Datalog

versión: 2024.1

8 de abril de 2024

Índice

1. Bases de datos deductivas

- Una base de datos deductiva proporciona mecanismos para que el gestor de bases de datos pueda inferir información a partir de la ya existente.
- Las deducciones se realizan a través de inferencias a partir de reglas y hechos.
- Una BD deductiva se compone de:
  - Base de datos extensional: información almacenada explícitamente.
  - Base de datos intensional: reglas que permiten inferir nueva información.
- El modelo de datos deductivo combina el modelo relacional con la programación lógica.
- Características fundamentales:
  - Consultas recursivas.
  - Negación estratificada.
- El modelo es más expresivo que las BDs relacionales, pero menos que los lenguajes de programación lógica.
- Aplicaciones: minería de datos, inteligencia artificial: sistemas expertos, sistemas basados en conocimiento, etc.

2. Datalog

- Datalog es un lenguaje utilizado para representar bases de datos deductivas y realizar consultas sobre las mismas.
- Se puede considerar como una versión restringida (y menos expresiva) que el lenguaje de programación lógica Prolog.
- Existen muchas implementaciones:
  - IRIS (`http://iris-reasoner.org/`)
  - 4QL (`http://4ql.org/`)
  - XSB (`http://xsb.sourceforge.net/`)
- Sistema DES (*Datalog Educational System*):
  - Desarrollado por Fernando Sáenz Perez (FdI - UCM), se distribuye bajo la licencia GNU GPL 3.
  - Disponible en `http://www.fdi.ucm.es/profesor/fernan/des/`
  - Soporta tres lenguajes de consulta:
    - Datalog.
    - Álgebra relacional.
    - SQL (con vistas recursivas).
  - Es un sistema enfocado en la simplicidad y facilidad de uso, no en la eficiencia.
- Comandos básicos del modo consola:

Comando	Significado
/consult nombre_fichero	Carga un fichero (Datalog, RA, o SQL) en la base de datos
/assert hecho_o_regla	Añade un hecho o regla a la base de datos.
/save_ddb nombre_fichero	Exporta la base de datos actual a un fichero Datalog
/cd nombre_dir	Cambia el directorio actual.
/quit	Abandona la consola DES

3. El lenguaje Datalog

- Constantes:
  - Numéricas: 0, 1.6, 23, -24.5
  - Secuencias de caracteres alfanuméricos: pueden contener un carácter de subrayado, deben empezar por una letra minúscula. Ejemplos: `ejemplo, pepe, esto_es_un_ejemplo`
  - Es posible utilizar cadenas de caracteres que no cumplan con estas condiciones escribiéndolas entre comillas. Ejemplos: `'David Fernández', '_vale', '2pasos'`
- Variables: secuencias de caracteres alfanuméricos que comienzan por letra mayúscula o carácter de subrayado. Ejemplos: `X, Nombre, Apellidos, _Edad, _` (esta última es una variable anónima).
- Átomos:
  - Permiten expresar hechos.
  - Símbolo de relación aplicado a una serie de variables o constantes. Ejemplo: `cliente(1, 'Javier', Edad, 2)`
  - Sintaxis: deben empezar por una letra minúscula.
  - Es posible utilizar en los átomos los siguientes operadores: `is, =<, <, >, >=, =, \=`.
  - Se pueden negar átomos: `not padre(javier, pepe)`
- Comentarios:
  - De una línea: `% Esto es un comentario de una línea`
  - De varias líneas: `/* Este puede ocupar varias líneas */`
- Valores `null`.
- Funciones de relación, condiciones, etc.

3.1. Hechos y objetivos

- La base de datos extensional comprende la información almacenada físicamente en la base de datos.
- Los hechos constituyen la base sobre la cual el sistema realiza inferencias.
- Un hecho es un átomo seguido de un punto.
- Ejemplo:

```
cuenta('Javier', 'Herranz', nomina, 12000).
cuenta('Ana', 'Martin', ahorro, 21500).
cuenta('Gerardo', 'de la Iglesia', ahorro, 1200).
cuenta('Manuel', 'Moreno', nomina, 5000).
cuenta('Lucia', 'Rodriguez', ahorro, 5000).
cuenta('Raquel', 'Velasco', null, 100).
```

- El orden de escritura de los hechos no es relevante.
- Las consultas sobre la base de datos se realizan mediante objetivos.
- Un objetivo es el nombre de una relación aplicado a una serie de argumentos. Cada uno de ellos puede ser una constante o una variable.
- Existen dos "modos de uso" de los objetivos:

1. Preguntar si un hecho es cierto.
2. Preguntar qué hechos son ciertos.

- Ejemplos:

  Cuenta de las personas cuyo apellido es 'Moreno':

```
DES> cuenta(Nombre, 'Moreno', Tipo, Saldo)
```

  Cuentas de tipo nomina:

```
DES> cuenta(Nombre, Apellidos, nomina, Saldo)
```

  Cuentas de tipo nómina con 12000 euros de saldo:

```
DES> cuenta(Nombre, Apellidos, nomina, 12000)
```

  Cuentas de aquellas personas cuyo nombre y apellidos coincidan:

```
DES> cuenta(X, X, Tipo, Saldo)
```

- Este conjunto de hechos es lo análogo a una tabla *cuenta* en base de datos relacionales.
- Podemos ver los hechos y los objetivos como funciones booleanas.

