Zeno: Diagnosing Performance Problems with Temporal Provenance

Yang Wu *Facebook*   Ang Chen *Rice University*   Linh Thi Xuan Phan *University of Pennsylvania*

Abstract

When diagnosing a problem in a distributed system, it is sometimes necessary to explain the *timing* of an event – for instance, why a response has been delayed, or why the network latency is high. Existing tools offer some support for this, typically by tracing the problem to a bottleneck or to an overloaded server. However, locating the bottleneck is merely the first step: the real problem may be some other service that is sending traffic over the bottleneck link, or a machine that is overloading the server with requests. These off-path causes do not appear in a conventional trace and will thus be missed by most existing diagnostic tools.

In this paper, we introduce a new concept we call *temporal provenance* that can help with diagnosing timing-related problems. Temporal provenance is inspired by earlier work on provenance-based network debugging; however, in addition to the functional problems that can already be handled with classical provenance, it can also diagnose problems that are related to timing. We present an algorithm for generating temporal provenance and an experimental debugger called Zeno; our experimental evaluation shows that Zeno can successfully diagnose several realistic performance bugs.

1 Introduction

Debugging networked systems is already difficult for functional problems, such as requests that are processed incorrectly, and this has given rise to a rich literature on sophisticated debugging tools. Diagnosing timing-related problems, such as requests that incur a high delay, adds another layer of complexity: delays are often nondeterministic and can arise from subtle interactions between different components.

Performance debugging has already been explored in prior work. For instance, distributed tracing systems [55, 48, 58, 21, 33, 28, 37, 34, 64, 18, 44] can record and analyze executions of a request. These systems offer operators a lot of help with debugging performance problems; for instance, Dapper [55] produces a "trace tree" – a directed graph whose vertices represent execution stages and whose edges represent causal relationships. If the operator observes that a request is taking unusually long, she can inspect its trace tree and look for bottlenecks, such as the RPCs to an overloaded server. Similarly, network provenance systems [69, 67, 61, 30, 60], such as DTaP [68], can be used to generate a causal explanation of an observed event, and the operator can then inspect this explanation for possible bottlenecks.

However, in practice, locating a bottleneck is only the first step. The operator must then find the *causes* of the bottleneck in order to fix the problem. Existing tools offer far less help with this step. For instance, suppose a misconfigured machine is sending a large number of RPCs to a storage backend, which becomes overloaded and delays requests from other clients. When the operator receives complaints from one of the clients about the delayed requests, she can inspect the trace tree or the provenance and identify the bottleneck (in this case, the storage backend). However, neither of these data structures explains *why* the bottleneck exists – in fact, the actual cause (in this case, the misconfigured machine) would not even appear in either of them!

The reason why existing approaches fall short in this scenario is that they focus exclusively on *functional causality* – they explain why a given computation had some particular result. This kind of explanation looks only at the direct inputs of the computation: for instance, if we want to explain the *existence* of a cup of coffee, we can focus on the source of the coffee beans, the source of the cup, and the barista's actions. In contrast, *temporal causality* may also involve other, seemingly unrelated computations: for instance, the reason *why it took too long* to get the cup of coffee might be the many customers that were waiting in front of us, which in turn might be the result of a traffic jam elsewhere that caused an unusually large number of customers to pass by the local store. At the same time, some functional dependencies may turn out to be irrelevant when explaining delays: for instance, even though the coffee beans were needed to make the coffee, they may not have contributed to the delay because they were already available in the store.

The above example illustrates that reasoning about temporal causality is very different from reasoning about functional causality. This is not a superficial difference: as we will show, temporal causality requires additional information (about event ordering) that existing tracing systems do not capture. Thus, although systems like Dapper or DTaP do record timestamps and thus may appear to be capable of reasoning about time, they are in fact limited to functional causality and use the timestamps merely as an annotation.

In this paper, we propose a way to reason about temporal causality, and we show how it can be integrated with an existing diagnostic technique – specifically, network provenance. The result is a technique we call *temporal provenance* that can reason about *both* functional *and* temporal causality. We present a concrete algorithm that can generate temporal provenance for distributed systems, and we describe Zeno, a prototype debugger that implements this algorithm. We have applied Zeno to seven scenarios with high delay that are based on real incident reports from Google Cloud En-

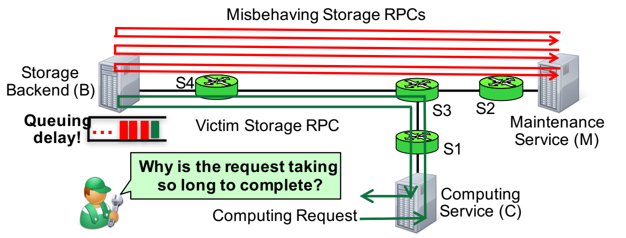


Figure 1: Scenario: The misconfigured maintenance service is overloading the storage backend and is causing requests from the computing service to be delayed.

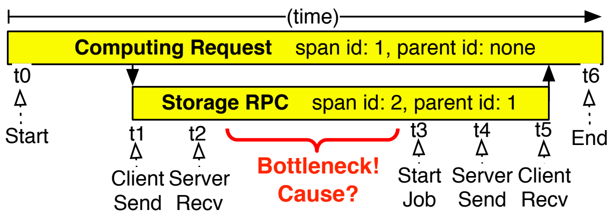


Figure 2: A trace tree for the slow computing requests in Figure 1. B received the storage RPC at t_2 but only started processing it at t_3, after a long queuing delay.

gine. Our evaluation shows that, in each case, the resulting temporal provenance clearly identifies the cause of the delay. We also show that the runtime overhead is comparable to that of existing tools, such as Zipkin [1], which is based on Google Dapper [55]. In summary, our contributions are:

- The concept of temporal provenance (Section 2);
- an algorithm that generates temporal provenance (Section 4);
- a post-processing technique that improves the readability of timing provenance graphs (Section 5);
- Zeno, a prototype debugger that records and displays temporal provenance (Section 6); and
- an experimental evaluation of Zeno (Section 7).

In the following two sections, we begin with an overview of timing diagnostics and its key challenges.

2 Overview

Figure 1 illustrates the example scenario we have already sketched above. In this scenario, an operator manages a small network that connects a maintenance service M, a computing service C, and a storage backend B. Both M and C communicate with the backend using RPCs. A job on M is misconfigured and is sending an excessive number of RPCs (red) to the storage backend. This is causing queuing at the backend, which is delaying RPCs from the computing service (green). The operator notices the delays on C, but is unaware of the misconfiguration on M.

We refer to this situation as a *timing fault*: the RPCs from C are being handled correctly, but not as quickly as the operator expects. A particularly challenging aspect of this scenario is that the cause of the delays that C's requests are experiencing (the misconfiguration on M) is not on the path from C to B; we call this an *off-path cause*.

Timing faults are quite common in practice. To illustrate this, we surveyed incidents disclosed by Google Cloud Platform [2], which occur across a variety of different cloud services and directly impact cloud tenants. To obtain a good sample size, we examined all incidents that happened from January 2014 until May 2016, and we selected all 95 incident reports that describe both the symptoms and the causes. We found that more than a third (34.7%) of these incidents were timing faults.

2.1 Prior work: Trace trees

Today, a common way to diagnose such a situation is to track the execution of the request and to identify the bottleneck – that is, components that are contributing unusual delays. For instance, a distributed tracing system would produce a "trace tree" [55]. Figure 2 shows an example tree for one of the delayed responses from the computing service C in Figure 1. The yellow bars represent basic units of work, which are usually referred to as *spans*, and the up/down arrows indicate causal relationships between a span and its parent span. A span is also associated with a simple log of timestamped records that encode events within the span.

Trace trees are helpful because they show the steps that were involved in executing the request: the computation was started at t_0 and issued an RPC to the storage backend at t_1; the backend received the RPC at t_2, started processing it at t_3, and sent a response at t_4, which the client received at t_5; finally, the computation ended at t_6. This data structure helps the operator to find abnormal delays: for instance, the operator will notice that the RPC waited unusually long ($t_2 \dots t_3$) before it was processed by B.

However, the operator also must understand what *caused* the unusual delay, and trace trees offer far less help with this step. In our scenario, the cause – the misbehaving maintenance service – never even appears in any span! The reason is that *trace trees include only the spans that are on the execution path* of the request that is being examined. In practice, off-path causes are very common: when we further investigated the 33 timing faults in our survey from above, we found that, in over 60% of the cases, the real problem was not on the execution path of the original request, so it would not have appeared in the corresponding trace tree.

2.2 Prior work: Provenance

Another approach that has been explored recently [61, 30, 60, 68, 69, 67] is to use *provenance* [26] as a diagnostic tool. Provenance is a way to obtain causal explanations of

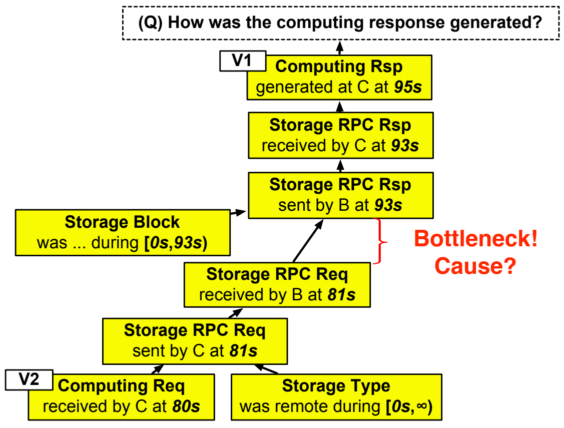

(Q) How was the computing response generated?
V1
Computing Rsp generated at C at 95s
Storage RPC Rsp received by C at 93s
Storage RPC Rsp sent by B at 93s
Storage Block was ... during [0s,93s)
Bottleneck! Cause?
Storage RPC Req received by B at 81s
Storage RPC Req sent by C at 81s
V2
Computing Req received by C at 80s
Storage Type was remote during [0s,∞)

Figure 3: Time-aware provenance, as in DTaP [68], for the example scenario from Figure 1.

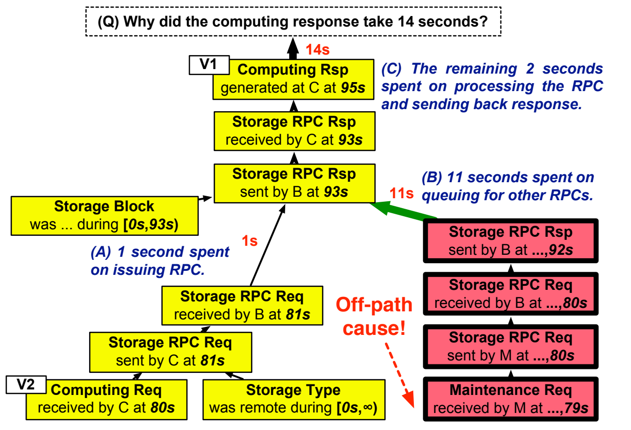

(Q) Why did the computing response take 14 seconds?
14s
V1
Computing Rsp generated at C at 95s
(C) The remaining 2 seconds spent on processing the RPC and sending back response.
Storage RPC Rsp received by C at 93s
Storage RPC Rsp sent by B at 93s
(B) 11 seconds spent on queuing for other RPCs.
11s
Storage Block was ... during [0s,93s)
(A) 1 second spent on issuing RPC.
1s
Storage RPC Rsp sent by B at ...,92s
Storage RPC Req received by B at ...,80s
Storage RPC Req sent by M at ...,80s
Maintenance Req received by M at ...,79s
Off-path cause!
Storage RPC Req received by B at 81s
Storage RPC Req sent by C at 81s
V2
Computing Req received by C at 80s
Storage Type was remote during [0s,∞)

Figure 4: Temporal provenance, as proposed in this paper, for the example scenario from Figure 1.

an event; a provenance system maintains, for each (recent) event in the network, a bit of metadata that keeps track of the event's direct causes. Thus, when the operator requests an explanation for some event of interest (say, the arrival of a packet), the system can produce a recursive explanation that links the event to a set of causes (such as the original transmission of the packet and the relevant routing state). Such a representation is useful because the diagnostician often finds herself swimming in a sea of possibilities: at any given moment, there are millions of events happening in the data center, and many of them *could* hypothetically be related to the observed symptom. Moreover, a substantial fraction of these events tend to be unusual in some way or another, which is why the use of anomaly detection often yields many false positives. In contrast, provenance is a way to quickly and reliably identify the (few) events that actually *were* causally related, which can be an enormous time saver.

