Deductive Databases

Jorge Galicia Auyón
Victor Flores García

18/12/2015

Index

An Overview
Deductive databases where born as an outgrowth of the field of logic programming in which mathematical logic is used to perform sophisticated inferences and draw appropriate conclusions while handling large amounts of data.

During the 1970's, based on the procedural interpretation of Horn clauses, and the need to develop a specialized theorem prover, the declarative language PROLOG (Programming in Logic) was developed in the university of Marseilles. By mid 1970's a compiler that could help translate Prolog clauses into instructions of abstract machine was developed and served as the basis for "question-answering systems".

Using logic programming as a query language would eventually result in the integration to a database management system. The resulting systems combined a declarative style for formulating queries and constraints.

In a traditional DBMS, an explicit procedural program code must be written to support declarative rules. However, since rules may change over the years, it is more convenient to change the declared deduction rules than record procedural programs. In this way, deductive databases not only store explicit information in the same way a relational database, but also stores rules, which enable inferences to be made based on the original stored data.

The term deductive database highlights the ability to use a logic programming style for expressing deductions concerning the contents of a database.

When is it recommended to use Deductive Databases?
Deductive databases provide additional expressive power to relational databases when it comes to recursive queries.

When you try to answer questions such as "Given a bill of materials with a certain unitary cost, what is the total cost to build a BMW motorcyle at today's prices?" , or, "Given that you can get from airport A to B and from B to C, can you book a plane from A to C?" the only option is by adding recursion. Recursive queries are useful to build hierarchy trees. Usually this involves joining a set with itself a defined number of times. Each recursive call is another separate call that will have to be joined into the total results.

Overall, deductive databases can range in a wide variety of applications:
In decision support for example, given a finite number of resources and an estimated future state, one can make inferences and take better informed decisions about future plans.

In the mining industry, large amounts of data may be collected through samples which then can be analyzed to see if there is large enough probability of finding a desired mineral on a specific region.

In biology, deductive databases have been used for protein structure analysis. By restraining user defined levels of similarities between three dimensional and topological structures, the system can use a function to automatically search for similar structures.

Mathematical Basis

Deductive database technology stands at the intersection of database, logic programming and artificial intelligence systems technologies. Their common denominator (regardless of the language being used) is *logic*. In order to understand how deductive databases work, some mathematical background is needed:

Propositional calculus

The propositional calculus is concerned with logical propositions which can have only true or false value. Prepositions (elementary statement of propositional calculus) are combined into complex statements using connectives and, or, not and implication. It is important to mention one inference rule called modus ponens, which says that if we know the proposition p, and p implies the proposition q, then we can conclude that we know q. This is:

$$\begin{gathered} p \rightarrow q \\ p \\ \therefore q \end{gathered}$$

The database in these systems consists of the truth values assigned to the various elementary prepositions known to the system.

First-order predicate calculus

The essential idea is that each proposition can be defined as a predication. Predication consists of a list of predicates. For instance Predicate(arg_1, arg_2, arg_3…arg_n) (P must be with capital letter and the arguments with a small letter). For example:

The train 110 goes to Brussels, can be transformed as:

`Go(110, brussels)` In this case the predicate has 2 arguments.

A simple predicate is one that assigns some property to the argument:

```
Train 110 is a TGV.
TGV(110)
```

More complex predicates represent relations between 2 arguments.

`Send(jane, paul,letter)` which can be read as Jane sends Paul a letter

Note that the order in the argument structure is important.

`Mother(lucie,diana)` represents Lucie is the mother of Diana.
`Mother(diana,lucie)` represents Diana is the mother of Lucie.

In both first order logic and regular propositional calculus, propositions are analogous to the tuples in a relational database system. Each proposition is grouped into predicates; these predicates are the relations in the Relational Database Systems. First order predicate calculus is a propositional system with the addition of variables and quantifiers. Two special symbols called quantifiers can appear in formulas, these are **universal quantifier (∀)** and **existential quantifier (∃).**

Considering the following example of a bus company that registers the cities to which each bus travel.

Propositions	Predicate
All buses from Paris go to Brussels.	`Go(X,paris,brussels)`
Some buses from Paris go to Brussels.	`Go(X,paris,brussels)`
Buses don't go from Paris to Brussels.	`Go(X,paris,brussels)`

In all three cases the predications represent identical relations between entities but this doesn't reflect the different meanings. The mechanism that allows the formalization of relations between sets and not between simple entities is the use of quantifiers. Both quantifiers return a TRUE or FALSE value according to the following rules:

- If F is a formula, then the formula$(\exists t)(F)$, where t is a tuple variable. The formula is TRUE if the formula F evaluates to TRUE for *some* (at least one) tuple assigned to free occurences of t in F.
- If F is a formula, then the formula $(\forall t)(F)$, where t is a variable tuple. The formula is TRUE if the formula F evaluates TRUE for every tuple in the universe. Otherwise FALSE is returned.

<u>Transforming the Universal and Existential Quantifiers</u>
We now introduce some transformations from mathematical logic that relate the universal and existential quantifiers. This is essential since for many RDBMS the universal quantifier is still not implemented. In general this transformations follow this principle: transform one type of quantifier into the other with negation; AND and OR replace one another; a negated formula becomes unnegated. Consider the following list of special transformations:

$(\forall x)\big(P(x)\big) \equiv \boldsymbol{NOT}\ (\exists x)(\boldsymbol{NOT}\ \big(P(x)\big))$
$(\exists x)\big(P(x)\big) \equiv \boldsymbol{NOT}\ (\forall x)(\boldsymbol{NOT}\ \big(P(x)\big))$
$(\exists x)(P(x)\ \boldsymbol{AND}\ Q(x)) \equiv \boldsymbol{NOT}\ (\forall x)(\boldsymbol{NOT}\ \big(P(x)\big)\ \boldsymbol{OR}\ \boldsymbol{NOT}\big(Q(x)\big))$
$(\exists x)(P(x)\ \boldsymbol{OR}\ Q(x)) \equiv \boldsymbol{NOT}\ (\forall x)(\boldsymbol{NOT}\ \big(P(x)\big)\ \boldsymbol{AND}\ \boldsymbol{NOT}\big(Q(x)\big))$
$(\forall x)(P(x)\ \boldsymbol{AND}\ Q(x)) \equiv \boldsymbol{NOT}\ (\exists x)(\boldsymbol{NOT}\ \big(P(x)\big)\ \boldsymbol{OR}\ \boldsymbol{NOT}\big(Q(x)\big))$
$(\forall x)(P(x)\ \boldsymbol{OR}\ Q(x)) \equiv \boldsymbol{NOT}\ (\exists x)(\boldsymbol{NOT}\ \big(P(x)\big)\ \boldsymbol{AND}\ \boldsymbol{NOT}\big(Q(x)\big))$

