Dynamically Distributed Query Evaluation

Trevor Jim
AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932
trevor@research.att.com

Dan Suciu
Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195–2350
suciu@cs.washington.edu

1. INTRODUCTION

Distributed query evaluation usually assumes a fixed topology, where the set of servers and the partitioning of data on the servers is known in advance. Given a query expression, an optimizer will first produce a global plan, then assign a server for each operator in the plan. The query is then evaluated distributively, according to that plan.

However, in the Web, topology is discovered dynamically: by following a link, we discover new links that were previously unknown. It is impossible to construct a global plan ahead of time, or to pre-assign sites to operators. The Web requires a reexamination of the distributed query evaluation paradigm.

In this paper, we propose a new paradigm for distributed query evaluation on the Web. We present the paradigm in terms of a simple query language, called *dynamically distributed datalog (d3log)*, which extends datalog by a single feature, *dynamic site discovery*. The following example gives the intuition:

$$R(x, y) :\!-\ E(x, w, z),\ w\$R(z, y)$$

This rule says that $R(x, y)$ holds whenever: (1) $E(x, w, z)$ holds; and (2) $R(z, y)$ holds *at site* w. That is, any site s can define its own, distinct relation R, and we write $s\$R(z, y)$ if $R(z, y)$ holds at site s. In our rule, the site w is drawn from the local relation E; using our Web analogy, w is like a link that we discover from E. Navigation (discovery and traversal of several links) is easy to express in d3log.

The simple addition of dynamic site discovery changes the character of query evaluation, and motivates novel evaluation techniques. The main paradigm change is the introduction of an *intensional answer*, in which a server responds to a query not with a table, but rather with a set of rules. To illustrate, consider the example above and assume that the server receives the query $R(1, y)$. Examining its local table E, it finds the tuples $E(1, s_2, 2)$ and $E(1, s_3, 3)$, i.e., it "discovers" that the sites s_2 and s_3 hold relevant answers. At this point, rather than contacting those sites directly, it may decide to return an answer instructing the client to contact those sites itself:

$$R(1, y) :\!-\ s_2\$R(2, y) \qquad R(1, y) :\!-\ s_3\$R(3, y)$$

This is an intensional answer, consisting of a set of rules rather than a set of tuples. Another possibility is for the server to contact site s_2 directly and return that answer, while instructing the client to contact site s_3 itself. Or, the server could contact both sites and return a traditional, extensional answer to the client.

Intensional answers offer several advantages for distributed query evaluation on the Web. First, they provide flexibility in coping with unavailable servers. If one server relies on another server that is down, it can return to its client a rule instructing the client to contact the second server: the client may decide to contact the second server at a later time. Second, they allow a server to offer a range of qualities of service: for high-paying customers the server might evaluate the query in full, including subqueries to remote sites, and return a set of tuples; for others it might perform only partial query evaluation, and return a combination of tuples and rules. Finally, intensional answers may improve overall bandwidth utilization, since they can be smaller than extensional answers.

In this paper we develop a framework for studying possible query evaluation techniques in the presence of dynamic site discovery. We present a query evaluation paradigm in which pairs of servers communicate by exchanging messages: a query is sent, and an intensional answer is returned. A server has many alternatives for constructing the intensional answer for a given query: we give a precise definition of what constitutes a correct answer. We explore the entire space of alternatives a server has in constructing a correct answer, by describing a generic, nondeterministic evaluation algorithm, whose possible runs return all answers in this space. We prove that all answers returned by the algorithm are correct. Then we show that particular deterministic algorithms, implementing specific optimizations, correspond to particular runs of the generic algorithms, hence they are correct too.

We started by studying query evaluation with site discovery for a non-recursive query language, similar to SQL. We soon discovered however that query evaluation in this new setting is deeply and intimately connected to recursion, for three reasons. First, recursion may be impossible to prohibit: the Web is not under centralized control, and one cannot constrain what links other sites may have. Second, it is difficult to detect: sites are unknown in advance, so recursion must be detected dynamically. Finally, we have found some applications, such as security infrastructures, where recursion arises naturally and inevitably. For example, public key directories are often built by delegation to a trusted party. A public key directory $s_1\$PKD$ may be defined by rules giving some locally known public keys, and a rule saying that it includes (trusts) all public keys from a directory $s_2\$PKD$. In turn, s_2 may store locally some public keys it knows, and have a rule saying that it trusts all keys from s_1; this is an example of mutual trust. Notice that the recursion here occurs at the global level, not at one particular server. We chose the language d3log in order to be able to model such global recursive behaviors.

Intensional answers have been considered before [12, 19], in the context of centralized, but not distributed, databases. The main motivation was that an intensional answer is often more informative than an extensional one. For example, given the query "find all empoyees whose salary is < 100000," the answer "all employees" is more informative than a long enumeration of tuples. The challenge there is to find the most concise intensional answer. In our setting, intensional answers are used in the context of distributed data, and the criteria for what is the "best" answer may depend on the distribution. For example, a natural place to stop evaluation and return an intensional answer is at references between sites, even though this may not give the most concise answer. Thus, we want to permit servers to return a range of answers, all the way from completely extensional (full evaluation) to completely intensional (return the local program).

The rest of the paper is organized as follows. In Section 2, we define d3log and its semantics, and we give some examples in Section 3. In Section 4, we introduce our distributed evaluation paradigm, and in Sections 5 and 6, we describe a variety of evaluation algorithms. In Section 7, we show how standard optimization techniques can be modified to handle dynamic site discovery. We discuss related work in Section 8, and we conclude in Section 9.

2. D3log

Syntax

We assume an infinite domain of constants, D, and a fixed collection of relation symbols, $R_1, \ldots, R_m$. We use w, x, y, z for variables, s for constants, and let t range over variables and constants. An atom a has the form $t\$R(t_1, \ldots, t_n)$: the intuition is that $R(t_1, \ldots, t_n)$ "holds" at site t. We say t is the site of $t\$R(t_1, \ldots, t_n)$, and an atom a is *site determined* if its site is a constant. A rule has the form

$$a \text{ :- } a_1, \ldots, a_n \qquad (1)$$

where $n \geq 0$ and a is site determined. Following standard terminology, a is called the *head*, $a_1, \ldots, a_n$ the *body*, and when $n = 0$ the rule is called a *fact*. We use f to range over facts, and by convention, we treat a fact, a :–, and its head, a, interchangeably. A *program*, $\mathcal{P}$, is a finite set of rules. We depart from datalog and blur the distinction between data and program, letting $\mathcal{P}$ hold both the facts (data) and the rules (program); we don't distinguish between edbs and idbs. We denote by $\mathcal{P}_s$ the rules *at site* s, i.e., those rules in $\mathcal{P}$ whose heads have site s; since $\mathcal{P}$ is finite, only finitely many sites hold rules. A *query* is a site-determined atom, q. We denote by $\mathit{inst}(q)$ the set of facts that are instantiations of q. For example, if $q = s_3\$R(x, 5)$, then $\mathit{inst}(q) = \{\ s_3\$R(1, 5), s_3\$R(2, 5), \ldots\ \}$.

Models, Semantics

There is a canonical translation of a d3log program $\mathcal{P}$ into a datalog program $\mathcal{P}^g$ called the *global datalog program*: each n-ary relation name R is translated into an $(n + 1)$-ary relation name R^g, and each atom $t\$R(t_1, \ldots, t_n)$ into $R^g(t, t_1, \ldots, t_n)$. We define a *model* of $\mathcal{P}$ to be a model of (the datalog program) $\mathcal{P}^g$; in this paper we equate a model with a set of facts, $\mathcal{F}$, but use d3log syntax for facts in $\mathcal{F}$, writing, e.g., $s_3\$R(1, 5)$ instead of $R^g(s_3, 1, 5)$. As usual, the *semantics* of $\mathcal{P}$ is its minimal model, $\mathcal{F}$; alternatively, $\mathcal{F} = \{\ f \mid \mathcal{P} \models f\ \}$. The *(extensional) answer* to a query q is the set of facts in $\mathcal{F}$ that are instances of q (i.e., $\mathit{inst}(q) \cap \mathcal{F}$).

D3log does not add expressive power to datalog. Our interest in d3log lies in studying evaluation algorithms that take the distribution into account.

Site-safe rules

Recall that a datalog rule is *safe* if all variables in the head also appear in the body [23]: this ensures that the program's semantics is finite. D3log requires a stronger safety rule: the rule in Eq. (1) is *site safe* if it is safe and for every atom $a_i = x\$R_i(\ldots)$ that is not site determined, the variable x occurs in some earlier atom a_j (i.e., $a_j = t_j\$R_j(\ldots, x, \ldots)$ and $j < i$). In particular, the first atom, a_1, is always site determined. For instance, the rule $s\$Q(x, y)$:- $s\$R(x, z), z\$S(z, y)$ is site safe. Rules such as $s\$Q(x)$:- $x\$R(y)$, $y\$R(x)$ and $s\$Q(x)$:- $x\$R(x)$ are not site safe, although their corresponding global rules are safe.

