Distributed Time-aware Provenance

Wenchao Zhou° Suyog Mapara* Yiqing Ren* Yang Li*
Andreas Haeberlen* Zachary Ives* Boon Thau Loo* Micah Sherr°

*University of Pennsylvania, Philadelphia, PA °Georgetown University, Washington, DC

{wzhou, msherr}@cs.georgetown.edu
{yangli2, suyogm, yiqingr, ahae, zives, boonloo}@cis.upenn.edu

ABSTRACT

The ability to reason about *changes* in a distributed system's state enables network administrators to better diagnose protocol misconfigurations, detect intrusions, and pinpoint performance bottlenecks. We propose a novel provenance model called *Distributed Time-aware Provenance* (DTaP) that aids forensics and debugging in distributed systems by explicitly representing time, distributed state, and state changes. Using a distributed Datalog abstraction for modeling distributed protocols, we prove that the DTaP model provides a sound and complete representation that correctly captures dependencies among events in a distributed system. We additionally introduce *DistTape*, an implementation of the DTaP model that uses novel distributed storage structures, query processing, and cost-based optimization techniques to efficiently query time-aware provenance in a distributed setting. Using two example systems (declarative network routing and Hadoop MapReduce), we demonstrate that DistTape can efficiently maintain and query time-aware provenance at low communication and computation cost.

1. INTRODUCTION

Troubleshooting the performance, configuration, and security problems of complex distributed applications and systems is becoming increasingly difficult. Such problems do not necessarily have a single, readily apparent cause; rather, they can arise from a particular *combination* of behaviors, both within the network and at different hosts, which can be very hard to find. Thus, it would be useful if distributed systems supported automated network-wide diagnostics that could assist administrators in determining the root cause(s) of a malfunction (*debugging*) [18], identifying the source of an intrusion (*forensics*) [10], or discovering the reason for suboptimal distributed performance (*performance profiling*) [20].

The key challenge in each of these tasks is to inspect the data flows, dependencies, and updates to distributed (networked) nodes' state — often in ways that are not predictable in advance. Existing domain-specific solutions [4, 18] typically work by recording some forensic data at each node, e.g., a list of past routing changes, which are then used to answer the administrator's questions on demand. However, tailoring the schema and the introspection mechanisms to each new application is cumbersome and inflexible. It would be preferable to have a generic solution that can be applied to arbitrary distributed systems.

As a first step towards enabling this, the system could track *data provenance* [2]; however, the above tasks involve a number of questions that traditional data provenance cannot answer very well. To explain why, we consider a simple use case from Internet interdomain routing: a network operator wants to investigate why his route to eBay changed from r_1 to r_2 a minute ago. To support this scenario:

- We must be able to capture **historical** information about past states and interactions within the systems, not just about the current state. Traditionally, data provenance records relationships among state within tables; historical provenance would record relationships among entries in event logs.
- We must take **messages and timing** into account, rather than simply looking at global sequences of events. In a distributed system, the local clocks of the different nodes can be slightly out of sync, messages take time to propagate from one node to another and are sometimes lost by the network, etc.
- We must choose the right **cost tradeoff**. For instance, if some of the distributed nodes' operations are deterministic, provenance could be recorded at multiple levels of detail: one could record the entire record of state derivations, or instead just record the inputs and use deterministic replay to re-derive missing state.
- We must have the ability to **distribute the storage** of the provenance to keep communication costs down: for performance reasons, centrally archiving the system's entire provenance is impractical. This means that we also need the ability to **detect when nodes tamper with the provenance**; otherwise, a compromised node could cover its traces and avoid detection.

The problems listed above require a set of solutions that are beyond the scope of a single paper. In prior work [23, 25], we have already made some progress towards the fourth goal by demonstrating how to maintain and query provenance in a system that is potentially compromised. In this paper, we tackle the remaining three goals from a database perspective, and we address the key question of what should be modeled in a time-aware provenance solution.

Specifically, this paper develops the foundations of *distributed time-aware provenance* (DTaP), which can meet the above requirements. DTaP captures time, distribution, and causality of updates; it enables the administrator of a distributed system to pose "ad hoc" queries over the system's prior states, communications patterns, event orderings, and more. We present a formal model and the semantics of DTaP. Finally, using a Datalog abstraction for modeling distributed protocols, we prove that DTaP provides a sound and

Proceedings of the VLDB Endowment, Vol. 6, No. 2
Copyright 2012 VLDB Endowment 2150-8097/12/12... $ 10.00.

complete representation of the causality of events in a distributed system. Due to practical challenges in distributed systems (such as loosely synchronized clocks, node failures, and interactions via message passing) we had to rethink several of the key design decisions behind data provenance in the design of DTaP.

We also study how much provenance information needs to be recorded for DTaP, and how much can be reconstructed on demand. We show that, based on a combination of recursive view maintenance and logging/deterministic replay, a recursive query can reconstruct the provenance of an update event that occurred at a specific time in the past. Thus, we obtain two alternative strategies for maintaining provenance: a *proactive* scheme in which provenance deltas are logged, and a *reactive* scheme in which only nondeterministic events (such as incoming messages) are logged. With the reactive approach, the provenance tree can be reconstructed on demand by re-executing portions of the distributed system.

The proactive and the reactive schemes have different tradeoffs: for instance, the proactive scheme tends to require more storage space but can often provide lower query latencies. Thus, the optimal strategy for a given application depends on factors such as query frequency, system runtime, and the ratio of local vs. distributed derivations. To exploit these tradeoffs, we have developed cost models for both schemes, and we describe techniques for determining the optimal maintenance strategy at runtime.

Finally, we present *DistTape*, a DTaP storage and query engine that supports DTaP queries. DistTape uses a declarative networking [12] engine for maintaining and querying the provenance graph, which is coupled with a logging and replay system. DistTape incrementally maintains and queries provenance in a workload-aware fashion, guarantees consistent and complete query results despite network variability (such as instabilities or oscillations), and explains not only why a given datum exists, but also why it has appeared, changed, or disappeared.

Using two example applications (declarative network routing [12] and Hadoop MapReduce [6]) with a combination of realistic network simulations and an actual testbed deployment, we demonstrate that DistTape is able to efficiently maintain and execute provenance queries at scale. Moreover, we validate that our cost model can accurately estimate the system's performance. In summary, this paper contributes:

- The DTaP model for distributed time-aware provenance (Sections 2 and 3);
- Proactive and reactive techniques for maintaining and querying provenance (Section 4);
- Cost models for both techniques, as well as a set of optimizations based on these models (Section 5); and
- An experimental evaluation in the context of DistTape, a practical DTaP storage and query engine (Section 6).

2. SYSTEM MODEL

Before we present DTaP in Section 3, we first introduce a distributed system model (based on distributed Datalog) and some basic concepts that will be useful for our formal definitions. We also describe some key challenges that DTaP needs to address in order to support a range of network forensic capabilities.

2.1 Overview

We consider a distributed system that consists of a set of *nodes* $N = \{N_1, N_2, ..., N_n\}$ that are connected by a network and can communicate by sending messages. The state of a node at a given point in time can be expressed as a set of *tuples* (typically with fixed schemas). We model user input as tuples that are inserted or deleted directly by users, and computations performed by the system as *derivations* of new tuples from existing tuples. We say that a tuple is a *base tuple* if it was inserted directly by a user; otherwise we say that it is a *derived tuple*. Derived tuples can be sent from one node to another as messages.

For concreteness, we use *Network Datalog* (NDlog) [12], a distributed variant of Datalog, to describe the possible derivations and dependencies among tuples that can exist in the system. (However, it should be possible to apply DTaP to distributed systems written in other languages, including legacy systems, as long as the dependencies between incoming and outgoing tuples can be modeled in a similar way; see, e.g., [23, 24]). To illustrate NDlog, we consider the simple example of a MINCOST protocol for network routing, in which the nodes compute the lowest-cost path between each pair of nodes using the following *rules*:

```
mc1 cost(@S,D,C) :- link(@S,D,C).
mc2 cost(@S,D,C) :- link(@Z,S,C1),
        mincost(@Z,D,C2), C=C1+C2.
mc3 mincost(@S,D,MIN<C>) :- cost(@S,D,C).
```

As in traditional Datalog, each NDlog rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`p` should be derived whenever `q1`, `q2`, . . . , and `qn` all exist at the same time". NDlog supports a *location specifier* in each predicate, which is written as an `@` symbol followed by the node on which the tuple resides. For example, any `cost` tuples that are derived via rule `mc1` should reside on the same node as the corresponding `link` tuples, since both carry the same location specifier `@S`.

In this program, the base tuple `link(@S,D,C)` exists if node `S` has a direct link to node `D` with cost `C`. The tuple `cost(@S,D,C)` is derived when `S` has a (possibly indirect) path to `D` with total cost `C`, which can either be a direct link (`mc1`) or a path through another node `Z` (`mc2`). Rule `mc3` aggregates all paths with the same sources and destinations to compute the minimal path cost.

In NDlog, the tuples themselves are stored in relational tables; each relation can have an optional primary key. The protocol runs continuously, and tuples can be derived or underived in response to changes to base tuples. For instance, `mincost` tuples may be updated if the cost of a link changes, since this can change the lowest-cost route.

2.2 NDlog Execution Model

Next, we briefly describe the execution model for NDlog rules. Our treatment here is informal and example-based; formal definitions and proofs are deferred to Section 3.

