1-1-2010

Efficient Querying and Maintenance of Network Provenance at Internet-Scale

Wenchao Zhou
*University of Pennsylvania*

Micah Sherr
*University of Pennsylvania, msherr@cis.upenn.edu*

Tao T
*University of Pennsylvania*

Xiaozhou Li
*University of Pennsylvania*

Boon Thau Loo
*University of Pennsylvania, boonloo@cis.upenn.edu*

*See next page for additional authors*

Follow this and additional works at: [http://repository.upenn.edu/cis_reports](http://repository.upenn.edu/cis_reports)

**Recommended Citation**
Wenchao Zhou, Micah Sherr, Tao T, Xiaozhou Li, Boon Thau Loo, and Yun Mao, "Efficient Querying and Maintenance of Network Provenance at Internet-Scale", January 2010.


This paper is posted at ScholarlyCommons, [http://repository.upenn.edu/cis_reports/922](http://repository.upenn.edu/cis_reports/922)
For more information, please contact libraryrepository@pobox.upenn.edu.
Efficient Querying and Maintenance of Network Provenance at Internet-Scale

Abstract
Network accountability, forensic analysis, and failure diagnosis are becoming increasingly important for network management and security. Such capabilities often utilize network provenance -- the ability to issue queries over network meta-data. For example, network provenance may be used to trace the path a message traverses on the network as well as to determine how message data were derived and which parties were involved in its derivation.

This paper presents the design and implementation of ExSPAN, a generic and extensible framework that achieves efficient network provenance in a distributed environment. We utilize the database notion of data provenance to "explain" the existence of any network state, providing a versatile mechanism for network provenance. To achieve such flexibility at Internet-scale, ExSPAN uses declarative networking in which network protocols can be modeled as continuous queries over distributed streams and specified concisely in a declarative query language. We extend existing data models for provenance developed in database literature to enable distribution at Internet-scale, and investigate numerous optimization techniques to maintain and query distributed network provenance efficiently. The ExSPAN prototype is developed using RapidNet, a declarative networking platform based on the emerging ns-3 toolkit. Experiments over a simulated network and an actual deployment in a testbed environment demonstrate that our system supports a wide range of distributed provenance computations efficiently, resulting in significant reductions in bandwidth costs compared to traditional approaches.

Comments

Author(s)
Wenchao Zhou, Micah Sherr, Tao T, Xiaozhou Li, Boon Thau Loo, and Yun Mao

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/922
Efficient Querying and Maintenance of Network Provenance at Internet-Scale

Wenchao Zhou∗  Micah Sherr†  Tao Tao∗  Xiaozhou Li∗  Boon Thau Loo∗  Yun Mao∗
∗University of Pennsylvania  †AT&T Research Labs
{wenchaoz, msherr, taot, xiaozhou, boonloo}@cis.upenn.edu, maoy@research.att.com

ABSTRACT
Network accountability, forensic analysis, and failure diagnosis are becoming increasingly important for network management and security. Such capabilities often utilize network provenance – the ability to issue queries over network meta-data. For example, network provenance may be used to trace the path a message traverses on the network as well as to determine how message data were derived and which parties were involved in its derivation.

This paper presents the design and implementation of ExSPAN, a generic and extensible framework that achieves efficient network provenance in a distributed environment. We utilize the database notion of data provenance to "explain" the existence of any network state, providing a versatile mechanism for network provenance. To achieve such flexibility at Internet-scale, ExSPAN uses declarative networking in which network protocols can be modeled as continuous queries over distributed streams and specified concisely in a declarative query language. We extend existing data models for provenance developed in database literature to enable distribution at Internet-scale, and investigate numerous optimization techniques to maintain and query distributed network provenance efficiently. The ExSPAN prototype is developed using RapidNet, a declarative networking platform based on the emerging ns-3 toolkit. Experiments over a simulated network and an actual deployment in a testbed environment demonstrate that our system supports a wide range of distributed provenance computations efficiently, resulting in significant reductions in bandwidth costs compared to traditional approaches.

Categories and Subject Descriptors
C.2.4 [Computer Systems Organization]: Computer Communication Networks—Distributed Systems; E.1 [Data Structures]: Distributed Data Structures; H.2.1 [Database Management]: Logical Design—Data Models

General Terms
Design, Management, Performance

1. INTRODUCTION
In database systems, data provenance [4] is a well-known concept, primarily used to answer questions concerning how query results are derived and which data sources they come from. A similar notion – network provenance [25] – is emerging in the networking domain. Network provenance describes the history and derivations of network state resulting from the execution of a distributed protocol. Typical network provenance use cases include discerning the origination of a message, the path that the message traversed on the network, and how communicated data were derived and which parties were involved in its derivation.

The capability to learn such information is essential to a diverse set of network management tasks such as performing network diagnostics, identifying malicious and misbehaving users, and enforcing trust management policies in distributed systems. Each goal has led to a series of application-specific proposals [21, 1, 11, 24, 9, 12] that focus on improving network support for accountability and providing efficient mechanisms to trace packets and information flows through the Internet.

This paper explores the generic data management challenges posed by the distribution, querying, and maintenance of network provenance at Internet-scale. Such scale has presented a unique challenge to provenance data management. Traditionally, provenance data are either stored in a centralized server or shared across only tens of nodes. In contrast, network applications in Internet domains usually involve thousands of nodes. Moreover, provenance computations are required to co-exist with existing network protocols. Bandwidth efficiency and minimal impact on convergence times are of significant importance.

As a step towards meeting these challenges, this paper proposes ExSPAN (EXtenSible Provenance Aware Networked systems), a platform that enables generic distributed systems to be specified, implemented, and analyzed with built-in distributed provenance support. ExSPAN provides a flexible framework for distributed querying of network meta-data. The type of network provenance ExSPAN provides can be customized along the dimensions of granularity (domains defined according to nodes, tuples, or networks), distribution (centralized or distributed), and representation (using derivation trees, binary decision diagrams [2], algebraic structures, etc.). We show that various distributed systems (in particular, diagnostics, network debugging, and distributed trust management) map naturally to network provenance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.

The name ExSPAN is inspired by the system’s use of provenance, which enables network state to be expanded and analyzed.
This paper makes the following contributions:

**Data model for network provenance.** We define a distributed data model for storing network provenance. Our data model builds upon current work on representing provenance information as relational tables [10, 5], with extensions to supported distributed storage and querying. We propose two forms of distribution: a value-based approach in which all relevant information is piggy-backed onto communicated tuples, and a bandwidth-efficient reference-based approach that lazily creates provenance markers (or pointers) that can be resolved on demand via a distributed query.

**Efficient provenance maintenance and querying.** To maintain network provenance efficiently, we leverage the distributed query processing capabilities of declarative networking [17, 16, 15]. Declarative networking models network protocols as continuous queries over distributed streams. Declarative networking programs permit a variety of distributed network protocols to be specified concisely in a declarative query language. Given a declarative networking program, we demonstrate an automatic rewrite strategy that will augment the original program with additional queries for maintaining provenance information for the network protocol. Moreover, additional distributed queries can be formulated to derive various representations of network provenance, hence achieving a unifying framework for synthesizing and analyzing distributed systems. We further propose a variety of query optimization techniques aimed at reducing communication latency and bandwidth utilization.

**ExSPAN prototype implementation and evaluation.** We present the prototype of ExSPAN. Our implementation utilizes RapidNet [18], a declarative networking platform developed using the ns-3 network simulator [19]. Our experiments over simulated networks and an actual deployment on a testbed environment demonstrate that ExSPAN supports a wide range of distributed provenance computations efficiently, resulting in significant reduction in bandwidth utilization compared with centralized approaches.

