

BDD - Cours 9

Introduction à Datalog

Celine Kuttler

9 novembre 2016

Datalog est une machine pour construire des nouveaux faits.

- ▶ 1977s : invention de Prolog, dans le contexte des bases de données. Idée : ajouter du calcul récursif aux requêtes relationnelles.
- ▶ 1980s : programmation logique populaire pour l'intelligence artificielle. présence industrielle forte, mais pas encore de killer app pour les requêtes récursives.
- ▶ 1990s : niches...
- ▶ depuis environ 2007 : renaissance de Datalog pour le web.
- ▶ nous travaillerons avec DES (des.sourceforge.net) de Fernando Saenz-Perez
- ▶ puis en fin de semestre nous travaillerons avec un outil en Prolog pour la normalisation des BDDs (support pédagogique du livre d'Elmasri et Navathe).

Hypothèse du monde clos

Un fait est considéré comme faux s'il n'est ni inclus dans la base de données des faits, ni démontrable en temps fini. Il n'y a pas de monde extérieur qui pourrait contenir des éléments inconnus au programme.

Cette hypothèse motive des contraintes syntaxiques (vues en fin de ce cours, si le temps le permet).

Contenu

Introduction a Datalog

- Faits et règles

- Différentes catégories de règles, et leur significations

- Requêtes récursives : chemins dans un graphe

Requêtes relationnelles en Datalog

- Opérations de l'algèbre relationnelle

- Aggregation etc en DES

Chemin le plus court : Datalog vs SQL

Semantique

- Sémantique des points fixes

- Exemple : graphe

- Mémoisation

- Sûreté

SAT

- Premier codage de formules booléennes en Datalog

- Qui est l'assassin ?

Différences entre Datalog et Prolog

Exemple 1 : rien que des faits

```
fete .  
femme(mia ).  
femme(jody ).  
femme(yolanda ).  
happy(yolanda ).  
joueAirGuitar(jody ).
```

Requêtes :

- ▶ tuple trouvé ou non.
- ▶ requête avec variables : présence de femmes ? noms des femmes ?
- ▶ notion de **but** : toute chose qui peut être prouvée



Résumé de la syntaxe (1)

- ▶ constantes
 - ▶ nombres (entiers et réels)
 - ▶ séquences de caractères alphanumériques, _ inclus, qui commencent avec une **minuscule**
- ▶ prédicats $p(a_1, a_2, \dots, a_n)$. Le prédicat p prend n arguments, qui sont des variables ou constantes. Le nom du prédicat doit commencer avec une minuscule.
- ▶ variables
 - ▶ X, Y (séquences qui commencent avec une **majuscule**),
 - ▶ _ (variable anonyme, tiret bas)

Comment construire des nouveaux faits ?

Modus Ponens

Du fait A , en combinaison avec la règle $A \Rightarrow B$, on déduit B .

Une base de connaissances contient :

- ▶ des faits et
- ▶ des règles.

Exemple 1 : la fête

- ▶ Fait : C'est la fête.
 F
- ▶ Règle : Quand c'est la fête, il y a de la musique.
 $F \Rightarrow M$
- ▶ Conclusion : Il y a de la musique !
 M

Une première règle en Datalog

```
fete .  
musique :- fete .
```

Comment lire une règle Datalog ?

En notation de logique propositionnelle, la règle Datalog

$$B : -A.$$

se lit comme **implication dans l'autre direction**,

$$A \Rightarrow B$$

Exemple 2 : règles et leurs interprétations

```
ecoute2LaMusique(yolanda) :- happy(yolanda).  
joueAirGuitar(mia) :- ecoute2LaMusique(mia).  
joueAirGuitar(yolanda) :-  
                        ecoute2LaMusique(yolanda).
```

Interprétation

- ▶ Quand Yoland est contente, elle écoute de la musique.
- ▶ Quand Mia écoute de la musique, elle joue de l'AirGuitar.
- ▶ Quand Yolanda écoute de la musique, elle joue de l'AirGuitar.

Exemple 2 : Requêtes, concernant des faits déduits

```
fete.femme(mia). ecoute2LaMusique(mia).  
femme(yolanda). happy(yolanda).  
  
ecoute2LaMusique(yolanda) :- happy(yolanda).  
joueAirGuitar(mia) :- ecoute2LaMusique(mia).  
joueAirGuitar(yolanda) :-  
                        ecoute2LaMusique(yolanda).
```

Requêtes

- ▶ Est-ce que Mia joue de l'AirGuitar ?
- ▶ Est-ce que Yolanda joue de l'AirGuitar ?
- ▶ Qui joue de l'AirGuitar ?

