DOI:10.1145/1592761.1592785

Declarative Networking

By Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica

Abstract

Declarative Networking is a programming methodology that enables developers to concisely specify network protocols and services, which are directly compiled to a dataflow framework that executes the specifications. This paper provides an introduction to basic issues in declarative networking, including language design, optimization, and dataflow execution. We present the intuition behind declarative programming of networks, including roots in Datalog, extensions for networked environments, and the semantics of long-running queries over network state. We focus on a sublanguage we call *Network Datalog* (*NDlog*), including execution strategies that provide crisp eventual consistency semantics with significant flexibility in execution. We also describe a more general language called *Overlog*, which makes some compromises between expressive richness and semantic guarantees. We provide an overview of declarative network protocols, with a focus on routing protocols and overlay networks. Finally, we highlight related work in declarative networking, and new declarative approaches to related problems.

1. INTRODUCTION

Over the past decade there has been intense interest in the design of new network protocols. This has been driven from below by an increasing diversity in network architectures (including wireless networks, satellite communications, and delay-tolerant rural networks) and from above by a quickly growing suite of networked applications (peer-to-peer systems, sensor networks, content distribution, etc.)

Network protocol design and implementation is a challenging process. This is not only because of the distributed nature and large scale of typical networks, but also because of the need to balance the extensibility and flexibility of these protocols on one hand, and their robustness and efficiency on the other hand. One needs to look no further than the Internet for an illustration of these hard trade-offs. Today's Internet routing protocols, while arguably robust and efficient, make it hard to accommodate the needs of new applications such as improved resilience and higher throughput. Upgrading even a single router is hard. Getting a distributed routing protocol implemented correctly is even harder. Moreover, in order to change or upgrade a deployed routing protocol today, one must get access to *each* router to modify its software. This process is made even more tedious and error-prone by the use of conventional programming languages.

In this paper, we introduce *declarative networking*, an application of database query language and processing techniques to the domain of networking. Declarative networking is based on the observation that network protocols deal at their core with computing and maintaining distributed state (e.g., routes, sessions, performance statistics) according to basic information locally available at each node (e.g., neighbor tables, link measurements, local clocks) while enforcing constraints such as local routing policies. Recursive query languages studied in the deductive database literature[27] are a natural fit for expressing the relationship between base data, derived data, and the associated constraints. As we demonstrate, simple extensions to these languages and their implementations enable the natural expression and efficient execution of network protocols.

In a series of papers with colleagues, we have described how we implemented and deployed this concept in the *P2* declarative networking system.[24] Our high-level goal has been to provide software environments that can accelerate the process of specifying, implementing, experimenting with and evolving designs for network architectures.

As we describe in more detail below, declarative networking can reduce program sizes by orders of magnitude relative to traditional approaches, in some cases resulting in programs that are line-for-line translations of pseudocode in networking research papers. Declarative approaches also open up opportunities for automatic protocol optimization and hybridization, program checking, and debugging.

2. LANGUAGE

In this section, we present an overview of the Network Datalog (*NDlog*) language for declarative networking. The *NDlog* language is based on extensions to traditional Datalog, a well-known recursive query language designed for querying graph-structured data in a centralized database. *NDlog*'s integration of networking and logic is unique from the perspectives of both domains. As a network protocol language, it is notable for the absence of any communication primitives like "send" or "receive"; instead, communication is implicit in a simple high-level specification of data partitioning. In comparison to traditional logic languages, it is enhanced to capture typical network realities including distribution, link-layer constraints on communication (and hence deduction), and soft-state[8] semantics.

We step through an example to illustrate the standard execution model for Datalog, and demonstrate its close connections to routing protocols, recursive network graph computations, and distributed state management. We then describe the *Overlog*[21] extensions to the *NDlog* language that support soft-state data and events.

A previous version of this paper was published in *Proceedings of ACM SIGMOD's International Conference of Management of Data* (2006).

2.1. Introduction to Datalog

We first provide a short review of Datalog, following the conventions in Ramakrishnan and Ullman's survey.[27] A Datalog program consists of a set of declarative *rules* and an optional *query*. Since these programs are commonly called "*recursive queries*" in the database literature, we use the term "query" and "program" interchangeably when we refer to a Datalog program.

A Datalog *rule* has the form p :- $q_1, q_2, \ldots, q_n$, which can be read informally as "q_1 and q_2 and … and q_n implies p." p is the *head* of the rule, and $q_1, q_2, \ldots, q_n$ is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* over *fields* (variables and constants), or functions (formally, *function symbols*) applied to fields. The rules can refer to each other in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial. The commas separating the predicates in a rule are logical conjuncts (*AND*); the order in which predicates appear in a rule body also has no semantic significance, though most implementations (including ours) employ a left-to-right execution strategy. Predicates in the rule body are matched (or *joined*) based on their common variables to produce the output in the rule head. The *query* (denoted by a reserved rule label `Query`) specifies the output of interest.

The predicates in the body and head of traditional Datalog rules are relations, and we refer to them interchangeably as predicates or relations. In our work, every relation has a *primary key*, which is a set of fields that uniquely identifies each tuple within the relation. In the absence of other information, the primary key is the full set of fields in the relation.

By convention, the names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter. Most implementations of Datalog enhance it with a limited set of side-effect-free function calls including standard infix arithmetic and various simple string and list manipulations (which start with "`f_`" in our syntax). Aggregate constructs are represented as aggregation functions with field variables within angle brackets (⟨⟩).