The remainder of this paper is organized as follows. In Section 2, we present a background introduction to declarative networking. In Section 3, we then present a taxonomy of provenance along three axes and outline various use cases in distributed systems analysis. Based on the taxonomy, Section 4 presents the data model for distributed provenance, and declarative networking queries for maintaining provenance information in a distributed fashion. Section 5 demonstrates a similar use of declarative networking to query for various representations of network provenance. Section 6 presents various optimization techniques. Our evaluation results in simulation and on an actual testbed are presented in Section 7. We then conclude with related work (Section 8) and future work (Section 9).

## 2. BACKGROUND

Given ExSPAN’s use of declarative networking, we briefly introduce declarative networking and the query language that will be used as a basis for enabling network provenance. The high level goal of declarative networks [17, 16, 15] is to build extensible network architectures that achieve a good balance of flexibility, performance, and safety. Declarative networks are specified using Network Datalog (NDlog), a distributed recursive query language used for querying network graphs. NDlog queries are executed using a distributed query processor to implement the network protocols and are continuously maintained as distributed views over existing network and host state. Declarative queries such as NDlog are a natural and compact way to implement a variety of routing protocols and overlay networks. For example, traditional routing protocols can be expressed in a few lines of code [17], and the Chord [23] distributed hash table in 47 lines of code [16]. When compiled and executed, these declarative networks perform efficiently relative to imperative implementations.

The techniques proposed in this paper can be generally realized using any sufficiently expressive distributed query processor. The advantage of using declarative networking is that several robust implementations exist that can be straightforwardly leveraged to develop ExSPAN. Moreover, since distributed protocols can themselves be expressed as declarative statements, declarative networking represents a natural means for unifying the synthesis and analysis of distributed protocols.

The declarative NDlog language used by ExSPAN is based on Datalog [20]. A Datalog program consists of a set of rules. Each rule has the form \( p :- q_{1}, q_{2}, \ldots, q_{n} \), which can be read informally as “if \( q_{1} \) and \( q_{2} \) and ... and \( q_{n} \) are true then \( p \)”. Here, \( p \) 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 with attributes (which are bound to variables or constants by the query) or Boolean expressions that involve function symbols (including arithmetic) applied to attributes. Predicates in NDlog are typically relations, although in some cases they may represent functions. Commas are interpreted as logical conjunctions (AND). The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter.

NDlog is a distributed variant of traditional Datalog, primarily designed for expressing distributed (recursive) computations. NDlog supports a location specifier in each predicate, expressed with the \( @ \) symbol followed by an attribute. This attribute denotes the location of each corresponding tuple.

### Figure 1: The MINCOST program in NDlog

For example, consider the three-rule MINCOST program shown in Figure 1. MINCOST computes the best path cost between each pair of nodes in a network. Rules \( sp1 \) and \( sp2 \) specify the definition of the derived tuple \( pathCost \). Rule \( sp1 \) computes all one-hop path cost based on the base tuples from the \( link \) relation. Rule \( sp2 \) expresses that “if there is a link from \( S \) to \( D \) of cost \( C \), and the best path cost from \( S \) to \( D \) is \( C1 \), then there is a path from \( S \) to \( D \) with cost \( C1+C2 \)” (we assume links are symmetric, i.e. if there is a link from \( S \) to \( D \) with cost \( C \), then a link from \( D \) to \( S \) with the same cost \( C \) also exists). Rule \( sp3 \) aggregates all paths with the same pair of source and destination to compute the best path cost. By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols.

When executed, MINCOST forms a distributed stream computation where streams of \( link \), \( pathCost \), and \( bestPathCost \) tuples are joined at different nodes to compute the best path costs. To maintain and derive tuples as the inputs to the rules are updated (e.g., \( link \) tuples are inserted), these queries are continuously executed. For more details on the incremental maintenance of declarative networking protocols, refer to references [13, 15].

### Figure 2: The PACKETFORWARD program in NDlog

NDlog also supports event predicates (that is, tables for transient state). Events can trigger rule executions but are not materialized. By convention, event predicate names start with “e”. The PACKETFORWARD program in Figure 2 illustrates how to use event predicates. Upon receiving an event \( ePacket \), the next hop is found
by joining the bestHop table. The packet is then sent as another ePacket event to the next hop.

3. NETWORK PROVENANCE

ExSPAN is a generic and customizable framework that enables a variety of types of network provenance, which can be categorized along three orthogonal axes: (i) granularity, which reflects the detail level of the provenance maintained for derived tuples; (ii) representation, which defines how provenance is encoded internally; and (iii) distribution, which describes how provenance is distributed.

Granularity. ExSPAN provides three levels of granularity for provenance encoding. *Tuple-level* provenance maximizes provenance detail by encoding all intermediary tuples used in a given derivation. For instance, the ability to trace a tuple’s construction makes tuple-level provenance a useful tool for debugging network protocols. As an example of tuple-level provenance, consider the network topology depicted in Figure 3 and the corresponding provenance graph for bestPathCost(@a,c,5) (computed using MinCost) shown in Figure 4. The tuple-level provenance for bestPathCost(@a,c,5) consists of all nodes and edges in the graph. In general, tuple-level provenance encodes the maximum amount of information, but incurs the largest communication overhead.

ExSPAN also supports *node-level* provenance in which provenance encodes only the nodes that are involved at each step of the derivation. For example, the node-level representation of bestPathCost(@a,c,5) is \((a, b \rightarrow a)\), reflecting the nodes along the two derivation paths. Node-level provenance is a useful means to determine which elements of the network are responsible for a given tuple.

Finally, ExSPAN may store provenance at the *trust domain level*. Here, groups of nodes within a trusted domain share a domain identifier. Provenance encodes sufficient information only on the trust domains involved in each derivation. Trust domain level provenance enables, for example, access control policies based on a *priori* established trust relationships.

Representation. ExSPAN supports storing provenance internally using *graph representation*. A provenance graph reflects the relations between output tuples and the base tuples that contribute to them. Each internal node represents a database relational operator (e.g., union, join, selection and projection) tagged with its location, while each edge denotes a data flow among the operators.

Figure 4 shows the provenance graph for bestPathCost(@a,c,5), derived and stored at node 5. Each operator (denoted by an oval) is annotated with ruleID@S, indicating that rule ruleID is executed at node 5. The edges in the graph show the intermediate computation results (i.e., pathCost and bestPathCost tuples).

Graph representation encodes tuple-level provenance information and is typically used to answer queries pertaining to fine-grained network state. For example, graph representation may be useful for debugging distributed systems [22, 14] and for accepting/rejecting network packets based on their traversed path.

Alternatively, provenance may be more compactly represented using *algebraic representation* [7, 3]. Algebraic representations encode provenance using the binary operations + and * (representing union and join, respectively). For instance, if \(\alpha\), \(\beta\), and \(\gamma\) are the respective unique tuple IDs for link(@a,c,5), link(@b,a,2) and link(@b,c,2), then the provenance of bestPathCost(@a,c,5) in Figure 4 is encoded as \(\alpha + \beta * \gamma\) (or \(<\alpha+a*b>\) when using node-level provenance).

Algebraic representations can be further condensed [13] by encoding them as boolean expressions stored in Binary Decision Diagrams (BDDs) [2]. For example, \(<\alpha+a'b>\) can be condensed to \(\alpha\) since the trustworthiness of node b is inconsequential given a. As long as node a is trusted by the node that receives the bestPathCost(@a,c,5) tuple, the tuple will be accepted, regardless of whether node b is trusted. Enforcing trust policies based on such condensed forms of provenance is useful in network protocols (e.g., BGP) in which updates should only be accepted if they originate from trusted sources. Similarly, one can utilize the algebraic formulation to compute a trust value for each derivation.

Distribution. ExSPAN stores provenance in either a centralized or a distributed fashion. In *centralized provenance* [14], the entire provenance of a tuple is stored along with the tuple’s content. In order to maintain complete centralized provenance, all provenance information is relayed to a centralized server. This approach presents a single bottleneck at the server, high aggregate bandwidth-utilization, and may not be feasible in a setting in which the distributed system being monitored spans administrative domains and/or large geographic locations.

