Datalog query language

Datalog is a logical query language (Data Logic). See textbook (Ullman ch. 5.3, 5.4 and 10.2)

Expressing relational algebra operations in Datalog

$R \cap S$

Answer(x1, …, xn) ← R(x1, ..., xn) AND S(x1, ..., xn)

$R - S$

Answer(x1, …, xn) ← R(x1, ..., xn) AND NOT S(x1, ..., xn)

$R \cup S$

Answer(x1, …, xn) ← R(x1, ..., xn)
Answer(x1, …, xn) ← S(x1, ..., xn)

$\sigma_{Xi=Xj}(R)$

Answer(x1, …, xn) ← R(x1, ..., xn) AND Xi = Xj

$\pi_{X1,X2}(R)$

Answer(x1, x2) ← R(x1, x2, ..., xn)

$R \times S$

Answer(x1, …, xn, y1, …, ym) ← R(x1, x2, ..., xn) AND S(y1, ..., ym)

$R \bowtie S$

Answer(x1, …, xn, y1, …, ym, z1, …, zk) ← R(x1, ..., xn, z1, … zk) AND S(y1, ..., ym, z1, … zk)

Some queries for relation Likes (name, fruits)

Who likes apple?
Answer(X) ← Likes(X, ’apple') (or Answer(X) ← Likes(X, Y) AND Y='apple')

List those names who doesn't like pear but like something else.
Answer(X) ← Likes(X, Y) AND NOT Likes(X, 'pear')

Who likes both apple and pear?
Answer(X) ← Likes(X, apple') AND Likes(X, 'pear')

Who likes raspberry or pear?
Answer(X) ← Likes(X, 'raspberry')
Answer(X) ← Likes(X, 'pear')

Who likes apple but doesn't like pear?
Answer(X) ← Likes(X, 'apple') AND NOT Likes(X, 'pear')

Who likes at least two different fruits?
Answer(X) ← Likes(X, Y) AND Likes(X, Z) AND Y<>Z

Who likes at least three different fruits?
Answer(X) ← Likes(X, Y) AND Likes(X, Z) AND Likes(X, W)
AND Y<>Z AND Z<>W AND Y<>W

Who likes at most two different fruits?
Min3(X) ← Likes(X, Y) AND Likes(X, Z) AND Likes(X, W)
AND Y<>Z AND Z<>W AND Y<>W
Max2(X) ← Likes(X, Y) AND NOT Min3(X)

Expressing the Datalog query in SQL using WITH

```
WITH
Min3(N) AS (
  SELECT DISTINCT L1.name FROM Likes L1, Likes L2, Likes L3
  WHERE L1.name=L2.name AND L2.name=L3.name
  AND L1.fruits <> L2.fruits AND L2.fruits <> L3.fruits
  AND L1.fruits <> L3.fruits)
SELECT name FROM Likes MINUS SELECT N FROM Min3;
```

Who likes exactly two different fruits?
Min2(X) ← Likes(X, Y) AND Likes(X, Z) AND Y<>Z
Min3(X) ← Likes(X, Y) AND Likes(X, Z) AND Likes(X, W)
AND Y<>Z AND Z<>W AND Y<>W
Exactly2(X) ← Min2(X) AND NOT Min3(X)

```
WITH
Min2(N) AS (
 SELECT DISTINCT L1.name FROM Likes L1, Likes L2
 WHERE L1.name=L2.name AND L1.fruits <> L2.fruits),
Min3(N) AS (
  SELECT DISTINCT L1.name FROM Likes L1, Likes L2, Likes L3
  WHERE L1.name=L2.name AND L2.name=L3.name
  AND L1.fruits <> L2.fruits AND L2.fruits <> L3.fruits
  AND L1.fruits <> L3.fruits)
SELECT N FROM Min2 MINUS SELECT N FROM Min3;
```

Who likes every fruit?
NotLikes(N,F) ← Likes(N,X) AND Likes(Y,F) AND NOT Likes(N,F)
LikesEvery(N) ← Likes(N,X) AND NOT NotLikes(N,Y)

```
WITH
NotLikes(N,F) AS (
  SELECT L1.name, L2.fruits FROM Likes L1, Likes L2
    MINUS
  SELECT name, fruits FROM Likes)
SELECT name FROM Likes MINUS SELECT N FROM NotLikes;
```