The execution of a NDlog program consists of insertions and deletions of individual tuples; we refer to these as *events*. Causal dependencies can exist between events; for instance, the insertion of a derived tuple causally depends on the insertion of the tuple(s) from which it was derived. It is these causal dependencies that will be captured by DTaP.

NDlog programs are executed using *pipelined semi-naïve* evaluation (PSN) [12]. PSN first requires rewriting each NDlog rule into *delta rules* of the form `action :- event, conditions, ...`. As an example, the generated delta rules for rule `mc2` in the MINCOST program are:

```
d1 +cost(@S,D,C) :- +link(@Z,S,C1),
        mincost(@Z,D,C2), C=C1+C2.
d2 -cost(@S,D,C) :- -link(@Z,S,C1),
        mincost(@Z,D,C2), C=C1+C2.
d3 +cost(@S,D,C) :- link(@Z,S,C1),
        +mincost(@Z,D,C2), C=C1+C2.
```

```
d4 -cost(@S,D,C) :- link(@Z,S,C1),
         -mincost(@Z,D,C2), C=C1+C2.
```

`d1-d2` and `d3-d4` are delta rules for the `link` and `mincost` predicates, respectively. Rules `d1` and `d3` describe insertions (+), and `d2` and `d4` describe deletions (−). For instance, in rule `d2`, `-link` is the event, `mincost` is the condition predicate, and `-cost` is the action that is taken when the event occurs and the condition holds.

For rules with aggregates (e.g. `mc3`), a similar set of insert/delete delta rules can be generated. The main difference here is that the action would result in an update of an aggregate in the rule head.

Since derivations can involve tuples on remote nodes (such as the rule `mc2` above), nodes must notify each other when they locally derive a tuple that could trigger a derivation on a remote node. This is done by sending a message that encodes the update. We assume that each transmitted message is eventually received, if retransmitted sufficiently often. Messages can be reordered by the network.

In PSN evaluation, each node has an *update pool* in which it buffers all updates whose effects have not yet been applied to its tuple set. The pool is initially empty, but updates can be added due to local inputs or due to messages received from other nodes. Whenever a node's pool is not empty, the node deterministically picks an update according to some policy[1], applies it to the tuple set, and then determines whether any additional (un)derivations have been triggered by the change. If so, the corresponding updates are added to the pool. This process continues until the pool becomes empty.

2.3 Execution Traces

The execution of an NDlog program can be characterized by the sequence of events that take place; we refer to this sequence as an *execution trace*. An execution trace can be used to explain a derivation that occurred during the execution – we can simply replay it and check which event triggered the derivation, and which conditions held at that time. A full trace can recursively explain all derivations; if we are only interested in some specific derivations (e.g., the ones queried by the network operator), a subtrace is generally sufficient.

Figure 1 shows an example scenario during the execution of the MINCOST program. At some past time t_2, the network protocol has changed its min-cost path between node `c` and `a` in response to updated link information that claimed there existed a shorter path between the two nodes. Figure 2 shows a part of the corresponding execution during which `+mincost(c,a,4)` is derived. The explanation for this event consists of the following subtrace (event tuples are highlighted in **bold**):

- At time $t_2@b$, node `b` discovered a new link to node `a` and thus inserted the base tuple **`+link(@b,a,1)`**.
- Rule `mc1` was triggered by `+link(@b,a,1)`, resulting in **`+cost(@b,a,1)`**.
- Rule `mc3` was used to derive **`+mincost(@b,a,1)`** from `+cost(@b,a,1)`.
- Rule `mc2` (specifically its delta rule `d3`) was triggered by `+mincost(@b,a,1)`. The condition was satisfied by the existing tuple `link(@b,c,3)` that had been derived at time t_0; the resulting update **`+cost(@c,a,4)`** was shipped to node `c`.
- At time $t_3@c$, `c` received `+cost(@c,a,4)` from `b` and derived **`+mincost(@c,a,4)`** using rule `mc3`, which replaced the higher-cost **`mincost(@c,a,5)`**.

[1]PSN execution requires FIFO processing of updates for eventual consistency, but other policies could be used instead.

Note that the ordering of edges (arrows) in Figure 2 reflects causality, in the form of a *happens-before* relationship. For example, `+link(@b,c,1)` happens before `+cost(@b,a,1)` as a result of executing rule `mc1`.

2.4 Challenges and Requirements

The aim of DTaP is to provide an explanation for the derivation of any network state. For example, in Figure 2, a network operator may issue a query asking for the explanation of `-mincost(@c,a,5)` at a particular time t_3 at node `c`. DTaP's explanations should provide the entire chain of events, leading from `+link(@b,c,3)` at time t_3. To illustrate why this is a challenging problem, we consider the following realities in any distributed systems:

- **Continuous processing.** Distributed systems run continuously: nodes constantly process new information and update their state in response to local events and incoming messages from other nodes. Thus, a given tuple might have existed at time t_x but not at time t_y, or it might have existed at both times, but for different reasons. DTaP should store enough information to return the correct explanation for a given time.
- **Updates.** Sometimes it is important to understand not only why a certain tuple exists, but also why it has appeared or changed. For example, to understand the route update presented in Figure 1, one would not only need to understand the derivation of the latest route, but also explain why the previous route was replaced by the current one. Existing distributed provenance engines such as ExSPAN [25] are unable to deal with recording explanations that evolve over time, let alone provide an explanation that causally links `-mincost(@c,a,5)` and `+mincost(c,a,4)` at time t_3.
- **Lack of synchrony.** There is no 'global' time that could be used to order events. For instance, when `c` received the message in Figure 2, its local clock might show an earlier time than `b`'s clock when it sent the message! Also, since information takes time to propagate from node to node, there may not be a single, globally consistent explanation: if a tuple is obtained through a long chain of derivations from tuples on other nodes, some of the underlying tuples may already have changed or disappeared. Hence, DTaP must capture time and dependencies at a logical level, based on rule execution and tuple instances.
- **Network effects.** Messages can be delayed and reordered. For instance, if `link(a,b,1)` is added and withdrawn within a short period of time, `mincost(@c,a,4)` would subsequently also be derived and deleted in quick succession, increasing the likelihood that the insert and subsequent delete messages are reordered in the network. The explanations of `+mincost(@c,a,4)` or `-mincost(@c,a,4)` should still be accurate in the presence of such message orderings. Message delays further complicate this, since reordering can separately happen to the actual network derivations and to the corresponding provenance metadata.

3. DTAP MODEL

We use NDlog's execution model as a basis for formalizing DTaP. Given a distributed system in NDlog, DTaP provenance data are used to provide an explanation as to why a given tuple τ or update event is located on node N_i at time t. Tuple τ can be viewed as a *materialization point* that applies a sequence of the update events on τ. Intuitively, the answer for a provenance query on the existence of τ on node N_i at time t can be formulated as a sequence of

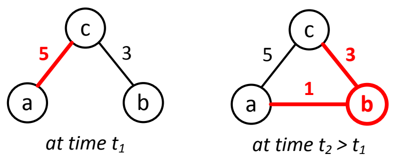


Figure 1: An example network, where the best path between node `c` and `a` changed at time t_2, due a change of the network topology.

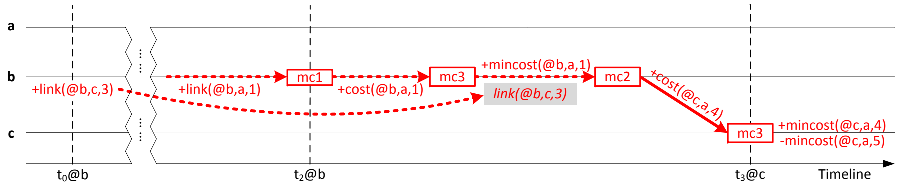


Figure 2: An execution subtrace of the MINCOST program that corresponds to scenario in Figure 1 and provides an explanation of +mincost(@c,a,4). Rectangles indicate that a rule is fired, dashed arrows indicate local event triggering, solid arrows indicate cross-node messages, and shaded boxes indicate the conditions for events.

query results for the update events (up to time t) on τ.[2] Hence, we focus our discussion on the provenance of update events.

3.1 Definitions

We now provide a more formal definition of the terms that were introduced in the preceding section. Our definitions are in terms of delta rules as described in Section 2.2.

Definition (Update): *An* update *is either $+\tau$ or $-\tau$, where τ is a tuple that is being derived $(+)$ or underived $(-)$. We write $\triangle\tau$ to denote an update of either type.*

Definition (Event): *An* event $d@N_i = (e, r, t, c, e')$ *represents the fact that delta rule r was triggered by update e and generated a set of updates e' at time t (relative to N_i's local clock), given the precondition c. The precondition c is a set of tuples that existed on N_i at time t that are used in the event.*

Definition (Trace): *A* trace $\mathcal{E}$ *of a system execution is an ordered sequence of events* $d_1@N_{i_1}, d_2@N_{i_2}, ..., d_m@N_{i_m}$.

Definition (Subtrace): *A* subtrace $\mathcal{E}' \subseteq \mathcal{E}$ *of a trace $\mathcal{E}$ is a subsequence of $\mathcal{E}$, i.e., $\mathcal{E}'$ consists of a subset of the events in $\mathcal{E}$ in the same order. In particular, we write $\mathcal{E}|N_i$ to denote the subtrace that consists of all the events on N_i in $\mathcal{E}$.*

Definition (Happens-before): *Given a (sub)trace $\mathcal{E}$ and two events $d_i@N_i$ and $d_j@N_j$ in $\mathcal{E}$, we consider $d_i@N_i$ to have* happened before *$d_j@N_j$ iff $d_i@N_i$ precedes $d_j@N_j$ in $\mathcal{E}$.*

Note that the *happens-before* relation can order events on different nodes independent of their node-local timestamps.

