TAP: Time-aware Provenance for Distributed Systems

Wenchao Zhou Ling Ding Andreas Haeberlen Zachary Ives Boon Thau Loo
University of Pennsylvania
{wenchaoz, lingding, ahae, zives, boonloo}@cis.upenn.edu

Abstract

In this paper, we explore the use of provenance for analyzing execution dynamics in distributed systems. We argue that provenance could have significant practical benefits for system administrators, e.g., for reasoning about changes in a system's state, diagnosing protocol misconfigurations, detecting intrusions, and pinpointing performance bottlenecks. However, to realize this vision, we must revisit several aspects of provenance management. As a first step, we present *time-aware provenance (TAP)*, an enhanced provenance model that explicitly represents time, distributed state, and state changes. We outline our research agenda towards developing novel query processing, languages, and optimization techniques that can be used to efficiently and securely query time-aware provenance, even in the presence of transient state or untrusted nodes.

1 Introduction

Provenance [2] has proven to be a versatile concept. It has been successfully applied to a variety of areas [3, 4, 7, 16, 17, 19], and this paper proposes an addition to this list. We observe that, in the context of distributed systems, it is very common for system administrators to perform analysis tasks that essentially amount to *network provenance* [21] queries. For example, they might ask diagnostic queries to determine the root cause of a malfunction [18], forensic queries to identify the source of an intrusion [10], or profiling queries to find the reason for suboptimal performance [20]. At the heart of all these queries is a question about data flows across nodes – in other words, a question about provenance. Thus, we should be able to bring to bear many of the techniques originally developed for provenance in other domains.

Prior work [21] has provided initial evidence that one can efficiently maintain and query provenance in distributed systems, even at Internet scale. However, to support the full range of functionality required for analyzing distributed systems, we must still address several open challenges. 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. Existing provenance systems cannot easily answer this question because it a) does not ask about the provenance of state, but rather about a state *change*; and b) it does not ask about state that currently exists, but rather about state that existed in the past.

Some existing systems [1, 16] do retain certain provenance data about disappeared state and state changes; for example, PASS [16] maintains versions of provenance, and infers the causes for state changes by comparing old and current versions. This is different from the property we have in mind. First, we want to explicitly capture *causality*: if a file A depends on a thousand other files and one of these files, say B, is changed, we want the provenance of the change in A to be attributed to the change in B. Second, since one of our potential use cases is forensics, we are interested in strong *security guarantees*, i.e., we would like provenance queries to be answerable even if an adversary is actively trying to cover his traces. Provenance should be remain accessible even if the adversary deletes telltale files, or even compromises some of the nodes on which the provenance is stored.

In summary, we see the main challenges as follows:

- **Challenge #1: Transient and inconsistent state.** We need new techniques for maintaining and querying provenance, such that consistent and complete query results are guaranteed despite network variability, such as instabilities or oscillations.
- **Challenge #2: Explanations for state changes.** We need an efficient mechanism that can explain not only why a certain datum exists, but also why it has appeared, disappeared, or changed.
- **Challenge #3: Security without trusted nodes.** We need a provenance system that can correctly answer provenance queries even if an attacker has managed to compromise some part of the system.

As a starting point, we introduce *time-aware provenance* (TAP), a novel provenance model that addresses the first two challenges. We also outline a research agenda towards addressing further aspects of provenance management in distributed systems. These include (1) new provenance models and maintenance strategies for capturing the time, distribution, and causality of updates, (2) novel query processing and optimization techniques for efficiently and securely answering queries at scale, and (3) provenance query languages that enable a declarative specification of time and changes.

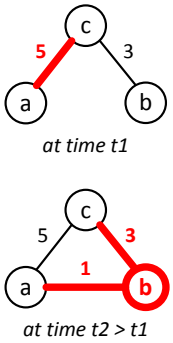

Figure 1: An three-node example network. The best path between node a and node c (highlighted) changes in response to a topology change.

