not computed from external data sources, several features are not supported in the context of an opened ODBC connection:

- SQL tracer
- Test case generator

5.1.10.4 Platform-specific Issues

ODBC connections are only supported by the provided binaries, and the source distributions for SWI-Prolog and SICStus Prolog.

If you use a 64 bit Windows OS, notice that you can select to run either a 64 bit version of DES or a 32 bit one (binaries built with SWI-Prolog are provided in the download area). In the first case (64 bit), you must use the Database Connectivity (ODBC) Data Source Administrator tool (Odbcad32.exe):

- The 32-bit version of the Odbcad32.exe file is located in the %systemdrive%Windows%SysWoW64 folder.
- The 64-bit version of the Odbcad32.exe file is located in the %systemdrive%Windows%System32 folder.

Also notice that a 64 bit driver requires also a 64 bit database installation. For instance, you can define a 32 bit ODBC connection to 32 bit MS Access installation and a 64 bit ODBC connection to a 64 bit Oracle installation. In this scenario, both connectinos cannot be opened from the same DES instance (which is either a 32 bit or 64 bit release).

5.1.11 Tested ODBC Drivers

Several data sources have been successfully tested on Windows XP/Vista/7 32 bit with both SICStus Prolog and SWI-Prolog executables and sources:

- IBM DB2 v9.7.200.358
- Oracle Database Express Edition 11g Release 2 (also tested with Windows 7 64 bit and SWI-Prolog 6.0.0 64 bit)
- SQL Server Express 2008 (including spatial components)
- MySQL 5.5.9
- PostgreSQL 9.1.3
- Access 2003
- Excel 2003
- CSV text files

5.2 Persistency

Since DES 3.0, it is possible to make predicates persist on either an external database, or datasheet or text file, i.e., any data source supported by an ODBC connection. This sections describes how to persist a predicate, use it, examine its schema, unpersist it, and also lists a couple of caveats.

5.2.1 Persisting a Predicate

An assertion is used to declare a persisted predicate, as in:

```
DES> :-persistent(p(a:int),mysql)
```

where its first argument is the predicate and its schema, and the second one is the ODBC connection name. This name can be omitted if the current connection is the one you want to use to persist the predicate, as in:

```
DES> /current_db
Info: Current database is 'mysql'. DBMS: mysql
DES> :-persistent(p(a:int))
```

You can confirm that predicate **p** has been declared as persistent with:

```
DES> /list_persistent
mysql:p(a:number(integer))
```

where the connection name is shown, followed by a semicolon and the predicate schema.

Also, if you have type information declared already, you can simply refer to the predicate with its name and arity in the persistency assertion:

```
DES> /use_db $des
DES> create table p(a int)
DES> /use_db mysql
DES> :-persistent(p/1)
DES> /list_persistent
mysql:p(a:number(integer))
```

The general form of a persistency assertion is as follows:

:-persistent(*PredSpec*[,*Connection*]))

This assertion makes a predicate to persist on an external RDBMS via an ODBC connection. ***PredSpec*** can be either the pattern ***PredName/Arity*** or ***PredName*(*Schema*)**, where ***Schema*** can be either ***ArgName1*, …, *ArgNameN*** or ***ArgName1*:*Type1*, …, *ArgNameN*:*TypeN***. If a connection name is not provided, the current open database is used. The local, default database **$des** cannot be used to persist, but an ODBC connection.

5.2.2 Using Persistent Predicates

You can assert facts as usual and query the persisted predicate **p**/1 as the following example shows:

```
DES> /assert p(1)
DES> p(X)
{
  p(1)
}
Info: 1 tuple computed.
```

And, as expected, it can seamlessly be combined with other non-persistent predicates, as in:

```
DES> /assert q(2)
DES> p(X),q(Y),X<Y
Info: Processing:
  answer(X,Y) :-
    p(X),
    q(Y),
    X < Y.
{
  answer(1,2)
}
Info: 1 tuple computed.
```

where **q(2)** is in the meaning of **q**/1.

Also, you can use SQL or RA languages to query such persistent predicates, as in:

```
DES> :-type(q(a:int))
DES> select * from p,q where p.a<q.a
answer(p.a:number(integer),q.a:number(integer)) ->
{
  answer(1,2)
}
Info: 1 tuple computed.
DES> p zjoin p.a<q.a q
answer(p.a:number(integer),q.a:number(integer)) ->
{
  answer(1,2)
}
Info: 1 tuple computed.
```

And persistent predicates can be combined even with external data coming from other ODBC connection, as in:

```
DES> /open_db access
DES> /dbschema t
Info: Database 'access'
Info: Table:
 * t(a:INTEGER(4))
DES> p(X),t(X)
Info: Processing:
  answer(X) :-
    p(X),
    t(X).
{
  answer(1)
}
Info: 1 tuple computed.
```

Here, the current database is **access** and all its data is available (as already introduced in Section 5.1.2); in particular, the table **t**, which contains in particular the tuple **t(1)**.

As well, one can retract the rules previously asserted. For instance:

```
DES> /retract p(1)
DES> /retract p(X):-r(X)
```

5.2.3 Processing a Persistency Assertion

Processing a persistency assertion means to make persistent a predicate, i.e., all of its current rules as well as rules added afterwards are stored in a persistent media, as a relational database. A fact is projected to a table whereas a rule is translated into an SQL view. Each persisted predicate is translated into a table for holding such facts and a view which is the union of all the SQL translations for its rules. Translating rules into SQL views includes an adaptation of Draxler's Prolog to SQL compiler [Drax92].

Any rule belonging to the definition of a predicate ***`pred`*** which is being made persistent is expected, in general, to involve calls to other predicates. Each callee (such other called predicate) can be:

- An existing relation in the external database.
- An already persisted predicate which is loaded in the local database.
- An already persisted predicate which is not yet loaded in the local database.
- A predicate which has not been made persistent yet.

For the first two cases, besides making ***`pred`*** persistent, nothing else is performed when processing its persistency assertion.

For the third case, a persistent predicate is automatically restored in the local database (c.f. next section), i.e., it is made available to the deductive engine.

For the fourth case, each non-persistent predicate is automatically made persistent, if possible, inferring its types. This is needed in order for the external database to be aware of a predicate which is only known by the deductive engine so far, as this database will eventually compute ***`pred`***.

However, not all rules can be made persistent for a number of reasons: including that the external database does not support some features, and the translations of some built-ins are not supported yet. In the current state of the implementation, the following conditions must hold for a rule to be made persistent:

- The rule does not contain calls to built-ins but comparison operators.
- The rule does not form a recursive cycle.

Nonetheless, the rule is kept in the in-memory database for computing the meaning of the predicate when needed. This is performed by the deductive engine, which couples the processing of the external database with its own processing to derive the meaning of the predicate. Therefore, all the deductive computing power is preserved although the external persistent media lacks some features as, for instance, recursion (think of MySQL and MS Access). Anyway, such rules which are not projected to the external database are stored on it as metadata information. This is needed to restore the complete definition of a persistent predicate upon restoring (c.f. next section). Further releases might contain relaxed conditions.

Any time a predicate is made persistent, its associated connection is opened if it not was opened already (the current connection is not changed, anyway). The connection is not closed even when you drop the assertion (see Section 5.2.6).

5.2.4 Restoring a Session

As expected, if you make a predicate persistent and quit DES, in a next session you can recover the state of this predicate. It is simply done by submitting again the same assertion as used to make the predicate persist for the first time.

However, note that any rule in the in-memory database for such a predicate will be persisted, too. This is to say that, for instance, if you have persisted already a predicate which is not loaded already, and you have a rule asserted a rule for this predicate, then the result of restoring its persistency is the union of the asserted rule and the rules in the external database. For instance, let's consider the following system session:

```
DES> :-persistent(p(a:int),mysql)
DES> /assert p(1)
```

Now, let's assume another system session (quit and restart DES):

```
DES> /assert p(2)
DES> :-persistent(p(a:int),mysql)
Info: Recovering existing data from external database for 'p'...
DES> /listing
p(1).
p(2).
Info: 2 rules listed.
```

As it can be seen, the resulting database is composed of the union of the external rules and the local rules.

Finally, restoring compiled rules in a different system session does not recover source rules as they were originally asserted. They are only recovered "as is" (i.e., compiled form and without textual variable names as they were originally typed) in the same system session. Let's consider the following:

```
DES> :-persistent(p(a:int),mysql)
DES> /assert p(X):-X=1;X=2
DES> /listing
p(X) :-
  X = 1
  ;
  X = 2.
Info: 1 rule listed.
DES> /drop_assertion :-persistent(p(a:int),mysql)
DES> /listing
p(X) :-
  X = 1
  ;
  X = 2.
Info: 1 rule listed.
DES> :-persistent(p(a:int),mysql)
DES> /listing
p(X) :-
  X = 1
  ;
  X = 2.
```

```
Info: 1 rule listed.
DES> /quit
```

Then, we open a new system session and type:

```
DES> :-persistent(p(a:int),mysql)
Info: Recovering existing data from external database...
DES> /listing
p(A) :-
  A = 2.
p(A) :-
  A = 1.
Info: 2 rules listed.
```

As can be seen, two rules are the result of the compilation of the originally asserted single rule with a disjunctive body. Also original variable names (only **X** in tnis case) are missing. However, a next release of DES might deal with this, allowing to restore the very same rules as the original ones.

5.2.5 Schema of Persistent Predicates

You can request the current database schema with:

```
DES> /dbschema
Info: Database '$des'
Info: No tables.
Info: View(s):
 * p(a:number(integer))
    - Defining SQL statement:
        CREATE VIEW p(a) AS
          SELECT ALL *
          FROM
            p_des_table;
    - Datalog equivalent rules:
Info: No integrity constraints.
```

where the persisted predicate is listed in the database schema of the default database **$des** and, therefore, it can be combined in a query with any predicate visible in this database.

Note that predicate **p** has been declared as a view depending on a table (with the same name as the predicate and view, but ending with "**_des_table**"). Since predicates are defined in general with intensional rules, the view **p** will contain those intensional rules whereas the table will contain the extensional rules (facts). For instance, assuming that the predicate **r** has been made persisted already in the same connection, we assert an intensional rule for **p**, and examine its schema:

```
DES> /assert p(X):-r(X)
DES> /dbschema p
Info: Database '$des'
Info: View:
 * p(a:number(integer))
    - Defining SQL statement:
        CREATE VIEW p(a) AS
          (
```

```
          SELECT ALL *
          FROM
            p_des_table
        )
        UNION ALL
        (
          SELECT ALL rel1.a
          FROM
            r AS rel1
        );
  - Datalog equivalent rules:
      p(1).
      p(2).
      p(X) :-
        r(X).
```

If you change the current database to the external one and request the schema for **p**, you get:

```
DES> /use_db mysql
DES> /dbschema p
Info: Database 'mysql'
Info: View:
 * p(a:integer(4))
```

which is the schema of view **p** as provided by the external database system. Now, the detailed metadata information supplied by **$des** is not available in the external database.

Also note that the above couple of commands can be simply written as a single one without resorting to change the current database, with:

```
DES> /dbschema mysql:p
```

5.2.6 Removing Predicate Persistency

Finally, one can unpersist a given predicate by simply dropping its assertion, as in:

```
DES> /drop_assertion :-persistent(p(a:int),mysql)
```

This retrieves all the data stored in the external database and stores it back in the in-memory database of DES. In addition to the view **p** and table **p_des_table** created in the external database for **p**, there is also a table **p_des_metadata** holding the Datalog intensional rules that have been made persistent. This is needed to recover the original rules as they were asserted (in its compiled Datalog form).

If you have persisted a predicate for which no type constraints has been given before, a type constraint is derived, if possible, and asserted. This type constraint remains even when the persistency assertion is removed. If you want to remove this too, then submit a **/drop_ic** command. The following session illustrates this:

```
DES> /dbschema
Info: Database '$des'
Info: No tables.
```

```
Info: No views.
Info: No integrity constraints.
DES> :-persistent(p(a:int),mysql)
DES> /dbschema
Info: Database '$des'
Info: No tables.
Info: View(s):
 * p(a:number(integer))
    - Defining SQL statement:
        CREATE VIEW p(a) AS
          SELECT ALL *
          FROM
            p_des_table;
Info: No integrity constraints.
DES> /drop_assertion :-persistent(p(a:int),mysql)
DES> /dbschema
Info: Database '$des'
Info: Table(s):
 * p(a:number(integer))
Info: No views.
Info: No integrity constraints.
DES> /drop_ic :-type(p(a:int))
DES> /dbschema
Info: Database '$des'
Info: No tables.
Info: No views.
Info: No integrity constraints.
```

If you want to completely remove a predicate, even its persistent representation, you can use the command **/abolish**, as in:

```
DES> /abolish p
DES> /dbschema
Info: Database '$des'
Info: No tables.
Info: No views.
Info: No integrity constraints.
DES> /listing p
Info: 0 rules listed.
DES> /use_db mysql
DES> /dbschema mysql:p
Info: Database 'mysql'
Error: No table or view found with name 'p'.
```

Also, dropping the SQL view corresponding to a predicate removes persistency, as in:

```
DES> :-persistent(t(a:int),mysql)
DES> /dbschema
Info: Database '$des'
Info: No tables.
Info: View(s):
 * t(a:number(integer))
    - Defining SQL statement:
        CREATE VIEW t(a) AS
```

```
          SELECT ALL *
          FROM
            t_des_table;
Info: No integrity constraints.
DES> drop view t
DES> /dbschema
Info: Database '$des'
Info: No tables.
Info: No views.
Info: No integrity constraints.
```

5.2.7 Schema and Data Visibility

The default database (DDB) is called **$des**, and it contains metadata of each predicate for which either a type assertion or an SQL table creation statement has been issued. If one makes a predicate persistent in an external database (EDB), its metadata as well as its data is visible both to DDB and EDB. The following session illustrates this:

```
DES> /use_db $des
DES> :-persistent(p(a:int),mysql)
DES> /assert p(1)
DES> /show_compilations on
DES> select * from p
Info: SQL statement compiled to:
answer(A) :-
  p(A).
answer(p.a:number(integer)) ->
{
  answer(1)
}
Info: 1 tuple computed.
DES> /use_db mysql
DES> select * from p
answer(a:integer(4)) ->
{
  answer(1)
}
Info: 1 tuple computed.
```

Note that in the first case (first **SELECT** above) when the current database is **$des**, DES solves the query (in this case retrieving tuples from DDB), and in the second case (second **SELECT** above), the query is directly submitted to the EDB, which solves it. In the first, case, the SQL statement is compiled to Datalog and solved by the deductive engine, and in the second one, data and metadata are collected from EDB and shown as a result. Retrieved types from an external database differ in general to those managed by DES, as it can be seen in this example. This is not an issue as long as equivalent types are found (in this case, **number(integer)** is considered as equivalent to **integer(4)**, as numeric size constraints are not handled by DES, up to now).

As already introduced in Section 5.1.7, even when a connection is opened, their data and metadata is not known unless it becomes the current database, as illustrated next:

```
DES> /use_db mysql
DES> create table q(a int)
DES> insert into q values (2)
Info: 1 tuple inserted.
DES> select * from q
answer(a:integer(4)) ->
{
  answer(2)
}
Info: 1 tuple computed.
DES> /use_db $des
DES> select * from q
Error: Unknown table or view "q"
DES> q(X)
Warning: Undeclared predicate(s): [q/1]
{
}
Info: 0 tuples computed.
```

However, a persisted predicate does have access to data and metadata in the EDB it was made persistent. To show this, and following the above system session, let's assert the following rule:

```
DES> /assert p(X):-q(X)
Warning: Undefined predicate(s): [q/1]
DES> p(X)
{
}
Info: 0 tuples computed.
DES> :-persistent(p(a:int),mysql)
DES> p(X)
{
  p(2)
}
Info: 1 tuple computed.
```

Here, the external database is assumed to hold a relation **q**/1 with a tuple **q(2)** in its meaning.

5.2.8 Applications

Persisting predicates opens a brand new scenario for several reasons: First, predicates are no longer limited by available memory; instead, persisted predicates are using as much secondary storage as needed and provided by the underlying external database. Predicate size limit is therefore moved to the external database. Second, processing is directed to the external database for rules that can be projected, and to the deductive engine for rules that can not. This way, one can take advantage of the external database performance and scalability. Third, queries which are not possible in an external database can be solved by the deductive engine. So, one can extend external database expressiveness with the added features in DES. Finally, as several ODBC connections are allowed at a time, different predicates can be made persistent in different DMBSs, which allows for interoperability among external relational engines and the local deductive engine, therefore enabling business intelligence applications.

For instance, let's consider MySQL, which does not support recursive queries up to its current version 5.5. The following predicate can be made persistent in this RDBMS even when it is recursive:

```
DES> :-persistent(path(a:int,b:int),mysql)
DES> /assert path(1,2)
DES> /assert path(2,3)
DES> /assert path(X,Y):-path(X,Z),path(Z,Y)
Warning: Recursive rule cannot be transferred to external
database (kept in local database for its processing):
path(X,Y) :-
  path(X,Z),
  path(Z,Y).
DES> path(X,Y)
{
  path(1,2),
  path(1,3),
  path(2,3)
}
Info: 3 tuples computed.
```

Here, non-recursive rules are stored in the external database whereas the recursive one is kept in the local database. External rules are processed by MySQL and local rules by the local deductive engine.

In addition, recall that you can use SQL on the current database schema (for which the persistent predicate schema is known). Then, even special SQL features included in DES, such as hypothetical queries, can be used. For example, and following the above system session:

```
DES> assume select 3,1 in path(a,b) select * from path
answer(path.a:number(integer),path.b:number(integer)) ->
{
  answer(1,1),
  answer(1,2),
  answer(1,3),
  answer(2,1),
  answer(2,2),
  answer(2,3),
  answer(3,1),
  answer(3,2),
  answer(3,3)
}
Info: 9 tuples computed.
```

This example also shows that DES is able to compute more queries than an RDBMS. For instance, neither MS SQL Server nor DB2 allow cycles in the above path definition. This is not the most important limitation of recursion in current RDBMSs, note that stratified recursion is not supported for more than one stratum. This means that recursive SQL queries involving **EXCEPT**, **NOT IN**, aggregates, ... are not allowed in current RDBMSs such as SQL Server and DB2. Another limitation is linear recursion: the above rules cannot be expressed in a RDMBS's SQL as there are several recursive calls. To name another, **UNION ALL** is enforced in those SQLs, so that just **UNION** is not allowed. For instance, the following query is rejected in any current commercial RDBMS, but accepted by DES:

```
DES> /duplicates on
DES> /multiline on
DES> CREATE TABLE edge(a int, b int);
DES> INSERT INTO edge VALUES(1,2);
Info: 1 tuple inserted.
DES> INSERT INTO edge VALUES(2,3);
Info: 1 tuple inserted.
DES> INSERT INTO edge VALUES(1,3);
Info: 1 tuple inserted.
DES> :-persistent(edge(a:int,b:int),mysql).
DES> :-persistent(path(a:int,b:int),mysql).
DES> WITH RECURSIVE path(a, b) AS
  SELECT * FROM edge
  UNION -- Discarding duplicates (ALL is not required)
  SELECT p1.a,p2.b
  FROM path p1, path p2
  WHERE p1.b=p2.a
SELECT * FROM path;
Warning: Recursive rule cannot be transferred to external
database (kept in local database for its processing):
path_2_1(A,B) :-
  path(A,C),
  path(C,B).
answer(path.a:number(integer),path.b:number(integer)) ->
{
  answer(1,2),
  answer(1,3),
  answer(2,3)
}
Info: 3 tuples computed.
```

Note the difference against the next query, which does not discard duplicates:

```
DES> WITH RECURSIVE path(a, b) AS
  SELECT * FROM edge
  UNION ALL -- Keeping duplicates
  SELECT p1.a,p2.b
  FROM path p1, path p2
  WHERE p1.b=p2.a
SELECT * FROM path;
Warning: Recursive rule cannot be transferred to external
database (kept in local database for its processing):
path(A,B) :-
  path(A,C),
  path(C,B).
answer(path.a:number(integer),path.b:number(integer)) ->
{
  answer(1,2),
  answer(1,3),
  answer(1,3),
  answer(2,3)
}
```

```
Info: 4 tuples computed.
```

5.2.9 Caveats

5.2.9.1 Incomplete Meanings

If a predicate **p** which depends on an external relation **r** is made persistent, then it may be the case that the default database engine cannot get the meaning of **r** but via **p**, as illustrated in the following example:

```
DES> /current_db
Info: The current database is '$des'. DBMS: $des
DES> /assert p(1)
DES> /assert p(X):-r(X)
Warning: Undefined predicate(s): [r/1]
DES> :-persistent(p(a:int),access)
DES> p(X)
{
  p(1),
  p(2),
  p(3)
}
Info: 3 tuples computed.
DES> r(X)
{
}
Info: 0 tuples computed.
DES> /use_db access
DES> /current_db
Info: The current database is 'access'. DBMS: access
DES> r(X)
{
  r(2),
  r(3)
}
Info: 2 tuples computed.
```

5.2.9.2 Opening and Closing Connections

Each time a persistent assertion is issued over a given connection, this connection is opened, although the current database is not changed to it. In addition, its is not closed although a **/drop_assertion** command was issued.

A connection cannot be closed if any persistent predicate remains on it.

5.2.9.3 Abolishing Predicates

The command **/abolish** not only abolishes rules in the deductive database but also those predicates that have been persistent in the external database, dropping their table and view definitions.

5.2.9.4 Null Values

Processing of null values involving LDB and EDB is not still supported as they have different representations. So, outer joins are not supported up to now.

5.2.9.5 External Database Processing

Only the transferred rules of persisted predicates can be processed by the EDB. In particular, neither Datalog queries nor SQL queries submitted from **`$des`** are translated into external SQL and therefore processed by such EDB. Only SQL queries in the same connection as the persisted predicate are processed by the EDB. However, future releases might translate queries submitted from **`$des`**.

5.2.9.6 Supported Platforms

5.3 Safety and Computability

5.3.1 Classical Safety

Built-in predicates are appealing, but they come at a cost, which was already noticed in Section 4.5. The domain of their arguments is infinite, in contrast to the finite domains 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. To illustrate this point, consider submitting the following view to the program file **`relop.dl`**:

```
less(X,Y) :- X < Y, c(X,Y).
```

Since the goal is **`less(X,Y)`**, and the computation is left to right, both **`X`** and **`Y`** are not range restricted when computing the goal **`X < Y`** and, therefore, this goal ranges over two infinite domains: the one for **`X`** and the one for **`Y`**. We do not allow the computation of such rules. However, if we reorder the two goals as follows:

```
less(X,Y) :- c(X,Y), X < Y.
```

we get the expected result:

```
{
  less(a1, b2),
  less(a2, b2)
}
```

Note, then, that built-in predicates affect declarative semantics, i.e., the intended meaning of the two former views should be the same, although actually it is not. Declarative semantics is therefore affected by the underlying operational mechanism. Notice, nonetheless, that Datalog is less sensitive to operational issues than Prolog and it could be said to be more declarative. First, because of terminating issues as already introduced, and second, because the problematic first view can be automatically transformed into the second, computation-safe, one, as we explain next.