3.2. Ajustes

- Una *sustitución* es una función que asocia a variables con constantes u otras variables. Ejemplo:

$$\theta = [X \mapsto 2, Y \mapsto pepe, Z \mapsto V]$$

- Aplicar una sustitución θ a un literal p consiste en reemplazar todas las variables de p por los valores especificados por la sustitución.
- Si una variable de p no esta vinculada a ningún valor en θ, no se reemplaza.
- La aplicación de θ a p se denota por $\theta(p)$. Ejemplo: $\theta(p(X, 13, Z)) = p(2, 13, V)$.
- Decimos que un átomo q se ajusta a p si existe una sustitución θ tal que $\theta(p) = q$. Ejemplo: $p(lucia, 23, Z)$ se ajusta a $p(X, 23, Y)$.
- El mecanismo de Datalog busca en la base de datos hechos (que pueden ser inferidos a partir de otros) de tal forma que se ajusten al objetivo.

3.3. Reglas

- La base de datos intensional contiene la información que se infiere de la ya existente en la base de datos.
- Se representa mediante un conjunto de reglas.
- Sintaxis de las reglas: $\underbrace{atomo}_{cabeza} : - \underbrace{literal_1, literal_2, \ldots, literal_n}_{cuerpo}$
- Si se satisface cada uno de los términos del cuerpo se considera que se satisface la cabeza. O lo que es lo mismo:

Dada la regla: $\underbrace{atomo}_{cabeza} : -\underbrace{literal_1, literal_2, \ldots, literal_n}_{cuerpo}$, si existe una sustitución θ de tal forma que $\theta(literal_1), \ldots, \theta(literal_n)$ sean conocidos (o deducibles) entonces se puede inferir $\theta(atomo)$.

- Este es el mecanismo de inferencia de la información.
- Ejemplo:
  - En una BB.DD. de un banco podemos saber si un cliente es "preferente" si sabemos cuánto dinero tiene en el banco y le decimos al sistema el valor a partir del cual ya se considera preferente.
  - Base de datos extensional:

```
cuenta('Pepe', 'Lopez', ahorro, 1000).
cuenta('Juan', 'Lopez', ahorro, 500).
cuenta('Lucas', 'Lopez', ahorro, 4000).
cuenta('Antonio', 'Lopez', ahorro, 5000).
cuenta('Manuel', 'Moreno', nomina, 5000).
```

  - Base de datos intensional:

```
clientes_gold(N, A) :- cuenta(N, A, ahorro, S), S > 20000.
```

  - Mediante el siguiente objetivo podemos obtener los clientes preferentes:

```
clientes_gold(X, Y).
```

    o consultar si un cliente concreto es preferente o no:

```
clientes_gold('Juan', 'Lopez').
```

- Interpretación lógica de las reglas:
- Una regla

$$atomo : -literal_1, literal_2, \ldots, literal_n$$

  se interpreta como

$$literal_1 \wedge literal_2 \wedge \ldots \wedge literal_n \Rightarrow atomo$$

- Básicamente representa algo deducible a través de una conjunción de hechos.
- Si la regla lógica no posee esta estructura se representa de otra forma equivalente.
- Ejemplo:

$$a \wedge (b \vee c) \Rightarrow p$$

  se representa como (calculado a "ojo"):

```
p :- a, b.
p :- a, c.
```

- Para programar cualquier inferencia que podamos pensar es necesario que tenga cierta forma.
- Forma normal conjuntiva:
  - Una fórmula está en forma normal conjuntiva (FNC) si es una conjunción de disyunción de literales:

- Ejemplos:
  - $(\neg p \vee q) \wedge (\neg q \vee p)$ está en FNC.
  - $(\neg p \vee q) \wedge (\neg q \rightarrow p)$ no está en FNC.
- G es una forma normal conjuntiva de F si está en forma norma conjuntiva y es equivalente a F.
  - Ejemplo: $\neg(p \wedge (q \rightarrow r))$ y $(\neg p \vee q) \wedge (\neg p \vee \neg r)$ son equivalentes.

- Si cada cláusula tiene exactamente un literal positivo (un literal no negado) puede transformarse en una regla.
- Ejemplo:
  - La expresión $a \wedge (b \vee c) \rightarrow p$ no está en FNC.
  - Transformada a FNC queda: $(p \vee \neg a \vee \neg b) \wedge (p \vee \neg a \vee \neg c)$
  - Reglas obtenidas:

```
p :- a, b.
p :- a, c.
```

- Las variables que aparecen en una regla se consideran universalmente cuantificadas en toda la regla:
  - La regla `p(X,Y) :- q(X), r(Y)` se interpreta como $\forall X \forall Y (q(X) \wedge r(Y)) \Rightarrow pp(X, Y)$.
  - En el ejemplo anterior:

```
clientes_gold(N, A) :- cuenta(N, A, ahorro, S), S > 20000.
```

    se entiende que vale para cualquier N, A y S.
  - Además, las variables que aparezcan únicamente en el cuerpo de una regla se pueden considerar también existencialmente cuantificadas en dicho cuerpo.