2.2. NDLog by example

We introduce *NDlog* using an example program shown below that implements the *path-vector protocol*, which computes in a distributed fashion, for every node, the shortest paths to all other nodes in a network. The path-vector protocol is used as the base routing protocol for exchanging routes among Internet Service Providers.

```
sp1 path(@Src,Dest,Path,Cost) :- link(@Src,Dest,Cost),
    Path=f_init(Src,Dest).
sp2 path(@Src,Dest,Path,Cost) :- link(@Src,Nxt,Cost1),
    path(@Nxt,Dest,Path2,Cost2), Cost=Cost1+Cost2,
    Path=f_concatPath(Src,Path2).
sp3 spCost(@Src,Dest,min<Cost>) :- path(@Src,Dest,Path,Cost).
sp4 shortestPath(@Src,Dest,Path,Cost) :-
    spCost(@Src, Dest,Cost), path(@Src,Dest,Path,Cost).
Query shortestPath(@Src,Dest,Path,Cost).
```

The program has four rules (which for convenience we label `sp1`–`sp4`), and takes as input a base (*extensional*) relation `link(Src,Dest,Cost)`. Rules `sp1`–`sp2` are used to derive "paths" in the graph, represented as tuples in the derived (*intensional*) relation `path(Src,Dest,Path,Cost)`. The `Src` and `Dest` fields represent the source and destination endpoints of the path, and `Path` is the actual path from `Src` to `Dest`. The number and types of fields in relations are inferred from their (consistent) use in the program's rules.

Since network protocols are typically computations over distributed network state, one of the important requirements of *NDlog* is the ability to support rules that express distributed computations. *NDlog* builds upon traditional Datalog by providing control over the storage location of tuples explicitly in the syntax via *location specifiers*. Each location specifier is a field within a predicate that dictates the partitioning of the table. To illustrate, in the above program, each predicate has an "`@`" symbol prepended to a single field denoting the location specifier. Each tuple generated is stored at the address determined by its location specifier. For example, each `path` and `link` tuple is stored at the address held in its first field `@Src`.

Rule `sp1` produces `path` tuples directly from existing `link` tuples, and rule `sp2` recursively produces `path` tuples of increasing cost by matching (joining) the destination fields of existing links to the source fields of previously computed paths. The matching is expressed using the repeated `Nxt` variable in `link(Src,Nxt,Cost1)` and `path(Nxt,Dest,Path2,Cost2)` of rule `sp2`. Intuitively, rule `sp2` says that "if there is a link from node `Src` to node `Nxt`, and there is a path from node `Nxt` to node `Dest` along a path `Path2`, then there is a path `Path` from node `Src` to node `Dest` where `Path` is computed by prepending `Src` to `Path2`." The matching of the common `Nxt` variable in `link` and `path` corresponds to a *join* operation used in relational databases.

Given the `path` relation, rule `sp3` derives the relation `spCost(Src,Dest,Cost)` by computing the minimum cost `Cost` for each source and destination for all input paths. Rule `sp4` takes as input `spCost` and `path` tuples and then finds `shortestPath(Src,Dest,Path,Cost)` tuples that contain the shortest path `Path` from `Src` to `Dest` with cost `Cost`. Last, as denoted by the `Query` label, the `shortestPath` table is the output of interest.

2.3. Shortest path execution example

We step through an execution of the *shortest-path NDlog* program above to illustrate derivation and communication of tuples as the program is computed. We make use of the example network in Figure 1. Our discussion is necessarily informal since we have not yet presented our distributed implementation strategies; in the next section, we show in greater detail the steps required to generate the execution plan. Here, we focus on a high-level understanding of the data movement in the network during query processing.

For ease of exposition, we will describe communication in synchronized *iterations*, where at each iteration, each

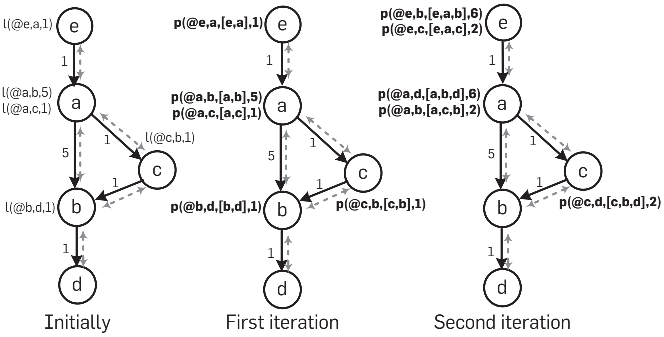


Figure 1. Nodes in the network are running the shortest-path program. We only show newly derived tuples at each iteration.

network node generates *paths* of increasing hop count, and then propagates these paths to neighbor nodes along links. We show only the derived paths communicated along the solid lines. In actual query execution, derived tuples can be sent along the bidirectional network links (dashed links).

In the first iteration, all nodes initialize their local path tables to 1-hop paths using rule `sp1`. In the second iteration, using rule `sp2`, each node takes the input paths generated in the previous iteration, and computes 2-hop paths, which are then propagated to its neighbors. For example, `path(@a,d,[a,b,d],6)` is generated at node *b* using `path(@b,d,[b,d],1)` from the first iteration, and propagated to node *a*. In fact, many network protocols propagate only the *nextHop* and avoid sending the entire path vector.

As paths are computed, the shortest one is incrementally updated. For example, node *a* computes the cost of the shortest path from `a` to `b` as 5 with rule `sp3`, and then finds the corresponding shortest path `[a,b]` with rule `sp4`. In the next iteration, node *a* receives `path(@a,b,[a,c,b],2)` from node `c`, which has lower cost compared to the previous shortest cost of 5, and hence `shortestPath(@a,b,[a,c,b],2)` replaces the previous tuple (the first two fields of source and destination are the primary key of this relation).

Interestingly, while *NDlog* is a language to describe networks, there are no explicit communication primitives. All communication is implicitly generated during rule execution as a result of data placement specifications. For example, in rule `sp2`, the `path` and `link` predicates have different location specifiers, and in order to execute the rule body of `sp2` based on their matching fields, `link` and `path` tuples have to be shipped in the network. It is the movement of these tuples that generates the messages for the resulting network protocol.

2.4. Language extensions

We describe two extensions to the *NDlog* language: *link-restricted rules* that limit the expressiveness of the language in order to capture physical network constraints, and a *soft-state storage model* commonly used in networking protocols.

Link-Restricted Rules: In the above path vector protocol, the evaluation of a rule must depend only on communication along the physical links. In order to send a message in a low-level network, there needs to be a link between the sender and receiver. This is not a natural construct in Datalog. Hence, to model physical networking components where full connectivity is not available, *NDlog* provides restrictions ensuring that rule execution results in communication only among nodes that are physically connected with a bidirectional link. This is syntactically achieved with the use of the special `link` predicate in the form of *link-restricted rules*. A *link-restricted* rule is either a *local* rule (having the same location specifier variable in each predicate), or a rule with the following properties:

1. There is exactly one `link` predicate in the body.
2. All other predicates (including the head predicate) have their location specifier set to either the first (source) or second (destination) field of the `link` predicate.

