CS 500: Fundamentals of Databases

Datalog
Recursive SQL
LogicBlox

supplementary material:
“Foundations of Databases” by Abiteboul, Hull, Vianu, Ch. 12
class notes

slides are based on teaching materials by
Serge Abiteboul and Jeff Ullman
Julia Stoyanovich (stoyanovich@drexel.edu)
Transitive closure of a graph

Graph $G$, represented by an edge relation $G(x, y)$

Q: Is there a path from 0 to 11?

Can we write this in SQL?
Datalog basics

\[ P(x,y) : \neg G(x,y) \]
\[ P(x,y) : \neg G(x,z), P(z,y) \]
\[ Q() : \neg P(0,11) \]

- Datalog stands for “database logic”, syntax is similar to Prolog, but there are important differences
- Programs are sets or rules, with 1 head relation predicate and 0 or more body relation predicates
- Read a rule as “if body then head”, e.g., “if there is an edge from x to y, then there is a path from x to y”
- Predicates in the body are also called subgoals, rule body is a conjunction (logical and) of its subgoals
- A program has a distinguished rule called query - this is what the program computes, other rules are in support of that computation

\( G \) stands for edges
\( P \) stands for paths
\( Q \) stands for query
Datalog basics

- A relation is *extensional* (*edb*) if it does not appear in the head of any rule.
- A relation is *intensional* (*idb*) if it appears in the head of at least 1 rule.
- Distinguishing feature of datalog - support for recursion. Most common case of recursion is when a relation appears in the head and in the body of the same rule.
- Not all datalog programs are recursive (will see many non-recursive programs today).
- We will look at set semantics of datalog, bag semantics is also possible.

Datalog Rule Set:

$P(x,y): \neg G(x,y)$

$P(x,y): \neg G(x,z), P(z,y)$

$Q(): \neg P(0,11)$

Initial Knowledge Base $I$:

$I: \{G(0,1), G(1,2), G(2,3)\}$

Tuples are called *facts*.
Datalog by example

\[ A(x, y) : \neg B(x, y) \]
\[ A(x, y) : \neg C(x, y, _, _) \]
\[ T() : \neg A(10, z) \]

\[ Q(x, y) : \neg R(x, z), \ S(z, y), \ x < y \]
\[ P(x, x) : \neg E(x, _) \]

- Variables are local
- If a variable is not reused, we don’t have to name it (use ‘_’)
- Arithmetic predicates are allowed in the body in addition to relational predicates
- Unlike in Prolog:
  - function symbols are not supported
  - data is separate from the program, a program executes on different \textit{edb} instances (its input), \textbf{maps finite \textit{edb} instances to finite \textit{idb} instances (its output)}

\( x, y, z \) are variables

10 is a constant
Safety

Every variable that appears anywhere in the rule must appear in a positive relational subgoal of the body

(We focus on positive datalog (no negation) for most of this class, but will illustrate some issues related to negation)

\[ V(x, y) : \neg P(x) \]
\[ W(x, y) : \neg P(x), y > x \] safe? explain!
\[ Q(x, y) : \neg R(x, z), S(z, y), T(x, y) \]
\[ U(x, y) : \neg R(x, z), S(z, y), T(x, w) \]

Why this condition?
Understanding datalog rules

\[ P(x, y) \leftarrow Q(x, z), R(z, y), Q(x, y) \]

\[ I : \{ Q(1,2), Q(1,3), R(2,3), R(3,1) \} \]

<table>
<thead>
<tr>
<th>( Q(x, z) )</th>
<th>( R(z, y) )</th>
<th>Consistent?</th>
<th>( Q(x, y) )</th>
<th>( P(x, y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>(2,3)</td>
<td>TRUE</td>
<td>FALSE</td>
<td>null</td>
</tr>
<tr>
<td>(1,2)</td>
<td>(3,1)</td>
<td>FALSE</td>
<td>N/A</td>
<td>null</td>
</tr>
<tr>
<td>(1,3)</td>
<td>(2,3)</td>
<td>FALSE</td>
<td>N/A</td>
<td>null</td>
</tr>
<tr>
<td>(1,3)</td>
<td>(3,1)</td>
<td>TRUE</td>
<td>TRUE</td>
<td>(1,1)</td>
</tr>
</tbody>
</table>
A fact $A$ is an **immediate consequence** for instance $I$ and program $P$ if

1. $A$ is an extensional fact in $I$ or,
2. for some instantiation of a rule $A :- A_1, \ldots, A_n$ in $P$, and each $A_1, \ldots, A_n$ is in $I$.