- Ejemplo:
  - Decimos que dos clientes son "amiguetes" si los saldos de su cuenta coinciden.
  - Es decir, el cliente con nombre N1 y apellidos A1 es amiguete del cliente con nombre N2 y apellidos A2 si existe un número S y sendos tipos T1 y T2, tales que:
    - La cuenta de N1, A1 tiene tipo T1 y saldo S.
    - La cuenta de N2, A2 tiene tipo T2 y saldo S.
  - Su expresión lógica es:

    $\forall N1 \forall A1 \forall N2 \forall A2 (\exists S \exists T1 \exists T2 cuenta(N1, A1, T1, S) \wedge cuenta(N2, A2, T2, S)) \Rightarrow amiguetes(N1, A1, N2, A2)$

    es equivalente a:

    $\forall N1 \forall A1 \forall N2 \forall A2 \forall S \forall T1 \forall T2 cuenta(N1, A1, T1, S) \wedge cuenta(N2, A2, T2, S)) \Rightarrow amiguetes(N1, A1, N2, A2))$

    que es equivalente a:

```
% amiguetes(Nombre1, Apellidos1, Nombre2, Apellidos2)
amiguetes(N1, A1, N2, A2) :-cuenta(N1, A1, T1, S),cuenta(N2, A2, T2, S).
```

- En el ejemplo anterior si hacemos una consulta podemos encontrar algo con el siguiente aspecto:

```
DES> amiguetes(N1, A1, N2, A2)
{
amiguetes('Ana','Martin','Ana','Martin'),
amiguetes('Gerardo','de la Iglesia','Gerardo','de la Iglesia'),
amiguetes('Javier','Herranz','Javier','Herranz'),
amiguetes('Lucia','Rodriguez','Lucia','Rodriguez'),
amiguetes('Lucia','Rodriguez','Manuel','Moreno'),
amiguetes('Manuel','Moreno','Lucia','Rodriguez'),
amiguetes('Manuel','Moreno','Manuel','Moreno'),
amiguetes('Raquel','Velasco','Raquel','Velasco')
}
```

- Cada cliente es amigo de sí mismo.
- Hemos de completar nuestra regla con restricciones de desigualdad:

```
% amiguetes(Nombre1, Apellidos1, Nombre2, Apellidos2)
amiguetes(N1, A1, N2, A2) :-
    cuenta(N1, A1, T1, S),
    cuenta(N2, A2, T2, S), N1 \= N2.
amiguetes(N1, A1, N2, A2) :-
    cuenta(N1, A1, T1, S),
    cuenta(N2, A2, T2, S), A1 \= A2.
```

  Cuestión: ¿por qué dos desigualdades y no una?.

- En el ejemplo anterior observamos que `T1` y `T2` no juegan ningún papel. Es preciso incluirlas exclusivamente por sintaxis: *cuenta* debe tener cuatro argumentos.
- Es posible sustituir estas variables (cuyo nombre es irrelevante) por variables anónimas.
- Las variables anónimas ajustan con cualquier elemento.
- El código anterior queda con la siguiente forma:

```
% amiguetes(Nombre1, Apellidos1, Nombre2, Apellidos2)
amiguetes(N1, A1, N2, A2) :-
    cuenta(N1, A1, _, S),
    cuenta(N2, A2, _, S), N1 \= N2.
amiguetes(N1, A1, N2, A2) :-
    cuenta(N1, A1, _, S),
    cuenta(N2, A2, _, S), A1 \= A2.
```

  Este código es más claro. Permite centrar la atención en las variables relevantes.

- El operador punto y coma permite expresar la disyunción entre dos literales. Esto permite expresar las dos reglas en una:

```
% amiguetes(Nombre1, Apellidos1, Nombre2, Apellidos2)
amiguetes(N1, A1, N2, A2) :-
cuenta(N1, A1, _, S),
cuenta(N2, A2, _, S), (N1 \= N2 ; A1 \= A2).
```

  Es innecesario pero aporta legibilidad.

3.4. Reglas automáticas

- Es posible expresar una consulta mediante una conjunción de literales.
- Ejemplo:

```
cuenta(_, A, nomina, S), S > 10000
```

con este objetivo crea la siguiente regla (y el objetivo correspondiente):

```
answer(A, S) :- cuenta(_, A, nomina, S), S > 10000
```

El aspecto de la consulta podría ser el siguiente:

```
DES> cuenta(_, A, nomina, S), S > 10000
Info: Processing:
    answer(A,S) :-
        cuenta(_,A,nomina,S),
        S>10000.
{
answer('Herranz',12000),
answer('Martin',21500)
}
```

Si sólo se desean conocer los apellidos, pueden utilizarse variables anónimas:

```
DES> cuenta(_, A, nomina, _S), _S > 10000
Info: Processing:
    answer(A) :-
        cuenta(_,A,nomina,_S),
        _S>10000.
{
answer('Herranz'),
answer('Martin')
}
```

3.5. Mecanismo de cómputo de Datalog

Consideremos el siguiente código en Datalog:

```
edad(pepe,13).
edad(juan,18).
edad(maria,23).
edad(ana,19).
amigos(pepe,juan).
amigos(juan,ana).
amigos(maria,juan).
amigos_adultos(X,Y):-amigos(X,Y),edad(X,V),V>17,edad(Y,W),W>17.
```

El usuario lanza un objetivo. Consideremos los siguientes casos:

1. `edad(pepe,13)`: el sistema no puede buscar ningún ajuste porque el objetivo no tiene variables. Solo mira si ese hecho está o no.
2. `edad(pepe,X)`: el sistema busca ajustes que hagan al objetivo cierto. En este caso $\theta = [X \mapsto 13]$. También podíamos haber enviado el objetivo `edad(X,13)`. En tal caso $\theta = [X \mapsto pepe]$.