Règles avec plusieurs conditions

En notation de logique propositionnelle, la règle Datalog

$$Z : \neg A_1, A_2, \dots, A_n.$$

se lit comme **implication dans la direction inverse**,

$$A_1 \text{ and } A_2 \text{ and } \dots \text{ and } A_n \implies Z$$

- ▶ dans la tête d'une règle, toujours un seul fait
- ▶ le corps de la règle est une **conjonction** de n faits
- ▶ si **toutes** les conditions sont satisfaites, on peut déduire le nouveau fait en tête, Z
- ▶ on appelle les A_i des **buts**. Chaque chose qui est à prouver en Datalog est un **but**.

Exemple 3 : Conjonction logique (et)

```
happy(vincent).  
ecoute2LaMusique(paul).  
  
joueAirGuitar(vincent) :-  
    ecoute2LaMusique(vincent), happy(vincent).
```

Syntaxe

La **virgule** exprime la conjonction en Datalog.

Interprétation

Vincent joue de l'AirGuitar, quand il est content **et** qu'il écoute de la musique.

Requêtes

- Est-ce que Vincent joue de l'AirGuitar ?

Exo : trouver une musicienne

Comment demander s'il existe une **femme** qui joue de l'**AirGuitar** ?



[autoview, mot-clé réservé (answer)]

Exemple 4 : Disjonction logique

Paul joue de l'AirGuitar, quand il est content, **ou** qu'il écoute de la musique.

Syntaxe

Pour exprimer un OU logique en Datalog, on écrit **deux règles**.

```
ecoute2LaMusique( paul ).  
  
joueAirGuitar( paul ) :- happy( paul ).  
  
joueAirGuitar( paul ) :- ecoute2LaMusique( paul ).
```

Requête

- ▶ Est-ce que Paul joue de l'AirGuitar ?

Résumé de la syntaxe (2)

Comment exprimer les opérateurs logiques en Datalog :

- ▶ conjonction : virgule
- ▶ disjonction : écrire deux règles
- ▶ implication $B : - A_1, \dots, A_n .$
 - ▶ le corps est une conjonction de n faits
 - ▶ dans la tête, un seul fait
 - ▶ si toutes ses conditions sont satisfaites, on peut déduire le nouveau fait B

Catégories de règles

Nous pouvons distinguer 3 catégories de règles, de plus en plus expressives :

- ▶ règles avec faits simples, c.a.d. constantes (quand c'est la fête, il y a de la musique)
- ▶ règles avec prédicats + constantes
- ▶ règles avec prédicats + variables

Règles avec variables

Le millionnaire

- ▶ Tous les millionnaires ont un coffre-fort.
 $\forall x : \textit{millionnaire}(x) \Rightarrow \textit{avoir_coffre_fort}(x)$
- ▶ en Datalog :
 $\textit{avoir_coffre_fort}(X) : \neg \textit{millionnaire}(X).$
- ▶ Fait : Balthazar Picsou est un millionnaire.
 $\textit{millionnaire}(bp)$
- ▶ Lorsqu'on pose la requête
 $\textit{avoir_coffre_fort}(bp)$, Datalog conclut du programme, que
Balthazar Picsou a un coffre-fort, et répond oui.

Base : bars

Schémas

sert(bar, biere ,prix)

frequente(personne, bar)

aime(personne,biere)

```
sert ( mcevans , lachouffe , 3.5 ).
```

```
sert ( mcevans , leffe , 2.5 ).
```

```
sert ( omnia , lachouffe , 4.5 ) .
```

```
sert ( taverneflamande , lachti , 1.9 ).
```

```
frequente ( timoleon , mcevans ).
```

```
aime ( timoleon , lachouffe ).
```

Exemple de règle avec variables

```
content(X) :- aime(X, Beer) , frequente(X, Bar) ,  
               sert ( Bar , Beer , _ ).
```

En français

Si quelqu'un aime une certaine bière, et fréquente un bar qui vend cette bière, alors il est content.