$T_P(K)$, the immediate consequence operator, is **monotone**

That is, the output of each subsequent round is a subset of the output of the previous round. When it is no longer a proper subset (no new facts are derived), the program terminates.
Example 12.3.3  Recall the program $P_{TC}$ for computing the transitive closure of a graph $G$:

$$T(x, y) \leftarrow G(x, y)$$
$$T(x, y) \leftarrow G(x, z), T(z, y).$$

Consider the input instance

$$I = \{G(1, 2), G(2, 3), G(3, 4), G(4, 5)\}.$$

Then we have

$$T_{P_{TC}}(I) = I \cup \{T(1, 2), T(2, 3), T(3, 4), T(4, 5)\}$$
$$T^2_{P_{TC}}(I) = T_{P_{TC}}(I) \cup \{T(1, 3), T(2, 4), T(3, 5)\}$$
$$T^3_{P_{TC}}(I) = T^2_{P_{TC}}(I) \cup \{T(1, 4), T(2, 5)\}$$
$$T^4_{P_{TC}}(I) = T^3_{P_{TC}}(I) \cup \{T(1, 5)\}$$
$$T^5_{P_{TC}}(I) = T^4_{P_{TC}}(I).$$

Thus $T^\omega_{P_{TC}}(I) = T^4_{P_{TC}}(I)$. 

---

Julia Stoyanovich
Naive evaluation

Algorithm Naive: Given a set of rules and an edb $I$

(1) Initialize all idb relations as empty

(2) Instantiate (with constants) variables of all rules in all possible ways. If all subgoals become true - we are inferring new fact(s) for the head relation

(3) Repeat (2) in rounds, as long as new idb facts are being inferred

Step (2) makes sense for finite edb and safe rules - each variable appears in some non-negated relational subgoal, this limits the active domain

Steps (1) - (3) converges to the least fixed point of the program and the edb, we’ll elaborate on this
Naive evaluation

\[ P(x,y) : -G(x,y) \]
\[ P(x,y) : -G(x,z), P(z,y) \]

Algorithm Naive: Given a set of rules and an \( edb \) \( I \):

1. Initialize all \( idb \) relations as empty
2. Instantiate (with constants) variables of all rules in all possible ways. If all subgoals become true - we are inferring new fact(s) for the head relation
3. Repeat (2) in rounds, as long as new \( idb \) facts are being inferred

\( I : \{ G(0,1), G(1,2), G(2,3) \} \)

\[ init : P(x,y) = \emptyset \]
round 1: \( P(0,1) P(1,2) P(2,3) \)
round 2: \( P(0,1) P(1,2) P(2,3) P(0,2) P(1,3) \)
round 3: \( P(0,1) P(1,2) P(2,3) P(0,2) P(1,3) P(0,3) \)
Example: Paris subway

Concorde ← Tuileries ← Palais-Royal ← Louvre ← Châtelet

St.-Germain → Odeon → St.-Michel

Pont de Sevres → Billancourt → Michel-Ange

Voltaire ← Republique ← FDR ← Iena

Links(line, station, next_station)
Additional examples

Links(line,station,next_station)

What are the stations reachable from Odeon? - directed graph version

\[ St_{reachable}(x,y) : \neg Links(_,x,y) \]
\[ St_{reachable}(x,y) : \neg Links(_,x,z), St_{reachable}(z,y) \]
\[ Q(y) : \neg St_{reachable}('Odeon',y) \]

What are the stations reachable from Odeon? - undirected graph version

\[ St_{reachable}(x,y) : \neg Links(_,x,y) \]
\[ St_{reachable}(x,y) : \neg Links(_,y,x) \]
\[ St_{reachable}(x,y) : \neg St_{reachable}(x,z), St_{reachable}(z,y), x \neq y \]
\[ Q(y) : \neg St_{reachable}('Odeon',y) \]
Additional examples