Site safety is needed because of the restricted means by which a site can learn about the existence of other sites. For a datalog program, $\mathcal{P}$, the *active domain*, $\mathit{adom}(\mathcal{P})$, is the set of all constants occurring in $\mathcal{P}$. If $\mathcal{P}$ is safe, then its semantics is the same under the standard (infinite) domain D or the (finite) active domain. In d3log, however, no single site s can compute the entire active domain, but only that part "accessible" from s, denoted adom_s, and defined recursively as follows: adom_s consists of $\mathit{adom}(\mathcal{P}_s)$ plus all constants in adom_{s_1}, for each $s_1 \in \mathit{adom}_s$. Obviously, for every site s, $\mathit{adom}_s \subseteq \mathit{adom}(\mathcal{P})$, and the inclusion may be strict. We can prove that d3log enjoys a property similar to datalog: if $\mathcal{P}$ is site safe, then the answer to a query q at site s is the same under the standard domain, D, and under the accessible active domain at s, adom_s. Hence, in the rest of the paper we will assume site safe programs and a finite domain.

As a side comment, we note that our definition of site safety is a form of range restriction [1], and that it relies on a certain order in the body of Eq. (1), which is equivalent to a join order. The results in this paper are not affected by the particular choice of join order. Choosing an order is essentially a join-order optimization problem, and is beyond the scope of our paper.

3. EXAMPLES

We illustrate a few real applications that can be modeled in d3log.

Security infrastructures

SD3 [13] and its predecessor, QCM [9, 10, 11], are systems for building secure nameservers, public key directories, and distributed repositories of security policies. In SD3, cryptographic *principals* take the place of d3log sites: a typical principal is a pair $s = k@i$ consisting of a public key k and an Internet address i. Principals make it possible to communicate securely over untrusted networks like the Internet. For example, $Q(x) \text{ :- } s\$PKD(\text{Alice}, x)$ is a query that securely obtains Alice's key from a public key directory (PKD) by an exchange of signed messages with the principal s. The server for $s\$PKD$ controls the private key corresponding to the public key k, and signs all responses with that key; the client has the public key k and can therefore verify that the response has not been tampered with on the untrusted network. In this way, knowledge of a single key can be bootstrapped into knowledge of many keys (from $s\$PKD$), solving the public key distribution problem.

In SD3, a public key directory associates names with principals. For example, Alice can define a PKD with rules

$$\begin{array}{l}
s_{\text{Alice}}\$PKD(\text{Alice}, s_{\text{Alice}}) \text{ :-} \\
s_{\text{Alice}}\$PKD(\text{Bob}, s_{\text{Bob}}) \text{ :-} \\
s_{\text{Alice}}\$PKD(x, y) \text{ :- } s_{\text{Alice}}\$PKD(\text{Bob}, z),\ z\$PKD(x, y)
\end{array}$$

Alice's PKD includes her own key (principal), s_{Alice}, and her friend Bob's key, s_{Bob}. Furthermore, Alice trusts Bob and would like to be able to communicate securely with his friends, so she includes his PKD in her own. Of course, since Alice and Bob are friends, Bob may want to include Alice's PKD in his PKD; this kind of recursion is commonplace.

DNS

The Domain Name System, or DNS, is the distributed database that maps Internet host names to numerical IP addresses. This mapping is defined by many different *nameservers*, each responsible for part of the mapping. Nameservers are organized hierarchically, following the hierarchy of domain names. We show one way that the behavior of DNS can be modeled in d3log: for brevity the illustration here is a simplification, but we give a more accurate model (which we have implemented) in the Appendix.

Below, we show a fragment of the program for the nameserver at IP address `198.41.3.38`, which is in charge of the domain `.com`:[1]

$$\begin{array}{l}
\texttt{198.41.3.38}\$DNS(x, y) \text{ :- } \texttt{198.41.3.38}\$NS(x_2, y_2), \\
\qquad\qquad\qquad\qquad\qquad x_2 < x,\ x_2\$DNS(x, y) \\
\texttt{198.41.3.38}\$NS(\texttt{att.com}, \texttt{192.128.133.77}) \text{ :-} \\
\texttt{198.41.3.38}\$NS(\texttt{ibm.com}, \texttt{198.81.209.2}) \text{ :-} \\
\ldots
\end{array}$$

[1]We give the addresses of the actual nameservers, as of December 2000.

This server knows the IP addresses of all the nameservers for subdomains of `.com`, including `att.com` and `ibm.com`. These addresses are stored in a local table, NS. For example, the second rule says that `192.128.133.77` is the IP address of the nameserver in charge of the `att.com` subdomain. This is encoded in the first rule, which simply says that a request for the IP address y of a given name, x, can be answered by the nameserver in charge of x. For example, if the nameserver receives a query,

$$\texttt{198.41.3.38}\$DNS(\texttt{www.att.com}, y),$$

then it can return the following rule as an answer:

$$\begin{array}{l}
\texttt{198.41.3.38}\$DNS(\texttt{www.att.com}, y) \text{ :-} \\
\qquad\qquad \texttt{192.128.133.77}\$DNS(\texttt{www.att.com}, y)
\end{array}$$

The client can then contact the server for `att.com` (and that nameserver can supply the IP address directly). Alternately, the nameserver could query the `att.com` nameserver itself, and return the response. Both of these evaluation strategies are possible in DNS, where they are known respectively as iterative and recursive resolution.

LDAP directories

An LDAP directory is a lightweight database with a hierarchical data model. Data is organized as a tree, where the nodes are called *entries*. LDAP servers are usually interconnected: a leaf node in one server may contain the address of another server. Such a leaf node is called a *reference*, and the semantics is that the entire tree at the other server is inserted at that leaf node. This behavior is similar to DNS, where a nameserver has references to other nameservers, and it can be modeled in d3log similarly. As in DNS, queries on distributed LDAPs can be evaluated in two modes. In *referral* mode, when the server encounters a reference node, that node is simply returned to the client, who then has to contact the other server. In *chaining* mode, the server contacts the other server on behalf of the client.

XML

An XML document may contain references to other XML documents: such a reference is called an *XLink*. The reference can be a small query, in a language called XPath, that can specify arbitrary navigation in the target XML document. The meaning is, again, that the target subtree is to be inserted in place of the XLink expression. This behavior can also be expressed in d3log. When an XML query attempts to traverse an XLink, the server may either return the entire XPath expression to the client (referral mode), or send an appropriate query to the other site (chaining mode), or fetch the remote XML subtree and evaluate the query locally.

4. DYNAMICALLY DISTRIBUTED EVALUATION

The novelty in d3log lies in its distributed evaluation paradigm. Given a distributed program $\mathcal{P}$, evaluation is initiated when a user asks a query q at some server s. We call a server evaluating a query a *client*, hence s is a client. Recall that s has a local program, denoted $\mathcal{P}_s$. During evaluation, messages are exchanged between pairs of sites. Each message involves a *query* (a site-determined atom) or an *answer*. The answer may be an arbitrary d3log program: an *intensional* *answer*. This generalizes answers in standard distributed databases [14, 18], which may only contain data: we call those *extensional answers*. The client alternates between evaluation and communication. It starts with the program $\mathcal{P}_s$ and performs some evaluation. At some point it sends some query q_1 to a server s_1 and receives an intensional answer $\mathcal{A}_1$, that is added to $\mathcal{P}_s$. After more evaluation, it sends some other query q_2 to another server s_2, and receives an intensional answer $\mathcal{A}_2$, etc. Eventually, the client decides to halt and returns an answer $\mathcal{A}$ as response to the original query q. The servers $s_1, s_2, \ldots$ need not be distinct: the client may contact a server multiple times, e.g., by requesting different queries. Evaluation of a query q_i at a server s_i proceeds in similar fashion: s_i becomes the client, and proceeds in a series of evaluation and communication steps, eventually returning $\mathcal{A}_i$.

We assume that all messages are synchronous, and hence the evaluation is sequential. This is a limitation, but it is not essential. The techniques described in this paper extend to asynchronous messages and parallel evaluation.

Our interest lies in the dynamic nature of this evaluation paradigm. One dynamic aspect is that the set of sites is not known in advance: a server may learn about new sites as the evaluation progresses. Another aspect is that the answer to a query may be intensional, i.e., a program as opposed to data. As we have seen, applications like DNS, LDAP, and XML exhibit some form of intensional answers, and in addition, intentional answers are the technical means by which we evaluate recursive distributed programs.

Correct Answers

Since a server has the freedom to return a variety of answers in response to a query, one may ask what constitutes a "correct" answer.

As usual, correctness can be broken down into two components, soundness and completeness. Soundness simply means that the answer should not lead the client to derive wrong facts. Completeness is more subtle. It is tempting to say simply that a complete answer must provide all of the information the client needs to answer the query. However, this intuition is not appropriate in a system that permits intensional answers: consider that an intensional answer may direct the client to contact another site. Such an answer clearly does not provide all of the information the client needs to answer the query.

Instead, we use the following intuition for completeness: when a client asks a server a query q, it should receive an answer that guarantees that it does not have to ask the server the query q (or any instance of q) again. Formalizing this intuition is tricky, however, as we show by the following example.