3.2 Provenance Model

Given an execution trace $\mathcal{E}$ of a specific NDlog program, DTaP encodes the provenance for the trace in a directed graph $G(\mathcal{E}) = (V, E)$, in which each vertex $v \in V$ represents an event in the trace, and each edge $(v_1, v_2) \in E$ represents a direct causal dependency between two such events. This graph is used to answer queries.

Vertices. DTaP's provenance graph can contain the following six types of vertices:

- INSERT$(t, \mathtt{n}, \tau)$ and DELETE$(t, \mathtt{n}, \tau)$: Tuple τ was inserted (deleted) on node `n` at time t.
- DERIVE$(t, \mathtt{n}, r, \tau)$ and UNDERIVE$(t, \mathtt{n}, r, \tau)$: Tuple τ was derived (underived) via delta rule r on node `n` at time t.
- SEND$(t, \mathtt{n}, \triangle\tau, \mathtt{n}')$ and RECEIVE$(t, \mathtt{n}', \triangle\tau, \mathtt{n})$: An update $\triangle\tau$ was sent (received) on node `n` at time t to (from) node `n'`.

[2]For instance, one could internally maintain a pointer for each update that points to the latest previous update on the same tuple. These pointers chain together related updates to ensure fast lookup during query execution.

The last two vertices are needed because a derivation on one node can involve tuples on another; the corresponding messages are represented explicitly in G.

Graph construction. The vertices are connected with edges according to the following guidelines:

- When a base tuple is inserted, an INSERT vertex is added.
- If a node N_i derives a tuple τ via delta rule r, a DERIVE vertex is added, which has incoming edges from all of r's preconditions, as well as from the triggering event, i.e., the INSERT that caused r to fire. The DERIVE vertex is then connected to a new INSERT vertex (if τ is local to N_i) or a new SEND vertex (if τ is sent to another node).
- When a message is received from another node, a RECEIVE vertex is added, with an incoming edge from the corresponding SEND vertex. The RECEIVE vertex is then connected to a new INSERT vertex.
- Whenever an INSERT vertex is added for a tuple τ that already has at least one derivation, an incoming edge is added to τ's most recent INSERT vertex (recall that tuples can have more than one derivation).
- When a tuple τ_1 displaces another tuple τ_2 due to a primary-key or aggregation constraint, an *update edge* is added from τ_1's INSERT vertex to τ_2's DELETE vertex.

The guidelines for deletions and underivations are analogous. Note that the graph is acyclic because edges are always added between an existing vertex and a new vertex, but never between two existing vertices. It is also monotonic because, as the execution $\mathcal{E}$ continues, new vertices and edges can be added but are never removed. Given the instantiated provenance graph $G(\mathcal{E})$, the provenance $G(\triangle\tau, \mathcal{E})$ of an update event $\triangle\tau$ on node N_i at time t is simply the subtree of $G(\mathcal{E})$ that is rooted at the corresponding INSERT(t, N_i, τ) (or DELETE(t, N_i, τ)) vertex.

Example. Let us revisit our running example from the previous sections. Figure 3 shows a piece of the DTaP graph that explains the deletion of the tuple `mincost(@c,a,5)` on node `c` at time t_3 that resulted from a new link between `a` and `b` that was added at time t_2. Consequently, the edge at the DELETE vertex of `mincost(@c,a,5)` (indicated by a dotted line) corresponds to an aggregation constraint — that is, the minimal cost changed because a lower-cost path to node `a` became available. The updated lowest cost (`cost(@c,a,4)`) was derived on node `b` at time t_2 (and subsequently sent to node `c`) because a) a link `b-c` with cost three already existed at time t_2 (since its insertion at time t_0), and b) the tuple `mincost(@b,a,1)` was newly derived at t_2 via rule `mc3`. The latter derivation was caused by the insertion of the base tuple `link(@b,a,1)`, which corresponds to the addition of the new link.

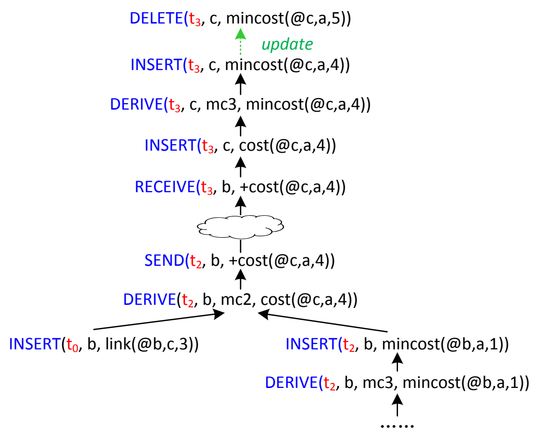

DELETE(t3, c, mincost(@c,a,5))
update
INSERT(t3, c, mincost(@c,a,4))
DERIVE(t3, c, mc3, mincost(@c,a,4))
INSERT(t3, c, cost(@c,a,4))
RECEIVE(t3, b, +cost(@c,a,4))
SEND(t2, b, +cost(@c,a,4))
DERIVE(t2, b, mc2, cost(@c,a,4))
INSERT(t0, b, link(@b,c,3))
INSERT(t2, b, mincost(@b,a,1))
DERIVE(t2, b, mc3, mincost(@b,a,1))
……

Figure 3: The DTaP provenance graph for explaining the deletion of `mincost(@c,a,5)`.

Note that the additional time dimension on the provenance graph enables another use of provenance: querying the *effects* of an update event. For example, if we want to determine how the insertion of the new link `a-b` has affected the system, we can simply locate the corresponding INSERT vertex in the graph and traverse the edges in the reverse direction.

3.3 Derivations and System Snapshots

Consider a delta rule of the form $\Delta\tau$ `:-` $\Delta\tau_1, \tau_2, \dots, \tau_k$. Since DTaP is used to explain a state *change*, i.e., the appearance or disappearance of particular tuples, only the provenance of the triggering tuple $\Delta\tau_1$ is relevant. For example, in Figure 3, `mincost(@c,a,4)` is derived from delta rule `d2` for rule `mc2`. The DERIVE vertex for `mincost(@c,a,4)` at time t_2 is connected to the INSERT vertex for the triggering event `+mincost(@b,a,1)` and the INSERT vertex for `+link(@b,c,3)`. If there are previous updates to `link(@b,c,3)` that occur before time t_2, these updates need to be included as part of the full provenance information. As a storage and querying optimization, instead of storing full provenance information of all preconditions, we introduce a new vertex that provides a compact representation of per-node state at a given time:

- EXIST(t, n, τ): State of tuple τ at a particular node n at its local time t_n. This vertex includes all vertices $\{\text{INSERT}(t, n, \tau) | t \leq t_n\} \cup \{\text{DELETE}(t, n, \tau) | t \leq t_n\}$. To retrieve the snapshot value of τ at time t, one can simply replay the sequence of insertions and deletions, canceling out deleted insertions according to standard bag semantics.

With the use of an EXIST vertex, we can summarize this information: all updates to `link(@b,c,3)` are replaced with a single EXIST vertex that indicates system state at the rule triggering time.

3.4 Correctness

Given the provenance $G(\triangle\tau, \mathcal{E})$ of an update event $\triangle\tau$, G should be "consistent" with the trace representation of the execution $\mathcal{E}$. We say $G(\triangle\tau, \mathcal{E})$ is correct if it is possible to extract a subtrace from G that has the properties of *validity*, *soundness*, *completeness*, and *minimality.* We first describe our subtrace extraction algorithm, followed by the correctness properties themselves.

Subtrace extraction. Given $G(\triangle\tau, \mathcal{E})$, the original subtrace can be obtained by running an algorithm $\mathcal{A}$ based on topological sort. Briefly, $\mathcal{A}$ converts each vertex in the provenance graph to an event and then uses a topological ordering to assemble the events into a trace – in other words, if two vertices $v_1, v_2 \in V$ correspond to events e_1, e_2 and are connected by an edge $(v_1, v_2) \in E$, then e_1 will appear in the trace before e_2. This is possible because the provenance graph is acyclic.

Correctness of subtrace. The extracted subtrace $\mathcal{A}(\Delta\tau, \mathcal{E})$ must satisfy the following four properties (theorems and proofs for all four properties are presented in [21]):

Property (Validity)*: $\mathcal{A}(\Delta\tau, \mathcal{E})$ is* valid *if, given the initial state $\mathcal{S}_I$, for all events $d_i@N_i = (e_i, r_i, t_i, c_i, e'_i) \in \mathcal{A}(\Delta\tau, \mathcal{E})$, (a) there exists $d_j@N_j = (e_j, r_j, t_j, c_j, e'_j)$ that precedes $d_i@N_i$ in $\mathcal{A}(\Delta\tau, \mathcal{E})$, $e_i \in e'_j$, and (b) for all $\tau_k \in c_i, \tau_k \in S_{i-1}$, where $S_0 \xrightarrow{d_1@N_1} S_1 \dots S_{i-2} \xrightarrow{d_{i-1}@N_{i-1}} S_{i-1}$.*

$\mathcal{A}(\Delta\tau, \mathcal{E})$ corresponds to a correct execution of the NDlog program whose provenance is being captured. Any event that triggers a delta rule evaluation must be generated before the rule is evaluated, and the conditions of the rule evaluation must hold at the time of the rule evaluation.