Provenance can be represented as a DAG, whose vertices represent events and whose edges represent direct causality. Figure 3 shows the DAG that a provenance system like DTaP [68] would generate for our example scenario. (We picked DTaP because it proposed a "time-aware" variant of provenance, which already considers a notion of time.) This data structure is considerably more detailed than a trace tree; for instance, it not only shows the path from the original request (V2) to the final response (V1), but also the data and the configuration state that were involved in processing the request along the way. However, the actual cause from the scenario (the misconfigured maintenance service) is *still* absent from the data structure. The reason is that DTaP's provenance is "time-aware" only in the sense that it can remember the provenance of past system states. It does annotate each event with a timestamp, as shown in the figure, but it does not reason about temporal causality. Thus, it actually does not offer very much extra help compared to trace trees: like the latter, it can be used to *find* bottlenecks, such as the high response times in the backend, but it is not able to *explain* them by identifying causally related events, such as the RPCs from the maintenance service. Tracking such dependencies between functional and temporal behavior, potentially across several components, is the problem we focus on.

2.3 Our approach

We propose to solve this problem with a combination of three insights. The first is that temporal causality critically depends on a type of information that existing tracing techniques tend not to capture: the *sequence* in which the system has processed requests, *whether the requests are related or not*. By looking only at functional dependencies, these techniques simply consider each request in isolation, and thus cannot make the connection between the slow storage RPC and the requests from the maintenance service that are delaying it. With provenance, we can fix this by including a second kind of edge $e_1 \rightarrow e_2$ that connects each event e_1 to the event e_2 that was processed on the same node and immediately after e_1. We refer to such an edge as a *sequencing edge* (Section 4.1). Notice that these edges essentially capture the well-known happens-before relation [45].

Our next insight is a connection between temporal reasoning and the critical path analysis from scheduling theory. When scheduling a set of parallel tasks with dependencies, the critical path is the longest dependency chain, and it determines the overall completion time. This concept is not directly applicable to off-path causes, but we have found a way to generalize it (Section 4.3). The result is a method that recursively allocates delay to the branches of a provenance tree, which yields a data structure that we call *temporal provenance*.

Our third insight has to do with readability. At first glance, temporal provenance is considerably richer than classical provenance because it considers not only functionally related events, but also events that could have contributed only delay (of which there can be many). However, in practice, many of these events do not actually contribute to the end-to-end delay, and the ones that do are often structurally similar – such as the maintenance requests in our example scenario – and can be aggregated. Thus, it is usually possible to extract a compact representation that can be easily understood by the human operator (Section 5).

Figure 4 shows the temporal provenance for a random computing request in our example scenario. Starting at the root, the provenance forks into two branches; the left branch (A) shows that one second was spent on issuing the RPC itself; and the right branch (B) shows that the majority of the delay (11 seconds) was caused by RPCs from the maintenance service (M). This tree has all the properties we motivated earlier: it provides a quantitative explanation of the delay, and it includes the actual cause (the maintenance service), even though it is an off-path cause and does not appear on the path the request has taken.

3 Background

Since temporal provenance is a generalization of network provenance, we begin with a brief description of the latter, and refer interested readers to [67] for more detail.

3.1 Network Datalog

For ease of exposition, we will present our approach in the context of *network datalog (NDlog)* [47]. The approach itself is not specific to either NDlog or to declarative languages; provenance has been applied to imperative systems that were written in a variety of languages [49, 22, 38, 35, 39, 60], and our own evaluation in Section 7 includes experiments with Google Dapper [55]. However, NDlog's declarative syntax makes the provenance very easy to see.

In NDlog, the state of a node is modeled as *tables*, each of which contains a number of *tuples*. For example, an RPC server might have a table called `RPC` that contains the RPC calls it has received from clients. Tuples can be manually inserted, or they can be programmatically derived from other tuples. The former represent external inputs and are called *base tuples*, whereas the latter represent computations in the system itself and are called *derived tuples*.

NDlog programs consist of *rules* that describe how tuples should be derived from one another. For instance, the rule `A(@L,P):-B(@L,Q),Q=3*P` says that a tuple `A(@L,P)` should be derived on node `L` whenever a `B(@L,Q)` tuple is also on that node, and `Q=3*P`. Here, P and Q are variables that are instantiated with values when the rule is applied; for instance, a `B(@L,6)` tuple would create an `A(@L,2)` tuple. The `@` operator specifies the location of the tuple.

Note that, in this declarative formulation, the direct causes of a tuple's existence are simply the preconditions of the rule that was used to derive it. For instance, if `A(@L,2)` was derived using the rule above, then the direct causes were the existence of `B(@L,6)` and the fact that `6=3*2`.

3.2 System model

If our goal was classical data provenance, the declarative description above would already be sufficient. However, since we are particularly interested in *timing*, we need to consider some more details of how the system works. For concreteness, we use an event-driven model: the system reacts to events such as packet arrivals and configuration changes; each node has a queue of events that it processes in a FIFO order; and each event can trigger one or more additional events, either locally or on another node. (Note that the "nodes" here do not necessarily have to correspond to physical machines; they could be different CPU cores, or line cards in a switch.) This model captures how pipelined semi-naïve evaluation [47] works in NDlog: the events are tuple insertions and deletions, and the processing corresponds to tuple derivations. However, more importantly, it is also a good description of networks and services with FIFO queues.

3.3 Classical provenance

In order to be able to answer provenance queries, a system must collect some additional metadata at runtime. Conceptually, this can be done by maintaining a large DAG, the *provenance graph*, that contains a vertex for every event that has occurred in the system, and in which there is an edge (a,b) between two vertices if event a was a direct cause of event b. (A practical implementation would typically not maintain this graph explicitly, but instead collect only enough information to reconstruct a recent subgraph when necessary; however, we will use this concept for now because it is easier to explain.) If the system later receives a provenance query QUERY(e) for some event e, it can find the answer by locating the vertex that corresponds to e and then projecting out the subgraph that is rooted at e. This subgraph will be the *provenance* of e.

For concreteness, we will use a provenance graph with six types of vertices, which is loosely based on [68]:

- INS($[t_s,t_e],N,\tau$), DEL($[t_s,t_e],N,\tau$): Base tuple τ was inserted (deleted) on node N during $[t_s,t_e]$;
- DRV($[t_s,t_e],N,\tau$), UDRV($[t_s,t_e],N,\tau$): Derived tuple τ acquired (lost) support on N during $[t_s,t_e]$;
- SND($[t_s,t_e],N\rightarrow N', \pm\tau$), RCV($[t_s,t_e],N\leftarrow N', \pm\tau$): $\pm\tau$ was sent (received) by N to (from) N' during $[t_s,t_e]$.

Note that each vertex is annotated with the node on which it occurred, as well as with a time interval that indicates when the node processed that event. For instance, when a switch makes a forwarding decision for a packet, it derives a new tuple that specifies the next hop, and the time $[t_s,t_e]$ that was spent on this decision is indicated in the corresponding DRV vertex. This will be useful (but not yet sufficient) for temporal provenance later on.

The edges between the vertices represent their causal relationships. A SND vertex has an edge from an INS or a DRV that produced the tuple that is being sent; a RCV has an edge from the SND vertex for the received message; and a DRV vertex for a rule `A:-B,C,D` has an edge from each precondition (B, C, and D) that leads to the vertex that produced the corresponding tuple. An INS vertex corresponds to an event that cannot be explained further (the insertion of a base tuple); thus, it has no incoming edges. The edges for the negative "twins" of these vertices – UDRV and DEL – are analogous.

The above definition has two useful properties. First, is recursive: the provenance of an event e includes, as subgraphs, the provenances of all the events that contributed to e. This is useful to an operator because she can start at the root and "drill down" into the explanation until she identifies a root cause. Second, there is a single data structure – the provenance graph – that can be maintained at runtime, without knowing a priori what kinds of queries will be asked later.

4 Temporal provenance

In this section, we generalize the basic provenance model from Section 3 to reason about the timing of events.

4.1 Sequencing edges

The provenance model we have introduced so far would produce provenance that looks like the tree in Figure 3: it would explain why the event at the top occurred, but it would not explain why the event occurred *at that particular time*. The fact that the vertices are annotated with timestamps, as in prior work [68], does not change this: the operator would be able to see, for instance, that the storage service took a long time to respond to a request, but the underlying *reason* (that requests from another node were queued in front of it) is not shown; in fact, it does not even appear in the graph!

To rectify this, we need to capture some additional information – namely, the *sequence* in which events were processed by a given node. Thus, we introduce a second type of edge that we call *sequencing edge*. A sequencing edge (v_1,v_2) exists between two vertices a and b iff either a) the corresponding events happened on the same node, and a was the event that immediately preceded b, or b) a is an SND vertex and b is the corresponding RCV vertex. We refer to the first type of edge as a *local* sequencing edge, and to the second type as a *remote* sequencing edge. In the illustrations, we will render the sequencing edges with green, dotted lines to distinguish them from the causal edges that are already part of classical provenance.

Although causal edges and sequencing edges often coincide, they are in fact orthogonal. For instance, consider the scenario in Figure 5(a). Here, a node X has two rules, `B:-A` and `C:-A`; in the concrete execution (shown on the timeline), A is inserted at time 0, which triggers both rules, but B is derived first, and then C. In the provenance graph (shown at the bottom), INS(A) is connected to DRV(B) by both a causal and a sequencing edge, since the two events happened back-to-back and B's derivation was directly caused by A's insertion. But DRV(B) is connected to DRV(C) only by a sequencing edge, since the former did precede the latter but was not a direct cause; in contrast, INS(A) is connected to DRV(C) only by a causal edge, since A's insertion did cause C's derivation, but the latter was directly delayed by another event.

4.2 Queries

Next, we turn to the question what a query for temporal provenance should look like, and what it should return. Unlike a classical provenance query QUERY(e), which aims to explain a specific event e, a temporal provenance query aims to explain a *delay* between *a pair of* events e_1 and e_2. For instance, in the scenario from Figure 1, the operator wanted to know why his request had taken so long to complete, which is, in essence, a question about the delay between the request itself (e_1) and the resulting response (e_2). Hence, we aim to answer queries of the form T-QUERY(e_1,e_2), which ask about delay between two events e_1 and e_2. Our only requirement is that the events are causally related – i.e., that there is a causal path from e_1 to e_2.