Consider a program $\mathcal{P}$ distributed over two sites, s_1 and s_2:

$$\begin{array}{ll} s_1\$R(1) :- & s_2\$R(1) :- \\ & s_2\$R(2) :- \\ s_1\$R(x) :- s_2\$R(x) & s_2\$R(x) :- s_1\$R(x) \end{array}$$

If a client asks site s_1 the query $q = s_1\$R(x)$, what are the intensional answers that s_1 is allowed to return? Following our intuition, the extensional answer to the query, $\mathcal{F}_q = \{s_1\$R(1), s_1\$R(2)\}$, should be considered a complete answer, as should the full program $\mathcal{P}_{s_1}$ of s_1.

But consider a more interesting case: the answer $\mathcal{A}_1 = \{s_1\$R(x) :- s_2\$R(x)\}$. Should it be considered complete? By one account it should, since the client is instructed to contact site s_2, which holds both facts $s_2\$R(1)$ and $s_2\$R(2)$ that, together with $\mathcal{A}_1$, would allow it to infer all facts in $\mathcal{F}_q$. But then, by a similar reasoning, when the client asks s_2 the query $s_2\$R(x)$, the answer $\mathcal{A}_2 = \{s_2\$R(x) :- s_1\$R(x)\}$ would be complete too (since it instructs us to contact s_1, which holds the fact $s_1\$R(1)$ and can infer $s_1\$R(2)$). However, if the client has only $\mathcal{A}_1$ and $\mathcal{A}_2$, it cannot obtain the desired answer $\mathcal{F}_q$ without again asking the query $s_1\$R(x)$ or $s_1\$R(x)$. So, $\mathcal{A}_1$ and $\mathcal{A}_2$ should not be considered complete, because they force the client to ask the same queries again.

This shows that the definition of completeness must take into account the rules $\mathcal{R}$ that the client is gathering from other sites. Completeness should say that the answer $\mathcal{A}$, together with the rules $\mathcal{R}$, should allow the client to infer all of the facts in $\mathcal{F}_q$; that is, $\mathcal{R} \cup \mathcal{A} \models \mathcal{F}_q$. So, we need to consider the rules $\mathcal{R}$ more closely.

First, note that we cannot restrict $\mathcal{R}$ to be just the set of rules at all the other sites, since there is no guarantee that the client will gather exactly those rules; instead the definition will quantify over all sets $\mathcal{R}$, i.e. completeness will be something like $\forall \mathcal{R}.\mathcal{R} \cup \mathcal{A} \models \mathcal{F}_q$. At the same time, there must be some limitations on $\mathcal{R}$: if the server could assume nothing about the rules $\mathcal{R}$, then the only complete answer would be the extensional answer, $\mathcal{A} = \mathcal{F}_q$.

We reach a reasonable restriction on $\mathcal{R}$ as follows. We know that, using $\mathcal{R}$, the client may eventually derive all facts in $\mathcal{F}$ (the minimal model of $\mathcal{P}$). We also know that the client will never ask queries about q again, hence $\mathcal{R}$ may not include any of the rules about q, denoted $\mathcal{A}_q$ (more on this below). Hence we can safely restrict $\mathcal{R}$ to be a set of rules for which $\mathcal{R} \cup \mathcal{A}_q \models \mathcal{F}$, and the definition of completeness becomes something like:

$$\forall \mathcal{R}.\ \mathcal{R} \cup \mathcal{A}_q \models \mathcal{F} \Longrightarrow \mathcal{R} \cup \mathcal{A} \models \mathcal{F}_q$$

We will prove later that the definition remains unchanged if we replace $\mathcal{F}_q$ with $\mathcal{F}$, hence we use the latter from now on. We postpone for the moment the definition of $\mathcal{A}_q$ which should capture everything the server knows about q. To illustrate the definition on our example, take for the time being $\mathcal{A}_q$ to be the set of all rules with $s_1\$q(x)$ in the head, i.e., $\mathcal{P}_{s_1}$. Then we can see that $\mathcal{A}_1$ is not complete: choose $\mathcal{R}$ to be $\mathcal{P}_{s_2}$ less the first rule. $\mathcal{R}$ satisfies the restriction $\mathcal{R} \cup \mathcal{A}_q \models \mathcal{F}$, but $\mathcal{R} \cup \mathcal{A}_1 \not\models \mathcal{F}$, since $\mathcal{R} \cup \mathcal{A}_1 \not\models s_1\$R(1)$.

Before giving the formal definition of completeness and of $\mathcal{A}_s$, we need some additional notation. If $\mathcal{R}$ is a set of rules, we write $Mod(\mathcal{R})$ for the set of all models of $\mathcal{R}$. We write $\mathcal{R}_1 \models \mathcal{R}_2$ if every model of $\mathcal{R}_1$ is a model of $\mathcal{R}_2$. We use θ to range over substitutions, mapping variables to constants; substitutions are extended to atoms and rules in the usual way. For any set $\mathcal{R}$ of rules and set $\mathcal{Q}$ of queries, we define $\mathcal{R}_{\mathcal{Q}}$ to be the rules of $\mathcal{R}$ whose heads unify with some query in $\mathcal{Q}$:

$$\begin{aligned}\mathcal{R}_{\mathcal{Q}} &= \{\ a \text{ :- } a_1, \ldots, a_n \mid (a \text{ :- } a_1, \ldots, a_n) \in \mathcal{R}, \\ &\qquad\qquad inst(a) \cap inst(\mathcal{Q}) \neq \emptyset\ \}.\end{aligned}$$

In particular, if $\mathcal{F}$ is a set of facts, then $\mathcal{F}_{\mathcal{Q}} = \mathcal{F} \cap inst(\mathcal{Q})$. For any set $\mathcal{R}$ of rules and set $\mathcal{F}$ of facts, we define $\mathcal{R}[\mathcal{F}]$ to be the instantiation of $\mathcal{R}$ by facts in $\mathcal{F}$:

$$\begin{aligned}\mathcal{R}[\mathcal{F}] &= \{\ \theta(a \text{ :- } a_1, \ldots, a_n) \mid (a \text{ :- } a_1, \ldots, a_n) \in \mathcal{R}, \\ &\qquad\qquad \theta(a), \theta(a_1), \ldots, \theta(a_n) \in \mathcal{F}\ \}.\end{aligned}$$

DEFINITION 1. *Let $\mathcal{P}$ be a d3log program with minimal model $\mathcal{F}$, and let $\mathcal{Q}$ be a set of queries. Let $\mathcal{A}$ be a set of rules (not necessarily a subset of $\mathcal{P}$), intended to be the answer to the queries $\mathcal{Q}$.*

1. *$\mathcal{A}$ is* sound *if $\mathcal{F} \in Mod(\mathcal{A})$.*
2. *$\mathcal{A}$ is* program-complete *for $\mathcal{Q}$ if for any set of rules $\mathcal{R}$, if $\mathcal{R} \cup \mathcal{P}[\mathcal{F}]_{\mathcal{Q}} \models \mathcal{F}$, then $\mathcal{R} \cup \mathcal{A} \models \mathcal{F}$.*
3. *$\mathcal{A}$ is* model-complete *for $\mathcal{Q}$ if $\forall \mathcal{F}' \subseteq \mathcal{F}$, if $\mathcal{F}' \in Mod(\mathcal{A})$ then $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}})$.*

Program-completeness captures our intuition above, for the case when $\mathcal{Q} = \{q\}$. Continuing our earlier discussion, we note that the definition uses $\mathcal{P}[\mathcal{F}]_{\{q\}}$ for "set of rules about q", $\mathcal{A}_q$. These are not all rules at s whose head unifies with q: instead, they are all ground instances of rules at s whose head are instances of q (more on this below). In particular, $\mathcal{P}[\mathcal{F}]_{\mathcal{Q}}$ is a sound and complete answer to the query $\mathcal{Q}$, and can be thought of as a canonical answer that the server s may return.

Model-completeness offers an alternative definition and is equivalent to requiring that, if some new fact $f \in \mathcal{F}$ can be derived from other facts $\mathcal{F}' \subseteq \mathcal{F}$ using a canonical complete answer (i.e., $\mathcal{P}[\mathcal{F}]_{\mathcal{Q}}$), then f should be derivable from $\mathcal{F}'$ using the answer $\mathcal{A}$. To see this, in one direction we argue that, if $\mathcal{F}' \notin Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}})$ then there exists $f \notin \mathcal{F}'$ s.t. $\mathcal{F}' \cup \mathcal{P}[\mathcal{F}]_{\mathcal{Q}} \models f$, hence $\mathcal{F}' \cup \mathcal{A} \models f$: the latter implies $\mathcal{F}' \notin Mod(\mathcal{A})$. For the other direction, if $\mathcal{F}' \cup \mathcal{P}[\mathcal{F}]_{\mathcal{Q}} \models f$ but $\mathcal{F}' \cup \mathcal{A} \not\models f$ then we denote $\mathcal{F}''$ the minimal model of $\mathcal{F}' \cup \mathcal{A}$, and have $f \notin \mathcal{F}''$ and also $\mathcal{F}'' \in Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}})$, hence $\mathcal{F}' \cup Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}}) \not\models f$, which is a contradiction.