Property (Soundness)*: $\mathcal{A}(\Delta\tau, \mathcal{E})$ is* sound *if it is a subtrace of some $\mathcal{E}'$ that is equivalent to $\mathcal{E}$ (written as $\mathcal{E}' \sim \mathcal{E}$). We say two traces $\mathcal{E}$ and $\mathcal{E}'$ are equivalent if, for all nodes N_i, $\mathcal{E}|_{N_i} = \mathcal{E}'|_{N_i}$.*

$\mathcal{A}(\Delta\tau, \mathcal{E})$ must preserve all happens-before relationships among events in the original execution trace obtained from running the NDlog program. Ideally, we would like $\mathcal{A}(\Delta\tau, \mathcal{E})$ to be a subtrace of $\mathcal{E}$. However, without synchronized clocks, we cannot always order concurrent events on different nodes. For practical purposes $\mathcal{E}$ and $\mathcal{E}'$ must be indistinguishable: each node must observe the same sequence of events in the same order.

Property (Completeness)*: $\mathcal{A}(\Delta\tau, \mathcal{E})$ is* complete *if it ends with the event $\Delta\tau$.*

$\mathcal{A}(\Delta\tau, \mathcal{E})$ must include all events necessary to reproduce τ. Note that the validity property already requires that any event that is needed for $\Delta\tau$ be included in $\mathcal{A}(\Delta\tau, \mathcal{E})$; hence, we can simply verify the completeness property of a valid trace by checking whether it ends with $\Delta\tau$.

Property (Minimality)*: $\mathcal{A}(\Delta\tau, \mathcal{E})$ is* minimal *if no valid trace $\mathcal{E}' \subset \mathcal{A}(\Delta\tau, \mathcal{E})$ is complete.*

$\mathcal{A}(\Delta\tau, \mathcal{E})$ must not include any events that do not contribute to τ.

4. MAINTENANCE AND QUERYING

This section presents techniques for generating and maintaining provenance information modeled by DTaP, followed by distributed query processing strategies for efficiently retrieving provenance data.

4.1 Representation of the Provenance Graph

DistTape uses a distributed Datalog (based on NDlog) evaluation engine to store provenance information. Provenance data are incrementally maintained as distributed views [11] over network state. DistTape makes use of four provenance tables – called `prov`, `ruleExec`, `send`, and `recv` – that are incrementally updated as the NDlog rules that model the protocols are executed. These tables store DTaP's provenance graph in a distributed fashion.

Tuple instances: The `prov` table maintains information about each tuple (including both current tuples and tuples that existed in the past) as well as the specific rule that triggered its derivation. Entry `prov(@N,VID,Time,RLoc,RID)` indicates that the tuple on node `N` with unique identifier `VID` was derived at time `Time` by a rule execution on node `RLoc` that is uniquely identified by `RID`. If `N` and `RLoc` are different, the tuple was sent from `RLoc` to `N`, and this communication is recorded in additional `recv` and `send` entries (see below). `VID` is generated based on a cryptographic hash of the

+/-	Loc	VID	Time	RLoc	RID	Derivation
$+$	b	$VID1 = \texttt{SHA1}(``+link" + b + c + 3 + t_0)$	t_0	$null$	$null$	`+link(@b,c,3)`
$+$	b	$VID2 = \texttt{SHA1}(``+link" + b + a + 1 + t_2)$	t_2	$null$	$null$	`+link(@b,a,1)`
$+$	b	$VID3 = \texttt{SHA1}(``+mincost" + b + a + 1 + t_2)$	t_2	b	$RID1$	`+mincost(@b,a,1)`
$+$	c	$VID4 = \texttt{SHA1}(``+cost" + c + a + 4 + t_3)$	t_3	c	$RID2$	`+cost(@c,a,4)`
$+$	c	$VID5 = \texttt{SHA1}(``+mincost" + c + a + 4 + t_3)$	t_3	c	$RID3$	`+mincost(@c,a,4)`
$-$	c	$VID6 = \texttt{SHA1}(``-mincost" + c + a + 5 + t_3)$	t_3	c	$RID3$	`-mincost(@c,a,5)`

Table 1: An example `prov` relation based on Figure 3. The table is horizontally partitioned across all nodes, based on the location specifier `Loc`. The last column is not stored in the table; it is included here to show the derivation that corresponds to each entry. The first column indicates an insertion ($+$) or a deletion ($-$).

+/-	RLoc	RID	Rule	ExecTime	Event	CList	Derivation
$+$	b	$RID1 = \texttt{SHA1}(``mc3" + b + VID2 + t_2)$	$mc3$	t_2	$VID2$	$null$	`mincost(@b,a,1)`
$+$	b	$RID2 = \texttt{SHA1}(``mc2" + b + VID1 + VID2 + t_2)$	$mc2$	t_2	$VID2$	$(VID1)$	`cost(@c,a,4)`
$+$	c	$RID3 = \texttt{SHA1}(``mc3" + c + VID4 + t_3)$	$mc3$	t_3	$VID4$	$null$	`mincost(@c,a,4)`

Table 2: An example `ruleExec` relation that corresponds to the DERIVE vertices shown in Figure 3. The last column shows the derivation rule that was executed in each instance.

Sender	VID	STime	RID	Derivation
b	$VID4$	t_2	$RID2$	`cost(@c,a,4)`

Receiver	VID	RTime	Sender	STime	Derivation
c	$VID4$	t_3	b	$t2$	`cost(@c,a,4)`

Table 3: Example `send` and `recv` relations that correspond to the SEND and RECEIVE vertices in Figure 3.

contents of the tuple and the time of its derivation; similarly, `RID` is a hash of the rule identifier, node location, and `VID` of the derived tuple. For base tuples, `RID` is set to `null`.

In order to correctly generate the above entries, NDlog programs undergo an automatic rewrite process to include the `RID` and `RLoc` information with each tuple derivation. This process ensures that the appropriate `prov` entry will be generated on the node to which the derivation is sent.

Rule execution instances: The `ruleExec` table maintains information about each execution of a rule (not just about each rule). Entry `ruleExec(@RLoc,RID,Rule,ExecTime,Event,CList)` indicates the execution of a `Rule` on `RLoc` at time `ExecTime`, triggered by an event `Event` (i.e., a tuple that changed, appeared, or disappeared) while the preconditions in `CList` were holding.

Message transmissions and arrivals: The `send` and `recv` tables maintain information about message exchanges. `send(@Sender,VID,STime,RID)` and `recv(@Receiver,VID, RTime,Sender,STime)` refer to the rule execution identified by `RID` that affected the tuple identified by `VID`; the corresponding message was sent by `Sender` at time `STime` and received at time `RTime`. Whenever a rule execution causes a message to be sent, `send` and `recv` entries are generated at the sender and receiver, respectively, and are timestamped using each node's local clock. To handle clock skew, the receiver stores the sender's timestamp at message transmission; this timestamp is included in each message along with the (un)derived tuple. This information is used during query processing to correctly match up `send` and `recv` entries.

Given the distributed nature of provenance storage, these tables are naturally partitioned based on their first attributes, and distributed among the nodes. For instance, `prov` entries are co-located with the tuples to which the update events were applied, and `ruleExec` entries are located on the nodes on which the rule executions were performed.

Tables 1, 2, and 3 show the entries for the tables above, based on the example provenance tree shown in Figure 3. The vertices defined by our provenance model (Section 3.2) are encoded in the above provenance tables as follows: INSERT and DELETE vertices are respectively represented as tuple insertions (`+prov`) and deletions (`-prov`). Likewise, DERIVE and UNDERIVE are stored as `+ruleExec` and `-ruleExec`. Edges between INSERT / DERIVE and DELETE / UNDERIVE pairs are represented by the `RID` and `VID` pairings in each `prov` entry. `recv` and `send` entries correspond to the RECV and SEND vertices. For each tuple uniquely identified by its primary key, each EXIST vertex consists of all updates (i.e., `+prov` and `-prov`) ordered by their timestamps.

4.2 Maintaining Provenance with Delta Rules

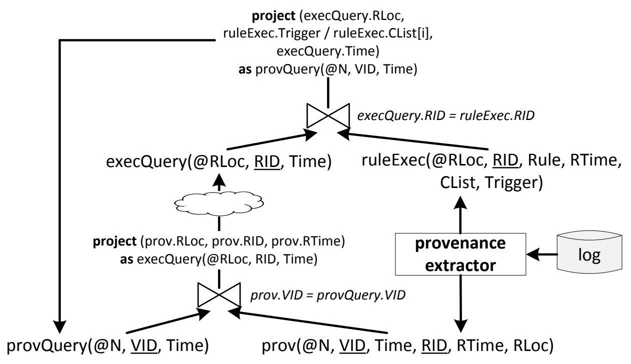
The DTaP graph can be captured via the evaluation of delta rules (see Section 2.2) of the form `action :- event, conditions`. In a delta rule of the form $\triangle p$ `:-` $p_1, \ldots, \triangle p_i, \ldots, p_n$, the *event* (in this case, $\triangle p_i$) is represented as an INSERT or DELETE vertex, the *conditions* (the other p_k) are represented as a sequence of INSERT (or DELETE) vertices that support the existence of p_k (EXIST vertex), and the *action* ($\triangle p$) is represented as a DERIVE or UNDERIVE vertex.