As a first approximation, we can answer T-QUERY(e_1,e_2) as follows. We first query the classical provenance $P :=$ QUERY(e_2). Since we require that e_1 and e_2 are causally related, P will include a vertex for e_1. We then identify all pairs of vertices (v_1,v_2) in P that are connected by a causal edge but not by a sequencing edge. We note that, a) in each such pair, v_2 must have been delayed by some other intervening event, and b) v_1 is nevertheless connected to v_2 via a multi-hop path along the sequencing edges. (The reason is simply that v_1 was one of v_2's causes and must therefore have happened before it.) Thus, we can augment the causal provenance by adding these sequencing paths, as well as the provenance of any events along such a path. The resulting provenance P' contains all the events that have somehow contributed to the delay between e_1 and e_2. We can then return P' as the answer to T-QUERY(e_1,e_2).

4.3 Delay annotations

As defined so far, the temporal provenance still lacks a way for the operator to tell *how much* each subtree has contributed to the overall delay. This is important for usability: the operator should have a way to "drill down" into the graph to look for the most important causes of delay. To facilitate this, we additionally annotate the vertices with the delay that they (and the subtrees below them) have contributed.

Computing these annotations is surprisingly nontrivial and involves some interesting design decisions. Our algorithm is shown in Figure 6; we explain it below in several re-

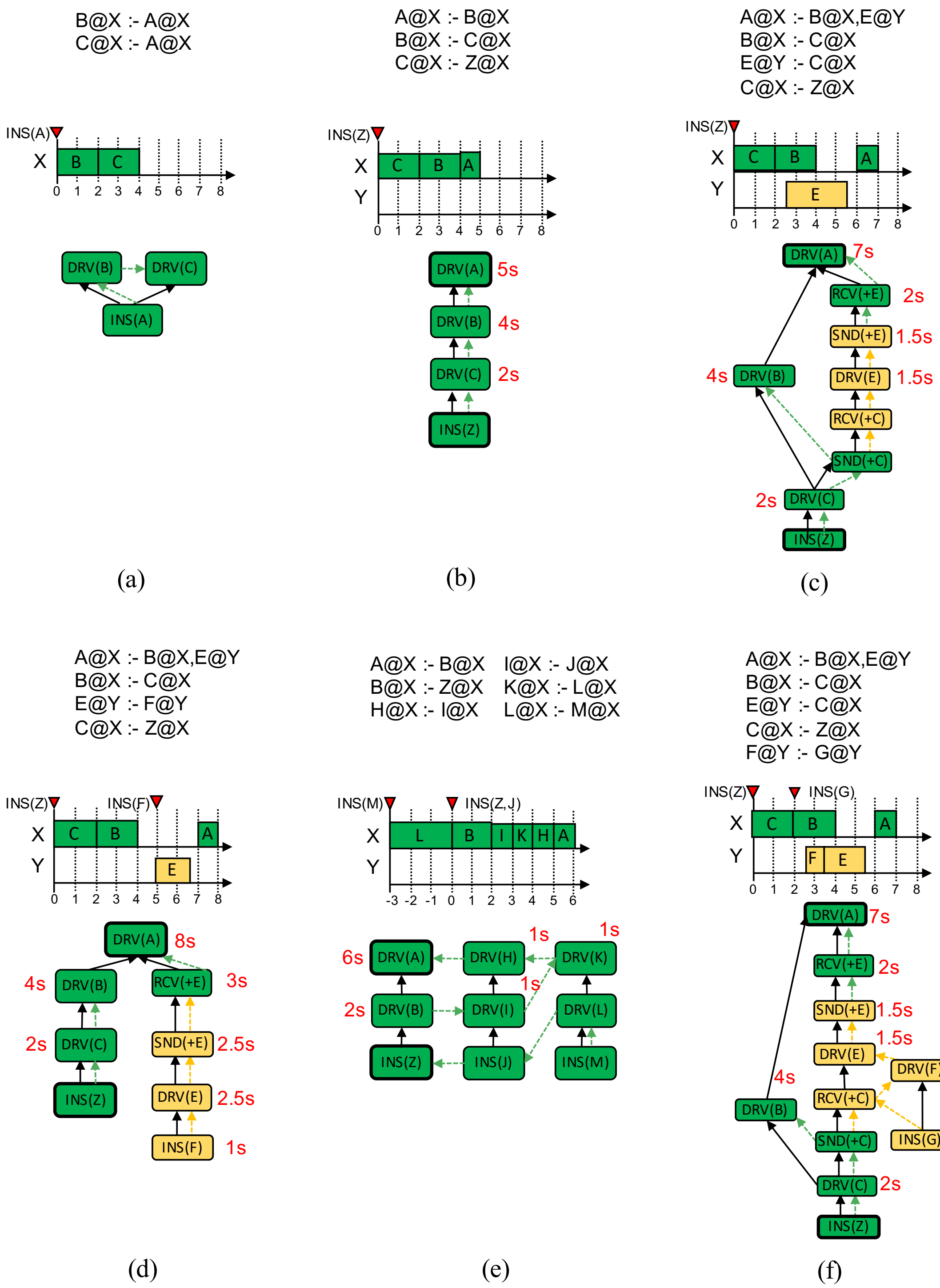


Figure 5: Example scenarios, with NDlog rules at the top, the timing of a concrete execution in the middle, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all scenarios; the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity.

```
 1: // the subtree rooted at $v$ is responsible for the delay during $[t_s, t_e]$
 2: function ANNOTATE($v$, $[t_s, t_e]$)
 3:     ASSERT($t_e == t_{end}(v)$)
 4:     if $[t_s, t_e] = \emptyset$ then
 5:         RETURN
 6:     // weight $v$ by the amount of delay it contributes
 7:     SET-WEIGHT($v$, $t_e - t_s$)
 8:     // recursive calls for functional children in order of appearance
 9:     C ← FUNCTIONAL-CHILDREN(v)
10:     T ← $t_s$
11:     while $C \neq \emptyset$ do
12:         $v' \leftarrow c \in C$ WITH MIN $t_{end}(c)$
13:         C ← $C \setminus \{v'\}$
14:         if $t_{end}(v') \geq T$ then
15:             ANNOTATE($v'$, $[T, t_{end}(v')]$)
16:             $T \leftarrow t_{end}(v')$
17:     // recursive calls for sequencing children
18:     s ← SEQUENCING-CHILD(v)
19:     E ← $t_{start}(v)$
20:     while $T < E$ do
21:         ANNOTATE(s, $[\text{MAX}(T, t_{start}(s)), E]$)
22:         $E \leftarrow t_{start}(s)$
23:         $s$ ← SEQUENCING-CHILD(s)
```

Figure 6: Algorithm for computing delay annotations (explained in Sections 4.3–4.5).

finements, using the simple examples in Figures 5(b)–(f). The examples are shown in the same format as in Figure 5(a): each shows a set of simple NDlog rules, the timing of events during the actual execution, and the resulting temporal provenance, with the delay annotations in red. The query is always the same: T-QUERY(INS(Z), DRV(A)); that is, we want to explain the delay between the insertion of Z and the derivation of A. One difference to Figure 5(a) is that some of the examples require two nodes, X and Y. To make the connections more visible, we show the vertices that belong to Y in orange, and the ones that belong to X in green, as in Figure 5(a). If a vertex did not contribute to the delay, we omit its annotation.

Our algorithm computes the delay annotations recursively. A function ANNOTATE is called on the root of the provenance; the function then invokes itself on (some of) the children to compute the annotations on the subgraphs. As a first approximation, this works as follows:

Rule #1: Annotate the top vertex v with the overall delay T, then subtract the execution time t_v of v, and repeat with v's child vertex, using delay $T - t_v$.

In our algorithm, this corresponds to line 7, which sets the weight for the current vertex, and the recursive call in line 15; lines 4–5 contain the base case, where the delay is zero.

4.4 Handling multiple preconditions

This approach works well for linear provenance, such as the one in Figure 5(b): deriving A from Z took 5s because it took 1s to compute A itself, and 4s to derive A's precondition, B; deriving B from Z took 4s because 2s were spent on B itself and another 2s on C. However, it does *not* work well for rules with multiple preconditions. Consider the scenario in Figure 5(c): A now has two preconditions, B and E, so the question is how much of the overall delay should be attributed to each.

Two answers immediately suggest themselves: 1) since B completed after 4s, we can attribute 4s to B and the remaining 2s to E, which finished later, or 2) we can attribute the entire 6s to E, because it was the last to finish. The latter is somewhat similar to the choice made in critical path analysis [34, 62]; however, the theorems in Section 4.6 actually require the former: if we find a way to speed up E (or cause F to be inserted sooner), this can only reduce the end-to-end delay by 3s. Any further reductions would have to come from speeding up B. This leads to the following refinement:

Refinement #2: Like rule #1, except that the remaining delay is allocated among the preconditions in the order in which they were satisfied.

This refinement is implemented in lines 11–13 and 16 of our algorithm, which iterate through the preconditions in the order of their appearance (that is, local derivation or arrival from another node) and allocate to each the interval between its own appearance and the appearance of its predecessor.

Notice that this approach deviates from critical-path analysis in an interesting way. Consider the scenario in Figure 5(d): here, the provenance has two "branches", one connected to the insertion of Z and the other to the insertion of F, but there is no causal path from Z to F. (We call such a branch an *off-path branch*.) This raises the question whether any delay should be apportioned to off-path branches, and if so, how much. Critical path analysis has no guidance to offer for this case because it only considers tasks that are transitively connected to the start task.

At first glance, it may seem that F's branch should not get any delay at all; for instance, F could be a configuration entry that is causally unrelated to Z and thus did not obviously contribute to a delay from Z to A. However, notice that all the "on-path" derivations (in Z's branch) finished at $t = 4s$, but A's derivation was nevertheless delayed until $t = 7s$ because E was not yet available. Thus, it seems appropriate that the other branch gets the remaining 3s.

4.5 Handling sequencing delays

The one question that remains is what to do if there is further delay after the last precondition is satisfied. This occurs in the scenario in Figure 5(e): although B is derived immediately after Z is inserted at $t = 0$, A's derivation is delayed by another 3s due to some causally unrelated derivations (I, K, and H). Here, the sequencing edges come into play: we can attribute the remaining delay to the predecessor along the *local* sequencing edge (here, DRV(H), which will subtract its own computation time and pass on any remaining delay to its own predecessor, etc. This brings us to the final rule:

Final rule: Like #2, except that, if any delay remains after the last precondition, that delay is attributed to the predecessors along the local sequencing edge.

This is implemented in lines 17–23.

So far, we have focused on the rule for DRV vertices, which is the most complex one. SND vertices are easy because they only have one (causal) child; RCV vertices are even easier because they cannot be delayed by sequencing; and INS vertices are trivial – they have no causal children.

4.6 Correctness

We have formally modeled the properties of temporal provenance, and we have proven that our algorithm achieves them. Due to lack of space, we cannot include the full model or the proofs here (they are available in Appendix A); however, we informally describe the properties below.