\text{Links(line,station,next\_station)}

What are the lines reachable from Odeon? - \text{directed graph version}

\begin{align*}
\text{St\_reachable}(x,y) & : \neg \text{Links}(_,x,y) \\
\text{St\_reachable}(x,y) & : \neg \text{Links}(_,x,z),\text{St\_reachable}(z,y) \\
\text{Li\_reachable}(x,u) & : \neg \text{Links}(u,x,\_ ) \\
\text{Li\_reachable}(x,u) & : \neg \text{Links}(u,\_,x) \\
\text{Li\_reachable}(x,u) & : \neg \text{St\_reachable}(x,y),\text{Li\_reachable}(y,u) \\
Q(u) & : \neg \text{Li\_reachable}'Odeon',u)
\end{align*}
Additional examples

`Links(line, station, next_station)`

What are the lines reachable from Odeon? - undirected graph version

```prolog
St_reachable(x, y) :- Links(_, x, y)
St_reachable(x, y) :- Links(_, y, x)
St_reachable(x, y) :- St_reachable(x, z), St_reachable(z, y), x ≠ y
Li_reachable(x, u) :- Links(u, x, _)
Li_reachable(x, u) :- Links(u, _, x)
Li_reachable(x, u) :- St_reachable(x, y), Li_reachable(y, u)
Q(u) :- Li_reachable('Odeon', u)
```
Additional examples

Can we go from Odeon to Chatelet?
\[ Q() : \neg St\_reachable('Odeon','Chatelet') \]

Are all pairs of stations connected?
\[ All\_pairs(x,y) : \neg Links(_,x,_) , Links(_,_,y), x \neq y \]
\[ Q() : \neg All\_pairs(x,y) , St\_reachable(x,y) \]

Both programs can run on either the directed or the undirected version of \( St\_reachable \).

Is there a cycle in the graph? - directed graph version only
\[ Q() : \neg St\_reachable(x,x) \]
Relational algebra vs. datalog

- Any relational algebra expression can be expressed in datalog - we’ll see how to do this now
- Any (one) datalog rule can be expressed in relational algebra
- A multi-rule datalog program is strictly more powerful than relational algebra, since rules are allowed to interact - recursion

- A syntactic difference: in datalog we do not refer to attributes by name, only by position (this is logic programming syntax, since a relation with attributes is viewed as a predicate with arguments)
- Both relational algebra and datalog work with sets, although both can be extended to bags
- We are working with positive datalog here, and positive relational algebra (no set difference)
## Relational algebra vs. Datalog

<table>
<thead>
<tr>
<th>Relational Algebra</th>
<th>Datalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(A, B, C)$</td>
<td>$S(A, B, C)$</td>
</tr>
<tr>
<td>$\sigma_{A&gt;3}(R)$</td>
<td>$Q(x, y, z) : \neg R(x, y, z), x &gt; 3$</td>
</tr>
<tr>
<td>$\pi_{A, B}(R)$</td>
<td>$Q(x, y) : \neg R(x, y, _)$</td>
</tr>
<tr>
<td>$R \cap S$</td>
<td>$Q(x, y, z) : \neg R(x, y, z), S(x, y, z)$</td>
</tr>
<tr>
<td>$R \cup S$</td>
<td>$Q(x, y, z) : \neg R(x, y, z)$</td>
</tr>
<tr>
<td>$Q(x, y, z) : \neg S(x, y, z)$</td>
<td></td>
</tr>
<tr>
<td>$R \times S$</td>
<td>$Q(u, v, w, x, y, z) : \neg R(u, v, w), S(x, y, z)$</td>
</tr>
<tr>
<td>$R \bowtie S$</td>
<td>$Q(x, y, z) : \neg R(x, y, z), S(x, y, z)$</td>
</tr>
</tbody>
</table>
Monotonicity

- Recall fixpoint semantics: convergence depends on monotonicity of $T_P(K)$, the immediate consequence operator.

- That is, the output of each subsequent round is a superset of the output of the previous round. When it is no longer a proper superset (no new facts are derived), the program terminates.

- Monotonicity of positive datalog can be shown based on monotonicity of union, select, project, product, join operators in relational algebra.

