Datalog extensions

Simone de Oliveira Santos

February 19, 2014

SociaLite: Datalog extensions for efficient Social network analysis

Motivations

- Rize of a large number of online social networks;
- Analysis are often built on top of fundamental graph algorithms;
- Need for a better computation model or query language to achieve the goal of letting consumers express queries on their personal social graphs;
- Datalog is a candidate for achieve this vision because of its high-level declarative semantics and support for recursion;

SociaLite: Datalog extensions for efficient Social network analysis

Motivations

- However, the relational representation in Datalog is not a good match for graph analysis.
- The performance of Datalog is not competitive when compared with other languages (such as Java) for social network analysis;
- General-purpose languages are more difficult to write analysis programs.

Goal: an extension of Datalog that delivers performance similar to that of highly optimized Java programs.

Contributions

SociaLite allows concise expression of graph algorithms, giving some degree of control over data layout and evaluation order.

- Tail-nested tables (new data representation)
- Recursive aggregate functions
- User-guide execution order
- Evaluation of SociaLite

Tail-Nested Tables

Data as indices

Number data items sequentially and use the number as an index into an array, i.e., *src:0..1000*.

New data representantion to declare relations

EDGE(int src:0..10000, (int sink, int len))[a].
PATH(int sink:0..10000, int dist).

[a] tail-nested table

Tail-Nested Tables

EDGE(int src:0..10000, (int sink, int len))

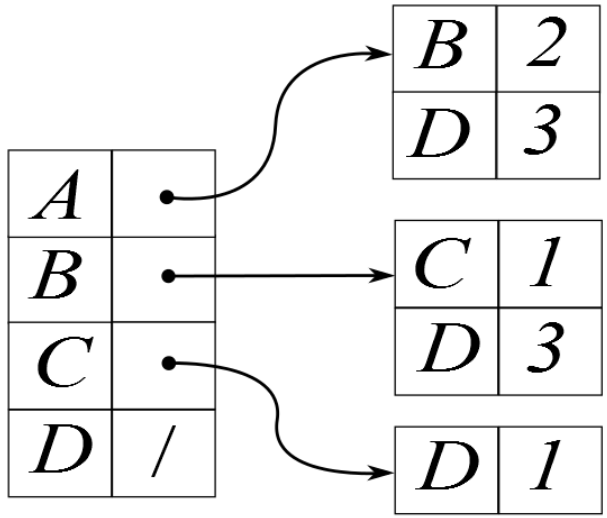


Figure : Tail-nested table

Tail-nested table is a generalization of the adjacency list. The last column of a table may contain pointers to two-dimensional tables.

Recursive Aggregate Functions

- The extension must have the capability to eliminate unnecessary tuples.

The shortest-paths algorithm

$$
\begin{aligned}
PATH(t, d) &: -EDGE(1, t, d). \\
&: -PATH(s, d_1), EDGE(s, t, d_2), \\
&\quad d = d_1 + d_2. \\
MINPATH(t, \$MIN(d)) &: -PATH(t, d).
\end{aligned}
$$

- Do not stop in the presence of cycles.
- Slow due to unnecessary computation of sub-optimal distances.

Recursive Aggregate Functions

- Aggregate functions are expressed as an argument in a head predicate;
- They are recursive if the head predicate is used as a body predicate as well.

The shortest-paths algorithm

EDGE(int *src:0..10000*, (int *sink*, int *len*)).
PATH(int *sink:0..10000*, int *dist*).
PATH(t, $\underline{\$MIN(d)}$) :- EDGE(1, t, d);
:- PATH(s,d_1), EDGE(s, t, d_2),
$d = d_1 + d_2$

Recursive Aggregate Functions

- Recursive aggregate function improves the ability to express graph algorithms;
- Without them the program generates all the possible paths before finding the minimum, and does not terminate in the presence of cycles;
- With them the program specifies that only the minimum paths are of interest, thus eliminating cycles.

The operational semantics of SociaLite

- All rules are repeatedly evaluated until convergence is achieved;
- Only one argument of a head predicate can be an aggregate function;
- The others are the qualifying parameters for the aggregate function.

Greatest Fixed-Point semantics

An operation is a **meet operation** if it is *idempotent*, *commutative* and *associative*;
A meet operation defines a semi-lattice; it induces a *partial order* $\sqsubseteq$ over a domain, such that the result of the operation for any two elements is the greatest lower bound of the elements with respect to $\sqsubseteq$.