Provenance may also be stored in a distributed fashion. In *value-based distributed provenance*, each derived tuple must include its entire provenance when transmitted between nodes. This paper introduces an alternative and more efficient means of storing distributed provenance which we call reference-based distributed provenance. In the reference-based variant, provenance information is dispersed among network nodes and lazily shipped. Here, only markers (pointers for subsequent traversal) are shipped with each tuple as the protocol executes, and the provenance information is fetched on demand via distributed queries. Rather than store complete provenance data at each tuple, tuples contain pointers that may be resolved recursively to reconstruct their derivations.

Reference-based distributed provenance imposes little communication overhead during query execution, but requires a (potentially expensive) distributed querying protocol to discern provenance information. Conversely, both centralized provenance and value-based provenance incur high communication costs when transmitting tuples (due to the provenance information contained in the
tuples and the propagation of provenance updates), but offer immediate provenance resolution.

Centralized provenance support may be straightforwardly achieved using traditional centralized database systems. Previous work [13] also shows value-based distributed provenance can be supported by modifying relational operators, such as INSERT, DELETE and JOIN, to include provenance information in the query evaluation towards a distributed fixpoint.

ExSPAN aims to support both centralized and distributed distribution. However, the focus of this paper is on enabling efficient network provenance using our reference-based approach.

4. MAINTAINING PROVENANCE

This section defines the data model used by ExSPAN to store and maintain network provenance in dynamic networks. The data model utilizes the graph-based representation introduced in Section 3 and applies to both centralized and distributed provenance. ExSPAN’s graph-based data model is amenable to storage using a distributed relational database, and is sufficiently general to be used as a basis for generating other provenance representations.

4.1 Data Model

Given a tuple \( T \), we model its provenance as an acyclic graph \( G(V,E) \). By disallowing cycles in the graph, we simplify the process of issuing distributed queries (Section 5) at the expense of not permitting cyclical derivations. We note that such cycles are fairly rare in the usage scenarios we have encountered in networking applications, but supporting them is an interesting area of future work.

The vertex set \( V \) consists of \( tuple 
vertices \) and \( rule 
execution 
vertices \). Each tuple vertex in the graph is either a base tuple or a computation result, and each rule execution vertex represents an instance of a rule execution given a set of input tuples. The edge set \( E \) consists of unidirectional edges that represent dataflows between tuples and rule execution vertices, where an edge from a tuple vertex \( t \) to a rule execution vertex \( o \) indicates \( t \) is taken as an input of \( o \). Conversely, an edge from a rule execution vertex \( o \) to a tuple vertex \( t \) denotes that \( t \) is the evaluation result of \( o \).

ExSPAN stores the graph representation of provenance in a relational table in a format similar to that used in existing work [6, 5] with the following modifications to enable efficient maintenance in a distributed setting: (We focus our discussion on reference-based distributed provenance, and adopt the declarative networking convention of having location predicates to denote tuples and their locations.)

To uniquely identify each vertex in a derivation graph, we assign a vertex ID (VID) to each vertex in the provenance graph, using cryptographic hash functions (e.g., SHA-1) to reduce the probability of collision.

For a tuple vertex, the VID is the hash of the tuple’s contents (i.e., its location specifier, table name, and attribute values). For instance, the VID of a tuple vertex for \( pathCost(\langle X, Y, C \rangle) \) is \( \text{VID} = \text{SHA1}("pathCost" + X + Y + C) \), where \( a + b \) denotes the concatenation of \( a \) and \( b \).

For a rule execution vertex, its unique identifier (RID) is the concatenation of the location where the rule resides, the rule name, and the input tuple nodes. For example, when rule \( r2 \) at node \( X \) is executed with input tuples \( t1 \) and \( t2 \), its RID is \( \text{SHA1}(r2^X \langle t2 + t1 \rangle) \).

4.1.1 Example Graph and Tables

Figure 5 shows the provenance graph for a derived tuple, \( bestPathCost(\langle a, c, 5 \rangle) \), using the example network depicted in Figure 3. Ovals represent the rule execution vertices and rectangles depict tuple vertices. The graph encodes how tuples are derived during the execution of the \text{MinCost} \ query. For instance, \( bestPathCost(\langle a, c, 5 \rangle) \) (bottom-right) is generated from rule \( r3 \) at node \( a \) taking \( pathCost(\langle a, c, 5 \rangle) \) as the input. Note that the graph representation itself is distributed, where the location of each vertex is shown using the location specifier \( \langle \rangle \). To trace further, \( pathCost(\langle a, c, 5 \rangle) \) has two derivations: the locally derivable one-hop path \( a \rightarrow c \) and the two-hop path \( a \rightarrow b \rightarrow c \) that requires the distributed join (in rule \( r2 \)) at node \( b \).

Table 1 presents the \( prov \) table that corresponds to the provenance graph shown in Figure 5. For instance, the \( prov \) table contains two entries (the 2nd and 3rd lines) for \( pathCost(\langle a, c, 5 \rangle) \), indicating that the tuple is derivable in two alternative ways: one that is directly derived from \( link(\langle a, c, 5 \rangle) \) and the other that is generated by joining \( link(\langle b, a, 3 \rangle) \) with \( bestPathCost(\langle b, c, 2 \rangle) \). As a special case for base tuples (e.g., the \( link \) tuples), we assign null as the RID to differentiate from the tuples derived via rule evaluation.

4.1.2 Value- and Reference-Based Provenance

The previously described data model incurs very small communication overhead while maintaining reference-based distributed provenance. Only the 20-byte \( Rloc \) and \( RID \) attributes must be af-
fixed to tuples in order to reconstruct the provenance information via a distributed query.

To derive the derivation of a tuple encoded with reference-based provenance, the provenance graph is traversed in a distributed fashion. Given a derivation identified by VID (i.e., the hash of the tuples’ contents), the corresponding VIDList can be retrieved by traversing the RLoc attribute value in the prov table and retrieving the contents corresponding to RID stored in the ruleExec table. The base tuples may then be retrieved by recursively traversing the entries in VIDList. Mechanisms for efficiently querying reference-based network provenance are described in more detail in Section 5.

In value-based distributed provenance, each transmitted tuple includes its entire provenance tree (that is, all the prov and ruleExec tuples that are relevant to its derivation). As we demonstrate in Section 7, the value-based provenance approach results in much higher communication overhead as compared to the reference-based approach. However, value-based distributed provenance is desirable for certain network management applications in which the decision to accept or reject an incoming message may depend on its (immediately available) provenance.

### 4.2 Distributed Provenance Maintenance

Given an NDlog program, incremental maintenance with provenance aims to achieve the following: Whenever a base tuple is inserted, rules incrementally recompute new derivations from existing NDlog rules. ExSPAN achieves incremental view maintenance through delta rules [13, 15], with additional bookkeeping to maintain multiple derivations of the same tuple. These delta rules are then processed in a pipelined fashion via the use of the pipelined semi-naïve algorithm (PSN) [15].

For a Datalog rule of the form: \( d \leftarrow d_1, d_2, \ldots, d_n \), a delta rule is generated for each derived predicate, where the \( k^{th} \) delta rule is of the form:

\[
\Delta d := d_{k1}, \ldots, d_{kn}, \Delta d_k := d_{k1}, \ldots, d_{kn}
\]

where \( \Delta d_k \) denotes a tuple \( t_k \in d_k \) that is used as input to the rule for computing new \( d \) tuples. Each delta \( \Delta d_k \) results in the creation of two delta rules, one for insertion and one for deletion. In PSN, tuples are processed one at a time in a pipelined fashion. Each node maintains a FIFO queue (ordered by arrival timestamp) of new input tuples. Each new tuple is dequeued and is used as input to its respective delta rule. The execution of a delta rule may generate new tuples which are either inserted into the local queue or sent to a remote node for further execution. Refer to references [13, 15] on details of PSN and handling of duplicate derivations.