Why not choose $\mathcal{A}_q$ to be all rules at s whose head unifies with q ? This would correspond to replacing $\mathcal{P}[\mathcal{F}]_{\mathcal{Q}}$ with $\mathcal{P}_{\mathcal{Q}}$ in the definition of program completeness. But the resulting definition would be too restrictive, making even $\mathcal{F}_{\mathcal{Q}}$ an incomplete answer. For example, consider the following single site program $\mathcal{P}$:

$$\begin{aligned} T(1,2) &\text{ :- } \\ T(x,y) &\text{ :- } T(y,x)\end{aligned}$$

Let $\mathcal{Q}$ consist of the single query $T(1,x)$, and let $\mathcal{A}$ be its extensional answer, $\mathcal{A} = \{T(1,2)\}$. Here $\mathcal{P}_{\mathcal{Q}} = \mathcal{P}$. If $\mathcal{R} = \emptyset$, we have $\mathcal{R} \cup \mathcal{P}_{\mathcal{Q}} \models \mathcal{F}$, but $\mathcal{R} \cup \mathcal{A} \not\models \mathcal{F}$ (because $\mathcal{A} \not\models T(2,1)$). The problem is that $\mathcal{P}_{\mathcal{Q}}$ includes rules that allow us to infer facts that are not about $\mathcal{Q}$. This problem disappears if we use $\mathcal{P}[\mathcal{F}]_{\mathcal{Q}} = \{T(1,2), (T(1,2) \text{ :- } T(2,1))\}$: then $\mathcal{R} = \emptyset$ no longer satisfies $\mathcal{R} \cup \mathcal{P}[\mathcal{F}]_{\mathcal{Q}} \models \mathcal{F}$.

Similarly, we might have tried to use $\mathcal{F}_{\mathcal{Q}}$ as the canonical complete answer, but doing so would have resulted in an overly restrictive definition too, this time making $\mathcal{P}_s$ an incomplete answer. This is illustrated by the example program $\mathcal{P}$ of the previous page, which has minimal model

$$\mathcal{F} = \{s_1\$R(1), s_1\$R(2), s_2\$R(1), s_2\$R(2)\}.$$

If we let $\mathcal{Q} = \{s_1\$R(x)\}$, then $\mathcal{F}_{\mathcal{Q}} = \{s_1\$R(1), s_1\$R(2)\}$ and $\mathcal{P}[\mathcal{F}]_{\mathcal{Q}} = \{s_1\$R(1); s_1\$R(1) \text{ :- } s_2\$R(1); s_1\$R(2) \text{ :- } s_2\$R(2)\}$. If $\mathcal{R} = \{s_2\$R(x) \text{ :- } s_1\$R(x)\}$, then $\mathcal{R} \cup \mathcal{F}_{\mathcal{Q}} \models \mathcal{F}$. But $\mathcal{R} \cup \mathcal{P}_{s_1} \not\models \mathcal{F}$, because the minimal model of $\mathcal{R} \cup \mathcal{P}_{s_1}$ is $\{s_1\$R(1), s_2\$R(1)\}$, and this is not a model of $\mathcal{F}$. So $\mathcal{P}_{s_1}$ would not be a complete answer. The problem disappears with $\mathcal{P}[\mathcal{F}]_{\mathcal{Q}}$, because $\mathcal{R} \cup \mathcal{P}[\mathcal{F}]_{\mathcal{Q}} \not\models \mathcal{F}$.

Program-completeness and model-completeness have independent justifications and seem unrelated. A perhaps surprising result is the fact that they are equivalent.

THEOREM 2. *Program-completeness and model-completeness are equivalent notions.*

The proof is given in the appendix. An examination of the proof shows two facts about program-correctness. First, an equivalent formulation is obtained by replacing the last $\mathcal{F}$ with $\mathcal{F}_{\mathcal{Q}}$: for all $\mathcal{R}$, if $\mathcal{R} \cup \mathcal{P}[\mathcal{F}]_{\mathcal{Q}} \models \mathcal{F}$, then $\mathcal{R} \cup \mathcal{A} \models \mathcal{F}_{\mathcal{Q}}$. Second, one can further restrict the set of rules $\mathcal{R}$ to contain only closed rules of the form a_0 :- or a_0 :- a_1, with $a_0, a_1 \in \mathcal{F}$.

Since the two notions of completeness are equivalent, we will use just "completeness" in the sequel, and will refer to the definition of program-completeness or model-completeness as convenient.

We say that $\mathcal{A}$ is *correct* for $\mathcal{Q}$ if $\mathcal{A}$ is sound and complete for $\mathcal{Q}$. Correctness serves as a fundamental protocol between a client and servers: the client assumes that all servers return correct answers to queries (the **Correctness Assumption**) to prove its program correct. Sometimes we settle for soundness only, i.e., servers are assumed to return sound answers to queries (the **Soundness Assumption**).

Example 3 We illustrate here how computation steps can be interleaved with communication steps. Consider a query $q = s\$R(u,v)$. Assume that the program at s, $\mathcal{P}_s$, is:

$$\begin{aligned} s\$R(x,u) &\text{ :- } s\$T(x,y), s\$T(y,z), z\$Q(u) \\ s\$T(s_1,s_2) &\text{ :- } \\ s\$T(s_2,s_3) &\text{ :- } \\ s\$T(s_2,s_4) &\text{ :- } \end{aligned}$$

The first rule alone forms a correct answer, and we could return it thus without doing any computations. An answer that involves some computation, but no communication, performs the join in the first rule:

$$\begin{aligned}\mathcal{A}_1 = \{\ & s\$R(s_1,u) \text{ :- } s_3\$Q(u) \\ & s\$R(s_1,u) \text{ :- } s_4\$Q(u)\ \}\end{aligned}$$

We will show in the next section that $\mathcal{A}_1$ is correct. Another answer involves the same computation, plus queries to sites

s_3 and s_4. Assume that, in response to the query $s_3\$Q(u)$, site s_3 returns the answer $s_3\$Q(s_5)$. Further, in response to the query $s_4\$Q(u)$, site s_4 returns the answer $s_4\$Q(u)$:– $s_5\$T(u, v)$. Then the following is also a correct answer:

$$\begin{aligned}\mathcal{A}_2 = \{\ & s\$R(s_1, s_5) \text{ :–}\\ & s\$R(s_1, u) \text{ :– } s_4\$Q(u)\\ & s_4\$Q(u) \text{ :– } s_5\$T(u, v)\ \}\end{aligned}$$

We will show in the next section that $\mathcal{A}_2$ is also correct.

We end this section with some simple facts that are proven in the appendix.

PROPOSITION 4. *Soundness has the following properties:*

1. *If $\mathcal{A} \subseteq \mathcal{P}$ then $\mathcal{A}$ is sound.*
2. *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are sound then so is $\mathcal{A}_1 \cup \mathcal{A}_2$.*
3. *If $\mathcal{A}$ is extensional then $\mathcal{A}$ is sound iff $\mathcal{A} \subseteq \mathcal{F}$.*

PROPOSITION 5. *Completeness has the following properties:*

1. *If $\mathcal{P}_\mathcal{Q} \subseteq \mathcal{A}$ then $\mathcal{A}$ is complete for $\mathcal{Q}$.*
2. *If $\mathcal{A}_1$ is complete for $\mathcal{Q}_1$ and $\mathcal{A}_2$ is complete for $\mathcal{Q}_2$ then $\mathcal{A}_1 \cup \mathcal{A}_2$ is complete for $\mathcal{Q}_1 \cup \mathcal{Q}_2$.*
3. *If $\mathcal{A}$ is extensional then $\mathcal{A} \cap \mathcal{F}$ is complete for $\mathcal{Q}$ iff $\mathcal{F}_\mathcal{Q} \subseteq \mathcal{A}$.*

Item 1 of Prop. 5 implies that any set of rules $\mathcal{A}$ is complete for $\mathcal{Q} = \emptyset$. Together with 2 this implies that if $\mathcal{A}$ is complete for $\mathcal{Q}$ then $\mathcal{A} \cup \mathcal{A}'$ is also complete for $\mathcal{Q}$, for any $\mathcal{A}'$. Finally, Prop. 4 item 3 and Prop. 5 item 3 mean that if $\mathcal{A}$ is extensional, then $\mathcal{A}$ is sound and complete for $\mathcal{Q}$ iff $\mathcal{F}_\mathcal{Q} \subseteq \mathcal{A} \subseteq \mathcal{F}$.

5. GENERIC EVALUATION ALGORITHM

We describe here a *generic* evaluation algorithm computing an answer $\mathcal{A}$ in response to a query q at a site s. The algorithm is non-deterministic, and may return several answers $\mathcal{A}$: we prove that all are correct. We use this generic algorithm to show in the next section that various deterministic algorithms, implementing different evaluation strategies and different optimization techniques, return correct answers, by showing that they correspond to particular runs of the generic algorithm. Thus, the generic algorithm offers a powerful tool for showing that various concrete algorithms return correct answers.

Let $\mathcal{P}$ be a d3log program, and let $\mathcal{P}_s$ be the fragment at site s. The algorithm proceeds in a series of steps. Each step i computes two sets: RC_i, a set of rules called the *rule cache*, and QS_i, a set of atoms called the *query set*. Start with $\text{RC}_0 = \mathcal{P}_s$, and $\text{QS}_0 = \emptyset$. At each step i, $i > 0$, the algorithm either inserts new elements into RC_i or QS_i, or stops. When inserting in RC_i and QS_i we adopt the convention that two rules (or queries) are equal if they are equal up to variable renamings. Before describing step i, we need some terminology. We say that QS_i *covers* a site-determined atom a_1 if the site in a_1 is s or there exists some atom $a \in \text{QS}_i$ and some substitution θ s.t. $a_1 = \theta(a)$.