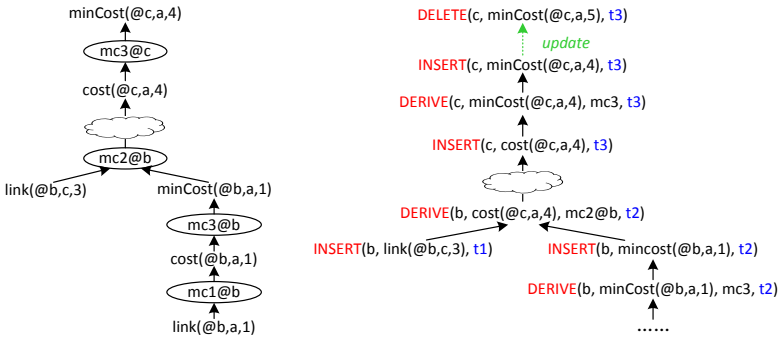

Figure 2: Comparison between classical provenance (left) and time-aware provenance (right). The tree on the left explains why the cheapest path from c to a has cost 4; the tree on the right explains why the cost has *changed* from 5 to 4. Message transmissions between nodes are indicated by a cloud.

2 Background

To set the stage for our subsequent discussion, we first describe a basic provenance model, which we will extend in Section 3 to arrive at TAP.

2.1 System Model

We assume that the distributed system consists of a set of *nodes*, and that the state of each node can be expressed as a set of *tuples* (typically with fixed schemas). The execution logic is encoded in a set of *derivation rules* that specify how tuples can be derived from each other or from *base tuples*, which correspond to inputs. For simplicity, we will assume that the derivation rules are explicit. This is the case, e.g., for systems that are written in a declarative language such as Network Datalog (NDlog) [13]. However, TAP is not specific to NDlog and can be applied to systems implemented using imperative languages.

As a concrete example, we show the rules for a very simple routing protocol, MINCOST [1], that computes the lowest cost between each pair of nodes in a network:

```
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).
```

Note particularly that the derivation rules include state from different nodes. In NDlog, this is expressed with the *location specifier* @, which is followed by the name of the node on which the tuple resides. In this system, 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. The protocol runs continuously, and updates path costs as links appear or disappear.

2.2 Network Provenance

Existing provenance models, e.g., the model in [2] and [21], capture the dependencies between tuples in a graph that consists of *tuple vertices* and *rule execution vertices*, and in which the edges represent data flows. Decentralized models for distributed systems – also known as *network provenance* models – partition this graph in some way, e.g., by the location specifiers, and store each partition on a different node.

Consider the execution of the MINCOST protocol in the three-node network that is shown in Figure 1. Initially, at time t_1, the system is in a quiescent state. Then, at time $t_2 > t_1$, a link with cost 1 is inserted between nodes a and c, which changes the topology of the network. As a consequence, the cost of c's cheapest path to a changes from 5 to 4; however, due to communication delays, the change is not propagated to c until time $t_3 > t_2$. The left part of Figure 2 shows the provenance of the resulting tuple mincost(@c,a,4), which is generated from rule mc3 at node c. It is based on the tuple cost(@c,a,4), which corresponds to the path c-b-a.

This example illustrates two limitations of existing network provenance systems. First, existing systems can correctly answer provenance queries for the updated tuple once the system is in a stable state again (after time t_3); however, the answer to such a query can be incorrect or incomplete between t_2 and t_3, when some nodes have already received the update and others have not. While inconsistencies due to transient state could

[1]The MINCOST protocol can be extended to specify more complex routing protocols, such as distance-vector and path-vector.

be resolved by maintaining provenance in bi-temporal databases [8, 11], we propose, in the next section, a provenance model that *inherently* captures temporal information. Second, because causality[2] is not explicitly represented in the provenance graph, it is difficult to trace a state change back to its root causes.

3 Time-aware Provenance

Time-aware provenance (TAP) addresses these limitations by adding the following two features: First, in addition to storing dependencies between tuples that currently exist, TAP also remembers dependencies between tuples that existed *at some point in the past*, which enables TAP to provide consistent answers to provenance queries even while the system is in a transient state. Second, TAP's provenance model explicitly represents tuple changes, as well as the dependencies between them.

Vertices. TAP's provenance graph contains the following four types of vertices:

- INSERT$(\mathtt{n}, \tau, t)$ and DELETE$(\mathtt{n}, \tau, t)$: Tuple τ was inserted (deleted) on node `n` at time t;
- DERIVE$(\mathtt{n}, \tau, R, t)$ and UNDERIVE$(\mathtt{n}, \tau, R, t)$: Tuple τ was derived (underived) via derivation rule R on node `n` at time t.