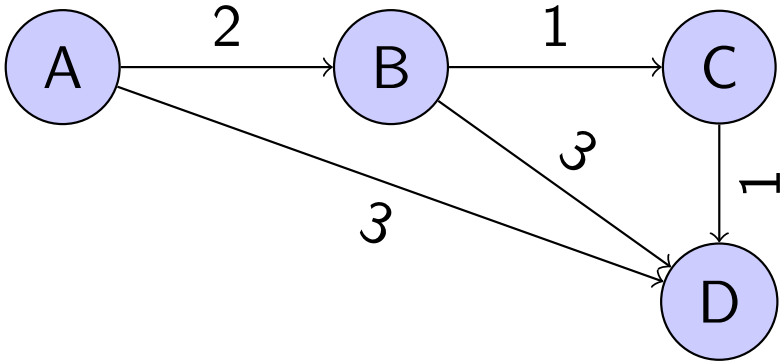
1. $R^* = h(R^*) = (g \circ f)(R^*)$
2. $R \sqsubseteq R^*$ for all R such that $R = h(R)$

Semi-Naive Evaluation

- It avoids redundant computation by joining only subgoals in the body of each rule with at least one new answer produced in the previous iteration.
- It can be extended to recursive aggregate functions if they are meet operations.

Input: A SociaLite program $h = g \circ f$, where $g : R \rightarrow R$ applies an aggregation function to each set of qualifying parameters and $f : R \rightarrow R$ represent the rest of the rules.

Examples



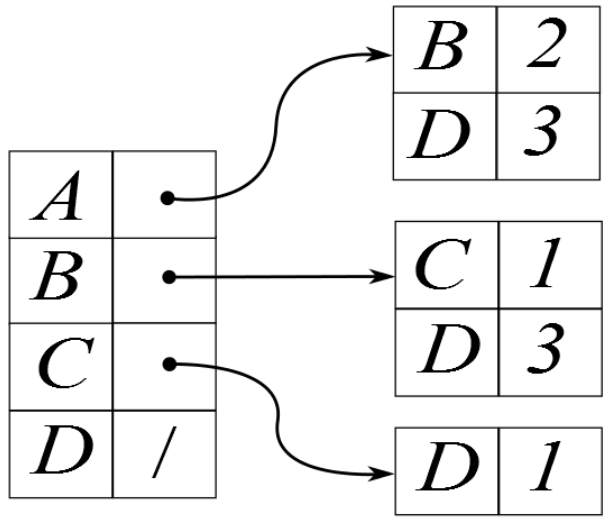


Figure : Tail-nested table

Examples

Method:

$R_0 \leftarrow \emptyset, \Delta_0 \leftarrow \emptyset$

$i \leftarrow 0$

repeat

$i \leftarrow 1 + 1$

$R_i \leftarrow g(f(\Delta_{i-1}) \cup R_{i-1})$

$\Delta_i \leftarrow R_i - R_{i-1}$

until $\Delta_i \neq \emptyset$

$$f(R) = \{< t, d > | \\ EDGE(1, t, d) \vee \\ (< s, d_1 > \in R \wedge EDGE(s, t, d_2) \wedge \\ d = d_1 + d_2)\}$$

$$g(R) = \{< t, min_{<t,d_1> \in R} d_1 > | \\ < t, d_1 > \in R\}$$

Examples

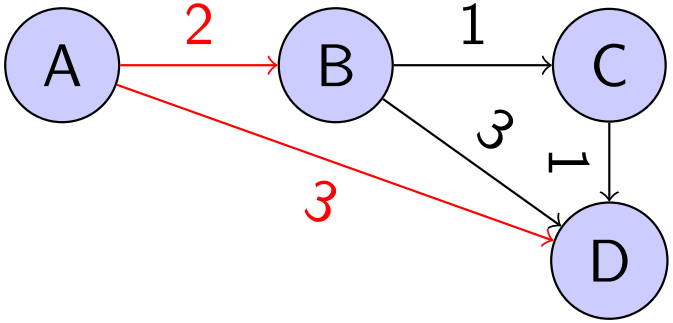
Iteration 1:

Application of f in Δ_0

.

$f(R) = \{< t, d >|$
$EDGE(1, t, d) \vee$
$(< s, d_1 > \in R \wedge EDGE(s, t, d_2) \wedge$
$d = d_1 + d_2)\}$

.