When a delta rule $\triangle p$ `:-` $p_1, \ldots, \triangle p_i, \ldots, p_n$ is fired at time t, DistTape performs the following steps:

- Generate a `+ruleExec` or `-ruleExec` tuple with timestamp t to represent the rule execution, and maintain pointers to the triggering event $\triangle p_i$ and preconditions $p_1, \ldots, p_n$ (excluding p_i).
- Generate a `+prov` or `-prov` tuple with timestamp t to represent the insertion or deletion event $\triangle p$, and to maintain a pointer to the generated `+/-ruleExec` tuple.
- If the generated event $\triangle p$ needs to be sent to another node, generate a pair of `send` and `recv` tuples at the sender and the receiver, respectively, with timestamps that correspond to each node's local clock.
- Finally, if the generated event $\triangle p$ results in a violation of a primary-key or aggregation constraint (e.g., the newly-generated tuple displaces another), generate an additional `+prov` or `-prov` tuple to represent the deletion caused by $\triangle p$. This corresponds to the update edge from Section 3.2.

4.3 Proactive and Reactive Maintenance

To answer provenance queries about past tuples or updates, the DTaP model contains a temporal dimension. This could in principle be implemented with provenance versioning, i.e., by keeping a full copy of the provenance whenever it changes. However, this would require an enormous amount of storage, particularly for long-running distributed systems with frequent updates. Moreover,



project (execQuery.RLoc, ruleExec.Trigger / ruleExec.CList[i], execQuery.Time) as provQuery(@N, VID, Time)
execQuery.RID = ruleExec.RID
execQuery(@RLoc, RID, Time)
ruleExec(@RLoc, RID, Rule, RTime, CList, Trigger)
project (prov.RLoc, prov.RID, prov.RTime) as execQuery(@RLoc, RID, Time)
provenance extractor
log
prov.VID = provQuery.VID
provQuery(@N, VID, Time)
prov(@N, VID, Time, RID, RTime, RLoc)

Figure 4: Logical query plan for recursive provenance queries. Underlined attributes are primary keys.

provenance versioning is unnecessary because DTaP's provenance is *monotonic*: the provenance of historic updates and tuples (which eventually make up a major portion of a provenance graph) is immutable.

For better efficiency, DistTape maintains provenance incrementally, i.e., it considers only the 'deltas' between adjacent versions, which are sufficient to reconstruct the full provenance graph. DistTape can store these deltas in two different ways:

- **Explicit deltas (proactive).** In this approach, all of the `+prov`, `-prov`, `+ruleExec`, and `-ruleExec` entries are stored explicitly in a temporally ordered log that is indexed by time. Compared to provenance versioning, the storage cost is considerably lower; however, the full provenance information must be reconstructed from the deltas before a query can be answered. To permit fast reconstruction of EXIST vertices during query execution (see Section 4.4), DistTape maintains reverse-time ordered pointers between all `+/- prov` entries that correspond to the same tuple. For instance, in Table 1, the `prov` entry with `VID4` (`+mincost(@c,a,4)`) points to the entry with `VID5` (`-mincost(@c,a,5)`) which in turn points to another entry for `+mincost(@c,a,5)`.
- **Per-node input logs (reactive).** In this approach, DistTape maintains only the non-deterministic inputs (`recv` entries for incoming messages, as well as tuple insertions and deletions) at each node. If the underlying application is deterministic, DistTape can replay these inputs at query time to reproduce the original execution of that node, and reconstruct the provenance on the fly. As an optimization, each derived tuple sent across nodes needs only to include the sender's timestamp.

The first approach represents a *proactive* style of provenance maintenance in which provenance information is stored explicitly in the form of deltas; the second approach represents a *reactive* style in which provenance information is reconstructed at query time. There exists a tradeoff between the two: the proactive approach results in lower query latencies (since there is less overhead for reconstruction) but requires more storage space.

4.4 Query Execution

For ease of exposition, we limit our discussion to the scenario in which the query result of interest is the entire provenance for the given update $\triangle\tau$ at time t. To query the provenance of such an update, DistTape executes a distributed recursive query that reconstructs the relevant subtree of the provenance graph from the four tables we have described in Section 4.1. Figure 4 shows the logical query plan for evaluating this distributed recursive query; the query starts at the root of the subtree and iteratively adds vertices and edges until a fixpoint is reached (at the base tuples). The results are then returned in the form of tuples from the `prov`, `ruleExec`, `send`, and `recv` tables that encode the relevant subtree.

In Figure 4, the initial query is represented as an input tuple `provQuery(@N,VID,Time)` to the logical plan. Based on this tuple, DistTape carries out the following steps:

- **Step 1: Retrieve rule execution instances.** Since the `VID` uniquely identifies $\triangle\tau$, DistTape uses it as a lookup into the `prov` table (via a database join) and then retrieves the corresponding `RID` used to derive the tuple, as well as the location `RLoc` at which the rule was fired. This corresponds to the generation of the DERIVE or UNDERIVE vertex. If `RLoc` is different from `Loc` (i.e., the tuple was derived from a remote rule execution), additional RECV and SEND vertices are generated by joining the `VID`s of derived tuples with the `recv` and `send` tables[3]; for readability, these extra operations have been omitted from Figure 4. Next, DistTape generates `execQuery` tuples to trigger queries on the `ruleExec` table.
- **Step 2: Expand dependent derivations.** DistTape ships the resulting `execQuery(@RLoc,RID,Time)` tuple to `RLoc` and there joins it with the local `ruleExec` table to recursively expand the child derivations that have resulted in $\triangle\tau$. Here, multiple additional `provQuery` tuples are generated: one for the trigger event for the delta rule `RID`, and another for each condition predicate value that occurred during the execution of `RID`. Each expansion generates an INSERT or DELETE vertex, depending on whether the trigger event was an insertion or a deletion, and each expanded condition generates an EXIST vertex, which includes additional INSERT and DELETE vertices to explain why the condition held at the relevant `Time`.
- **Repeat until fixpoint.** Steps 1 and 2 are performed recursively until all child nodes are expanded. As the query progresses, the `provQuery` events are recursively propagated from the root of the provenance tree (where the queried update resides) towards the child nodes in order to construct the entire subtree. Each level of the tree can be expanded in parallel at different nodes. Upon reaching the leaf nodes (which correspond to base tuples), the query results are returned back to the root along the reverse path. At each level, the parent node only returns its portion of the query result (subtree) after all the child nodes have completed their respective subqueries.

For our prototype, we have implemented the query plan from Figure 4 in NDlog, and we execute it on a distributed recursive query engine [12]. To customize the query and to return other annotations of provenance [9], DistTape supports user-defined functions for augmenting the query plan.

4.5 Reconstructing Provenance on Demand

Steps 1 and 2 assume that the entire `prov` and `ruleExec` tables are already constructed and available when the query is issued. To support the proactive and reactive maintenance techniques from Section 4.3, DistTape needs an additional provenance extraction operator to reconstruct the `prov` and `ruleExec` entries from deltas or input logs whenever a `provQuery` or `execQuery` is received.

Reconstruction with provenance deltas. When the log consists of deltas, the `prov` entries are reconstructed as follows. Given a query

[3]After retrieving the `recv` entry based on `VID` and `RTime`, we use the `STime` (sender's timestamp) attribute in `recv` to fetch the appropriate `send` entry on the sender's side. This avoids explicit time synchronization.

`provQuery(@N,VID,Time)`, the provenance extractor is invoked using `VID` and `Time` as the lookup keys. Using a fast binary-search data structure indexed by time, the extractor searches the log for an entry corresponding to `VID` at time `Time`, and returns the corresponding `+/-prov` tuple. In some cases, if an EXIST vertex is required, the provenance extractor first finds the latest `prov` entry at time `Time` and then follows the chain of updates backwards in time to retrieve all tuples with a `VID` smaller than `Time`. An analogous mechanism is used to reconstruct the `ruleExec` tuples by searching for the corresponding `RID` and `RTime`.

Reconstruction with input logs. In the reactive implementation, instead of searching for the appropriate `prov` and `ruleExec` entries, reconstruction involves replaying the entire log (messages and changes to base tuples) at the relevant nodes until the specified input time (`Time` or `RTime`) is reached. Each `recv` entry from the log is replayed on a reference implementation of the distributed system to regenerate `prov` and `ruleExec` entries.

In theory, one can always start replaying the input logs from the very beginning of the system execution. However, this can be costly when the application has been active for a long time, particularly if the derivation rules are computationally expensive. DistTape reduces this overhead by periodically recording a *checkpoint*. Each node only checkpoints its local state – specifically, the currently extant tuples and any unprocessed updates in the local pool. This is sufficient because each node replays only its local execution, and it allows DistTape to avoid the complex mechanisms needed for consistent global checkpoints. The input log can then be incrementally applied, starting from the latest checkpoint.

A possible optimization is to cache `prov` and `ruleExec` entries from previous replays in case they can be used in a subsequent query. This avoids unnecessary replays. Additional methods for improving querying and maintenance performance are discussed in the next section.

5. COST-BASED OPTIMIZATIONS

The maintenance approaches introduced in the previous section offer a spectrum of tradeoffs between maintenance overhead and querying performance. The best tradeoff depends on a variety of factors, some of which we discuss below.

Querying frequency. We expect that the cost for query processing will be a function of (1) how frequently queries are issued, (2) how far apart the checkpoints are in the log, and (3) how much work is required to replay a log segment. If queries are expected to be rare, we can save space by maintaining input logs and taking checkpoints only occasionally. In this case, answering a query can be expensive because the relevant parts of the provenance graph must be reconstructed by replaying the execution of certain nodes from their latest checkpoint.