- With this, and using induction on the number of rounds, one can show that naive datalog evaluation produces the least fixed point, i.e., the minimal set of facts provable from the edb by the program.
Negation

- Why is negation problematic: negative relational subgoals in datalog?
- When combined with recursion, negation introduces non-monotone behavior, i.e., the program may not converge to a unique fixed point.
- Example: a fact from R is either in P or in Q, but not in both

\[
P(x) : \neg R(x), Q(x)
\]
\[
Q(x) : \neg R(x), P(x)
\]

\[I = \{R(0)\}\]

<table>
<thead>
<tr>
<th>round</th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{(0)}</td>
<td>{(0)}</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>3</td>
<td>{(0)}</td>
<td>{(0)}</td>
</tr>
<tr>
<td>4</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

aggregation + recursion can also lead to non-monotone behavior
Recursive SQL

- The WITH statement in SQL allows to define temporary relations (recursive or not)
- To write recursive WITH statements, use the RECURSIVE keyword and use the temporary relation within the statement that defines it
- Example: Flights relation below, what cities are reachable from San Francisco (SFO)

```sql
create table Flights(
    airline char(2),
    src char(3),
    dest char(3),
    departs time,
    arrives time,
    primary key (airline, src, dest, departs)
);

with recursive Reaches(src, dest) as (
    select src, dest from Flights
    union
    select R1.src, R2.dest
    from Reaches R1, Reaches R2
    where R1.dest = R2.src)
select * from Reaches;
```

**ERROR:** recursive reference to query "reaches" must not appear more than once
**LINE 5:** from Reaches R1, Reaches R2
Recursive SQL

- The WITH statement in SQL allows to define temporary relations (recursive or not)
- To write recursive WITH statements, use the RECURSIVE keyword and use the temporary relation within the statement that defines it
- Example: Flights relation below, what cities are reachable from San Francisco (SFO)

```sql
create table Flights(
  airline char(2),
  src char(3),
  dest char(3),
  departs time,
  arrives time,
  primary key (airline, src, dest, departs)
);
```

```sql
with recursive Reaches(src, dest) as ( 
  select src, dest from Flights 
  union 
  select R.src, F.dest 
  from Reaches R, Flights F 
  where R.dest = F.src 
)
select * from Reaches where src = 'SFO';
```

<table>
<thead>
<tr>
<th>src</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFO</td>
<td>DEN</td>
</tr>
<tr>
<td>SFO</td>
<td>DAL</td>
</tr>
<tr>
<td>SFO</td>
<td>NYC</td>
</tr>
<tr>
<td>SFO</td>
<td>CHI</td>
</tr>
</tbody>
</table>

(4 rows)
Recursion

\[ S(x,y) := \neg P(x,z), P(y,z), x \neq y \]
\[ C(x,y) := \neg P(x,xp), P(y,yp), S(xp,yp) \]
\[ C(x,y) := P(x,xp), P(y,yp), C(xp,yp) \]
\[ R(x,y) := \neg S(x,y) \]
\[ R(x,y) := \neg R(x,z), P(y,z) \]
\[ R(x,y) := \neg R(z,y), P(x,z) \]

- Draw a dependency graph to identify cycles
- 2 cycles: R and C; these predicates are recursive, S and P are not
- P must be non-recursive, why?
Dependency graphs

\[ P(x,y) : \neg G(x,y) \]
\[ P(x,y) : \neg G(x,z), P(z,y) \]
\[ P(x) : \neg R(x), Q(x) \]
\[ Q(x) : \neg R(x), P(x) \]

P and Q are mutually recursive

SQL only supports linear recursion: no rule has more than 1 term subgoal that is mutually recursive with the head. (That includes self-loops.)

The reason is efficiency, not semantics, i.e., monotonicity is not a problem.
Dependency graphs

SQL only supports linear recursion: no rule has more than 1 term subgoal that is mutually recursive with the head. (That includes self-loops.)

\[
\begin{align*}
R(x,y) :&= F(\_, x, y, \_, \_, \_)
R(x,y) :&= \neg R(x,z), R(z,y)
Q(x,y) :&= \neg R('SFO', y)
\end{align*}
\]

\[
\begin{align*}
R(x,y) :&= F(\_, x, y, \_, \_, \_)
R(x,y) :&= \neg R(x,z), F(z,y)
Q(x,y) :&= \neg R('SFO', y)
\end{align*}
\]
Recall: Naive evaluation

\[ P(x, y) : -G(x, y) \]
\[ P(x, y) : -G(x, z), P(z, y) \]

\[ I : \{ G(0,1), \ G(1,2), \ G(2,3) \} \]

**Algorithm Naive**: Given a set of rules and an *edb* \( I \)

1. Initialize all *idb* relations as empty
2. Instantiate (with constants) variables of all rules in all possible ways. If all subgoals become true - we are inferring new fact(s) for the head relation
3. Repeat (2) in rounds, as long as new *idb* facts are being inferred