We can check whether a rule is safe in the sense that all its variables are range restricted and, then, reorder the goals for allowing its computation. First, we need a notion of safety, which intuitively seems clear but that actually is undecidable [ZCF+97]. Some simple sufficient conditions for the safety of Datalog programs can be imposed, which means that rules obeying these conditions can be safely computed, although there are rules that, even violating some conditions, can be actually computed. We impose the following (weak) conditions [Ullm95, ZCF+97] for safe rules adapted to our context:

1. Any variable *X* in a rule *r* is safe if:
   a. *X* occurs in some positive goal referring to a user-defined predicate
   b. *r* contains some equality goal *X*=*Y*, where *Y* is safe (*Y* can be a constant, which, obviously, makes *X* safe)
   c. A variable *X* in the goal *X is Expression* is safe whenever all variables in *Expression* are safe
2. A rule is safe if all its variables are safe.

Notice that these conditions, currently supported by the system, are weak since they assume that user-defined predicates are safe, which is not always the case (but only require analysing locally each rule for deciding weak safety). To make these conditions stronger, 1.a. has to be changed to: "*X* occurs in some positive goal referring to a *safe* user-defined predicate", and add "3. A predicate is safe if all of its variables are safe". The changed conditions would require a global analysis of the program, which is not supported by DES up to now.

The built-in predicate **`is`** has the same problem as comparison operators as well, but it only demands ground its second argument (cf. condition 1.c above). Negation requires its argument to have no unsafe variables. In addition, to be correctly computed, the restrictions in the domains of the safe variables it may contain should be computed before. The reader is referred to Section 3.6 in [Ullm95] for finding the problems when interpreting rules with negation.

DES provides a check that allows deciding if a rule is safe and, if so, it follows a program transformation for reordering its goals in order to make it computable in a left-to-right order. This transformation does not come by default, and it can be changed with the command **`/safe`** ***`Switch`***, where ***`Switch`*** can take two values: **`on`**, for enabling program transformation, and **`off`**, for disabling this transformation. If ***`Switch`*** is not included, then the command informs whether program transformation is enabled or disabled.

The analysis performed by the system at compile-time warns about safety and computability as follows:

1. Raise an error if:
   a. A goal involving a comparison operator *will* be non-ground at run-time.
   b. The expression **`E`** in a goal **`X is E`** *will* be non-ground at run-time.
   c. The goal **`not(G)`** contains unsafe variables or its safe variables are not restricted so far.
2. Raise a warning if:
   a. A goal involving a comparison operator *may* be non-ground at run-time.
   b. The expression **`E`** in a goal **`X is E`** *may* be non-ground at run-time.

This analysis is performed in several cases:

- Whenever a rule is asserted (either manually with the command **`/assert`** or automatically when consulting programs). A rule is always asserted, even when it is detected as unsafe or it may raise an exception at run-time. Recall that safety is undecidable and there are rules detected as unsafe that can be actually and correctly computed.

- When a query, conjunctive query (autoview) or view is submitted. They are rejected and not computed if unsafety or uncomputability is detected and cannot be repaired (because program transformation is disabled or there is no way). Notice that there can be unsafe or uncomputable rules already consulted than can yield an incorrect result or raise a run-time exception.

Concluding, one can expect a correct answer whenever no unsafe, uncomputable rule has been asserted to an empty database. Recall that the local analysis relies on the weak condition that assumes that the consulted rules are safe.

Next, an example of unsafe rule including negation is provided. As introduced, such a rule, when asserted, raises an error, but it is asserted in any case in order to show its misbehaviour.

```
DES> /assert q(0)
DES> /assert p(X):-not(q(X))
Error: not(q(X)) might not be correctly computed because of the
unrestricted variable(s):
  [X]
Warning: This rule is unsafe because of variable(s):
  [X]
DES> p(X)
{
}
Info: 0 tuples computed.
```

As the domain of **X** in **p(X)** is not range restricted, no tuples are found in the left-to-right top-down search. If we submit a query as **p(1)**, the negation **not(q(1))** *should* be proven:

```
DES> p(1)
{
}
Info: 0 tuples computed.
```

However, as illustrated, there is no tuples in the answer for such a query. The misbehaviour of the rule for **p**/1 emerges here due to the way answers are computed via an extension table. As far as the query **p(1)** is subsumed by a previous call (**p(X)**), results in the extension table are reused. But if the extension table is cleared, then **p(1)** can be proved:

```
DES> /clear_et
DES> p(1)
{
  p(1)
}
Info: 1 tuple computed.
```

Notice that both calls can occur during a computation, disabling the opportunity to clear the extension table, as in:

```
DES> p(X),p(1)
```

```
Info: Processing:
  answer(X) :-
    p(X),
    p(1).
{
}
Info: 0 tuples computed.
```

A similar situation happens with equality:

```
DES> p(X),X=1
Info: Processing:
  answer(X) :-
    p(X),
    X = 1.
{
}
Info: 0 tuples computed.
```

Also notice that, if simplification mode is enabled with the command **`/simplification on`**, then this conjunctive query is simplified and computed as follows:

```
DES> p(X),X=1
Info: Processing:
  answer(1) :-
    p(1).
{
  answer(1)
}
Info: 1 tuple computed.
```

5.3.2 Safety for Aggregates and Duplicate Elimination

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`**, variable **`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 compilations, 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. Notice, however, that you can still manually write an unsafe call to these system-generated predicates, yielding to incorrect results, as the following examples illustrates:

```
DES> /assert t(1)
```

```
DES> /assert s(2)
DES> /assert l(X):-lj(t(X),s(Y),X=Y)
DES> /development on
DES> /listing
'$p0'(X,Y) :-
  '$p1'(X,Y).
'$p0'(X,'$NULL'(A)) :-
  t(X),
  not('$p1'(X,Y)).
'$p1'(X,Y) :-
  X = Y,
  t(X),
  s(Y).
l(X) :-
  lj('$p0'(X,Y)).
s(2).
t(1).
Info: 6 rules listed.
DES> '$p0'(X,Y)
{
  '$p0'(1,'$NULL'(0))
}
Info: 1 tuple computed.
DES> /list_et
Answers:
{
  not('$p1'(1,A)),
  t(1),
  '$p0'(1,'$NULL'(0))
}
Info: 3 tuples in the answer table.
Calls:
{
  '$p0'(A,B)
}
Info: 1 tuple in the call table.
```

Extension table contains the non-ground entry **not('$p1'(1,A))**, which is not safe.

5.4 Modes for Unsafe Predicates

Modes in Prolog are used to declare properties of predicates at call and/or exit times. Here, we borrow modes to specify *expected* properties for a predicate in order to be correctly computed. We use mode 'i' (for an input argument) and 'o' (for an output argument) in a different way as in Prolog standard (which, indeed does not include these symbols) so that 'i' means that the argument is expected to be ground at call time, and 'o' means that it is not, though it might be. Whereas in safe Datalog, all modes should be 'o' , in DES we can find 'i' modes as well because unsafe predicates are allowed. For instance, because there are infinite built-ins as comparison operators (<, ...), it is interesting to allow 'i' modes as well, as in the next example, that is intended to compute the first **T** natural numbers:

```
p(T,1).
```

```
p(T,X) :- p(T,Y),X=Y+1,X<T.
```

Expected goals must have a ground first argument, as:

```
p(100,X)
```

which returns the first 100 naturals. Otherwise, a run-time exception is raised:

```
DES> p(X,Y)
Exception: Non ground argument(s) found in goal 1<T in the
instanced rule:
           p(T,X) :-
             p(T,1),
             1<T,
             X=1+1.
           Asserted at 10:23:37 on 7-28-2013.
```

So, each time a rule is asserted, it is checked for classical safety and, if not safe, a mode assertion is stored, indicating the input requirement of offending arguments. The assertion has the following syntax:

```
:-mode(ModeSchema)

ModeSchema ::= PredName(Mode,...,Mode)

Mode ::= i  % The argument must be ground at call time
Mode ::= o  % The argument can be a free variable at call time
```

In the example above, the automatically-stored assertion is:

```
:-mode(p(i,o)).
```

This can listed with the command **`/list_modes`**, which lists all asserted modes, and **`/list_modes`** ***`N/A`*** for a give predicate of name ***`N`*** and arity ***`A`***.

Therefore, such declarations are understood more from a documentation point-of-view than from constraints (as types, referential integrity constraints, ...), as mode assertions recall users about expected properties for the queries (in addition to the first message they got when compiling an unsafe rule). If no mode is asserted for a given predicate, it is classical-safe.

5.5 Source-to-Source Transformations

Currently, two source-to-source transformations are possible under demand: First, as explained in the previous section, when safety transformations are enabled via the command **`/safe on`**, rule bodies are reordered to try to produce a safe rule. Second, when simplification is enabled via the command **`/simplification on`**, rule bodies containing equalities, **`true`**, and **`not(`*`BooleanValue`*`)`** are simplified.

In addition, there is also place for several automatic transformations (cf. Section 5.7 to know how to display such transformations):

- A clause containing a disjunctive body is transformed into a sets of clauses with conjunctive bodies.
- A clause containing an outer join predicate is transformed into an executable form.

- A clause containing an aggregate predicate is transformed into an executable form including grouping criterion.
- A clause containing the goal **not(is_null(+*Term*))** is transformed into a clause with this goal replaced by **is_not_null(+*Term*)**.

5.6 Multi-line Mode

By default, DES command prompt reads single-line inputs and, therefore, ending termination character is optional (as the dot (**.**) in Datalog and the semicolon (**;**) in SQL and RA). But, when writing a long query, as usual in SQL, breaking down the sentence along several lines enhances readability. This is also possible in DES by enabling multi-line mode with the command **/multiline on**. However, in this scenario, the terminating character must be issued in order to know when to finish parsing the input query. Returning to single-line mode is just by issuing **/multiline off**.

With multi-line input, multi-line remarks (enclosed between **/*** and ***/**) are also allowed. Note that nested remarks are supported, too, as:

```
/*
  First remark
  /*
    Second, nested remark
  */
```

*/

5.7 Development Mode

This section is focused at those interested in modifying and extending the system. So, from a system implementor viewpoint, it is handy to show several implementation-specific issues such as source-to-source transformations and internal representation of null values. To this end, the command **/development** [**on**|**off**] has been made available. Let's consider the following system session:

```
DES> /development off
DES> /assert p(X):-X=1;X=2
DES> /assert c(C):-count(p(X),X,C)
DES> /assert q(1)
DES> /assert l(X,Y):-lj(p(X),q(Y),X=Y)
DES> /listing

c(C) :-
  count(p(X),X,C).
l(X,Y) :-
  lj(p(X),q(Y),X = Y).
p(X) :-
  X = 1
  ;
  X = 2.
q(1).

Info: 4 rules listed.
```

```
DES> l(X,Y)
{
  l(1,1),
  l(2,null)
}
Info: 2 tuples computed.
```

Next, we enable the development mode for listings:

```
DES> /development on
DES> l(X,Y)

{
  l(1,1),
  l(2,'$NULL'(59))
}
Info: 2 tuples computed.
```

Here, the internal representation of nulls is available. If we request the listing of the stored rules in development mode:

```
DES> /listing

'$p0'(A,'$NULL'(B)) :-
  p(A),
  not('$p1'(A,C)).
'$p0'(A,B) :-
  '$p1'(A,B).
'$p1'(A,B) :-
  p(A),
  q(B),
  A = B.
c(C) :-
  count(p(X),X,'[]',C).
l(X,Y) :-
  '$p0'(X,Y).
p(X) :-
  X = 2.
p(X) :-
  X = 1.
q(1).

Info: 8 rules listed.
```

Here, we see several source-to-source transformations: First, the left join, then the aggregate count, and finally the disjunctive rule.

Development listings also allows to inspect the extension table looking at (repeated) facts involving nulls, as follows:

```
DES> /assert q(null)
DES> /assert q(null)
DES> q(X)
```

```
{
  q(1),
  q(3),
  q('$NULL'(64)),
  q('$NULL'(67))
}
Info: 4 tuples computed.
```

Compare this to the non-development mode:

```
DES> /development off
DES> q(X)
{
  q(1),
  q(3),
  q(null)
}
Info: 3 tuples computed.
```

Also, one can be aware from where nulls come because of their IDs, as in:

```
DES> /assert p(null)
DES> /listing p

p('$NULL'(70)).
p(X) :-
  X = 1.
p(X) :-
  X = 2.

Info: 3 rules listed.

DES> l(X,Y)
{
  l(1,1),
  l(2,'$NULL'(72)),
  l('$NULL'(70),'$NULL'(74))
}
Info: 3 tuples computed.
```

Observe above ID 70. There, the data source rule providing such an entry in the answer is the first rule of **p**.

As SQL statements and RA expressions are compiled to Datalog programs, the command **/show_compilations on** enables the display of compilations each time an SQL statement is submitted, as the following example illustrates:

```
DES> /show_compilations on
DES> create table t(a int, b int)
DES> create table s(a int, b int)
DES> select * from t where a>1 union select * from s where b<2
Info: SQL statement compiled to:
answer(A,B) :-
  distinct(answer_2_1(A,B)).
answer_2_1(A,B) :-
  t(A,B),
```

```
  A > 1.
answer_2_1(A,B) :-
  s(A,B),
  B < 2.
answer(t.a, t.b) ->
{
}
Info: 0 tuples computed.
```

5.8 Datalog and SQL Tracers

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. However, this does not apply to Prolog, also acknowledged as a declarative language, because one can follow the execution of a goal via the SLD resolution tree and use the four-port debugging approach.

Datalog stems from logic programming and Prolog in particular, and it can be also understood as a subset of Prolog. However, its operational behaviour is quite different, since the outcome of a query represents all the possible resolutions, instead of a single one as in Prolog. In addition, tabling (cf. Section 5.5) and program transformations (due to outer joins, aggregates, simplifications, disjunctions, ...) make tracing cumbersome.

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 to examine the syntactical graph of a query, which possibly depends on other views or predicates (SQL or Datalog, resp.) This graph may be cyclic when recursive views or predicates are involved. However, a given node in the graph will be traversed only once. 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).

Next, tracing for both Datalog queries and SQL views are explained and illustrated with examples.

5.8.1 Tracing Datalog Queries

The command **`/trace_datalog`** ***`Goal`*** **`[`*`Order`*`]`** allows to trace a Datalog goal in the given order (**`postorder`** or the default **`preorder`**). Goals should be basic, i.e., no conjunctive or disjunctive goals are allowed. For instance, let's consider the program in the file **negation.dl** and its dependency graph, shown in Figure 3. A tracing session could be as follows:

```
DES> /c negation
Warning: Undefined predicate(s): [d/0]
DES> /trace_datalog a
Info: Tracing predicate 'a'.
{
  a
}
Info: 1 tuple in the answer table.
Info : Remaining predicates: [b/0,c/0,d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'b'.
{
  not(b)
}
Info: 1 tuple in the answer table.
Info : Remaining predicates: [c/0,d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'c'.
{
  c
}
Info: 1 tuple in the answer table.
Info : Remaining predicates: [d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'd'.
{
}
Info: No more predicates to trace.
```

5.8.2 Tracing SQL Views

Tracing SQL views is similar to tracing Datalog queries, but, instead of posing a goal (involving in general variables and constants) to trace, only the name of a view should be given. For example, let's consider the file **family.sql**, which contains view definitions for **ancestor** and **parent**, where tables **father** and **mother** are involved in the latter view. Note that this view is recursive since it depends on itself:

```
create view parent(parent,child) as
    select * from father
  union
    select * from mother;

create or replace view ancestor(ancestor,descendant) as
    select parent,child from parent
  union
    select parent,descendant
      from parent,ancestor where parent.child=ancestor.ancestor;
```

Then, tracing the view **ancestor** is as follows:

```
DES-SQL> /trace_sql ancestor
Info: Tracing view 'ancestor'.
{
  ancestor(amy,carolIII),
  ...
  ancestor(tony,carolIII)
}
Info: 16 tuples in the answer table.
Info : Remaining views: [parent/2,father/2,mother/2]
Input: Continue? (y/n) [y]:
Info: Tracing view 'parent'.
{
  parent(amy,fred),
  ...
  parent(tony,carolII)
}
Info: 8 tuples in the answer table.
Info : Remaining views: [father/2,mother/2]
Input: Continue? (y/n) [y]:
Info: Tracing view 'father'.
{
  father(fred,carolIII),
  ...
  father(tony,carolII)
}
Info: 4 tuples in the answer table.
Info : Remaining views: [mother/2]
Input: Continue? (y/n) [y]:
Info: Tracing view 'mother'.
{
  mother(amy,fred),
  ...
  mother(grace,amy)
}
Info: 4 tuples in the answer table.
Info: No more views to trace.
DES-SQL> /trace_datalog father(X,Y)
Info: Tracing predicate 'father'.
{
  father(fred,carolIII),
  ...
  father(tony,carolII)
}
Info: 4 tuples in the answer table.
Info: No more predicates to trace.
```

5.9 Datalog Declarative Debugger

Our approach [CGS07] to debug Datalog programs is anchored to the semantic level instead of the computation level. We have implemented a novel way of applying declarative debugging, also called algorithmic debugging (a term first coined in the logic programming field by E.H. Shapiro [Shap83]) to Datalog programs. With this approach, 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). Our system uses a question-answering procedure which starts when the user detects an unexpected answer for some query. Then, if possible, it points to the program fragment responsible of the incorrectness.

The debugging process consists of two phases. During the first phase the debugger builds a computation graph (CG) for the initial query Q w.r.t. the program P. This graph represents how the meanings of queries are constructed. See more details in [CGS07]. The second phase consists of traversing the CG to find either a buggy vertex or a set of related incorrect vertices. The vertex associated to the initial query Q is marked automatically as non-valid by the debugger. The rest of the vertices are marked initially as unknown. In order to minimize the number of questions asked by a declarative debugger, several traversing strategies have been studied [Caba05,Silv07]. However, these strategies are only adequate for declarative debuggers based on trees and not on graphs. The currently implemented strategy already contains some ideas of how to minimize the number of questions in a CG:

- First, the debugger asks about the validity of vertices that are not part of cycles in order to find a buggy vertex, if it exists. Only when this is no longer possible, the vertices that are part of cycles are visited.
- Each time the user indicates that a vertex (Query = FactSet) is valid, i.e., the validity of the answer for the subquery Query is ensured, the tool changes to valid all the vertices with queries subsumed by Query.
- Each time the user indicates that a vertex (Query = FactSet) is non-valid, the tool changes to non-valid all the vertices with queries subsumed by Query.

The last two items help to reduce the number of questions, deducing automatically the validity of some vertices from the validity of others.

As an example, we show a debugger session for the query **br_is_even** in the program **parity.dl**, which has been changed to contain an error in the following rule:

```
has_preceding(X) :- br(X), br(Y), Y>X. %error: Y>X should be Y<X
```

In this case, the user expects the answer for the query **br_is_even** to be **{br_is_even}**, because the relation **br** contains two elements: **a** and **b**. However, the answer returned by the system is **{}**, which means that the corresponding query was unsuccessful.

The available command for starting a debugging session is **/debug_datalog *Goal***, where ***Goal*** is a basic goal, i.e., no conjunctive or disjunctive goals are allowed. Therefore, the user can start a typical debugging session as follows:

```
DES> /debug_datalog br_is_even

Is br(a) = {br(a)}  valid(v)/nonvalid(n)/abort(a) [v]? v
Is has_preceding(a) = {has_preceding(a)}
valid(v)/nonvalid(n)/abort(a) [v]? n
Is br(E) = {br(a),br(b)}  valid(v)/nonvalid(n)/abort(a) [v]?

 Error in relation:  has_preceding/1
 Witness query    :  has_preceding(a) -> {has_preceding(a)}
```

```
 More information?   (yes(y)/no(n)/abort(a)) [n]? y

Is the witness query a wrong answer(w)/missing
answer(m)/abort(a) [w]? w

 Error in relation:  has_preceding/1
 Error in rule     :
  has_preceding(X) :-
    br(X),
    br(Y),
    Y > X.
           File :
c:/fernan/research/bddeduc/des/releases/des3.0/des3.0windows32si
cstus/des/examples/parity.dl
           Lines: 18,19
```

In this particular case, only three questions are necessary to find out that the relation **has_preceding** is incorrectly defined. In addition, by requesting for more information, we can even find out the offending rule in the predicate.

5.10 SQL Declarative Debugger

As in the previous section, here we focus on a declarative approach to debugging, following [CGS12a] (former version of the debugger is based on [CGS11b] and subsumed by the current one, which is a brand new implementation). There, possible erroneous objects correspond to views, and the debugger looks for erroneous views asking the user whether the result of a given view is as expected.

When the user starts the debugger for a view with the command **/debug_sql *View***, the debugger builds internally its computation tree and starts the debugging session. The root of the tree is the view under debugging, its nodes can be either views or tables, and children of a view are all of the views and tables occurring in that view (table nodes do not have children). This tree is traversed and the validity (whether the view outcome matches its intended meaning) of each node is asked to the user. If a given node is checked as valid, its subtree is assumed to be valid and it is no longer traversed. Otherwise, the node itself or one of its descendants is assumed to be nonvalid. In this case, the subtree is traversed to find the erroneous node.