If queries are more frequent, we can trade some space for a lower query-processing cost by (1) taking checkpoints more frequently, which reduces the expected length of the log segment that needs to be replayed, and/or (2) maintaining provenance deltas rather than input logs. The latter reduces the computational cost because replay needs only to incrementally apply the changes to the provenance data, instead of repeating the processing steps that produced them.

System runtime. Many distributed systems run for an indefinite amount of time. For example, the Internet's interdomain routing system has been running for decades. In such systems, checkpoints are indispensable because it is not practical for the querier to replay the execution of the system, or even just a single node, from the very beginning. On the other hand, there are distributed systems that run only for a limited time. For example, in this case, replaying the entire log may be practical, and if so, we can save even more space by not maintaining checkpoints at all.

NDlog Program	
N_{pred}	# of predicates in rule body
N_{dep}	Node-level depth of the provenance graph
N_{exec}	# of derivation rules triggered by a message
N_{dup}	# of duplicate derivations per tuple
System Performance	
S_{log}/S_{chk}	Size of a log entry / checkpoint
T^{w}_{log}/T^{r}_{log}	Time taken to append / retrieve a log entry
T^{w}_{chk}/T^{r}_{chk}	Time taken to save / load a checkpoint
T_{rule}	Time taken to execute a delta rule
$T_{latency}$	Average propagation delay between two nodes
Input Workload	
F_{msg}	Message freq. (# of messages per unit time)
F_{qry}	Query freq. (# of queries per unit time for the entire system)
I_{chk}	Checkpoint interval (# of unit time between adjacent checkpoints)

Table 4: Summary of the statistics used for optimizations.

Local derivations. Distributed systems differ in the relative frequency of remote derivations (i.e., derivations that involve message exchanges between nodes). When most derivations are remote, both provenance deltas and input logs should perform equally well since most state changes (which are recorded in provenance deltas) are due to incoming messages (which are recorded in the input logs). However, there are systems where most derivations are local; for example, a distributed machine-learning algorithm might just send a very few messages to transfer the raw data and the results. In this case, input logs should consume significantly less space than provenance deltas, but they would need much more computation when the provenance graph needs to be reconstructed to answer a query.

5.1 Cost Model

In order to decide which maintenance strategy(-ies) to adopt, we develop a cost model that captures the tradeoffs between maintenance and querying overhead. Our cost model takes as its inputs a set of runtime statistics collected from the system, including workload properties (e.g., message and query frequencies), the characteristics of the running protocol (e.g., the ratio of local derivations), and the measured overhead for reading and writing log entries.

The model parameters are summarized in Table 4 and are broadly classified into three categories: (1) properties of the NDlog program that the system executes, (2) properties of the nodes on which the system is deployed, and (3) properties of the workload. Unless otherwise specified, model parameters are system-wide; they are obtained by profiling the deployed system at runtime. Each node first averages its local statistics (e.g., the number of messages per unit time), and then the results are averaged across all nodes.

NDlog program. The first set of parameters relate to the distributed protocol itself. Since we have assumed (in Section 2.1) that the protocol is specified in NDlog, the properties are expressed in terms of the structure of the program. N_{pred} denotes the average number of predicates in a rule body; for instance, in the MINCOST program from Section 2.1, rule `mc2` has a complexity of two because its rule body contains the predicates `link` and `mincost`. N_{dep} denotes the average node-level depth of a provenance tree for any tuple derived using the program. Note that this is not the same as vertex-level depth; for instance, in Figure 3, $N_{dep} = 2$, since the graph is partitioned at nodes c and b. N_{exec} is the number of derivation rules that are triggered (executed) by an incoming message at a given node. This includes all local rules executed until a local fixpoint is reached. For instance, in the MinCost program from Section 2.1, an incoming `path` message will trigger rule `r3`, which may further trigger rule `r2` (if the received `path` is optimal). Finally, N_{dup} is the average number of duplicate derivations per tuple. N_{pred} can be analyzed statically from the protocol specification, whereas N_{dep} and N_{exec} are collected at runtime, e.g., by observing the previous provenance query results and rule executions.

System performance. The next set of properties relates to the runtime environment in which the distributed system is deployed, e.g. with respect to I/O, computation power, and message propagation delay in the network. S_{log} and S_{chk} are the average sizes of a log entry and a checkpoint, respectively; T^w_{log} and T^r_{log} are the times required to append and retrieve a log entry; T^w_{chk} and T^r_{chk} are the times required to save and load a checkpoint; T_{rule} is the average execution rule execution time; and $T_{latency}$ is the average propagation delay between two nodes.

Input workload. The final set of properties relate to external inputs that drive the execution of the protocol. For example, the rate at which links are updated in the MinCost program has a direct impact on the number of times rules are fired on a node and the size of the log. Rather than capture the rate of change at a predicate level, we instead measure the average frequency of incoming messages at each node during protocol execution, denoted as F_{msg}. Unlike other parameters, F_{qry} represents a system-wide total, i.e., it represents the number of queries issued to the entire network.

DistTape captures parameters at a coarse granularity (typically, system-wide averages of per-node averages). Our model can capture costs at finer granularity, e.g., at the level of individual relational operators, but the requisite fine-grained data leads to a massive number of parameters. As we show in Section 6, coarse-grained statistics are sufficient for the model to provide accurate estimates on actual system performance.

5.2 Applying the Cost Model

The proactive and reactive provenance maintenance techniques offer storage and latency tradeoffs: the proactive scheme has a higher log storage overhead but offers lower query latency than the reactive strategy. DistTape applies the above cost model to estimate the storage and latency overheads, and then, as discussed in Section 5.3, selects the strategy that is likely to perform best.

- **Storage overhead.** In the proactive approach, the log stores `+/-prov`, `+/-ruleExec`, and incoming `recv` tuples as they are generated by the underlying protocol. Each log entry requires S_{log} space. With a message frequency of F_{msg}, the `recv` tuple yields a storage overhead of $F_{msg} \times S_{log}$, and the `prov` and `ruleExec` yield $2 \times F_{msg} \times N_{exec} \times S_{log}$.

  For the reactive approach, DistTape needs only maintain one log entry for each incoming message. Here, the storage overhead is $(F_{msg} \times S_{log}) + (I_{chk} \times S_{chk})$, where the former term denotes the cost of storing the messages and the latter is the cost of storing the checkpoints.

- **Query latency**. We consider the time taken to construct a vertex (T_{vertex}) in the provenance graph in both the proactive and reactive cases.

  In the proactive approach, given a delta rule execution, three log retrievals are required to get the `recv`, `prov`, and `ruleExec` entries for the trigger event. We repeat this retrieval process for all duplicate derivations. Hence, $T_{vertex} = 3 \times T^r_{log} \times N_{dup}$.

  In the reactive approach, checkpoints need to be retrieved and replayed, requiring T^r_{chk} time. The expected number of log messages that must be retrieved and replayed by executing rules between two checkpoint intervals is $F_{msg} \times I_{chk}$. Hence, $T_{vertex} = T^r_{chk} + F_{msg} \times I_{chk} \times N_{exec} \times T_{rule}$.

  To estimate the average per-query latency, we multiply by the average depth of the tree, and the time to replay each vertex (T_{vertex}) and propagation delays in sending the query results along the path of the tree, resulting in $N_{dep} \times (T_{vertex} + 2 \times T_{latency})$. By taking into account the query frequency (F_{qry}), one can further estimate the aggregate latency of all queries over a period of time.

5.3 Optimizations using the Cost Model

The above cost model can be used to select the maintenance mode that is likely to perform best according to a given metric. The choice of proactive or reactive provenance depends upon the underlying protocol and workload, as well as the metric (i.e., storage or query latency) that the administrator of the distributed system would like to optimize. If the administrator's goal is simply to minimize storage (resp. latency), then the straightforward approach of selecting the maintenance mode that incurs the least storage (resp. latency) cost is sufficient. More flexibility can be achieved with more complex objective functions; for instance, the administrator can adopt a strategy that uses reactive provenance when the estimated latency is lower than some threshold value.

Selecting the appropriate provenance maintenance mechanism can be done either prior to system deployment or at runtime. In the former approach, which we have adopted for our prototype implementation, DistTape relies on *performance profiles* that capture the costs of a representative system. Cost-based analysis of the performance profile determines whether proactive or reactive provenance maintenance is likely to yield better performance.

Another possibility is *dynamic adaptation*, that is, adjusting the provenance maintenance mechanism at runtime. This requires the ability to reconstruct provenance from a log that contains a mix of provenance deltas and per-node input logs. Here, DistTape takes advantage of the property that the two maintenance modes are interchangeable. For instance, given a query for a `prov` entry at time `t`, DistTape can start replaying the log from the most recent checkpoint before `t`. If a `recv` message is encountered, the execution rules are fired to derive the corresponding `prov` and `ruleExec` tuples. On the other hand, if a `+/-prov` delta is read from the log, it can similarly be used to directly update the `prov` table.

6. EVALUATION

We have built a DistTape prototype based on the RapidNet declarative networking engine [16]. We extended RapidNet by adding logging and replay capabilities to its distributed query processor in order to support time-aware provenance graphs and process time-based provenance queries.