As views are incrementally recomputed due to new insertions, each rule execution and new derivation results in the creation of new prov and ruleExec entries. Similarly, whenever a base tuple is deleted, all derivations resulted from NDlog rules that depend on the base tuple in the program are incrementally deleted, resulting in cascaded deletions of the respective prov and ruleExec entries in the provenance graphs of deleted tuples.

To perform the above incremental provenance maintenance (for both insertion and deletion), ExSPAN leverages the distributed query processing capabilities of its declarative networking engine. Given any NDlog program, additional NDlog provenance maintenance rules are automatically generated.

#### 4.2.1 MinCost Example

Before describing the provenance compilation procedure in detail, we first present an example to demonstrate the intuition behind the generation of new provenance maintenance rules. The provenance maintenance rules for rule sp2 in Figure 1 are automatically rewritten by ExSPAN as follows:

<table>
<thead>
<tr>
<th>Rule Exec</th>
<th>Provenance Node</th>
<th>RLoc</th>
<th>R</th>
<th>VIDList</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>r00 ePathCostTemp(RLoc,S,D,C,RID,R,List) :-</td>
<td>link(S,C), bestPathCost(S,C), C=C1+C2, Z=V, RLoc=Z, R='sp2', PID=E_sha1('link'+S+C1), PID2=E_sha1('bestPathCost'+2D+C2), List=f_concat(FID1, FID2), R=I_sha1(R=RLoc,List).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r01 ePathCost(S,D,C,RID,RLoc) :-</td>
<td>ePathCostTemp(RLoc,S,D,C,RID,R,List).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r03 pathCost(S,D,C) :-</td>
<td>ePathCost(S,D,C,RID,RLoc).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r04 prov(S,D,VID,RID,RLoc) :-</td>
<td>ePathCost(S,D,C,RID,RLoc), VID=I_sha1('pathCost'+2D+C).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: An example prov relation. The table is horizontally partitioned across all nodes, based on the location specifier Loc. The last column Derivation is not part of the table, but indicates the actual tuple derivation that corresponds to each prov entry.

<table>
<thead>
<tr>
<th>Loc</th>
<th>VID</th>
<th>RID</th>
<th>RLoc</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>VID3 = SHA1(\text{&quot;link&quot;} + a + c + 5)</td>
<td>null</td>
<td>a</td>
<td>link(S,D,C)</td>
</tr>
<tr>
<td>a</td>
<td>VID5 = SHA1(\text{&quot;pathCost&quot;} + a + c + 5)</td>
<td>RID2</td>
<td>a</td>
<td>pathCost(S,D,C)</td>
</tr>
<tr>
<td>a</td>
<td>VID7 = SHA1(\text{&quot;bestPathCost&quot;} + a + c + 5)</td>
<td>RID5</td>
<td>a</td>
<td>bestPathCost(S,D,C)</td>
</tr>
<tr>
<td>b</td>
<td>VID1 = SHA1(\text{&quot;link&quot;} + b + c + 2)</td>
<td>null</td>
<td>b</td>
<td>link(S,D,C)</td>
</tr>
<tr>
<td>b</td>
<td>VID2 = SHA1(\text{&quot;pathCost&quot;} + b + a + 3)</td>
<td>null</td>
<td>b</td>
<td>pathCost(S,D,C)</td>
</tr>
<tr>
<td>b</td>
<td>VID4 = SHA1(\text{&quot;pathCost&quot;} + b + c + 2)</td>
<td>RID1</td>
<td>b</td>
<td>pathCost(S,D,C)</td>
</tr>
<tr>
<td>b</td>
<td>VID6 = SHA1(\text{&quot;bestPathCost&quot;} + b + c + 2)</td>
<td>RID4</td>
<td>b</td>
<td>bestPathCost(S,D,C)</td>
</tr>
</tbody>
</table>

Table 2: An example ruleExec relation that corresponds to the derivations shown in the prov table in Table 1. The last column Derivation indicates the actual derivation for the given rule execution instance.
Rules $r_{20}$ through $r_{24}$ re-implement the original rule $sp_2$ with additional rules for creating and maintaining provenance information. Rule $r_{20}$ takes as input the original rule body predicates $link$ and $bestPathCost$, and generates a new local event $ePathCost$ that contains the results of the derivation (attributes $S$, $D$, and $C$), as well as new attributes $RLoc$ (rule location), $R$ (rule label), $RID$ (hash digest of rule), and $List$ (VIDs of children vertices) necessary for the generation of the provenance information. The attributes stored in $ePathCost$ contain all the information necessary to instantiate the rule $ruleExec$ entry locally (via rule $r_{22}$) and create the $prov$ entry at the remote node. This is achieved by a message event $ePathCost$ that is sent from the node in which the rule is derived ($RLoc$) to node $z$, the destination of the original derivation in rule $sp_2$. Rule $r_{23}$ receives the $ePathCost$ event and generates the $pathCost$ derivation. The $prov$ entry is populated by rule $r_{24}$.

Since rule $r_{23}$ generates the original $pathCost$ derivation in $sp_2$, the above set of rules subsumes the original rule, adding additional information for provenance computation. Note that in addition to generating the correct derivation and provenance information ($ruleExec$ and $prov$ entries), the above program imposes only minimal additional communication overhead. The message event $ePathCost$ contains only two additional attributes $RID$ and $RLoc$. The bandwidth utilization is significantly lower compared with the value-based distributed approach, the latter of which requires shipping the entire provenance information with each tuple.

Finally, note that the above rewritten program integrates well with the use of cascading deletions used in PSN evaluation of declarative networking programs. Rule $r_{20}$ compiles into a series of insert and deletion delta rules that guarantees that whenever the input $link$ and $bestPathCost$ tuples are inserted/deleted, $ePathCost$ will trigger either an insert or delete delta rule appropriately, leading to the insertion/deletion of appropriate entries in the $prov$ and $ruleExec$ tables.

### 4.2.2 Rule Generation Algorithm

Generalizing the above example, Algorithm 1 shows the basic approach towards generating provenance maintenance rules. The algorithm takes as input a localized NDlog program that consists of a set of rules $RS$, each of the form:

$$h(@H_1, ..., @H_n) \leftarrow t_1(@X, P_{11}, ..., P_{1f}), ..., t_n(@X, P_{n1}, ..., P_{nf}),$$

where the rule body consists of $n$ predicates $t_1, t_2, ..., t_n$ and $p$ constraints/assignments $C_1, C_2, ..., C_p$. The derived rule head predicate $h$ consists of $o$ attributes which are generated from executing the rule body. Here, we assume that all body predicates are executed at location $X$. We further consider the case in which the rule head location $H_1$ is at a different location from $X$ (and hence the derivation is sent across the network).

The output of running the algorithm is the set $RS'$, where for each rule $r \in RS$, five new rules are generated for provenance maintenance and executing the original derivation. The first rule generates the local event $ehTemp$ (line 4) which contains all information required for creating the local $ruleExec$ entry (line 5) corresponding to the meta-data for rule execution, sending the event message $eh$ to the target node $H_1$ (line 6) to create the corresponding result tuple $t_1$. (line 7) and the remote $prov$ entry (line 8). Note that the only additional attribute shipped with each message $eh$ is the $(RID, RLoc)$ pair necessary to reconstruct the provenance information.

It is worth emphasizing that Algorithm 1 also applies to NDlog programs that contain aggregates. In this paper, we restrict the Algorithm to the MIN and MAX aggregates. For such aggregates, the provenance information can be traced to the minimum or maximum tuple respectively as its child derivation. This is in contrast to other aggregates such as COUNT and SUM that require storing all input tuples as its provenance information.

### 5. Querying Provenance

The previous section presents the distributed provenance data model and introduces an algorithm for automatically rewriting NDlog programs to support provenance. In this section, we describe how ExSPAN allows users to query the provenance information via distributed queries.