3. `edad(X,Y)`: el sistema busca ajustes que lo hagan cierto. Hay varios, por ejemplo: $\theta = [X \mapsto maria, Y \mapsto 23]$.

4. `amigos_adultos(juan,ana)`: el sistema plantea el ajuste $\theta = [X \mapsto juan, Y \mapsto ana]$ a cada uno de los literales de la regla:

   - Como es cierto `amigos(juan,ana)` intenta comprobar el siguiente literal. Si hubiera sido falso habría interrumpido el procesamiento y determinado que `amigos_adultos(juan,ana)` es *falso*.
   - A continuación se lanza `edad(juan,V)`. El objetivo busca ajustes que hagan el literal cierto (en este caso $\theta = [V \mapsto 18]$).
   - Pasa al siguiente literal `V>17` (con *V=18*).
   - El proceso continua hasta que todos los literales de la regla sean ciertos o alguno sea falso.

5. `amigos_adultos(W,Z)`: sería similar con la diferencia de que se plantean más ajustes.

3.6. Programando con Datalog

- Programar con Datalog consiste en representar la información con funciones booleanas.
- Hechos y reglas son funciones booleanas. Un hecho es una función booleana constante.
  - Dado un conjunto de objetos, si queremos representar una propiedad booleana de estos objetos utilizamos un función lógica con un argumento.
  - Si se trata de propiedades no booleanas (existe más de un valor), utilizamos una función lógica de dos argumentos (en el que el segundo argumento será el valor de la propiedad). Por ejemplo: `edad(juan,18)`.
  - Si estamos representando representaciones entre objetos utilizaremos funciones booleanas con dos argumentos (si la relación es booleana, o más si no lo es).

3.7. Reglas recursivas

- Las reglas recursivas es uno de los mecanismos más expresivos de Datalog.
- Es la característica que hace que Datalog sea muy potente en términos de expresividad.
- Permiten la especificación de relaciones complejas.

3.7.1. Árboles genealógicos

- Consideremos la genealogía de la figura 1.
- Podemos establecer los siguientes hechos básicos:

```
hombre(abraham).
hombre(clancy).
hombre(herbert).
hombre(homer).
hombre(bart).
mujer(mona).
mujer(jackie).
mujer(marge).
mujer(patty).
mujer(selma).
```

Figura 1: Datalog y los Simpson

```
mujer(lisa).
mujer(maggie).
mujer(ling).
```

```
progenitor(abraham, herbert).
progenitor(abraham, homer).
progenitor(mona, homer).
progenitor(clancy, marge).
progenitor(jackie, marge).
progenitor(clancy, patty).
progenitor(jackie, patty).
progenitor(clancy, selma).
progenitor(jackie, selma).
progenitor(homer, bart).
progenitor(marge, bart).
progenitor(homer, lisa).
progenitor(marge, lisa).
progenitor(homer, maggie).
progenitor(marge, maggie).
progenitor(selma, ling).
```

- Podemos expresar las siguientes reglas no recursivas:
  - Reglas sobre la paternidad:
    - Semántica de `padre(X,Y)`: X es padre de Y. Semántica de `madre(X,Y)`: X es madre de Y.
    - En Datalog se expresa:

      ```
      padre(X,Y) :- progenitor(X,Y), hombre(X).
      madre(X,Y) :- progenitor(X,Y), mujer(X).
      ```

  - Relaciones de hermandad:
    - Semántica de `hermanos(X,Y)`: X es hermano o hermana de Y.
    - En Datalog se expresa:

      ```
      hermanos(X,Y) :- progenitor(Z,X), progenitor(Z,Y), X \= Y.
      ```

  - De modo similar podemos indicarle al sistema en qué consiste ser "tía" y "abuela":

    ```
    tia(X,Y) :- mujer(X), hermanos(X,Z), progenitor(Z,Y).
    abuela(X,Y) :- madre(X,Z), progenitor(Z,Y).
    ```

- Regla recursiva:
  - Una regla remite a otra del mismo tipo.
  - Ejemplo:

    ```
    antecesor(X,Y) :- progenitor(X,Y).
    antecesor(X,Y) :- progenitor(X,Z), antecesor(Z,Y).
    ```

    produce el siguiente resultado:

    ```
    DES> antecesor(X,Y)
    {
        antecesor(abraham,bart),
    ```

```
    antecesor(abraham,herbert),
    antecesor(abraham,homer),
    antecesor(abraham,lisa),
    antecesor(abraham,maggie),
    antecesor(clancy,bart),
    antecesor(clancy,ling),
    ...
}
```

- Otro ejemplo:
  - Ser de la misma generación `mg(X,Y)`.
  - Se codifica:

```
mg(X,Y) :- hermanos(X,Y).
mg(X,Y) :- progenitor(Z,X), progenitor(V,Y), mg(Z,V).
```

produce el siguiente resultado:

```
DES> mg(X,Y)
{
    mg(bart,ling),
    mg(bart,lisa),
    mg(bart,maggie),
    mg(herbert,homer),
    mg(homer,herbert),
    mg(ling,bart),
    ...
}
```

3.7.2. Caminos en un grafo

- Representar un grafo es muy sencillo con Datalog: simplemente hay que especificar las aristas.
- Ejemplo:

```
arista(a,b).
arista(b,d).
arista(d,c).
arista(d,e).
arista(e,f).
arista(f,c).
arista(f,g).
arista(g,d).
```