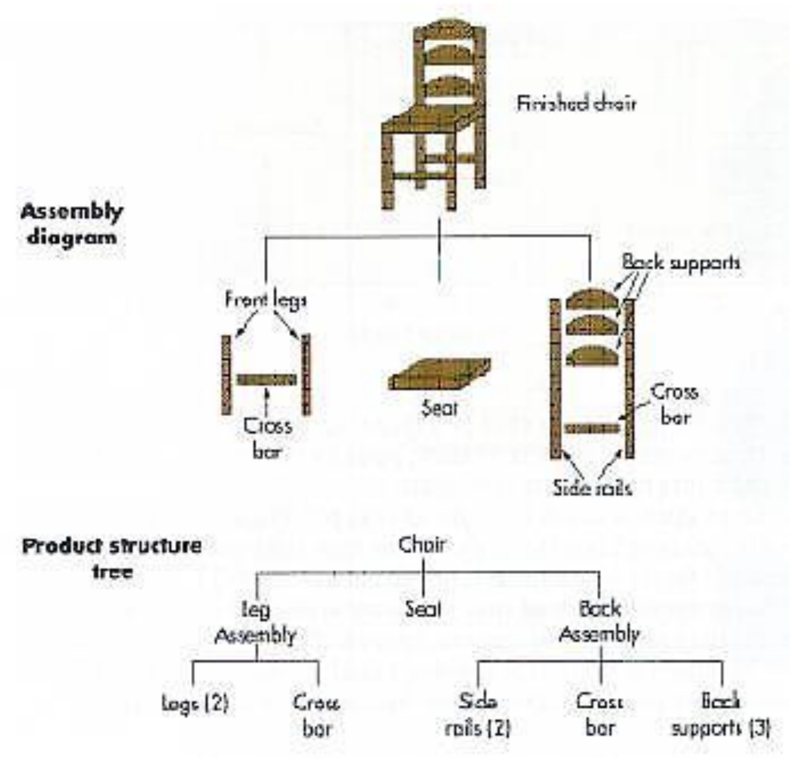
Programming Language for Deductive Databases
The language used to specify rules in a deductive database system is called declarative language. A declarative language is characterized to specify what to achieve rather than how to achieve it. They are completely different from imperative languages because in this kind of languages you will tell the machine how to do something and as a result you will get what you wrote was supposed to happen. In a declarative language you will tell the machine what you want to compute and you let her figure out how to compute the result.

For many years, Prolog was by far the most popular logic programming language.

The notation used is based on providing predicates with unique names:

As it was described before, a predicate has an implicit meaning which is suggested by the predicate name and a fixed number of arguments. The arguments can be variables or constant values. If the arguments are constant values the predicate states that a fact is true. When it has constants and variables, the predicate is considered as a query or as part of a rule. All predicate names and variables start with an upper case letter. The constant values of character type must be identified with a lowercase letter.

Consider a database that contains all assembling parts to produce one chair.



Facts: ASSEMBLY(component1, component2, qty)
```
ASSEMBLY(chair, leg_assembly,1)
ASSEMBLY(chair, back_assembly,1)
ASSEMBLY(chair, seat,1)
ASSEMBLY(leg_assembly,leg,2)
ASSEMBLY(leg_assembly,crossbar,1)
ASSEMBLY(back_assembly, side_rails,2)
ASSEMBLY(back_assembly,cross_bar,1)
ASSEMBLY(back_assembly,back_support,3)
```

Rules

```
COMPONENT(X,Y) :- ASSEMBLY(X,Y,Qty)
COMPONENT(X,Y) :- ASSEMBLY(X,Z,Qty), COMPONENT(Z,Y,Qty)
SUBPART(X,Y) :- ASSEMBLY(Y,X)
```

Queries

```
COMPONENT(back_assembly,Y)?
COMPONENT(back_assembly,leg)?
```

There are three predicate names, *assembly, component and subpart*.

Facts

The ASSEMBLY predicate is defined via a set of facts, each of which has three arguments: a part name, followed by the direct subpart in which the component is subdivided. The quantity expresses the number of subparts needed to produce the part. These facts correspond to the actual data that is stored in the database, and they can be considered as a set of tuples in a relation ASSEMBLY with three attributes whose schema is:

```
ASSEMBLY(Part, Subpart, Qty).
```

Thus, ASSEMBLY(X,Y,Z) states the fact that part X is assembled with Z subparts of Y. Notice the omission of the attribute names in the Prolog notation. The names are only represented by the position of each argument in the predicate.

Rules

The other two predicates are defined by rules. The ability to specify recursive rules is one of the most important contributions of this notation. A rule is of the form head :- body, where :- is read as if and only if. A rule has a single predicate in the left of the :- symbol called the head. In the right side you can have one or more predicates called the body or premise of the rule. When a predicate contains only constants, it is called to be ground.

A rule specifies that if a particular assignment of constant values to the variables in the body makes all the predicates true, it also makes the head true by using the same assignment of constant values to variables. A rule provides us with a way of generating new facts that are instantiations of the head of the rule. These new facts are based on facts that already exist, corresponding to instantiations of predicates on the body rule. It's important to remark that a comma in the body implicitly apply the AND operator to these predicates.

Recursive rules

Consider the definition of COMPONENT rule, whose first argument is a part name and whose second argument is either a direct or an indirect subpart of the part. By indirect subpart we mean the subpart of a subpart down to any number of levels. Thus, COMPONENT(X,Y) stands for the fact that X is a superior of Y through direct or indirect supervision.
The predicate component is defined using two rules. The first rule stands all the direct subparts for X component. The second recursive rule states that if A(X,Z) and

COMPONENT(Z,Y) are both true, then the rule body predicates in the body side is the same rule as the head. In general, the rule body defines a number of premeiss such that if they are all true, we can deduce a conclusion that is also true. If there exists two or more rules with the same head, it is equivalent to the use of OR logical operator.

Built-in predicates
Prolog system contains a number of built-in predicates that the system can interpret directly. These typically include the equality comparison, less or equal to, more or equal to (=, <=, >=,>,<). This comparison operators can be treated as binary predicates.

Query
A query involves a predicate symbol with some variable arguments, and its answer is to deduce all the different constant combinations that, when bound ro the variables, can make the predicate true. For example, in first query

```
COMPONENT(back_assembly,Y)?
```

We are interested in all the subparts that form the back_assembly part. When a query has only constant symbols as arguments it returns either true or false.

Prolog was at first expected to be used as a database language, but some of its features were not suited to database applications [Particularly, Prolog can be quite sensitive to the order of statements. For instance, in Prolog, the rule “If A and B are true, then C is true” might not be the same as the logically equivalent “If B and A are true, then C is true”] and this motivated the definition of an alternative database and logic programming language, known as **Datalog**.

Datalog vs Prolog
Syntactically, Datalog is very similar to Prolog. The evolution from Prolog to Datalog assumes going from a record-oriented language (solved one answer at a time) to a set-oriented language (solved by computing its meaning once). This transformation was parallel to the evolution of database systems from the hierarchical and network data models to the relational data model.

Datalog is a significant extensión of relational algebra and calculus. Datalog is equivalent to relational algebra but with an additional fixpoint operator.

In prolog, the programmer is responsible for the efficiency of the queries since he can decide to order the rules in an optimal way, in Datalog this is not possible, because the user might query any predicate and it is therefore the responsibility of the system to determine a good execution order.
Additionally, In contrast to Prolog:

- In a Datalog program the order of statements literals does not matter and can be stated in any order.
- Datalog queries on finite sets are guaranteed to terminate, in Prolog, however, recursion can lead to non-termination
- Datalog doesn't allow function symbols in arguments, for example, p (1, 2) is admissible but p (f (1), 2) is not
- imposes certain stratification restrictions on the use of negation and recursion,
- requires that every variable that appears in the head of a clause also appears in a nonarithmetic positive (i.e. not negated) literal in the body of the clause,
- requires that every variable appearing in a negative literal in the body of a clause also appears in some positive literal in the body of the clause

Generally, algorithms in deductive databases are divided in 2: bottom-up and top-down (also known as backward chaining). Bottom-up algorithms generate logical consequences of the database until all answers to the goal are found. On the other hand, Top-down algorithms start with the goal and reduce it to subgoals.