En logique

$\forall X, Bar, Beer, Prix :$

$aime(X, Beer)$ and $frequente(X, Bar)$ and $sert(Bar, Beer, Prix)$
 $\rightarrow content(X)$

Questions auxquelles Datalog sait répondre

Exemples

- ▶ Timoléon est-il content ?
- ▶ Qui est content ?

```
sert(mcevans, lachouffe, 3.5).  
sert(mcevans, leffe, 2.5).  
sert(omnia, lachouffe, 4.5)  
sert(taverneflamande, lachti, 1.9)
```

```
frequente(timoleon, mcevans).
```

```
aime(timoleon, lachouffe).
```

```
content(X) :- aime(X, Beer) , frequente(X, Bar) ,  
               sert(Bar, Beer, _).
```

Exemple de règle avec comparaison

Syntaxe

comparaisons $A \text{ op } B$, ou A et B sont des constantes ou variables, et op un opérateur de comparaison

$$\text{bonmarche}(B) \quad :- \quad \text{sert}(B, _ , P), \quad P \leq 2.0 \quad .$$

Interprétation de la règle

Un bar est bon marché, s'il sert une bière à moins de deux euros.

Astuce

L'occurrence de la variable P dans le prédicat *sert* permet une liaison de cette variable. Uniquement après cette liaison, la variable peut être comparée.

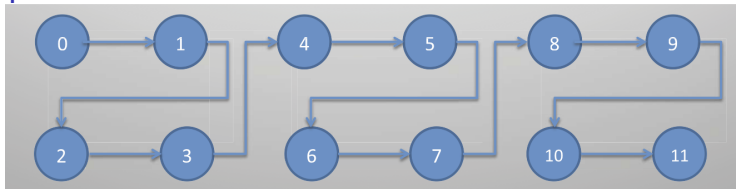
Signification des règles Datalog

Datalog est une machine pour construire des nouveaux faits.

Première approximation de la sémantique pour règles avec variables, non-récurrentes.

- ▶ prends les valeurs de variables qui rendent le corps de la règle vrai (il faut rendre vrai chacun des sous-buts)
- ▶ considère les valeurs que peuvent prendre les variables de la tête
- ▶ ajoute le tuple créé dans l'étape précédente, à la relation en tête de règle

Graphe



Le prédicat $e/2$ exprime un lien direct entre deux sommets (e pour anglais : edge) :

```
e(0,1). e(1,2). e(2,3).  
e(3,4). e(4,5). e(5,6).  
e(6,7). e(7,8). e(8,9).  
e(9,10). e(10,11).
```

But : tester l'existence d'un **chemin** entre deux noeuds

- ▶ Requête pour tester l'existence des chemin de **longueur fixe**, p.ex. 2 et 3, entre deux noeuds.
- ▶ Peut-on poser les requêtes correspondantes en SQL ?

Chemins dans un graphe

- ▶ Comment tester si deux sommets (edge) sont connectés, quelque soit la longueur du chemin ?
- ▶ Définir un prédicat $p/2$, qui exprime un **chemin** (path) entre deux sommets passant par un nombre arbitraire de liens.

$$p(X,Y) :- e(X,Y).$$
$$p(X,Z) :- e(X,Y), p(Y,Z).$$

Contenu

Introduction a Datalog

- Faits et règles

- Différentes catégories de règles, et leur significations

- Requêtes récursives : chemins dans un graphe

Requêtes relationnelles en Datalog

- Opérations de l'algèbre relationnelle

- Aggregation etc en DES

Chemin le plus court : Datalog vs SQL

Semantique

- Sémantique des points fixes

- Exemple : graphe

- Mémoisation

- Sûreté

SAT

- Premier codage de formules booléennes en Datalog

- Qui est l'assassin ?

Différences entre Datalog et Prolog

Datalog et l'algèbre relationnelle

- ▶ Nous montrons comment exprimer, en Datalog, des requêtes sur une base de données relationnelle, du type SELECT-FROM-WHERE, ou des requêtes en algèbre relationnelle
- ▶ les mêmes opérations qu'on peut définir en algèbre relationnelle, ou SQL de base, peuvent être définies en Datalog
- ▶ Nous supposons un nom de relation (symbole de prédicat) r en Datalog, pour chaque tableau R d'une base de données relationnelle

Exemple : la boutique en DES

- ▶ `articles(aid : int, anom : string, acoul : string)`
- ▶ `fournisseurs(fid : int, fnom : string, fad : string)`
- ▶ `catalogue(fid : int, aid : int, prix : float)`
- ▶ **déclaration de types** (create table) + ajout à la EDB (inserts) :

```
:-type( articles( aid:int ,anom:string ,acoul:string ) ).  
:-type( fournisseurs( fid:int ,fnom:string ,fad:string ) ).  
:-type( catalogue( fid:int ,aid:int ,prix:float ) ).
```

```
articles(1, 'Left_Handed_Toaster_Cover', 'rouge' ).  
articles(2, 'Smoke_Shifter_End', 'noir' ).
```

```
...
```

```
fournisseurs(1, 'kiventout', '59_rue_du_Chti ,  
_F-75001_Paris' ).
```

```
fournisseurs(2, 'Big_Red_Tool_and_Die', '4_My_Way,  
_Bermuda_Shorts ,_OR_90305,_USA' ).
```

```
...
```

```
catalogue(1,1,36.10).
```

```
catalogue(1,2,42.30).
```