- Podemos introducir la idea de "alcanzabilidad " a través del predicado `camino(X,Y)` (existe camino entre `X` e `Y`).
- Codificado:

```
camino(X,Y) :- arista(X,Y).
camino(X,Y) :- arista(X,Z), camino(Z,Y).
```

puede producir el siguiente resultado:

```
DES> camino(e,Z).
{
    camino(e,c),
    camino(e,d),
    camino(e,e),
    camino(e,f),
    camino(e,g)
}
```

- Podemos especificar la idea de ciclo:

```
ciclo(Z) :- camino(Z,Z).
```

puede producir el siguiente resultado:

```
DES> ciclo(Z)
{
    ciclo(d),
    ciclo(e),
    ciclo(f),
    ciclo(g)
}
```

4. Datalog avanzado

4.1. Reuniones

- Datalog posee mecanismos para combinar tablas de forma similar al del esquema relacional.
- Permite los siguientes tipos de reuniones:
  - *Natural join.*
  - *Left outer join.*
  - *Right outer join.*
  - *Full outer join.*

4.1.1. Reuniones internas

- No existe ninguna función especial para representar reuniones internas.
- Pueden expresarse de manera sencilla mediante reglas.
- Ejemplo:

```
% cliente(Id, Nombre, Apellidos)
cliente(1, 'Javier', 'Herranz').
cliente(2, 'Ana', 'Martin').
cliente(3, 'Gerardo', 'de la Iglesia').
cliente(4, 'Manuel', 'Moreno').
cliente(5, 'Lucia', 'Rodriguez').
cliente(6, 'Raquel', 'Velasco').
cliente(7, 'Ignacio', 'Martin').
```

```
% cuenta(Id, Tipo, Saldo, IdTitular)
cuenta(1, nomina, 12000, 1).
cuenta(2, ahorro, 15000, 1).
cuenta(3, nomina, 21500, 2).
cuenta(4, ahorro, 1200, 3).
cuenta(5, nomina, 5000, 4).
cuenta(6, ahorro, 5000, 5).
cuenta(7, null, 100, 6).
cuenta(8, nomina, 1500, 6).
```

La siguiente regla implementa una reunión interna de ambas "tablas":

```
% cuenta(Id, Tipo, Saldo, IdTitular)
clientes_cuentas(N, A, T, S) :-
    cliente(Id_Cliente, N, A), cuenta(_, T, S, Id_Cliente).
```

Un posible resultado:

```
% cuenta(Id, Tipo, Saldo, IdTitular)
DES> clientes_cuentas(N, A, T, S)
{
    clientes_cuentas('Ana','Martin',nomina,21500),
    clientes_cuentas('Gerardo','de la Iglesia',ahorro,1200),
    clientes_cuentas('Javier','Herranz',ahorro,15000),
    clientes_cuentas('Javier','Herranz',nomina,12000),
    clientes_cuentas('Lucia','Rodriguez',ahorro,5000),
    clientes_cuentas('Manuel','Moreno',nomina,5000),
    clientes_cuentas('Raquel','Velasco',nomina,1500),
    clientes_cuentas('Raquel','Velasco',null,100)
}
```

- Las funciones `is_null` e `is_not_null` permiten comprobar si un atributo es nulo (no utilizar en comparaciones):

```
DES> clientes_cuentas(N, A, T, S), is_not_null(T)
Info: Processing:
answer(N,A,T,S) :-
clientes_cuentas(N,A,T,S),
is_not_null(T).
{
    answer('Ana','Martin',nomina,21500),
    answer('Gerardo','de la Iglesia',ahorro,1200),
    answer('Javier','Herranz',ahorro,15000),
    answer('Javier','Herranz',nomina,12000),
    answer('Lucia','Rodriguez',ahorro,5000),
    answer('Manuel','Moreno',nomina,5000),
    answer('Raquel','Velasco',nomina,1500)
}
```

4.1.2. Reuniones externas

- Reunión externa por la izquierda (*left outer join*): `lj(RelIzqda, RelDcha, CondicionJoin)`
- Reunión externa por la derecha (*right outer join*): `rj(RelIzqda, RelDcha, CondicionJoin)`

- Reunión externa completa (*full outer outer join*): `fj(RelIzqda, RelDcha, CondicionJoin)`
- Ejemplo:

```
clientes_cuentas_left(N, A, T, S) :-
    lj(cliente(Id, N, A), cuenta(_, T, S, IdC), Id = IdC).
```

```
DES> clientes_cuentas_left(N, A, T, S)
{
    clientes_cuentas_left('Ana','Martin',nomina,21500),
    clientes_cuentas_left('Gerardo','de_la_Iglesia',ahorro,1200),
    clientes_cuentas_left('Ignacio','Martin',null,null),
    clientes_cuentas_left('Javier','Herranz',ahorro,15000),
    clientes_cuentas_left('Javier','Herranz',nomina,12000),
    clientes_cuentas_left('Lucia','Rodriguez',ahorro,5000),
    clientes_cuentas_left('Manuel','Moreno',nomina,5000),
    clientes_cuentas_left('Raquel','Velasco',nomina,1500),
    clientes_cuentas_left('Raquel','Velasco',null,100)
}
```