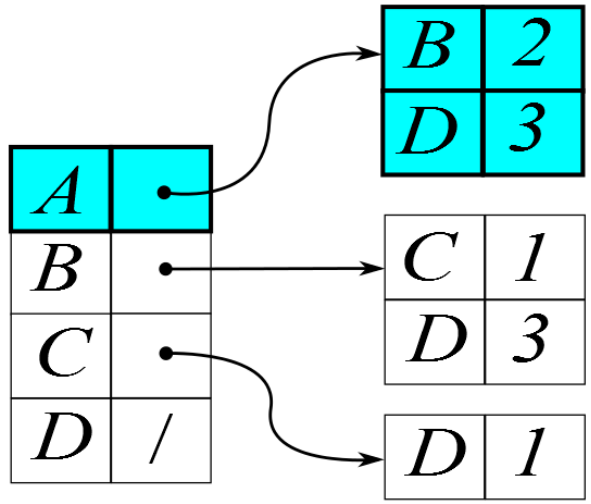


Figure : Tail-nested table

{PATH(B,2), PATH(D,3)}

Examples

Application of G in $\{PATH(B,2), PATH(D,3)\}$.

$g(R) = \{< t, min_{<t,d_1>\in R} d_1 > | < t, d_1 > \in R\}$

Result:

$R_1 = \{PATH(B,2), PATH(D,3)\}$

Application of Δ

$\Delta_i \leftarrow R_i - R_{i-1}$

$R_0 = \emptyset$

$R_1 = \{PATH(B,2), PATH(D,3)\}$

Result:

$\Delta_1 = \{PATH(B,2), PATH(D,3)\}$

Examples

Iteration 2:

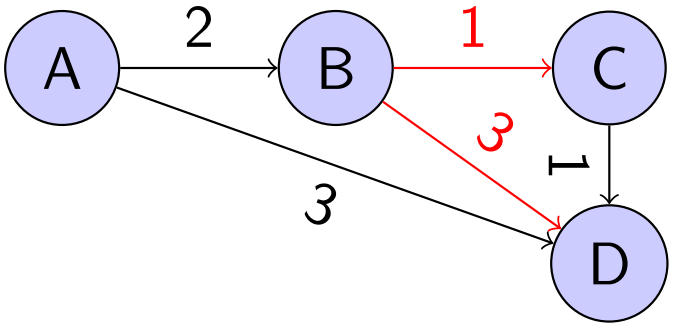
Application of f in Δ_1

$\Delta_1 = \{PATH(B, 2), PATH(D, 3)\}$

.

$f(R) = \{< t, d > |$
$EDGE(1, t, d) \vee$
$(< s, d_1 > \in R \wedge EDGE(s, t, d_2) \wedge$
$d = d_1 + d_2)\}$

.



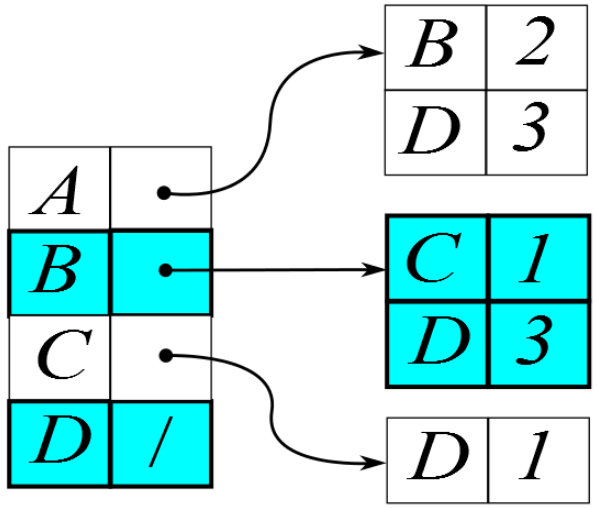


Figure : Tail-nested table

{PATH(B,2), PATH(D,3),
PATH(C,3), PATH(D,5)}

Examples

Application of G in
$\{PATH(B,2), PATH(D,3),$
$PATH(C,3), PATH(D,5)\}$.