Projection $\pi_{Acoul}(Articles)$

Afficher toutes les couleurs d'articles

```
couleur(C) :- articles(_,_,C).
```

Requête et résultat avec l'outil DES :

```
DES> couleur(X)
{
  couleur(argente),   couleur(cyan),
  couleur(magenta),   couleur(noir),
  couleur(opaque),    couleur(rouge),
  couleur(superjaune), couleur(vert)
}
Info: 8 tuples computed.
```

en SQL :

```
CREATE VIEW couleur AS
SELECT acoul FROM articles
```

Selection σ

Afficher tous les noms d'articles verts. Afficher tous les noms d'articles rouges.

```
art_vert (Anom)
  :- articles (_, Anom, 'vert').
art_rouge (Anom)
  :- articles (_, Anom, Acoul), Acoul='rouge'.

art_vert (X).
art_rouge (X).
```

```
CREATE VIEW art_rouge(anom) as
SELECT anom FROM articles WHERE acoul='rouge';
```

```
CREATE VIEW art_vert(anom) as
SELECT anom FROM articles WHERE acoul='vert';
```

```
select anom from art_rouge;
select anom from art_vert;
```

Intersection

articles existant en rouge **et** en vert

```
rouge_et_vert(X) :- art_rouge(X), art_vert(X).
```

en SQL :

```
CREATE VIEW rouge_et_vert AS  
(  
  art_rouge INTERSECT art_vert  
)  
  
select * from rouge_et_vert;
```

Différence

les articles existant en rouge, **mais pas** en vert

```
rouge_pas_vert(X) :-  
    art_rouge(X), not(art_vert(X)).
```

en SQL :

```
CREATE VIEW rouge_pas_vert AS  
(art_rouge MINUS art_vert)
```

Union

les articles rouges **ou** verts

```
rouge_ou_vert(X) :- art_rouge(X).  
rouge_ou_vert(X) :- art_vert(X).
```

en SQL :

```
CREATE VIEW rouge_ou_vert AS  
    (art_rouge UNION art_vert)
```


Jointure

Qui vend quel article a quel prix ?

```
quivendquoi (Anom, Acoul, Fid, Prix)
:- articles (Aid, Anom, Acoul),
   catalogue (Fid, Aid, Prix).
```

Variable Aid partagée entre les deux relations. Ou bien, test d'égalité entre deux variables :

```
quivendquoi (Anom, Acoul, Fid, Prix)
:- articles (A_aid, Anom, Acoul),
   catalogue (Fid, C_aid, Prix), A_aid=C_aid.
```

en SQL :

```
CREATE VIEW quivendquoi AS
SELECT anom, acoul, fid, prix
FROM articles c join catalogue a using (aid);

select * from quivendquoi;
```

Produit cartésien

toutes les combinaisons de noms de fournisseurs et noms d'articles.

```
cart (Anom, Fnom) :-  
    articles (_, Anom, _), fournisseurs (_, Fnom, _).
```

Requête et résultat :

```
DES> cart(X,Y)  
{  cart('7_Segment_Display', 'Alien_Aircraft_Inc.'),  
  cart('7_Segment_Display', 'Autolux'),  
  ...  
  cart('Smoke_Shifter_End', 'Vendrien'),  
  cart('Smoke_Shifter_End', 'kiventout')}  
Info: 54 tuples computed.
```

```
CREATE VIEW cart AS  
SELECT anom, fnom FROM articles , fournisseurs
```

Quantification existentielle

Articles offerts par au moins un fournisseur

```
vendu(Anom) :-  
    catalogue(_, Aid, _), articles(Aid, Anom, _).
```

```
SELECT anom FROM articles a WHERE exists  
    (select * from catalogue c where c.aid=a.aid)
```

Quantification universelle

Il faut passer par la négation, et donc des strates d'ordre supérieur.
Nous n'avons pas le temps de voir la stratification, une contrainte syntaxique, imposée en Datalog pour utiliser la négation.