DistTape enables time-aware network provenance for both existing declarative networks as well as legacy systems. In the latter case, dependencies at each node must be extracted and modeled as distributed Datalog programs. The question of how to extract provenance from applications and how to support legacy systems has been studied in prior work (cf. [23]) and existing techniques can be used for this purpose. For example, if the legacy system's source code is available, dependencies can be explicitly reported to DistTape by adding upcalls to the code; if the code is not available, provenance can be inferred by observing the nodes' inputs and outputs and by correlating them with an external specification of the

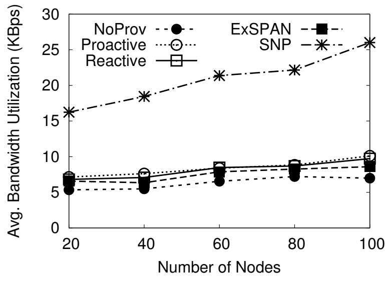


Figure 5: Average per-node bandwidth utilization (KBps).

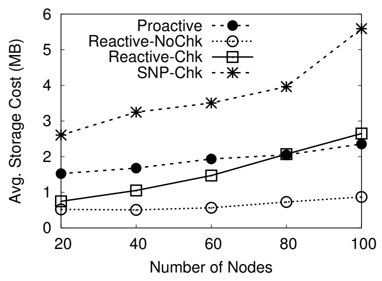


Figure 6: Average per-node storage overhead (MB).

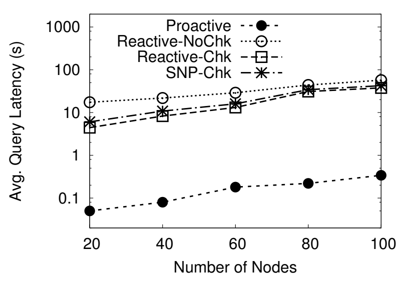


Figure 7: Query latencies (seconds) for various network sizes.

application's protocol. Given the availability of existing techniques to extract provenance, we concentrate our evaluation on DistTape's ability to efficiently provide time-aware provenance.

We evaluate DistTape using the following maintenance techniques from Section 4: (1) `NoProv` is a baseline in which no provenance information is maintained or queried; (2) `Proactive` maintains provenance proactively using logs to store provenance deltas; and (3) `Reactive-Chk` and `Reactive-NoChk` maintain provenance reactively, with (`Chk`) or without (`NoChk`) checkpointing using per-node input logs.

We compare DistTape to two prior distributed provenance systems: `ExSPAN` [25], which maintains provenance only for the derivations of *current* system state (and not historical state), and (2) `SNP-Chk` (Secure Network Provenance [23]), which supports tamper-evident historical provenance with periodic checkpointing. To make the results comparable, we use the same checkpoint interval for both `Reactive-Chk` and `SNP-Chk`.

We measure DistTape's main overheads: storage overhead due to logs, communication overhead from provenance maintenance and querying messages, CPU overhead due to executing rules and maintaining provenance, query latency from network propagation delays, and computation at each node due to log replays.

6.1 Maintenance Overhead and Query Latency

Our first set of experiments executes multiple instances of DistTape using the ns-3 [14] network simulator. (ns-3 is a discrete event-driven network simulator that, like its predecessor ns-2, emulates all layers of the network stack and supports configurable loss, packet queuing, and network topology models.) By first running DistTape over ns-3, we can study the scalability trends of DistTape in a more controlled and repeatable environment. In Section 6.2, we will revisit a similar set of experiments on a smaller scale within our local cluster testbed.

Our experiments focus on the provenance maintenance and querying overheads of a declarative path-vector routing protocol called PATHVECTOR that consists of four rules. PATHVECTOR is an extension of the MINCOST protocol from Section 2.1; it stores an additional path attribute that encodes the actual path from a given source to a destination. PATHVECTOR computes the shortest path (with minimal hop count) between any two nodes. We have chosen to evaluate PATHVECTOR because path-vector protocols are used widely; for example, they serve as the basis for the Internet's inter-domain routing protocol.

For our experiments, we use the declarative version of PATHVECTOR that was previously developed by Loo et al. [12]. The program is executed continuously over a dynamic network; as links are updated, new `mincost` tuples are derived, and provenance logs are generated for subsequent reconstruction. Each DistTape node runs the PATHVECTOR protocol; in steady state, we add or delete two links every second. Our simulations last for 300 virtual seconds and are executed on machines with X3450 Xeon 2.66 GHz processors and 4 GB memory that run Fedora 12 (64-bit).

Provenance bandwidth overhead. In our first experiment, we vary the number of simulated DistTape nodes from 20 to 100. Figure 5 shows the average amount of bandwidth used by each node; since checkpointing increases only local storage and does not result in any additional communication, we show only the results for the reactive maintenance without checkpoints (labeled as `Reactive`). As expected, `NoProv`'s bandwidth usage is the lowest. We also observe that `Proactive` and `Reactive` incur roughly the same bandwidth overhead. The reason is that both techniques add a few attributes (such as sender timestamp, address, and tuple identifier) to each message that is sent as part of the protocol execution. In both cases, the per-node bandwidth requirement in the 100-node network is less than 10.5 KBps. In comparison, `ExSPAN`'s overhead is lower – only 88% of that of DistTape's `Proactive` and `Reactive` – because ExSPAN messages do not contain timestamps for tracking historical information. `SNP` is the most expensive: it consumes 2.5 times more bandwidth than DistTape. The overhead is due to a 1,024-bit signature and an acknowledgment that `SNP` includes in each message it transmits.

Provenance storage overhead. DistTape's per-node storage overheads are shown in Figure 6. We report only the storage used for logs (i.e., the overhead incurred by DistTape), so we omit the results for `NoProv` and `ExSPAN`, which maintain no logs. In `Reactive-Chk` and `SNP-Chk`, checkpoints are taken once every minute.

`Proactive` produces the largest logs because it explicitly stores each change to the `prov` and `ruleExec` tuples. `Reactive-NoChk` produces the smallest logs, at approximately one third of the size of the `Proactive` logs. The storage overhead for `Reactive-Chk` grows linearly with network size. Beyond a network size of 80, `Reactive-Chk` actually exceeds `Proactive`, since checkpoints grow with the size of the network. However, we note that the crossover point depends on additional factors such as the update rate of base tuples. Overall, the storage overhead is modest: in the largest network (100 nodes), it amounts to 2.65 MB in 300 seconds (approximately 9 KBps). The logs maintained by `SNP-Chk` and `Reactive-Chk` are fairly similar, and their checkpoints have roughly the same size; nevertheless, `SNP-Chk`'s storage overhead is more than twice as high because it must also record the signatures and acknowledgments in the log.

Query latency. Our second set of experiments evaluates the query latency of `Proactive`, `Reactive-NoChk`, and `Reactive-Chk`. We use the same experimental setup as above. Additionally, we randomly issue 100 queries from different nodes, each of which retrieves the provenance of a particular tuple derived in the past.

Figure 7 shows the average query latency for various network sizes. Since query latency depends on the depth of each provenance tree (which is bounded by the network diameter), we observe a diminishing increase in query latency as network size increases.

With `Proactive`, each query's latency is dominated by network propagation delay; hence, the average query latency is within 0.34 seconds. `Reactive-Chk` must additionally replay the logs from a given checkpoint, resulting in a higher average query latency: 37.7 seconds on average, for a network size of 100. `SNP-Chk` needs 4.8 seconds more than `Reactive-Chk` because it must also perform integrity checks on the checkpoints and logs during replay, which involves verifying the cryptographic signatures. As expected, `Reactive-NoChk`'s query latency is the highest because it must replay the *entire* log; however, recall from Figure 6 that `Reactive-NoChk` also has a much lower storage overhead than the other schemes, so there is a tradeoff between storage overhead and latency. We do not consider `ExSPAN` for this experiment because it does not support historical queries.

6.2 Cost Model Validation on Actual Testbed

We next validate the cost model presented in Section 5.1. Our goal is to show that the model accurately predicts the storage and latency costs in a variety of different configurations. In particular, we show that, based on measurements of the parameters described in Table 4 (e.g., number of predicates, message frequencies, etc.) and the storage and query formulas specified in Section 5.2, DistTape accurately estimates the storage overhead and query latency.

We repeat the previous experimental setup on actual physical machines in a local cluster testbed. Here, we utilize machines with a similar hardware/software configuration as the machine used in simulation. The machines are connected using high-speed Gigabit Ethernet. All DistTape nodes run the same code as before, but communicate using actual network sockets instead of ns-3's simulated network stack.

Path-vector. We deploy the path-vector protocol on 60 DistTape nodes (utilizing 60 cores on 15 cluster machines) for a duration of 900 seconds. Table 5 shows the differences between the estimated per-node storage (predicted using the cost model) and the corresponding actual storage overhead for each experimental run, repeated for different link update intervals. In both `Proactive` and `Reactive-Chk` (with a per-minute checkpoint interval), our results indicate that our cost model is accurate: the differences between the estimated and the measured storage costs range from 0.7% to 20.5%. Unlike the simulation results, the storage overhead for `Reactive-Chk` is significantly higher than `Proactive`. This is largely due to the relatively large checkpoints (compared to the I/O logs) that dominate the storage cost for `Reactive-Chk`. Our cost model is able to provide accurate estimation, such that users can fine-tune the checkpoint intervals accordingly.

Table 6 shows the differences between estimated and measured query latencies in a similar setup. The execution time is lower than in the earlier simulations, since our testbed experiments are carried out with lower update frequency, resulting in shorter log replay times between checkpoints. As before, our cost model accurately predicts query latencies for different frequencies of link updates; the differences between the estimated and measured latencies range from 0.0% to 10.0%.