**Intuition:** Before describing ExSPAN’s provenance query mechanisms, we first present the intuition of provenance querying through an example. We consider the provenance graph shown in Figure 5 and a query for the full provenance of tuple $bestPathCost(@a,c,5)$. The initial $prov$ entry for this tuple (corresponding to the entry with $VID=VID_1$ in Table 1) indicates that the tuple is derived from the execution of rule $sp_3$ at node $a$. The query then retrieves the corresponding entry in the $ruleExec$ table (in this case the entry for $RID=RID_1$). The query subsequently traces back to the input tuple corresponding to $VID$, i.e., $pathCost(@a,c,5)$, by following the pointers maintained in $VIDList$. The process continues recursively by next retrieving the $prov$ entries for $VID$, followed by the corresponding $ruleExec$ entries.

Since $pathCost(@a,c,5)$ has two derivations (via $sp_{18}$ or $sp_{20}$), two queries are initiated to further query the provenance along the two derivations. This results in cross-node communication, since $sp_{20}$ is located at a remote node. The recursive query stops at base tuples, e.g., $link(@a,c,5)$, $link(@b,c,2)$, and $link(@b,a,3)$. Provenance information is returned through the reverse direction of the path traversed by the queries.

#### 5.1 Distributed Recursive Query Formulation

Based on the above intuition, we next demonstrate the mechanisms by which ExSPAN formulates distributed queries to derive various representations from reference-based distributed provenance.

To derive provenance information, ExSPAN utilizes NDlog programs that express distributed recursive queries. These queries traverse provenance graphs (in the form the $prov$ and $ruleExec$ tables) in a distributed fashion, returning results to the querying node. ExSPAN’s flexibility permits different granularities and representa-
tions of provenance (see Sec. 3). The programmer may select the type of network provenance by modifying the query specifications.

The following NDlog program demonstrates a generic distributed graph traversal operation on tables prov and ruleExec. The entire program is written in ten NDlog rules: two base rules (edbl and c0) and two pairs of four rules for recursively querying the prov (idb1-idb4) and ruleExec (rv1-rv4; not shown) tables. The rules are continuous, long-running queries that are initially installed at every ExSPAN node for handling distributed provenance queries.

// Base case
edbl eProvResults@[Ret,QID,VID,Prov) :-
eProvQuery@[X,QID,VID,Ret), prov@[X,VID,RID,RLoc),
RID=NULL, Prov=f_pEDB(VID).
// Count number of children for each VID
c0 numChild@[X,VID,COUNT=<>) :- prov@[X,VID,RID,RLoc).
// Initializing Buffer
idb1 pResultTmp@[X,QID,Ret,VID,f_empty()) :-
eProvQuery@[X,QID,VID,Ret), prov@[X,VID,RID,RLoc),
RID!-NULL.
// Recursive case
idb2 eRuleQuery@[RLoc,RQID,RID,X) :-
eProvQuery@[X,QID,VID,Ret), prov@[X,VID,RID,RLoc),
RQID=£_shal(QID+RID).
// Buffer sub-results
idb3 pResultTmp@[X,QID,RID,VID,Buf] :-
eRuleQuery@[RLoc,RQID,RID,Fro),
pResultTmp@[X,QID,RID,Buf),
RQID=£_shal(QID+RID), Buf=f_concat(Buf,Prov).
// Calculate and return results
idb4 eProvResults@[Ret,QID,VID,Prov) :-
pResultTmp@[X,QID,RID,VID,Buf),
numChild@[X,VID,C], Buf=set(Buf,Prov).

To customize provenance computations in the distributed graph traversal query, we introduce three user defined functions: f_pEDB, f_pIDB, and f_pRULE, which operate on the base tuples (f_pEDB), intermediate derivations (f_pIDB), and rule execution instance (f_pRULE). In Section 5.2, we describe these functions in greater detail, and via examples, show how they can be customized to return different provenence representations and granularities.

The initial provenance query is indicated by the event eProvQuery@[X,QID,VID,Ret), where node Ret issues a query to retrieve the provenance information of tuple VID stored at X. To uniquely identify the query, an additional attribute QID is added. Note that upon receiving this query, node X executes rules edbl, idb1, and idb2.

Rule edbl is the base case and applies when the tuple VID is a base tuple (EDB), as indicated by the fact that it has no associated rule execution instance (that is, RID is null). In such cases, the provenance information is f_pEDB(VID) – the result of applying the user-defined function for EDBs to VID. For example, f_pEDB may simply return the tuple itself, indicating the base tuple is involved in the derivation.

Rule idb1 initializes the pResultTmp table, which is later used to buffer intermediate query results. Rule idb2 represents the recursive case in which the prov table is retrieved. Each entry with matching VID in the prov table indicates a rule execution instance that leads to the derivation of VID. These rule execution instances are additionally retrieved and buffered in pResultTmp by issuing a remote query eRuleQuery@[RLoc,RQID,RID,X). This requires sending a message to several matching RLoc nodes. The process continues recursively, where the nodes receiving the eRuleQuery message retrieve the matching ruleExec tuples, and recursively traverse children derivations until the base case is reached.

ExSPAN applies rule idb3 when all children derivations have returned with the provenance information. The resulting provenance information is then combined in rule idb4 using the f_pIDB function and the results are returned to the query node.

An additional four rules rv1-rv4 (similar to idb1-idb4) perform a similar traversal of the ruleExec tables. We omit these rules due to space constraints. The intuition behind these rules is that the user recursively traverses prov and ruleExec tables across nodes until the entire provenance tree has been obtained. Since each rule execution takes several predicates as input, an additional user defined function f_pRULE enables the user to customize how the various inputs to the rule can be combined in the provenance tree.

5.2 Query Customization

Given the general querying framework present in Section 5.1, we now describe how users may customize provenance queries to meet various application requirements.

5.2.1 First Example: Provenance Polynomials

Our first customization example stores provenance information in the form of an algebraic expression called a provenance semiring [7]. Provenance can be encoded as an algebraic structure with two binary operations — addition and multiplication — indicated by “+” and “·”, where “+” indicates the combination of tuples with union and projection and “·” denotes a natural join over tuples. The literals in the algebraic expression represent base tuples. By customizing the “+” and “·” operators, various types of classic provenance annotation can be encoded. For example, r1(A + r2(B · C)) indicates that rule r2 applies JOIN on tuples B and C, and the result is then UNIONed with A in rule r1.

To return provenance query results as polynomials, the three user-defined functions are implemented as follows:

- **f_pEDB(VID)** takes as input the VID that uniquely identifies the base tuple. The function simply returns the base tuple itself or its primary keys (which can be retrieved by reading a systems table that maps VIDs to tuples).

- **f_pIDB(Derivations,Loc)** takes as input Derivations that contain the polynomials (D1, D2, ..., Dn) that represent all possible n ways to derive the tuple, and Loc, the location specifier of the tuple. The function iterates over all entries in Derivations and applies a “+” operation across them. The resulting polynomial expression is annotated with the location as (D1 + D2 + ... + Dn)@Loc.

- **f_pRULE(ChildPred,R,RLoc)** takes as input ChildPred, representing the polynomials of all n input tuples (P1, P2, ..., Pn) that are used in the execution of rule r at location Loc. The function iterates over all entries in ChildPred and applies a “·” operation across them. As above, the result is affixed with a rule label and location. The function returns the polynomial (R@RLoc)(P1 · P2 · ... · Pn).

5.2.2 Additional Examples

Table 3 presents additional representative examples of possible provenance customizations. Similar to our notation in Section 5.2.1, (D1, D2, ..., Dn) contains the provenance annotations of all possible ways to derive a given tuple VID, and (P1, P2, ..., Pn) denotes similar annotations of input tuples to a particular rule execution instance.

The first example, NodeSet, returns the set of nodes that participate in the derivation of a tuple. For each base tuple, f_pEDB(VID) returns the node ID where VID is stored. f_pIDB(Derivations,Loc) and f_pRULE(ChildPred,R,RLoc) both return the union of the set of nodes. Note that this query can be trivially extended to return the number of unique nodes participating in the query.
The cached results are then sent back on the reverse path back to the node conducting the query without further traversal.