Fonctionnalités supplémentaires de DES

Les opérations montrées jusqu'ici peuvent être faites avec n'importe quel DATALOG. Pour augmenter son attractivité, DES offre :

- ▶ fonctions d'agrégation : count, min,max,avg,sum
 - ▶ Versions avec 1,2 ou 3 arguments pour différents contextes.
- ▶ group by - having
- ▶ différentes jointures

Combien d'articles ?

```
DES> count(articles(_,_,_),C)

Info: Processing:
  answer(C) :-
    count(articles(_,_,_),[],C).
{
  answer(13)
}
Info: 1 tuple computed.
```

Contenu

Introduction a Datalog

- Faits et règles

- Différentes catégories de règles, et leur significations

- Requêtes récursives : chemins dans un graphe

Requêtes relationnelles en Datalog

- Opérations de l'algèbre relationnelle

- Aggregation etc en DES

Chemin le plus court : Datalog vs SQL

Semantique

- Sémantique des points fixes

- Exemple : graphe

- Mémoisation

- Sûreté

SAT

- Premier codage de formules booléennes en Datalog

- Qui est l'assassin ?

Différences entre Datalog et Prolog

Chemin le plus court en Datalog [source : F. Saenz-Perez]

```
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,           % assure la terminaison
    L is L0+1.         % cree L avec valeur L0+1

shortest_paths(X,Y,L) :-
    min(path(X,Y,Z),Z,L). % L: min pour Z
```

Le prédicat count/2, compte le nombre de résultats pour la requête (premier paramètre) et associe ce nombre au second paramètre.

La condition $L0 < Max$ assure la terminaison (elle interdit de boucler infiniment dans un cycle du graphe). Variables de min/3 : une fonction, Z le paramètre de cette fonction pour lequel on veut obtenir le minimum, qui est associé à L

Requêtes récursives en SQL

- ▶ requêtes récursives dans le standard SQL depuis la quatrième révision SQL :99 (nom alternatif : SQL3)
- ▶ en Postgres depuis version 8.4 (2009)

```
WITH [RECURSIVE] with_query [, ...]  
SELECT ...
```

syntaxe pour with_query :

```
query_name [ (column_name [, ...]) ]  
AS (SELECT ...)
```

- ▶ supposez la table *edge(origin, destination)* pour représenter le graphe

CREATE OR REPLACE VIEW

shortest_paths(Origin , Destination , Length) AS
WITH RECURSIVE

path(Origin , Destination , Length) AS
 (SELECT e.*,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 ;

% requete en SQL

SELECT * FROM shortest_paths ;

Contenu

Introduction a Datalog

- Faits et règles

- Différentes catégories de règles, et leur significations

- Requêtes récursives : chemins dans un graphe

Requêtes relationnelles en Datalog

- Opérations de l'algèbre relationnelle

- Aggregation etc en DES

Chemin le plus court : Datalog vs SQL

Semantique

- Sémantique des points fixes

- Exemple : graphe

- Mémoisation

- Sûreté

SAT

- Premier codage de formules booléennes en Datalog

- Qui est l'assassin ?

Différences entre Datalog et Prolog

Extension vs intension

Prédicats

- ▶ extension : les prédicats dont les relations sont enregistrées dans la base, comme faits.
- ▶ intention : des prédicats définis par des règles (c.a.d. en tête)

Terminologie

- ▶ EDB (extensional database) : collection de relations extensionnelles
- ▶ IDB (intensional database) : collection de relations intensionnelles

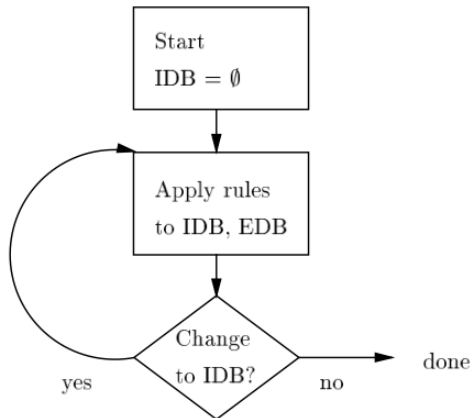
Exemple

Quels prédicats du schéma *bar* sont extensionnels, quels intensionnels ? Et pour les graphes et les chemins ?

Sémantique des points fixes

Idée de l'algorithme :

Iterative Fixed-Point Evaluates Recursive Rules

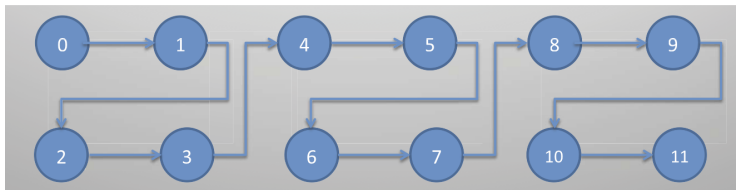


Sémantique des points fixes

Verbalisation de l'algorithme :