The right half of Figure 2 shows a piece of the TAP graph that would have been generated in the example scenario from Section 2.2. Overall, the graph shows that the tuple `mincost(@c,a,5)` was deleted on node `c` at time t_3 because the new link `a-c` was inserted at time t_1. For example, the node DERIVE(`mc2@b`) shows that `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_1), and b) the tuple `mincost(@b,a,1)` was newly derived at t_2 via rule `mc3`. Note that, among the immediate predecessors of a DERIVE (or UNDERIVE) vertex, the INSERT (or DELETE) with the most recent timestamp is the event that triggered the rule. 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.

Interestingly, the additional time dimension on the provenance graph enables another use of provenance, namely querying the *effects* of a state change. 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 and traverse the edges in the reverse direction.

[2]Note that TAP's concept of causality is typically referred to as *data lineage* in the database literature. Our notion of causality differs from [14, 15], which take the lineage information to infer 'actual' contributions of base tuples to the query results.

Edges. In most existing provenance models, the edges represent data flows. TAP's provenance graph contains these edges as well, but, in order to answer queries about state changes, it additionally needs to capture a 'causality flow' between updates. In many cases, the two flows are aligned, but there are cases where they differ. For example, if a primary-key constraint exists in the system, the derivation of a tuple τ_1 may cause the deletion of a tuple τ_2 that shares τ_1's primary key, even though no data flows from τ_1 to τ_2. A similar situation can occur for other types of constraints, such as aggregation. To represent such causality flows, TAP's provenance graph includes additional *update edges*.

The right part of Figure 2 contains an instance of such an edge at the DELETE vertex of `mincost(@c,a,5)` (indicated by a dotted line). This deletion was caused by the aggregation constraint, i.e., the minimal cost changed because a lower-cost path to node `a` became available.

Derivations. The TAP graph can be captured via the evaluation of *delta rules* of the form `action :- event, conditions, ...`. These rules can be obtained by rewriting the original derivation rules, using standard techniques from incremental view maintenance [5]. Thus, it should be possible to leverage existing distributed query processing engines with only minor changes. Briefly, for each derivation rule `p :-` p_1`,` p_2`, ...,` p_n, we generate two delta rules for each predicate p_i – one for insertions and the other for deletions. The rules are of the form $\triangle p$ `:-` p_1`, ...,` $\triangle p_i$`, ...,` 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, and the action itself ($\triangle p$) is represented as a DERIVE or UNDERIVE. Each action then in turn causes a new event.

4 Provenance Maintenance

The graph representation of TAP can be stored in relational tables in a format similar to that in [21]. Each vertex can be maintained as a tuple according to the schema presented in Section 3, along with an additional attribute that stores the (potentially distributed) pointers to its direct contributing vertices.

In theory, the additional time dimension could be implemented by performing *provenance versioning*, i.e., by keeping a copy of the provenance tables whenever the data dependencies change. However, the storage cost would be enormous, especially in distributed systems that run for a long time with continuous updates. We discuss three alternative approaches that are likely to be more efficient; each comes with its own set of tradeoffs.

4.1 Three approaches

Provenance deltas. Instead of maintaining the full provenance information in each version, we can only record the deltas between adjacent versions. This reduces the storage cost considerably, but, when answering a query, the deltas need to be incrementally applied to regenerate the full provenance information.

Per-node input logs. If each node runs a deterministic algorithm, we can choose not to actively maintain provenance during execution time at all. Instead, we can simply record all the inputs (such as network messages, disk reads, etc) at each node. During a query, we can use deterministic replay to reproduce the system execution, and we can generate provenance on the fly.

System input logs. To reduce the storage overhead even further, we can record only the raw inputs (i.e., the base tuples) of the entire system. If the system's execution is deterministic, we can replay it based on the recorded inputs and generate the provenance information as before. This approach comes at the expense of higher querying overhead: since only the raw system-wide inputs are recorded, we cannot independently replay a single node; instead, we must replay the *entire* system execution.

To avoid replaying the deltas (or input logs) from the very beginning of the system execution, each node can periodically record a *checkpoint* of its current state. Thus, replay can start from the latest checkpoint. To save space, the checkpoints could be discarded after a certain amount of time.

4.2 Tradeoffs

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 by 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 only needs to incrementally apply the changes to the provenance data, but not repeat 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, a MapReduce cluster might be set up to process just a small number of large jobs, and many multiplayer games only last for a few hours. 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.