Hadoop. We next validate the accuracy of our cost model on Hadoop MapReduce (version 1.0.0) [6]. We modified Hadoop so that the dependencies between incoming and outgoing tuples are reported to DistTape and modeled as NDlog rules. Briefly, MapReduce consists of a *map* followed by a *reduce* phase. In the Map phase, each Map worker applies the user-defined Map function on each input tuple, and then locally combines intermediate results based on the partitioning key. In the *reduce* phase, a reduce worker combines the

Upd. Interval (seconds)	**Proactive**		**Reactive-Chk**	
	Estimated	*Actual*	*Estimated*	*Actual*
4	1.0 MB	0.83 MB	4.4 MB	4.01 MB
2	1.4 MB	1.41 MB	4.5 MB	4.41 MB
1	2.2 MB	2.15 MB	4.8 MB	5.03 MB

Table 5: Comparisons between estimated storage (obtained from the cost model) and actual measured storage (in MB) for the path-vector protocol.

Upd. Interval (seconds)	**Proactive**		**Reactive-Chk**	
	Estimated	*Actual*	*Estimated*	*Actual*
4	0.011 s	0.010 s	2.7 s	2.5 s
2	0.012 s	0.012 s	3.3 s	3.6 s
1	0.013 s	0.014 s	4.9 s	4.7 s

Table 6: Comparisons between estimated average query latency (obtained from cost model) and actual measured end-to-end latency (in seconds) for the path-vector protocol.

outputs from Map workers and performs the reduce function. The dependency logic between incoming and output tuples of the map and reduce phases can be modeled as two NDlog rules each. Hence, the number of rules triggered per input (Table 4) is $N_{exec} = 2$.

We run Hadoop's WordCount program on up to 40 cores within our cluster. The program (WordCount) counts the number of occurrences of each word, given an input document size of 9.1 GB derived from the WebBase dataset (dated 12/2010). In the first setup, 100 map tasks and 40 reduce tasks are executed, while in the second setup, 40 map tasks and 16 reduce tasks are executed.

Unlike the earlier path-vector experiment, the MapReduce execution is a one-time execution of static input data (as opposed to a continuously executed routing protocol). Since there is no need to perform periodic checkpoints for the duration of the program execution, we compare only `Proactive` and `Reactive-NoChk`. Table 7 shows that the `Proactive` approach incurs 12 times larger storage overhead than `Reactive-NoChk` (which only needs to maintain the original input files and the intermediate data communicated from map workers to reduce workers). This is largely due to the fact that the SHA1-based IDs assigned to `prov` and `ruleExec` entries are significantly larger than the input keywords: on average, a word is around 14 B (due to a large amount to HTML-specific tags), whereas each `prov` (or `ruleExec`) tuple is 35 B[4], i.e., the average size of a log entry (Table 4) is $S_{log} = 35$ B. Previous work [8, 15] has shown that MapReduce-specific optimization is possible to further reduce the size of log entries, however, we decide to retain the general-purpose provenance encoding in our evaluation.

Our estimated storage overhead is calculated based on the formula introduced in Section 5.2: the storage overhead is $N_{msg} \times (S_{log} + 2 \times N_{exec} \times S_{log})$, where N_{msg} denotes the number of input records to the map and reduce workers. Given that each input event triggers *exactly* two derivation rules, our cost model accurately estimates the storage overhead based on the number of input records to the map and reduce workers (reported by Hadoop).

Table 8 summarizes the average query latencies. We observe that, as expected, `Proactive` results in a lower query latency since no replay is necessary. Nevertheless, even with replay, `Reactive-NoChk` returns each query within 66 seconds. Replay latency is higher when the number of map/reduce workers is reduced since each worker is responsible for a larger set of input tuples (hence increasing the replay overhead of reexecuting the spe-

[4]In the Hadoop experiment, we use the first 80 bits of the SHA1 hashes for the IDs of `prov` and `ruleExec` tuples.

Number of Map/Reduce	Proactive		Reactive-NoChk	
	Estimated	*Actual*	*Estimated*	*Actual*
100/40	115.6 GB	115.6 GB	9.7 GB	9.7 GB
40/16	113.9 GB	113.9 GB	9.5 GB	9.5 GB

Table 7: Comparisons between estimated storage (obtained from the cost model) and actual measured storage (in GB) for Hadoop MapReduce.

Number of Map/Reduce	Proactive		Reactive-NoChk	
	Estimated	*Actual*	*Estimated*	*Actual*
100/40	6 s	8 s	41 s	47 s
40/16	15 s	19 s	62 s	66 s

Table 8: Comparisons between estimated average query latency (obtained from cost model) and actual measured end-to-end latency (in seconds) for Hadoop MapReduce

cific worker to generate dependency logic between incoming / outgoing tuples). In all cases, we note that our cost-model provides a good estimation of query latency.

7. RELATED WORK

Provenance [1] has proven to be a versatile concept. It has been successfully applied to a variety of areas, including probabilistic databases [17, 19], collaborative databases [5], file systems [7, 13], and scientific computations [3]. In contrast to existing provenance models, DTaP explicitly represents distributed state, includes a temporal dimension, and supports provenance of state changes.

ExSPAN [25] describes how data provenance can be efficiently distributed and queried in a distributed system using a relational encoding, horizontal data partitioning, and a distributed Datalog implementation. However, unlike DistTape, ExSPAN supports only provenance of the *current* state of a distributed system; it cannot answer queries about state changes or past system states.

SNP [23] can answer provenance queries in distributed systems that have been partially compromised by a malicious adversary. SNP supports a form of historical provenance, but its use of tamper-evident logs and cryptographic signatures incurs high computation and storage costs. Although appropriate for the adversarial settings in which it is designed to operate, SNP may be overly burdensome for many other applications. DistTape can be seen as a generalization of SNP to many different storage models, and its cost-based techniques can be used to adaptively choose the best strategy for a given application.

An initial version of DistTape's provenance model has been presented in a workshop paper [22]. The present paper adds (1) a refined model that is a provably sound and complete representation of causal dependencies between events in a distributed system; (2) a system for maintaining and querying provenance that combines distributed recursive view maintenance, logging, and system replay; (3) cost-based optimization techniques for choosing between proactive and reactive styles of provenance maintenance/querying; and (4) an implementation and an experimental evaluation.

8. CONCLUSION

In this paper, we have presented DistTape, a time-aware provenance system for querying distributed systems for changes in state over time. We have proven that the DTaP model satisfies four correctness properties (validity, soundness, completeness, and minimality), and have developed a complete prototype implementation. Our evaluation demonstrates the efficiency of provenance maintenance and querying, and showcases various proactive/reactive maintenance strategies with and without the use of checkpointing. These techniques have different tradeoffs in storage, computation, and bandwidth, which motivate our use of cost functions for deciding which strategy to use based on collected system statistics.

Acknowledgments

This work is partially supported by NSF grants CCF-0820208, CNS-0845552, CNS-1054229, CNS-1065130, CNS-1149832, CNS-1204347, CNS-1223825 and IIS-0812270, and AFOSR MURI grant FA9550-08-1-0352. This material is based upon work supported by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4020. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Project Agency and Space and Naval Warfare Systems Center Pacific.

9. REFERENCES

[1] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc. ICDT*, pages 316–330, 2001.
[2] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
[3] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *Proc. SIGMOD*, pages 1345–1350, 2008.
[4] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating internet routing instabilities. In *Proc. SIGCOMM*, pages 205–218, 2004.
[5] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *Proc. VLDB*, pages 675–686, 2007.
[6] Hadoop. http://hadoop.apache.org/.
[7] R. Hasan, R. Sion, and M. Winslett. Preventing history forgery with secure provenance. *ACM Trans. Storage*, 5(4):1–43, 2009.
[8] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *Proc. CIDR*, pages 273–283, 2011.
[9] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *Proc. SIGMOD*, pages 951–962, 2010.
[10] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005.
[11] M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. Maintaining recursive views of regions and connectivity in networks. *IEEE Trans. on Knowl. and Data Eng.*, 22:1126–1141, 2010.
[12] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking. *Commun. ACM*, 52:87–95, 2009.
[13] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX ATC*, pages 43–56, 2006.
[14] Network Simulator 3. http://www.nsnam.org/.
[15] H. Park, R. Ikeda, and J. Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12):1351–1354, 2011.
[16] RapidNet. http://netdb.cis.upenn.edu/rapidnet/.
[17] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *Proc. VLDB Endowment*, 1(1):797–808, 2008.
[18] R. Teixeira and J. Rexford. A measurement framework for pin-pointing routing changes. In *Proc. ACM SIGCOMM Network Troubleshooting Workshop*, pages 313–318, 2004.
[19] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. CIDR*, pages 262–276, 2005.
[20] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *Proc. NSDI*, pages 57–70, 2011.
[21] W. Zhou. Secure time-aware provenance for distributed systems. University of Pennsylvania Ph.D. dissertation, 2012. Available at http://netdb.cis.upenn.edu/papers/zhou-thesis.pdf.
[22] W. Zhou, L. Ding, A. Haeberlen, Z. Ives, and B. T. Loo. TAP: Time-aware provenance for distributed systems. In *Proc. TaPP*, pages 1–6, 2011.
[23] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proc. SOSP*, pages 295–310, 2011.
[24] W. Zhou, Q. Fei, S. Sun, T. Tao, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. NetTrails: A declarative platform for provenance maintenance and querying in distributed systems. In *Proc. SIGMOD*, pages 1323–1326, 2011.
[25] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD*, pages 615–626, 2010.