Considering the file **pets1.sql** in the directory **examples/SQLDebugger** (the problem is explained in the same file), we find that the view **Guest** returns an unexpected answer:

```
DES> /process examples/SQLDebugger/pets1.sql
...
DES> select * from Guest;

answer(Guest.id:number(integer),Guest.name:string(varchar(50)))
->
{
  answer(1,'Mark Costas'),
  answer(2,'Helen Kaye'),
  answer(3,'Robin Scott')
}
```

```
Info: 3 tuples computed.
```

In fact, only **Robin Scott** is expected in the result set. Then, we can debug that view as follows:

```
DES> /debug_sql Guest
Info: Debugging view 'Guest'.
{
  1 - 'Guest'(1,'Mark Costas'),
  2 - 'Guest'(2,'Helen Kaye'),
  3 - 'Guest'(3,'Robin Scott')
}
Input: Is this the expected answer for view 'Guest'?
(y/n/m/mT/w/wN/a/h) [n]: n
Info: Debugging view 'CatsAndDogsOwner'.
{
  1 - 'CatsAndDogsOwner'(1,'Wilma'),
  2 - 'CatsAndDogsOwner'(2,'Lucky'),
  3 - 'CatsAndDogsOwner'(3,'Rocky')
}
Input: Is this the expected answer for view 'CatsAndDogsOwner'?
(y/n/m/mT/w/wN/a/h) [y]: n
Info: Debugging view 'NoCommonName'.
{
  1 - 'NoCommonName'(1),
  2 - 'NoCommonName'(2),
  3 - 'NoCommonName'(3)
}
Input: Is this the expected answer for view 'NoCommonName'?
(y/n/m/mT/w/wN/a/h) [y]: n
Info: Debugging view 'LessThan6'.
{
  1 - 'LessThan6'(1),
  2 - 'LessThan6'(2),
  3 - 'LessThan6'(3),
  4 - 'LessThan6'(4)
}
Input: Is this the expected answer for view 'LessThan6'?
(y/n/m/mT/w/wN/a/h) [y]: y
Info: Debugging view 'AnimalOwner'.
{
  1 - 'AnimalOwner'(1,'Kitty',cat),
  2 - 'AnimalOwner'(1,'Wilma',dog),
  3 - 'AnimalOwner'(2,'Lucky',dog),
  4 - 'AnimalOwner'(2,'Wilma',cat),
  5 - 'AnimalOwner'(3,'Oreo',cat),
  6 - 'AnimalOwner'(3,'Rocky',dog),
  7 - 'AnimalOwner'(4,'Cecile',turtle),
  8 - 'AnimalOwner'(4,'Chelsea',dog)
}
Input: Is this the expected answer for view 'AnimalOwner'?
(y/n/m/mT/w/wN/a/h) [y]: y
Info: Buggy relation found: CatsAndDogsOwner
```

In this example, tables have been trusted, but it is also possible to ask the user for the validity of the involved tables in the debugging process via the command **/debug_sql Guest trust_tables(no)**. In this example session, validity of table Owner would be asked to the user.

5.10.1 Trusted Specifications

In SQL, the following scenario is very usual: A set of correct views is updated to improve its efficiency. The new set of views includes both new views and improved versions of some old views, keeping their names and intended answers. Sometimes, the new, usually more involved system, no longer produces the expected results. We allow to use the first, reliable version, which we call a *trusted specification* during the subsequent debugging session.

For instance, let's consider that the user has corrected the former example, which is now working properly. Now, suppose that, in order to improve readability, the set of views is changed by removing **AnimalOwner**, adding instead a new view **CatOrDogOwner**, and modifying **LessThan6** and **CatsAndDogsOwner**, which now make use of **CatOrDogOwner**.

Next, the modified and new views (**Guest** and **NoCommonName** remain the same; this new version is located in file **examples/SQLDebugger/pets2.sql**) are listed.

```
create or replace view CatsOrDogsOwner(id,aname,specie) as
  select O.id, P.name, P.specie
  from Owner O, Pet P, PetOwner PO
  where O.id = PO.id and P.code = PO.code
        and (specie='cat' or specie='dog');

 create or replace view CatsAndDogsOwner(id,aname) as
  select A.id, A.aname
  from CatsOrDogsOwner A, CatsOrDogsOwner B
  where A.id=B.id and A.specie=B.specie;

 create or replace view LessThan6(id) as
  select id from CatsOrDogsOwner
  group by id having count(*)<6;
```

The intended answer of the views with the same name is kept. In the case of **CatOrDogOwner**, its intended answer is the multiset of owners with their pet names and species, but limited to cats and dogs.

The very same computation tree as for **pets1.sql** results after replacing literals **AnimalOwner** by **CatOrDogOwner**. However, the new set of views is erroneous, since the **WHERE** condition **A.specie=B.specie** of **CatsAndDogsOwner** should be **A.specie <> B.specie**, in order to ensure that the owner has at least one dog and one cat.

Now, the user again detects an unexpected result from the view **Guest** since its outcome incorrectly includes the owner with identifier 4: **Tom Cohen**. A new debugging session starts, but now the old version of the views (in the file **pets_trust**) can be used as a trusted specification as follows:

```
DES> /process examples/SQLDebugger/pets2.sql
...
DES> /debug_sql Guest
trust_file('examples/SQLDebugger/pets_trust')

Info: Debugging view 'Guest'.
{
  1 - 'Guest'(3,'Robin Scott'),
  2 - 'Guest'(4,'Tom Cohen')
}
Input: Is this the expected answer for view 'Guest'?
(y/n/m/mT/w/wN/a/h) [n]: n
Info: view 'NoCommonName' is nonvalid w.r.t. the trusted file.
Info: view 'LessThan6' is valid w.r.t. the trusted file.
Info: view 'CatsAndDogsOwner' is nonvalid w.r.t. the trusted
file.
Info: Debugging view 'CatsOrDogsOwner'.
{
  1 - 'CatsOrDogsOwner'(1,'Kitty',cat),
  2 - 'CatsOrDogsOwner'(1,'Wilma',dog),
  3 - 'CatsOrDogsOwner'(2,'Lucky',dog),
  4 - 'CatsOrDogsOwner'(2,'Wilma',cat),
  5 - 'CatsOrDogsOwner'(3,'Oreo',cat),
  6 - 'CatsOrDogsOwner'(3,'Rocky',dog),
  7 - 'CatsOrDogsOwner'(4,'Chelsea',dog)
}
Input: Is this the expected answer for view 'CatsOrDogsOwner'?
(y/n/m/mT/w/wN/a/h) [y]:
Info: Buggy view found: CatsAndDogsOwner
```

Here, the debugger traverses the computation tree as before, but the user is not asked for views in the set of trusted views, and the erroneous view is caught with only one final check (compared to the four checks that would be needed otherwise). The debugger detects that the new version of **`CatsAndDogsOwner`** is erroneous.

5.10.2 Missing and Wrong Tuples

The debugger also allows the user to specify the error type, indicating if there is either a missing answer (a tuple was expected but it is not in the result) or a wrong answer (the result contains an unexpected tuple). This information is used for slicing the associated queries, keeping only those parts that might be the cause of the error. The validity of the results produced by sliced queries is easier to determine, thus facilitating the location of the error.

5.10.2.1 Missing Tuples

Let's consider another following example (located at **`examples/SQLDebugger/example1.sql`**): The loyalty program of an academy awards an intensive course for students that satisfy the following constraints:

- The student has completed the basic level course (level = 0).
- The student has not completed an intensive course.
- To complete an intensive course, a student must either pass the all in one course, or the three initial level courses (levels 1, 2 and 3).

The database schema includes three tables:

- **`courses(id,level)`** contains information about the standard courses, including their identifier and the course level
- **`registration(student,course,pass)`** indicates that the student is in the course, with pass taking the value true if the course has been successfully completed
- **`allInOneCourse(student,pass)`** contains information about students registered in a special intensive course, with pass playing the same role as in registration.

File **`example1.sql`** contains the SQL views selecting the award candidates. The first view is **`standard`**, which completes the information included in the table registration with the course level. The view **`basic`** selects those standard students that have passed a basic level course (level 0). View **`intensive`** defines as intensive students those in the table **`allInOneCourse`**, together with the students that have completed the three initial levels. However, this view definition is erroneous: We have forgotten to check that the courses have been completed (flag **`pass`**). Finally, the main view **`awards`** selects the students in the basic but not in the intensive courses. Suppose that we try the query **`select * from awards`**, and that in the result we notice that the student **`Anna`** is missing. We know that **`Anna`** completed the basic course, and that although she registered in the three initial levels, she did not complete one of them, and hence she is not an intensive student. Thus, the result obtained by this query is nonvalid.

So, the user starts the debugger as **`Anna`** is not among the (possibly large) list of student names produced by view **`awards`**. The debugging session proceeds as follows:

```
DES> /process examples/SQLDebugger/awards1
...
DES> /debug_sql awards
Info: Debugging view 'awards'.
{
  1 - awards('Carla')
}
Input: Is this the expected answer for view 'awards'?
(y/n/m/mT/w/wN/a/h) [n]: m'Anna'
Info: Debugging view 'intensive'.
Input: Should 'intensive' include a tuple of the form 'Anna'?
(y/n/a) [y]: n
Info: Debugging view 'standard'.
Input: Should 'standard' include a tuple of the form 'Anna,1,1'?
(y/n/a) [y]: y
Info: Debugging view 'standard'.
Input: Should 'standard' include a tuple of the form 'Anna,2,1'?
(y/n/a) [y]: y
Info: Debugging view 'standard'.
Input: Should 'standard' include a tuple of the form 'Anna,3,0'?
(y/n/a) [y]: y
Info: Buggy view found: intensive
```

The first answer **`m'Anna'`** indicates that (**`’Anna’`**) is missing in the view awards. Next, the user indicates that view intensive should not include (**`’Anna’`**). The debugger then asks three simple questions involving the view standard. After checking the information for **Anna**, the user indicates that the listed tuples are correct. Then, the tool points out **intensive** as the buggy view, after only five simple questions. Observe that intermediate views can contain hundreds of thousands of tuples, but the slicing mechanism helps to focus only on the source of the error.

5.10.2.2 Wrong Tuples

Let's consider a modification of the database defined in **awards1.sql** as found in file **awards2.sql**, where the view **basicLevelStudents** has been incorrectly defined. We process this file, inspect the outcome of **awards** and notice that **Anna** should not be in the result set. Then, we proceed with the debugging session as follows:

```
DES> /process examples/SQLDebugger/awards2
...
DES> /debug_sql awards
Info: Debugging view 'awards'.
{
  1 - awards('Ana'),
  2 - awards('Mica')
}
Input: Is this the expected answer for view 'awards'?
(y/n/m/mT/w/wN/a/h) [n]: w1
Info: Debugging view 'intensiveStudents'.
{
  1 - intensiveStudents('Juan')
}
Input: Is this the expected answer for view 'intensiveStudents'?
(y/n/m/mT/w/wN/a/h) [y]:
Info: Debugging view 'candidates'.
Input: Should 'candidates' include a tuple of the form 'Ana'?
(y/n/a) [y]: n
Info: Debugging view 'basicLevelStudents'.
Input: Should 'basicLevelStudents' include a tuple of the form
'Ana'? (y/n/a) [y]: n
Info: Debugging view 'salsaStudents'.
Input: Should 'salsaStudents' include a tuple of the form
'Ana,1,teach1'? (y/n/a) [y]:
Info: Debugging view 'salsaStudents'.
Input: Should 'salsaStudents' include a tuple of the form
'Ana,2,teach2'? (y/n/a) [y]:
Info: Debugging view 'salsaStudents'.
Input: Should 'salsaStudents' include a tuple of the form
'Ana,3,teach1'? (y/n/a) [y]:
Info: Buggy view found: basicLevelStudents
```

5.10.2.3 Displaying Extended Information

Enabling verbose output allows to extend the display with further information as, e.g., view definitions when they are asked for its validity. As well, enabling development output allows to check how the logic program that represents the computation tree is built (c.f. [CGS12a]). For that, use the following commands, resp.:

```
DES> /verbose on
```

```
Info: Verbose output is on.

DES> /development on
Info: Development listings are on.
```

5.11 SQL Test Case Generator

Checking that a view produces the same result as its intended interpretation is a daunting task when large databases and both dependent and correlated queries are considered. Test case generation provides tuples that can be matched to the intended interpretation of a view and therefore be used to catch possible design errors in the view.

A test case for a view in the context of a database is a set of tuples for the different tables involved in the computation of the view. Executing a view for a *positive* test case (PTC)[7] 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 (NTC) 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.

A PTC in a basic query means that at least one tuple in the query domain satisfies the **where** condition. In the case of aggregate queries, a PTC will require finding a valid aggregate verifying the **having** condition, which in turn implies that all its rows verify the **where** condition.

In the case of basic query, a NTC will contain at least one tuple in the result set of the view not verifying the **where** condition. In queries containing aggregate functions, this tuple either does not satisfy either the **where** condition or the **having** condition. Set operations are also allowed in both PTC and NTC generation.

It is possible to obtain a test case which is both positive and negative at the same time thus achieving *predicate coverage* with respect to the where and having clauses (in the sense of [AO08]). We will call these tests PNTCs. For instance, consider the following system session:

```
DES-SQL> create table t(a int primary key)
DES-SQL> create view v(a) as select a from t where a=5
DES-SQL> /test_case v
Info: Test case over integers:
[t(5),t(-5)]
```

The test case {**t(5)**,**t(4)**} is a PNTC. However, a PNTC is not always possible to be generated. For instance, it is possible for the following view to generate both PTCs and NTCs but no PNTC:

```
create view v(a) as
select a
from t
where a=1 and not exists (select a from t where a<>1);
```

[7] That is, executing the view using as input data for the tables those in the PTC.

The only one PTC for this view is {**t(1)**} (modulo duplicates). There are many NTCs, as, e.g., {**t(2)**} and {**t(1),t(2)**}.

The command **/test_case *View* [*Options*]** allows two kind of options: first, to specify which *class* of test case is to be generated: **all** (PNTC, the default option), **positive** (PTC) or **negative** (NTC). The second option specifies an *action*: the results are to be displayed via the option **display** (default option), added to the corresponding tables (**add** option) or the contents of the tables replaced by the generated test case tuples (**replace** option).

For experimenting with the domain of attributes, we provide the command **/tc_domain *Min Max***, which defines de range of values the integer attributes may take. This range is determinant in the search of test cases in a constraint network that can easily become too complex as long as involved views grow. So, keeping this domain small allows to manage bigger problems.

String constants occurring in all the views on which the view for the test case generated depends are mapped to integers in the same domain, starting from 0. So, the size of the domain has to be larger enough to hold, at least, the string constants in those views.

Also, we provide the command **/tc_size *Min Max*** for specifying the size of the test case generated, in number of tuples. Again, keeping this value small helps in being able to cope with bigger problems.

Currently, we provide support for integer and string attributes. Binary distributions, and both SICStus and SWI Prolog source distributions allow the functionality described.

5.12 Batch Processing

There are two ways for processing batch files:

1. If the file **des.ini** is located at the distribution directory, its contents are interpreted as input prompts and executed before giving control to the user at start-up of the system.
2. The command **/process *filename*** (or **/p** as a shorthand) allows to process each line in the file as it was an input, the same way as before. If no file extension is given and ***filename*** does not exists, then **.ini**, **.sql**, and **.ra** are appended in turn to filename and tried in that order for finding an existing file.

When processing batch files, prompt inputs starting with the symbol **%** are interpreted as comments. This way, the batch file des.ini may contain comments. The user can also interactively input such comments, but again produce no effects.

Batch processing can include logging to produce output. This is useful to feed the system with batch input and get its output in a file, maybe avoiding any interactive input. For example, consider the following **des.ini** excerpt:

```
% Dump output to output.txt
/log output.txt
/pretty_print off
% Process (Datalog, SQL, ... queries and commands)
/c examples/fib
```

```
fib(100,F)
% End log
/nolog
```

The result found in output.txt should be (modulo blank lines):

```
DES> /pretty_print off
Info: Pretty print is off.
DES> % Process (Datalog, SQL, ... queries and commands)
DES> /c examples/fib
Warning: N > 1 may raise a computing exception if non-ground at
run-time.
Warning: N2 is N - 2 may raise a computing exception if non-
ground at run-time.
Warning: N1 is N - 1 may raise a computing exception if non-
ground at run-time.
Warning: Next rule is unsafe because of variable(s):
  [N]
fib(N,F) :- N > 1,N2 is N - 2,fib(N2,F2),N1 is N -
1,fib(N1,F1),F is F2 + F1.
DES> fib(100,F)
{
  fib(100,573147844013817084101)
}
Info: 1 tuple computed.
DES> % End log
DES> /nolog
```

5.13 Messages

DES system messages are prefixed by:

- **`Info`**: An information message which requires no attention from the user. Several information messages are hidden with the command **`/verbose off`**, which is the default mode.
- **`Warning`**: A warning message which does not necessarily imply an error, but the user is requested to focus on its origin. These messages are always shown.
- **`Error`**: An error message which requires attention from the user. These messages are always shown.
- **`Exception`**: An exception message which requires attention from the user. These messages are always shown. Examples of exception messages include instantiation errors and undefined predicates.

Prolog exceptions are caught by DES and shown to the user without any further processing. Depending on the Prolog platform, the system may continue by itself; otherwise the user must type **`des.`** (including the ending dot) to continue. Upon exceptions, the extension table is cleared and stratification is recomputed. Note that the latter computation may take a long time if there are multiple tables and views (typically in opened ODBC connections for DBMS's as Oracle and SQL Server).

5.14 Commands

The input at the prompt (i.e., commands or queries) must be written in a line (i.e., without carriage returns, although it can be broken by the DES console due to space limitations) and can end with an optional dot.

Commands are issued by preceding the command with a slash (`/`) at the DES system prompt. Command arguments are not a comma-separated list enclosed between brackets as usual, but they simply occur separated by at least one blank. This enables short typing.

Command names and binary flags (on/off switches) are not case sensitive.

Ending dots are considered as part of the argument wherever they are expected. For instance, **`/cd ..`** behaves as **`/cd ...`** (this command changes the working directory to the parent directory). In this last case, the final dot is not considered as part of the argument. The command **`/ls .`** shows the contents of the working directory, whereas **`/ls ..`** shows the contents of the parent directory (which behaves as **`/ls ...`**).

Filenames and directories can be specified with relative or absolute names. There is no need of enclosing such names between separators. For instance, file or directory names can contain blanks (for Windows users) and you neither need to use double quotes nor are allowed to use them.

Since commands are submitted with a preceding slash, they are only recognized as commands in this way. Therefore, you can use command names for your relation names without name clashes.

When consulting Datalog files, filename resolution works as follows:

- If the given filename ends with **`.dl`**, DES tries to load the file with this (absolute or relative) filename.
- If the given filename does not end with **`.dl`**, DES firstly tries to load a file with .dl appended to the end of the filename. If such a file is not found, it tries to load the file with the given filename.

In command arguments, when applicable, you can use relative or absolute pathnames. In general, you can use a slash (`/`) as a directory delimiter, but depending on the platform, you can also use the backslash (`\`). Also, it might be needed to enclose pathnames between single quotes (`'`).

See Section 4.1.2 for information about DES queries.

Some commands are labelled with *TAPI enabled*, which means that they can be submitted to the textual application programming interface (TAPI). There is additional information for such commands in Section 5.15.2.

Next, commands are described, where italics indicate a parameter which must be supplied by the user. Square brackets indicate an optional keyword or parameter (excepting the first two DES Database commands for consulting and reconsulting files, following Prolog syntax). If a parameter is not accepted, please try again enclosing it between single quotes (`'`).

5.14.1 DES Database

- **/[*FileNames*]**
  Load the Datalog programs found in the comma–separated list **[*Filenames*]**, discarding both rules already loaded, integrity constraints, and SQL table and view definitions. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
  Examples:
  Assuming we are on the examples distribution directory, we can write:
  DES> /[mutrecursion,family]
  TAPI enabled.
  See also **/consult *Filename***.

- **/[+*FileNames*]**
  Load the Datalog programs found in the comma–separated list ***Filenames***, keeping rules already loaded, integrity constraints, and SQL table and view definitions. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
  TAPI enabled.
  See also **/[*Filenames*]**.

- **/abolish**
  Delete the Datalog database. This includes all the local rules (including those which are the result of SQL compilations) and external rules (persisted predicates). Integrity constraints, and SQL table and view definitions are removed. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/abolish *Name***
  Delete the predicates matching ***Name***. This includes all their local rules (including those which are the result of SQL compilations) and external rules (persisted predicates). Their integrity constraints, and SQL table and view definitions are removed. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/abolish *Name/Arity***
  Delete the predicates matching the pattern ***Name/Arity***. This includes all their local rules (including those which are the result of SQL compilations) and external rules (persisted predicates). Their integrity constraints, and SQL table and view definitions are removed. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/assert *Head*[*:-Body*]**
  Add a Datalog rule. If ***Body*** is not specified, it is simply a fact. Rule order is irrelevant for Datalog computation. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/consult *FileName***
  Load the Datalog program found in the file ***Filename***, discarding the rules already loaded, integrity constraints, and SQL table and view definitions. The extension table is cleared, and the predicate dependency graph and strata are recomputed. The default extension **.dl** for Datalog programs can be omitted.
  Examples:

Assuming we are on the distribution directory, we can write:
`DES> /consult examples/mutrecursion`
which behaves the same as the following:
`DES> /consult examples/mutrecursion.dl`
`DES> /consult ./examples/mutrecursion`
`DES> /consult c:/des3.3.1/examples/mutrecursion.dl`
This last command assumes that the distribution directory is **`c:/des`**`3.3.1.`
Synonyms: **`/c`**, **`/restore_ddb`**.
TAPI enabled.

- **`/check_db`**
  Check database consistency w.r.t. declared integrity constraints (types, existency, primary key, candidate key, foreign key, functional dependency, and user-defined). Display a report with the outcome

- **`/des`** ***`Input`***
  Force DES to solve ***`Input`***. If ***`Input`*** is an SQL query, DES solves it instead of relying on external DBMS solving. This allows to try the more expressive queries which are available in DES (as, e.g., hypothetical and non-linear recursive queries)

- **`/drop_ic`** ***`Constraint`***
  Drop the specified integrity constraint, which starts with "**`:-`**" and can be either one of:
  - **`:- type(Table, [Column:Type])`**
  - **`:- nn(Table, Columns)`**
  - **`:- pk(Table, Columns)`**
  - **`:- ck(Table, Columns)`**
  - **`:- fk(Table, Columns, RTable, RColumns)`**
  - **`:- fd(Table, Columns, DColumns)`**
  - **`:- Goal`**

  where Goal specifies a user-defined integrity constraint). Only one constraint can be dropped at a time. Alternative syntax for constraint is also allowed.
  TAPI enabled.

- **`/listing`**
  List the loaded Datalog rules. Neither integrity constraints nor SQL views and metadata are displayed.

- **`/listing`** ***`Name`***
  List the loaded Datalog rules matching ***`Name`***. Neither integrity constraints nor SQL views and metadata are displayed.

- **`/listing`** ***`Name/Arity`***
  List the loaded Datalog rules matching the pattern ***`Name/Arity`***. Neither integrity constraints nor SQL views and metadata are displayed.

- **`/listing`** ***`Head`***
  List the Datalog loaded rules whose heads are subsumed by the head ***`Head`***. Neither integrity constraints nor SQL views and metadata are displayed.

- **`/listing`** ***`Head:-Body`***
  List the Datalog loaded rules that are subsumed by ***`Head:-Body`***. Neither integrity constraints nor SQL views and metadata are displayed.

- **`/list_modes`**
  List the expected modes for unsafe predicates in order to be correctly computed. Modes can be '**`i`**' (for an input argument) and '**`o`**' (for an output argument)

- **`/list_modes`** ***`Name`***
  List expected modes, if any, for predicates with name ***`Name`*** in order to be correctly computed. Modes can be '**`i`**' (for an input argument) and '**`o`**' (for an output argument)

- **`/list_modes`** ***`Name/Arity`***
  List expected modes, if any, for the given predicate ***`Name`***/***`Arity`*** in order to be correctly computed. Modes can be '**`i`**' (for an input argument) and '**`o`**' (for an output argument)

- **`/reconsult`** ***`FileName`***
  Load a Datalog program found in the file ***`Filename`***, keeping the rules already loaded. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
  TAPI enabled.
  See also **`/consult`** ***`Filename`***.
  Synonyms: **`/r`**.