$g(R) = \{< t, min_{<t,d_1>\in R} d_1 > |$
$< t, d_1 > \in R\}$
Result:
$R_2 = \{PATH(B,2), PATH(D,3),$
$PATH(C,3)\}$

Application of Δ_2
$\Delta_2 \leftarrow R_2 - R_1$
$R_1 = \{PATH(B,2), PATH(D,3)\}$
$R_2 = \{PATH(B,2), PATH(D,3),$
$PATH(C,3)\}$
Result:
$\Delta_2 = \{PATH(C,3)\}$

Examples

Iteration 3:

Application of f in Δ_2

$\Delta_2 = \{PATH(C, 3)\}$

.

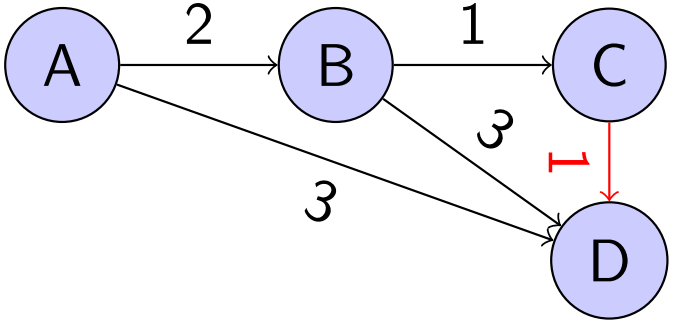
$f(R) = \{< t, d > |$

$EDGE(1, t, d) \vee$

$(< s, d_1 > \in R \wedge EDGE(s, t, d_2) \wedge$

$d = d_1 + d_2)\}$

.



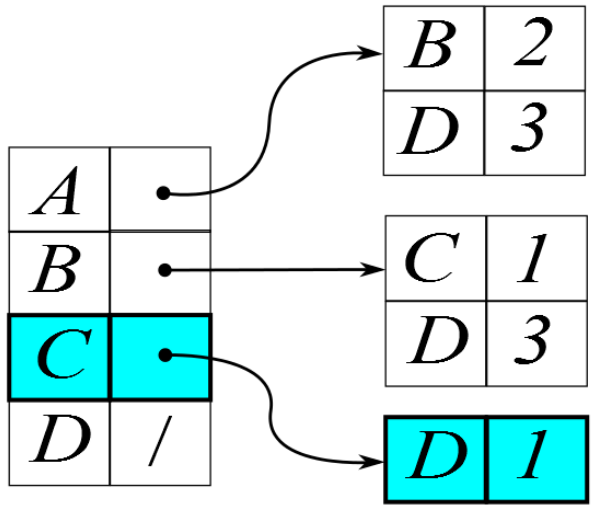


Figure : Tail-nested table

{PATH(B,2),PATH(D,3),PATH(D,4)}

Examples

Application of G in
$\{PATH(B,2), PATH(D,3),$
$PATH(C,3), PATH(D,4)\}$.

$g(R) = \{< t, min_{<t,d_1>\in R} d_1 > |$
$< t, d_1 > \in R\}$
Result:
$R_3 = \{PATH(B,2), PATH(D,3),$
$PATH(C,3)\}$

Application of Δ_3
$\Delta_3 \leftarrow R_3 - R_2$
$R_2 = \{PATH(B,2), PATH(D,3),$
$PATH(C,3)\}$
$R_3 = \{PATH(B,2), PATH(D,3),$
$PATH(C,3)\}$
Result:
$\Delta_3 = \emptyset$
Algorithm stops
Return R_3

Ordering

SociaLite lets users control the order in which the graphs are traversed. The programmer can declare that a column in a table is to be sorted (similar to SQL).

$$R(< type > f_1, < type > f_{2,\ldots}) \text{ orderby[asc|desc],}\ldots$$

When a sorted column is included in a rule, it indicates to the compiler that the rows are to be evaluated in the order specified.