This syntactic constraint precisely captures the requirement that we be able to operate directly on a network whose link connectivity is not a full mesh. Further, as we demonstrate in Section 3, link-restriction also guarantees that all programs with only link-restricted rules can be rewritten into a canonical form where every rule body can be evaluated on a single node, with communication to a head predicate along links. The following is an example of a link-restricted rule:

```
p(@Dest,...) :- link(@Src,Dest...),p1(@Src,...),
                p2(@Src,...),..., pn(@Src,...).
```

The rule body of this example is executed at `@Src` and the resulting *p* tuples are sent to `@Dest`, preserving the communication constraints along links. Note that the body predicates of this example all have the same location specifier: `@Src`, the source of the link. In contrast, rule `sp2` of the *shortest path* program is link-restricted but has some relations whose location specifier is the source, and others whose location specifier is the destination; this needs to be rewritten to be executable in the network, a topic we return to in Section 3.2.

In a fully connected network environment, an *NDlog* parser can be configured to bypass the requirement for link-restricted rules.

Soft-State Storage Model: Many network protocols use the soft-state approach to maintain distributed state. In the soft-state storage model, stored data have an associated *lifetime* or time-to-live (TTL). A soft-state datum needs to be periodically refreshed; if more time than a TTL passes without a datum being refreshed, that datum is deleted. Soft state is often favored in networking implementations because in a very simple manner it provides well-defined eventual consistency semantics. Intuitively, periodic refreshes to network state ensure that the eventual values are obtained even if there are transient errors such as reordered messages, node disconnection, or link failures. However, when persistent failures occur, no coordination is required to register the failure: any data provided by failed nodes are organically "forgotten" in the absence of refreshes.

We introduced soft-state into the *Overlog*[21] declarative networking language, an extension of *NDlog*. One additional feature of *Overlog* is the availability of a `materialized` keyword at the beginning of each program to specify the TTL of predicates. For example, the definition `materialized(link, {1,2}, 10)` specifies that the `link` table has its primary key set to the first and second fields (denoted by `{1,2}`), and each `link` tuple has a lifetime of 10 seconds. If the TTL is set to infinity, the predicate will be treated as *hard state*, i.e., a traditional relation that does not involve timeout-based deletion.

The *Overlog* soft-state storage semantics are as follows. When a tuple is derived, if there exists another tuple with the same primary key but differences on other fields, an *update* occurs, in which the new tuple replaces the previous one. On the other hand, if the two tuples are identical, a *refresh* occurs, in which the existing tuple is extended by its TTL.

If a given predicate has no associated `materialize` declaration, it is treated as an *event* predicate: a soft-state predicate with TTL = 0. Event predicates are transient tables, which are used as input to rules but not stored. They are primarily used to "trigger" rules periodically or in response to network events. For example, utilizing *Overlog*'s built-in `periodic` event predicate, the following rule enables node `X` to generate a `ping` event every 10 seconds to its neighbor `Y` denoted in the `link(@X,Y)` predicate:

```
ping(@Y,X) :- periodic(@X,10),link(@X,Y).
```

Subtleties arise in the semantics of rules that mix event, soft-state and hard-state predicates across the head and body. One issue involves the expiry of soft-state and event tuples, as compared to deletion of hard-state tuples. In a traditional hard-state model, deletions from a rule's body relations require revisions to the derived head relation to maintain consistency of the rule. This is treated by research on materialized view maintenance.[13] In a pure soft-state model, the head and body predicates can be left inconsistent with each other for a time, until head predicates expire due to the lack of refreshes from body predicates. Mixtures of the two models become more subtle. We provided one treatment of this issue,[19] which has subsequently been revised with a slightly different interpretation.[9] There is still some debate about the desired semantics, focusing on attempts to provide an intuitive declarative representation while enabling familiar event-handler design patterns used by protocol developers.

3. EXECUTION PLAN GENERATION

Our runtime execution of *NDlog* programs differs from the traditional implementation patterns for both network protocols and database queries. Network protocol implementations often center around local state machines that emit messages, triggering state transitions at other state machines. By contrast, the runtime systems we have built for *NDlog* and *Overlog* are distributed dataflow execution engines, similar in spirit to those developed for parallel database systems, and echoed in recent parallel map-reduce implementations. However, the recursion in Datalog introduces cycles into these dataflows. The combination of recursive flows and the asynchronous communication inherent in wide-area systems presents new challenges that we had to overcome.

In this section, we describe the steps required to automatically generate a distributed dataflow execution plan from an *NDlog* program. We first focus on generating an execution plan in a centralized implementation, before extending the techniques to the network scenario.

3.1. Centralized plan generation

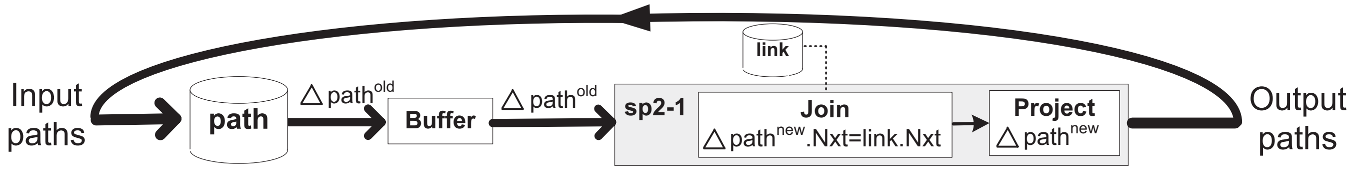
In generating the centralized plan, we utilize the well-known *semi-naïve fixpoint*[3] Datalog evaluation mechanism that ensures no redundant evaluations. As a quick review, in semi-naïve (SN) evaluation, input tuples computed in the previous iteration of a recursive rule execution are used as input in the current iteration to compute new tuples. Any new tuples that are generated for the first time in the current iteration, and only these new tuples, are then used as input to the next iteration. This is repeated until a fixpoint is achieved (i.e., no new tuples are produced).