Particularly for Prolog, bottom-up method is less efficient than top-down method because many facts that have nothing in common with the original problem are derived, therefore, Prolog uses top-down evaluation as a standard method of computation.

In terms of efficiency, when comparing a typical Prolog top-down execution with the Datalog bottom-up execution, generally, the bottom-up evaluation reduces overall costs in terms of the number of I/O operations since bottom-ups perform on a set-oriented computation in contrast to tuple oriented computation applied in top-down methods. Additionally, bottom-up methods are able to avoid infinite loops by detecting possibly cyclic subgoals and therefore termination is guaranteed.

The main disadvantage of bottom-up algorithms is that bottom-up algorithms are not goal-oriented. Thus, in certain situations it computes the entire deductive closure of a program in order to answer a particular question, therefore the search can involve a lot of irrelevant computation.

Datalog Particularities

<u>Safety</u>

In Datalog, two things are needed from a rule in order to avoid infinite results:

- Every variable in the head of the rule must also be present in a non-arithmetic positive literal in the body
- Every variable in a negative literal of the body must also be in some positive literal in the body

<u>Negation</u>

Using negation when dealing with recursive queries, semantics can be unclear.
For example:

```
T(b).
G(x) :- T(x) , not S(x).
S(x) :- T(x) , not G(x).
```

What happens when querying on G(b) or S(b)?

The poblem is that there could be two minimal fixpoints for this program: If rule 1 is applied first then R(b) will be in G(b) but if Rule 2 is applied first, it will be in S(b).

T depends on S if some rule with T in the head contains S or (recursively) some predicate that depends on S, in the body.

Therefore a stratification rule is needed: If T depends on not S, then S cannot depend on T (or not T).

If a program is stratified, the tables in the program can be partitioned into strata:
ƒ
Stratum 0: All database tables.
Stratum I: Tables defined in terms of tables in Stratum I and lower strata.
ƒ
If T depends on not S, S is in lower stratum than T

Semantics of a stratified program, therefore is given as:
First, compute the least fixpoint of all tables in Stratum 1. (Since stratum 0 tables are already fixed.)
Then, compute the least fixpoint of tables in Stratum 2; then the least fixpoint of tables in Stratum 3, and so on, stratum-by-stratum

Several extensions have been made to Datalog: to support aggregate functions, to allow object-oriented programming, or to allow disjunctions (or) as heads of clauses. These extensions have significant impacts on the definition of Datalog's semantics and on the implementation of a corresponding Datalog interpreter.
While any conjunctive query can be written as a datalog rule, not every datalog program can be written as a conjunctive query. In fact, only single rules over extensional predicate symbols can be easily rewritten as an equivalent conjunctive query. The problem of deciding whether for a given datalog program there is an equivalent nonrecursive program (corresponding to a positive relational algebra query, or, equivalently, a formula of positive existential first-order logic, or, as a special case, a conjunctive query) is known as the datalog boundedness problem and is undecidable

An Alternative: Recursiveness in SQL

Previous versions of SQL could handle recursive queries but since there was no “recursive statement” you could not retrieve more than one ancestry with a single query:

```
SELECT p1.parent AS grandparent
FROM Parent p1, Parent p2
WHERE p1.child = p2.parent
AND p2.child = ’Bart’
```

To find components that are one level deeper you would need another join and the best option was to join the result of query 1 with the table 2. To find all components, you would need as many joins as there are levels in the given instance.
However, since SQL3 (1999) a “With” statement has been added which allows recursion. On a previous versión there was an alternative statement “connect by” however, this statement could not handle cycles. Whenever it detected a cycle, it returned an error.
The general recursive statement is as follows:

```
WITH Ancestor(anc, desc) AS // name given to each recursion
((SELECT parent, child FROM Parent) // original table
UNION
(SELECT a1.anc, a2.desc // recursive query using recursion as from tables
FROM Ancestor a1, Ancestor a2
WHERE a1.desc = a2.anc))
SELECT anc /* select all ancestors from the recursive table, where start point is
’Bart’ */
FROM Ancestor
WHERE desc = ’Bart’;
```

The query will start from an empty table and compute the query until a fixed point operator is met (in a function, a fix point is a point in which the function of x is equal to x itself. For example, the fixpoint of f(x)= x /2 is 0, since 0 is the only point where f(x) and x are equal. Similarly, in a recursive query, the query will stop when no new information is generated). In the first step, parents and children can be deduced from ancestor-descendant relationships, after the first step, in each subsequent steps we use the facts deduced in previous steps to get more ancestor-descendent relationships. The query will stop when no new facts can be proven.

Additionally, some widely used database systems include ideas and algorithms developed for Datalog. For example, the SQL:1999 standard includes recursive queries, and the Magic Sets algorithm (initially developed for the faster evaluation of Datalog queries) is implemented in IBM's DB2.Moreover, Datalog engines are behind specialised database systems such as Intellidimension's database for the semantic web.

Datalog Educational System (DES)

DES is a deductive database system which allows both SQL and Datalog as query languages. This system is targeted to educational purposes, for this reason it won't be the most competitive and efficient software in the market. It is free, open-source, portable, supports extensions to pure Datalog in the form of stratified negation, strong constraints and duplicates. In the SQL mode views are supported.

DES system was born for the need to teach to postgraduate students deductive database concepts. It was first released in 2004, in a moment when no open source systems where on the market. It allows the following

- Nulls and outer joins operators.
- Duplicates and duplicate elimination.
- Datalog and SQL queries sharing the same database.
- Tabling-based deductive engine implementing stratified negation.
- Source levels tracers and declarative debuggers from Datalog and SQL.

At the beginning DES was developed to be used via an interactive command shell. Now a day more appealing environments are available. In the application developed to do this project Java-based IDE ACIDE was used. ACIDE supports syntax coloring, project management, interactive console, configurable buttons and history of transactions.

DES is implemented on top of Prolog interpreter this means that it can be run on any OS supported by such prolog interpreter. There are portable executables that can be run from Windows, Linux and MAC OS.

<u>Languages supported</u>

This platform supports both Prolog ISO and SQL:2008 ISO standards respectively. DES supports the following constructions:

- Aggregates: group_by(Relation, Grouping_Variables, Condition) creates groups from Relation R usuing the variables (columns) in which the group by will take place. In condition expressions as sum(Variable) must be included. If no grouping is needed, expressins as sum(Relatino, Variable, Result) are also admitted.
- Outer join operations: Nulls operations are allowed, for example rj(L,R,C) returns the right outer join of relations L, R that satisfy the join condition C.
- Negation: not(Flight) computes the negation of Flight by means of negation as failure.
- Duplicate Elimination: the syntaxis is distinct(Relation) which computes distinct tuples of Relation when duplicates are enabled.