The properties we considered fall into two categories. The first category consists of basic properties that provenance is generally expected to have; for instance, the provenance should describe a correct execution of the system (validity), it should respect happens-before relationships (soundness), it should be self-contained and fully explain the relevant event (completeness), and it should only contain events that are actually necessary for the explanation (minimality). We have formalized these properties using existing definitions from [68]. Since these definitions are for provenance in general and do not consider the temporal aspect at all, our proofs basically indicate that we did not "break" anything.

The second category contains the properties of the delay annotations that our algorithm creates. Since this is a key contribution of this paper, we briefly sketch our approach. We carefully define what it means for a derivation $\tau : -c_1, c_2, \ldots$ to be directly "delayed" by one of its preconditions, and we then recursively extend this definition to transitive delays (that is, one of the c_i was itself delayed by one of its own preconditions, etc.). Our first theorem (Section A.5) states that each vertex is labeled with the amount of (direct or transitive) delay that is contributed by the subtree that is rooted at that vertex. Our second theorem (Section A.6) essentially says that, if there is a vertex v in a temporal provenance tree that is annotated with T and the sum of the annotations on its children and immediate predecessors is $S < T$, then it is possible to construct another valid (but hypothetical) execution in which v's execution time is reduced by $(T - S)$ and in which the derivation finishes $(T - S)$ units of time earlier. This shows that the annotations really do correspond to the "potential for speedup" that we intuitively associate with the concept of delay.

4.7 Limitations

Temporal provenance is not magic: when the real reasons for a delay are complex – e.g., many small but unrelated factors that simply add up – the temporal provenance will likewise by complex and will not show a single root cause. Even in cases where there really is a single unusual factor that causes a lot of delay, temporal provenance does not always single it out, since it has no notion of what is unusual, or which delays are avoidable; instead, it will simply identify *all* causes, annotate each with the delay it caused, and leave the decision to the operator. (However, it could be combined with an additional root-cause analysis, e.g., the one from [30].) Finally, unlike functional provenance, temporal provenance might experience a "Heisenberg effect" in certain cases – that is, collecting the necessary information could subtly alter the timing of the system and prevent the very bugs from appearing that the operator wishes to diagnose (or trigger new, different ones).

5 Improving readability

As defined above, temporal provenance is already useful for diagnostics because it can explain the reasons for a delay between two events. However, the provenance may not be as succinct as a human operator would prefer due to two reasons. First, the temporal provenance for $[e_1, e_2]$ contains the entire classical provenance of e_2 as a subgraph, even though some of the functional causes did not actually contribute to the delay. Second, sequencing delay is often the result of many similar events that each contribute a relatively small amount of delay. To declutter the graph, we perform two post-processing steps.

5.1 Pruning zero-delay subgraphs

Our first post-processing step hides any vertices that are annotated with zero (or not annotated at all) by the ANNOTATE function. The only exception is that we keep vertices that are connected via a causal path (i.e., a path with only causal edges) to a vertex that is annotated with a positive delay. For instance, in Figure 5, the original INS(Z) vertex – the starting point of the interval – would be preserved, even though the insertion itself did not contribute any delay.

To illustrate the effect of this step we consider the example in Figure 5(f), which is almost identical to the one in Figure 5(c), except that an additional, unrelated derivation (F) occurred before the derivation of E. Here, the INS(G) and the DRV(F) would be hidden because they do not contribute to the overall delay.

5.2 Provenance aggregation

Our second post-processing step aggregates structurally similar subgraphs [54]. This helps with cases where there are many events that each contribute only a very small amount of delay. For instance, in our scenario from Figure 1, the delay is caused by a large number of RPCs from the maintenance service that are queued in front of the RPC. The "raw" temporal provenance contains a subtree for each such RPC. During post-processing, these nearly identical subtrees would be aggregated into a single subtree whose weight is the sum of the weights of the individual trees, as shown in Figure 4.

There are two key challenges with this approach. The first is to decompose the temporal provenance into smaller subgraphs that can potentially be aggregated. At first glance, there are exponentially many decompositions, so the problem seems intractable. However, we observe that (1) aggregation is most likely to be possible for sequencing delays, which are often due to similar kinds of events (network packets, RPCs) that have a similar provenance; and that (2) the corresponding subtrees can easily be identified because they are laterally connected to the functional provenance through a chain of sequencing edges. Thus, we can extract candidates simply by following such lateral sequencing edges and by taking the subgraphs below any vertices we encounter.

The second challenge is deciding whether two subgraphs can be aggregated. As a first approximation, this is a graph isomorphism problem, and since our provenance graphs have a bounded chromatic index (which roughly corresponds to the number of preconditions in the largest rule), the classic algorithms – e.g., [23] – should work well. However, in our case the candidate subgraphs are often similar but rarely identical; thus, we need to define an equivalence relation that controls which vertices and are safe to aggregate. We use a simple heuristic that considers two vertices to be similar if they share a tuple name and have been derived on the same node. To aggregate two subgraphs, we start at their root vertices; if the vertices are similar, we merge them, annotate them with the sum of their individual annotations, and recursively attempt to merge pairs of their children. If the vertices are not similar, we stop aggregation at that point and connect the two remaining subgraphs directly to the last vertex that was successfully aggregated.

The aggregation procedure is commutative and associative; thus, rather than attempting aggregation for all pairs of subgraphs, we can simply try to aggregate each new subgraph with all existing aggregates. In our experience, the $O(N^2)$ complexity is not a problem in practice because N is often relatively small and/or most of the subgraphs are similar, so there are very few aggregates.

6 The Zeno debugger

We have built a temporal provenance debugger called Zeno with five components in $23,603$ lines of code.

Runtime: To demonstrate that Zeno can work with different languages and platforms, we built three different front-ends. The first is integrated with RapidNet [3] and enables Zeno to generate temporal provenance for NDlog programs. The second is integrated with the Zipkin [1] framework – a cross-language distributed tracing library that is based on Google Dapper [55] and can run a network of microservices written in Node.js [4] (JavaScript), Pyramid [5] (Python), and WEBrick [6] (Ruby). The third is integrated with Mininet [7], which we use to emulate a network environment with P4 switches [17]. All front-ends share the same back-end for reasoning about temporal provenance. In our evaluation, we use the first and the third front-end for SDN applications, and the second one for native Zipkin services.

Provenance recorder: At runtime, our debugger must record enough metadata to be able to answer provenance queries later on. Previous work [69, 46, 61] has already shown that provenance can be captured at scale; this is typically done either (1) by explicitly recording all events and their direct causes, or (2) by recording only nondeterministic inputs at runtime, and by later replaying the execution with additional instrumentation to extract events and causes if and when a query is actually asked [68]. The Zipkin front-end follows the first approach, because Zipkin already has well-defined interfaces to capture both base events and intermediate events (such as RPC invocations and completions), which yields a complete trace tree for each request. Therefore, Zeno merely adds a post-processing engine that converts the trace trees to functional provenance and that infers most of the sequencing edges from the recorded sequence of events across all trace trees. In addition, Zeno extends the Zipkin runtime with dtrace [27] to capture sequencing edges that cannot be directly inferred (e.g., edges representing lock contention). The NDlog front-end uses the second approach and is based on an existing record+replay engine from [68]. The Mininet platform leverages P4 switches to directly obtain ingress/egress timestamps. (More on this below.) In both approaches, we record timestamps at microsecond-level precision, which should be sufficient in practice [57].

Query processor: The third and final component accepts queries T-QUERY(e_1,e_2) from the operator, as defined in Section 4.2, and it answers each query by first generating the raw temporal provenance and then applying the post-processing steps. The resulting graph is then displayed to the operator.

Retention policy: To prevent the storage requirements from growing indefinitely, our prototype maintains the provenance data only for a configurable retention time R, and prunes it afterwards. Because of this, the result of a T-QUERY(e_1,e_2) can be incomplete: for instance, if a particular forwarding decision was made based on a routing table entry that is older than R, the corresponding branch of the provenance tree will be "pruned" at that point, since the entry's provenance will already have been discarded. However, if e_1 is no older than R, all vertices that would be annotated with a nonzero delay *will* be included, so, if most queries are about recent events, this is not a big sacrifice to make. If desired, the retention policy could easily be refined or replaced.

P4 integration: Obtaining sequencing edges is not always straightforward, especially at the network switches. Fortunately, we can leverage the in-band network telemetry (INT) capability [16] in emerging switches [24] to obtain sequencing edges. These switches can stamp into a packet's header the ingress/egress timestamps at each queue, which can then be used to obtain sequencing edges. If two packets p_1 and p_2 traverse the same queue and their ingress/egress timestamps were t_{i1}/t_{e1} and t_{i2}/t_{e2}, with $t_{i1} < t_{i2} < t_{e1} < t_{e2}$, then we know that p_2 must have been queued after p_1, and Zeno can add a sequencing edge to the provenance graph. We have implemented an extension in our prototype to approximate this capability; note, however, that modern switches can perform these operations directly in hardware at line speed.

7 Evaluation

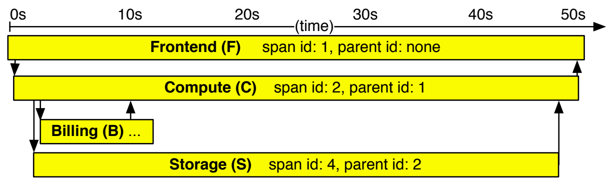
In this section, we report results from an experimental evaluation of our debugger. We have designed our experiments to answer four high-level questions: 1) Is temporal provenance useful for debugging realistic timing faults? 2) What is the cost for maintaining temporal provenance? 3) How fast can our debugger process diagnostic queries? And 4) Does temporal provenance scale well with the network size?

We ran our experiments on a Dell OptiPlex 9020 workstation, which has a 8-core 3.40 GHz Intel i7-4770 CPU with 16 GB of RAM. The OS was Ubuntu 16.04, and the kernel version was 4.10.0. Parts of the Zipkin front-end ran on a MacBook Pro, which has a 4-core 2.40 GHz Intel i5-4258 CPU with 8GB of RAM. The OS was macOS 10.13.2, and the kernel version was 17.3.0.

7.1 Diagnostic scenarios

We reproduced seven representative scenarios that we sampled from incidents reported by Google Cloud Engine [2]:

- **R1, Z1: Misbehaving maintenance task [8].** Clients of the Compute Engine API experienced delays of up to two minutes because a scheduled maintenance task caused queuing within the compute service. This is the scenario from Section 2.
- **R2, Z2: Elevated API latency [9].** A failure caused the URL Fetch API infrastructure to migrate to a remote site. This increased the latency, which in turn caused clients to retry, worsening the latency. Latencies remained high for more than 3 hours.
- **R3: Slow deployments after release [10].** A new release of App Engine caused the underlying pub/sub infrastructure to send an update to each existing instance. This additional load slowed down the delivery of deployment messages; thus, the creation of new instances remained slow for more than an hour.
- **R4: Network traffic changes [11].** Rapid changes in external traffic patterns caused the networking infrastructure to reconfigure itself repeatedly, which created a substantial queue of modifications. Since the network registration of new instances had to wait on events in this queue, the deployment of new instances was slowed down for 90 minutes.
- **Z3: Lock contention [12].** User-submitted jobs experiences increased execution time for over 13 hours because lock contention in an underlying component slowed down query scheduling and execution.
- **Z4: Slow load jobs [13].** Load jobs to an analytics service experienced long latencies for 29 hours. The service received an elevated number of jobs that exceeded its ingestion capacity and caused new jobs to wait increasingly longer to be scheduled.
- **M1: Network congestion [14].** Two cloud services experienced high latency for over 6 hours due to network congestion.