- **`/restore_ddb`** ***`Filename`***
  Restore the Datalog database in the given file (same as **`consult`**) . Constraints (type, nullability, primary key, candidate key, functional dependency, foreign key, and user-defined) are also restored, if present in ***`Filename`***

- **`/retract`** ***`Head`*****`[`*****`:-Body`*****`]`**
  Delete the first Datalog rule that unifies with ***`Head:-Body`*** (or simply with ***`Head`***, if ***`Body`*** is not specified. In this case, only facts are deleted). The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **`/retractall`** ***`Head`***
  Delete all the Datalog rules whose heads unify with ***`Head`***. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **`/save_ddb [force]`** ***`Filename`***
  Save the current Datalog database to the file ***`Filename`***. If option **`force`** is included, no question is asked to the user should the file exists already. Constraints (type, nullability, primary key, candidate key, functional dependency, foreign key, and user-defined) are also saved

5.14.2 ODBC Database

- **`/open_db`** ***`Name`*** **`[`*****`Options`*****`]`**
  Open and set the current ODBC connection to ***`Name`***, where ***`Options`***=[**`user(`*****`Username`*****`)`**] [**`password(`*****`Password`*****`)`**]. This connection must be already defined at the OS layer.
  TAPI enabled

- **`/close_db`**
  Close the current ODBC connection.

TAPI enabled

- **`/close_db`** ***`Name`***
  Close the given ODBC connection.
  TAPI enabled
- **`/current_db`**
  Display the current ODBC connection name and DSN provider.
  TAPI enabled
- **`/show_dbs`**
  Display the open database connections.
  TAPI enabled
- **`/use_db`** ***`Name`***
  Make ***`Name`*** the current ODBC connection.
  TAPI enabled

5.14.3 Debugging and Test Case Generation

- **`/debug_datalog`** ***`Goal`*** **`[`*`Level`*`]`**
  Start the debugger for the basic goal ***`Goal`*** at predicate or clause levels, which is indicated with the options **`p`** and **`c`** for ***`Level`***, respectively. Default is **`p`**.
- **`/debug_sql`** ***`View`*** **`[`*`Options`*`]`**
  Debug an SQL view where:
  `Options`**`=[trust_tables([yes|no])] [trust_file(`*`FileName`*`)]`**
  Defaults are trust tables and no trust file. It might be needed to enclose ***`FileName`*** between single quotes.
- **`/trace_datalog`** ***`Goal`*** **`[`*`Order`*`]`**
  Trace a Datalog goal in the given order (**`postorder`** or the default **`preorder`**).
- **`/trace_sql`** ***`View`*** **`[`*`Order`*`]`**
  Trace an SQL view in the given order (**`postorder`** or the default **`preorder`**).
- **`/test_case`** ***`View`*** **`[`*`Options`*`]`**
  Generate test case classes for the view ***`View`***. ***`Options`*** may include a class and/or an action parameters. The test case class is indicated by the values **`all`** (positive-negative, the default), **`positive`**, or **`negative`** in the class parameter. The action is indicated by the values **`display`** (only display tuples, the default), **`replace`** (replace contents of the involved tables by the computed test case), or **`add`** (add the computed test case to the contents of the involved tables) in the action parameter.
- **`/tc_size`** ***`Min Max`***
  Set the minimum and maximum number of tuples generated for a test case.
- **`/tc_size`**
  Display the minimum and maximum number of tuples generated for a test case.
- **`/tc_domain`** ***`Min Max`***
  Set the domain of values for test cases between ***`Min`*** and ***`Max`***.
- **`/tc_domain`**
  Display the domain of values for test cases.

5.14.4 Tabling

- **/clear_et**
  Delete the contents of the extension table.
- **/list_et**
  List the contents of the extension table in lexicographical order. First, answers are displayed, then calls.
- **/list_et *Name***
  List the contents of the extension table matching ***Name***. First, answers are displayed, then calls.
- **/list_et *Name/Arity***
  List the contents of the extension table matching the pattern ***Name/Arity***. First, answers are displayed, then calls.

5.14.5 Operating System

- **/cat *Filename***
  Type the contents of ***Filename*** enclosed between the following lines:
  %% BEGIN *AbsoluteFilename* %%
  %% END *AbsoluteFilename* %%
  Synonym: **/type *Filename***.
- **/cd *Path***
  Set the current directory to ***Path***.
  TAPI enabled.
- **/cd**
  Set the current directory to the directory where DES was started from.
  TAPI enabled.
- **/edit *Filename***
  Edit ***Filename*** by calling the predefined external text editor. This editor is set with the command **/set_editor**
- **/pwd**
  Display the absolute filename for the current directory.
  TAPI enabled.
- **/ls**
  Display the contents of the current directory in alphabetical order. First, files are displayed, then directories.
  Synonym: **/dir**.
- **/ls *Path***
  Display the contents of the given directory in alphabetical order. It behaves as **/ls**.
  Synonym: **/dir *Path***.
- **/set_editor**
  Display the current external text editor
- **/set_editor *Editor***
  Set the current external text editor to ***Editor***

- **`/shell`** ***`Command`***
  Submit ***`Command`*** to the operating system shell.
  Notes for platform specific issues:
  - Windows users:
    `command.exe` is the shell for Windows 98, whereas **`cmd.exe`** is the one for Windows NT/2000/2003/XP/Vista/7.
  - SICStus users:
    Under Windows, if the environment variable **`SHELL`** is defined, it is expected to name a Unix like shell, which will be invoked with the option **`-c`** ***`Command`***. If SHELL is not defined, the shell named by **`COMSPEC`** will be invoked with the option **`/C`** ***`Command`***.
  - Windows and Linux/Unix executable users:
    The same note for SICStus is applied.

  Synonyms: **`/s`**.

- **`/rm`** ***`FileName`***
  Delete ***`FileName`*** from the file system.
  Synonyms: **`/del`**.

5.14.6 Log

- **`/log`**
  Display the current log file, if any.
- **`/log`** ***`Filename`***
  Set the current log to the given filename and mode: **`write`** (overwrite existing file, if any, or creates a new one) or **`append`** (append to the contents of the existing file).
- **`/nolog`**
  Disable logging.

5.14.7 Informative

- **`/apropos`** ***`Keyword`***
  Display detailed help about ***`Keyword`***, which can be a command or built-in.
  Synonyms: **`/help`**.
- **`/builtins`**
  List predefined operators, functions, and predicates.
- **`/check`**
  Display whether integrity constraint checking is enabled.
- **`/compact_listings`**
  Display whether compact listings are enabled.
- **`/dbschema`**
  Display the database schema: Database name, tables, views and Datalog constraints. A Datalog integrity constraint is displayed under a table if it only refers to this table, and under the Datalog integrity constraints otherwise. If a constraint is created with a **`CREATE TABLE`** ***`Tablename`*** statement, it is listed under the table ***`Tablename`*** even when it refers to other tables or views
  TAPI enabled

Synonyms: **`/db_schema`**.

- **`/dbschema`** ***`Name`***
  Display the database schema for the given connection, view or table name.
  TAPI enabled
  Synonyms: **`/db_schema`**.
- **`/dbschema`** ***`Connection:Name`***
  Display the database schema for the given view or table name in the given connection.
  TAPI enabled
  Synonyms: **`/db_schema`**.
- **`/dependent_relations`** ***`Relation`***
  Display the name of relations that directly depend on relation ***`Relation/Arity`***.
  TAPI enabled
- **`/dependent_relations`** ***`Relation/Arity`***
  Display in format Name/Arity those relations that directly depend on relation ***`Relation/Arity`***.
  TAPI enabled
- **`/des_sql_solving`**
  Display whether DES is forced to solve SQL queries for external DBs. If enabled, this allows to experiment with more expressive queries as, e.g., hypothetical and non-linear recursive queries targeted at an external DBMS.
- **`/des_sql_solving`** ***`Switch`***
  Enable or disable DES solving for SQL queries when the current database is an open ODBC connection (**`on`** or **`off`**, resp.)
- **`/development`**
  Display whether development listings are enabled.
- **`/development`** ***`Switch`***
  Enable or disable development listings (**`on`** or **`off`**, resp.). These listings show the source-to-source translations needed to handle null values, Datalog outer join built-ins, and disjunctive literals.
- **`/duplicates`**
  Display whether duplicates are enabled.
- **`/hypothetical`**
  Display whether hypothetical queries are enabled (**`on`**) or not (**`off`**)
- **`/nulls`**
  Display whether nulls are enabled (**`on`**) or not (**`off`**)
- **`/sql_left_delimiter`**
  Display the SQL left delimiter as defined by the current database manager (either DES or the external DBMS via ODBC).
  TAPI enabled
- **`/sql_right_delimiter`**
  Display the SQL left delimiter as defined by the current database manager (either DES or the external DBMS via ODBC) .

TAPI enabled

- **`/help`**
  Display resumed help on commands.
  Shorthands: **`/h`**.
- **`/help` *`Keyword`***
  Display detailed help about ***`Keyword`***, which can be a command or built-in.
  Synonyms: **`/apropos`**.
- **`/is_empty` *`relation_name`***
  Display **`$true`** if the given relation is empty, and **`$false`** otherwise.
  TAPI enabled
- **`/list_tables`**
  List table names.
  TAPI enabled
- **`/list_table_schemas`**
  List table schemas.
  TAPI enabled
- **`/list_table_constraints` *`table_name`***
  List table constraints for ***`table_name.`***
  TAPI enabled
- **`/list_views`**
  List view names.
  TAPI enabled
- **`/list_view_schemas`**
  List view schemas.
  TAPI enabled
- **`/negation`**
  Display the selected algorithm for solving negation (**`strata`** or **`et_not`**).
- **`/pdg`**
  Display the current predicate dependency graph.
- **`/pdg` *`Name`***
  Display the current predicate dependency graph restricted to the first predicate found with name ***`Name`***.
- **`/pdg` *`Name/Arity`***
  Display the current predicate dependency graph restricted to the predicate with name ***`Name`*** and ***`Arity`***.
- **`/pretty_print`**
  Display whether pretty print listings is enabled.
- **`/pretty_print` *`Switch`***
  Enable or disable pretty print for listings (**`on`** or **`off`**, resp.)
- **`/prompt`**
  Display the prompt format.
- **`/prompt` *`Switch`***

Set the format of the prompt. The value **des** sets the prompt to **DES>**. The value **des_db** adds the current database name **DB** as **DES:DB>**. Finally, **plain** sets the prompt to **>**. Note that, in any case, if a language other than Datalog is selected, the language name is also displayed before **>**.

- **/referenced_relations *Relation***
  Display the name of relations that are directly referenced by a foreign key in relation ***Relation.***
  TAPI enabled
- **/referenced_relations *Relation/Arity***
  Display in format Name/Arity those relations that are directly referenced by a foreign key in relation ***Relation/Arity.***
  TAPI enabled
- **/relation_exists *relation_name***
  Display **$true** if the given relation exists, and **$false** otherwise.
  TAPI enabled
- **/relation_schema *relation_name***
  Display relation schema of ***relation_name.***
  TAPI enabled
- **/running_info**
  Display whether running information (as the incremental number of consulted rules as they are read) is to be displayed.
- **/running_info *Switch***
  Enable or disable display of running information (**on** or **off**, resp.)
- **/safe**
  Display whether safety transformation is enabled.
- **/simplification**
  Display whether program simplification is enabled.
- **/show_compilations**
  Display whether compilations from SQL DQL statements to Datalog rules are to be displayed.
- **/show_compilations *Switch***
  Enable or disable display of extended information about compilation of SQL DQL statements to Datalog clauses (**on** or **off**, resp.)
- **/show_sql**
  Display whether SQL statements which are sent to an external database are to be displayed
- **/show_sql *Switch***
  Enable or disable display of SQL statements which are sent to an external database (**on** or **off**, resp.)
- **/status**
  Display the current system status, i.e., verbose mode, the selected negation algorithm, logging, elapsed time display, program transformation, and system version.

- ***/strata***
  Display the current stratification as a list of pairs (PredName/Arity, Stratum).
- **/timing**
  Display whether elapsed time display is enabled.
- **/timing *Switch***
  Disable or enable either a basic or detailed elapsed time display (**off**, **on**, **detailed**, resp.)
- **/format_timing**
  Display whether formatted timing is enabled.
- **/format_timing *Switch***
  Enable or disable formatted timing (**on** or **off**, resp.). Given that **ms**, **s**, **m**, **h** represent milliseconds, seconds, minutes, and hours, respectively, times less than 1 second are displayed as **ms**; times between 1 second and less than 60 are displayed as **s.ms**; times between 60 seconds and less than 60 minutes are displayed as **m:s.ms**; and times from 60 minutes on are displayed as **h:m:s.ms**
- **/verbose**
  Display whether verbose output is either enabled or disabled (**on** or **off**, resp.)
- **/verbose *Switch***
  Enable or disable verbose output messages (**on** or **off**, resp.)
- **/version**
  Display the current DES system version.

5.14.8 Query Languages

- **/datalog**
  Switch to Datalog interpreter (all queries are parsed and executed first by Datalog engine. If it is not a Datalog query, then it is tried first as an SQL statement. If it is neither SQL, finally it is tried as an RA expression).
- **/datalog *Query***
  Trigger Datalog resolution for the query ***Query*** (the query is parsed and executed in Datalog, but if a parsing error is found, it is tried first as an SQL statement and second as an RA expression).
- **/hypothetical *Switch***
  Enable or disable hypothetical queries (**on** or **off**, resp.)
- **/nulls *Switch***
  Enable or disable nulls (**on** or **off**, resp.)
- **/prolog**
  Switch to Prolog interpreter (all queries are parsed and executed in Prolog).
- **/prolog *Goal***
  Trigger Prolog's SLD resolution for the goal ***Goal***.
- **/ra**
  Switch to RA interpreter (all queries are parsed and executed in RA).

- **`/ra`** ***`Query`***
  Trigger RA evaluation for the query ***`Query`***.
- **`/sql`**
  Switch to SQL interpreter (all queries are parsed and executed in SQL).
- **`/sql`** ***`SQL_statement`***
  Trigger SQL resolution for ***`SQL_statement`***.

5.14.9 TAPI-related

See also Section 5.15.2 for more information.

- **`/tapi`** ***`Input`***
  Process ***`Input`*** and format its output for TAPI communication. Only a limited set of possible inputs are allowed (cf. Section 5.15)
- **`/test_tapi`**
  Test the current TAPI connection
  TAPI enabled

5.14.10 Miscellanea

- **`/check`** ***`Switch`***
  Enable or disable integrity constraint checking (**`on`** or **`off`**, resp.)
- **`/compact_listings`** ***`Switch`***
  Enable or disable compact listings (**`on`** or **`off`**, resp.)
- **`/display_answer`**
  Display whether display of computed tuples is enabled
- **`/display_answer`** ***`Switch`***
  Enable or disable display of computed tuples (**`on`** or **`off`**, resp.) The number of tuples is still displayed
- **`/display_nbr_of_tuples`**
  Display whether display of the number of computed tuples is enabled
- **`/display_nbr_of_tuples`** ***`Switch`***
  Enable or disable display of the number of computed tuples (**`on`** or **`off`**, resp.)
- **`/duplicates`** ***`Switch`***
  Enable or disable integrity constraint checking (**`on`** or **`off`**, resp.)
- **`/negation`** ***`Algorithm`***
  Set the required ***`Algorithm`*** for solving negation (**`strata`** or **`et_not`**) .
- **`/halt`**
  Quit the system.
  Synonyms: **`/quit`**, **`/q`**, **`/exit`**, **`/e`**.
- **`/nulls`**
  Display whether nulls are enabled
- **`/nulls`** ***`Switch`***
  Enable or disable nulls (on or off, resp.)

- **/multiline**
  Display whether multi-line input is enabled.
- **/multiline *Switch***
  Enable or disable multi-line input (**on** or **off** resp.)
- **/order_answer**
  Display whether displayed answers are ordered by default
- **/order_answer *Switch***
  Enable or disable a default (ascending) ordering of displayed computed tuples (**on** or **off**, resp.)
- **/output *Switch***
  Enable or disable display output (**on** or **off**, resp.)
- **/process *Filename***
  Process the contents of ***Filename*** as if they were typed at the system prompt. Extensions by default are: **.sql** and **.ini**. When looking for a file f, the following filenames are checked in this order: **f**, **f.sql**, and **f.ini**.
  Synonyms: **/p**.
- **/restore_default_status**
  Restore the status of the system to the initial status, i.e., sets all user-configurable flags to their initial values, including the default database and the start-up directory
- **/safe *Switch***
  Enable or disable program transformation (**on** or **off**, resp.)
- **/simplification *Switch***
  Enable or disable program simplification (**on** or **off**, resp.). Rules with equalities, **true**, and **not(*BooleanValue*)** are simplified.
- **/statistics *Keyword***
  Display statistics for ***Keyword*** (**runtime** or **total_runtime**). For **runtime**, this command displays the CPU time used while executing, excluding time spent in memory management tasks or in system calls since the last call to this command. For **total_runtime** , this command displays the total CPU time used while executing, including memory management tasks such as garbage collection but excluding system calls since the last call to this command.
- **/start_stopwatch**
  Start stopwatch. Precision depends on host Prolog system (1 second or milliseconds).
- **/stop_stopwatch**
  Stop stopwatch. Precision depends on host Prolog system (1 second or milliseconds).
- **/display_stopwatch**
  Display stopwatch. Precision depends on host Prolog system (1 second or milliseconds).

5.14.11 Implementor

- **`/debug`**
  Enable debugging in the host Prolog interpreter
- **`/indexing`**
  Display whether hash indexing on memo tables is enabled
- **`/indexing`** ***`Switch`***
  Enable or disable hash indexing on memo tables (**`on`** or **`off`**, resp.) Default is enabled, which shows a noticeable speed-up gain in some cases
- **`/optimize_cc`**
  Display whether complete computations optimization is enabled
- **`/optimize_cc`** ***`Switch`***
  Enable or disable complete computations optimization (**`on`** or **`off`**, resp. and enabled by default). Fixpoint iterations and/or extensional database retrievals might been saved
- **`/optimize_ep`**
  Display whether extensional predicates optimization is enabled
- **`/optimize_ep`** ***`Switch`***
  Enable or disable extensional predicates optimization (**`on`** or **`off`**, resp. and enabled by default). Fixpoint iterations and extensional database retrievals are saved for extensional predicates as a single linear fetching is performed for computing them
- **`/optimize_nrp`**
  Display whether non-recursive predicates optimization is enabled
- **`/optimize_nrp`** ***`Switch`***
  Enable or disable non-recursive predicates optimization (**`on`** or **`off`**, resp. and enabled by default). Memoing is only performed for top-level goals
- **`/optimize_st`**
  Display whether stratum optimization is enabled
- **`/optimize_st`** ***`Switch`***
  Enable or disable stratum optimization (**`on`** or **`off`**, resp. and enabled by default). Extensional table lookups are saved for non-recursive predicates calling to recursive ones, but more tuples might be computed if the non-recursive call is filtered, as in this case an open call is submitted instead (i.e., not filtered)
- **`/optimize_sn`**
  Display whether differential semi-naive optimization is enabled
- **`/optimize_sn`** ***`Switch`***
  Enable or disable differential semi-naive optimization (**`on`** or **`off`**, resp. and enabled by default). Computing linear recursive predicates saves reusing tuples in older fixpoint iterations.
- **`/nospyall`**
  Remove all Prolog spy points in the host Prolog interpreter. Disable debugging

- **/nospy *SPred*[/*Arity*]**
  Remove the spy point on the given predicate in the host Prolog interpreter
- **/spy *Pred*[/*Arity*]**
  Set a spy point on the given predicate in the host Prolog interpreter
- **/system *Goal***
  Submit ***Goal*** to the underlying Prolog system
- **/terminate**
  Terminate the current DES session without halting the host Prolog system
  Synonym: **/t**.
- **/write *String***
  Write ***String*** to console. ***String*** can contain system variables as **$stopwatch$** (which holds the current stopwatch time) and **$total_elapsed_time$** (which holds the last total elapsed time) (See Subsection 5.14.11.1 for system variables)
- **/writeln *String***
  As **/write** but adding a new line at the end of the string
- **/write_to_file *File String***
  Write ***String*** to ***File***. If ***File*** does not exist, it is created; otherwise, previous contents are not deleted and ***String*** is simply appended to ***File***. ***String*** can contain system variables as **$stopwatch$** (which holds the current stopwatch time) and **$total_elapsed_time$** (which holds the last total elapsed time) (See Subsection 5.14.11.1 for system variables)
- **/writeln_to_file *File***
  As **/write_to_file** but writing a new line

5.14.11.1 System variables

The following are the system variables which can be used when writing strings to either the console or a file with the commands **write**, **writeln**, **write_to_file**, and **writeln_to_file**:

- **$computation_time$** last elapsed time due to computing (eliding parsing and display time)
- **$display_time$** last elapsed time due to display (eliding parsing and computing time)
- **$parsing_time$** last elapsed time due to parsing (eliding computing and display time)
- **$stopwatch$** current stopwatch time
- **$last_stopwatch$** stopwatch time for its last stop
- **$total_elapsed_time$** last total elapsed time

In addition, any dynamic predicate of arity 1 implemented in Prolog. as included in source files can be accessed as a (read-only) system variable The following is a (possibly non-updated) list of such predicates (the file **des.pl** contains all declarations of such predicates):

- **`$optimize_cf$`** Flag indicating whether complete flag optimization is enabled
- **`$optimize_cc$`** Flag indicating whether complete computation optimization is enabled
- **`$optimize_ep$`** Flag indicating whether extensional predicate optimization is enabled
- **`$optimize_nrp$`** Flag indicating whether non-recursive predicate optimization is enabled
- **`$optimize_st$`** Flag indicating whether stratum optimization is enabled
- **`$optimize_sn$`** Flag indicating whether semi-naive differential optimization is enabled
- **`$edb_retrievals$`** Flag indicating the number of EDB retrievals during fixpoint computation
- **`$et_lookups$`** Flag indicating the number of ET lookups
- **`$ct_lookups$`** Flag indicating the number of CT lookups
- **`$cf_lookups$`** Flag indicating the number of CF lookups
- **`$fp_iterations$`** Flag indicating the number of iterations during fixpoint computation
- **`$verbose$`** Verbose mode flag
- **`$pretty_print$`** Pretty print for listings (takes more lines to print)
- **`$et_flag$`** Extension Table flag
- **`$strata$`** Result from a stratification
- **`$pdg$`** Predicate Dependency Graph
- **`$user_predicates$`** List of user predicates
- **`$recursive_predicates$`** List of recursive predicates
- **`$extensional_predicates$`** List of extensional predicates
- **`$non_recursive_predicates$`** List of non-recursive predicates
- **`$nr_nd_predicates$`** List of non-recursive predicates which do not depend on any recursive predicates
- **`$null_id$`** Integer identifier for nulls, represented as '$NULL'(i), where 'i' is the null identifier
- **`$rule_id$`** Integer identifier for rules, represented as datalog(Rule,NVs,i,Lines,FileId,Kind), where 'i' is the rule identifier
- **`$duplicates$`** Flag indicating whether duplicates are enabled
- **`$timing$`** Flag indicating elapsed time display: on, off or detailed
- **`$format_timing$`** Flag indicating whether formatting of time is enabled or disabled: on or off