3 rounds to convergence

\[ \text{init} : P(x, y) = \emptyset \]
\[ \text{round 1:} P(0, 1) \ P(1, 2) \ P(2, 3) \]
\[ \text{round 2:} P(0, 1) \ P(1, 2) \ P(2, 3) P(0, 2) \ P(1, 3) \]
\[ \text{round 3:} P(0, 1) \ P(1, 2) \ P(2, 3) P(0, 2) \ P(1, 3) \ P(0, 3) \]

Very inefficient: all *idb* facts are re-derived on each round!
Datalog evaluation

- **Naive**: bottom-up, derives new facts from facts that hold during the previous rounds
- **Semi-naive**: bottom-up, more efficient than naive, attempts to avoid re-computation from round to round
- **Query-subquery (QSQ)**: top-down, related to the proof-theoretic semantics of datalog, which we did not discuss
- **Magic set**: bottom-up
Semi-naive evaluation

- Intuition: if a fact was derived for the first time at round $i$, it must depend on some fact or facts derived at round $i-1$
- For each $idb$ predicate over relation $R$, keep both $R$ and $delta-R$
- Deltas represent new facts inferred on the most recent round

**Algorithm Semi-naive**: Given a set of rules and an $edb$ I

1. Initialize (a) $idb$ relations using only rules without $idb$ subgoals; and (b) $delta-idb$ relations to be equal to the corresponding $idb$ relations
2. For each $idb$ relation $R$, and for each rule with $R$ in the head, compute $delta-R$ with one subgoal treated as a delta relation, others as regular relations (do for all possible choices of the delta-subgoal)
3. Remove from new $delta-R$ all facts that are already in $R$
4. Compute $R :- R U delta-R$
5. Repeat (2-4) in rounds, as long as new $idb$ facts are being inferred
Semi-naive: example

\[ P(x, y) : -G(x, y) \]
\[ P(x, y) : -G(x, z), P(z, y) \]

\[ P = G \quad \Delta P = G \]

Iterate until \( \Delta P = \emptyset \)
\[ \Delta P = \pi_{1,3}(G \triangleright \triangleleft \Delta P) \]
\[ \Delta P = \Delta P - P \]
\[ P = \Delta P \cup P \]

<table>
<thead>
<tr>
<th>round</th>
<th>G</th>
<th>P</th>
<th>delta-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>(0,1) (1,2) (2,3)</td>
<td>(0,1) (1,2) (2,3)</td>
<td>(0,1) (1,2) (2,3)</td>
</tr>
<tr>
<td>1</td>
<td>(0,1) (1,2) (2,3)</td>
<td>(0,1) (1,2) (2,3) (0,2) (1,3)</td>
<td>(0,2) (1,3)</td>
</tr>
<tr>
<td>2</td>
<td>(0,1) (1,2) (2,3)</td>
<td>(0,1) (1,2) (2,3) (0,2) (1,3) (0,3)</td>
<td>(0,3)</td>
</tr>
<tr>
<td>3</td>
<td>(0,1) (1,2) (2,3)</td>
<td>(0,1) (1,2) (2,3) (0,2) (1,3) (0,3)</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
Semi-naive: another example

\[ R(x,y) : -S(x,y) \]
\[ R(x,y) : -T(x,y,\_ ) \]
\[ R(x,y) : -R(x,z), R(z,y) \]

Init : \[ R = S \cup \pi_{1,2}T \]
\[ \Delta R = S \cup \pi_{1,2}T \]

Iterate until \( \Delta R = \emptyset \)
\[ \Delta R = \pi_{1,3}((\Delta R \bowtie R) \cup (R \bowtie \Delta R)) \]
\[ \Delta R = \Delta R - R \]
\[ R = \Delta R \cup R \]

