TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einsatz und Realisierung von Datenbanken* im SoSe25

Alice Rey, Maximilian Reif, Tobias Goetz (i3erdb@in.tum.de)
http://db.in.tum.de/teaching/ss25/impldb/

Blatt Nr. 05

Hinweise Die Datalogaufgaben können auf `https://souffle.db.in.tum.de/` getestet werden. Auf der Seite kann unter *examples* ein entsprechender Datensatz geladen werden. Die neuen IDB Regeln sollten am Ende der EDB definiert und dann im Query-Eingabefeld abgefragt werden.

Zusätzlich zu der in der Vorlesung vorgestellten Syntax hier noch eine Kurzübersicht der Vergleichsoperatoren: $X < Y, Y > X$ (kleiner, größer), $X <= Y, X >= Y$ (kleiner oder gleich, größer oder gleich), $X = Y, X! = Y$ (gleich, ungleich), $!pred(X, Y)$ (existiert nicht pred(X,Y)).

Hausaufgabe 1

Bleiben wir bei dem bekannten Universitätsschema:

```
Assistenten(PersNr, Name, Fachgebiet, Boss)
hoeren(MatrNr, VorlNr)
pruefen(MatrNr,VorlNr, PersNr, Note)
Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
Professoren(PersNr,Name,Rang, Raum)
voraussetzen(Vorg,Nachf)
Studenten(MatrNr, Name, Semester)
```

Formulieren Sie folgende Anfragen in Datalog und testen Sie sie:

a) Geben Sie alle *Professoren* an, die mindestens eine Prüfung abgehalten haben.

```
.decl pruefendeProfs(name: symbol)

pruefendeProfs(NAME) :- professoren(PNr,NAME,_,_),pruefen(_,_,PNr,_).
.output pruefendeProfs
```

b) Übersetzen Sie folgenden Ausdruck des Domänenkalküls in Datalog. Machen Sie sich der Bedeutung des Ausdrucks bewusst.

$$\{[t] \mid \exists\ v,s,g([v,t,s,g] \in \text{Vorlesungen} \wedge \exists\ v2([v,v2] \in \text{voraussetzen} \wedge \exists\ s2,g2([v2,\text{'Wissenschaftstheorie'},s2,g2] \in \text{Vorlesungen})))\}$$

Es sind die Titel der direkten Voraussetzungen für die *Vorlesung* Wissenschaftstheorie.

```
.decl vorWi(titel: symbol)
vorWi(Titel) :- vorlesungen(V,Titel,_,_),voraussetzen(V,V2),
                    vorlesungen(V2,"wissenschaftstheorie",_,_).
.output vorWi
```

c) Joinen Sie die nachfolgende Datalog-Anfrage so, dass die Titel der Vorlesungen ausgegeben werden. Was bedeutet diese Anfrage?

```
.decl geschwisterVL(N1: number, N2: number)
.decl nahverwandtVL(N1: number, N2: number)
geschwisterVL(N1,N2):-voraussetzen(V,N1),voraussetzen(V,N2), N1<N2.
nahverwandtVL(N1,N2):-geschwisterVL(N1,N2).
nahverwandtVL(N1,N2):-geschwisterVL(M1,M2),voraussetzen(M1,N1),
                              voraussetzen(M2,N2).
```

Es sind entweder „Geschwistervorlesungen“ (ein selber Vorfahre) oder „Vettern“ und „Basen“ (Cousins/Cousinen).

```
.decl nvVT(titel1: symbol, titel2: symbol)
nvVT(Titel1, Titel2) :- vorlesungen(N1,Titel1,_,_),
              vorlesungen(N2,Titel2,_,_),nahverwandtVL(N1,N2).
.output nvVT
```

Hausaufgabe 2

Geben Sie Datalog Regeln an, die Studenten (Namen angeben) finden, die von einem Prüfer geprüft worden, der selbst nicht die geprüfte Vorlesung gehalten hat. Das korrekte Ergebnis für diese Anfrage ist `Russels` Prüfling, `Carnap`. Führen Sie die Anfrage im Datalog Tool aus! .decl fremdgeprueft(SN: symbol, PID: number, VPID: number)

```
fremdgeprueft(SN,PID,VPID) :-
   studenten(SID,SN,_), pruefen(SID,V,PID,_),
   vorlesungen(V,_,_,VPID), PID!=VPID.
```

Hausaufgabe 3

Definieren Sie das Prädikat `sg(X,Y)` das für “same generation” steht. Zwei Personen gehören zur selben Generation, wenn Sie mindestens je ein Elternteil haben, das derselben Generation angehört.