0s
10s
20s
(time)
30s
40s
50s
Frontend (F) span id: 1, parent id: none
Compute (C) span id: 2, parent id: 1
Billing (B) ...
Storage (S) span id: 4, parent id: 2

Figure 7: Zipkin trace tree for scenario Z1, which shows that the RPC to the storage service is the bottleneck, but the actual cause (the RPCs from the maintenance service) is off-path and thus is absent.

We reproduced four scenarios in RapidNet (R1–R4) and four in the microservice environment (Z1–Z4), including two scenarios in both environments. The microservice scenarios used five to eight servers. (We do not model switches in the microservice scenarios.) We used single-core nodes for Z1 and Z2, but we used up to four cores for Z3 and Z4, to test Zeno's ability to handle concurrency; in this case, we spread the workload equally across the available cores. The first two RapidNet scenarios use four switches, one controller, and three servers; for the remaining RapidNet scenarios, we used four switches and one controller but a larger number of servers (115 for R3, and 95 for R4). We reproduced the final scenario in Mininet (M1) with 20 P4 switches with 16 hosts organized in a three-tiered Fat-tree topology, where the sequencing edges were obtained using the ingress/egress timestamps exported by the P4 switches [17].

7.2 Identifying off-path causes

A key motivator for this work is the fact that off-path causes for a delay are often not even visible with an existing debugger. To test whether Zeno addresses this, we generated trace trees (using Zipkin) and classic provenance (using DTaP), and compared their ability to identify off-path causes.

Figure 7 shows a Zipkin trace tree for Z1. A human operator can clearly see that the API call to the frontend took 50 seconds, and that the compute RPC and the storage RPC

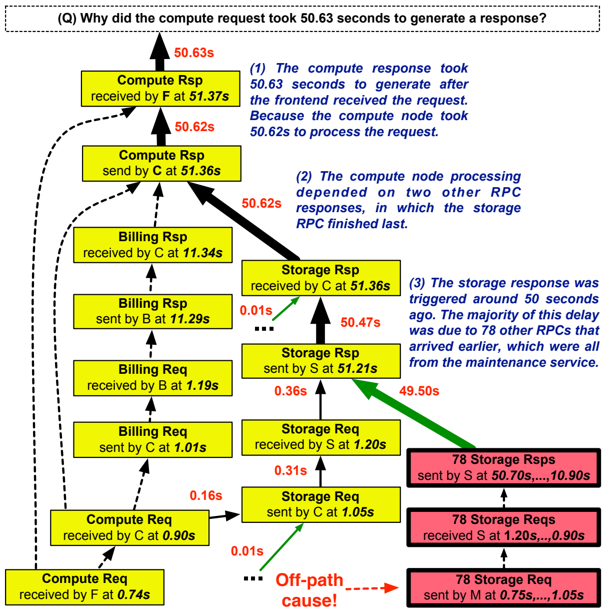

(Q) Why did the compute request took 50.63 seconds to generate a response?
50.63s
Compute Rsp received by F at 51.37s
(1) The compute response took 50.63 seconds to generate after the frontend received the request. Because the compute node took 50.62s to process the request.
50.62s
Compute Rsp send by C at 51.36s
(2) The compute node processing depended on two other RPC responses, in which the storage RPC finished last.
50.62s
Billing Rsp received by C at 11.34s
Storage Rsp received by C at 51.36s
(3) The storage response was triggered around 50 seconds ago. The majority of this delay was due to 78 other RPCs that arrived earlier, which were all from the maintenance service.
Billing Rsp sent by B at 11.29s
0.01s
50.47s
Storage Rsp sent by S at 51.21s
Billing Req received by B at 1.19s
0.36s
49.50s
Billing Req sent by C at 1.01s
Storage Req received by S at 1.20s
78 Storage Rsps sent by S at 50.70s,...,10.90s
0.31s
0.16s
Storage Req sent by C at 1.05s
Compute Req received by C at 0.90s
78 Storage Reqs received S at 1.20s,...,0.90s
0.01s
Compute Req received by F at 0.74s
Off-path cause!
78 Storage Req sent by M at 0.75s,...,1.05s

Figure 8: Temporal provenance for scenario Z1. In contrast to the trace tree in Figure 7, the off-path cause (the requests from the maintenance service) does appear and can easily be found by starting at the root and by following the chain of vertices with the highest delay.

both took almost as long. The latter may seem suspicious, but the trace tree contains no further explanation. Similarly, the classic provenance tree for Z1, which are essentially the yellow vertices in Figure 8, offers a more comprehensive explanation compared to the trace tree; however, like the trace tree, it also misses the actual off-path cause. This consistently happened in all experiments with Zipkin and DTaP: since these systems do not reason about temporal causality, the underlying cause was never included in any of their results. On the other hand, Z1's temporal provenance (all vertices in Figure 8) not only captures the information from Zipkin or DTaP, but also explains that the requests from the off-path maintenance service are causing the delay.

7.3 Size of the provenance

The provenance has to be simple enough for the operator to make sense of it. Recall that, before showing the provenance to operators, our debugger (1) prunes vertices that do not contribute to the overall delay, and (2) aggregates subgraphs that are structurally similar. To quantify how well these techniques work, and whether they do indeed lead to a readable explanation, we re-ran the diagnostic queries in Section 7.1 with different subsets of these steps disabled, and we measured the size of the corresponding provenance graphs.

Figure 9 shows our results. The leftmost bars show the size of the raw temporal provenance, which ranged from 748 to 2,564 vertices. A graph of this size would be far too complex for most operators to interpret. However, not all of these

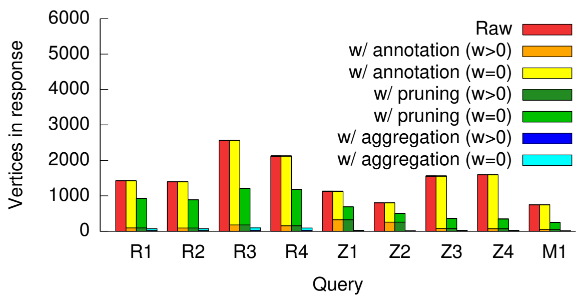


Figure 9: Size of the temporal provenance for all scenarios in Section 5, with different post-processing steps.

vertices actually contribute to the overall delay. The second set of bars shows the number of vertices that Zeno would annotate with a nonzero delay ($w > 0$) and a zero delay ($w = 0$), respectively: only 4.6–32.1% of all vertices actually contributed any delays. However, the subgraphs with nonzero delays nevertheless remain too large to read effectively.

Our first post-processing step prunes vertices and subtrees that are annotated with zero delay and that do not make a causal contribution. As the third set of bars shows, this reduces the size of the graph by more than 30% in all scenarios. The second and final post-processing step coalesces structurally similar subtrees and aggregates their delays. As the rightmost set of bars shows, this is extremely effective and shrinks the graph to between 13 and 93 vertices; the number of vertices that actually contribute delay is between 11 and 28. (Recall that vertices with a causal contribution are preserved even if they do not contribute delay.) A provenance graph of this size should be relatively easy to interpret.

To explain where the large reduction comes from, we sketch the raw provenance tree – without post-processing – for scenario Z1 in Figure 10. The structure of this tree is typical for the ones we have generated. First, there is a relatively small "backbone" (shown in yellow) that functionally explains the result and roughly corresponds to classical provenance. Second, there is a large number of small branches (shown in red) along long sequencing chains (shown in green) that describe the sources of any delay; these are collapsed into a much smaller number of branches, or even a single branch. Third, there are further branches (shown in white) that are connected via sequencing edges but do *not* contribute any delay; these are pruned entirely. The last two categories typically contain the vast majority of vertices, and our post-processing steps shrink them very effectively, which in this case yields the much simpler tree from Figure 8.

7.4 Runtime overhead

Next, we quantified the overhead of collecting the metadata for temporal provenance at runtime. We ran a fixed workload of 1,000 requests in all scenarios, and measured the overall

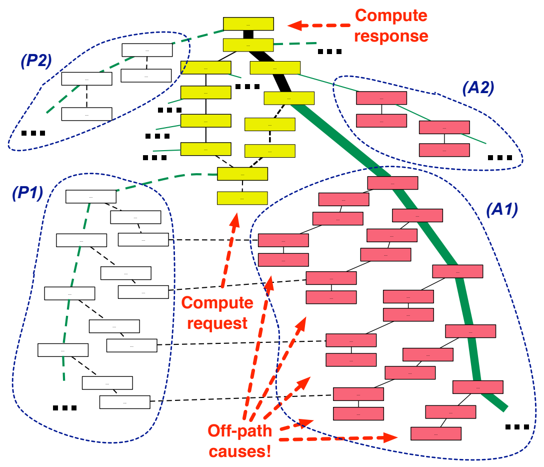


Figure 10: Sketch of the *raw* temporal provenance for scenario Z1. The post-processing steps from Section 5 reduce this to the provenance shown in Figure 8.

latency and the storage needed to maintain provenance. Zipkin is closely based on Dapper, which incurs low overhead in production systems [55]; for instance, instrumenting every request in a web search cluster increased latency by 16.3% and decreased throughput by 1.48% [55]. Temporal provenance mostly uses the data Zipkin collects, but does not modify its collection system; the dtrace [27] extension, which complements Zipkin traces, incurs an additional latency increase of 0.8% and a storage overhead of 270 bytes per *RPC*. In Mininet, each packet consumes 106 bytes, which includes the packet header and timestamps. In RapidNet, maintaining classical provenance alone resulted in a latency increase of 0.3–1.2% and a storage overhead of 145–168 bytes per *input event*. Maintaining temporal provenance causes an additional latency increase of 0.4–1.5% and a storage overhead of 49 bytes per *event*. Notice that, for temporal provenance, it is not enough to merely record input events, since this would not necessarily reproduce the timing or sequence of events.

The total storage consumption also depends on the retention time R. (Recall that Zeno prunes provenance data beyond R.) R needs to be large enough to cover the interval between the root cause and the time the query is issued. Intuitively, a small R should be sufficient because root causes of current issues are usually not in the distant past. To confirm this intuition, and to estimate a suitable value for R, we re-examined our survey of incidents disclosed by Google Cloud Platform [2]. We found 12 timing faults whose descriptions included timestamps for both the symptom and the root cause; in 11 of the 12 cases, the interval between the root cause and the symptom was less than 30 minutes.

7.5 Query processing speed

When the operator queries the temporal provenance, our debugger must execute the algorithm from Section 4 and apply the post-processing steps from Section 5. Since debugging is an interactive process, a quick response is important. To see whether our debugger can meet this requirement, we measured the turnaround time for all queries, as well as the fraction of time consumed by each of the major components.

Figure 11(a) shows our results. We make two high-level observations. First, for scenarios where provenance is captured using deterministic replay (R1–R4), the turnaround time is dominated by the replay and by the storage lookups that would be needed even for classical provenance. This is expected because neither our annotation algorithm nor the post-processing steps are computationally expensive. Second, although the queries vary in complexity and thus their turnaround times are difficult to compare, we observe that none of them took more than 10 seconds, which should be fast enough for interactive use. Notice that this delay is incurred only once per query; the operator can then explore the resulting temporal provenance without further delays.