<table>
<thead>
<tr>
<th>round</th>
<th>S</th>
<th>T</th>
<th>R</th>
<th>delta-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
<td>(0,1)</td>
<td>(0,1)</td>
</tr>
<tr>
<td></td>
<td>(1,2)</td>
<td>(2,5,3)</td>
<td>(1,2)</td>
<td>(1,2)</td>
</tr>
<tr>
<td></td>
<td>(2,5)</td>
<td></td>
<td>(2,5)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
<td>(0,1)</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>(1,2)</td>
<td>(2,5,3)</td>
<td>(1,2)</td>
<td>(1,5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(2,5)</td>
<td>(1,5)</td>
</tr>
<tr>
<td>2</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
<td>(0,1)</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>(1,2)</td>
<td>(2,5,3)</td>
<td>(1,2)</td>
<td>(1,5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(2,5)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(0,1)</td>
<td>(0,1,3)</td>
<td>(0,1)</td>
<td>(0,2)</td>
</tr>
<tr>
<td></td>
<td>(1,2)</td>
<td>(2,5,3)</td>
<td>(1,2)</td>
<td>(1,5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(2,5)</td>
<td></td>
</tr>
</tbody>
</table>
Datalog: why do we care?

• A beautiful set of problems / solutions
• Support for (a limited form) of recursion in SQL, triggers - convergence questions
• More recently: a true revival

- Declarative networking [e.g. Lou et al]
- Data integration and exchange [e.g. Clio, Orchestra]
- Program verification [e.g. Semmle]
- Data extraction from the Web [e.g. Gottlob, Lixto]
- Knowledge representation [e.g. Gottlob]
- Artifact and workflows [e.g. ActiveXML]
- Web data management [e.g. Webdamlog]

LogicBlox - will see next!
LogicBlox

LogiQL

LogiQL combines the best ideas from database query languages and (functional) programming languages. LogiQL is designed to be expressive and practical, with only a few simple language constructs.

LogiQL Playground

https://developer.logicblox.com/playground/

LogicBlox 4 Reference Manual

https://developer.logicblox.com/content/docs4/core-reference/webhelp
LogiQL

- A dialect of datalog
- Relations are in 6NF- at most one non-key attribute per relation
- Predicates of the form $R(x_1, .., x_n)$ or $R[x_1, ..., x_{n-1}] = x_n$
  - both can be base predicates (extensional, EDB) or derived predicates (intensional, IDB)
  - more on the distinction between $R(x_1, .., x_n)$ and $R[x_1, ..., x_{n-1}] = x_n$
- Derivation rules
  - specify how facts are calculated
  - support for arithmetic operations and aggregation
- Integrity constraints
  - specify types of attributes, types are declared (better!) or inferred
  - functional dependencies
  - inclusion constraints
LogiQL

basic predicate declaration: key attributes only
  person(x) -> string(x).

functional predicate declaration: one non-key attribute (cal)
  food[name, cat] = cal -> string(name), string(cat), int(cal).

functional dependencies

  friend_of(x,y) -> string(x), string(y).
  friend_of(x,y) -> person(x), person(y).

  person_age[x] = n -> string(x), int(n).
  person_age(x,_) -> person(x).
LogiQL

derived predicates

friend_of_friend(x,y) -> string(x), string(y).
friend_of_friend(x,y) -> person(x), person(y).
friend_of_friend(x,y) <- friend_of(x,p), friend_of(p,y), x!= y.

<- in LogiQL means the same as :- in datalog

arithmetic operations

older_friend[x,y] = d -> string(x), string(y), int(d).
older_friend(x,y,_) -> person(x), person(y).
older_friend(x,y,d) <- friend_of(x, y), person_age(x,n), person_age(y,m),
    n > m, d = n - m.

aggregation

number_of_friends(x,n) -> string(x), int(n).
number_of_friends(x,_) -> person(x).
number_of_friends[x] = n <- agg << n = count() >> friend_of(x,)._
adding tuples to an EDB

person(x) -> string(x).
+person("Ann").
+person("Bob").
+person("Cory").
+person("Dave").
+person("Emma").

friend_of(x,y) -> string(x), string(y).
friend_of(x,y) -> person(x), person(y).
+friend_of("Ann","Bob").
+friend_of("Ann","Cory").
+friend_of("Cory","Dave").
+friend_of("Dave","Emma").

person_age[x] = n -> string(x), int(n).
person_age(x,_) -> person(x).
+person_age("Ann",30).
+person_age("Bob",25).
+person_age("Cory",28).
+person_age("Dave",22).
+person_age("Emma",18).
executing commands

addblock 'person(x) -> string(x).'

addblock 'friend_of(x,y) -> string(x), string(y).
friend_of(x,y) -> person(x), person(y).'

exec '+person("Ann").
+person("Bob").
+person("Cory").
+person("Dave").
+person("Emma").'}