1. Suppose que tous les prédicats de la IDB sont vides.
Uniquement les faits de la EDB sont présents initialement.
2. Construction de relations IDB de plus en plus grandes :
 - ▶ Applique les règles aux tuples dans la EDB, et ajoute les tuples obtenus aux relations IDB.
 - ▶ Utilise les tuples ajoutés à la IDB dans l'étape précédente pour ajouter encore plus de tuples à la IDB, avec des nouvelles applications de règles.
3. Continue à appliquer les règles, **jusqu'à ce que cela n'ajoute plus de nouveaux tuples. On a atteint un point fixe.** Si les règles sont sûres, il n'y aura qu'un nombre fini de tuples satisfaisant les corps des règles, et donc, le point fixe sera atteint avec un nombre borné de répétitions.

Exemple pour la sémantique : graphe



Le prédicat $e/2$ exprime un **lien** direct entre deux sommets :

$e(0,1). e(1,2). e(2,3). e(3,4). e(4,5). e(5,6).$
 $e(6,7). e(7,8). e(8,9). e(9,10). e(10,11).$

Le prédicat $p/2$ exprime un **chemin (path)** entre deux sommets passant par un nombre arbitraire de liens :

$p(X,Y) :- e(X,Y).$

$p(X,Z) :- e(X,Y), p(Y,Z).$

Exemple : que peut-on déduire du programme ?

```
p(X,Y) :- e(X,Y).  
p(X,Z) :- e(X,Y), p(Y,Z).  
ok :- p(0,11).
```

Nous montrons qu'il existe un chemin de 0 à 11 (on peut déduire plus du programme).

- ▶ **instantiations des règles**, puis déduction de nouveaux tuples dans la IDB, utilisant des tuples existants.
- ▶ au tableau ...

EDB :

```
e(10,11).  
e(9,10).  
e(8,9).  
e(7,8).  
e(6,7).  
e(5,6).  
e(4,5).  
e(3,4).  
e(2,3).  
e(1,2).  
e(0,1).
```

IDB :

--

Exemple : que peut-on déduire du programme ?

$p(X,Y) :- e(X,Y).$

$p(X,Z) :- e(X,Y), p(Y,Z).$

$ok :- p(0,11).$

- ▶ pas 1 : règle 1 avec $e(10,11)$, ajout de $p(10,11)$ à la IDB.
- ▶ puis : règle 2 avec le prochain lien, et le dernier tuple ajouté à la IDB. injection d'un tuple supplémentaire la IDB. **répète.**
- ▶ règle 3 : quand possible.

EDB :

$e(10,11).$
 $e(9,10).$
 $e(8,9).$
 $e(7,8).$
 $e(6,7).$
 $e(5,6).$
 $e(4,5).$
 $e(3,4).$
 $e(2,3).$
 $e(1,2).$
 $e(0,1).$

IDB :

$p(10,11).$
 $p(9,11).$
 $p(8,11).$
 $p(7,11).$
 $p(6,11).$
 $p(5,11).$
 $p(4,11).$
 $p(3,11).$
 $p(2,11).$
 $p(1,11).$
 $p(0,11).$
 $ok.$

(pas encore
saturée!)

Comment expliquer l'absence de boucle infinie ?

```
a(X) :- b(X).                % test.dl  
  
b(X) :- a(X).
```

- ▶ DES rend le résultat rapidement pour la requête $a(4)$
- ▶ pourtant, la recherche de preuve devrait continuer à l'infini (d'après ce que nous avons vu)

Mémoisation

Technique d'optimisation de code visant à réduire le temps d'exécution. Datalog mémorise les faits qu'il a déjà prouvés, ou tenté de prouver. **Évite de répéter le même calcul deux fois**. Cette technique est absente en Prolog.

Commande DES pour voir ce qui est mémorisé : *list_et*

Motivation pour la sûreté

Sources de problèmes

Mauvaise utilisation des prédicats prédéfinis (comparaisons, négation ...), et des variables.

- ▶ Le résultat d'une requête doit être **une relation finie**
- ▶ Certains types de buts (sous-requetes) génèrent un nombre infinis de lignes, alors qu'il est impossible que la table d'une relation R soit de taille illimitée
- ▶ il faut éliminer maximum les warnings **unsafe**

Règles correctes, ou sûres

Une règle est sûre, quand

chaque variable, et notamment,

- ▶ dans la tête
- ▶ dans une négation
- ▶ dans une comparaison

apparaît également sous forme positive dans le corps de la règle, dans un prédicat défini dans le même programme.

Le résultat d'une requête est toujours de taille finie.