The SN rewritten rule for rule `sp2` is shown below:

sp2-1 $\Delta path^{new}$(@Src,@Dest,Path,Cost) :-
link(@Src,Nxt,Cost1),
$\Delta path^{old}$(@Nxt,Dest,Path2,Cost2),
Cost=Cost1+Cost2,
Path=*f_concatPath*(Src,Path2).

Figure 2 shows the dataflow realization for a centralized implementation of rule `sp2-1` using the conventions of *P2*.[24]

Figure 2. Rule strand for a centralized implementation of rule sp2-1 in *P2*. Output paths that are generated from the strand are "wrapped back" as input into the same strand.



The *P2* system uses an execution model inspired by database query engines and the Click modular router,[14] which consists of elements that are connected together to implement a variety of network and flow control components. In addition, *P2* elements include database operators (such as joins, aggregation, selections, and projects) that are directly generated from the rules.

We will briefly explain how the SN evaluation is achieved in *P2*. Each SN rule is implemented as a *rule strand*. Each strand consists of a number of relational operators for selections, projections, joins, and aggregations. The example strand receives new `delta_path_old` tuples generated in the previous iteration to generate new paths (`delta_path_new`), which are then inserted into the path table (with duplicate elimination) for further processing in the next iteration.

In Algorithm 1, we show the pseudocode for a centralized implementation of multiple SN rule strands where each rule has the form:

$$\Delta p_j^{new} :- p_1^{old},\ldots,p_{k-1}^{old},\Delta p_k^{old},p_{k+1},\ldots,p_n,b_1,b_2,\ldots,b_m.$$

$p_1, \ldots, p_n$ are recursive predicates and $b_1, \ldots, b_m$ are base predicates. Δp_k^{old} refers to p_k tuples generated for the first time in the previous iteration. p_k^{old} refers to all p_k tuples generated before the previous iteration. These rules are logically equivalent to rules of the form:

$$\Delta p_j^{new} :- p_1,\ldots,p_{k-1},\Delta p_k^{old},p_{k+1},\ldots,p_n,b_1,b_2,\ldots,b_m.$$

The earlier rules have the advantage of avoiding redundant inferences within each iteration.

Algorithm 1 Semi-naïve (SN) Evaluation in *P2*

```
while $\exists B_k.size > 0$
    $\forall B_k$ where $B_k.size > 0$, $\Delta p_k^{old} \leftarrow B_k.flush()$
    execute all rule strands
    foreach recursive predicate $p_j$
        $p_j^{old} \leftarrow p_j^{old} \cup \Delta p_j^{old}$
        $B_j \leftarrow \Delta p_j^{new} - p_j^{old}$
        $p_j \leftarrow p_j^{old} \cup B_j$
        $\Delta p_j^{new} \leftarrow \phi$
```

In the algorithm, B_k denotes the buffer for p_k tuples generated in the previous iteration (Δp_k^{old}). Initially, p_k, p_k^{old}, Δp_k^{old}, and Δp_k^{new} are empty. As a base case, we execute all the rules to generate the initial p_k tuples, which are inserted into the corresponding B_k buffers. Each subsequent iteration of the while loop consists of flushing all existing Δp_k^{old} tuples from B_k and executing all rule strands to generate Δp_j^{new} tuples, which are used to update p_j^{old}, B_j, and p_j accordingly. Note that only new p_j tuples generated in the current iteration are inserted into B_j for use in the next iteration. Fixpoint is reached when all buffers are empty.

3.2. Distributed plan generation

In the distributed implementation of the path-vector program, nonlocal rules whose body predicates have different location specifiers cannot be executed at a single node, since the tuples that must be joined are situated at different nodes in the network. A *rule localization* rewrite step ensures that all tuples to be joined are at the same node. This allows a rule body to be locally computable.

Consider the rule `sp2` from the shortest-path program, where the `link` and `path` predicates have different location specifiers. These two predicates are joined by a common `@Nxt` address field. Figure 3 shows the corresponding logical query plan depicting the distributed join. The clouds represent an "exchange"-like operator[11] that forwards tuples from one network node to another; clouds are labeled with the `link` attribute that determines the tuple's recipient. The first cloud (`link.Nxt`) sends `link` tuples to the neighbor nodes indicated by their destination address fields, in order to join with matching `path` tuples stored by their source address fields. The second cloud (`path.Src`) transmits for further processing new `path` tuples computed from the join, setting the recipient according to the source address field.

Based on the above distributed join, rule `sp2` can be rewritten into the following two rules. Note that all predicates in the body of `sp2a` have the same location specifiers; the same is true of `sp2b`.

```
sp2a linkD(@Nxt,Src,Cost) :- link(@Src,Nxt,Cost).
sp2b path(@Src,Dest,Nxt,Path,Cost) :- linkD(@Nxt,Src,Cost1),
           path(@Nxt,Dest,Path2,Cost2),Cost=Cost1+Cost2,
           Path = f_concatPath(Src,Path2).
```

The rewrite is achievable because the `link` and `path` predicates, although at different locations, share a common join address field. The details of the rewrite algorithm and associated proofs are described in a longer article.[20]

Returning to our example, after rule localization we perform the SN rewrite, and then generate the rule strands shown in Figure 4. Unlike the centralized strand in Figure 2, there are now three rule strands. The extra two strands (`sp2a@Src` and `sp2b-2@Nxt`) are used as follows. Rule strand `sp2a@`

Figure 3. Logical query plan for rule sp2.