Simple example creating new tables and doing simple queries.

```
DES> % SQL
DES> CREATE TABLE student(name STRING, last_name STRING, age
INT);
DES> INSERT INTO student VALUES ('Jorge', 'Galicia',26)
Info: 1 tuple inserted.
DES> INSERT INTO student VALUES ('Victor','Flores',31)
Info: 1 tuple inserted.
DES> Select * from student
answer(student.name:string,student.last_name:string,student.a
ge:int) ->
{
answer('Jorge','Galicia',26),
answer('Victor','Flores',31)
}
Info: 2 tuples computed.

DES> % DATALOG
DES> :-type(student(name:string, last_name:string, age:int))
DES> /assert student('Juan','Ruiz',24)
DES> /assert student('David','Beck',20)
DES>  student(X,Y,Z)

{
student('David','Beck',20),
student('Jorge','Galicia',26),
student('Juan','Ruiz',24),
student('Victor','Flores',31)
}
Info: 4 tuples computed.
```

Notice that both tables were read even if two different query languages were used. To introduce any new rule the command /assert is provided. The program above was directly introduced to the system prompt but I could also had been stored in a file and processed with the command /process FileName. If the program you want to consult is a Datalog file, use the command /consult FileName.

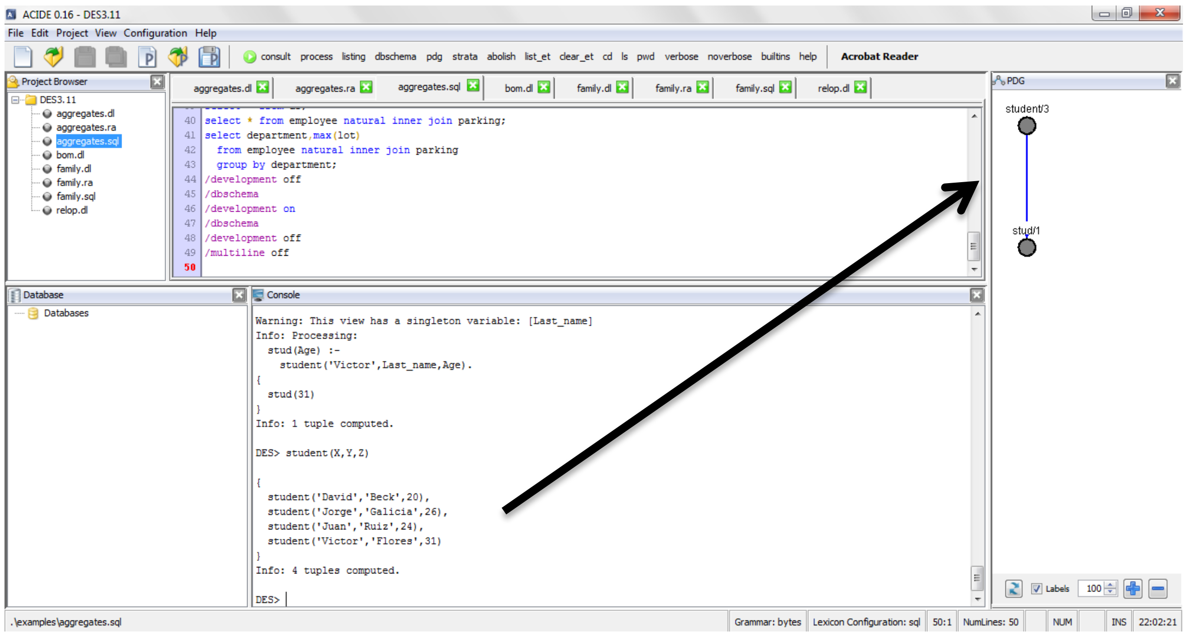
```
DES> % SQL
Select age from student where name='Victor'
answer(student.age:int) ->
{
answer(31)
}
Info: 1 tuple computed.
DES> % DATALOG
DES> student('Victor',X,Y)
{
student('Victor','Flores',31)
}
Info: 1 tuple computed.
```

Note that the first query returns the tuples of student projected by the age argument. To get a similar output relation we use a temporary view in Datalog as follows:

```
stud(Age):-student('Victor',Last_name,Age)
Info: Processing:
stud(Age) :- student('Victor',Last_name,Age).
{
stud(31)
}
Info: 1 tuple computed.
```

In last view we defined both the projection columns and renamed the relations. In ACID, always when making a relation or predicate between two different tables a predicate dependency graph is built. A PDG is a tool to visualize dependencies among predicates. All its nodes are predicates and the arcs represents relationships.

For last view the PDG is this one:



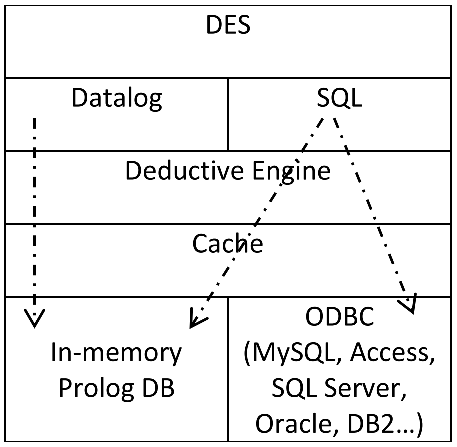
```
select * from employee natural inner join parking;
select department,max(lot)
  from employee natural inner join parking
  group by department;
/development off
/dbschema
/development on
/dbschema
/development off
/multiline off
```

```
Warning: This view has a singleton variable: [Last_name]
Info: Processing:
  stud(Age) :-
    student('Victor',Last_name,Age).
{
  stud(31)
}
Info: 1 tuple computed.

DES> student(X,Y,Z)

{
  student('David','Beck',20),
  student('Jorge','Galicia',26),
  student('Juan','Ruiz',24),
  student('Victor','Flores',31)
}
Info: 4 tuples computed.

DES>
```

.\examples\aggregates.sql | Grammar: bytes | Lexicon Configuration: sql | 50:1 | NumLines: 50 | NUM | INS | 22:02:21

System Architecture of DES

Every Datalog program is stored in what is called an In-memory Prolog Database. All queries are processed in the Deductive Engine, which relies completely on cache memory to store all fixed point computations. The result is then stored in what is called an Extension Table (ET). To display the data results from a Datalog query it is necessary to inspect ET for entries matching the query after its solving.



SQL views and tables are stored using in this two ways:

- Use the same logic as in prolog. Views are translated into Datalog programs and tables into predicates consisting only of facts. SQL queries results are returned as if they were Datalog programs. Results are also cached in an ET and displayed eventually from this table. After the query is performed all information is removed from the ET table.
- Use the ODBC bridge to access external database, and taking advantage from their solving performance, persistency and scalability. When submitting a CREATE VIEW statement, this data is forwarded to the external database through the ODBC connection. After the operations succeeded in the other DBMS, result is returned to DES which does not use the Prolog DB for SQL statements anymore.

What is done when asserting a new Datalog rule in DES?

Many steps are developed when a new rule is defined in Datalog. First, a syntactic tree is built of all valid rules. If there are errors in this phase, an exception is raised with error location and source data.