- **`$safe$`** Flag indicating whether program transformation for safe rules is allowed
- **`$simplification$`** Flag indicating whether program simplification for performance is allowed
- **`$language$`** Flag indicating the current default query language
- **`$start_path$`** Path on first initialization
- **`$development$`** Flag indicating a development session. Listings and consultings show source and compiled rules
- **`$safety_warnings$`** Flag indicating whether safety warnings are enabled
- **`$last_autoview$`** Flag indicating the last autoview executed. This autoview should be retracted upon exceptions
- **`$current_db$`** Flag indicating the current opened DB
- **`$trusting$`** Flag indicating whether a trust file is being processed
- **`$trusted_views$`** Predicate containing trusted view names
- **`$output$`** Flag indicating whether output is enabled (on or off)
- **`$check_ic$`** Flag indicating whether integrity constraint checking is enabled (on or off)
- **`$my_odbc_query_handle$`** Flag indicating the handle to the last ODBC query
- **`$compact_listings$`** Flag indicating whether compact listings are enabled
- **`$show_compilations$`** Flag indicating whether SQL to DL compilations are displayed
- **`$show_sql$`** Flag indicating whether externally-processed SQL statements are displayed
- **`$state$`** States for various flags to be restored upon exceptions
- **`$running_info$`** Flag indicating whether running info is to be displayed (number of consulted rules)
- **`$tapi$`** Flag indicating whether a tapi command is being processed
- **`$hypothetical$`** Flag indicating whether hypothetical queries are enabled (on or off)
- **`$indexing$`** Flag indicating whether indexing on extension table is enabled (on or off)
- **`$computed_tuples%`** Flag with the number of computed tuples during fixpoint computation (for running info display)
- **`$display_answer$`** Flag indicating whether answers are to be displayed upon solving (on or off)
- **`$display_nbr_of_tuples$`** Flag indicating whether the number of tuples are to be displayed upon solving (on or off)

- **`$order_answer$`** Flag indicating whether the answer is to be displayed upon solving (on or off)
- **`$multiline$`** Flag indicating whether multiline input is enabled (on or off)
- **`$my_statistics$`** Flag for statistics
- **`$host_statistics$`** Flag for host statistics
- **`$stopwatch$`** Flag indicating stopwatch elapsed time
- **`$des_sql_solving$`** Flag indicating whether DES solving is forced for external DBMSs
- **`$prompt$`** Flag indicating the prompt format
- **`$editor$`** Flag indicating the current external editor, if defined already
- **`$nulls$`** Flag indicating whether nulls are allowed

5.15 Textual API

Rather than providing a Prolog underlying system dependent API, DES provides a textual API (TAPI, Textual Application Programming Interface) for its communication to external applications. It can used via standard input and output streams, as provided by the OS.

Such interface has been guided by the demands of the ACIDE GUI (Graphical User Interface) in order to allow users to interact with the system via a Java application. This way, it is possible to inspect and modify database schema and table contents, both those managed by DES and also external data sources as RDBMS's, spreadsheets or csv plain files connected by an ODBC connection. However, this TAPI can be used from any application wrote in any language and running on any platform, provided that it can handle input and output standard streams.

Several existing commands, statements and queries can be processed via this interface. As well, new commands and statements have been added to support the GUI requirements described above. Input syntax is as for DES, whereas answers follow a concrete format for easing their parsing. Any input to this interface must be prepended by the command **`/tapi`**, and cannot be spread beyond a single line, as shown next:

Input: **`/tapi /test_tapi`**
Output: **`$success`**

Notice that after the command **`/tapi`**, another command follows: **`/test_tapi`**, which is only intended to test whether a successful connection between the external application and DES can be established. If so, the answer **`$success`** is sent to the output stream. The usual DES command prompt is not sent, as well as no extra blank lines (even if compact listings are disabled, cf. Section 5.14.10). Any input after /tapi can also be submitted in the DES command prompt, but following the usual DES output, instead of the TAPI-oriented way.

A typical scenario for accessing DES from an external application is to start a process from this application and connecting adequately input and output streams. If run on Windows, use the console application **`des.exe`** for such process; otherwise, use **`des`** (both provided in the binary distribution for your concrete operating system).

5.15.1 Notes about the Interface

- Text in font **`Courier New`** are for textual input and output. ***`Italized Courier New`*** stand for input that the TAPI user must provide with a concrete input. For example, description for dropping a table includes: **`/tapi drop table`** ***`table_name`***, where ***`table_name`*** is the placeholder for your concrete table to be dropped.
- Lines starting with % are remarks which are not needed to be included (they are only for explanatory purposes)
- Types returned by a database or predicate handled by DES include:
  - **`string(varchar)`**
  - **`string(varchar(`*`N`*`))`**
  - **`string(char(`*`N`*`))`**
  - **`number(integer)`**
  - **`number(float)`**

  Where ***`N`*** is an integer greater than 0.
- Types returned by ODBC databases depend on the concrete external DBMS.
- Character strings as returned by DES are enclosed between single quotes. This allows in particular to distinguish these strings from the **`null`** value, which can occur in any data type.
- Datalog identifiers in TAPI inputs must be enclosed between single quotes should they contain special characters (as blanks, commas and quotes). If an identifier contains a single quote, this must be written twice as, e.g., **`'pete''s'`** , which represents **`pete's`**
- DDL (Data Definition Language) statements for SQL and Datalog include:
  - **`CREATE TABLE`** (SQL)
  - **`CREATE VIEW`** (SQL)
  - **`RENAME`** (SQL)
  - **`:-`*`strong_constraint`*** (Datalog)
- DQL (Data Query Language) SQL statements include:
  - **`SELECT`**
  - **`WITH`**
- Any input to command **`/tapi`** is processed as a DES input. However, output is only formatted for those commands and queries as listed in sections 5.15.2 and 5.15.3. So, feeding unsupported inputs to **`/tapi`** might produce unexpected results. Users of TAPI are expected to ask for other commands and/or statements needed for their concrete applications. Feedback is welcome.

5.15.1.1 Identifiers

As SQL identifiers can contain special characters which can be missed with other language constructors, they are enclosed between delimiters in such a case. This document contains an abbreviated notation: ***`name`*** and ***`column_name`***, for table and views in the former, and columns in the second. When an SQL identifier is written as part of a TAPI input, they must be enclosed between the characters ***`L`*** and ***`R`*** (left and right delimiters, respectively). Characters for such delimiters depend on the external DBMS. For instance, MS Access requires **`[`** and **`]`**, resp., but standard SQL defines double quotes for both (**`"`**) (MS Access does not support this).

In order to know what are such characters for the current connection, one can submit the following commands:

/tapi /sql_left_delimiter

/tapi /sql_right_delimiter

Datalog identifiers suffer a similar situation but they must be enclosed, if needed because containing special characters, between single quotes. For example:

/tapi /listing 't'

Datalog identifiers as returned by DES are not delimited, though.

5.15.1.2 Kinds of Answers

Any input can return either a successful answer (with a syntax described for each supported command and statement) or an error. There are several kinds of answers:

- *Regular*:
  - Successful answer with no return data:
    $success
  - Error:
    $error
    code
    text
    ...
    text
    $eot

    Where ***code*** is the error code and ***text*** is its textual description, which can consist of several lines. Last line is the text for denoting end of transmission. Error codes are digits starting by either 0 (denoting an exception error), or 1 (denoting a warning), or 2 (denoting an extended informative message).

- *Boolean:*

  Only one line, either one of the following:
  - **$true**
  - **$false**

  If an error occurs, it is output as in the regular answer.

- Defined specifically for a given command or statement.

  If an error occurs, it is output as in the regular answer.

5.15.2 TAPI-enabled Commands

This section shows each supported command for TAPI communication.

- Command:
  /tapi /sql_left_delimiter

  Answer:

  Only one line with a single character corresponding to the SQL left delimiter as defined by the database manager (either DES or the external DBMS via ODBC).

  Example assuming an ODBC connection to MS Access:

Input:
/tapi /sql_left_delimiter

Output:
[

- Command:
  /tapi /sql_right_delimiter

  Answer:

  Only one line with a single character corresponding to the SQL right delimiter as defined by the database manager (either DES or the external DBMS via ODBC).

  Example assuming an ODBC connection to MS Access:

  Input:
  /tapi /sql_right_delimiter

  Output:
  ]

- Command:
  /tapi /cd

  Answer:

  Only one line with the full path DES was started from.

  Example:

  Input:
  /tapi /cd

  Output:
  c:/des

- Command:
  /tapi /cd *Path*

  Answer:

  Only one line with the full new path.

  Example:

  Input:
  /tapi /cd examples

  Output:
  c:/des/examples

- Command:
  /tapi /consult *File*
  /tapi /c *File*
  /tapi /[*File*]

  Answer:

Information about the loaded program and a final line containing **$eot**.

Examples:

Input:
```
/tapi /[family]
```

Output:
```
Info: 11 rules consulted.
$eot
```

Input:
```
/tapi /c family,fact
```

Output:
```
Warning: N > 0 may raise a computing exception if non-
ground at run-time.
Warning: N1 is N - 1 may raise a computing exception if
non-ground at run-time.
Warning: F is N * F1 may raise a computing exception if
non-ground at run-time.
Warning: Next rule is unsafe because of variable(s):
  [F,N]
fac(N,F) :-
  N > 0,
  N1 is N - 1,
  fac(N1,F1),
  F is N * F1.
Info: 13 rules consulted.
$eot
```

- Command:
  /tapi /reconsult *Files*
  /tapi /r *Files*
  /tapi /[+*Files*]

  Answer:

  Information about the loaded program and a final line containing **$eot**.

  Example:

  Input:
  ```
  /tapi /[+family]
  ```

  Output:
  ```
  Info: 11 rules consulted.
  $eot
  ```

- Command:
  /tapi /test_tapi

  Answer:

  Regular.

  Remarks:

This command is used to test the current connection.

Example:

Input:
/tapi /test_tapi

Output:
$success

- Command:
  /tapi /open_db *db*

  Arguments:
  db: Database connection name. Not delimited.

  Answer:

  Regular.

  Remarks:

  This command is used to open an ODBC connection (cf. Section 5.14.2).

  Example:

  Input:
  /tapi /open_db test

  Output:
  $success

- Command:
  /tapi /close_db

  Answer:

  Regular.

  Remarks:

  This command is used to close the current ODBC connection (cf. Section 5.14.2).

  Example:

  Input:
  /tapi /close_db

  Output:
  $success

- Command:
  /tapi /current_db

  Answer:

  Two lines: the first one containing the current ODBC connection name and the second one the external DBMS (cf. Section 5.14.2).

  Remarks:

This command is used to get the current ODBC connection name (cf. Section 5.14.2).

Example:

Input, assuming that the ODBC connection **`test`** is already opened:
`/tapi /current_db`

Output:
`test`
`access`

- Command:
  `/tapi /relation_exists` ***`relation_name`***

  Arguments:
  `relation_name`: Relation (table, view or predicate) name, which must be enclosed between delimiters if needed.

  Answer:

  Boolean.

  Remarks:

  This command returns **`$true`** if the given relation exists, and **`$false`** otherwise.

  Example:

  Input:
  `/tapi /relation_exists "v"`

  Output:
  `$true`

- Command:
  `/tapi` ***`ddl_query`***

  Answer:

  Regular.

  Remarks:

  This DDL statement returns **`$success`** upon a successful processing.

  Example:

  Input:
  `/tapi create table [t]([a] int)`

  Output:
  `$success`

- Command:
  `/tapi /dependent_relations` ***`pattern`***

Where ***`pattern`*** can be either ***`relation_name`*** or ***`relation_name`***/***`arity`***, where ***`relation_name`*** stands for a relation name and ***`arity`*** for its arity.

Answer:

```
relation_name
...
relation_name
$eot
```

Where ***relation_name*** stands for relation names.

Remarks:

Display the names of relations that directly depend on the given relation. Relations are returned alphabetically sorted.

Example:

Input, considering that views **z1** y **z2** reference table **t**:

```
/tapi /dependent_relations "t"
```

Output:

```
z1
z2
$eot
```

- Command:

  ```
  /tapi /list_table_schemas
  ```

  Answer:

  ```
  table_name(column_name:type,..., column_name:type)
  table_name(column_name:type,..., column_name:type)
  ...
  table_name(column_name:type,..., column_name:type)
  $eot
  ```

  Where ***table_name*** stands for table names, ***column_name*** is a column name, ***type*** is the column type, and **$eot** is the end of the transmission.

  Remarks:

  Return table schemas.

  Tables are returned alphabetically sorted.

  Example:

  Input:

  ```
  /tapi /list_table_schemas
  ```

  Output:

  ```
  t(a:number(integer))
  $eot
  ```

- Command:

  ```
  /tapi /list_view_schemas
  ```

  Answer:

  ```
  view(column_name:type,..., column_name:type)
  view(column_name:type,..., column_name:type)
  ...
  view(column_name:type,..., column_name:type)
  $eot
  ```

Where ***view_name*** stands for view names, ***column_name*** is a column name, ***type*** is the column type, and **$eot** is the end of the transmission.

Remarks:

Return view schemas.

Views are returned alphabetically sorted.

Example:

Input:
```
/tapi /list_view_schemas
```

Output:
```
v(a:number(integer),b:string(varchar(20)))
$eot
```

- Command:
  /tapi /list_table_constraints *table_name*

  Arguments:
  table_name: Table name (enclosed between SQL delimiters, if needed).

  Answer:
  ```
  NN
  $
  PK
  $
  CK
  ...
  CK
  $
  FK
  ...
  FK
  $
  FD
  ...
  FD
  $
  IC
  ...
  IC
  $eot
  ```

  Where **$** is a delimiter for different kinds of integrity constraints, ***NN*** is a single line with the names of columns with existency constraint, ***PK*** is a single line with the primary key constraint, ***CK*** are candidate keys, ***FK*** are foreign keys, ***FD*** are functional dependencies, ***IC*** are user-defined integrity constraints, and **$eot** is the end of transmission.

  Remarks:

  List table constraints.

  If there are no constraints of a given type, no line is written.

  Example:

Input:
/tapi /list_table_constraints "s"

Output (no existency constraint, primary key {**b**}, no candidate key, foreign key {**s.[a]**} → {**t.[a]**}, functional dependency **a** → **b**, and user-defined integrity constraint **:- t(X),s(X,X).**):

```
$
b
$
$
s.[a] -> t.[a]
$
[a] -> [b]
$
:- t(X),s(X,X).
$eot
```

- Command:
  /tapi /relation_schema *relation_name*

  Arguments:

  relation_name: Relation name (either a table or view), which must be enclosed between SQL delimiters if needed.

  Answer:

  relation_kind
  relation_name
  column_name
  type
  column_name
  type
  ...
  column_name
  type
  $eot

  Remarks:

  Return relation schema of ***relation_name***. First line in the answer is the kind of relation (either **$table** for a table or **$view** for a view), followed by its name in the second line. Next and successive pair of lines contain the column name and column type.

  Example:

  Input:
  /tapi /relation_schema "t"

  Output:

  ```
  $table
  t
  a
  number(integer)
  $eot
  ```

- Command:
  /tapi /drop_ic *constraint*

Arguments:

`constraint`: Constraint following Datalog syntax (cf. Section 4.1.15.8).

Answer:

Regular.

Example:

Input:
`/tapi /drop_ic :-pk('s',['b'])`

Output:
`$success`

- Command:
  `/tapi /dbschema` ***`view_name`***

  Arguments:

  `view_name`: View name as an SQL identifier, which needs to be enclosed between SQL delimiters if needed.

  Answer:

  ```
  relation_kind
  relation_name
  column_name
  type
  ...
  column_name
  type
  $
  SQL
  ...
  SQL
  $
  Datalog
  ...
  Datalog
  $eot
  ```

  Remarks:

  First line in the answer is the kind of relation (**`$view`**), followed by its name in the second line. Next and successive pair of lines contain the column name and its type. Next lines contain the SQL definition of the view, starting with a line containing the delimiter **`$`**. Next lines contain the Datalog definition of the view, starting with a line containing the delimiter **`$`**. Finally, end of transmission is the last line.

  Both Datalog and SQL outputs are displayed depending on whether pretty print is disabled or not (cf. Section 5.14.7), i.e., each statement or rule can be in a single line or multiple lines.

  Example:

  Input:
  `/tapi /dbschema "v"`

Output:
```
$view
v
a
number(integer)
b
string(varchar(20))
$
SELECT ALL *
FROM (t
      NATURAL INNER JOIN
      s);
$
$eot
```

- Command:
  /tapi /is_empty *relation_name*

  Arguments:

  relation_name: Relation name (either a table or a view), which must be enclosed between SQL delimiters if needed.

  Answer:

  Boolean.

  Remarks:

  Return **$true** is relation ***relation_name*** is empty (i.e., it contains no tuples in its meaning) and **$false** otherwise.

  Example:

  Input:
  ```
  /tapi /is_empty "t"
  ```
  Output:
  ```
  $false
  ```

5.15.3 TAPI-enabled Queries

This section shows each supported query for TAPI communication.

- Query:
  /tapi *sql_ddl_query*

  Where ***sql_ddl_query*** can be any SQL DDL query (cf. Section 4.2.4).

  Answer:

  Regular.

  Examples:

  Input:
  ```
  /tapi create table t(a int)
  ```
  Output:
  ```
  $success
  ```

Input:
/tapi rename table t to q

Output:
$success

- Query:
  /tapi *sql_dml_query*

  Where ***sql_dml_query*** can be any SQL DML query (cf. Section 4.2.5).

  Answer:

  If successful, one single line with the number of affected tuples.

  Examples:

  Input:
  /tapi insert into [t] values(3)

  Output:
  1

  Input:
  /tapi insert into [t] values('3')

  Output:
```
$error
0
Type mismatch [number(integer)] (table declaration)
$eot
```

- Query:
  /tapi *sql_dql_query*

  Where ***sql_dql_query*** can be any SQL DQL query (cf. Section 4.2.6).

  Answer:
```
relation_name
column_name
type
...
column_name
type
$
value
...
value
$
...
$
value
...
value
$eot
```

Where ***relation_name*** is the name of the answer relation, ***column_name*** is a column name, ***type*** is the column type, ***value*** is the column value, **$** is the record delimiter and **$eot** is the end of the transmission.

Remarks:

This DQL statement returns in the first line the name of the answer relation, the first column name and its type in the next two lines, and so for all of its columns. Then, each or the tuples in the relation preceded by the record delimiter (**$**). Last line is the end of transmission.

Examples:

Input, considering that table **s** contains tuples {(**1**,**'abc'**), (**null**,**'def'**), (**null**,**null**)}:

```
/tapi select * from [s]
```

Output:

```
answer
s.a
number(integer)
s.b
string(varchar(20))
$
1
'abc'
$
null
'def'
$
null
null
$eot
```

Input, considering an empty table **s**:

```
/tapi select * from [s]
```

Output:

```
answer
s.a
number(integer)
s.b
string(varchar(20))
$eot
```

5.16 ISO Escape Character Syntax

Special characters in constants and user identifiers can be specified by prepending a backslash to a escape-sequence. This feature depends on its support by the underlying Prolog system, so that the reader is referenced to read corresponding entry in the manual of such system.

Currently, escape-sequences can only be specified in files to be consulted, but not at the command prompt.

Common escape-sequences are:

- \a
  Alarm (ASCII character code 7)
- \b
  Backspace (ASCII character code 8)
- \d
  Delete (ASCII character code 127)
- \e
  Escape (ASCII character code 27)
- \f
  Form feed (ASCII character code 12)
- \n
  Line feed/Newline (ASCII character code 10)
- \r
  Carriage return (ASCII character code 13). Go to the start of the line, without feeding a new line
- \t
  Horizontal tab (ASCII character code 9)
- \v
  Vertical tab (ASCII character code 11)
- \xhex-digit...\
  A character code represented by the hexadecimal digits.

5.17 Notes about the Implementation of DES

DES is implemented with the original ideas found in [Diet87, TS86, FD92], that deal with termination issues of Prolog programs. These ideas have been already used in the deductive database community. Our implementation uses extension tables for achieving a top–down driven bottom–up approach. In its current form, it can be seen as an extension of the work in [Diet87, FD92] in the sense that, in addition, we deal with negation, undefined (although incomplete) information, nulls and aggregates, also providing a more efficient tabled mechanism. Also, the implementation follows a different approach: Instead of translating rules, we interpret them.

DES does not pretend to be an efficient system but a system capable of showing the nice aspects of the more powerful form of logic we can find in Datalog systems wrt. relational database systems.

5.17.1 Tabling[8]

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

[8] For a complementary understanding of this section, the reader is advised to read [Diet87].

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 us to detect whether a call has been previously solved and we can use the results in the extension table (if any).