**Cache invalidation** Cached provenance query results become invalidated when a tuple is inserted or deleted. Upon receiving the update event for tuple updates, all the caches that depend on the tuple should be invalidated through an invalidation propagation procedure. This mechanism is similar to how distributed value-based provenance is maintained; however, rather than shipping the whole tuple (an expensive operation), the cache invalidation procedure requires only that an invalidation flag be sent.

After a cache is invalidated, future queries for the same tuple will have to perform distributed queries to compute the correct result and update the cache. However, since a tuple may have multiple derivations (some of which correspond to intact cache entries), the use of caching may still offer some performance benefits. To illustrate, consider the network depicted in Figure 5. Suppose the link between nodes a and c has failed. The link deletion will invalidate the cache for tuples pathCost(a, c, 5), bestPathCost(a, c, 5) and the rule execution vertices along the path. However, if a query is issued for bestPathCost(a, c, 5) after the link failure, when the traversal reaches the rule execution vertex r2@b[link(@b,a), bestPath(@b,c)], the intact cached result will be directly returned, eliminating the need to further traverse the derivation graph.

### 6.2 Query Traversal Order

At each tuple vertex being traversed, the program shown in Section 5 simultaneously issues queries to all possible derivations. In essence, the distributed queries traverse the provenance graph using Breadth First Order (BFS). Intuitively, BFS must flood the queries throughout the whole provenance graph before any sub-results are obtained.

We explore another querying traversal order, Depth First Order (DFS), in which alternative derivations are explored at each tuple vertex. DFS may incur longer querying latencies than BFS since the former can stall before a sub-result is received. However, DFS provides the opportunity for bandwidth savings for threshold-based queries in which a user asks, for example, whether a tuple has more than T derivations or whether fewer than T’ unique nodes participate in the derivation. DFS allows such threshold-based queries to terminate as soon as the threshold is reached, without incurring additional communication overhead. That is, DFS trades off query

### Table 3: User-defined functions for various queries

<table>
<thead>
<tr>
<th>Function</th>
<th>Node Set</th>
<th># of Derivations</th>
<th>Derivability Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_pIDB</td>
<td>[NodeID]</td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>f_pEDB</td>
<td>∑n∈N Dn</td>
<td>∏n∈N Dn</td>
<td>Vn∈N Dn</td>
</tr>
<tr>
<td>f_pRULE</td>
<td>∏n∈N Pn</td>
<td>∏n∈N Pn</td>
<td>Nn∈N Pn</td>
</tr>
</tbody>
</table>

When queries are frequent, subsequent queries can leverage the results from prior queries.

### Caching scheme

Unlike traditional caching in a centralized database, the reference-based distributed provenance cache is distributed across several nodes in the network. Whenever a node N issues a distributed query to retrieve provenance information for a tuple VID, the resulting query results are not only cached at node N, they are also stored at intermediate nodes as the query results are returned along the reverse path.

Specifically, whenever rule idb4 (or the equivalent rule for ruleExec traversal) is triggered, it indicates the completion of a query at an intermediate derivation. The query result will be maintained in events eProvResults or eRuleResults. Before eProvResults or eRuleResults is sent back to the query issuer, these results are cached in a cache (N, VID, Results) table that stores (at node N) the provenance Results for VID. Further attributes can be added to distinguish results based on provenance representation. Note that subsequent queries need not be for the exact tuple (i.e., VID) in order to benefit from the cache: since the intermediate results are cached along the reverse path, any graph traversal query that reaches node N and requires a subgraph rooted at VID can use the cache results. The cached results are then sent back on the reverse path back to the node conducting the query without further traversal.

**Cache invalidation** Cashed provenance query results become invalidated when a tuple is inserted or deleted. Upon receiving the update event for tuple updates, all the caches that depend on the tuple should be invalidated through an invalidation propagation procedure. This mechanism is similar to how distributed value-based provenance is maintained; however, rather than shipping the whole tuple (an expensive operation), the cache invalidation procedure requires only that an invalidation flag be sent.

After a cache is invalidated, future queries for the same tuple will have to perform distributed queries to compute the correct result and update the cache. However, since a tuple may have multiple derivations (some of which correspond to intact cache entries), the use of caching may still offer some performance benefits. To illustrate, consider the network depicted in Figure 5. Suppose the link between nodes a and c has failed. The link deletion will invalidate the cache for tuples pathCost(a, c, 5), bestPathCost(a, c, 5) and the rule execution vertices along the path. However, if a query is issued for bestPathCost(a, c, 5) after the link failure, when the traversal reaches the rule execution vertex r2@b[link(@b,a), bestPath(@b,c)], the intact cached result will be directly returned, eliminating the need to further traverse the derivation graph.
latency in favor of reduced communication overhead for threshold-based queries.

The following modifications are required to adapt the BFS distributed graph traversal program (see Section 5) to a DFS query:

```prolog
idb2a pQList((X,QID,AGGList<RID,ThLoc>)) :-
   eProvQuery((X,QID,UID,Ret),
   prov((X,UID,RID,ThLoc), RID=NULL).

idb2b eIterate((X,QID,N)) :- pResultTmp((X,QID,Ret,UID,Buf),
   numChild((X,UID),CN=X,NC=Y), 
   f_pIDB(Buf,X)<Threshold.

idb2c eRuleQuery((RLoc,RQID,RID,X)) :-
   eIterate((X,QID,N),
   pQList((X,QID,RLID,RLoc),
   RID=f_item(RID), RLoc=f_item(RLoc)).

idb4' eProvResults((@Ret,QID,UID,Prov)) :-
   pResultTmp((@X,QID,Ret,UID,Buf),
   numChild((@X,UID),CN=X,NC=Y), 
   Prov=f_pIDB(Buf,X)||Prov>Threshold.
```

 Instead of starting queries for each derivation simultaneously, the revised program iterates through the list of alternative derivations. The exploration of the next derivation is started only if the results of the previous explorations have been received.

To allow the result to be returned as soon as the threshold requirement is satisfied, the original `idb2` rule is replaced by three rules (`idb2a` through `idb2c`) and an additional condition is added to `idb4` (as shown in rule `idb4`). Additional rules for conducting a DFS traversal for each rule execution vertex are omitted due to space constraints.

The `pQList` table in rule `idb2a` is a temporary table that maintains the list of rule execution vertices and their location. We introduce a special aggregation function `AGGList` that generates lists for the designated attributes, indicated as `A1` to `AN`, for the tuples in each group.

Whenever there is an update in the `pResultTmp` table in rule `idb2b`, the revised program counts the number of the results that have been ever received by checking the size of the buffer (that is, the `Buf` attribute in `pResultTmp`). If the current result is under the threshold, the program expands the next derivation, if one exists. If not, rule `idb4` is triggered and the result is returned.

In addition to BFS and DFS, random moonwalk traversal can easily be implemented by randomly selecting `N` alternative derivations to explore, where `N` is a pre-defined constant. This technique is particularly useful when the number of a tuple’s derivations is significantly large. The random moonwalk pinpoints with high probability the pivotal tuple that contributes to a derivation. In the context of networking applications, such random moonwalks are useful for ascertaining the dominating sources of incoming traffic.

### 6.3 Condensed Provenance

Our third optimization technique applies a previously proposed compression scheme known as absorption provenance [13] to the algebraic representation of provenance (see Section 5.2.1). Absorption provenance aims to reduce the number of variables in an algebraic representation. For example, consider the algebraic expression `a * (a + b)`. By applying boolean absorption rules [13], the expression is reduced to `a * (a + b) = a + (a * b) = a`. Note that savings in size comes at the expense of information loss. In our example, absorption provenance loses the fact that `b` is also involved in the derivation. However, such absorbed encodings can still retain sufficient information for derivability tests or enforcing security policies based on the trust of source origins (base tuples).