Verwenden Sie beispielsweise die folgende Ausprägung einer ElternKind Relation. Das erste Element ist hier das Kind, das zweite ein Elternteil.

```
.decl parent(child: symbol, parent: symbol)
parent("c","a").
parent("d","a").
parent("d","b").
parent("e","b").
parent("f","c").
parent("g","c").
parent("h","d").
parent("i","d").
parent("i","e").
parent("f","e").
parent("j","f").
parent("j","h").
parent("k","g").
parent("k","i").
```

a) Definieren Sie das Prädikat in Datalog.

```
.decl sg(h1: symbol, h2: symbol)
```

Lösung: Vgl. Übungsbuch. / Tutorfolien

```
sg(x,x) :- parent(_,x). // X als Elternteil
sg(x,x) :- parent(x,_). // X als Kind
// X,Y Kind von U und V, U und V gleiche Generation
sg(x,y) :- sg(u,v),parent(x,u),parent(y,v).
.output sg
```

b) Demonstrieren Sie die naive Ausführung des Prädikats.

Schritt	S
Initialisierung	{}
Schritt 1	[c,c], [a,a], [d,d], [b,b], [e,e], [f,f], [g,g], [h,h], [i,i], [j,j], [k,k]
Schritt 2	[c,c], [a,a], [d,d], [b,b], [e,e], [f,f], [g,g], [h,h], [i,i], [j,j], [k,k] [c,d], [d,c], [d,e], [e,d], [f,g], [g,f], [h,i], [i,h], [i,f], [f,i]
Schritt 3	[c,c], [a,a], [d,d], [b,b], [e,e], [f,f], [g,g], [h,h], [i,i], [j,j], [k,k], [c,d], [d,c], [d,e], [e,d], [f,g], [f,h], [f,i], [g,f], [g,h], [g,i], [h,i], [h,i], [h,f], [h,g], [i,h], [i,f], [i,g], [j,k], [k,j]
Schritt 4	liefert keine zusätzlichen Elemente! Algorithmus terminiert

c) Erläutern Sie das Vorgehen bei der seminaiven Auswertung.

Schritt	ΔS
Initialisierung	[c,c], [a,a], [d,d], [b,b], [e,e], [f,f], [g,g], [h,h], [i,i], [j,j], [k,k]
Schritt 1	[c,d], [d,c], [d,e], [e,d], [f,g], [g,f], [h,i], [i,h], [i,f], [f,i]
Schritt 2	[f,h], [g,h], [g,i], [h,f], [h,g], [i,g], [j,k], [k,j]
Schritt 3	bringt keine zusätzlichen Elemente! Algorithmus terminiert

Gruppenaufgabe 4

Ist folgendes Datalog-Programm stratifiziert?

$$\begin{aligned} p(X,Y) &:- q_1(Y,Z), \neg q_2(Z,X), q_3(X,P). \\ q_2(Z,X) &:- q_4(Z,Y), q_3(Y,X). \\ q_4(Z,Y) &:- p(Z,X), q_3(X,Y). \end{aligned}$$

Ist das Programm sicher – unter der Annahme, dass p, q_1, q_2, q_3, q_4 IDB- oder EDB-Prädikate sind?

Loesung: Vgl. Übungsbuch. Das Programm ist **nicht stratifizert**, aber **sicher**. Es ist nicht stratifiziert, weil q_2 von p abhängt, aber negiert in p vorkommt. Es ist sicher, weil alle Variablen in IDB- oder EDB-Prädikaten gebunden sind.
Zur Wiederholung: Eine Regel ist **sicher** gdw. alle Variablen **eingeschränkt** sind. Variable X ist in einer Regel **eingeschränkt**, falls sie im Rumpf enthalten ist und sie:

- in einem positiven Prädikat vorkommt (nicht Vergleichsprädikat),
- $X = c$ (Konstante) oder
- $X = Y$, wenn Y bereits nachgewiesen ist.