4.2. Agrupaciones

- Instrucción básica: `group_by(Rel, VarsGroup, CondsGroup)`
- Realiza un agrupamiento de los resultados de la relación `Rel`, reuniendo aquellos que coincidan en las variables indicadas en `VarsGroup`. En `CondsGroup` podemos utilizar funciones de agregación y ligar sus valores a variables externas.
- `VarsGroup` es una lista `[Var-1, ..., Var-n]`
- `CondsGroup` se compone de una o varias condiciones `Var = Exp`, donde `Exp` puede contener funciones de agregación.
- Ejemplo:

```
clientes_cuentas_left(Id, N, A, T, S) :-
    lj(cliente(Id, N, A), cuenta(_, T, S, _IdC), Id = _IdC).

clientes_num_cuentas(N, A, NumCuentas) :-
    group_by(
        clientes_cuentas_left(Id, N, A, T, S),
        [Id, N, A],
        NumCuentas = count(S)
    ).
```

```
DES> clientes_num_cuentas(N, A, NumCuentas)
{
    clientes_num_cuentas('Ana','Martin',1),
    clientes_num_cuentas('Gerardo','de_la_Iglesia',1),
    clientes_num_cuentas('Ignacio','Martin',0),
    clientes_num_cuentas('Javier','Herranz',2),
    clientes_num_cuentas('Lucia','Rodriguez',1),
    clientes_num_cuentas('Manuel','Moreno',1),
    clientes_num_cuentas('Raquel','Velasco',1)
}
```

4.3. Funciones de agregación

- `count(X)`: Número de resultados en las que `X` es distinto de null.
- `count`: Número de resultados.
- `sum(X) / times(X)`: Suma/multiplicación de los valores de `X`.
- `avg(X)`: Media aritmética de los valores de `X`.
- `min(X) / max(X)`: Mínimo/máximo de los valores de `X`.
- Ejemplo:

```
clientes_num_cuentas_y_saldo_total(N, A, NumCuentas, SaldoTotal) :-
    group_by(
        clientes_cuentas_left(Id, N, A, T, S),
        [Id, N, A],
        (NumCuentas = count(T), SaldoTotal = sum(S))
    ).
```

```
DES> clientes_num_cuentas_y_saldo_total(N, A, NumCuentas, SaldoTotal)
{
    clientes_num_cuentas_y_saldo_total('Ana','Martin',1,21500),
    clientes_num_cuentas_y_saldo_total('Gerardo','de_la_Iglesia',1,1200),
    clientes_num_cuentas_y_saldo_total('Ignacio','Martin',0,null),
    clientes_num_cuentas_y_saldo_total('Javier','Herranz',2,27000),
    clientes_num_cuentas_y_saldo_total('Lucia','Rodriguez',1,5000),
    clientes_num_cuentas_y_saldo_total('Manuel','Moreno',1,5000),
    clientes_num_cuentas_y_saldo_total('Raquel','Velasco',1,1600)
}
```

- Se pueden añadir condiciones sobre la agregación:

```
clientes_num_cuentas_y_saldo_total(N, A, NumCuentas, SaldoTotal) :-
    group_by(
        clientes_cuentas_left(Id, N, A, T, S),
        [Id, N, A],
        (NumCuentas = count(T), SaldoTotal = sum(S))
    ), SaldoTotal > 10000.
```

```
DES> clientes_num_cuentas_y_saldo_total(N, A, NumCuentas, SaldoTotal)
{
    clientes_num_cuentas_y_saldo_total('Ana','Martin',1,21500),
    clientes_num_cuentas_y_saldo_total('Javier','Herranz',2,27000)
}
```

4.4. Aritmética

- Pueden realizarse operaciones aritméticas mediante el operador `is`.
- Sintaxis: `X is Expresión`
- Ejemplo:

```
% articulo(Nombre, Cantidad, PrecioUnidad)
articulo('Aceite', 2, 6.95).
articulo('Azúcar', 1, 1.20).
articulo('Leche', 3, 0.90).

total_articulo(Nombre, Total) :-
        articulo(Nombre, Cantidad, PrecioUnidad),
        Total is Cantidad * PrecioUnidad.
total_compra(Suma) :-
        group_by(total_articulo(_, Total), [], Suma = sum(Total)).
```

Algunos resultados posibles:

```
DES> total_articulo(N, T)
{
    total_articulo('Aceite',13.9),
    total_articulo('Azúcar',1.2),
    total_articulo('Leche',2.7)
}
```

```
DES> total_compra(T).
{
    total_compra(17.8)
}
```

- Los argumentos de las relaciones sólo pueden ser constantes o variables; nunca expresiones. El siguiente código es incorrecto:

```
total_articulo(Nombre, Cantidad * PrecioUnidad) :-
    articulo(Nombre, Cantidad, PrecioUnidad).
```

- Las variables que aparezcan en las expresiones aritméticas han de estar ligadas a algún valor. El siguiente código es incorrecto:

```
p(X, Z) :- Z is X+Y.
```

La `Y` no está ligada.

- El siguiente código mira la distancia entre dos nodos de un grafo:
  - No puede tener ciclos.
  - Es preciso indicar la lista de los nodos.

```
camino(X,X,0).
camino(X,Y,Dist) :-
    camino(X,Z,Dist1),
    arista(Z,Y), Dist is Dist1 + 1.
```

Se puede calcular la distancia de un nodo a los demás:

```
DES> camino(a,Y,D)
{
    camino(a,a,0),
    camino(a,b,1),
```

```
    camino(a,c,3),
    camino(a,c,5),
    camino(a,d,2),
    camino(a,e,3),
    camino(a,f,4),
    camino(a,g,5)
}
```

Cuestión: ¿qué pasaría si tuviera ciclos?.