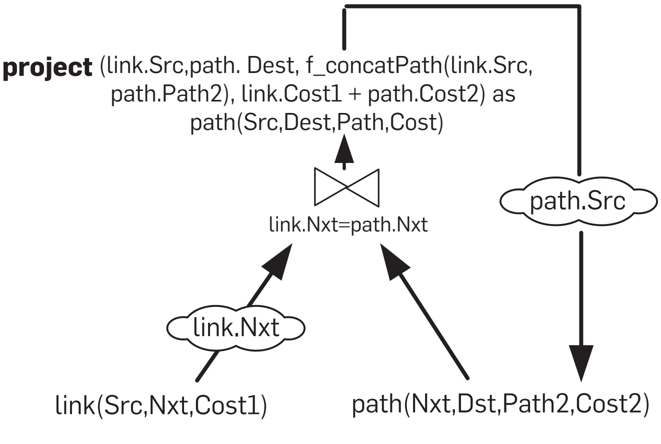
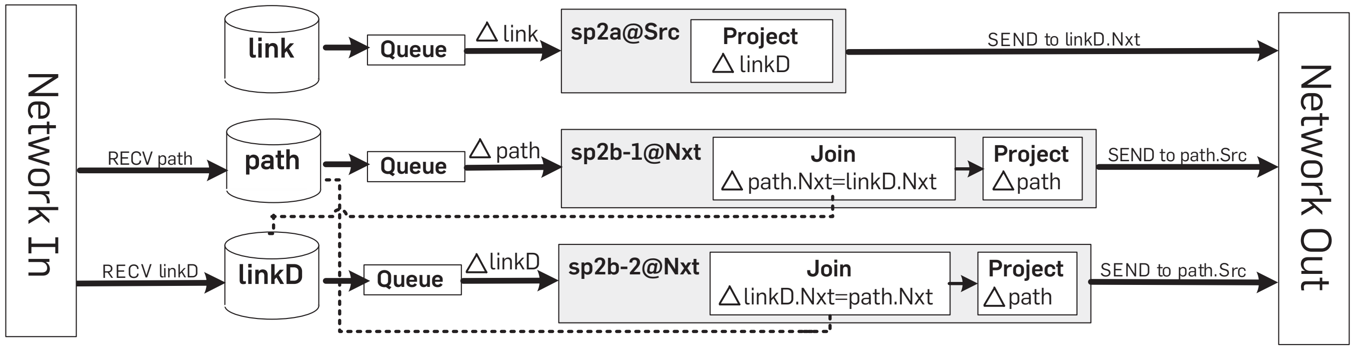

project (link.Src,path. Dest, f_concatPath(link.Src, path.Path2), link.Cost1 + path.Cost2) as path(Src,Dest,Path,Cost)
link.Nxt=path.Nxt
path.Src
link.Nxt
link(Src,Nxt,Cost1)
path(Nxt,Dst,Path2,Cost2)

Figure 4. Rule strands for the distributed version of sp2 after localization in *P2*.



`Src` sends all existing links to the destination address field as `linkD` tuples. Rule strand `sp2b-2@Nxt` takes the new `linkD` tuples it received via the network and performs a join operation with the local `path` table to generate new paths.

3.3. Relaxing semi-naïve evaluation

In our distributed implementation, the execution of rule strands can depend on tuples arriving via the network, and can also result in new tuples being sent over the network. Traditional SN evaluation completely evaluates all rules on a given set of facts, i.e., completes the *iteration*, before considering any new facts. In a distributed execution environment where messages can be delayed or lost, the completion of an iteration in the traditional sense can only be detected by a consensus computation across multiple nodes, which is expensive; further, the requirement that many nodes complete the iteration together (a "barrier synchronization" in parallel computing terminology) limits parallelism significantly by restricting the rate of progress to that of the slowest node.

We address this by making the notion of iteration local to a node. New facts might be generated through local rule execution, or might be received from another node while a local iteration is in progress. We proposed and proved correct a variation of SN iteration called *pipelined semi-naïve* (PSN) to handle this situation.[20] PSN extends SN to work in an asynchronous distributed setting. PSN relaxes SN evaluation to the extreme of processing each tuple as it is received. This provides opportunities for additional optimizations on a per-tuple basis. New tuples that are generated from the SN rules, as well as tuples received from other nodes, are used immediately to compute new tuples without waiting for the current (local) iteration to complete.

Algorithm 2 Pipelined Semi-naïve (PSN) Evaluation

```
while $\exists\ Q_k.size > 0$
    $t_k^{old,i} \leftarrow Q_k.dequeueTuple()$
    foreach rule strand execution
        $\Delta p_j^{new,i+1}$ :—
            $p_1,\ldots,p_{k-1}, t_k^{old,i}, p_{k+1},\ldots, p_n, b_1, b_2,\ldots, b_m$
        foreach $t_j^{new,i+1} \in \Delta p_j^{new,i+1}$
          if $t_j^{new,i+1} \notin p_j$
            then $p_j \leftarrow p_j \cup t_j^{new,i+1}$
                 $Q_j.enqueueTuple(t_j^{new,i+1})$
```

Algorithm 2 shows the pseudocode for PSN. Each tuple, denoted *t*, has a superscript (*old*/*new*, *i*) where *i* is its corresponding iteration number in SN evaluation. Each processing step in PSN consists of dequeuing a tuple $t_k^{old,i}$ from Q_k and then using it as input into all corresponding rule strands. Each resulting $t_j^{new,i+1}$ tuple is pipelined, stored in its respective p_j table (if a copy is not already there), and enqueued into Q_j for further processing. Note that in a distributed implementation. Q_j can be a queue on another node, and the node that receives the new tuple can immediately process the tuple after the enqueue into Q_j. For example, the dataflow in Figure 4 is based on a distributed implementation of PSN, where incoming `path` and `linkD` tuples received via the network are stored locally, and enqueued for processing in the corresponding rule strands.

To fully pipeline evaluation, we have also removed the distinctions between p_j^{old} and p_j in the rules. Instead, a timestamp (or monotonically increasing sequence number) is added to each tuple at arrival, and the join operator matches each tuple only with tuples that have the same or older timestamp. This allows processing of tuples immediately upon arrival, and is natural for network message handling. This represents an alternative "book-keeping" strategy to the rewriting used in SN to ensure no repeated inferences. Note that the timestamp only needs to be assigned locally, since all the rules are localized.

We have proven elsewhere[20] that PSN generates the same results as SN and does not repeat any inferences, as long as the *NDlog* program is monotonic and messages between two network nodes are delivered in FIFO order.