7.6 Scalability

To evaluate the scalability of Zeno with regard to the network size, we tested the turnaround time and provenance size of query R3 on larger networks with up to 700 nodes. We obtained these networks by adding more servers.

Turnaround time: As we can see from the left part of Figure 11(b), the turnaround time increased linearly with the network size, but it was within 65 seconds for all cases. As the breakdown shows, the increase mainly comes from the latency increase of the historical lookups and of the replay. This is because the additional nodes and traffic caused the size of the logs to increase. This in turn resulted in a longer time to replay the logs, and to search through the events. Profiling [15] shows that log replay is dominated by sending and receiving packets in RapidNet [3] (Recall from Section 6 that the replay engine is based on an existing one from [68].). Because the replay runs on a single machine, we can optimize turnaround time by reducing unnecessary I/O.

Size of the provenance: The right part of Figure 11(c) shows that the size of the raw provenance grew linearly to the network size – by 7x from $1,939$ to $13,960$ vertices – because traffic from additional servers caused additional delays, which required extra vertices to be represented in the provenance. With the annotation and aggregation heuristics applied, the number of vertices that actually contributed delay still grew, because more hops were congested due to busier networks. However, the increase – a factor of 1.5, from 28 to 40 – is much less than the increase in the network size (7x), which suggests that the heuristic scales well.

8 Related Work

Provenance: None of the provenance systems we are aware of [61, 30, 60, 68, 69, 67, 49, 22, 38, 35, 39] can reason about temporal causality, which is essential for diagnosing timing

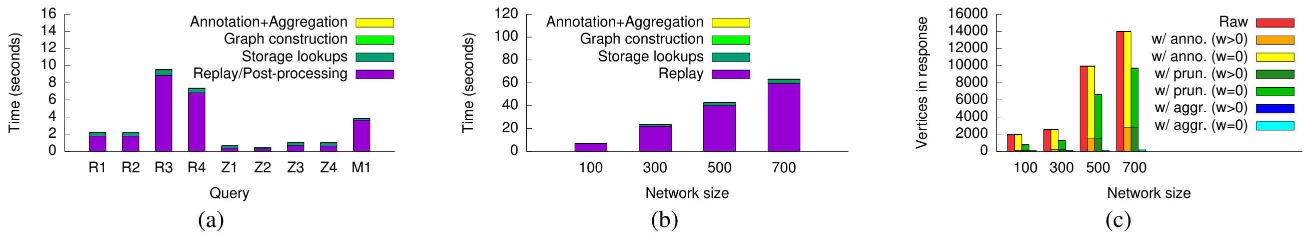


Figure 11: Turnaround for all queries in Section 7.1 (a). Scalability of turnaround (b) and provenance size (c) for R3.

faults. This is true even for DTaP [68] and its predecessor TAP [66], which are "time-aware" only in the very limited sense that they can remember the provenance of past system states. As our experiments confirm, these systems are not able to find off-path causes of timing faults.

Tracing: Tracing systems broadly fall into two classes. The first class of systems *infer* causality, e.g., using log messages [34], static analysis [64, 53], or external annotations [18, 44]; however, the inference is not always accurate, so such systems can have false positives and negatives. For example, Roy et al. [53] localizes network faults in real time by correlating end-host flow metrics with network paths that flows traverse; however, the technique relies on statistical analysis and applies best to huge data centers where the rich structure and massive volume of traffic information reduces imprecision. The second class of systems avoid this imprecision by *recording* causality, at the expense of instrumentation [55, 48, 58, 21, 33, 28, 37, 40, 57, 56]. For example, Canopy [40] annotates traces with performance data; SwitchPointer [57] divides time into epochs and records the epochs during which a packet was forwarded. To our knowledge, our approach is the first to explicitly record temporal *causality* using sequencing edges and thus also the first to offer precise reasoning about the causes of timing behavior.

Performance diagnosis: Existing systems have used machine learning or statistical analysis for performance diagnosis [63, 21, 19, 41, 20, 32] – they perform learning on the normal system behaviors and use learned models for diagnosis. This tends to work well when there is abundant training data, but its power is limited when diagnosing rare anomalies or occasional glitches, which are often the trickiest and most time-consuming problems to debug. Performance diagnosis can also be done by comparing "good" and "bad" instances [51, 52, 50, 54] and analyzing their differences, when both types of instances are available. Since these types of diagnosis do not use causality, the analysis is not always precise. DiffProv [30] does rely on causality, but it focuses exclusively on functional but not temporal causality. We believe that Zeno may be able to benefit from a similar differential diagnosis to narrow down the root causes further.

Timing faults: Our approach is potentially useful for diagnosing timing faults in real-time systems, where tasks have deadlines [31]. Researchers have proposed solutions to control program timing, but they either require specialized hardware [36] or incur significant overhead [29]. Worst-case execution time analysis [59] can estimate an upper bound on the execution time of a program, but it does not reason about the causes of delays.

Queuing theory: Queuing theory [42, 43, 25] has been used to model, analyze, and optimize the performance of distributed systems. This approach, however, assumes a certain distribution of arrival patterns in the input workloads, which may not always hold in practice, and it does not automatically identify the causes of a performance violation. In contrast, temporal provenance can help diagnosing problems in practical systems without assumptions on the input model.

9 Conclusion

Diagnosing timing-related issues is a tricky business that requires expertise and a considerable amount of time. Hence, it seems useful to develop better tools that can at least partially automate this process. Existing tools work well for functional problems, but they fail to identify the root causes of temporal problems; this requires a very different approach that involves different information and a new way of reasoning about causality. We have proposed temporal provenance as a concrete solution to this problem. Although temporal provenance takes the concept of provenance in a somewhat different direction than the existing work on functional provenance, the two lines of work share the same starting point (classical provenance) and thus look very similar to the operator, which helps with usability. The experimental results from our prototype debugger suggest that temporal provenance can provide compact, readable explanations for temporal behavior, and that the necessary metadata can be collected at a reasonable cost.

Acknowledgments: We thank our shepherd Ariosto Panda and the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grants CNS-1563873, CNS-1703936, CNS-1750158, CNS-1703936 and CNS-1801884, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056 and HR0011-17-C-0047.

References

[1] http://zipkin.io/.

[2] https://status.cloud.google.com/summary.

[3] http://netdb.cis.upenn.edu/rapidnet/.

[4] https://nodejs.org/.

[5] https://trypyramid.com/.

[6] https://github.com/ruby/webrick.

[7] http://mininet.org/.

[8] https://status.cloud.google.com/incident/compute/15039.

[9] https://status.cloud.google.com/incident/appengine/14005.

[10] https://status.cloud.google.com/incident/appengine/15005.

[11] https://status.cloud.google.com/incident/compute/15057.

[12] https://status.cloud.google.com/incident/bigquery/18003.

[13] https://status.cloud.google.com/incident/bigquery/18007.

[14] https://status.cloud.google.com/incident/appengine/15023.

[15] https://github.com/gperftools/gperftools.

[16] In-band network telemetry. http://p4.org/wp-content/uploads/fixed/INT/INT-current-spec.pdf.

[17] The P4 language. https://github.com/p4lang/.

[18] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proc. SOSP* (Oct. 2003).

[19] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proc. OSDI* (Oct. 2012).

[20] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. SIGCOMM* (2007).

[21] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *Proc. OSDI* (Dec. 2004).

[22] BATES, A., TIAN, D., BUTLER, K. R., AND MOYER, T. Trustworthy whole-system provenance for the Linux kernel. In *Proc. USENIX Security* (Aug. 2015).

[23] BODLAENDER, H. L. Polynomial algorithms for graph isomorphism and chromatic index on partial k-Trees. *Journal of Algorithms* (1990), 631–643.

[24] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR 44*, 3 (2014).

[25] BOUDEC, J.-Y. L., AND THIRAN, P. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, vol. LNCS 2050. Springer, 2001.

[26] BUNEMAN, P., KHANNA, S., AND WANG-CHIEW, T. Why and where: A characterization of data provenance. In *Proc. ICDT* (Jan. 2001).

[27] CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proc. USENIX ATC* (2004).

[28] CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Whodunit: Transactional profiling for multi-tier applications. In *Proc. SOSP* (Oct. 2007).

[29] CHEN, A., MOORE, W. B., XIAO, H., HAEBERLEN, A., PHAN, L. T. X., SHERR, M., AND ZHOU, W. Detecting covert timing channels with time-deterministic replay. In *Proc. OSDI* (Oct. 2014).

[30] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. The Good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proc. SIGCOMM* (Aug. 2016).

[31] CHEN, A., XIAO, H., PHAN, L. T. X., AND HAEBERLEN, A. Fault tolerance and the five-second rule. In *Proc. HotOS* (May 2015).

[32] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. DSN* (2002).

[33] CHEN, Y.-Y. M., ACCARDI, A., KICIMAN, E., PATTERSON, D. A., FOX, A., AND BREWER, E. A. *Path-based failure and evolution management*. PhD thesis, University of California, Berkeley, 2004.

[34] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The mystery machine: end-to-end performance analysis of large-scale Internet services. In *Proc. OSDI* (Oct. 2014).

[35] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight provenance for smart phone operating systems. In *Proc. USENIX Security* (Aug. 2011).

[36] EDWARDS, S. A., AND LEE, E. A. The case for the precision timed (PRET) machine. In *Proc. DAC* (June 2007).

[37] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *Proc. NSDI* (Apr. 2007).

[38] GEHANI, A., AND TARIQ, D. Spade: Support for provenance auditing in distributed environments. In *Proc. Middleware* (Dec. 2012).

[39] HASAN, R., SION, R., AND WINSLETT, M. The case of the fake picasso: Preventing history forgery with secure provenance. In *Proc. FAST* (2009).

[40] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., ET AL. Canopy: An end-to-end performance tracing and analysis system. In *Proc. SOSP* (2017).

[41] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed diagnosis in enterprise networks. In *Proc. SIGCOMM* (Aug. 2009).

[42] KLEINROCK, L. *Queueing Systems, Volume 1: Theory*. Wiley-Interscience, 1975.

[43] KLEINROCK, L. *Queueing Systems, Volume 2: Computer Applications*. Wiley-Interscience, 1976.

[44] KOSKINEN, E., AND JANNOTTI, J. Borderpatrol: Isolating events for black-box tracing. In *Proc. EuroSys* (Mar. 2008).

[45] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978).

[46] LOGOTHETIS, D., DE, S., AND YOCUM, K. Scalable lineage capture for debugging DISC analysis. Tech. Rep. CSE2012-0990, UCSD.

[47] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative networking. *Communications of the ACM 52*, 11 (Nov. 2009), 87–95.

[48] MILLER, B. P. Dpm: A measurement system for distributed programs. *IEEE Transactions on Computers 37*, 2 (1988), 243–248.

[49] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. In *Proc. USENIX ATC* (May 2006).

[50] NAGARAJ, K., KILLIAN, C., AND NEVILLE, J. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. NSDI* (Apr. 2012).

[51] NOVAKOVIĆ, D., VASIĆ, N., NOVAKOVIĆ, S., KOSTIĆ, D., AND BIANCHINI, R. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. USENIX ATC* (June 2013).

[52] REYNOLDS, P., KILLIAN, C. E., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *Proc. NSDI* (May 2006).

[53] ROY, A., ZENG, H., BAGGA, J., AND SNOEREN, A. C. Passive realtime datacenter fault detection and localization. In *Proc. NSDI* (2017).