Local derivations. Distributed systems differ in the relative frequency of remote derivations, which involve message exchanges between nodes, and purely local derivations. 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 a lot less space than provenance deltas, but they would need a lot more computation when the provenance graph needs to be reconstructed to answer a query.

Trust. If all the nodes in the distributed system are trusted, we can safely optimize for query performance. However, if the system is large enough, it almost inevitably contains, at any given time, some nodes that are faulty or have been misconfigured or compromised. In this case, provenance deltas are risky because the querier cannot easily see whether a given delta was recorded correctly. Input logs are safer because the querier itself regenerates the provenance. Ideally, the correctness and completeness of the inputs in the logs would be verifiable through some other means.

In summary, there is no clear 'winner' among the three approaches we have proposed in Section 4.1; rather, the best approach depends on the specific use case in which TAP is applied. It would be interesting to design a provenance system that can adaptively select the best approach at runtime – for example, based on the observed query frequency and the workload characteristics.

5 Provenance Querying

To query TAP's provenance graph, a suitable query language is needed. We are currently working on TapQL, an extension of ProQL [9] that incorporates new language primitives to enable query specifications for time and changes. To illustrate, the following TapQL query can be used to analyze the (transitive) effects of a link insertion at a particular time in protocol execution:

```
FOR [-mincost $X] <-+ [+link $Y]
WHERE $Y.time>t
INCLUDE PATH $X <-+ $Y
RETURN $X
```

`[-mincost $X]` binds variable `X` to the DELETE vertices of `mincost`, and `[+link $Y]` binds `Y` to the INSERT vertices of `link`. `INCLUDE PATH $X <-+ $Y` confines the search within the subgraph between vertices `X` and `Y`. The query returns the DELETE vertices of the `mincost` tuples that are triggered by the insertion of `link` tuples (with timestamps bigger than t).

If TAP maintains both forward and backward edges (i.e., from causes to effects and vice versa), provenance queries can ask for causality or effect chains, or even a combination of both. For example, to analyze the effects of a `link` insertion, one iterates through all the INSERT `link` vertices and follows the forward edges to reach DELETE `mincost` vertices. On the other hand, to query the cause of a deleted `mincost` vertex would require edge traversal in the opposite direction. We are planning to enhance TapQL to allow users to specify either cause or effect queries.

5.1 Querying Strategy

To process TapQL queries, we propose a novel multi-staged query processing strategy (shown in Figure 3), which is specifically optimized for replays of provenance deltas or input logs. Our proposed strategy consists of the following three stages:

- A *macroquery* iterates through potential candidate tuples, e.g., all the `mincost` tuples in the example query, and issues microqueries to determine the provenance of each candidate tuple. Microqueries can be evaluated in parallel to optimize the overall query latency.
- A *microquery* performs a distributed recursive evaluation for the provenance of a single tuple. In essence, this amounts to a recursive traversal of the provenance graph [21] until base tuples are reached. When a tuple satisfies the constraints in the query, the microquery should return the provenance of that tuple in the desired form, e.g. as a set of base tuples.

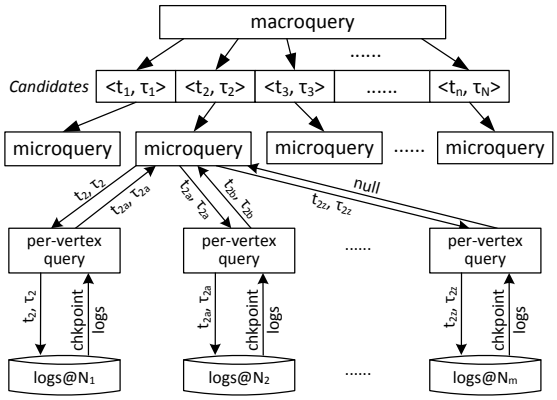

Figure 3: The multi-staged querying strategy. $\langle t_x, \tau_x \rangle$ refers to a tuple τ_x at a given time t_x.

- A *vertex query* returns the set of vertices that are adjacent to a given vertex, as well as the corresponding edges. Vertex queries are the basic building block of the provenance querying. For instance, in Figure 3, $\langle t_{2a}, \tau_{2a} \rangle$ is the returned result for $\langle t_2, \tau_2 \rangle$. Vertex queries can be answered by replaying the provenance deltas or input logs to reproduce the relevant parts of the provenance graph.