3.4. Incremental maintenance

In practice, most network protocols are executed over a long period of time, and the protocol incrementally updates and repairs routing tables as the underlying network changes (link failures, node departures, etc.). To better map into practical networking scenarios, one key distinction that differentiates the execution of *NDlog* from earlier work in Datalog is our support for continuous rule execution and result materialization, where all tuples derived from *NDlog* rules are materialized and incrementally updated as the underlying network changes. As in network protocols, such incremental maintenance is required both for timely updates and for avoiding the overhead of recomputing all routing tables "from scratch" whenever there are changes to the underlying network. In the presence of insertions and deletions to base tuples, our original incremental view maintenance implementation utilizes the count algorithm[13] that ensures only tuples that are no longer derivable are deleted. This has subsequently been improved[18] via the use of a compact form of data provenance encoded using binary decision diagrams shipped with each derived tuple.

In general, updates could occur very frequently, at a period that is shorter than the expected time for a typical query to reach a fixpoint. In that case, query results can never fully reflect the state of the network. We focus our analysis instead on a *bursty* model. In this weaker, but still fairly realistic model, updates are allowed to happen during query processing. However, we make the assumption that after a burst of updates, the network eventually *quiesces* (does not change) for a time long enough to allow all the queries in the system to reach a fixpoint. Unlike the continuous model, the bursty model is amenable to simpler analysis; our results on that model provide some intuition as to the behavior in the continuous update model as well.

We have proven[20] that in the presence of reliable, in-order delivery of messages, link-restricted *NDlog* rules under the bursty model achieve a variant of the typical distributed systems notion of *eventual consistency*, where the eventual state of the quiescent system corresponds to what would be achieved by rerunning the queries from scratch in that state.

4. USE CASES

In the past 3 years, since the introduction of declarative networking and the release of *P2*, several applications have been developed. We describe two of the original use cases that motivated our work and drove several of our language and system designs: *safe extensible routers* and *overlay network development*. We will briefly mention new applications in Section 5.

4.1. Declarative routing

The Internet's core routing infrastructure, while arguably robust and efficient, has proven to be difficult to evolve to accommodate the needs of new applications. Prior research on this problem has included new hard-coded routing protocols on the one hand, and fully extensible Active Networks[31] on the other. *Declarative routing*[21] explores a new point in this design space that aims to strike a better balance between the extensibility and robustness of a routing infrastructure.

With declarative routing, a routing protocol is implemented by writing a simple query in *NDlog*, which is then executed in a distributed fashion at the nodes that receive the query. Declarative routing can be viewed as a restrictive instantiation of Active Networks for the control plane, which aims to balance the concerns of expressiveness, performance and security, properties which are needed for an extensible routing infrastructure to succeed.

Security is a key concern with any extensible system particularly when it relates to nontermination and the consumption of resources. *NDlog* is amenable to static analysis due to its connections to Datalog. In terms of query execution, *pure* Datalog (without any negation, aggregation, or function symbols) has polynomial time and space complexities in the size of the input. This property provides a natural bound on the resource consumption. However, many extensions of Datalog (including *NDlog*) augment the core language in various ways, invalidating its polynomial complexity.

Fortunately, static analysis tests have been developed to check for the termination of an augmented Datalog query on a given input.[15] In a nutshell, these tests identify recursive definitions in the query rules, and check whether these definitions terminate. Examples of recursive definitions that terminate are ones that evaluate monotonically increasing (decreasing) predicates whose values are upper (lower) bounded. Moreover, the declarative framework is amenable to other verification techniques, including theorem proving,[32] model checking,[25] and runtime verification.[28]

NDlog can express a variety of well-known routing protocols (e.g., distance vector, path vector, dynamic source routing, link state, multicast) in a compact and clean fashion, typically in a handful of lines of program code. Moreover, higher-level routing concepts (e.g., QoS constraints) can be achieved via simple modifications to these queries. Finally, writing the queries in *NDlog* illustrates surprising relationships between protocols. For example, we have shown that *distance vector* and *dynamic source routing* protocols differ only in a simple, traditional query optimization decision: the order in which a query's predicates are evaluated.

To limit query computation to the relevant portion of the network, we use a query rewrite technique, called *magic sets rewriting*.[4] Rather than reviewing the Magic Sets optimization here, we illustrate its use in an example. Consider the situation where instead of computing all-pairs shortest paths, we are only in computing the shortest paths from a selected group of source nodes (`magicSrc`) to selected destination nodes (`magicDst`). By modifying rules `sp1–sp4` from the path-vector program, the following computes only paths limited to sources/destinations in the `magicSrc/ magicDst` tables, respectively.

```
sp1-sd pathDst(@Dest,Src,Path,Cost) :- magicSrc(@Src),
       link(@Src,Dest,Cost), Path=f_init(Src,Dst).
sp2-sd pathDst(@Dst,Src,Path,Cost) :-
       pathDst(@Nxt,Src,Path1,Cost1),link(@Nxt,Dest,Cost2),
       Cost=Cost1+Cost2, Path=f_concatPath(Path1,Dest).
sp3-sd spCost(@Dest,Src,min<Cost>) :- magicDst(@Dest),
       pathDst(@Dest,Src,Path,Cost).
sp4-sd shortestPath(@Dest,Src,Path,Cost) :-
       spCost(@Dest,Src,Cost), pathDst(@Dest,Src,Path,Cost).
Query shortestPath(@Src,Dest,Path,Cost).
```

Our evaluation results[21] based on running declarative routing protocols on the PlanetLab[26] global testbed and in a local cluster show that when all nodes issue the same query, the query execution has similar scalability properties as the traditional distance vector and path-vector protocols. For example, the convergence latency for the path-vector program is proportional to the network diameter, and converges in the same time as the path-vector protocol. Second, the per-node communication overhead increases linearly with the number of nodes. This suggests that our approach does not introduce any fundamental overheads. Moreover, when there are few nodes issuing the same query, query optimization and work-sharing techniques can significantly reduce the communication overhead.