Recursive Datalog

An example from the Lecture slides:

Extensional (EDB) relation:
Par(c,p) -> meaning: p is a parent of c (child)
Sibling means brother or sister
Generalized cousins: people with common ancestors one or more generations back:

Recursive Datalog program expressing Cousins:

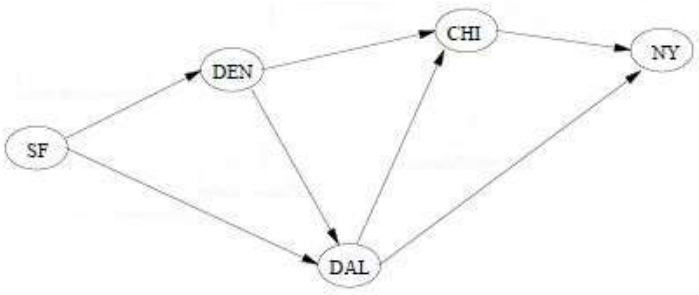
Sib(x,y) ← Par(x,p) AND Par(y,p) AND x<>y
Cousin(x,y) ← Sib(x,y)
Cousin(x,y) ← Par(x,xp) AND Par(y,yp) AND Cousin(xp,yp)

Expressing the recursive Datalog query in SQL using WITH

```
WITH
Sib(a,b) AS
  (SELECT p1.c, p2.c FROM Par p1, Par p2
   WHERE p1.p = p2.p AND p1.c <> p2.c),
Cousin(x,y) AS
  (SELECT * FROM Sib
    UNION ALL
   SELECT p1.c, p2.c FROM Par p1, Par p2, Cousin
   WHERE p1.p = Cousin.x AND p2.p = Cousin.y)
CYCLE x,y SET is_cycle TO 'Y' DEFAULT 'N'
SELECT DISTINCT * FROM Cousin WHERE x <= y ORDER BY 1,2;
```

A new table for recursive "Datalog-like" queries
(see in TextBook and in UW_10_2_Recursion.pdf)

FLIGHT(airline VARCHAR2(10), orig VARCHAR2(15), dest VARCHAR2(15), cost NUMBER);




```
INSERT INTO flight VALUES('Lufthansa', 'San Francisco', 'Denver', 1000);
INSERT INTO flight VALUES('Lufthansa', 'San Francisco', 'Dallas', 10000);
INSERT INTO flight VALUES('Lufthansa', 'Denver', 'Dallas', 500);
INSERT INTO flight VALUES('Lufthansa', 'Denver', 'Chicago', 2000);
INSERT INTO flight VALUES('Lufthansa', 'Dallas', 'Chicago', 600);
INSERT INTO flight VALUES('Lufthansa', 'Dallas', 'New York', 2000);
```

```
INSERT INTO flight VALUES('Lufthansa', 'Chicago', 'New York', 3000);
INSERT INTO flight VALUES('Lufthansa', 'Chicago', 'Denver', 2000);
```

The tuples are not the same as in TextBook. I inserted an extra tuple in order to be a cycle in the graph of Flight table: orig = 'Chicago' and dest = 'Denver';

Let's see relation Reaches(x,y) which answers the following query.
For what pairs of cities (x, y) is it possible to get from city x to city y by taking one or more flights?

A recursive DATALOG program for the REACHES table:
Reaches(X,Y) ← Flight(X,Y)
Reaches(X,Y) ← Reaches(X,Z) AND Reaches(Z,Y) AND X <> Y

Most DBMS-s allow only linear recursion in recursive queries, which means:
no rule has more than one subgoal that is mutually recursive with the head.
(See mutual recursion in textbook, ch. 10.2)
In the previous datalog program the second rule violates this requirement.

The datalog program below contains only linear recursion:

Reaches(X,Y) ← Flight(X,Y)
Reaches(X,Y) ← Flight(X,Z) AND Reaches(Z,Y) AND X <> Y

SQL query

```
WITH  reaches(orig, dest) AS
 (
  SELECT orig, dest FROM flight
   UNION ALL
  SELECT flight.orig, reaches.dest FROM flight, reaches
  WHERE flight.dest = reaches.orig AND flight.orig <> reaches.dest
  )
  CYCLE orig,dest SET is_cycle TO 'Y' DEFAULT 'N'
SELECT  distinct orig, dest, is_cycle FROM reaches order by 1,2,3;
```