Step $(i+1)$ nondeterministically chooses one of the following three actions:

Evaluate. Choose a rule $r : a_0$:– $a_1, \ldots, a_n$ and a fact a :– in RC_i such that QS_i covers a_1, and $a = \theta(a_1)$ for some substitution θ. Define a new rule $r' : \theta(a_0)$:– $\theta(a_2), \ldots, \theta(a_n)$, and define

$$\text{QS}_{i+1} \leftarrow \text{QS}_i, \quad \text{RC}_{i+1} \leftarrow \text{RC}_i \cup \{r'\}.$$

This action is triggered only if $r' \notin \text{RC}_i$.

Communicate. Choose a rule $r : a_0$:– $a_1, \ldots, a_n$ in RC_i such that QS_i does not cover a_1. Let s_1 be the site of a_1. Send query a_1 to site s_1, wait for the answer $\mathcal{A}_1$, and define

$$\text{QS}_{i+1} \leftarrow \text{QS}_i \cup \{a_1\}, \quad \text{RC}_{i+1} \leftarrow \text{RC}_i \cup \mathcal{A}_1.$$

Stop. Stop and return as answer $\mathcal{A} = \text{RC}_i$.

Evaluate encapsulates traditional selection-join query evaluation; for instance in Example 3, the join in first rule is performed by first unifying the first atom ($s\$T(x, y)$) with all the facts about T, then unifying the second atom ($s\$T(y, z)$) with all these facts. **Communicate** sends a query to another server, then records that query in QS to avoid sending it later again.

A few observations. In the **Evaluate** step, the new rule r' is site safe, if the original rule was site safe: it follows by induction that all rules in RC_i are site safe. In the **Communicate** step, the site of a_1 is a constant (not a variable) since the rule is site safe.

Our first result is termination. For that we assume that all queries are answered in finite amount of time (this is a strong assumption: more on that in the next section).

THEOREM 6. *The generic server algorithm always terminates, i.e., there are no infinite runs.*

The proof is given in the Appendix and uses the fact that the accessible active domain is finite.

Next, we prove a crucial property: that at each step i of the algorithm a certain invariant holds.

THEOREM 7. *Under the **Soundness Assumption**, RC_i is sound. Under the **Correctness Assumption**, RC_i is correct for $\{q\} \cup \text{QS}_i$.*

Proof: by induction on i. For the base case, recall that $\text{RC}_0 = P_s$ and $\text{QS}_0 = \emptyset$, and by Props. 4(1) and 5(1), P_s

is correct for $\{q\}$. The inductive case has two subcases. If RC_{i+1} results from an **Evaluate** step, then $\mathrm{QS}_{i+1} = \mathrm{QS}_i$ and $\mathrm{RC}_{i+1} = \mathrm{RC}_i \cup \{r\}$, where r is a logical consequence of RC_i, and hence, soundness and completeness are preserved as a consequence of Props. 4 and 5. If RC_{i+1} results from a **Communicate** step, then $\mathrm{QS}_{i+1} = \mathrm{QS}_i \cup \{q'\}$ and $\mathrm{RC}_{i+1} = \mathrm{RC}_i \cup \mathcal{A}$, where (under the **Soundness Assumption**) $\mathcal{A}$ is sound or (under the **Correctness Assumption**) $\mathcal{A}$ is correct for $\{q'\}$. Under the **Soundness Assumption**, Prop. 4(2) implies that RC_{i+1} is sound; under the **Correctness Assumption**, Prop. 4(2) implies that RC_{i+1} is complete for QS_{i+1}. **End proof.**

Correctness of the generic algorithm follows from the Theorem.

COROLLARY 8. *Under the **Soundness Assumption**, any answer $\mathcal{A}$ returned by the generic algorithm is sound. Under the **Correctness Assumption**, any answer returned is correct for the input query q.*

Corollary 8 and the definitions of soundness and completeness show that, when used by the user (who expects an extensional answer, i.e., facts only, no rules), the generic algorithm returns a correct answer:

COROLLARY 9. *Let $\mathcal{A}$ be an extensional answer to the query q returned by the generic algorithm. Under the **Soundness Assumption**, $\mathcal{A} \subseteq \mathcal{F}$. Under the **Correctness Assumption**, $\mathcal{F}_{\{q\}} \subseteq \mathcal{A} \subseteq \mathcal{F}$.*

An Important Optimization

Consider the last two rules in answer $\mathcal{A}_2$ from Example 3:

$$s\$R(s_1, u) \coloneq s_4\$Q(u) \qquad s_4\$Q(u) \coloneq s_5\$T(u, v)$$

When the client receives them it doesn't know that the last rule is complete for the query $s_4\$Q(u)$: instead it will send the query $s_4\$Q(u)$ to server s_4 again, duplicating the work done by s. To avoid this, we modify the distributed evaluation paradigm to let servers notify clients about all the queries their answers are complete for. Luckily, we have all the machinery in place already. The response to a query q consists now of both an answer $\mathcal{A}$ and a set of queries $\mathcal{Q}$, s.t. $q \in \mathcal{Q}$ and $\mathcal{A}$ is correct for $\mathcal{Q}$. The generic algorithm is extended to return both $\mathcal{A} = \mathrm{RC}_i$ and $\mathcal{Q} = \{q\} \cup \mathrm{QS}_i$ in the **Stop** action. The **Communicate** action is modified to expect a reply $(\mathcal{A}_1, \mathcal{Q}_1)$, and QS_{i+1} becomes $\mathrm{QS}_i \cup \mathcal{Q}_1$. Theorems 6 and 7 remain unchanged. This simple extension is quite powerful in eliminating duplicate work. Returning to Example 3, in the case of $\mathcal{A}_2$ the answer returned to the client would consists of both $\mathcal{A}_2$ and $\mathcal{Q} = \{s\$R(u, v), s_4\$Q(u)\}$. This tells the client that $\mathcal{A}_2$ already contains everything needed to answer $s_4\$Q(u)$, and that it should not contact s_4 again if it needs the query in the future.

Reducing the size of intensional answers

The following proposition offers some simple methods to reduce the size of an intensional answer by deleting some of the rules.

PROPOSITION 10. *Let $\mathcal{P}$ be a d3log program, $\mathcal{Q}$ a query set, $\mathcal{A}$ a set of rules, and $r \in \mathcal{A}$. (1) If $\mathcal{A}$ is sound, then $\mathcal{A} - \{r\}$ is sound. (2) If $\mathcal{A}$ is correct for $\mathcal{Q}$ and $r \notin \mathcal{A}_{\mathcal{Q}}$ then $\mathcal{A} - \{r\}$ is correct for $\mathcal{Q}$. (3) Let r be of the form $a_0 \coloneq a_1, \ldots, a_n$, $n > 0$, s.t. $\mathcal{Q}$ covers each of $a_1, \ldots, a_n$, and the facts in $\mathcal{A}$ are closed under applications of r. Then $\mathcal{A} - \{r\}$ is also a correct answer for $\mathcal{Q}$.*

We can now show that $\mathcal{A}_1$ and $\mathcal{A}_2$ in Example 3 are correct. Answer $\mathcal{A}_1$ is obtained by executing **Evaluate** only, until no more changes are possible, then applying Prop. 10 to eliminate redundant rules. $\mathcal{A}_2$ is obtained by allowing, in addition, **Communicate** to execute on two queries, $s_3\$Q(u)$ and $s_4\$Q(u)$, but not on the fourth query, $s_5\$T(u, v)$.

6. APPLICATIONS

Query evaluation in distributed databases

In distributed databases [18, 14], usually the query expression sent to a site s is a SQL expression (or a logical plan), rather than an atom. We can model that in d3log by assuming that s defines a view q whose body is that expression, and the client asks for that view. Hence $\mathcal{P}_s$ has a single rule defining the view q, and all atoms in the body have site s. Query evaluation is performed locally, and an extensional answer $\mathcal{A}$ is returned. This corresponds to a particular run of the generic algorithm where only the **Evaluate** action is performed, until no more changes are possible (followed by repeated applications of Prop. 10 to eliminate all facts other than those in $inst(q)$). It follows that $\mathcal{A}$ is correct.

Failed communications

Some query request may never be answered due to server or network failure. We can modify the generic algorithm to cope with lost messages. In **Communicate**, after a certain timeout, we abort the request and let $\mathcal{A}_1 = \emptyset$. Since $\emptyset$ is always a sound answer (but not necessarily complete), the answers returned by the modified algorithm are still sound.

Intensional Answers

When a site s receives a query q, it can immediately return its entire program, $\mathcal{P}_s$. This is guaranteed to be a correct answer (by Props. 4 and 5). This is a form of *referral* because, in general, the program $\mathcal{P}_s$ can contain rules that refer to another site. In the database literature this has been called an *intensional answer* [12, 19] (in this paper we use the term *intensional* in a broader sense), and in the security literature it has been called a *hint* [15].