Exemples de violations :

```
p(X) :- q(Y).  
celibataire(X) :- not(marie(X,Y)).  
celibataire(X) :- personne(X), not(marie(X,Y)).  
homme(X) :- not(femme(X)).
```

Mauvais exemple avec variable

```
c(X) :- d.  
d.
```

Permet de prouver un nombre infini de faits.

Mauvais exemples avec comparaison et arithmétique

```
% insecure1.dl  
p(X) :- X>10.
```

```
% insecure2.dl  
  
c(Y) :- c(X), Y+1=X.
```

Requêtes closes vs requêtes avec variables.

Contenu

Introduction a Datalog

- Faits et règles

- Différentes catégories de règles, et leur significations

- Requêtes récursives : chemins dans un graphe

Requêtes relationnelles en Datalog

- Opérations de l'algèbre relationnelle

- Aggregation etc en DES

Chemin le plus court : Datalog vs SQL

Semantique

- Sémantique des points fixes

- Exemple : graphe

- Mémoisation

- Sûreté

SAT

- Premier codage de formules booléennes en Datalog

- Qui est l'assassin ?

Différences entre Datalog et Prolog

La clause WHERE en SQL

- ▶ renvoie, pour un tuple du tableau, soit vrai, soit faux
- ▶ est donc une **condition/formule booléenne**
- ▶ les formules booléennes ne sont pas triviales à coder
- ▶ tester la satisfaisabilité d'une formule booléenne est un problème difficile (connu sous **SAT**)

Satisfaisabilité d'une formule booléenne (SAT)

- ▶ la question s'il existe une interprétation, qui rend cette formule vraie
- ▶ interprétation : association de valeurs aux variables
- ▶ ce problème est **NP-complet** (algo en M1)
- ▶ on peut **vérifier** rapidement une solution
- ▶ on ne sait pas **trouver** les solutions efficacement, quand la taille des formules augmente

Formules booléennes et Datalog

- ▶ Datalog sait aussi bien trouver que vérifier des solutions a des problèmes.
- ▶ Pour pouvoir résoudre une problème logique, il faut savoir
 - ▶ d'abord l'exprimer en logique booléenne,
 - ▶ puis le **coder en Datalog**

Datalog nous rend la solution.

- ▶ Ce que nous faisons :
 - ▶ un puzzle logique, puis son codage en logique, puis en Datalog
 - ▶ préliminaire : **codage systématique** d'un formule booléenne, en Datalog, partant d'une formule booléenne en **forme normale conjonctive**

(A or not B) and (A or not C) and (B or C) and not A

$\text{bool}(0).$

$\text{bool}(1).$

$p1(A,B,C) :- A=1, \text{bool}(A), \text{bool}(B), \text{bool}(C).$

$p1(A,B,C) :- B=0, \text{bool}(A), \text{bool}(B), \text{bool}(C).$

$p2(A,B,C) :- A=1, \text{bool}(A), \text{bool}(B), \text{bool}(C).$

$p2(A,B,C) :- C=0, \text{bool}(A), \text{bool}(B), \text{bool}(C).$

$p3(A,B,C) :- B=1, \text{bool}(A), \text{bool}(B), \text{bool}(C).$

$p3(A,B,C) :- C=1, \text{bool}(A), \text{bool}(B), \text{bool}(C).$

$p4(A,B,C) :- A=0, \text{bool}(A), \text{bool}(B), \text{bool}(C).$

$r(A,B,C) :- p1(A,B,C), p2(A,B,C), p3(A,B,C), p4(A,B,C)$

$\text{sat}(A,B,C) :- r(A,B,C), \text{bool}(A), \text{bool}(B), \text{bool}(C).$

Un meurtre (1/2)

Connaissances autour d'un meurtre :

- M1 Si Jean n'a pas rencontré Pierre, alors soit Pierre est l'assassin, ou Jean est un menteur.
- M2 Si Pierre n'est pas l'assassin, alors Jean n'a pas rencontré Pierre, et le délit a eu lieu après minuit.
- M3 Si le délit a eu lieu après minuit, alors Pierre est l'assassin, ou Jean n'est pas un menteur.

Modélisation en logique propositionnelle :

Variables propositionnelles :

- ▶ A : Jean a rencontré Pierre
- ▶ B : Pierre est l'assassin
- ▶ C : Jean est un menteur
- ▶ D : le délit a eu lieu après minuit.