- Otro ejemplo. El factorial:

```
%factorial(N, Fact)
factorial(0,1).
factorial(N,F) :-
    N > 0, NAux is N-1, factorial(NAux,FAux), F is N * FAux.
```

4.5. Negación

- En Datalog se considera que una relación no se cumple si no se puede deducir de la ya existente en la base de datos.
- La negación se identifica con la ausencia de información.
- El operador `not` permite comprobar si una determinada relación no es cierta.
- Ejemplo:

```
clientes_sin_cuenta_ahorro(N, A) :-
clientes_cuentas(N, A, _, _),
    not clientes_cuentas(N, A, ahorro, _).
```

```
DES> clientes_sin_cuenta_ahorro(N,A).
{
    clientes_sin_cuenta_ahorro('Ana','Martin'),
    clientes_sin_cuenta_ahorro('Manuel','Moreno'),
    clientes_sin_cuenta_ahorro('Raquel','Velasco')
}
```

4.5.1. Negación estratificada

- La negación puede ser problemática cuando aparece en relaciones definidas recursivamente.
- Las mayoría de implementaciones Datalog utiliza una versión restringida de la negación, conocida como negación estratificada.
- Una definición de relación, que contenga una negación, se considera segura si no aparece en un camino de cómputo recursivo.
- Comprobación de la corrección:
  - A partir del conjunciones de las reglas construimos un grafo de dependencias.
  - Si un predicado q forma parte de la definición de otro p trazaremos una arista de q a p.
  - Si el predicado q aparece sin negar le asignaremos a la arista de q a p una etiqueta $+$.

- Si el predicado q aparece negado le asignaremos a la arista de q a p una etiqueta $-$.
- El código se considera seguro si no existe ningún ciclo con una etiqueta $-$.

- El siguiente código es seguro:

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

- Mediante el comando `/pdg` de DES se puede obtener una representación del grafo de dependencias.

5. Más allá de Datalog: Prolog

5.1. ¿Qué es la programación declarativa?

- La idea clave consiste en describir relaciones entre los datos sin concretar los algoritmos.
- Se describe qué debe ser computado y no cómo debe ser computado.
- En consecuencia no hay estructuras de control. Tampoco hay variables mutables, es decir, no hay asignación:
  - ¿Qué es una variable mutable?.
  - Es una variable "normal": a lo largo de la ejecución de un programa puede tener varios valores.
  - En Prolog cuando una variable toma un valor ya no puede ser modificado.
- Contrasta con la programación imperativa (o procedimental), en la que hay un flujo de programa determinado por las acciones del mismo.
- Ventajas de la programación declarativa:
  - Permite al programador concentrarse en la formulación del problema y lo libera del control del algoritmo.
  - Los programas son más fáciles de manejar, transformar y verificar.
- Hay multitud de lenguajes declarativos o con inspiración declarativa (como SQL).
- El paradigma de programación declarativa se compone, a su vez, de dos paradigmas:
  - Programación lógica.
  - Programación funcional.
- Existen dos grandes familias de lenguajes declarativos:
  - Lenguajes lógicos, cuyo representante más conocido es Prolog.
  - Lenguajes funcionales, cuyo representante más puro es Haskell.
- En la práctica:
  - La eficiencia también importa: la ausencia de control es relativa y el programador no queda completamente liberado de la algoritmia.
  - Pero el programador trabaja a un nivel de abstracción superior y queda liberado de fuentes comunes de error: uso de punteros, pasos por referencia/valor, condiciones de parada de bucles, etc.

- Estos estilos de programación enriquecen considerablemente las habilidades del programador, aunque posteriormente utilice algún lenguaje imperativo.
- El diseño declarativo es normalmente más conciso, elegante, claro y fácil de mantener.

- Algunas características de Prolog:
  - Sintaxis: subconjunto de la lógica de primer orden. Aunque la sintaxis de Datalog está basada en la de Prolog, éste último es más expresivo (permite utilizar listas).
  - Semántica declarativa: basada en la propia lógica.
  - Semántica operacional: Unificación + resolución SLD (mecanismo de funcionamiento).
  - Enriquecido con multitud de librerías.

5.2. Prolog: listas

- Existe muchos interpretes/compiladores de Prolog. Nosotros utilizaremos SWI-prolog.
- Descargable en `http://www.swi-prolog.org/`
- La complejidad de Prolog desborda los límites de esta asignatura. Nos limitaremos a estudiar las listas.

Listas en Prolog:

- Una lista es una secuencia ordenada de elementos que puede tener cualquier longitud.
- Los elementos de una lista pueden ser cualquier término (constantes, variables, estructuras) u otras listas.
- Una lista puede definirse recursivamente como:
  - Una lista vacía: `[]` o
  - Una lista con dos componentes:
    - Cabeza: primer elemento de la lista.
    - Cola: resto de la lista.
  - Una lista con este formato se representa mediante `[X|Y]`:
    - Donde `X` es un elemento e `Y` es una lista.
    - Una lista con un solo elemento también se puede representar como `[X|Y]` (si `Y=[]`) o `[X]`.
  - Ejemplos:

Lista	Cabeza	Cola
`[a,b,c]`	`a`	`[b,c]`
`[a]`	a	`[]`
`[]`	no tiene	no tiene
`[[a,b],c]`	`[a,b]`	`[c]`
`[a,[b,c]]`	`a`	`[[b,c]]`
`[a,[b,c],d]`	`a`	`[[b,c],d]`
`[a+b,c+d]`	`a+b`	`[c+d]`

Nota: `+` es el operador concatenación.