One promising direction stems from our surprising observation on the synergies between query optimization and network routing: a wired protocol (distance-vector protocol) can be translated to a wireless protocol (dynamic source routing) by applying the standard database optimizations of magic sets rewrite and predicate reordering. More complex applications of query optimization have begun to pay dividends in research, synthesizing new hybrid protocols from traditional building blocks.[5, 17] Given the proliferation of new routing protocols and a diversity of new network architecture proposals, the connection between query optimizations and network routing suggests that query optimizations may help us inform new routing protocol designs and allow the hybridization of protocols within the network.

4.2. Declarative overlays

In declarative routing, we demonstrated the flexibility and compactness of *NDlog* for specifying a variety of routing protocols. In practice, most distributed systems are much more complex than simple routing protocols; in addition to routing, they typically also perform application-level message forwarding and handle the formation and maintenance of a network as well.

In our subsequent work on *declarative overlays*,[21] we demonstrate the use of the *Overlog* to implement practical application-level *overlay networks*. An overlay network is a virtual network of nodes and logical links that is built on top of an existing network with the purpose of implementing a network service that is not available in the existing network. Examples of overlay networks on today's Internet include commercial content distribution networks,[1] peer-to-peer (P2P) applications for file-sharing[10] and telephony,[29] as well as a wide range of experimental prototypes running on PlanetLab.

In declarative overlays, applications submit to *P2* a concise *Overlog* program that describes an overlay network, and the *P2* system executes the program to maintain routing tables, perform neighbor discovery and provide forwarding for the overlay.

Declarative overlay programs are more complex than routing due to the handling of message delivery, acknowledgments, failure detection, and timeouts. These programs also heavily utilize soft-state features in *Overlog* not present in the original *NDlog* language. Despite the increased complexity, we demonstrate that our *NDlog* programs are significantly more compact compared to equivalent C++ implementations. For instance, the Narada[7] mesh formation and a full-fledged implementation of the Chord distributed hash table[30] are implemented in 16 and 48 rules, respectively. In the case of the Chord DHT presented by Loo et al.,[19] there are rules for performing various aspects of Chord, including initial joining of the Chord network, Chord ring maintenance, finger table maintenance, recursive Chord lookups, and failure detection of neighbors.

We note that our Chord implementation is roughly two orders of magnitude less code than the original C++ implementation. This is a quantitative difference that is sufficiently large that it becomes qualitative: in our opinion (and experience), declarative programs that are a few dozen lines of code are markedly easier to understand, debug, and extend than thousands of lines of imperative code. Moreover, we demonstrate[19, 21] that our declarative overlays achieve the expected high-level properties of their respective overlay networks for both static and dynamic networks. For example, in a static network of up to 500 nodes, the measured hop-count of lookup requests in the Chord network conformed to the theoretical average of $0.5 \times \log_2 N$ hops, and the latency numbers were within the same order of magnitude as published Chord numbers.

5. CONCLUSION

In Jim Gray's Turing Award Lecture,[12] one of his grand challenges was the development of "automatic programming" techniques that would be (a) 1000× easier for people to use, (b) directly compiled into working code, and (c) suitable for general purpose use. Butler Lampson reiterated the first two points in a subsequent invited article, but suggested that they might be more tractable in domain-specific settings.[16]

Declarative Networking has gone a long way toward Gray's vision, if only in the domain of network protocol implementation. On multiple occasions we have seen at least two orders of magnitude reduction in code size, with the reduced linecount producing qualitative improvements. In the case of Chord, a multi-thousand-line C++ library was rewritten as a declarative program that fits on a single sheet of paper—a software artifact that can be studied and holistically understood by a programmer in a single sitting.

We have found that a high-level declarative language not only simplifies a programmer's work, but refocuses the programming task on appropriately high-level issues. For example, our work on declarative routing concluded that discussions of routing in wired vs. wireless networks should not result in different protocols, but rather in different compiler optimizations for the same simple declaration, with the potential to be automatically blended into new hybrid strategies as networks become more diverse.[5, 17] This lifting of abstractions seems well suited to the increasing complexity of modern networking, introducing software malleability by minimizing the affordances for over-engineering solutions to specific settings.

Since we began our work on this topic, there has been increasing evidence that declarative, data-centric programming has much broader applicability. Within the networking domain, we have expanded in multiple directions from our initial work on routing, to encompass low-level network issues at the wireless link layer[6] to higher-level logic including both overlay networks[21] and applications like code dissemination, object tracking, and content distribution. Meanwhile, a variety of groups have been using declarative programming ideas in surprising ways in many other domains. We briefly highlight two of our own follow-on efforts.

Secure Distributed Systems: Despite being developed independently by separate communities, logic-based security specifications and declarative networking programs both extend Datalog in surprisingly similar ways: by supporting the notion of context (location) to identify components (nodes) in distributed systems. The *Secure Network Datalog*[33] language extends *NDlog* with basic security constructs for implementing secure distributed systems, which are further enhanced with type checking and meta-programmability in the *LBTrust*[23] system for supporting various forms of encryption/authentication, delegation, for distributed trust management.

Datacenter Programming: The BOOM[2] project is exploring the use of declarative languages in the setting of Cloud Computing. Current cloud platforms provide developers with sequential programming models that are a poor match for inherently distributed resources. To illustrate the benefits of declarative programming in a cloud, we used *Overlog* as the basis for a radically simplified and enhanced reimplementation of a standard cloud-based analytics stack: the Hadoop File System (HDFS) and MapReduce infrastructure. Our resulting system is API-compatible with Hadoop, with performance that is equivalent or better. More significantly, the high-level *Overlog* specification of key Hadoop internals enabled a small group of graduate students to quickly add sophisticated distributed features to the system that are not in Hadoop: hot standby master nodes supported by MultiPaxos consensus, scaleout of (quorums of) master nodes via data partitioning, and implementations of new scheduling protocols and query processing strategies.

In addition to these two bodies of work, others have successfully adopted concepts from declarative networking, in the areas of mobility-based overlays, adaptively hybridized mobile ad-hoc networks, overlay network composition, sensor networking, fault-tolerant protocols, network configuration, replicated filesystems, distributed machine learning algorithms, and robotics. Outside the realm of networking and distributed systems, there has been an increasing use of declarative languages—many rooted in Datalog—to a wide range of problems including natural language processing, compiler analysis, security, and computer games. We maintain a list of related declarative languages and research projects at http://declarativity.net/related.