Answering with $\mathcal{P}_s$ can be inefficient in general: much of $\mathcal{P}_s$ may be irrelevant to the query. For certain applications however this may be a reasonable option. For example, in security applications, a server may refuse to evaluate a query from an untrusted client (to avoid a denial of service attack) and return its program instead: often that program simply contains references to other servers, which the client must then contact. Furthermore, Prop. 10 may be used to reduce the size of the intensional answer, if desired.

Naive Chaining

In a *chaining* algorithm, a server s that receives a query may query another site s_1; that, in turn, may send a query to a site s_2. Eventually, a new query may be sent to s, indirectly on behalf of its own request. It is easy to extend s to run multiple threads of the generic algorithm, to handle new requests while old ones are suspended. The problem is that chaining creates the possibility of loops, and some form of loop detection will be needed to ensure termination. (Recall that Theorem 6 assumed that each query is answered in finite time; this fails, of course, in the presence of loops.) Loops are prohibited in DNS and LDAP: the data is supposed to be arranged in a tree and form no cycles. In XML, however, XLinks may form cycles, and loop detection is necessary. We illustrate here a simple loop detection method.

We modify the naive chaining algorithm by tagging each query q sent to a server s with a set $\mathcal{S}$ of sites that are waiting for q to terminate. Every time a **Communication** step is attempted, the algorithm first checks the destination site s_1. If $s_1 \notin \mathcal{S}$, then the **Communication** step is executed. If $s_1 \in \mathcal{S}$ then the step is not executed, since that may result in a loop. Obviously both choices are legal in the generic algorithm, hence the answer returned by s is still correct (assuming all other servers return correct answers). This trivially detects loops, hence avoids nontermination (Thm. 6 applies). It is also a naive detection method: more complex methods are possible, but beyond the scope of this paper.

7. OPTIMIZATIONS

Standard optimization techniques can be used to speed d3log evaluation, but they may need to be modified to take dynamic site discovery into account. We use magic set rewriting [2, 3] as a canonical example.

Suppose a site s receives a query $s\$R(5, y)$ and its program $\mathcal{P}_s$ contains a single idb rule

$$s\$R(x, y) :\!-\ s\$E(x, w),\ w\$R(y, 6),$$

where $s\$E$ is an edb relation. Our bottom-up algorithm does not compute the answer to the query efficiently: it computes the entire relation $s\$R$ before selecting those tuples with 5 in the first position as the answer to the query.

Magic set rewriting is a datalog optimization that produces a more efficient program for computing the query. We suggest the following as a magic set rewriting of our d3log program:

$$\begin{array}{l} s\$R(5, y) :\!-\ s\$R^{bf}(5, y) \\ \mathit{magic}(5) :\!- \\ s\$R^{bf}(x, y) :\!-\ \mathit{magic}(x),\ s\$E(x, w),\ w\$R(y, 6) \end{array}$$

This introduces a new relation, $s\$R^{bf}$, useful for computing tuples of $s\$R$ whose first position is known (bound) and whose second position is unknown (free). Intuitively, the bottom-up algorithm will evaluate this program faster than the original program, because it will only compute those tuples of $s\$R$ with 5 in the first position. Because of dynamic site discovery, however, it is possible that the algorithm will have to compute more tuples of $s\$R$. For example, if $s\$E(5, s)$ holds, then by the rule for $s\$R^{bf}$, we need to compute $s\$R(y, 6)$. In datalog, this would mean that we should have included rules for a relation $s\$R^{fb}$ that would push the selection of 6 into the second position of $s\$R$.

In the case of d3log, the need for $s\$R^{fb}$ could not be determined without examining the contents of the edb relation $s\$E$, and, in general in d3log, it might require propagating the magic set rewriting across sites. Our approach is to defer this work until we know that it is necessary. This can be accomodated by the following slight modification of the naive chaining algorithm. When a site s receives a query q, it performs our variant of magic set rewriting on $\mathcal{P}_s$ w.r.t. q. This results in a program $\mathcal{C}$ that is used as the initial state of a modified version of the naive client algorithm. The site sets QS_0 to $\{q\}$. The algorithm is modified to treat s like a remote site—that is, in step i it may send a "query" to s if that query is not covered by the query set. The site s handles such a "query" by performing magic set rewriting on the query and its program $\mathcal{P}_s$, and using the result as its "reply."

Going back to our example, when the algorithm finds it needs to compute $s\$R(y, 6)$, it finds it is not an instance of the original query $s\$R(5, y)$, so it causes the "query" $s\$R(y, 6)$ to be sent to s, which replies with the magic set rewriting

$$\begin{array}{l} s\$R(y, 6) :\!-\ s\$R^{fb}(y, 6) \\ \mathit{magic}_2(6) :\!- \\ s\$R^{fb}(x, y) :\!-\ \mathit{magic}_2(y),\ s\$E(x, w),\ w\$R(y, 6) \end{array}$$

So, now the rule cache holds a bigger magic set rewriting.

Our technical machinery can easily be extended to accomodate magic set rewriting; with some slight additions to handle fresh relation names, all of our lemmas and theorems continue to hold.

8. RELATED WORK

Distributed databases have been an active area of research since the early 80's, and today every database vendor offers support for them. We refer to [18] for a textbook and [14] for an overview of the current state of the art.

Distributed evaluation of datalog programs poses additional challenges due to recursion. Early work [8] focused on parallelizing a centralized datalog program, to speed up evaluation. Each rule is sent to a different site responsible for firing that rule, and sites exchange tuples, cooperating in the computation of the least fixpoint. Other approaches to parallelization attempt to reduce the number of communications by probabilistic methods [16] or by preanalyzing the data [6]. In other work [25, 17], the setting is closer to ours, in that edbs and/or the datalog program are distributed a priori. Evaluation proceeds in parallel, with each site evaluating a local fixpoint, then exchanging data with other sites, then resuming the evaluation. A major component of this approach is termination detection. In all these cases, the information exchanged between sites consists of data only, and no intensional answers are considered.

Intensional answers in response to queries over datalog programs are considered by Imielinski [12], and are motivated both conceptually and computationally. Conceptually, an intensional answer is sometimes more informative than a table. Computationally, an intensional answer may be much smaller than the table and permits the computation to migrate from the server to the client, which is useful in some applications. Imielinski's techniques go beyond what the generic algorithm in Sec. 5 can do, and aim at producing the "most intensional answer" possible, ideally consisting of a datalog program whose unique idb is the query predicate. By contrast, in our work we are concerned with describing a large space of possible answers, and letting a server apply optimization techniques to choose the best one. This space could be extended with the techniques in [12]. Another application of intensional answers are parachute queries [4], which are used in a mediator based data integration system when one of the sources is unavailable.

Our work was directly motivated by QCM [9, 10, 11], a system for building security infrastructures. As explained in Sec. 3, answers in QCM are signed, so the client can verify that the answer indeed comes from the correct server. Signing, of course, requires a private (secret) key, and it is sometimes considered a security threat to keep a private key on a machine accessible from the open Internet. For this reason, sometimes QCM servers do not have access to private keys, but rather to *certificates*, which are pre-signed documents whose contents are the rules of the server. Such a server can answer queries with certificates; so certificates are another motivation for intensional answers. In addition, a QCM client may be asked to evaluate a query using a certificate. For example, if a QCM client is given a document "$K\$PKD(\text{Alice}, K_{\text{Alice}})$:–" signed by the private key of K, it can evaluate the query $Q(x)$:– $K\$PKD(\text{Alice}, x)$ without sending a message to K.[2] This is essentially the problem of answering queries using views [7, 24] or caches [20, 5], and it can be applied to d3log as well. D3log extends QCM by treating recursion and intensional answers; SD3 [13] is the secure extension of d3log, and is the successor of QCM.

Another novel approach to distributed query evaluation can be found in [21, 22], where a distributed query plan is expressed in terms of the pi calculus.

9. CONCLUSION AND FUTURE WORK

We have described a new distributed query evaluation paradigm for the Web, modeling dynamic site discovery. Its salient feature is that now servers may return an intensional answer in response to a query, as opposed to data only. A server has numerous alternatives for constructing the intensional answer for a given query: we have defined what a correct answer means, and described a generic nondeterministic algorithm that encapsulates several such choices. We proved that all runs of the nondeterministic algorithm result in a correct answer, and applied that to showing that specific optimization strategies produce correct answers.

An optimizer at a server would normally choose a best plan for computing an intensional answer based on a variety of factors, including data statistics, server state, and user profile. Such an optimizer is part of future work. What we have achieved here however is a guarantee that such an optimizer will return a correct answer, as long as the plan it generates corresponds to some particular run of the nondeterministic generic algorithm.

10. REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24–26, 1986, Cambridge, Massachusetts*, pages 1–16. ACM, 1986.

[3] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23–25, 1987, San Diego, California*, pages 269–283. ACM, 1987.

[4] P. Bonnet and A. Tomasic. Partial answers for unavailabel data sources. In *International Conference on Flexible Query Answering Systems*, volume 1495 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 1998.

[5] Chungmin Melvin Chen and Nick Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *4th International Conference on Extending Database Technology*, volume 779 of *Lecture Notes in Computer Science*, pages 323–336. Springer-Verlag, 1994.

[6] Guozhu Dong. On distributed processibility of datalog queries by decomposing databases. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31–June 2, 1989*, pages 26–35. ACM Press, 1989.