Un meurtre (1/2)

Formalisation

M1 $\text{not } A \Rightarrow (B \text{ or } C)$

M2 $\text{not } B \Rightarrow (\text{not}(A) \text{ and } D)$

M3 $D \Rightarrow (B \text{ or } \text{not}(C))$

Sol Conjonction des formules M1,M2 et M3

Réécriture sans implication

M1 $A \text{ or } B \text{ or } C$

M2 $B \text{ or } (\text{not}(A) \text{ and } D)$

M3 $\text{not}(D) \text{ or } B \text{ or } \text{not}(C)$

M1 $A \text{ or } B \text{ or } C$

M2 $B \text{ or } (\text{not}(A) \text{ and } D)$

M3 $B \text{ or } \text{not}(C) \text{ or } \text{not}(D)$

Sol = M1 and M2 and M3

```
bool(0).
```

```
bool(1).
```

```
m1(A,B,C) :- A=1,      bool(A),bool(B),bool(C).
```

```
m1(A,B,C) :- B=1,      bool(A),bool(B),bool(C).
```

```
m1(A,B,C) :- C=1,      bool(A),bool(B),bool(C).
```

```
m2(A,B,D) :- B=1,      bool(A),bool(B),bool(D).
```

```
m2(A,B,D) :- A=0,D=1, bool(A),bool(B),bool(D).
```

```
m3(B,C,D) :- B=1,      bool(B),bool(C),bool(D).
```

```
m3(B,C,D) :- C=0,      bool(B),bool(C),bool(D).
```

```
m3(B,C,D) :- D=0,      bool(B),bool(C),bool(D).
```

```
sol(A,B,C,D) :- m1(A,B,C),m2(A,B,D),m3(B,C,D).
```

La solution

- ▶ A : Jean a rencontré Pierre
- ▶ B : Pierre est l'assassin
- ▶ C : Jean est un menteur
- ▶ D : le délit a eu lieu après minuit.

```
DES> sol(A,B,C,D)
```

```
{  
  sol_1(1,1,0,0),  
  sol_1(1,1,1,0)  
}
```

```
Info: 2 tuples computed.
```

Contenu

Introduction a Datalog

- Faits et règles

- Différentes catégories de règles, et leur significations

- Requêtes récursives : chemins dans un graphe

Requêtes relationnelles en Datalog

- Opérations de l'algèbre relationnelle

- Aggregation etc en DES

Chemin le plus court : Datalog vs SQL

Semantique

- Sémantique des points fixes

- Exemple : graphe

- Mémoisation

- Sûreté

SAT

- Premier codage de formules booléennes en Datalog

- Qui est l'assassin ?

Différences entre Datalog et Prolog

Différences entre Datalog et Prolog

Prolog est vu au S6 dans l'UE Logique.

- ▶ Tout ce qui peut être écrit en Datalog (à l'exception des constructions spécifiques de DES pour le groupage et l'aggregation), peut être écrit en Prolog.
- ▶ Prolog permet d'écrire des faits plus complexes que Datalog (distinction entre symboles de fonction et prédicats). Prolog permet du filtrage de motif sur les termes complexes. Par exemple, des listes sont des termes complexes. On peut donc en Prolog écrire des fonctions sur des listes, ce qui n'est pas possible en Datalog.
- ▶ En Prolog, on dessine manuellement des arbres pour expliquer le comportement d'un programme. Ces arbres s'appellent des arbres de résolution.

Différences entre Datalog et Prolog

Prolog est vu au S6 dans l'UE Logique.

- ▶ Prolog, contrairement à Datalog, n'a pas de sémantique formelle. Nous avons vu en survol une des trois sémantiques de Datalog, aujourd'hui.
- ▶ En Prolog, il n'y a pas de stratification. En Datalog, la stratification rend l'utilisation de la négation sûre.
- ▶ Pour comprendre la négation en Prolog, il faut s'imaginer de couper des branches dans des arbres de preuve (voir UE Logique du S6).

Différences entre Datalog et Prolog

Prolog est vu au S6 dans l'UE Logique.

- ▶ Prolog n'a pas de message d'erreurs permettant de détecter des mauvais usages du langage. Il faut maîtriser des heuristiques syntaxiques complexes pour contourner des bugs. Ces derniers mènent à la non-termination de programmes. La non-termination peut être visualisée par des branches infinies dans l'arbre de résolution, qui ne peut que être obtenu manuellement. . .
- ▶ En Prolog, seulement certaines des variantes syntaxiques admises d'un programme se comportent correctement. Par exemple, en Prolog, pour la simple recherche dans un arbre (plus simple qu'un graphe), une seule variante fonctionne sur quatre.
- ▶ Toutefois, Prolog a des avantages. Les listes permettent d'écrire des programmes puissants. Nous allons en utiliser un en fin de semestre, en TP.