5.2 Optimizations

There are several opportunities for optimizations at each stage of our proposed querying strategy. For instance, we can apply the following two optimizations, originally proposed in [21], at the microquery level:

- *Early query termination.* During microquery evaluation, the query can be terminated early if the results are guaranteed to be in (or out of) the result set. This is common for queries that compute monotonically increasing aggregate values with selection predicates, e.g., a query for vertices with more than a given number of unique derivations. In this case, the results can be returned (or discarded) as soon as the constraint is satisfied (or can no longer be satisfied).
- Another optimization is to *cache* query results, and reuse them for answering subsequent queries. Note that cache invalidation is not needed – as the query results are the *fact* of the execution history, and thus will not change over time.
- The overall querying performance would greatly benefit from reducing the overhead of performing vertex queries, which is carried out by replaying provenance deltas (or input logs). Existing work [12] has proposed techniques that allow

efficient incremental view maintenance using the already-captured provenance information.

One can also apply optimizations at the macroquery level. Traditional database optimizations, such as cost estimation of alternative query plans and heuristics, should generally be applicable here. For instance, if a provenance query involves tuples in multiple relations, we can start from relations with low cardinalities.

Interestingly, we can trade query response time for communication overhead at the macroquery level by using cached query results at the microquery level. Instead of issuing all microqueries at once, the query processor can choose to execute one microquery at a time, to maximize the likelihood of cache hits for subsequent queries.

6 Secure Provenance Querying

So far, we have assumed that all the nodes cooperate with the querier. However, in large distributed systems, it is not uncommon for some of the nodes to be faulty or compromised by an adversary. When a system contains such compromised nodes, provenance could be useful as a forensic tool; however, the adversary could attempt to cover his traces by deliberately tampering with the provenance information, causing query results to be incorrect and/or incomplete. Thus, it would be useful to have a provenance system that can give correctness guarantees even when it is under attack.

Not all adversaries are equally powerful. Often, adversaries only manage to compromise non-privileged software on the affected nodes. If the provenance information is extracted and maintained by a privileged component, such as the operating system kernel or a hypervisor, it is still possible to answer queries correctly. However, sometimes adversaries manage to compromise even privileged components, and can effectively take complete control over the affected nodes. During such attacks, correct answers to provenance queries would be particularly useful, but they are also particularly difficult to obtain.

We are currently working on a system that can answer distributed provenance queries even when some nodes have been completely compromised by an adversary. A perfect solution to this problem is impossible: for example, if the provenance of a tuple τ is stored on a set S of nodes, a query for the provenance of τ cannot be answered correctly if the adversary manages to compromise all the nodes in S. However, based on ideas from tamper-evident logging and auditing [6], it is possible to obtain a practical system with only slightly weaker guarantees.

Acknowledgments

This work was supported by NSF grants IIS-0477972, IIS-0713267, CNS-0721541, IIS-0812270, CCF-0820208, CNS-0845552, CNS-1040672, CNS-1054229, and AFOSR MURI grant FA9550-08-1-0352.

References

[1] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42, 2004.

[2] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc. ICDT*, 2001.

[3] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *Proc. SIGMOD*, 2008.

[4] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *Proc. VLDB*, 2007.

[5] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proc. SIGMOD*, 1993.

[6] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, Oct 2007.

[7] R. Hasan, R. Sion, and M. Winslett. Preventing history forgery with secure provenance. *ACM Trans. Storage*, 5(4):1–43, 2009.

[8] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. A glossary of temporal database concepts. *SIGMOD Record.*, 21:35–43, 1992.

[9] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *Proc. SIGMOD*, 2010.

[10] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005.

[11] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Trans. on Knowl. and Data Eng.*, 10:1–20, 1998.

[12] 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.

[13] 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.

[14] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. *IEEE Data Eng. Bull.*, 33(3):59–67, 2010.

[15] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endowment*, 4:34–45, 2010.

[16] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX ATC*, 2006.

[17] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *Proc. VLDB Endowment*, 1, 2008.

[18] R. Teixeira and J. Rexford. A measurement framework for pinpointing routing changes. In *Proc. ACM SIGCOMM Network Troubleshooting Workshop*, Sep 2004.

[19] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. CIDR*, 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*, 2011.

[21] 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*, 2010.