The algorithm which implements this idea is depicted next:

```
% Already called. Call table with an entry for the current call
memo(G) :-
  build(G,Q),    % Build in Q the same call with fresh variables
  called(Q),     % Look for a unifiable call in CT for the current call
  subsumes(Q,G), % Test whether CT call subsumes the current call
  !,             %
  et_lookup(G).  % If so, use the results in answer table (ET)

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

This algorithm, first, tests whether there is a previous call that subsumes[9] the current call. 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 we cannot derive any information, 2) otherwise, process the new call knowing that there is no call or 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, through recursion, we can deliver new answers for the same call). This so–called memoization process is implemented with the predicate **memo/1** in the file **des.pl** of the distribution, and will also be referred to as a memo function in the rest of this manual.

Negative facts are produced when a negative goal is proved by means of negation as failure (closed world assumption). 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 as proving a positive goal.

The command **/list_et** shows the current state of the extension table, both for answers and calls already obtained by solving one or more queries (incidentally, recall that you can focus on the contents of the extension table for a given predicate, cf. Section 5.14.4). This command is useful for the user when asking for the meaning of relations, and for the developer for examining the last calls being performed. 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 empty after the execution of a temporary

[9] A term T1 subsumes a term T2 if T1 is "more general" than T2 and both terms are unifiable. Eg: **p(X,Y)** subsumes **p(a,Z)**, **p(X,Y)** subsumes **p(U,V)**, **p(X,Y)** subsumes **p(U,U)**, but **p(U,U)** neither subsumes **p(a,b)**, nor **p(X,Y)**.

view.[10] 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. The command **/clear_et** clears the extension table contents, both for calls and answers.

5.17.2 Fixpoint Computation

The tabling mechanism is insufficient in itself for computing all of the 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, we have to ensure that we produce all the possible information by finding a fixpoint of the memo function. The algorithm implementing this is depicted next:

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

First, the call table is emptied 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 stable model of the query (plus perhaps other stable models 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 we only add new entries 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 is finite (recall that there are no compound terms such as s^k**(z)**). 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**.

It is also possible that both a positive and a negative fact have been inferred for a given call. Then, an undefined fact replaces the contradictory information. The implementation simply removes the contradictory facts and informs about the

[10] The contents of the extension table in this case should be restored instead of being cleared; left for further improvements.

undefinedness. As already indicated (see Section 6.8.1), the algorithm for determining undefinedness is incomplete.

5.17.3 Dependency Graphs and Stratification: Negation, Outer Joins, and Aggregates

Each time a program is consulted or modified (i.e., via submitting a temporary view or changing the database), a predicate dependency graph is built [ZCF+97]. This graph shows the dependencies, through positive and negative atoms, among predicates in the program. Also, a negative dependency is added for each outer join goal and aggregate goal.

This dependency graph is useful for finding a stratification for the program [ZCF+97]. A stratification collects predicates into numbered strata (1..N). A basic 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. With our approach, we only resort to compute by stratum when a negative dependency occurs in the predicate dependency graph restricted to the query; 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. See also next section on optimizations.

Outer join and aggregate goals are also collected into strata as if they were negative atoms in order to have their answer set completely defined and therefore ensure termination of the computation algorithm in presence of null values (for outer joins) and incomplete set of values (for aggregates).

5.17.4 Optimizations

DES is not targeted at performance by any means: it is implemented on top of Prolog, it uses the (slower in most systems) Prolog dynamic database, it does not allow user-defined indexes, implemented algorithms are not the best ones, several tasks are redone sparingly (although they can be actually saved), and so on. Once that said, there has been still a minor room for optimizing performance so that projects of the size DES is intended for can be successfully achieved. Below, we list some of such optimizations that can be enabled or disabled at user request (this feature is more oriented to the system implementors for knowing the impact on performance of such optimizations). Each optimization is listed in a subsection along with the command (between brackets) that is used for disabling or enabling it (with the switch **`off`** and **`on`**, respectively).

5.17.4.1 Complete Computations (`/optimize_cc`)

Each call during the computation of a stratum (stratum saturation) is remembered in addition to its outcome (in the answer table). Even when the calls are removed in each fixpoint iteration (recall Section 5.17.2), most general ones do persist as a collateral data structure to be used for saving computations should any of them is called again during either computing a higher stratum or a subsequent query solving. '**`cc`**' stands for completed computation, so that if a call is marked as a completed computation, it is not even tried if called again. This means the following two points: 1) During the computation of the memo function, calls already computed are not tried to be solved again, and only the entries in the memo table are returned. 2) Moreover, computing the memo function is completely avoided if a subsuming already-computed call can be found. In the first case, that saves solving goals in computing the memo function. In the second case, that completely saves fixpoint computation.

The following system session shows how this optimization works. First, we enable statistics collection, enable verbose output to automatically display statistics results, disable all the optimizations, assert the fact **`p(1)`** and submit the query **`p(X)`**:

```
DES> /statistics on
DES> /verbose on
DES> /optimize_cc  off
Info: Complete computations optimization is off.
DES> /optimize_ep  off
Info: Extensional predicate optimization is off.
DES> /optimize_nrp off
Info: Non-recursive predicates optimization is off.
DES> /optimize_st  off
Info: Stratum optimization is already disabled.
DES> /optimize_sn  off
Info: Differential semi-naive optimization is already disabled.
DES> /assert p(1)
Info: Computing predicate dependency graph...
Info: Computing strata...
Info: Rule asserted.
DES> p(X)
Info: Parsing query...
Info: Query successfully parsed.
Info: Solving query p(X)...
Info: Displaying query answer...
Info: Sorting answer...
{
  p(1)
}
Info: 1 tuple computed.
Info: Fixpoint iterations: 2
Info: EDB retrievals     : 2
Info: IDB retrievals     : 0
Info: ET retrievals      : 4
Info: ET look-ups        : 6
Info: CT look-ups        : 2
Info: CF look-ups        : 0
```

As the statistics show, 2 fixpoint iterations have been needed to deduce the output. In the first one, the rule **`p(1)`** is read for the first time. Then, in the second iteration, it is read again and as the answer table has not changed, then this means that the fixpoint has been reached. The display "EDB retrievals" shows those two fact reads (EDB stands for Extensional Database).

If the same query is submitted again:

```
DES> p(X)
Info: Parsing query...
Info: Query successfully parsed.
Info: Solving query p(X)...
Info: Displaying query answer...
Info: Sorting answer...
```

```
{
  p(1)
}
Info: 1 tuple computed.
Info: Fixpoint iterations: 1
Info: EDB retrievals     : 1
Info: IDB retrievals     : 0
Info: ET retrievals      : 4
Info: ET look-ups        : 4
Info: CT look-ups        : 1
Info: CF look-ups        : 0
```

then only 1 iteration is needed to reach the fixpoint, and only one EDB retrieval is done, as the answer table contained an entry for **p(1)** already for the same call. This illustrates point 1 above.

Now let's enable the optimization, previously deleting the contents of the answer table so that we are in the same starting situation again:

```
DES> /clear_et
Info: Extension table cleared.
DES> /optimize_cc on
Info: Complete flag optimization is on.
DES> p(X)
Info: Parsing query...
Info: Query successfully parsed.
Info: Solving query p(X)...
Info: Displaying query answer...
Info: Sorting answer...
{
  p(1)
}
Info: 1 tuple computed.
Info: Fixpoint iterations: 2
Info: EDB retrievals     : 2
Info: IDB retrievals     : 0
Info: ET retrievals      : 4
Info: ET look-ups        : 6
Info: CT look-ups        : 2
Info: CF look-ups        : 1
```

As before, 2 fixpoint iterations and 2 EDB retrievals are needed. But, if we submit again the query:

```
DES> p(X)
Info: Parsing query...
Info: Query successfully parsed.
Info: Solving query p(X)...
Info: Displaying query answer...
Info: Sorting answer...
{
  p(1)
}
Info: 1 tuple computed.
Info: Fixpoint iterations: 0
```

```
Info: EDB retrievals     : 0
Info: IDB retrievals     : 0
Info: ET retrievals      : 2
Info: ET look-ups        : 2
Info: CT look-ups        : 0
Info: CF look-ups        : 1
```

then, as the computation for the goal **p(X)** is complete, then no fixpoint iterations are needed. For the same reason, no EDB retrievals are needed, as just the contents of the memo table are returned. This illustrates point 2 above.

5.17.4.2 Extensional Predicates (/optimize_ep)

Extensional predicates are not needed to be iteratively computed. So, no fixpoint computation is needed for them. They are known from the predicate dependency graph simply because they occur in the graph without incoming arcs. For them, a linear fetching is enough to derive their meanings. '**ep**' stands for 'extensional predicates'.

In the following system session we illustrate this with the fact **p(1)**:

```
DES> p(X)
Info: Parsing query...
Info: Query successfully parsed.
Info: Solving query p(X)...
Info: Displaying query answer...
Info: Sorting answer...
{
  p(1)
}
Info: 1 tuple computed.
Info: Fixpoint iterations: 1
Info: EDB retrievals     : 1
Info: IDB retrievals     : 0
Info: ET retrievals      : 2
Info: ET look-ups        : 3
Info: CT look-ups        : 0
Info: CF look-ups        : 0
```

where there are 1 fixpoint iteration and only one EDB retrieval. This optimization is independent from the completed computations optimization.

Successive calls will render the same behavior, unless the complete computations optimization is enabled:

```
DES> p(X)
Info: Parsing query...
Info: Query successfully parsed.
Info: Solving query p(X)...
Info: Displaying query answer...
Info: Sorting answer...
{
  p(1)
}
Info: 1 tuple computed.
Info: Fixpoint iterations: 0
```

```
Info: EDB retrievals     : 0
Info: IDB retrievals     : 0
Info: ET retrievals      : 2
Info: ET look-ups        : 2
Info: CT look-ups        : 0
Info: CF look-ups        : 1
```

where no fixpoint iterations and no EDB retrievals are needed.

5.17.4.4 Non-recursive Predicates (/optimize_nrp)

Each non-recursive predicate can be extracted out from the fixpoint iterative cycle because its meaning can be computed by requesting all its solutions at once. Further fixpoint iterations won't develop new tuples, so this would be useless. In fact, this is true for each non-recursive rule of a given predicate. Though, this optimization is not available yet.

The following example shows the predicate **p** as composed of a fact and a rule. First, it is computed with all optimizations disabled:

```
DES> /assert p(1)
DES> /assert p(X):-X=1+1
DES> p(X)
{
  p(1),
  p(2)
}
Info: 2 tuples computed.
Info: Fixpoint iterations: 2
Info: EDB retrievals     : 2
Info: IDB retrievals     : 2
Info: ET retrievals      : 8
Info: ET look-ups        : 8
Info: CT look-ups        : 2
Info: CF look-ups        : 0
```

Then, enabling non-recursive predicates optimization and submitting the same query:

```
DES> /optimize_nrp on
Info: Non-recursive predicates optimization is on.
DES> /clear_et
DES> p(X)
{
  p(1),
  p(2)
}
Info: 2 tuples computed.
Info: Fixpoint iterations: 1
Info: EDB retrievals     : 1
Info: IDB retrievals     : 1
Info: ET retrievals      : 4
Info: ET look-ups        : 4
Info: CT look-ups        : 0
Info: CF look-ups        : 0
```

In only one fixpoint iteration the meaning is computed for which 1 EDB and 1 IDB retrievals are needed (the fact and rule, respectively).

5.17.4.5 Stratum (`/optimize_st`)

A predicates which contain no recursive rules but calls to recursive predicates do not need to be computed in the same iterative fixpoint computation. If this optimization is enabled, such predicates are isolated from recursive ones in another stratum, so that iterative cycles are saved for them. This situation occurs, for instance, when compiling SQL queries to Datalog, as the intermediate relation **`answer`** is introduced. Next system session illustrates this:

```
DES> :-type(p(a:int))
DES> /display_answer off
DES> /display_nbr_of_tuples off
DES> /timing on
DES> /assert p(1)
DES> /assert p(X):-p(Y),X=Y+1,Y<500
DES> select * from p
Info: Solving query answer(A)...
answer(p.a:number(integer)) ->
Info: Fixpoint iterations: 500
Info: EDB retrievals     : 500
Info: IDB retrievals     : 1000
Info: ET retrievals      : 627246
Info: ET look-ups        : 252999
Info: CT look-ups        : 1500
Info: CF look-ups        : 0
Info: Total elapsed time: 02.755 s.

DES> /optimize_st  on
DES> select * from p
Info: Solving query answer(A)...
Info: Computing by stratum of [p(A)].
answer(p.a:number(integer)) ->
Info: Fixpoint iterations: 2
Info: EDB retrievals     : 502
Info: IDB retrievals     : 504
Info: ET retrievals      : 381248
Info: ET look-ups        : 128757
Info: CT look-ups        : 1006
Info: CF look-ups        : 0
Info: Total elapsed time: 01.888 s.
```

With this optimization enabled, less extension table lookups are needed and the result is therefore computed faster. However, note that non-termination might raise when breaking strata if using the metapredicate **`top`**: This is because **`top`** requires the amount of tuples as indicated from its goal argument. If this goal is isolated in a higher stratum, no top constraint is propagated to the lower stratum, as in:

```
DES> :- type(p(a:int))
DES> /assert p(1)
DES> /assert p(X):-p(Y),X=Y+1
DES> select top 2 * from p
answer(p.a:number(integer)) ->
```

```
{
  answer(1),
  answer(2)
}
Info: 2 tuples computed.
DES> /optimize_st on
DES> select top 2 * from p
... non-terminating query
```

That is, as the SQL query has been compiled to:

```
answer(A) :-
  top(10,p(A)).
```

then, predicate **answer**/1 is located at stratum 2 and predicate **p**/1 at stratum 1:

```
DES> /strata
[(p/1,1),(answer/1,2)]
```

and DES tries to solve first the goal **p(X)** (not **top(10,p(A))**)[11] which proves to be non-terminating as there is no top constraint on **p**. Further releases might cope with this issue.

5.17.5 Indexing (/indexing)

There is no provision for user indexes up to now. However, indexing on memo tables can be enabled or disabled at user request. There are three tables which are indexed: the answer table, the call table, and the complete computation table. The first one stores the computed results for the calls during query solving and it is used in the tabling scheme for avoiding to recompute already known goals. The second one stores the calls so that it is possible to know whether a subsuming call has been done already. The third table stores for each call whether its computation has been completed or not.

5.17.6 Porting to Unsupported Systems

DES is implemented with several Prolog files: **des.pl**, **des_dcg.pl**, **des_sql.pl**, **des_ra.pl**, **des_sql_debug.pl**, **des_dl_debug.pl**, **des_types.pl**, **des_tc.pl**, and **des_glue.pl**. The first file contains the common predicates for all of the platforms (both Prolog interpreters and operating systems) following the Prolog ISO standard. File **des_dcg.pl**, contains the definition of DCG expansion (which varies from one system to another). Files **des_sql.pl** and **des_ra.pl** contain the SQL and RA processor, respectively. Files **des_sql_debug.pl** and **des_dl_debug.pl** contain the SQL and Datalog declarative debuggers. File **des_types.pl**, contains the type checking and inference system. File **des_tc.pl** contains the SQL test case generator code. The last file **des_glue.pl** contains Prolog system specific code, which vary from a system to another. Adapting the predicates found there should not pose problems, provided that the Prolog interpreter and operating system feature some basic characteristics (mainly about the file system commands). In particular, finite domain constraints is a must for

[11] And secondly it would try the goal **answer(X)**, although in this case it is unable because of the non-terminating first goal.

supporting several features of DES, such as type inference and test case generation. If you plan to port DES to other systems not described here, you will have to modify the system specific Prolog file to suit your system. If so, and if you want to figure as one of the system contributors, please send an e–mail message with the code and reference information to: **fernan@sip.ucm.es**, accepting that your contribution will be under the GNU Lesser General Public License. (See the appendix for details.)

6. Examples

The DES distribution contains the directory **examples** which shows several features of the system. Unless explicitly noted, all queries have been solved after the commands **/verbose off** and **/pretty_print off** have been executed.

6.1 Relational Operations (files relop.{dl,sql,ra})

The program **relop.dl** is intended to show how to mimic with Datalog rules the basic relational operations that can be found in the file **relop.sql**. It contains three relations (**a**, **b**, and **c**), which are used as arguments of relational operations. In order to have loaded this program and be able to submit queries you can consult it with **/c relop**. In the remarks below, relational operator symbols are represented with ASCII characters, as **=|x|** to denote the left outer join ⟕, the letter **x** to simply denote the Cartesian product, and the letter **U** for the set union.

```
% (Extended) Relational Algebra Operations

% pi(X)(c(X,Y)) : Projection of the first argument of c
projection(X) :- c(X,Y).

% sigma(X=a2)(a) : Selecting tuples from a such that its first
argument is a2
selection(X) :- a(X), X=a2.

% a x b : Cartesian product of relations a and b
cartesian(X,Y) :- a(X), b(Y).

% a |x| b : Natural inner join of relations a and b
inner_join(X) :- a(X), b(X).

% a =|x| b : Left outer join of relations a and b
left_join(X,Y) :- lj(a(X), b(Y), X=Y).

% a |x|= b : Right outer join of relations a and b
right_join(X,Y) :- rj(a(X), b(Y), X=Y).

% a =|x|= b : Full outer join of relations a and b
full_join(X,Y) :- fj(a(X), b(Y), X=Y).

% a U b : Set union of relations a and b
union(X) :- a(X) ; b(X).

% a - b: Set difference of relations a and b
difference(X) :- a(X), not(b(X)).
```

Once the program is consulted, you can query it by, for example:

```
DES> projection(X)
{
  projection(a1),
  projection(a2)
}
Info: 2 tuples computed.
```

The result of a query is the meaning of the view, i.e., the fact set for the query derived from the program whether intensionally or extensionally. In the above example, **`projection(X)`** corresponds to the projection of the first argument of relation **`c`**.

The second view in Section 4.1.5 returns:

```
Info: Processing:
  a(X) :- b(X).
{
  a(a1),
  a(a2),
  a(a3),
  a(b1),
  a(b2)
}
Info: 5 tuples computed.
```

For abolishing this program and execute the SQL statements in **`relop.sql`**, you can type **`/abolish`** and **`/process relop.sql`**. Note that the extension can be omitted in the **`process`** command.

Here, we depart from the Datalog interpreter and, if you are to submit SQL queries, it is useful to switch to the SQL interpreter via the command **`/sql`** as inputs will be parsed only by the SQL parser. Otherwise, it will be tried to be identified as a Datalog input, and then as an SQL input.

Note that in the file **`relop.sql`** listed below, strings are enclosed between apostrophes. This is not needed in the Datalog language. In order to execute the contents of this file, type **`/process relop.sql`**.

```
% Switch to SQL interpreter
/sql
% Creating tables
create or replace table a(a);
create or replace table b(b);
create or replace table c(a,b);
% Listing the database schema
/dbschema
% Inserting values into tables
insert into a values ('a1');
insert into a values ('a2');
insert into a values ('a3');
insert into b values ('b1');
insert into b values ('b2');
insert into b values ('a1');
```

```
insert into c values ('a1','b2');
insert into c values ('a1','a1');
insert into c values ('a2','b2');
% Testing the just inserted values
select * from a;
select * from b;
select * from c;
% Projection
select a from c;
% Selection
select a from a where a='a2';
% Cartesian product
select * from a,b;
% Inner Join
select a from a inner join b on a.a=b.b;
% Left Join
select * from a left join b on a.a=b.b;
% Right Join
select * from a right join b on a.a=b.b;
% Full Join
select * from a full join b on a.a=b.b;
% Union
select * from a union select * from b;
% Difference
select * from a except select * from b;
```

If we have created the relations in Datalog, we cannot access them from SQL unless they had been either defined as tables or views or declared with types. For example, following the first alternative and after consulting the file **relop.dl**, we can submit:

```
create table a(a varchar);
```

And, then, accessing with an SQL statement the tuples that were asserted in Datalog:

```
DES-SQL> select * from a;
answer(a.a) ->
{
  answer(a1),
  answer(a2),
  answer(a3)
}
Info: 3 tuples computed.
```

Otherwise, an error is submitted:

```
Error: Unknown table or view "a"
```

Following the second alternative and after consulting the file **relop.dl**, we can declare types for **a**:

```
DES-SQL> /datalog :-type(a,[a:varchar])
DES-SQL> select * from a
answer(a.a) ->
{
```

```
  answer(a1),
  answer(a2),
  answer(a3)
}
Info: 3 tuples computed.
```

6.2 Paths in a Graph (files `paths.{dl,sql,ra}`)

This program[12] introduces the use of recursion in DES by defining the graph in Figure 1 and the set of tuples <origin, destination> such that there is a path from origin to destination.

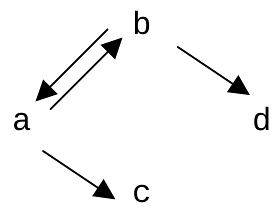


Figure 1. Paths in a Graph

The file **paths.dl** contains the following Datalog code, which can be consulted with **/c paths**:

```
% Paths in a Graph

edge(a,b).
edge(a,c).
edge(b,a).
edge(b,d).

path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

The query **path(X,Y)** yields the following answer:

```
{
  path(a,a),
  path(a,b),
  path(a,c),
  path(a,d),
  path(b,a),
  path(b,b),
  path(b,c),
  path(b,d)
}
Info: 8 tuples computed.
```

The file **paths.sql** contains the SQL counterpart code, which can be executed with **/process paths.sql**:

```
create table edge(origin,destination);
insert into edge values('a','b');
```

[12] Adapted from [TS86].

```
insert into edge values('a','c');
insert into edge values('b','a');
insert into edge values('b','d');
create view paths(origin,destination) as
  with
    recursive path(origin,destination) as
      (select * from edge)
      union
      (select path.origin,edge.destination
       from path,edge
       where path.destination =edge.origin)
  select * from path;
```

So, you can get the same answer as before with the SQL statement:

```
DES-SQL> select * from paths;
answer(paths.origin, paths.destination) ->
{
  answer(a,a),
  answer(a,b),
  answer(a,c),
  answer(a,d),
  answer(b,a),
  answer(b,b),
  answer(b,c),
  answer(b,d)
}
Info: 8 tuples computed.
```

Another shorter formulation is allowed in DES with the following view definition:

```
create view path(origin,destination) as
  select * from
  (select * from edge)
  union
  (select path.origin,edge.destination
   from path,edge
   where path.destination=edge.origin)
```

You can finally compare this with the RA formulation:

```
paths(origin,destination) :=
   select true (edge)
  union
   project paths.origin,edge.destination
    (edge zjoin paths.destination=edge.origin paths);
```

6.3 Shortest Paths (file `spaths.{dl,sql,ra}`)

Thanks to aggregate predicates, one can code the following version of the shortest paths problem (file **`spaths.dl`**), which uses the same definition of edge as the previous example:

```
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.

sp(X,Y,L) :-
   min(path(X,Y,Z),Z,L).
```

Note that the infinite computation that may raise from using the built-in is/2 is avoided by limiting the total length of a path to the number of edges in the graph.

The following query returns all the possible paths and their corresponding minimal distances:

```
DES> sp(X,Y,L)
{
  sp(a,a,2),
  sp(a,b,1),
  sp(a,c,1),
  sp(a,d,2),
  sp(b,a,1),
  sp(b,b,2),
  sp(b,c,2),
  sp(b,d,1)
}
Info: 8 tuples computed.
```

Below is the SQL formulation for the same problem (file **spaths.sql**) :

```
DES-SQL> create or replace view
spaths(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;

DES-SQL> select * from spaths
answer(spaths.origin, spaths.destination, spaths.length) ->
{
  answer(a,a,2),
  answer(a,b,1),
  answer(a,c,1),
  answer(a,d,2),
  answer(b,a,1),
  answer(b,b,2),
  answer(b,c,2),
```

```
  answer(b,d,1)
}
Info: 8 tuples computed.
```

A possible RA formulation follows:

```
max_length(max_length) :=
  group_by [] count(*) true (edge);

path(origin,destination,length) :=
   project origin,destination,1 (edge)
  union
   project path.origin,edge.destination,path.length+1
        (
         path
         zjoin path.destination=edge.origin and
              path.length<max_length
         (edge product max_length)
        );

spaths(origin,destination,length) :=
  group_by origin,destination origin,destination,min(length)
true
    (path);
```

And its query:

```
/ra select true (spaths);
```

6.4 Family Tree (files **family.{dl,sql,ra}**)

This (yet another classic) program defines the family tree shown in Figure 2, the set of tuples <**parent**,**child**> such that **parent** is a parent of **child** (the relation **parent**), the set of tuples <**ancestor**,**descendant**> such that **ancestor** is an ancestor of **descendant** (the relation **ancestor**), the set of tuples <**father**,**child**> such that **father** is the father of **child** (the relation **father**), and the set of tuples <**mother**,**child**> such that **mother** is the mother of **child** (the relation **mother**).