To implement absorption, we utilize BDDs [2] to encode provenance. BDDs provide a natural way to encode the algebraic representation of the provenance, and by default, apply absorption to save storage space. Since BDDs are frequently used in circuit synthesis and formal verification applications, highly optimized libraries provide abstract BDD types as well as Boolean operators that operate on them: pairs of BDDs can be ANDed or ORed; individual BDDs can be negated; and variables within BDDs can be set to true or false.

Note that the use of absorption provenance applies to both centralized provenance and value/reference-based distributed provenance. For example, in centralized and value-based provenance, the provenance information shipped for each tuple can be stored as BDDs. Similarly, for reference-based distributed provenance, query results can be returned in the form of BDD representations.

### 7. EVALUATION

In this section, we evaluate the ExSPAN network provenance system. The goals of our evaluation are twofold: (1) to measure the performance overhead incurred by value- and reference-based distributed network provenance; and (2) to study the effectiveness of optimizations at reducing communication cost.

**Implementation and Experimental Setup** ExSPAN is implemented as an add-on to ns-3 [19], an emerging discrete event-driven network simulator aimed to replace ns-2. ns-3 emulates all layers of the network stack, supporting configurable loss, packet queuing, and network topology models. ns-3 supports both a simulation mode, enabling the examination of ExSPAN’s performance under various network topologies and conditions, as well as a deployment mode in which different hosts in a testbed environment execute the network provenance system. ExSPAN makes extensive utilization of RapidNet [18], a declarative networking platform that compiles NDlog programs into applications that are executed by the ns-3 runtime. ExSPAN uses the identical codebase for both simulation and deployment modes.

We generate transit-stub topologies for our simulation experiments using the GT-ITM topology generator [8]. The transit-stub topology consists of eight nodes per stub, three stubs per transit node, and four nodes per transit domain. We increase the number of nodes in the network by increasing the number of domains. Links between adjacent transit nodes experience a 50 ms latency and have a 1 Gbps bandwidth capacity; transit-stub connections have a 10 ms latency and a 100 Mbps capacity; the respective latency and bandwidth between stub nodes are 2 ms and 50 Mbps.

Our deployment experiments are executed within a local cluster of eight dual-core Intel 2.8GHz Pentium D hosts and 16 quad-core machines with Intel Xeon 2.33GHz CPUs. All machines run Linux 2.6 and are interconnected by high-speed Gigabit Ethernet. ExSPAN communicates messages between nodes via UDP packets. To increase the size of our network, we execute two instances of ExSPAN on each quad-core machine, enabling us to scale the experiments to 40 nodes. The deployment results described in Section 7.4 reflect the average of five executions of the experiment.

**Applications** As workloads for our simulation and deployment experiments, we implement three NDlog applications: MinCost, introduced in Section 2, computes the costs of the best (least cost) paths between pairs of nodes. PATHVECTOR extends MinCost, enabling a node to discover the best path (transmitted as a vector of nodes) to a specified destination. Both MinCost and PATHVECTOR operate on the control plane, enabling nodes to discover routes to peers. In contrast, the PACKETFORWARD application operates on the data plane, relaying data packets using previously discovered paths.

For all experiments, each node is initialized with a `link` tuple for each of its neighbors. That is, nodes have `a priori` knowledge of their local links and use MinCost and PATHVECTOR to discover
The use of BDD representation, value-based provenance (line "Value-based Prov. (BDD)") incurs very little communication overhead, increasing the amount of information that must be communicated using value-based provenance. However, due to space savings in communicating pointers rather than values, the reference-based technique imposes significantly less overhead (6% in the 300-node network) than the value-based technique (45% in the same network).

In addition to enabling provenance on the control plane, it may also be useful to provide provenance information on the data plane. As described above, the PACKETFORWARD program relays packets according to shortest-path “next hop” information stored at each node. Figure 8 shows the average bandwidth over time when forwarding packets in a 200-node network. Here, each node selects a peer at random and transmits 1024 byte tuples at the rate of 100 tuples per second. The overheads for PACKETFORWARD are roughly equivalent for value- and reference-based provenance. Sending packets with provenance incurs a small overhead, but is subsumed by the large payloads.

Due to the memory constraints of running large-scale ns-3 simulations on a single machine, we experimented with up to 500 nodes in our simulations. However, our results clearly demonstrate promising scalability trends for all protocols: the average communication costs of MINCOST and PATHVECTOR programs (without provenance) scale linearly with the number of nodes. This matches the expected scalability behavior for these two protocols. Moreover, with the addition of reference-based provenance, we note that the communication costs continues to scale linearly, hence maintaining the original scalability trends, demonstrating that reference-based provenance incurs minimal impact on the scalability of an existing protocol.

7.2 Incremental Maintenance

Figure 6: Average communication cost (MB) for MINCOST.

Figure 7: Average communication cost (MB) for PATHVECTOR.

Figure 8: Average bandwidth cost (MBps) for PACKETFORWARD.

Figure 9: Average bandwidth cost (MBps) for MINCOST when the network experiences a high level of churn.

Figure 10: Average bandwidth cost (MBps) for PATHVECTOR when the network experiences a high level of churn.

Figure 11: Average bandwidth cost (KBps) with and without caching.
The experiments described above model a static topology in which nodes neither leave nor join the network and links never fail. Here, we evaluate ExSPAN’s ability to mitigate a high level of node churn and link failure. We model churn by adding or deleting ten randomly selected stub-to-stub links in a 200-node simulated network (originally containing 315 stub-to-stub links) every 0.5 seconds, with addition or deletion occurring with equal probability.

Figures 9 and 10 show the respective average per-node bandwidth costs of running the M\textsc{in}cost and Path\textsc{vector} protocols. In both instances, reference-based provenance does not incur significant bandwidth overhead (the lines for “No Prov.” and “Ref-based Prov.” closely overlap). For example, the maximum increase in bandwidth due to reference-based provenance (relative to conducting the query without provenance) is 0.07 and 0.05 Mbps for M\textsc{in}cost and Path\textsc{vector}, respectively.

Value-based provenance consumes significantly greater bandwidth as complete provenance information must be affixed to each transmitted tuple. The respective increases in bandwidth for M\textsc{in}cost and Path\textsc{vector} are 1.0 and 0.30 Mbps—1329% and 500% greater than the equivalent overheads for reference-based provenance.

For both reference- and value-based provenance, a new fixpoint is reached within 0.5 seconds, indicating that ExSPAN is resistant to even high levels of churn regardless of the type of provenance used.

### 7.3 Distributed Queries

Sections 7.1 and 7.2 evaluate the performance of provenance maintenance in ExSPAN. In this set of evaluations, we study the performance of distributed querying of provenance using the framework presented in Section 5. In addition, we validate the effectiveness of optimizations in reducing communication overhead during query. The experiments are performed using a 100-node simulated network which runs the M\textsc{in}cost protocol. Our query measurements begin after the network has reached a fixpoint.

We utilize three queries in our evaluation: POLYNOMIAL, BDD, and \textsc{#derivation}. POLYNOMIAL acquires the provenance of an arbitrary tuple in the form of provenance polynomials (see Section 5.2.1). BDD, as discussed in Section 6.3, encodes provenance in a more compact format. Using the customization described in Section 5.2.2, \textsc{#derivation} computes the number of alternative derivations for a given tuple.

**Caching** Figure 11 plots the average per-node bandwidth overhead when each node issues five POLYNOMIAL queries per second with each query targeted to a randomly selected bestPathCost tuple. Without caching, the average bandwidth utilization for each node is approximately 50 KBps. Each query therefore incurs an average bandwidth cost of 0.1KBps, an acceptable overhead for most current networks. (Of course, the precise cost of conducting POLYNOMIAL queries in other settings depends upon the provenance of the queried tuples.) The overhead imposed by POLYNOMIAL is due in part to its requirement that results must contain complete information regarding all possible tuple derivations.