[7] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS–97*, May 1997.

[8] Allen Van Gelder. A message passing framework for logical query evaluation. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28–30, 1986*, pages 155–165. ACM Press, 1986.

[9] Carl A. Gunter and Trevor Jim. Design of an application-level security infrastructure. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[10] Carl A. Gunter and Trevor Jim. Generalized certificate revocation. In *Conference Record of POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.

[11] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. *Software: Practice & Experience*, 30(15):1609–1640, September 2000.

[2]The syntax of QCM is actually based on set comprehensions, but we use a datalog style here for consistency of presentation.

[12] Tomasz Imielinski. Intelligent query answering in rule based systems. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 275–312. Morgan Kaufmann, 1988.

[13] Trevor Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, May 2001.

[14] Donald Kossmann. The state of the art in distributed query processing. Submitted to ACM Computing Surveys, 1999.

[15] Butler Lampson and Ron Rivest. SDSI—a simple distributed security infrastructure. `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[16] Sérgio Lifschitz and Victor Vianu. A probabilistic view of datalog parallelization. In Georg Gottlob and Moshe Y. Vardi, editors, *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11–13, 1995, Proceedings*, volume 893 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 1995.

[17] Wolfgang Nejdl, Stefano Ceri, and Gio Wiederhold. Evaluating recursive queries in distributed databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):104–121, 1993.

[18] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, second edition, 1999.

[19] Alain Pirotte, Dominique Roelants, and Esteban Zimanyi. Controlled generation of intensional answers. In J.W. Schmidt and A.A. Stogny, editors, *First International East/West Database Workshop on Next Generation Information Technology*, volume 504 of *Lecture Notes in Computer Science*, pages 251–270. Springer-Verlag, 1990.

[20] Nick Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *ACM Transactions on Database Systems*, 16(3):535–563, 1991.

[21] Arnaud Sahuguet, Benjamin Pierce, and Val Tannen. Chaining, referral, subscription, leasing: New mechanisms in distributed query optimization. Submitted for publication, 2000.

[22] Arnaud Sahuguet, Benjamin Pierce, and Val Tannen. Distributed query optimization: Can mobile agents help? Submitted for publication, 2000.

[23] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[24] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of the Sixth International Conference on Database Theory*, 1997.

[25] Ouri Wolfson and Abraham Silberschatz. Distributed processing of logic programs. In Haran Boral and Per-Ake Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1–3, 1988*, pages 329–336. ACM Press, 1988.

APPENDIX

A. PROOFS

Proof of Theorem 2: Assume that $\mathcal{A}$ is model-complete for $\mathcal{Q}$. We prove the counterpositive statement of program-completeness. Let $\mathcal{R}$ be s.t. $\mathcal{R}, \mathcal{A} \not\models \mathcal{F}$. Let $\mathcal{F}'$ be some model (e.g., the minimal model) of $\mathcal{R}, \mathcal{A}$ that does not contain $\mathcal{F}$. Consider $\mathcal{F} \cap \mathcal{F}'$: it is in $Mod(\mathcal{A})$, hence (by model-completeness) it is in $Mod(\mathcal{P}[\mathcal{F}]_\mathcal{Q})$. We would like to show that $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_\mathcal{Q})$. Indeed, take some rule r in $\mathcal{P}[\mathcal{F}]_\mathcal{Q}$, s.t. $body(r) \subseteq \mathcal{F}'$. We also have $body(r) \subseteq \mathcal{F}$ (by definition of $\mathcal{P}[\mathcal{F}]$), hence $body(r) \subseteq \mathcal{F} \cap \mathcal{F}'$. Hence $head(r) \in \mathcal{F}'$. We have shown $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_\mathcal{Q})$ and, since $\mathcal{F}' \in Mod(\mathcal{R})$ we have $\mathcal{F}' \in Mod(\mathcal{R}, \mathcal{P}[\mathcal{F}]_\mathcal{Q})$, hence $\mathcal{R}, \mathcal{P}[\mathcal{F}]_\mathcal{Q} \not\models \mathcal{F}$.

Assume now that $\mathcal{A}$ is program-complete for $\mathcal{Q}$. Let $\mathcal{F}' \subseteq \mathcal{F}$ s.t. $\mathcal{F}' \in Mod(\mathcal{A})$. We want to show that $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_\mathcal{Q})$. Suppose not: then there exists a rule $r \in \mathcal{P}[\mathcal{F}]_\mathcal{Q}$ s.t. $body(r) \subseteq \mathcal{F}'$ and $head(r) \notin \mathcal{F}'$. Define $\mathcal{R}_1 = \{ f \text{ :- } head(r) \mid f \in \mathcal{F} \}$. Thus, $\mathcal{R}_1$ is a program whose rules say that if $head(r)$ is true, then everything in $\mathcal{F}$ is true. Define $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{F}'$. We have[3] $\mathcal{R} \cup \mathcal{A} \not\models \mathcal{F}$ because $\mathcal{F}'$ is a model of $\mathcal{R} \cup \mathcal{A}$. By program-completeness we must have $\mathcal{R} \cup \mathcal{P}[\mathcal{F}]_\mathcal{Q} \not\models \mathcal{F}$. But this is not the case since $\mathcal{F}' \cup \mathcal{P}[\mathcal{F}]_\mathcal{Q} \models head(r)$, which implies $\mathcal{R} \cup \mathcal{P}[\mathcal{F}]_\mathcal{Q} \models \mathcal{F}$. **End proof.**

Proof of Prop. 5: We use model-completeness in all proofs. (1) Let $\mathcal{F}' \subseteq \mathcal{F}$, $\mathcal{F}' \in Mod(\mathcal{A}) \subseteq Mod(\mathcal{P}_\mathcal{Q})$. To show $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_\mathcal{Q})$, let f be a fact s.t. $\mathcal{F}' \cup \mathcal{P}[\mathcal{F}]_\mathcal{Q} \models f$. Since $\mathcal{P}[\mathcal{F}]_\mathcal{Q} \subseteq \mathcal{P}_\mathcal{Q}[\mathcal{F}]$ we have $\mathcal{F}' \cup \mathcal{P}_\mathcal{Q}[\mathcal{F}] \models f$, hence $\mathcal{F}' \cup \mathcal{P}_\mathcal{Q} \models f$, which means $f \in \mathcal{F}'$, because $\mathcal{F}' \in Mod(\mathcal{P}_\mathcal{Q})$. (2) let $\mathcal{F}' \subseteq \mathcal{F}$, $\mathcal{F}' \in Mod(\mathcal{A}_1 \cup \mathcal{A}_2) = Mod(\mathcal{A}_1) \cap Mod(\mathcal{A}_2)$. For each $i = 1, 2$ we have $\mathcal{F}' \in Mod(\mathcal{A}_i)$, hence $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}_i})$. We have $\mathcal{P}[\mathcal{F}]_{\mathcal{Q}_1 \cup \mathcal{Q}_2} = \mathcal{P}[\mathcal{F}]_{\mathcal{Q}_1} \cup \mathcal{P}[\mathcal{F}]_{\mathcal{Q}_2}$, hence $Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}_1 \cup \mathcal{Q}_2}) = Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}_1}) \cap Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}_2})$, hence $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_{\mathcal{Q}_1 \cup \mathcal{Q}_2})$. (3) First assume $\mathcal{F}_\mathcal{Q} \subseteq \mathcal{A}$. If $\mathcal{F}' \subseteq \mathcal{F}$, $\mathcal{F}' \in Mod(\mathcal{A} \cap \mathcal{F})$ then the latter implies $\mathcal{A} \cap \mathcal{F} \subseteq \mathcal{F}'$, while the former implies $\mathcal{F}_\mathcal{Q} \subseteq \mathcal{F}'$, hence $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_\mathcal{Q})$. Now assume $\mathcal{A} \cap \mathcal{F}$ is complete for $\mathcal{Q}$. Suppose $\mathcal{F}_\mathcal{Q} \not\subseteq \mathcal{A} \cap \mathcal{F}$, and let $f \in \mathcal{F}_\mathcal{Q} - \mathcal{A} \cap \mathcal{F}$. Define $\mathcal{F}' = \mathcal{F} - \{f\}$. We have $\mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{F}' \in Mod(\mathcal{A} \cap \mathcal{F})$. By completeness we have $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_\mathcal{Q})$. But then we show that $\mathcal{F}' \in Mod(\mathcal{P})$, which is obviously a contradicion. Indeed, consider some rule $a \text{ :- } a_1, \ldots, a_n \in \mathcal{P}$ and some substitution θ s.t. $\theta(a_1), \ldots, \theta(a_n) \in \mathcal{F}'$. If $\theta(a) = f$ then $\theta(a) \in \mathcal{F}_\mathcal{Q}$, and the rule $\theta(a \text{ :- } a_1, \ldots, a_n)$ is in $\mathcal{P}[\mathcal{F}]_\mathcal{Q}$, which would imply $\theta(a) \in \mathcal{F}'$ (since $\mathcal{F}' \in Mod(\mathcal{P}[\mathcal{F}]_\mathcal{Q})$), a contradiction. If $\theta(a) \neq f$, then $\theta(a) \in \mathcal{F}'$, showing that $\mathcal{F}' \in Mod(\mathcal{P})$. **End proof.**