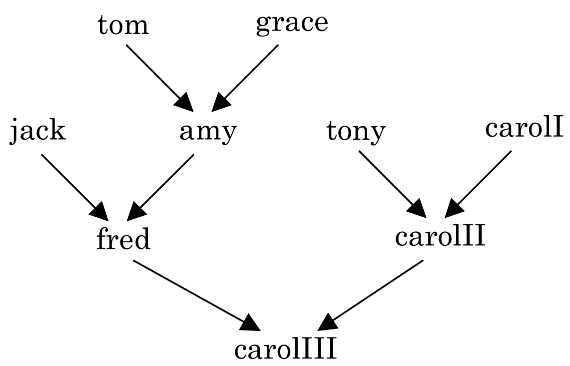


Figure 2. Family Tree

The file **family.dl** contains the following Datalog code, which can be consulted with **/c family**:

```
father(tom,amy).
father(jack,fred).
father(tony,carolII).
father(fred,carolIII).
mother(grace,amy).
mother(amy,fred).
mother(carolI,carolII).
mother(carolII,carolIII).

parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

The query **ancestor(tom,X)** yields the following answer (that is, it computes the set of descendants of **tom**):

```
{
  ancestor(tom,amy),
  ancestor(tom,carolIII),
  ancestor(tom,fred)
}
Info: 3 tuples computed.
```

Solving the view:

```
son(S,F,M) :- father(F,S),mother(M,S).
```

yields the following answer, computing the set of sons:

```
Info: Processing:
  son(S,F,M) :- father(F,S),mother(M,S).
{
  son(amy,tom,grace),
  son(carolII,tony,carolI),
  son(carolIII,fred,carolII),
  son(fred,jack,amy)
}
Info: 4 tuples computed.
```

The file **family.sql** contains the SQL counterpart code, which can be executed with **/process family.sql**:

```
create table father(father,child);
insert into father values('tom','amy');
insert into father values('jack','fred');
insert into father values('tony','carolII');
insert into father values('fred','carolIII');
create table mother(mother,child);
insert into mother values('grace','amy');
insert into mother values('amy','fred');
insert into mother values('carolI','carolII');
insert into mother values('carolII','carolIII');
create view parent(parent,child) as
  select * from father
  union
  select * from mother;
```

```
create or replace view ancestor(ancestor,descendant) as
  select parent,child from parent
  union
  select parent,descendant from parent,ancestor
   where parent.child=ancestor.ancestor;
```

The two example queries above can be formulated in SQL as:

```
select * from ancestor where ancestor='tom';

select child,father,mother
 from father,mother
 where father.child=mother.child;
```

And also as RA queries as:

```
/ra select ancestor='tom' (ancestor);

project child,father,mother
  (father zjoin father.child=mother.child mother);
```

6.5 Basic Recursion Problem (file `recursion.dl`)

This example is intended to show that queries involving recursive predicates do terminate thanks to DES fixpoint solving, by contrast with Prolog's usual SLD resolution.

```
p(0).
p(X) :- p(X).
p(1).
```

The query **`p(X)`** returns the inferred facts from the program irrespective of the apparent infinite recursion in the second rule. (Note that the Prolog goal **`p(1)`** does not terminate. You can easily check it out with **`/prolog p(1)`**.)

6.6 Transitive Closure (files `tranclosure.{dl,sql,ra}`)

With this example, we show a possible use of mutual recursion by means of a Datalog program that defines the transitive closure of the relations p and q[13]. It can be consulted with **`/c tranclosure`**.

```
p(a,b).
p(c,d).
q(b,c).
q(d,e).
pqs(X,Y) :- p(X,Y).
pqs(X,Y) :- q(X,Y).
pqs(X,Y) :- pqs(X,Z),p(Z,Y).
pqs(X,Y) :- pqs(X,Z),q(Z,Y).
```

The query **`pqs(X,Y)`** returns the whole set of inferred facts that model the transitive closure.

---

[13] Taken from [Diet87].

File **tranclosure.sql** contains the SQL counterpart code, which can be executed with **/process tranclosure.sql**:

```
create table p(x,y);
insert into p values ('a','b');
insert into p values ('c','d');
create table q(x,y);
insert into q values ('b','c');
insert into q values ('d','e');
create view pqs(x,y) as
  select * from p
  union
  select * from q
  union select pqs.x,p.y from pqs,p where pqs.y=p.x
  union select pqs.x,q.y from pqs,q where pqs.y=q.x;
```

The query **select * from pqs** returns the same answer as before.

File **tranclosure.ra** contains the RA formulation:

```
pqs(x,y) :=
   p
  union
   q
  union
   project pqs.x,p.y (pqs zjoin pqs.y=p.x p)
  union
   project pqs.x,q.y (pqs zjoin pqs.y=q.x q);

/ra select true (pqs)
```

6.7 Mutual Recursion (files **mutrecursion.{dl,sql,ra}**)

The following program shows a basic example about mutual recursion:

```
p(a).
p(b).
q(c).
q(d).
p(X) :- q(X).
q(X) :- p(X).
```

Submitting the goal **p(X)**, we get:

```
{
  p(a),
  p(b),
  p(c),
  p(d)
}
Info: 4 tuples computed.
```

which is the same set of values for arguments for the query **q(X)**. The file **mrtc.dl** is a combination of this example and that of the previous section.

The file **mutrecursion.sql** contains the SQL counterpart code, which can be executed with **/process mutrecursion.sql**:

```
/sql
/assert p(a)
/assert p(b)
/assert q(c)
/assert q(d)
-- View q must be given a prototype for view p to be defined
create view q(x) as select * from q;
create or replace view p(x) as select * from q;
create or replace view q(x) as select * from p;
```

Note that it is needed to build a void view for **q** in order to have it declared when defining the view **p**. The void view is then replaced by its actual definition. The contents of both views can be tested to be equal with:

```
select * from p;
select * from q;
```

File **mutrecursion.ra** contains the RA formulation:

```
-- View q must be given a prototype for view p to be defined
q(x) := select true (q);
p(x) := select true (q);
q(x) := select true (p);

select true (p);
select true (q);
```

6.8 Farmer-Wolf-Goat-Cabbage Puzzle (file puzzle.dl)

This example[14] shows the classic Farmer–Wolf–Goat–Cabbage puzzle (also Missionaries and Cannibals as another rewritten form). The farmer, wolf, goat, and cabbage are all on the north shore of a river and the problem is to transfer them to the south shore. The farmer has a boat which he can row taking at most one passenger at a time. The goat cannot be left with the wolf unless the farmer is present. The cabbage, which counts as a passenger, cannot be left with the goat unless the farmer is present. The following program models the solution to this puzzle. The relation **state/4** defines the valid states under the specification (i.e., those situations in which there is no danger for any of the characters in our story; a state in which the goat is left alone with the cabbage may result in an eaten cabbage) and imposes that there is a previous valid state from which we depart from. The arguments of this relation are intended to represent (from left to right) the position (north –**n**– or south –**s**– shore) of the farmer, wolf, goat, and cabbage. We use the relation **safe/4** to verify that a given configuration of positions is valid. The relation **opp/2** simply states that north is the opposite shore of south and vice versa.

```
% Initial state
state(n,n,n,n).
% Farmer takes Wolf
state(X,X,U,V) :-
  safe(X,X,U,V),
```

[14] Adapted from [Diet87].

```
  opp(X,X1),
  state(X1,X1,U,V).
% Farmer takes Goat
state(X,Y,X,V) :-
  safe(X,Y,X,V),
  opp(X,X1),
  state(X1,Y,X1,V).
% Farmer takes Cabbage
state(X,Y,U,X) :-
  safe(X,Y,U,X),
  opp(X,X1),
  state(X1,Y,U,X1).
% Farmer goes by himself
state(X,Y,U,V) :-
  safe(X,Y,U,V),
  opp(X,X1),
  state(X1,Y,U,V).

% Opposite shores (n/s)
opp(n,s).
opp(s,n).

% Farmer is with Goat
safe(X,Y,X,V).
% Farmer is not with Goat
safe(X,X,X1,X) :- opp(X,X1).
```

If we submit the query **`state(s,s,s,s)`**, we get the expected result:

```
{
  state(s,s,s,s)
}
Info: 1 tuple computed.
```

That is, the system has proved that there is a serial of transfers between shores which finally end with the asked configuration (this problem is not modeled to show this serial). If we ask for the extension table contents regarding the relation **`state/4`** (with the command **`/list_et state/4`**), we get for the answers:

```
{
  state(n,n,n,n),
  state(n,n,n,s),
  state(n,n,s,n),
  state(n,s,n,n),
  state(n,s,n,s),
  state(s,n,s,n),
  state(s,n,s,s),
  state(s,s,n,s),
  state(s,s,s,n),
  state(s,s,s,s)
}
Info: 10 tuples in the answer set.
```

This is the complete set of valid states which includes all of the valid paths from **`state(n,n,n,n)`** to **`state(s,s,s,s)`**. However, the order of states to reach the latter is not given, but we can find it by observing this relation, i.e.:

```
state(n,n,n,n) → Farmer takes Goat to south shore →
state(s,n,s,n) → Farmer returns to north shore →
state(n,n,s,n) → Farmer takes Wolf to south shore →
state(s,s,s,n) → Farmer takes Goat to north shore →
state(n,s,n,n) → Farmer takes Cabbage to south shore →
state(s,s,n,s) → Farmer returns to north shore →
state(n,s,n,s) → Farmer takes Goat to south shore →
state(s,s,s,s)   Final safe state
```

Observe that there is two states in the relation **state/4** which do not form part of the previous path:

```
state(s,n,s,s)
state(n,n,n,s)
```

These states come from another possible path:[15]

```
state(n,n,n,n) → Farmer takes Goat to south shore →
state(s,n,s,n) → Farmer returns to north shore →
state(n,n,s,n) → Farmer takes Cabbage to south shore →
state(s,n,s,s) → Farmer takes Goat to north shore →
state(n,n,n,s) → Farmer takes Wolf to south shore →
state(s,s,s,n) → Farmer takes Goat to north shore →
state(s,s,n,s) → Farmer returns to north shore →
state(n,s,n,s) → Farmer takes Goat to south shore →
state(s,s,s,s)   Final safe state
```

6.8.1 Dealing with paths (file `puzzle1.dl`)

As just illustrated, the sequence of movements needed to find a feasible solution can be inferred from the answer table. Nonetheless, it is possible to outcome such sequences even when there is no provision for data structures. The idea is to code sequences of movements into a single plain type, as an integer. We can resort, for instance, to build a decimal number whose digits, as read from *right to left*, indicate the selected movement in the sequence. If we number the movement alternatives from 1 to 4 (in the same order as rules occur at the program text) the first solution above can be coded as 2412342, and the second one as 2432142.

Modeling in this way, we can rewrite the predicate state by adding a first argument as the sequence needed to reach a given state, and the stetps already performed. This is useful to build the code as adding a number (identifying the alternative rule) multiplied by the n-th power of ten, where n is the number of steps already done. The following two example rules illustrates this:

```
% 0. Initial state
state(0,0,n,n,n,n).
% 1. Farmer takes Wolf
state(C,S,X,X,U,V) :-
  safe(X,X,U,V),
```

[15] Remember that the system returns *all* of the possible solutions.

```
  opp(X,X1),
  state(C1,S1,X1,X1,U,V),
  S is S1+1,
  bound(B),
  S<B,
  C is C1+1*10**S1.
```

Solving the new program yields:

```
DES> state(C,S,s,s,s,s)
{
  state(2412342.0,7,s,s,s,s),
  state(2432142.0,7,s,s,s,s)
}
Info: 2 tuples computed.
```

Which is explained as follows:

```
* Solution 1: state(2412342.0,7,s,s,s,s)
0: Initial state
   North: Farmer,Goat,Cabbage,Wolf
   South: empty
2: Farmer takes goat to the South shore
   North: Cabbage,Wolf
   South: Farmer,Goat
4: Farmer returns to North shore
   North: Farmer,Cabbage,Wolf
   South: Goat
3: Farmer takes cabbage to the South shore
   North: Wolf
   South: Farmer,Cabbage,Goat
2: Farmer takes goat to the North shore
   North: Farmer,Goat,Wolf
   South: Cabbage
1: Farmer takes wolf to the South shore
   North: Goat
   South: Farmer,Cabbage,Wolf
4: Farmer returns to North shore
   North: Farmer,Goat
   South: Cabbage,Wolf
2: Farmer takes goat to the South shore
   North: empty
   South: Farmer,Goat,Cabbage,Wolf

* Solution 2: state(2432142.0,7,s,s,s,s)
0: Initial state
   North: Farmer,Goat,Cabbage,Wolf
   South: empty
2: Farmer takes goat to the South shore
   North: Cabbage,Wolf
   South: Farmer,Goat
4: Farmer returns to North shore
   North: Farmer,Cabbage,Wolf
   South: Goat
1: Farmer takes wolf to the South shore
```

```
   North: Cabbage
   South: Farmer,Goat,Wolf
2: Farmer takes goat to the North shore
   North: Farmer,Goat,Cabbage
   South: Wolf
3: Farmer takes cabbage to the South shore
   North: Goat
   South: Farmer,Cabbage,Wolf
4: Farmer returns to North shore
   North: Farmer,Goat
   South: Cabbage,Wolf
2: Farmer takes goat to the South shore
   North: empty
   South: Farmer,Goat,Cabbage,Wolf
```

6.9 Paradoxes (files `russell.{dl,sql,ra}`)

When negation is used, we can find paradoxes, such as the Russell's paradox (the barber in a town shaves every person who does not shave himself) shown in the next example (please note that this example is not stratified and, in general, we cannot ensure correctness for non-stratifiable programs):

```
DES> /verbose on
Info: Verbose output is on.

DES> /c russell
Info: Consulting russell...
  shaves(barber,M) :-
    man(M),
    not(shaves(M,M)).
  man(barber).
  man(mayor).
  shaved(M) :-
    shaves(barber,M).
  end_of_file.
Info: 4 rules consulted.
Info: Computing predicate dependency graph...
Info: Computing strata...
Warning: Non stratifiable program.
```

If we submit the query **`shaves(X,Y)`**, we get the positive facts as well as a set of undefined inferred information (in our example, whether the barber shaves himself), as follows (here, verbose output is enabled):

```
DES> shaves(X,Y)
Warning: Unable to ensure correctness for this query.
{
  shaves(barber,mayor)
}
Info: 1 tuple computed.
Undefined:
{
  shaves(barber,barber)
```

```
}
Info: 1 tuple undefined.
```

If we look at the extension table contents by submitting the command **/list_et**, we get as answers:

```
Answers:
{
  man(barber),
  man(mayor),
  not(shaves(mayor,mayor)),
  shaves(barber,mayor)
}
Info: 4 tuples in the answer set.
```

We can see that, in particular, we have proved additional negative information (the mayor does not shaves himself) and that no information is given for the undefined facts. The current implementation uses an incomplete algorithm for finding such undefined facts. We can see this incompleteness by adding the following rule:

```
shaved(M) :- shaves(barber,M).
```

The query **shaved(M)** returns:

```
Warning: Unable to ensure correctness for this query.
{
  shaved(mayor)
}
Info: 1 tuple computed.
```

That is, the system is unable to prove that **shaved(barber)** is undefined.

If you look at the predicate dependency graph and the stratification of the program:

```
DES> /pdg

Nodes: [man/1,shaved/1,shaves/2]
Arcs : [shaves/2-shaves/2,shaves/2+man/1,shaved/1+shaves/2]

DES> /strata

[non-stratifiable]
```

you get the predicate dependency graph shown in Figure 4, and you are informed that the program is non-stratifiable. This figure shows a negation in a cycle, so that the program is not stratifiable. (The system warned of this situation when the program was loaded.)

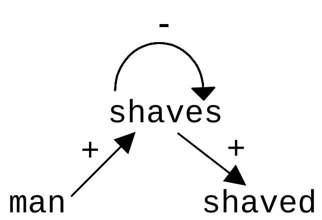


Figure 4. Predicate Dependency Graph for **russell.dl**

However, even when a program is non-stratifiable, there may exist a query with an associated predicate dependency subgraph so that negation does not occur in any cycle. For instance, this occurs with the query **man(X)** in this program:

```
DES> man(X)
Info: Stratifiable subprogram found for the given query.
{
  man(barber),
  man(mayor)
}
Info: 2 tuples computed.
```

Here, the system recomputed the strata for the predicate dependency subgraph, and informed that it found a stratifiable subprogram for such a query. In this simple case, no more negations were involved in the subgraph, but more elaborated dependencies can be found in other examples (cf. Sections 6.10 and 6.11).

Stratification may be needed for programs without negation as long as a temporary view contains a negated goal. Consider the following view under the program **relop.dl** (rules in the program with negation are not present in the subgraph for the query **d(X)**):

```
DES> d(X) :- a(X), not(b(X))
Info: Processing:
  d(X) :- a(X),not(b(X)).
{
  d(a2),
  d(a3)
}
Info: 2 tuples computed.
```

In this view, the query **d(X)** is solved with a solve-by-stratum algorithm, described in Section 5.17.3. In this case, this means that the goal **b(X**) is solved before obtaining the meaning of **d(X)** because **b** is in a lower stratum than **d** and it is needed for the computation of **d**.

The basic paradox **p:-not(p)** can be found in the file paradox.dl, whose model is undefined as you can test with the query **p**.

6.10 Parity (file parity.dl)

This example program[16] is intended to compute the parity of a given base relation **br(X)**, i.e., it can determine whether the number of elements in the relation (cardinality) is even or odd by means of the predicates **br_is_even**, and **br_is_odd**, respectively. The predicate **next** defines an ascending chain of elements in **br** based on their textual ordering, where the first link of the chain connects the distinguished node **nil** to the first element in **br**. The predicates **even** and **odd** define the even, resp. odd, elements in the chain. The predicate **has_preceding** defines the elements in **br** such that there are previous elements to a given one (the first element in the

[16] Adapted from [ZCF+97].

chain has no preceding elements). The rule defining this predicate includes an intended error (fourth rule in the example) which will be used in Section 6.13 to show how it is caught by the declarative debugger.

```
% Pairs of non-consecutive elements in br
between(X,Z) :-
  br(X), br(Y), br(Z), X<Y, Y<Z.

% Consecutive elements in the sequence, starting at nil
next(X,Y) :-
  br(X), br(Y), X<Y, not(between(X,Y)).
next(nil,X) :-
  br(X), not(has_preceding(X)).

% Values having preceding values in the sequence
has_preceding(X) :-
  br(X), br(Y), Y>X. %error: Y>X should be Y<X

% Values in an even position of the sequence, including nil
even(nil).
even(Y) :-
  odd(X), next(X,Y).

% Values in an odd position of the sequence
odd(Y) :-
  even(X), next(X,Y).

% Succeeds if the cardinality of the sequence is even
br_is_even :-
  even(X), not(next(X,Y)).

% Succeeds if the cardinality of the sequence is odd
br_is_odd :-
  odd(X), not(next(X,Y)).

% Base relation
br(a).
br(b).
```

6.11 Grammar (file `grammar.dl`)

Parsers can also be coded as Datalog programs. In this example[17], a simple left-recursive grammar analyser is coded for the following grammar rules.

A –> a

A –> Ab

A –> Aa

It was tested with the input string "ababa", which is coded with the relation **`t(F,T,L)`**, **`F`** for the position of token **`T`** that ends at position **`L`**.

[17] Taken from [FD92].

```
t(1,a,2).
t(2,b,3).
t(3,a,4).
t(4,b,5).
t(5,a,6).
a(F,L) :- t(F,a,L).
a(F,L) :- a(F,M), t(M,b,L).
a(F,L) :- a(F,M), t(M,a,L).
DES> a(1,6)
{
  a(1,6)
}
Info: 1 tuple computed.
```

6.12 Fibonacci (file `fib.{dl,sql,ra}`)

The all-time classics Fibonacci program[18] can be coded in DES thanks to arithmetic built-ins. It can be formulated as follows:

```
fib(0,1).
fib(1,1).
fib(N,F) :-
  N>1,
  N2 is N-2,
  fib(N2,F2),
  N1 is N-1,
  fib(N1,F1),
  F is F2+F1.
```

Since DES is implemented with extension tables, computing high Fibonacci numbers is possible with linear complexity:

```
DES> fib(1000,F)
{
fib(1000,7033036771142281582183525487718354977018126983635873274
2604905087154537118196933579742249494562611733487750449241765991
0881863632654502236471060120533741212738673391111981393731255987
67690091902245245323403501)
}
Info: 1 tuple computed.
```

Also, it is possible to formulate this in SQL, even when the next view features non-linear recursion (file `fib.sql`):

```
create view fib(n,f) as
  select 0,1
  union
  select 1,1
  union
  select fib1.n+1,fib1.f+fib2.f
  from fib fib1, fib fib2
```

[18] Taken from [FD92].