- Los siguientes ejemplos muestran funciones lógicas para saber si un elemento dado está o no en una lista:

```
miembro(E,L) :- L=[X|Y], X=E.
miembro(E,L) :- L=[X|Y], miembro(E,Y).
```

```
miembro(E,[X|Y]) :- X=E.
miembro(E,[X|Y]) :- miembro(E,Y).
```

```
miembro(X,[X|Y]).
miembro(E,[X|Y]) :- miembro(E,Y).
```

```
miembro(X,[X|_]).
miembro(X,[_|Y]) :- miembro(X,Y).
```

- Otras operaciones con listas:

```
/* nel(Lista,N) <- el numero de elementos de la lista Lista es N */
nel([],0).
nel([X|Y],N) :- nel(Y,M),
N is M+1.
```

```
/* es_lista(Lista) <- Lista es una lista */
es_lista([]).
es_lista([_|_]).
```

```
/* concatena(A,B,C) <- concatenación de las listas A y B
dando lugar a la lista C */
concatena(A,B,C) :- A=[], C=B.
concatena(A,B,C) :- A=[X|D], concatena(D,B,E), C=[X|E].
```

o bien:

```
/* concatena(L1,L2,L3) <- concatenación de las listas L1 y L2
dando lugar a la lista L3 */
concatena([],L,L).
concatena([X|L1],L2,[X|L3]) :- concatena(L1,L2,L3).
```

```
/* ultimo(Elem,Lista) <- Elem es el ultimo elemento de Lista */
ultimo(X,[X]).
ultimo(X,[_|Y]) :- ultimo(X,Y).
```

```
/* inversa(Lista,Inver) <- Inver es la inversa de la lista Lista */
inversa([],[]).
inversa([X|Y],L) :- inversa(Y,Z),
concatena(Z,[X],L).
```

```
/* borrar(Elem,L1,L2) <- se borra el elemento Elem de la lista L1
obteniéndose la lista L2 */
borrar(X,[X|Y],Y).
borrar(X,[Z|L],[Z|M]) :- borrar(X,L,M).
```

```
/* subconjunto(L1,L2) <- la lista L1 es un subconjunto de lista L2 */
subconjunto([X|Y],Z) :- miembro(X,Z),
subconjunto(Y,Z).
subconjunto([],Y).
```

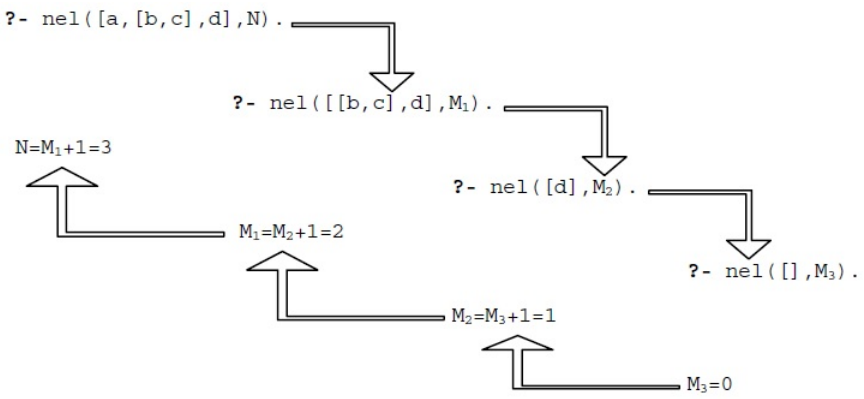

Figura 2: Secuencia de llamadas para el cómputo de la función `nel`

```
/* insertar(Elem,L1,L2) <- se inserta el elemento Elem en la lista L1
obteniéndose la lista L2 */
insertar(E,L,[E|L]).
insertar(E,[X|Y],[X|Z]) :- insertar(E,Y,Z).
```

```
/* permutacion(L1,L2) <- la lista L2 es una permutación de lista L1 */
permutacion([],[]).
permutación([X|Y],Z) :- permutacion(Y,L),
insertar(X,L,Z).
```

Recursión:

- Veamos a través de un ejemplo cómo funcional el mecanismo de cómputo de Prolog.
- Consideremos el predicado `nel` (calcula el número de elementos de una lista).
- En la recursión encontramos dos partes:
  1. Descendemos y construimos el árbol hasta encontrar el valor que unifica con la condición de parada.
  2. Ascendemos por el árbol asignando valores a las variables que teníamos pendientes en las sucesivas llamadas.

Quicksort con Prolog:

```
partition(_, [], [], []).
partition(P, [X|Xs], [X|Ls], Gs) :- X =< P, partition(P, Xs, Ls, Gs).
partition(P, [X|Xs], Ls, [X|Gs]) :- X > P, partition(P, Xs, Ls, Gs).

qsort([], []).
qsort([X|Xs], Zs) :-
    partition(X, Xs, Ls, Gs),
    qsort(Ls, LsSort),
    qsort(Gs, GsSort),
    append(LsSort, [X|GsSort], Zs).
```

```
?- qsort([6,1,4,9,2], Xs).
Xs = [1, 2, 4, 6, 9].
```