Proof of Theorem 6: We will prove that for any run there exists some i s.t. $\forall j > i, \text{RC}_i = \text{RC}_j$ and $\text{QS}_i = \text{QS}_j$. Define $\text{QS} = \bigcup_{i \geq 0} \text{QS}_i$. Obviously QS is finite, because we have only a finite number of relation symbols and a finite number of constants in the accessible active domain, hence we can only build finitely many distinct queries (up to variable renamings). Hence there exists some j s.t. $\forall i > j, \text{QS}_i = \text{QS}_j$. Define $\text{RC} = \bigcup_{i \geq 0} \text{RC}_i$: we show now that RC is finite. We need a definition here. Given a rule r, $a_0 \text{ :- } a_1, \ldots a_n$, we say that a rule r' is a *postfix* rule of r if r' has the form $\theta(a_0 \text{ :- } a_k, a_{k+1}, \ldots, a_n)$, where $k = 1, \ldots, n$ and θ is a substitution. "Postfix" is a transitive relation, i.e., if r' is a postfix rule of r and r'' is a postfix rule of r', then r'' is a postfix rule of r. "Postfix" is also reflexive, i.e., for every rule r, r is a postfix rule of itself. Obviously, for a given rule r there are only finitely many postfix rules (up to variable renamings), because there are only finitely many postfixes and only finitely many constants. Let $\text{QS} = \{ q_1, \ldots, q_m \}$, let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be the answers obtained in response to the queries $q_1, \ldots, q_m$, and let $\mathcal{A} = \mathcal{P}_s \cup \bigcup_{i=1,m} \mathcal{A}_i$: obviously, $\mathcal{A}$ is finite. We show by induction on i that every rule in RC_i is a postfix rule of some rule in $\mathcal{A}$. This trivially holds for $i = 0$, since $\text{RC}_0 = \mathcal{P}_s$. Assuming it holds for i, there are two cases. In a **Evaluate** step $\text{RC}_{i+1} = \text{RC}_i \cup \{ r' \}$: here the new rule r' added is a postfix rule of the rule $r \in \text{RC}_i$, and we use the fact that "postfix" is transitive. In a **Communicate** step $\text{RC}_{i+1} = \text{RC}_i \cup \mathcal{A}_j$, where $\mathcal{A}_j$ is the answer to some query $q_j \in \text{QS}$. Here the new rules are already in $\mathcal{A}$, and we use the fact that "postfix" is reflexive. This completes the induction. It follows that RC is finite, up to variable renaming, hence there exists some j s.t. $\forall i > j, \text{RC}_i = \text{RC}_j$, which completes the proof. **End proof**.

[3]In particular we also have $\mathcal{R} \cup \mathcal{A} \not\models \mathcal{F}_\mathcal{Q}$, showing that $\mathcal{F}_\mathcal{Q}$ may replace the rightmost occurrence of $\mathcal{F}$ in the definition of program-completeness.

B. THE DOMAIN NAME SYSTEM

The Domain Name System, or DNS, is the distributed database that maps Internet host names to numerical IP addresses. This mapping is defined by many different *nameservers*, each of which is responsible for part of the mapping. The mapping is apportioned to nameservers hierarchically, in a way closely related to the hierarchy of domain names. For example, the figure below gives a portion of the DNS namespace. Each label represents a domain, whose name is the sequence of labels read from right to the root at the left, separated by dots.

```
. --+-- com --+-- att --+-- research --+-- www
    |                   |              |
    +-- edu --+-- mit   +-- www        +-- raptor
              |
              +-- cmu
```

The mapping to IP addresses is maintained by several nameservers through the use of *resource records*, including address (A) records, nameserver (NS) records, and start-of-authority (SOA) records. The d3log program below shows how resource records encode the DNS mapping; we use actual IP addresses.

```
site 198.41.0.4 ; # NAMESERVER FOR .
SOA(.,a.root-servers.net.)           :- ;
NS(.,a.root-servers.net.)            :- ;
A(a.root-servers.net.,198.41.0.4)    :- ;

NS(com.,a.gtld-servers.net.)         :- ;
A(a.gtld-servers.net.,198.41.3.38) :- ;

site 198.41.3.38 ; # NAMESERVER FOR com.
SOA(com.,a.gtld-servers.net.)        :- ;
NS(com.,a.gtld-servers.net.)         :- ;
A(a.gtld-servers.net.,198.41.3.38) :- ;

NS(att.com.,kcgw1.att.com.)          :- ;
A(kcgw1.att.com.,192.128.133.77)     :- ;
```

```
NS(.,a.root-servers.net.)          :- ;
A(a.root-servers.net.,198.41.0.4)  :- ;

site 192.128.133.77 ; # NAMESERVER FOR att.com.
SOA(att.com.,kcgw1.att.com.)              :- ;
NS(att.com.,kcgw1.att.com.)               :- ;
A(kcgw1.att.com.,192.128.133.77)          :- ;

NS(research.att.com.,ns.research.att.com.) :- ;
A(ns.research.att.com.,192.20.225.4)       :- ;

NS(.,a.root-servers.net.)                 :- ;
A(a.root-servers.net.,198.41.0.4)         :- ;
A(www.att.com.,192.20.3.54)               :- ;

site 192.20.225.4 ; # NAMESERVER FOR research.att.com.
SOA(research.att.com.,ns.research.att.com.) :- ;
NS(research.att.com.,ns.research.att.com.)  :- ;
A(ns.research.att.com.,192.20.225.4)        :- ;

NS(.,a.root-servers.net.)                   :- ;
A(a.root-servers.net.,198.41.0.4)           :- ;

CNAME(www.research.att.com.,
      akpublic.research.att.com.) :- ;
A(akpublic.research.att.com.,192.20.225.10) :- ;
A(raptor.att.com.,135.207.23.32)            :- ;
```

For example, A(a.root-servers.net.,198.41.0.4) is a record indicating that the IP address of host a.root-servers .net is 198.41.0.4. NS(.,a.root-servers.net.) indicates that a.root-servers.net is the nameserver for the part of the domain name mapping starting at the root, except for portions (subtrees) delegated to other nameservers. The record NS(com.,a.gtld-servers.net.) at the nameserver a.root-servers.net shows that the root nameserver has delegated the com subtree to another nameserver, a.gtld-servers.net. Each nameserver also maintains a start-of-authority record; for example, SOA(.,a.root-servers.net.) is maintained at the root nameserver. (Both NS records and SOA records are needed in DNS because of replication, which we do not discuss.)

Accessing the DNS mapping may require querying multiple nameservers. This is accomplished by a “client” nameserver that we operate in chaining mode. The client nameserver has the following program:

```
Down(x,n,a) :- x$SOA(n2,n3), n !>= n2,
               NS(.,n4), A(n4,a4),
               Down(a4,n,a) ;

Down(x,n,a) :- x$SOA(n2,n3), n>n2,
               x$NS(n4,n5), n>=n4, n4>n2,
               x$A(n5,a5),
               Down(a5,n,a) ;

Down(x,n,a) :- x$SOA(n2,n3), n>=n2, x$A(n,a) ;

Down(x,n,a) :- x$SOA(n2,n3), n>=n2, x$CNAME(n,n4),
               x$A(n4,a) ;

DNS(n,a) :- Down(198.41.0.4,n,a) ;
```

Queries on the relation DNS can be used to get the address of a host. For example, DNS(www.att.com.,a) will get the address of AT&T’s web server.

DNS is defined in terms of a helper relation: Down(x,n,a) holds if the nameserver at IP address x says that a is the address of host n. DNS invokes Down with a particular nameserver address, in this case, the address of the root nameserver. Note that Down is not a safe rule. However, when magic set rewriting is applied to the program, we obtain a safe set of rules that can be executed bottom-up.

The first Down rule says that if the nameserver x is not an authority for the name that we are looking up, we should find the root nameserver (which every nameserver knows), find its address, and consult the root nameserver.

The second rule says that if x is an authority for the name we are looking up, but it has delegated responsibility for the name to another nameserver, we should look up the address of that nameserver and see what it says.

The third rule says that if x has the address of n, it can just be returned.

The fourth rule says that if x is an alias, we should look up its canonical name (CNAME) and find its address.

A search for the address of a host name would then start at the root and proceed down the hierarchy of nameservers, which loosely corresponds to the hierarchy of domain names.

This is not the only way to encode DNS in our language. For example, instead of using the recursive program Down, we could put the following rules at each nameserver:

```
DNS(n,a) :- SOA(n2,a2), n !>= n2,
            NS(.,n3), A(n3,a3),
            a3$DNS(n,a) ;
DNS(n,a) :- SOA(n2,a2), n>n2,
            NS(n3,n4), n>n3,
            A(n4,a3), a3$DNS(n,a) ;
DNS(n,a) :- A(n,a) ;
DNS(n,a) :- CNAME(n,n2), A(n2,a) ;
```

The idea is that instead of querying nameservers for A, NS, and SOA records, the client nameserver will ask each nameserver x about its DNS relation x$DNS. Then each nameserver will consult its own A, NS, and SOA records, and return a DNS fact, or refer the client to another nameserver, or, if in chaining mode, contact other nameservers itself. This is the strategy described in Section 3.