Second, if validation was correct a preprocessing stage is performing consisting of:

- Simplification of successive applications of not(Goal) and transformation of predicates are needed. For instance =, < are converted in /=, >=. Finally a compound goal is defined as the body of a rule for a new predicate where its arguments are the relevant variables. Finally a compound goal is allowed, for example not(p(X),q(X)) is transformed in the following rule: not (q(X)) where q(X):-p(X),q(X).
- All aggregation predicates including compound goals are computed before the aggregation itself (sum, average, etc).
- Disjunctive bodies are allowed, therefore a rule including disjunctive bodies is transformed. For instance p(X):-q(X);r(X) is translated into p(X):-s(X) and s(X):-q(x) and s(X):-r(X) are added to the program.

<u>Compile-Time Analyses</u>

DES runs constantly compile time analysis to detect unsafe rules. This kind of analysis tries to detect user-defined predicates which are always infinite. For instance, the rule less(X,Y):-X<Y is unsafe because the sign < can be source of infinite data:

```
DES> less(X,Y):-X<Y
Error: X<Y will raise a computing exception at run-time.
Warning: This view is unsafe because of variables: [X,Y]
```

Here is another source of usafety detected by the Compile-Time analysis. In this case, since variable Y is not bound, and all tuples in t/2 are considering for computing his outcome.

```
DES> distinct([X],t(X,Y))

Warning: This view has a singleton variable: [Y]
Info: Processing:
answer(X) :- distinct([X],t(X,Y)).
Warning: Undefined predicate: [t/2]
{
}
Info: 0 tuples computed.
```

How to deal with constraints?
This are one of most important features relational databases offer, primary keys and foreign keys just to mention the most important ones. In deductive database systems that implement stable models, the use of constraints is understood as filters. In these cases, since a database can have several models, only the models that fulfil the constraints are included in the answer. The part of the answer that doesn't meet all the requirements of the constraints is simply not shown as part of the answer.
In DES this constraints are understood in a similar way as in relational database models. In the following example a primary key constraint is entered in the following table:

```
%Datalog
DES> :-type (train(from:string, to:string, duration:int))
DES> /assert train('Bruxelles','Anvers',45)
DES> /assert train('Anvers','Rotterdam',90)
DES> /assert train('Bruxelles','Paris',80)
DES> /assert train('Paris','Nice',180)

DES> :-pk(train,[to,from]) //the primary key is between
attributes to and from.
DES> /assert train('Anvers','Rotterdam',95)
Error: Primary key violation train.[from,to]when trying to
insert: train(Anvers,Rotterdam,95)
```

Let's consider another case of constraints; in this case we will establish a constraint for limiting the duration from an origin to a destination.

Next relation shows the time between two cities that have direct and no direct trains.

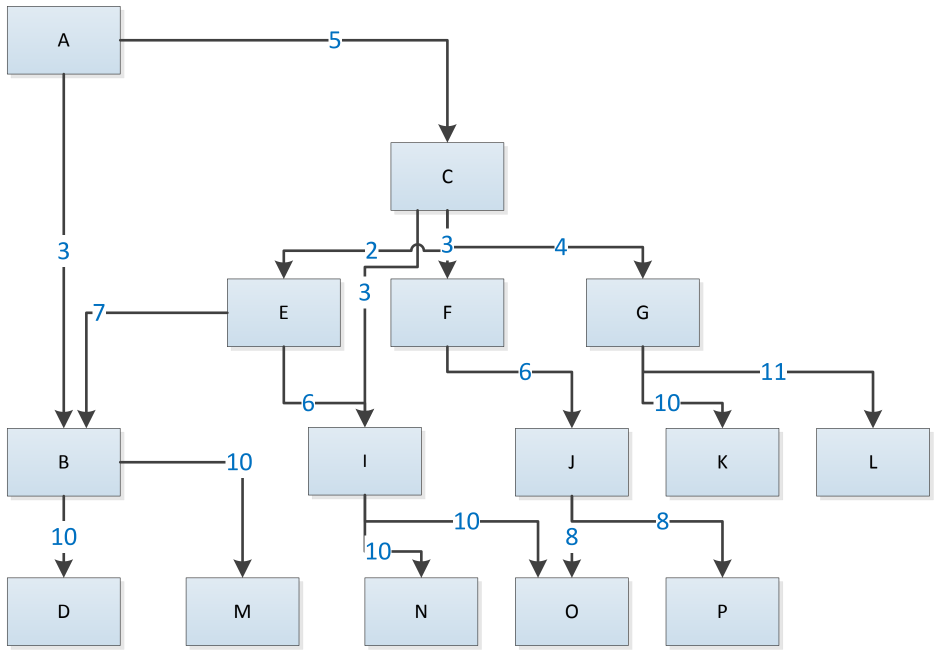
```
DES> /assert connected(X,Y,Z):-train(X,W,Z1),
connected(W,Y,Z2), Z is Z1+Z2
DES> connected('Bruxelles','Nice',Z)
{
  connected('Bruxelles','Nice',260)
}
Info: 1 tuple computed.
DES> :-group_by(connected(X,Y,Z),[X,Y],S=sum(Z)),S>=250 //
this constraint won't let to add new destination time more
than 250 minutes.
```

Tracing and debugging
SQL and Datalog are both declarative high abstraction languages. This means, you tell the machine what you would like to happen and let it figure out how to do it. When a query is demanded, an execution plan for it includes transformations considering data statistics to enhance performance. Therefore, instead of following a more imperative approach to tracint, we focus on a naïve declaration approach which only takes into account the output is correct or not.

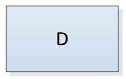
Practical Implementation of a deductive database system
We demonstrate the implementation of deductive databases by the use of 2 practical applications:

Application 1: Bill of Materials BoM
In the following example it is described the BoM (Bill of Materials) required to build a product in a fabric. The BoM provides the manufacturer's part name (Capital letter) and the quantity needed for each component. Consider a car factory, to build component A of car X, components B, C, D, E, F, G, H, I, J, K, L, M, N, O, P are needed in different quantities.

Figure 1: BoM of Product A



<u>For each square:</u>



The capital letter represents the name of the part. The number in each arrow is the quantity of subparts that the part needs to be built. For instance, part B is formed with 10 subparts of D and 10 subparts of M.

For simplicity this example considers a small amount of subparts; however a bigger database can be loaded to perform the same type of queries. The queries will be evaluated with both languages SQL and Datalog.

SQL

Create table

```
CREATE TABLE [dbo].[PartList](
[no] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY CLUSTERED ,
[Part] [varchar](8) NULL,
[subpart] [varchar](8) NULL,
[qty] [int] NULL)
```

This query displays the following table after adding the corresponding values to the table:

No	Part	Subpart	Quantity
1	A	C	5
2	A	B	3
3	C	E	2
4	C	F	3
5	C	G	4
6	C	I	3
7	E	B	7
8	E	I	6
9	F	J	6
10	G	K	10
11	G	L	11
12	B	D	10
13	B	M	10
14	I	O	10
15	I	N	10
16	J	P	8
17	J	O	8

Queries

First Level BoM

Which pieces are necessary to create a piece identified as C? This list will include direct subparts, subparts of subparts, etc. Moreover if a piece is used many times, the subparts will only be used in the list one time.

```
WITH Tempview(PART, SUBPART, QTY) AS
((SELECT PL.PART, PL.SUBPART, PL.QTY FROM PARTLIST PL WHERE
PART='C')
UNION ALL
      SELECT Child.PART, Child.SUBPART, Child.QTY
      FROM Tempview Father, PartList Child
      WHERE Father.SUBPART=Child.PART
      )
SELECT distinct PART, SUBPART, QTY FROM Tempview
ORDER BY PART,SUBPART, QTY
```

The recursive SQL query displays the following table:

Part	Subpart	Quantity
B	D	10
B	M	10
C	E	2
C	F	3
C	G	4
C	I	3
E	B	7
E	I	6
F	J	6
G	K	10
G	L	11
I	N	10
I	O	10
J	O	8
J	P	8

The result can be interpreted as follows: to build part C it is necessary to have 10 parts D which are subparts of B, 10 parts of M which are subparts of B, 2 parts of E which are subparts of C… etcetera.