```
  where fib1.n=fib2.n+1 and fib1.n<10;
```

As well, next there is a possible RA formulation (file **fib.ra**):

```
fib(n,f) :=
  project 0,1 (dual)
  union
  project 1,1 (dual)
  union
  project fib1.n+1,fib1.f+fib2.f
  (rename fib1(n1,f1) (fib)
   zjoin
   n1=n2+1 and n1<10
   rename fib2(n2,f2) (fib));
```

6.13 Hanoi Towers (file `hanoi.dl`)

Another well-known toy puzzle is the towers of Hanoi, which can be coded as:

```
hanoi(1,A,B,C).
hanoi(N,A,B,C) :-
  N>1,
  N1 is N-1,
  hanoi(N1,A,C,B),
  hanoi(N1,C,B,A).
```

We can submit the following query for 10 discs:

```
DES> hanoi(10,a,b,c)
{
  hanoi(10,a,b,c)
}
Info: 1 tuple computed.
```

Note that the answer to this query does not reflect the movements of the discs, which can be otherwise shown as the intermediate results kept in the extension table:

```
DES> /list_et hanoi
Answers:
{
  hanoi(1,a,c,b),
  hanoi(1,b,a,c),
  hanoi(1,c,b,a),
  hanoi(2,a,b,c),
  hanoi(2,b,c,a),
  hanoi(2,c,a,b),
  hanoi(3,a,c,b),
  hanoi(3,b,a,c),
  hanoi(3,c,b,a),
  hanoi(4,a,b,c),
  hanoi(4,b,c,a),
  hanoi(4,c,a,b),
  hanoi(5,a,c,b),
  hanoi(5,b,a,c),
  hanoi(5,c,b,a),
```

```
  hanoi(6,a,b,c),
  hanoi(6,b,c,a),
  hanoi(6,c,a,b),
  hanoi(7,a,c,b),
  hanoi(7,b,a,c),
  hanoi(7,c,b,a),
  hanoi(8,a,b,c),
  hanoi(8,b,c,a),
  hanoi(8,c,a,b),
  hanoi(9,a,c,b),
  hanoi(9,c,b,a),
  hanoi(10,a,b,c)
}
Info: 27 tuples in the answer set.
...
```

6.14 Other Examples

Directory examples include some other examples as the files **bom.dl** (bill of materials) and **trains.dl** (train connections) which show more example applications including negation. Other examples are **orbits.dl** (a cosmos tiny database), **sg.dl** (same generation for a family database), **tc.dl** (transitive closure), and **empTraining.{ra,sql}** (taken from [Diet01]). Also, the folder **persistent** contains examples for persisting predicates, the folder **ontology** includes examples of authoring ontologies, including some documentation, and folders **DLDebugger** and **SQLDebugger** include examples for debugging Datalog programs and SQL views, respectively.

7. Contributions

This section collects the contributions from external developers up to now:

- Test Case Generator.
  Authors: Rafael Caballero-Roldán, Yolanda García-Ruiz, and Fernando Sáenz-Pérez
  Date: 10/2009 (upgraded version supported since DES 1.8.0)
  Description: Tool for generating test cases for SQL views
  License: LGPL
  Contact: Yolanda García-Ruiz (Implementor)
- Datalog Declarative Debugger.
  Authors: Rafael Caballero-Roldán, Yolanda García-Ruiz, and Fernando Sáenz-Pérez
  Date: 5/2007
  Description: Tool for the declarative debugging of Datalog programs
  License: LGPL
  Contact: Yolanda García-Ruiz (Implementor)
- ACIDE (A Configurable Development Environment).
  Authors: Diego Cardiel Freire, Juan José Ortiz Sánchez, Delfín Rupérez Cañas (SI 2006/2007), Miguel Martín Lázaro (SI 2007/2008), and Javier Salcedo Gómez (SI 2010/2011), Pablo Gutiérrez García-Pardo, Elena Tejeiro Pérez de Ágreda, Andrés Vicente del Cura (SI 2012/2013) led by Fernando Sáenz.
  Date: 3/2007 (ACIDE 0.1, first version), 11/2008 (ACIDE 0.7), 7/2011 (ACIDE 0.8),

12/2012 (ACIDE 0.9, current version)
Description: This project is aimed to provide a multiplatform configurable integrated development environment which can be configured in order to be used with any development system such as interpreters, compilers and database systems. Features of this system include: project management, multifile editing, syntax colouring, and parsing on-the-fly (which informs of syntax errors when editing programs prior to the compilation).
License: GPL.
Project Web Page: **`http://acide.sourceforge.net/`**

- Emacs development environment.
  Author: Markus Triska.
  Date: 2/22/2007
  Description: Provides an integration of DES into Emacs. Once a Datalog file has been opened, you can consult it by pressing F1 and submit queries and commands from Emacs. This works at least in combination with SWI Prolog (it depends on the –s switch); other systems may require slight modifications.
  License: GPL.
  Project Web Page: **`http://stud4.tuwien.ac.at/~e0225855/index.html`**
  Contact: **`markus.triska@gmx.at`**
  Installation: Copy **`des.el`** (in the contributors web page) to your home directory and add to your **`.emacs`**:

```
(load "~/des")
; adapt the following path as necessary:
(setq des-prolog-file "~/des/systems/swi/des.pl")
(add-to-list 'auto-mode-alist '("\\.dl$" . des-mode))
```

  Restart Emacs, open a **`*.dl`** file to load it into a DES process (this currently only works with SWI Prolog). If the region is active, F1 consults the text in the region. You can then interact with DES as on a terminal.

8. Related Work

There has been a high amount of work around deductive databases [RU95] (its interest delivered many workshops and conferences for this subject) which dealt to several systems. However, to the best of our knowledge, there is no a friendly system oriented to introducing deductive databases with several query languages to students. Nevertheless, on the one hand, we can comment some representative deductive database systems. On the other hand, also some technological transfers to face real-world problems.

8.1 Deductive Database Systems

4QL [MS11] is a recent development of a rule-based database query language with negation allowed in bodies and heads of rules, which is founded on a four-valued semantics with truth values: true, false, inconsistent and unknown. It provides means for a uniform treatment of Open and Local Closed World, other nonmonotonic/commonsense formalisms, including various variants of default reasoning, autoepistemic reasoning and other formalisms application-specific disambiguation of inconsistent information, including defeasible reasoning.

ConceptBase [JJNS98] is a multi-user deductive object manager mainly intended for conceptual modeling and coordination in design environments. It is multiplatform, object-oriented, it enjoys integrity constraints, database updates and several other interesting features.

The LDL project at MCC lead to the LDL++ system [AOTWZ03], a deductive database system with features such as X-Y stratification, set and complex terms, database updates and aggregates. It can be currently used through Internet using a Java–enabled client.

DLV [FP96] is a multiplatform system for disjunctive Datalog with constraints, true negation (à la Gelfond & Lifschitz) and queries. It includes the K planning system, a frontend for abductive diagnosis and Reiter's diagnosis, support for inheritance, and an SQL front-end which prototypes some novel SQL3 features. DLV^{DB} is an extension of DLV which provides interfaces with relational databases, taking advantage of their efficient implementations to speed-up computations.

XSB [RSSWF97] (**`http://xsb.sourceforge.net/`**) is an extended Prolog system that can be used for deductive database applications. It enjoys a well–founded semantics for rules with negative literals in rule bodies and implements tabling mechanisms. It runs both on Unix/Linux and Windows operating systems. Datalog++ [Tang99] is a front-end for the XSB deductive database system.

bddbddb [WL04] stands for BDD-Based Deductive DataBase. It is an implementation of Datalog that represents the relations using binary decision diagrams (BDDs). BDDs are a data structure that can efficiently represent large relations and provide efficient set operations. This allows bddbddb to efficiently represent and operate on extremely large relations.

IRIS (Integrated Rule Inference System) [IRIS2008] is a Java implementation of an extensible reasoning engine for expressive rule-based languages provided as an API. Supports safe or un-safe Datalog with (locally) stratified or well-founded negation as failure, function symbols and bottom-up rule evaluation.

Coral [RSSS94] is a deductive system with a declarative query language that supports general Horn clauses augmented with complex terms, set–grouping, aggregation, negation, and relations with tuples that contain (universally quantified) variables. It only runs under Unix platforms. There is also a version which allows object–oriented features, called Coral++ [SRSS93].

FLORID (F-LOgic Reasoning In Databases) [KLW95] is a deductive object-oriented database system supporting F-Logic as data definition and query language. With the increasing interest in semistructured data, Florid has been extended for handling semistructured data in the context of Information Integration from the Semantic Web.

The NAIL! project delivered a prototype with stratified negation, well–founded negation, and modularity stratified negation. Later, it added the language Glue, which is essentially single logical rules, with SQL statements wrapped in an imperative conventional language [PDR91, DMP93]. The approach of combining two languages is similar to the aforementioned Coral, which uses C++. It does not run on Windows platforms.

Another deductive database following this combination of declarative and imperative languages is Rock&Roll [BPFWD94].

ADITI 2 [VRK+91] is the last version of a deductive database system which uses the logic/functional programming language Mercury. It does not run on Windows platforms. There is no further development planned for Aditi.

See also the Datalog entry in Wikipedia (**`http://en.wikipedia.org/wiki/Datalog`**).

8.2 Technological Transfers

Datalog has been extensively studied and is gaining a renowned interest thanks to their application to ontologies [FHH04], semantic web [CGL09], social networks [RS09], policy languages [BFG07], and even for optimization [GTZ05]. 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.

The first commercial oriented deductive database system was the Smart Data System (SDS) and its declarative query language Declarative Reasoning (DECLARE) [KSSD94], with support for stratified negation and sets. Currently, XSB and DLV have been projected to spin-off companies and they develop deductive solutions to real-world problems.

9. Future Enhancements

The following list (in order of importance) suggests some points to address for enhancing DES:

- Disjunctive heads
- Information about cycles involving negation in the loaded program
- Complete algorithm for finding undefined information
- Constraints (reals, integers, enumerated types)
- Precise error reporting for SQL and Datalog syntax errors

If you find worthwhile for your application either some of the points above, or others not listed, please inform the author for trying to guide the implementation to the most demanded points.

10. Caveats and Limitations

- Datalog:
    - No compound terms as arguments in user relations
    - Termination is ensured up to arithmetic. There is no provision for numerical bounds
    - No database updates via Datalog rules are allowed
    - Rules in consulted files must end with a dot, in contrast to command prompt inputs in single-line mode, which the dot is optional. Rules in a

consulted file may span on multiple lines and ending dot is mandatory, irrespective the multi-line mode

- SQL:
  - User identifiers (including tables, views, column names) are case sensitive
  - Some incorrect SQL statements are not rejected (as those containing a `GROUP BY` clause and columns in the projection list which do not occur in the grouping list). Rather, they either raise exceptions at run-time or return non-ground answers
  - Computable SQL statements follow the grammar in Section 4.2.8 of this manual. The current grammar parses extra clauses which cannot be computed yet (e.g., `ANY`, ...)
  - View definitions in a WITH clause are global, in contrast to the SQL standard
  - Some DBMS's as IBM DB2 via an ODBC connection use uppercase user identifiers, even when they are declared in lowercase. The
  - See also Section 5.1.7 regarding ODBC connections
- SQL debugger:
  - SQL debugging is not supported for ODBC connections, up to now
- Test case generator:
  - Test case generation is not supported for ODBC connections, up to now
- SQL tracer:
  - SQL tracing is not supported for ODBC connections, up to now
- Miscellanea:
  - Enabling duplicates can notably harm performance (cf. Fibonacci example)
  - Users should not write predicate identifiers starting with the symbol '$'. Otherwise, unexpected behaviour might happen
  - Batch processing cannot be nested
- Prolog systems' specific issues:
  - SWI-Prolog distributions do not allow arithmetic expressions involving `log/2`

11. Release Notes

This section lists release notes of the current DES version.

11.1 Version 3.3.1 of DES (released on July, 28th, 2013)

- Enhancements:
  - No longer need to load a file for debugging Datalog goals

- Less memory requirements for debugging Datalog goals
- Expected modes for predicates with unsafe rules are automatically added
- Some dead code removal
- New commands:
  - **`/restore_default_status`** Restore the status of the system to the initial status, i.e., sets all user-configurable flags to their initial values, including the default database and the start-up directory
  - **`/list_modes`** List the expected modes for unsafe predicates in order to be correctly computed. Modes can be 'i' (for an input argument) and 'o' (for an output argument)
  - **`/list_modes`** ***`Name`*** List expected modes, if any, for predicates with name ***`Name`*** in order to be correctly computed
  - **`/list_modes`** ***`Name/Arity`*** List expected modes, if any, for the given predicate ***`Name/Arity`*** in order to be correctly computed
- New port to SWI-Prolog 6.4.1

- Changes:
  - Fixpoint iterations displayed in statistics show the number of iterations in **`solve_star`** (*solve** [Diet87])
  - Disabled EDB optimization
  - Current file system path is not displayed when changing it unless verbose mode is enabled
  - Unified predicate existence message for **`/debug_datalog`**
  - Added current database info to **`/status`**
- Fixed bugs:
  - Incorrect message display when trying to successively disable a multi-value flag
  - Uppercase letters in the extended set of symbols were not parsed
  - Non-recursive predicate optimization was not compatible with Datalog debugging

12. Acknowledgements

The author wishes to thank J. Wielemaker for providing such an amazing free Prolog system and supporting help. Mats Carlsson and Per Mildner, at SICS, supported the development providing help and new capabilities in the ODBC library. Also, thanks to all the people providing feedback, since they are guiding DES to suit more demanded requirements. Contributors are specially acknowledged: Markus Triska, for developing the Emacs IDE and also author of the SWI-Prolog clpfd library, and the students Diego Cardiel Freire, Juan José Ortiz Sánchez, Delfín Rupérez Cañas, Miguel Martín Lázaro, Javier Salcedo Gómez, Pablo Gutiérrez García-Pardo, Elena Tejeiro Pérez de Ágreda, and Andrés Vicente del Cura, who developed and improved

ACIDE. Thanks to Yolanda García and Rafael Caballero for making possible to declarative debug Datalog and SQL databases. They are also key authors in the inclusion of test case generation for SQL views. In particular, Yolanda took the implementation effort supported by Rafael. Gabriel Aranda López and Sonia Estévez Martín generated Mac OSX Snow Leopard and Leopard executables, resp. for versions up to DES 2.6. Enrique Martín Martín fixed the Linux distribution of DES 1.5.0. Finally, thanks to the Spanish projects FAST-STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD-UCM (UCM-BSCH-GR35/10-A-910502) which supported this work.

Appendix A. License

A.1 Software License

DES licensing comes from the ideas of the Free Software Foundation. Since version 3.0, it is distributed under version 3 of the GNU Lesser General Public License (LGPL), which supplements version 3 of the GNU General Public License.

DES is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

DES is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

DES versions prior to 3.0 were distributed under GNU General Public License (GPL).

A.2 Documentation License

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being
LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[Agra88] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", IEEE Transactions on Software Engineering archive, Volume 14 Issue 7, July 1988.

[AO08] P. Ammann and J. Offutt, "Introduction to Software Testing", Cambridge University Press, 2008.

[AOTWZ03] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo, "The deductive database system LDL++", TPLP, 3(1):61–94, 2003.

[BFG07] M. Becker, C. Fournet, and A. 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.

[BPFWD94] M.L. Barja, N.W. Paton, A. Fernandes, M.H. Williams, A. Dinn, "An Effective Deductive Object–Oriented Database Through Language Integration", In Proc. of the 20th VLDB Conference, 1994.

[Caba05] Caballero, R., A declarative debugger of incorrect answers for constraint functional-logic programs, in: WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming (2005), pp. 8–13.

[CGL09] A. Calì, G. Gottlob, and T. 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.

[CGS06b] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, "Towards a Set Oriented Calculus for Logic Programming", Programación y Lenguajes, P. Lucio y F. Orejas (editors), CIMNE, pp. 41-50, Barcelona, Spain, September, 2006.

[CGS07] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, "A New Proposal for Debugging Datalog Programs", 16th International Workshop on Functional and (Constraint) Logic Programming, 2007.

[CGS08] 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), LNCS 4925, pp. 143-159, Springer, 2008.

[CGS10a] R. Caballero, Y. García-Ruiz and F. Sáenz-Pérez, "Applying Constraint Logic Programming to SQL Test Case Generation", In 10th International Symposium on Functional and Logic Programming (FLOPS 2010), 2010.

[CGS11b] R. Caballero, Y. García-Ruiz and F. Sáenz-Pérez, "Algorithmic Debugging of SQL Views", Eigth Ershov Informatics Conference, PSI'11, Novosibirsk, Akademgorodok, Russia, June, 2011.

[CGS12a] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, "Declarative Debugging of Wrong and Missing Answers for SQL Views", In 11th International Symposium on Functional and Logic Programming (FLOPS 2012), Springer, Lecture Notes in Computer Science, Kobe, Japan, May, 2012.

[Chan78] C.L. Chang, "Deduce 2: Further Investigations of Deduction in Relational Databases", H. Gallaire and J. Minker (eds.), Logic and Databases, Plenum Press, 1978.

[Diet87] S.W. Dietrich, "Extension Tables: Memo Relations in Logic Programming", IV IEEE Symposium on Logic Programming, 1987.

[Diet01] S.W. Dietrich, "Understanding Relational Database Query Languages,", Prentice Hall, 2001.

[DMP93] M. Derr, S. Morishita, and G. Phipps, "Design and Implementation of the Glue–NAIL Database System", In Proc. of the ACM SIGMOD International Conference on Management of Data, pp. 147–167, 1993.

[Drax92] Draxler, Chr., A Powerful Prolog to SQL Compiler, CIS-Bericht-92-61, Centrum für Informations und Sprachverarbeitung, Ludwig-Maximilians-Universität München, 1992.

[FD92] C. Fan and S. W. Dietrich, "Extension Table Built-ins for Prolog", Software - Practice and Experience Vol. 22 (7), pp. 573-597, July 1992.

[FHH04] R. Fikes, P.J. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the Semantic Web. J. Web Sem., 2(1):19–29, 2004.

[FP96] Wolfgang Faber and Gerald Pfeifer. DLV homepage, since 1996. url **`http://www.dlvsystem.com/`**.

[GR68] C.C. Green and B. Raphael, "The Use of Theorem–Proving Techniques in Question–Answering Systems", Proceedings of the 23rd ACM National Conference, Washington D.C., 1968.

[GTZ05] S. Greco, I. Trubitsyna, and E. Zumpano. NP Datalog: A Logic Language for NP Search and Optimization Queries. Database Engineering and Applications Symposium, International, 0:344–353, 2005.

[GUW02] H. Garcia-Molina, J. D. Ullman, J. Widom, "Database Systems: The Complete Book", Prentice-Hall, 2002.

[HA92] M. A. W. Houtsma and P. M. G. Apers, " Algebraic optimization of recursive queries", Data & Knowledge Engineering, Volume 7 Issue 4, March 1992.

[IRIS2008] IRIS-Reasoner, **`http://iris-reasoner.org`**.

[ISO00] ISO/IEC. ISO/IEC 132111-2: Prolog Standard. 2000.

[JGJ+95] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, S. Eherer: ConceptBase - a deductive object base for meta data management. In Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases, Vol. 4, No. 2, 167-192, 1995.

System available at: **http://www-i5.informatik.rwth-aachen.de/CBdoc/**

[KLW95] M. Kifer, G. Lausen, J. Wu, "Logical Foundations of Object Oriented and Frame Based Languages", Journal of the ACM, vol. 42, p. 741-843, 1995.

[KSSD94] W. Kiessling, H. Schmidt, W. Strauss, and G. Dünzinger, "DECLARE and SDS: Early Efforts to Commercialize Deductive Database Technology", VLDB Journal, 3, pp. 211–243, 1994.

[KT81] C. Kellogg and L. Travis, "Reasoning with Data in a Deductively Augmented Data Management System", H. Gallaire, J. Minker, and J. Nicolas (eds.), Advances in Data Base Theory, Volume 1, Plenum Press, 1981.

[Lloy87] J. Lloyd, "Foundations of Logic Programming", Springer Verlag, 1987.

[Mink87] J. Minker, "Perspectives in Deductive Databases", Technical Report CS–TR–1799, University of Maryland at College Park, March 1987.

[MN82] J. Minker and J.–M. Nicolas, "On Recursive Axioms in Deductive Databases, Information Systems", 16(4):670–702, 1991.

[MS11] J. Małuszyński and A. Szałas: Living with Inconsistency and Taming Nonmonotonicity. To appear in Datalog 2.0, G. Gottlob, G. Grasso, O. de Moor, and A. Sellers, eds., LNCS 6702, 334-398, Springer-Verlag, 2011.

[PDR91] G. Phipps, M. A. Derr, and K.A. Ross, "Glue–NAIL!: A Deductive Database System". In Proc. of the ACM SIGMOD Conference on Management of Data, pp. 308–317, 1991.

[Robi65] J.A. Robinson, "A Machine–Oriented Logic Based on the Resolution Principle", Journal of the ACM, 12:23–41, 1965.

[RS09] R. Ronen and O. 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.

[RSSS94] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The Coral deductive system. VLDB Journal, 3(2):161–210, 1994.

[RSSWF97] P. Rao, Konstantinos F. Sagonas, Terrance Swift, David Scott Warren, and Juliana Freire, "XSB: A System for Efficiently Computing WFS", Logic Programming and Non–monotonic Reasoning, 1997.

[RU95] R. Ramakrishnan and J.D Ullman, "A Survey of Research on Deductive Database Systems", Journal of Logic Programming, 23(2): 125–149, 1995.

[SD91] C. Shih and S. W. Dietrich, "Extension Table Evaluation of Datalog Programs with Negation", Proceedings of the IEEE International Phoenix Conference on Computers and Communications, Scottsdale, AZ, March 1991, pp. 792-798.

[Sae07] F. Sáenz-Pérez, "ACIDE: An Integrated Development Environment Configurable for LaTeX", The PracTeX Journal, 2007, Number 3, ISSN 1556-6994, August, 2007.

[Shap83] Shapiro, E., "Algorithmic Program DeBugging", ACM Distinguished Dissertation, MIT Press, 1983.

[SICStus] SICS, **`http://www.sics.se/sicstus`**.

[Silv07] Silva, J., A Comparative Study of Algorithmic Debugging Strategies, in: Proc. of International Symposium on Logic-based Program Synthesis and Transformation LOPSTR 2006, 2007, pp. 134–140.

[SRSS93] D. Srivastava, R. Ramakrishnan, S. Sudarshan, and P. Seshadri, "Coral++: Adding Object–Orientation to a Logic Database Language", Proceedings of the International Conference on Very Large Databases, 1993.

[Tang99] Z. Tang, "Datalog++: An Object-Oriented Front-End For The Xsb Deductive Database Management System", http://citeseer.ist.psu. edu/tang99datalog.html.

[TS86] H. Tamaki and T. Sato, "OLD Resolution with Tabulation", Proceedings of ICLP'86, Lecture Notes on Computer Science 225, Springer–Verlag, 1986.

[Ullm95] J.D. Ullman. Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies), Computer Science Press, 1995.

[VRK+91] J. Vaghani, K. Ramamohanarao, D.B. Kemp, Z. Somogyi, and P.J. Stuckey, "Design Overview of the Aditi Deductive Database System", In Proc. of the 7th Intl. Conf. on Data Engineering, pp. 240–247, 1991.

[Wiele] J. Wielemaker, **`http://www.SWI-Prolog.org`**.

[WL04] J. Whaley and M. Lam, Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In: Prog. Lang. Design and Impl., 2004.

[ZCF+97] C. Zaniolo, S. Ceri, C. Faloutsos, T.T. Snodgrass, V.S. Subrahmanian, and R. Zicari, "Advanced Database Systems", Morgan Kauffmann Publishers, 1997.

[ZF97] U. Zukowski and B. Freitag, "The Deductive Database System LOLA", In: J. Dix and U. Furbach and A. Nerode (Eds.). Logic Programming and Nonmonotonic Reasoning. LNAI 1265, pp. 375–386. Springer, 1997.