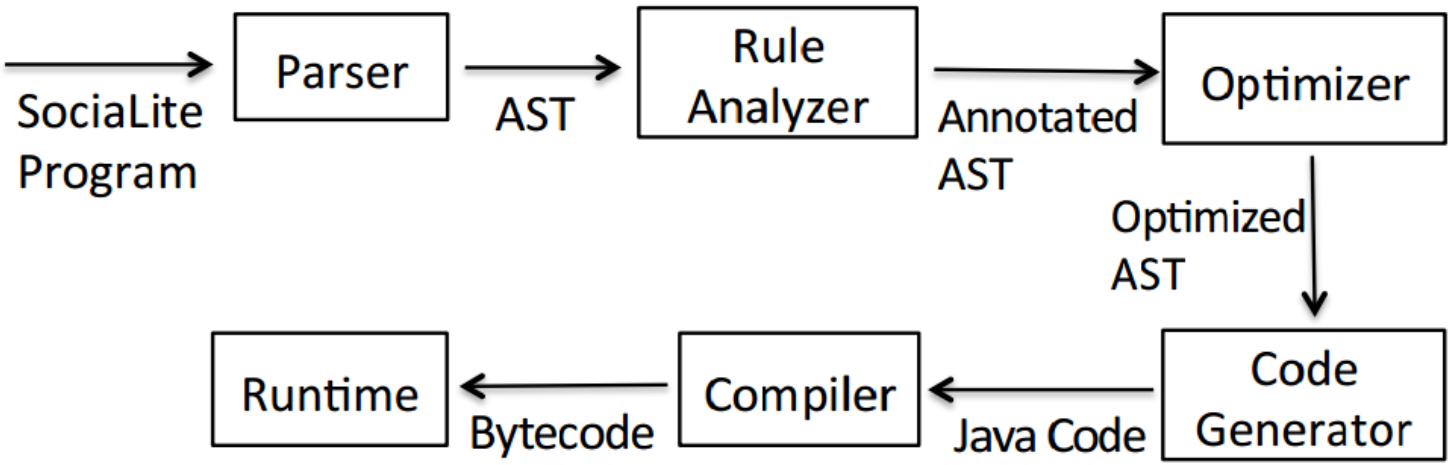
User-Specified Funtions

Users can supply natively implemented functions and use them in SociaLite rules. A Java function F with n arguments can be invoked in SociaLite with $\$F(a_1, \ldots, a_n)$, which can return one or more results.

SociaLite has pre-defined aggregate functions such as $SUM, $MIN, and $MAX.

The SociaLite compiler accepts a SociaLite program, together with additional Java functions, and translates it into Java source code.

System overview



Extension of SociaLite

Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis

With distributed SociaLite,
programmers simply annotate how data are to be distributed, then the necessary communication is automatically inferred to generate parallel code for cluster of multi-core machines.

Extension of SociaLite

(a) Datalog:

$$\textsc{Path}(t, d) \quad :- \quad t = 1, d = 0.0.$$
$$\textsc{Path}(t, d) \quad :- \quad \textsc{Path}(s, d_1), \textsc{Edge}(s, t, d_2), d{=}d_1{+}d_2.$$
$$\textsc{MinPath}(t, d) \quad :- \quad \textsc{Path}(t, \$\textsc{Min}(d)).$$

(b) Sequential SociaLite:

$\textsc{Edge}$ (int s:0..9999, (int t, double $dist$)).
$\textsc{Path}$ (int t:0..9999, double $dist$).

$$\textsc{Path}(t, \$\textsc{Min}(d)) :- \quad t = 1, d = 0.0;$$
$$:- \quad \textsc{Path}(s, d_1), \textsc{Edge}(s, t, d_2), d{=}d_1{+}d_2.$$

(c) Parallel SociaLite:

$\textsc{Edge}$ [int s:0..9999] ((int t, double $dist$)).
$\textsc{Path}$ [int t:0..9999] (double $dist$).

$$\textsc{Path}[t](\$\textsc{Min}(d)) :- \quad t = 1, d = 0.0;$$
$$:- \quad \textsc{Path}[s](d_1), \textsc{Edge}[s](t, d_2), d{=}d_1{+}d_2.$$

Figure : Comparison Datalog extension

The WebdamLog System

Motivations

- The need for a unified approach for the management of distributed content;
- Personal information management;
- A declarative language to facilitate the adoption of our platform by non-technical users.

Goal: A declarative high-level language in the style of Datalog, to support the distribution of both data and knowledge over a network of heterogeneous peers.

Overview

WebdamLog is a datalog-style language that emphasizes cooperation between distributed autonomous peers communication in an asynchronous manner, and supports

- updates
- distribution
- negation
- delegation

Example

Making a photo album of Alice and Bob using photos of closer friends.

Tasks:

1. identify friends using social networks;
2. find out where they keep their photos and how access them;
3. select photos with both Alice and Bob;
4. ask a friend to verify if the photos are appropriate;
5. exclude someone, if necessary, from the set of source.