Query 2
Summarized BoM: In this query it is just asked how many parts of each type are needed to build part C?

```
WITH TempView(PART, SUBPART, QTY) AS
((SELECT PL.PART, PL.SUBPART, PL.QTY FROM PARTLIST PL WHERE
PART='C')
UNION ALL
      SELECT Father.PART, Child.SUBPART, Child.QTY*Father.QTY
      FROM TempView Father, PARTLIST Child
      WHERE Father.SUBPART=Child.PART
)
SELECT distinct PART, SUBPART, SUM(QTY) AS 'TOTAL QTY USED' FROM
TempView
GROUP BY PART, SUBPART
ORDER BY PART,SUBPART
```

A recursive query is needed to display the correct data. Note that a multiplication is done to get the total number of subparts needed to build a given part. For instance, to build C, it is necessary to have 7*2=14 subparts B.

Part	Subpart	Total Qty Used
C	B	14
C	D	140
C	E	2
C	F	3
C	G	4
C	I	15
C	J	18
C	K	40
C	L	44
C	M	140
C	N	150
C	O	294
C	P	144

Query 3: Level Control
In this query the number of levels of necessary parts is restricted. For example, if the question: Which are the first two levels of parts necessaries to create C part?

```
WITH TempView(PLevel, PART, SUBPART, QTY) AS
(SELECT 1,PL.PART, PL.SUBPART, PL.QTY FROM PARTLIST PL WHERE
PART='C'
UNION ALL
SELECT Father.PLevel+1,Child.PART, Child.SUBPART, Child.QTY
FROM TempView Father, PARTLIST Child
WHERE Father.SUBPART=Child.PART
AND Father.PLevel<2
)
SELECT distinct PART,PLevel, SUBPART, QTY FROM TempView
```

Part	Plevel	Subpart	Total Qty Used
C	1	E	2
C	1	F	3
C	1	G	4
C	1	I	3
E	2	B	7
E	2	I	6
F	2	J	6
G	2	K	10
G	2	L	11
I	2	N	10
I	2	O	10

Datalog
Similar to SQL, a table is declared to store the initial data. Each one of this data is called a fact. In DES to store a fact it is necessary to write the keyword /assert before writing the fact to store.

```
:-type (part,[x:string, y:string, z:int]) . //Table is
created
/assert part(a,b,3) .
/assert part(a,c,5) .
/assert part(c,e,2) .
/assert part(c,f,3) .
/assert part(c,g,4) .
/assert part(c,i,3) .
/assert part(e,b,7) .
/assert part(e,i,6) .
/assert part(f,j,6) .
/assert part(g,k,10) .
/assert part(g,l,11) .
/assert part(b,d,10) .
/assert part(b,m,10) .
/assert part(i,o,10) .
/assert part(i,n,10) .
/assert part(j,p,8) .
/assert part(j,o,8) .
```

Declaration of Views: in this part subparts are declared with three main rules.

```
:-type (subpart,[x:string, y:string ]). //Table is created
/assert subpart(X,Y):-part(X,Y,Z).
/assert subpart(X,Y):-subpart(X,Z), subpart(Z,Y).
```

A view subpart is declared, first rule states that every direct subpart is a subpart of piece X. Next subpart declaration states that every subpart of a subpart is a part of the original part desired. This is called a bottom up rule since it generates logical consequences of the database until all answers to the goal are found.

Query 1: How many subparts does each part have?
This is a simple query in Datalog whose syntax is as follows:

```
group_by(subpart(X,Y),[X],Z=count)
```

The result of last query is:

```
answer(X,Z) :-
group_by(subpart(X,Y),[1],[X],Z=count).
{
answer(a,14),
answer(b,2),
answer(c,13),
answer(e,6),
answer(f,3),
answer(g,2),
answer(i,2),
answer(j,2)
}
Info: 8 tuples computed.
```

Query 2: How many pieces are needed to build A or B parts.

```
:-type (part_n,[a:string, b:string, c:int ]) //Table is
created
/assert part_n(X,W,Z):-part(X,W,Z)
/assert part_n(X,W,Z):-part_n(X,Y,Z1),part_n(Y,W,Z2),Z is
Z1*Z2
group_by(part_n(a,X,Y),[X],Z=sum(Y))
```

The result of last query is:

```
Info: Processing:
answer(X,Z) :-
group_by(part_n(a,X,Y),[2],[X],Z=sum(Y)).
{
answer(b,73),
answer(c,5),
answer(d,730),
answer(e,10),
answer(f,15),
answer(g,20),
answer(i,75),
answer(m,730)
}
Info: 8 tuples computed.
```

First rule states that all direct subparts of element X must be included in the table. The second rule states that indirect subparts of element X must be included in the table. Note that the quantity is the multiplication of the quantity needed to produce the direct and indirect subparts.

Primary Keys in Datalog
Suppose that you want the combination X,Y in part(X,Y,Z) to be unique. It is possible to add this rule in DES using the following syntax:

```
:-pk(part,[x,y])
```

If we try to insert value part(a,b,2) into the table, the primary key constraint doesn't let to insert the value.

```
/assert part(a,b,2)
Error: Primary key violation part.[x,y]
when trying to insert: part(a,b,2)
```

Nullability Constraint
Columns can be imposed to contain a concrete value rather than null. In this case the constraint is as follows:

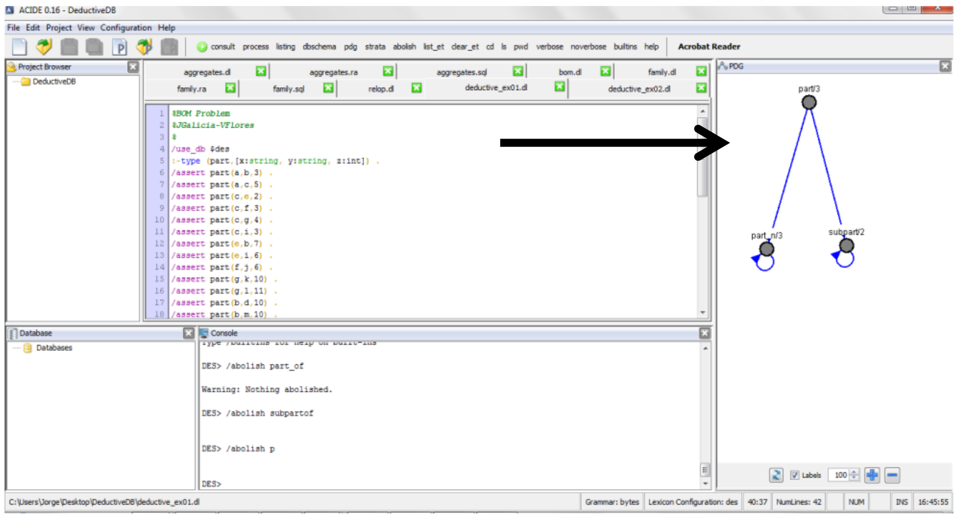
```
:-nn(part,[x])
```

If we want to insert a null value in x column the constraint won't let us insert the value.
Any of the constraints can be dropped using the following statement:

```
/drop_ic name_constraint
```