Jede Variable eines negierten Prädikates muss bereits eingeschränkt sein.[a][b]
Ein Datalog-Programm ist **stratifiziert**, wenn in einer Regel p alle negierten Prädikate $not(q_i)$ nicht von p abhängen (kein Zyklus) und alle positiven Prädikate vorher oder unabhängig von der Reihenfolge (wie bei Rekursion) ausgewertet werden. Formal ausgedrückt können wir jedem Prädikat eine sogenannte Stratifikationsnummer zuordnen. Negierte Prädikate müssen eine echt kleinere Stratifikationsnummer besitzen, positive Prädikate eine maximal so große Stratifikationsnummer wie das davon abhängige Prädikat.

[a] `https://www.dbis.informatik.uni-goettingen.de/Teaching/DB/db-datalog.pdf`
[b] `http://pages.cs.wisc.edu/~paris/cs784-s17/lectures/lecture9.pdf`

Gruppenaufgabe 5

Gegeben sei folgende Faktenbasis, die einen direkten azyklischen Graphen (DAG) darstellt.

```
.decl kante(a: number, b: number)
kante(1,2).
kante(2,3).
kante(3,4).
kante(2,5).
kante(5,3).
```

1. Geben Sie in Datalog ein Prädikat `pfad(V,N,L)` an, dass alle möglichen Pfade von `V` nach `N` mit Länge `L` ausgibt.

   ```
   .decl pfad(von: number, nach: number, laenge: number)


   pfad(V,N,L) :- kante(V,N), L=1.
   pfad(V,N,L) :- pfad(V,X,LX), kante(X,N), L=LX+1.
   ```

2. Geben Sie nun das Prädikat `kuerzestePfade(V,N,L)` an, das pro Beginn `V` und Ziel `N` nur den kürzesten Pfad ausgibt.

   ```
   .decl langepfade(von: number, nach: number, laenge: number)
   langepfade(V,N,L2) :- pfad(V,N,L), pfad(V,N,L2), L<L2.
   .decl kuerzestePfade(von: number, nach: number, laenge: number)
   kuerzestePfade(V,N,L) :- pfad(V,N,L), !langepfade(V,N,L).
   .output kuerzestePfade
   ```

3. Bestimmen Sie nun den längsten kürzesten Pfad `.decl laengsterkuerzesterPfad(L: number)`.

```
.decl kurzekuerzestePfade(L: number)
kurzekuerzestePfade(L) :- kuerzestePfade(_,_,L), kuerzestePfade(_,_,L2), L<L2.
laengsterkuerzesterPfad(L) :- kuerzestePfade(_,_,L), !kurzekuerzestePfade(L).
.output laengsterkuerzesterPfad
```

4. Erstellen Sie in SQL eine rekursive CTE `pfad(V,N,L)`, die die Länge aller Pfade im DAG ausgibt.

```
with recursive kante(V,N) as (values (1,2), (2,3), (3,4),(2,5),(5,3)),
  pfad(V,N,L) as ( select k.V, k.N, 1 from kante k union
   select k.V, p.N, p.L + 1 from kante k, pfad p where k.N = p.V)
```

5. Basierend auf `pfad(V,N,L)`, geben Sie die Länge des längsten kürzesten Pfades aus.

```
select max(min) from (select v,n,min(l) from pfad group by v,n) tmp;
```

Hausaufgabe (wird nicht in der Übung besprochen)

Schreiben Sie zu dem U-Bahn-Netz-Beispiel auf der Datalog Seite (unter Examples) folgende Anfragen in Datalog:

1. Erstellen Sie den Stationsplan für den U-Bahnhof Fröttmanning, der alle Stationen, die ohne Umstieg erreichbar sind, auflistet.

```
bidirekt(A,B,L) :- direkt(A,B,L), A!=B.
bidirekt(A,B,L) :- direkt(B,A,L), A!=B.
bidirekt(A,B,L) :- bidirekt(A,X,L), direkt(X,B,L), A!=B.

.decl froettmanning_direkt(B: symbol)
froettmanning_direkt :- bidirekt("froettmanning",B,_)
```

2. Erstellen Sie für Garching-Forschungszentrum einen Plan, der alle erreichbaren Stationen, die minimale Anzahl an Umstiegen und Stops auflistet. Beschreiben Sie Ihren Ansatz ausführlich.