Example

Making a photo album of Alice and Bob using photos of closer friends.
Tasks:

1. identify friends using social networks;
2.
3.
4.
5.

[rule at sue]

```
source@sue($name) :- friends@aliceFB($name)
source@sue($name) :- friends@bobFB($name)
```

This rule computes the union of Alice's and Bob's Facebook contacts in relation *source* on Sue's peer.

Example

Making a photo album of Alice and Bob using photos of closer friends.
Tasks:

1.
2. find out where they keep their photos and how access them;
3. select photos with both Alice and Bob;
4.
5.

[rule at sue]

```
album@sue($photo,$name) :- source@sue($name),
                           photoLocation@$name($peer), photos@$peer($photo),
                           features@$peer($photo,alice), features@$peer($photo,bob)
```

This rule delegates steps (2) and (3) to the peers corresponding to the friends of Alice and Bob.

Example

Making a photo album of Alice and Bob using photos of closer friends.
Tasks:

1.
2.
3.
4. ask a friend to verify if the photos are appropriate;
5.

[rule at dan]

```
album@sue($photo,dan) :- photoLocation@dan($peer),
                         photos@$peer($photo), features@$peer($photo,alice)
                         features@$peer($photo,bob)
```

Dan is a friend, then Sue's peer will delegate this rule to Dan's peer.

Example

Making a photo album of Alice and Bob using photos of closer friends.
Tasks:

1.
2.
3.
4.
5. exclude someone, if necessary, from the set of source.

[rule at sue]

```
source@sue($name) :- friends@aliceFB($name), not blocked@sue($name)
source@sue($name) :- friends@bobFB($name), not blocked@sue($name)
```

By inserting/removing facts in *blocked@sue*, Sue controls who can participate.

Webdamlog Model

Assuming the existence of a countable set of data values (set of relation names and set of peer names) and a countable set of variables.

Relation is an expression $m@p$ where:

- m is a relation name
- p is a peer name

Schema is an expression (π, E, I, σ) where:

- π is a possibly infinite set of peer names
- E is a set of extensional relations of the form $m@p$ for $p \in \pi$
- I is a set of intensional relations of the form $m@p$ for $p \in \pi$
- σ, the sorting function, specifies for each relation $m@p$, an integer $\sigma(m@p)$ that is its sort (arity).

Webdamlog Model

Fact is an expression $m@p(a_1 \ldots a_n)$ where:

- $n = \sigma(m@p)$
- $a_1 \ldots a_n$ are data values

An example of fact is: $picture@myalbum(1771.jpg, "Paris", 11/11/2011)$

Rule is an expression of the form:

[at p] $\$R@\$P(\$U) :- (\neg)\$R_1@\$P_1(\$U_1), \ldots, (\neg)\$R_n@\$P_n(\$U_n)$, where:

- $\$R, \$R_i$ are relation terms
- $\$P, \$P_i$ are peer terms
- $\$U, \$U_i$ are tuples of terms

Webdamlog Model

At a particular point in time a peer *p* has:

- a *state* consisting of some facts
- some rules specified locally
- possibly some rules that have been delegated by other peer

Peers evolve by:

- updating their base of facts
- sending facts to other peers
- updating their delegations to other peers

Webdamlog Model

- A rule is *deductive* if the head relation is intensional. Otherwise, it is *active*.
- A rule in a peer p is *local* if all P_i in all body relations are from p.
- A rule is *fully local* if the head relation is also from p.

A local computation happens at a particular peer. The peer performs the following:

- Some local deduction of intensional facts
- The derivation of extensional facts that either define its next state or are sent as messages
- The delegation of rules to other peers.

Webdamlog Model

Classes or rules:

- **Local deduction** Fully local deductive rules are used to derive intensional fats locally.
- **Update** Local active rules are used for sending messages, facts, that modify the databases of the others peers that receive them.
- **View delegation** Provide some form of view materialization. For instance, a rule result in providing at q a view for some data from p.
- **General delegation** The remaining rules allow a peer to install arbitrary rule at other peers.

Webdamlog Model

Convergence of WebdamLog

- Positive WebdamLog
- Strongly-stratified Webdamlog

The WebdamLog system

Extension of a existing datalog engine.

- **IRIS**, implemented in Java and supports the main strategies for efficient evaluation of standard local datalog
- **Bud**, implemented in Ruby scripting language, and provide mechanisms for asynchronous communication between peers. (the chosen)