Predicate dependency graph

It is possible to see the predicate dependency graph of a relation using the PDG window in DES. In next figure the PDG for last relations is shown:



```
%BOM Problem
%JGalicia-VFlores
%
/use_db $des
:-type (part,[x:string, y:string, z:int]) .
/assert part(a,b,3) .
/assert part(a,c,5) .
/assert part(c,e,2) .
/assert part(c,f,3) .
/assert part(c,g,4) .
/assert part(c,i,3) .
/assert part(e,b,7) .
/assert part(e,i,6) .
/assert part(f,j,6) .
/assert part(g,k,10) .
/assert part(g,l,11) .
/assert part(b,d,10) .
/assert part(b,m,10) .
```

```
Type /builtins for help on built-ins

DES> /abolish part_of

Warning: Nothing abolished.

DES> /abolish subpartof


DES> /abolish p


DES>
```

Top to down and bottom up expressions evaluation

DES has an option where run time can be displayed whenever the console launches a query. This option will be used to evaluate the following expressions:

Expression 1:

```
    :-type (subpart,[x:string, y:string ]).
    /assert subpart(X,Y):-part(X,Y,Z).
    /assert subpart(X,Y):-subpart(X,Z), subpart(Z,Y).
```

Expression 2:

```
    :-type (subpart2,[x:string, y:string ]).
    /assert subpart2(X,Y):-part(X,Y,Z).
    /assert subpart2(X,Y):-part(X,Z,W), subpart2(Z,Y).
```

Both expressions display the same answer: the subparts of a determined part X. The first fact will be evaluated using the bottom up approach. Expression two uses top-down approach.

Query	Time in milliseconds	
	Expression 1	Expression 2
`subpart(X,Y)/subpart2(X,Y)`	44	22
`subpart(a,X)/subpart2(a,X)`	53	19
`group_by(subpart(X,Y),[X],Z=count)/ group_by(subpart2(X,Y),[X],Z=count)`	35	30

For this example, using DES system it is much faster to evaluate expressions using top-down approach.

Application 2: Airline System

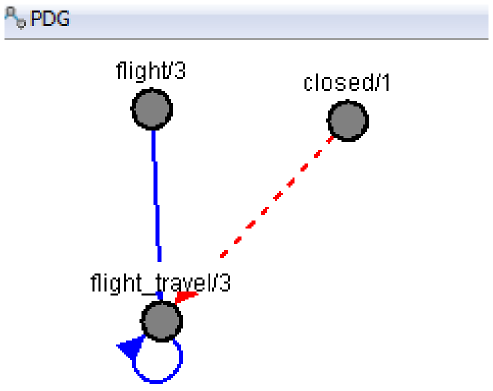
Let's consider the following example, which states flight links that a company offers to their clients (flight/3 for origin, destination and duration) between airports. The company has direct flights to many destinations; moreover the company offers flights with connections to many other destinations. All flight travel of the company are stored in the flight_travel table.

```
:-type(flight,[origin:string,
destination:string,duration:int]). //Table flight is created

:-pk(flight,[origin, destination])

/assert flight('guatemala','mexico',90).
/assert flight('guatemala','panama',120).
/assert flight('guatemala','costa rica',70).
/assert flight('mexico','amsterdam',600).
/assert flight('mexico','frankfurt',660).
/assert flight('mexico','madrid',540).
/assert flight('panama','dubai',1020).
/assert flight('panama','buenos aires',180).
/assert flight('panama','paris',550).
/assert flight('mexico','panama',150).
/assert flight('madrid','paris',120).
/assert flight('paris','london',60).
/assert flight('paris','bruxelles',60).
/assert flight('paris','amsterdam',110).
/assert flight('frankfurt','berlin',45).
/assert flight('frankfurt','rome',120).
/assert flight('frankfurt','oslo',130).
/assert flight('frankfurt','moscou',240).
/assert flight('guatemala','los angeles',250).
/assert flight('los angeles','tokio',840).
/assert flight('dubai','tokio',300).

:-type (flight_travel,[origin: string, destination: string,
duration : int ]). //Create table flight_travel
```

The table flight travel should include direct and connection flights to different cities. The total flight time of a flight_travel must be the sum of times between two or more direct flights involving a connection. It is possible that one flight option is not available, this is modeled with the not closed statements. It will be clarified in the examples.

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

/assert flight_travel(X,Y,Z):-
flight(X,W,Z1),flight_travel(W,Y,Z2),not closed (X), not
closed (Y),not closed (W),Z is Z1+Z2.
```

Predicate dependency graph



Query 1
List all destinations from Guatemala

```
flight_travel('guatemala',X,Y).
{
flight_travel(guatemala,amsterdam,690),
flight_travel(guatemala,amsterdam,780),
flight_travel(guatemala,amsterdam,860),
flight_travel(guatemala,amsterdam,900),
flight_travel(guatemala,berlin,795),
flight_travel(guatemala,bruxelles,730),
flight_travel(guatemala,bruxelles,810),
flight_travel(guatemala,bruxelles,850),
flight_travel(guatemala,'buenos aires',300),
flight_travel(guatemala,'buenos aires',420),
flight_travel(guatemala,'costa rica',70),
flight_travel(guatemala,dubai,1140),
flight_travel(guatemala,dubai,1260),
flight_travel(guatemala,frankfurt,750),
flight_travel(guatemala,london,730),
flight_travel(guatemala,london,810),
flight_travel(guatemala,london,850),
flight_travel(guatemala,'los angeles',250),
flight_travel(guatemala,madrid,630),
flight_travel(guatemala,mexico,90),
flight_travel(guatemala,moscou,990),
flight_travel(guatemala,oslo,880),
flight_travel(guatemala,panama,120),
flight_travel(guatemala,panama,240),
flight_travel(guatemala,paris,670),
flight_travel(guatemala,paris,750),
flight_travel(guatemala,paris,790),
flight_travel(guatemala,rome,870),
flight_travel(guatemala,tokio,1090),
flight_travel(guatemala,tokio,1440),
flight_travel(guatemala,tokio,1560)
}
Info: 31 tuples computed.
```

Query 2
Notice that the query throws all possible travels from Guatemala to any destination. If we want to show only one option having the minimum time to a destination the query is as follows:

```
group_by(flight_travel('guatemala',X,Y),[X],Z=min(Y))
{
answer(amsterdam,690),
answer('buenos aires',300),
answer('costa rica',70),
answer(dubai,1140),
answer(frankfurt,750),
answer('los angeles',250),
answer(madrid,630),
answer(mexico,90),
answer(panama,120),
answer(paris,670),
answer(tokio,1090)
}
Info: 11 tuples computed.
```

Deductive queries
Assumptions can be used in combination with any of the features of DES. Hypothetical queries are common need in several scenarios, related mainly with business intelligence applications. These are called "what if" queries. In last example we defined flight and flight_travel relations using the not closed arguments.

Query 3
Consider that panama airport is closed because there is a big storm. Which are the possible flights going from Guatemala?