Vereinfachte Lösung (betrachten nur in Fahrtrichtung)

```
.decl aufwand(von: symbol, nach: symbol, linie: symbol,
    stopps: number, umstiege: number)
% Erreichbar naechster Stop
aufwand(A,B,L,S,U) :- direkt(A,B,L), S=0, U=0, A="garching_forschungszentrum".
% Erreichbar auf gleicher Linie
aufwand(A,B,L,S,U) :- aufwand(A,C,L,SX,UX), direkt(C,B,L), S=SX+1, U=UX.
% Erreichbar durch umsteigen
aufwand(A,B,L,S,U) :- aufwand(A,C,LA,SX,UX), direkt(C,B,LB),
   S=SX+1, LA!=LB, L=LB, U=UX+1.
```

Lösung mit Richtungs- und Linienwechsel.

```
% Merke Richtung in die gefahren wird (R=vorwaerts oder rueckwaerts)
.decl bdirekt(von: symbol, nach: symbol, linie: symbol, richtung: symbol)
bdirekt(A,B,L,R) :- direkt(A,B,L), R="v".
bdirekt(A,B,L,R) :- direkt(B,A,L), R="r".

% Maximale Anzahl der Stopps und maximale Anzahl der Umstiege, ist noetig
% falls die Rekursion in einem Kreis im Graph festhaengt.
% Ohne Aggregation einfach 49 statt SMAX bei aufwand(...) einsetzen
.decl smax(x: number)
smax(SMAX) :- SMAX = count: direkt(_,_,_).
.decl linien(x : symbol)
linien(x) :- direkt(_,_,x).
.decl umax(x: number)
umax(UMAX) :- UMAX = count: linien(_).
% Erreichbar naechster Stop
.decl aufwand(von: symbol, nach: symbol, linie: symbol, richtung: symbol,
    stopps: number, umstiege: number)
aufwand(A,B,L,R,S,U) :- bdirekt(A,B,L,R), S=1, U=0, A!=B.
% Erreichbar auf gleicher Linie
aufwand(A,B,L,R,S,U) :- aufwand(A,C,L,R,SX,UX), bdirekt(C,B,L,R),
   S=SX+1, U=UX, A!=B, smax(SMAX), S<SMAX.
% Erreichbar durch Umsteigen auf andere Linien.
% Richtungswechsel erlaubt.
% Maximal 5 Umstiege möglich, da nur 6 verschiedene Linien
aufwand(A,B,L,R,S,U) :- aufwand(A,C,LA,_,SX,UX), bdirekt(C,B,LB,R),
   S=SX+1, LA!=LB, L=LB, U=UX+1, A!=B, umax(UMAX), U<UMAX, smax(SMAX), S<SMAX.
```

Im *aufwand* Prädikat ist ein Richtungswechsel ohne gleichzeitigen Linienwechsel nicht berücksichtigt. Dies ist auch nicht notwendig, da am Startpunkt durch *bdirekt* in beide Richtungen gestartet werden kann, und bei einem Linienwechsel dann auch jedesmal die Möglichkeit besteht, die Richtung frei zu wählen.
Wegen der Struktur der Linien im Beispielgraph (sie treffen sich nur am Sendlinger Tor) ist die Loesung des *aufwand* Prädikats schon jeweils die kürzeste Strecke. Bei einem Netz mit zwei Treffpunkten wären durch die Richtungswechsel Kreise möglich und das Minimum für jede Strecke müsste mit folgendem Prädikat gefunden werden.

```
.decl minaufwand(von: symbol, nach: symbol, stopps: number, umstiege: number)
minaufwand(A,B,S,U) :- aufwand(A,B,_,_,S,_),aufwand(A,B,_,_,_,U),
   U = min ux : aufwand(A,B,_,_,_,ux),
   S = min sx : aufwand(A,B,_,_,sx,_).

.decl minaufwand_gf(nach: symbol, stopps: number, umstiege: number)
minaufwand_gf(B,S,U) :- minaufwand("garching_forschungszentrum",B,S,U).
.output minaufwand_gf
```

3. Gibt es zwischen allen Paaren von Stationen Verbidungen mit maximal einmaligem Umsteigen? Versuchen Sie Gegenbeispiele mittels Datalog zu finden.

```
// Paare von Stationen, die mit genau einmal Umsteigen verbunden sind
.decl einUmstieg(start: symbol, change: symbol, end: symbol)
einUmstieg(s,c,e) :- bidirekt(s, c, l), bidirekt(c, e, m), l != m.

// Gegenbeispiele werden ausgeben
.decl res(start: symbol, end: symbol)
.output res
res(s,e) :- bidirekt(s, _, _), bidirekt(e, _, _), !einUmstieg(s,_,e), !bidirekt(s,e,_).
```