As shown in Figure 11, POLYNOMIAL’s overhead can be significantly reduced by enabling the caching optimization described in Section 6.1. Using caching, the average bandwidth utilization decreases to 20KBps after two seconds. The performance improvement is attributed to the fact that queries are more likely to benefit from the cached results of previous queries as time progresses.

Figure 12 presents the cumulative fraction of query completion times. Regardless of whether caching is used, results are returned in less than 0.3 seconds, highlighting that ExSPAN’s provenance...
querying mechanisms are latency-wise efficient. The figure also shows the advantage of enabling caching: 80% of queries are returned within less than 50 ms if caching is enabled, a 67% improvement over query latency when caching is disabled.

**Query Traversal Order** To study the trade-offs between different query traversal orders, we conducted experiments in which nodes utilize the #DERIVATION query to determine whether a bestPathCost tuple has more than three alternative derivations (the average number of alternative derivations for bestPathCost is approximately three). The experiment is performed on three variants of the #DERIVATION query: (a) BFS, (b) DFS, and (c) DFS-THRESHOLD (DFS with threshold-based pruning). We use the same experimental setup as the caching experiment—there is each node in the 100-node network issues five queries per second for randomly selected bestPathCost tuples.

Figure 13 shows the average bandwidth consumption for different query traversal orders. As can be discerned from the figure, the bandwidth costs incurred by BFS and DFS are roughly equivalent (since both must traverse the entire provenance graph before a result is concluded). In contrast, DFS-THRESHOLD results in a 40% decrease in bandwidth consumption, due largely to its avoidance of a full traversal of provenance graphs for tuples with multiple derivations.

Figure 14 plots the cumulative distribution of query completion times for the query traversal strategies. Although the median latency is roughly equivalent for BFS and DFS, the latter experiences a long-tail distribution. For example, less than 80% of BFS queries complete within 0.16 seconds. In contrast, 80% of DFS queries require 0.45 seconds.

BFS’s query completion is largely determined by the traversal depth in the provenance graph. Unlike BFS, DFS traverses alternative derivations in order, resulting in longer querying completion latencies. By terminating the query as soon as three derivations are explored, DFS-THRESHOLD avoids the long-tail distribution experienced by DFS. Using DFS-THRESHOLD, the query completion time for 80% of the queries decreases from 0.45 to 0.3 seconds.

**Absorption Provenance** We next compare the performance of the POLYNOMIAL and BDD queries. We use the same network configuration and query rates as were applied in previous experiments. Figure 15 shows the average bandwidth incurred by the POLYNOMIAL and BDD queries. POLYNOMIAL incurs 18Kbps (57%) more bandwidth than BDD, due mainly to BDD’s compact binary representation. As described in Section 6.3, BDD additionally decreases communication overhead by condensing provenance information using lossy compression (at the expense of information loss).

POLYNOMIAL and BDD have near-identical performance when defined in terms of query completion latency. The latency of a query is largely decided by its traversal depth. Since both query techniques follow BFS query traversal order and operate on the same topology, the distributions of query completion latencies across nodes is consistent across the two strategies.

### 7.4 Testbed Experiments

To empirically evaluate ExSPAN’s computation and communication properties, we installed 40 instances of ExSPAN in a local cluster. Since nodes on the cluster are fully connected via a shared switch, we impose a less trivial virtual topology as follows: to ensure reachability, nodes are arranged in a ring structure. All links are bidirectional; that is, if there is a link between nodes a and b, then node a maintains a tuple link(@a,b,1) and node b has a tuple link(@b,a,1). Each node in the network has links to its two neighbors (hence achieving the ring structure). Additionally, each node has a link to a random peer such that the maximum degree of all nodes is three (a link to each ring neighbor and a third to a random peer). All nodes execute the PATHVECTOR protocol.

As with the simulation experiments, our reference-based provenance technique significantly reduces the overhead of provenance compared to the value-based approach. When sending no provenance information, the average per-node bandwidth cost of executing PATHVECTOR is 1.24 KB before a fixpoint is reached. Reference-based provenance increases this cost by 29%, far less than the 204% increase caused by value-based provenance. This trend can be observed from Figure 16 which plots the average per-node bandwidth over time for the experiments. The relative overheads of reference- and value-based provenance mirror our earlier simulation results (see Section 7.1).

In addition to examining bandwidth costs, our deployment provides a mechanism to study the computational overhead of using the various provenance techniques. Figure 17 shows the fixpoint time for different network sizes. (As an invariant of network size, the degree of each node in the network is fixed at three.) As can be discerned from the Figure, neither provenance technique imposes any significant increase in fixpoint time.

The results of our deployment experiments therefore indicate that reference-based provenance achieves a substantial decrease in communication cost as compared to value-based techniques, while imposing little or no increase in fixpoint latency.

### 8. RELATED WORK

ExSPAN is related to a large body of work in the database literature on enabling provenance support in database systems. Of particular relevance to our work are recent attempts at storing provenance information in relational databases [10, 5]. The data and storage model in ExSPAN is also based on the relational model, but extends the basic model to enable distribution via the use of value-based and reference-based distributed provenance. In the space of distributed query processing, Liu et al. [13] explored a mechanism similar to the value-based provenance used by ExSPAN. Their approach stores BDD-based provenance information with each tuple for tracking derivability information.

In terms of application scenarios, our efforts in this paper are largely targeted at networking applications as surveyed by Zhou et al. [25]. Of closest relevance is the recent use of provenance in P2P data integration [10]. The operating environment however differs significantly. ExSPAN is targeted at Internet-scale deployments (i.e., across at least thousands of nodes) with relatively small network state per node, as opposed to only tens of databases storing large amounts of data. Hence, the techniques developed in ExSPAN focus on network-centric metrics such as reducing communication overhead, minimizing query latency, and avoiding negative impact on the convergence times of existing protocols. Rather than using a heavy-weight database system, ExSPAN leverages a declarative networking engine that provides networking and querying capabilities at Internet-scale, enabling it to be easily integrated into existing distributed systems.

While the focus of ExSPAN is on enabling provenance for large-scale distributed systems and networking protocols, in principle, the system is sufficiently general to enable other traditional use cases of provenance in P2P data integration. Exploring such use cases in ExSPAN via the use of declarative networking is an interesting avenue of future work.

### 9. CONCLUSION

This paper presents ExSPAN, a scalable framework for achiev-
ing network provenance in a distributed environment. ExSPAN utilizes declarative networking techniques and rewrite rules to efficiently affix provenance information to distributed network queries, enabling administrators to easily add accountability, trust management, and failure diagnostic capabilities to their networks.

To achieve provenance at Internet-scale, we introduce novel techniques for communicating network provenance. ExSPAN significantly reduces communication overhead by distributing provenance information among nodes. In contrast to existing approaches in which complete derivation trees must be attached to each communicated message, ExSPAN appends short provenance pointers to tuples to identify the nodes that maintain the relevant provenance information. Simulation and implementation results demonstrate that our reference-based provenance techniques impose substantially less communication overhead than existing approaches. For example, when providing provenance information for the path vector routing protocol, ExSPAN exhibits one ninth the communication overhead as incurred using traditional value-based distributed provenance techniques while achieving equivalent fixpoint latencies. Additionally, we present several optimization techniques for efficiently querying provenance information.

As future work, we are investigating mechanisms to protect the confidentiality and authenticity of provenance information. The ability to provide formal security guarantees for provenance data enables new classes of routing algorithms in which decisions can be based not only on the contents of messages, but also on the matter in which messages are created and transported. We are also exploring the integration of ExSPAN with legacy distributed systems. Here, the goal is to use ExSPAN to analyze the protocol behavior by capturing coarse-grained provenance information obtained by having these systems export their network state and incoming/outgoing messages to ExSPAN as tables.

10. ACKNOWLEDGMENTS

This work is supported in part by NSF grants IIS-0812270, CCF-0820208, CNS-0831376, CAREER CNS-0845552, and OSD/AFOSR MURI on Collaborative Policies and Assured Information Sharing.

11. REFERENCES