```
closed('panama')=>group_by(flight_travel('guatemala',X,Y),[X]
,Z=min(Y))
{
answer(amsterdam,690),
answer('costa rica',70),
answer(frankfurt,750),
answer('los angeles',250),
answer(madrid,630),
answer(mexico,90),
answer(tokio,1090)
}
Info: 7 tuples computed.
```

Note that all flights going from Panama were excluded from the list.

Query 4
Give all possible cities whose possible destination is Tokio.

```
group_by(flight_travel(X,'tokio',Y),[X],Z=min(Y))
{
answer(dubai,300),
answer(guatemala,1090),
answer('los angeles',840),
answer(mexico,1470),
answer(panama,1320)
}
Info: 5 tuples computed.
```

Query 5
Give all possible cities whose possible destination is Tokio considering that Paris and Mexico airports are closed.

```
closed('paris') /\ closed('mexico') =>
group_by(flight_travel(X,'tokio',Y),[X],Z=min(Y))
{
answer(dubai,300),
answer(guatemala,1090),
answer('los angeles',840),
answer(panama,1320)
}
Info: 4 tuples computed.
```

If the city had been Brussels the result would have been empty since to go to Brussels the only city of departure is Paris.

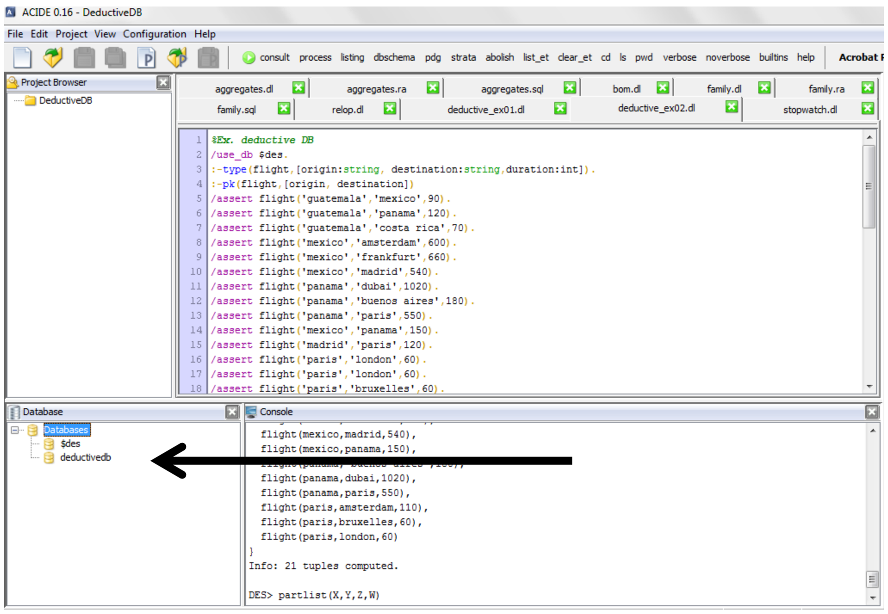
```
closed('paris') /\closed('mexico')=>flight_travel(X,'bruxelles',Z)
answer(X,Z) :-
    closed(paris)
    /\
    closed(mexico)
    =>
    flight_travel(X,bruxelles,Z).
{
}
Info: 0 tuples computed.
```

Charging a Database to DES
This section includes descriptions about the connection to relational database system via ODBC connections, persistence and safety and compatibility. DES provides support for connections to relational database management systems in order to provide data source 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.
Almost any relational database can be accessed from DES using an ODBC connection. ODBC drivers are usually bundled with OS platforms as Windows OS's.
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.

```
      /open_db deductivedb
```

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

```
      /db_schema
      //Result
      Info: Database 'deductivedb'
      Info: Table(s):
* Flight(Origin:nchar(20),Destination:nchar(20),Time:float(8))
* Part(part:nchar(20),subpart:nchar(20),quantity:int(4))
* PartList(no:int
identity(4),Part:varchar(8),subpart:varchar(8),qty:int(4))
 * dual(void:int(4))
Info: View(s):
 * RPL(Part:varchar(8),Subpart:varchar(8),Quantity:int(4))
Info: No integrity constraints.
```



```
%Ex. deductive DB
/use_db $des.
:-type(flight,[origin:string, destination:string,duration:int]).
:-pk(flight,[origin, destination])
/assert flight('guatemala','mexico',90).
/assert flight('guatemala','panama',120).
/assert flight('guatemala','costa rica',70).
/assert flight('mexico','amsterdam',600).
/assert flight('mexico','frankfurt',660).
/assert flight('mexico','madrid',540).
/assert flight('panama','dubai',1020).
/assert flight('panama','buenos aires',180).
/assert flight('panama','paris',550).
/assert flight('mexico','panama',150).
/assert flight('madrid','paris',120).
/assert flight('paris','london',60).
/assert flight('paris','london',60).
/assert flight('paris','bruxelles',60).
```

```
  flight(mexico,madrid,540),
  flight(mexico,panama,150),
  flight(panama,'buenos aires',180),
  flight(panama,dubai,1020),
  flight(panama,paris,550),
  flight(paris,amsterdam,110),
  flight(paris,bruxelles,60),
  flight(paris,london,60)
}
Info: 21 tuples computed.

DES> partlist(X,Y,Z,W)
```

All of these databases can be accessed and queried from DES as if they were local.

```
      flight(X,Y,Z)
      {
      flight(dubai,tokio,300),
      flight(frankfurt,berlin,45),
      flight(frankfurt,moscou,240),
      flight(frankfurt,oslo,130),
      flight(frankfurt,rome,120),
      flight(guatemala,'costa rica',70),
      flight(guatemala,'los angeles',250),
      flight(guatemala,mexico,90),
      flight(guatemala,panama,120),
      flight('los angeles',tokio,840),
      flight(madrid,paris,120),
      flight(mexico,amsterdam,600),
      flight(mexico,frankfurt,660),
      flight(mexico,madrid,540),
      flight(mexico,panama,150),
      flight(panama,'buenos aires',180),
      flight(panama,dubai,1020),
      flight(panama,paris,550),
      flight(paris,amsterdam,110),
      flight(paris,bruxelles,60),
      flight(paris,london,60)
      }
      Info: 21 tuples computed.
```

All queries where developed with small amount of data just for illustration purposes. However if more data want to be added to the model, any database can be charged and then queried utilizing the same principles. Consider that DES was developed to have a simple, interactive, multiplatform and affordable system (*not necessarily efficient)* for students, so that they can get the fundamental concepts behind deductive database with Relational Algebra, Datalog and SQL. 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.

Bibliography:

https://www.fdi.ucm.es/profesor/fernan/FSP/SCG11aSlides.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.260&rep=rep1&type=pdf

http://www.fdi.ucm.es/profesor/fernan/des/html/download.html

http://www.cs.duke.edu/courses/fall04/cps116/lectures/11-recursion.pdf

http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed-english/Ch25_DedDB-95.pdf