For the moment, these various efforts represent individual instances of Lampson's domain-specific approach to Gray's automatic programming challenge. In the coming years, it will be interesting to assess whether these solutions prove fruitful, and whether it is feasible to go after Gray's challenge directly: to deliver an attractive general-purpose declarative programming environment that radically simplifies wide range of tasks.

References

1. Akamai. Akamai Content Distribution Network. 2006. http://www.akamai.com.
2. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.C. BOOM: Data-centric programming in the datacenter. Technical Report UCB/EECS-2009-98, EECS Department, University of California, Berkeley, Jul 2009.
3. Balbin I., Ramamohanarao, K. A generalization of the differential approach to recursive query evaluation. *J. Logic Prog. 4*, 3 (1987), 259–262.
4. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J. Magic sets and other strange ways to implement logic programs. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1986).
5. Chu, D., Hellerstein, J. Automating rendezvous and proxy selection in sensor networks. In *Eighth International Conference on Information Processing in Sensor Networks (IPSN)* (2009).
6. Chu, D.C., Popa, L., Tavakoli, A., Hellerstein, J.M., Levis, P., Shenker, S., Stoica, I. The design and implementation of a declarative sensor network system. In *5th ACM Conference on Embedded Networked Sensor Systems (SenSys)* (2007).
7. Chu, Y.-H., Rao, S.G., Zhang, H. A case for end system multicast. In *Proceedings of ACM SIGMETRICS* (2000), 1–12.
8. Clark, D.D. The design philosophy of the DARPA internet protocols. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (Stanford, CA, 1988), ACM, 106–114.
9. Condie, T., Chu, D., Hellerstein, J.M., Maniatis, P. Evita raced: metacompilation for declarative networks. In *Proceedings of VLDB Conference* (2008).
10. Gnutella. http://www.gnutella.com.
11. Graefe, G. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1990).
12. Gray, J. What next? A few remaining problems in information technlogy, SIGMOD Conference 1999. ACM Turing Award Lecture, Video. *ACM SIGMOD Digital Symposium Collection 2*, 2 (2000).
13. Gupta, A., Mumick, I.S., Subrahmanian, V.S. Maintaining views incrementally. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1993).
14. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F. The click modular router. *ACM Trans. Comp. Sys. 18*, 3 (2000), 263–297.
15. Krishnamurthy, R., Ramakrishnan, R., Shmueli, O. A framework for testing safety and effective computability. *J. Comp. Sys. Sci. 52*, 1 (1996), 100–124.
16. Lampson, B. Getting computers to understand. *J. ACM 50*, 1 (2003), 70–72.
17. Liu, C., Correa, R., Li, X., Basu, P., Loo, B.T., Mao, Y. Declarative Policy-based adaptive MANET routing. In *17th IEEE International Conference on Network Protocols (ICNP)* (2009).
18. Liu, M., Taylor, N., Zhou, W., Ives, Z., Loo, B.T. Recursive computation of regions and connectivity in networks. In *Proceedings of IEEE Conference on Data Engineering (ICDE)* (2009).
19. Loo, B.T. The Design and Implementation of Declarative Networks (Ph.D. Dissertation). Technical Report UCB/EECS-2006-177, UC Berkeley (2006).
20. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I. Declarative networking: language, execution and optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (2006).
21. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I. Implementing declarative overlays. In *Proceedings of ACM Symposium on Operating Systems Principles* (2005).
22. Loo, B.T., Hellerstein, J.M., Stoica, I., Ramakrishnan, R. Declarative routing: extensible routing with declarative queries. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (2005).
23. Marczak, W.R., Zook, D., Zhou, W., Aref, M., Loo, B.T. Declarative reconfigurable trust management. In *Proceedings of Conference on Innovative Data Systems Research (CIDR)* (2009).
24. P2: Declarative Networking System. http://p2.cs.berkeley.edu.
25. Perez, J.N., Rybalchenko, A., Singh, A. Cardinality abstraction for declarative networking applications. In *Proceedings of Computer Aided Verification (CAV)* (2009).
26. PlanetLab. Global testbed. 2006. http://www.planet-lab.org/.
27. Ramakrishnan, R., Ullman, J.D. A survey of research on deductive database systems. *J. Logic Prog. 23*, 2 (1993), 125–149.
28. Singh, A., Maniatis, P., Roscoe, T., Druschel, P. Distributed monitoring and forensics in overlay networks. In *Proceedings of Eurosys* (2006).
29. Skype. Skype P2P Telephony. 2006. http://www.skype.com.
30. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H. Chord: a scalable P2P lookup service for internet applications. In *SIGCOMM* (2001).
31. Tennenhouse, D.L., Smith, J.M., Sincoskie, W.D., Wetherall, D.J., Minden, G.J. A survey of active network research. *IEEE Commun. Mag. 35*, 1 (1997), 80–86.
32. Wang, A., Basu, P., Loo, B.T., Sokolsky, O. Towards declarative network verification. In *11th International Symposium on Practical Aspects of Declarative Languages (PADL)* (2009).
33. Zhou, W., Mao, Y., Loo, B.T., Abadi, M. Unified declarative platform for secure networked information systems. In *Proceedings of IEEE Conference on Data Engineering (ICDE)* (2009).

Boon Thau Loo (boonloo@cis.upenn.edu) University of Pennsylvania, Philadelphia, PA.

Tyson Condie (tcondie@cs.berkeley.edu) University of California, Berkeley, CA.

Minos Garofalakis (minos@softnet.tuc.gr) Technical University of Crete, Greece.

David E. Gay (david.e.gay@intel.com) Intel Research, Berkeley, CA.

Joseph M. Hellerstein (hellerstein@cs.berkeley.edu) University of California, Berkeley, CA.

Petros Maniatis (petros.maniatis@intel.com) Intel Research, Berkeley, CA.

Raghu Ramakrishnan (ramakris@yahoo-inc.com) Yahoo! Research, Silicon Valley.

Timothy Roscoe (troscoe@inf.ethz.ch) ETH Zurich, Switzerland.

Ion Stoica (istoica@cs.berkeley.edu) University of California, Berkeley, CA.