[54] SAMBASIVAN, R. R., ZHENG, A. X., DE ROSA, M., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *Proc. NSDI* (Apr. 2011).

[55] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure.

[56] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *Proc. OSDI* (2016).

[57] TAMMANA, P., AGARWAL, R., AND LEE, M. Distributed network monitoring and debugging with switchpointer. In *Proc. NSDI* (2018).

[58] TIERNEY, B., JOHNSTON, W., CROWLEY, B., HOO, G., BROOKS, C., AND GUNTER, D. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. HPDC* (July 1998).

[59] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. The worst-case execution-time problem–overview of methods and survey of tools. *ACM TECS 7*, 3 (May 2008), 36:1–36:53.

[60] WU, Y., CHEN, A., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Automated bug removal for software-defined networks. In *Proc. NSDI* (Mar. 2017).

[61] WU, Y., ZHAO, M., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Diagnosing missing events in distributed systems negative provenance. In *Proc. SIGCOMM* (Aug. 2014).

[62] YANG, C.-Q., AND MILLER, B. P. Critical path analysis for the execution of parallel and distributed programs. In *Proc. DCS* (1988).

[63] ZHANG, X., TUNE, E., HAGMANN, R., JNAGAL, R., GOKHALE, V., AND WILKES, J. CPI 2: CPU performance isolation for shared compute clusters. In *Proc. EuroSys* (Apr. 2013).

[64] ZHAO, X., ZHANG, Y., LION, D., ULLAH, M. F., LUO, Y., YUAN, D., AND STUMM, M. lprof: A non-intrusive request flow profiler for distributed systems. In *Proc. OSDI* (Oct. 2014).

[65] ZHOU, W. *Secure Time-Aware Provenance For Distributed Systems*. PhD thesis, University of Pennsylvania, 2012.

[66] ZHOU, W., DING, L., HAEBERLEN, A., IVES, Z., AND LOO, B. T. TAP: Time-aware provenance for distributed systems. In *Proc. TaPP* (June 2011).

[67] ZHOU, W., FEI, Q., NARAYAN, A., HAEBERLEN, A., LOO, B. T., AND SHERR, M. Secure network provenance. In *Proc. SOSP* (Oct. 2011).

[68] ZHOU, W., MAPARA, S., REN, Y., LI, Y., HAEBERLEN, A., IVES, Z., LOO, B. T., AND SHERR, M. Distributed time-aware provenance. In *Proc. VLDB* (Aug. 2013).

[69] ZHOU, W., SHERR, M., TAO, T., LI, X., LOO, B. T., AND MAO, Y. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD* (June 2010).

A Formal Model

Temporal provenance preserves all properties of classical provenance (validity, soundness, completeness, and minimality). We have obtained the corresponding proofs by extending the formal model from TAP/DTaP [65]. Although there are some parts of the proof from [65] that require few or no changes (e.g., because they only relate to functional provenance and not to sequencing), we present the full formal model here for completeness. Our extensions include the following:

- Temporal provenance has a different set of vertex types (Section 3.3) and contains sequencing edges (Section 4.1); consequently, temporal provenance graphs are constructed differently (Section A.2).
- The validity property, in addition to its prior requirements from TAP, requires that the temporal provenance include all the events necessary to reproduce the execution temporally (Section A.3).
- The proofs follow the same structure as in TAP, but are adjusted to handle the different graph structure and the stronger validity property of temporal provenance (Section A.4).

We have also formally modeled the properties of the delay annotations that our algorithm creates (and that were not part of [65]):

- Definitions of direct delay and transitive delay; and a theorem states that each vertex is labeled with the amount of delay that is contributed by the subtree that is rooted at that vertex (Section A.5).
- A theorem states that the annotations do correspond to the "potential for speedup" that we intuitively associate with the concept of delay (Section A.6).

A.1 Background: Execution Traces

To set stage for the discussion, we introduce some necessary concepts for reasoning about the execution of the system in our temporal provenance model.

An *execution trace* of an NDlog program can be characterized by a sequence of *events* that take place in the system, starting from the initial system state. Each event on a node captures the execution of a particular rule r that is triggered by a certain tuple τ, under the existence of some other tuples on the node, and that results in a new tuple being derived or an existing tuple being underived (i.e., lost). We formally define them below.

Definition (Event)*: An event $d@n$ on a node n is represented by $d@n = (\tau, r, t_s, t_e, c, \pm\tau')$, where*

- *τ is the tuple that triggers the event,*
- *r is the derivation rule that is being triggered,*
- *t_s is the time at which r is triggered (called start timestamp),*
- *t_e is the time at which r finishes its execution (called end timestamp),*
- *c is the set of tuples that are preconditions of the event, which must exist on n at time t_s, and*
- *τ' is the tuple that is derived $(+)$ or underived $(-)$ as a result of the derivation.*

Definition (Trace)*: A trace $\mathcal{E}$ is a sequence of events $\langle d_1@n_1, d_2@n_2, \ldots, d_k@n_k\rangle$ that reflects an execution of the system from the initial state $\mathcal{S}_0$, i.e.,*

$$\mathcal{S}_0 \xrightarrow{d_1@n_1} \mathcal{S}_1 \xrightarrow{d_2@n_2} \cdots \xrightarrow{d_k@n_k} \mathcal{S}_k.$$

To quantify the timing behaviors of the system, it is necessary to reason about the order among events. Ideally, we would like to have a total ordering among all events in all nodes in the system; however, due to the lack of fully synchronized clocks, this is difficult to achieve in distributed systems. To address this, we introduce the concept of trace *equivalence* that preserves the total ordering of events on each node, without imposing a total ordering among events across nodes. Intuitively, two traces $\mathcal{E}$ and $\mathcal{E}'$ are considered equivalent if the subsequence of events that every node observes in $\mathcal{E}$ is the same as that is observed in $\mathcal{E}'$.

Definition (Subtrace): *$\mathcal{E}'$ is a subtrace of $\mathcal{E}$ (written as $\mathcal{E}' \subseteq \mathcal{E}$) iff $\mathcal{E}'$ is a subsequence of $\mathcal{E}$. We denote by $\mathcal{E}|n$ the subtrace of $\mathcal{E}$ that consists of all and only the events of $\mathcal{E}$ that take place on node n.*

Definition (Equivalence): *Two traces $\mathcal{E}$ and $\mathcal{E}'$ are equivalent (written as $\mathcal{E} \sim \mathcal{E}'$) iff for all nodes n, $\mathcal{E}|n = \mathcal{E}'|n$.*

By definition, the equivalence relation is transitive: if $\mathcal{E} \sim \mathcal{E}'$ and $\mathcal{E}' \sim \mathcal{E}''$, then $\mathcal{E} \sim \mathcal{E}''$.
Example: As an example, consider the following traces:

$$\mathcal{E}_1 = \langle d_1@n_1, d_2@n_2, d_3@n_1, d_4@n_2\rangle,$$

$$\mathcal{E}_2 = \langle d_1@n_1, d_2@n_2, d_4@n_2, d_3@n_1\rangle,$$

$$\mathcal{E}_3 = \langle d_1@n_1, d_2@n_2, d_3@n_1\rangle.$$

It is easy to observe that $\mathcal{E}_1$ and $\mathcal{E}_2$ are equivalent, since $\mathcal{E}_1|n_1 = \mathcal{E}_2|n_1 = \langle d_1@n_1, d_3@n_1\rangle$ and $\mathcal{E}_1|n_2 = \mathcal{E}_2|n_2 = \langle d_2@n_2, d_4@n_2\rangle$. In contrast, $\mathcal{E}_3$ is a subtrace of $\mathcal{E}_1$, but it is not equivalent to $\mathcal{E}_1$ (since $\mathcal{E}_3|n_2 \neq \mathcal{E}_1|n_2$).

A.2 Graph construction

We now describe our algorithm for constructing the temporal provenance that explains the reasons for a delay between two events. As discussed in Section 4.6, temporal provenance is *recursive* – the temporal provenance for $[e', e]$ includes, as subgraphs, the temporal provenances of all events that contributed to both e and the delay from e' to e. Leveraging this property, we can construct the temporal provenance of a pair of events "on demand" using a top-down procedure, without the need to materialize the entire provenance graph.

Towards this, we first define a function RAW-QUERY that, when called on a vertex v in the temporal provenance graph, returns two sets of immediate children of v: the first consists of vertices that are connected to v via causal edges, and the second consists of vertices that are connected to v via sequencing edges. Given an execution trace $\mathcal{E}$ of the system, the temporal provenance for a diagnostic query T-QUERY(e',e) can be obtained by first constructing a vertex v_e that describes e (i.e., a DRV/UDRV/RCV vertex for e), and then calling RAW-QUERY recursively on the vertices starting from v_e until reaching the leaf vertices (lines 1-18); note that a vertex returned by a RAW-QUERY call is connected to its parent vertex via either a causal edge and/or a sequencing edge, depending on the set(s) it belongs to (lines 12-17). The resulting graph, denoted by $G(e', e, \mathcal{E})$, includes all necessary events to explain both e and the delay from e' to e. However, as it requires delay annotation (Sections 4.3–4.5) to be useful for diagnostics, we refer to it as the "raw" temporal provenance of T-QUERY(e',e).

The RAW-QUERY(v) procedures rely on a helper function called PREV-VERTEX to find vertices that are connected to v via sequencing edges. For ease of exposition, we first explain the pseudo-code of PREV-VERTEX in Figure 12: given an interval $[t', t]$ and a node N (supplied by RAW-QUERY calls), PREV-VERTEX finds the chain of preceding events that happened on N during $[t', t]$; it first locates the last event v whose execution ends at t and constructs a corresponding vertex (lines 51-60); it then shortens the interval to until the starting timestamp of v and recursively find prior events on N (line 61); it stops until the interval is exhausted or if no event can be found (line 67); finally, it recursively connects this chain of events using sequencing edges and returns the last event in the chain to its caller (line 65-66). For example, consider the provenance graph from Figure 13: a PREV-VERTEX($[2.5s, 3.5s]$,Y) call will first find the DRV(F) event, which ends at exactly $t = 3.5s$; it constructs a vertex and shortens the interval to $[2.5s, 2.5s]$, by excluding the execution time of DRV(F); this interval is passed into a recursive call – PREV-VERTEX($[2.5s, 2.5s]$,Y) – that finds the event of and constructs a vertex for INS(G); the recursion then stops because the interval becomes empty (because INS(G) takes a positive amount of time); the two constructed vertices are connected via sequencing edges and the last event in the chain – DRV(F) – is returned to the caller.

Figure 12 shows the pseudo-code for RAW-QUERY(v) depending on the type of v (DRV, UDRV, SND, RCV, INS and DEL). Note that each

```
function CONSTRUCT-GRAPH($v_e$)
    $G \leftarrow \{v_e\}$ // the "raw" temporal provenance graph
    // a queue of vertices that need explanation
    $NodeToProcess \leftarrow \{v_e\}$
    while $NodeToProcess \neq \emptyset$ do
        $v \leftarrow NodeToProcess$.POP()
        $S, S' \leftarrow$ RAW-QUERY($v$)
        for $v' \in \{S \cup S'\}$ do
            if $v' \notin G$ then
                $G \leftarrow G \cup v'$ // add vertices
                $NodeToProcess$.PUSH($v'$)
        for $v' \in S$ do
            // add causal edges
            $G \leftarrow G \cup (v', v)_{causal}$
        for $v' \in S'$ do
            // add sequencing edges
            $G \leftarrow G \cup (v', v)_{sequencing}$
    RETURN $G$
function RAW-QUERY(DRV($[t_s, t_e]$,N,$\tau$, $\tau$:- $\tau_1, \tau_2, \ldots, \tau_m$))
    S $\leftarrow \emptyset$
    $t_e^{max} \leftarrow 0$ // the last precondition was satisfied at $t_e^{max}$
    for $\tau_i \in \{\tau_1, \tau_2, \ldots, \tau_m\}$ do
        Find $d_i@N = (\tau', r, t'_s, t'_e, \{c_1, c_2, \ldots, c_k\},\ \pm\tau_i) \in \mathcal{E}$:
            $t'_e \leq t_s$ and $t'_e$ is maximized
        $t_e^{max} \leftarrow$ MAX($t_e^{max}$, $t'_e$)
        if $r = r_{\mathsf{ins}}$ then
            S $\leftarrow$ S $\cup$ INS($[t'_s, t'_e]$,$N$,$\tau_i$)
        else if $r = r_{\mathsf{rcv}}$ then
            S $\leftarrow$ S $\cup$ RCV($[t'_s, t'_e]$,$N \leftarrow r.N$,$\pm\tau_i$)
        else
            S $\leftarrow$ S $\cup$ DRV($[t'_s, t'_e]$,$N$,$\tau_i$, $\tau_i$:-$\tau'$,$c_1$,$c_2$,$\ldots$,$c_k$)
    // include all preceding events that happened after the last
    // precondition was satified and before the derivation of $\tau$
    RETURN (S; PREV-VERTEX($[t_e^{max}, t_s]$, $N$))
function RAW-QUERY(INS($[t_s, t_e]$,$N$,$\tau$))
    RETURN ($\emptyset$; $\emptyset$)
function RAW-QUERY(SND($[t_s, t_e]$,$N \rightarrow N'$,$\pm\tau$))
    Find $d@N = (\tau', r, t'_s, t'_e, \{c_1, c_2, \ldots, c_k\}, \pm\tau) \in \mathcal{E}$:
        $t'_e \leq t_s$ and $r \neq r_{\mathsf{rcv}}$ and $t'_e$ is maximized.
    if $r = r_{\mathsf{ins/del}}$ then
        RETURN (INS/DEL($[t'_s, t'_e]$,$N$,$\tau$);
                    PREV-VERTEX($[t'_e, t_s]$,$N$))
    else
        RETURN (DRV/UDRV($[t'_s, t'_e]$,$N$,$\tau$, $\tau$:- $\tau'$, $c_1, c_2, \ldots$);
                    PREV-VERTEX($[t'_e, t_s]$, $N$))
function RAW-QUERY(RCV($[t_s, t_e]$,$N \leftarrow N'$,$\pm\tau$))
    Find $d@N'$=($\tau'$,$r$,$t'_s$,$t'_e$,$\pm\tau$) $\in \mathcal{E}$:
        $t'_e \leq t_s$ and $r = r_{\mathsf{snd}}$ and $t'_e$ is maximized
    $v \leftarrow$ SND($[t'_s, t'_e]$,$N' \rightarrow N$,$\pm\tau$)
    // a remote sequencing edge exists from the SND vertex
    RETURN ($v$; $v$)
function PREV-VERTEX($[t', t]$,$N$)
    if $t' < t$ then
        // If an immediate preceding event exists, then add a
        // sequencing edge from the corresponding vertex.
        if $\exists$ $d@N$=($\tau'$,$r$,$t_s$,$t_e$,$\{c_1, c_2, ...\}$,$\pm\tau$): $t_e = t$ then
            $v \leftarrow null$
            if $r = r_{\mathsf{snd}}$ then
                $v \leftarrow$ SND($[t_s, t_e]$,$N \rightarrow r.N$,$\pm\tau$)
            else if $r = r_{\mathsf{rcv}}$ then
                $v \leftarrow$ RCV($[t_s, t_e]$,$N \leftarrow r.N$,$\pm\tau$)
            else if $r = r_{\mathsf{ins/del}}$ then
                $v \leftarrow$ INS/DEL($[t_s, t_e]$,$N$,$\tau$)
            else
                $v \leftarrow$ DRV/UDRV($[t_s, t_e]$,$N$,$\tau$,$\tau$:-$\tau'$,$c_1$,$c_2$,...)
            $v'$ = PREV-VERTEX($[t', t_s]$,$N$)
            if $v'! = null$ then
                // recusively add preceding events until the
                // entire $[t', t]$ interval is explained
                $G \leftarrow G \cup (v', v)_{sequencing}$
            RETURN $v$
    RETURN $null$
function RAW-QUERY(UDRV($[t_s, t_e]$,N,$\tau$, $\tau$:- $\tau_1, \tau_2, \ldots, \tau_m$))
    S $\leftarrow \emptyset$
    $t_e^{max} \leftarrow 0$ // the last precondition was satisfied at $t_e^{max}$
    for $\tau_i \in \{\tau_1, \tau_2, \ldots, \tau_m\}$ do
        Find $d_i@N = (\tau', r, t'_s, t'_e, \{c_1, c_2, \ldots, c_k\},\ \pm\tau_i) \in \mathcal{E}$:
            $t'_e \leq t_s$ and $t'_e$ is maximized
        $t_e^{max} \leftarrow$ MAX($t_e^{max}$, $t'_e$)
        if $r = r_{\mathsf{ins/del}}$ then
            S $\leftarrow$ S $\cup$ INS/DEL($[t'_s, t'_e]$,$N$,$\tau_i$)
        else if $r = r_{\mathsf{rcv}}$ then
            S $\leftarrow$ S $\cup$ RCV($[t'_s, t'_e]$,$N \leftarrow r.N$,$\pm\tau_i$)
        else
            S $\leftarrow$ S $\cup$ DRV/UDRV($[t'_s, t'_e]$,$N$,$\tau_i$, $\tau_i$:-$\tau'$,$c_1$,$c_2$,$\ldots$,$c_k$)
    // includes all preceding events that happened after the last
    // precondition was satified and before the underivation of $\tau$
    RETURN (S; PREV-VERTEX($[t_e^{max}, t_s]$, $N$))
function RAW-QUERY(DEL($[t_s, t_e]$,$N$,$\tau$))
    RETURN ($\emptyset$; $\emptyset$)
```

Figure 12: Algorithm for constructing temporal provenance graph for a given execution trace $\mathcal{E}$. The trace $\mathcal{E}$ consists of events (Definition A.1), which are recorded at runtime or reconstructed via replay. The function RAW-QUERY(v), when called on a vertex v, returns two sets of immediate children of v, which are connected to v via causal edges and sequencing edges, respectively. It calls PREV-VERTEX($[t', t]$,N) as a subprocedure, which finds a chain of vertices connected via sequencing edges that immediately precedes v during $[t', t]$.

Algorithm 1 Extracting traces from provenance

```
1: // This algorithm extracts the trace $\mathcal{A}(e', e, \mathcal{E})$ from the "raw" temporal provenance $G(e', e, \mathcal{E})$; for ease of explanation, we rewrite $G$ as $(V, E)$, where $V$ represents vertices and $E$ represent edges
2: proc EXTRACTTRACE($G = (V, E)$)
3: // Calculate the out-degree of every vertex in $G$
4: for all $v \in V$ do $degree(v) \leftarrow 0$
5: for all $e = (v, v') \in E$ do $degree(v)$++
6: // Generate the race based on topological sort
7: $trace \leftarrow \emptyset$
8: $NodeToProcess \leftarrow V$
9: while $NodeToProcess \neq \emptyset$ do
10:    // Select the next event based on topological ordering and timestamps
11:    select $v \in NodeToProcess : degree(v) = 0$ and $\nexists v'$
         that is located on the same node and has a larger end timestamp
12:    $NodeToProcess$.REMOVE($v$)
13:    if type($v$) = DRV or UDRV or SND then
14:        $preconditions \leftarrow \emptyset$
15:        for $\forall\ (v', v) \in E$ s.t. $(v', v)$ is a causal edge do
16:            $preconditions$.ADD(tuple($v'$)) // tuple($v'$) is a precondition
17:        // find the trigger from the preconditions
18:        $trigger \leftarrow \tau' \in preconditions$:
             (a) $\tau'$ is a message, or (b) $\tau'$ is a state and
             $\nexists \tau'' \in preconditions$ that has a larger end timestamp
19:        $preconditions$.DELETE($trigger$)
20:        $event \leftarrow (trigger$, rule($v$), startTime($v$),
21:                 endTime($v$), $preconditions$, tuple($v$))
22:        $trace.push\_front(event)$
23:    if type($v$) = RCV then
24:        $output \leftarrow tuple(v)$
25:        $trigger \leftarrow tuple(v')$: $(v', v) \in E$ s.t. type($v'$) = SND
26:        $event \leftarrow (trigger$, rule($v$), startTime($v$),
27:                 endTime($v$), $\emptyset$, tuple($v$))
28:        $trace.push\_front(event)$
29:    if type($v$) = INS or DEL then
30:        $output \leftarrow tuple(v)$
31:        $event \leftarrow (\emptyset$, rule($v$), startTime($v$),
32:                 endTime($v$), $\emptyset$, tuple($v$))
33:        $trace.push\_front(event)$
34:    for all $(v', v) \in E$, $degree(v') \leftarrow degree(v') - 1$
35: return $trace$
```

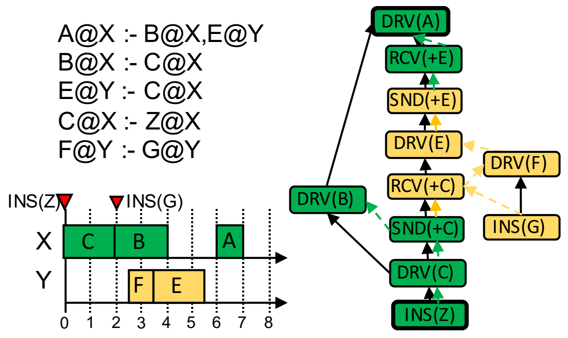


Figure 13: An example scenario, with NDlog rules at the top left, the timing of a concrete execution in the bottom left, and the temporal provenance graph at the right. The query is T-QUERY(INS(Z), DRV(A)); the start and end vertices are marked in bold. Vertex names are shortened and some fields are omitted for clarity.

DRV or UDRV vertex is also associated with the corresponding derivation rule. Next, we explained the pseudo-code of RAW-QUERY(v) for each vertex type in more detail. For ease of exposition, we use the provenance graph from Figure 13:

- To explain a SND vertex, we find the most recent event in the original trace that produced (or deleted) the tuple that is being sent (line 37), construct an INS (or DEL) or a DRV (or UDRV) vertex for the found event, and add an incoming causal edge from the constructed vertex (lines 38-41); in addition, a SND vertex has an incoming sequencing edge from the chain of preceding events that happened after the message was produced or deleted (the PREV-VERTEX calls in lines 39 and line 41). For example, in the temporal provenance graph from Figure 13, the SND(+C) vertex has a causal edge from the DRV(C) vertex, because DRV(C) functionally triggered SND(+C); in addition, a PREV-VERTEX($[2s, 2s]$,X) call adds a sequencing edge from DRV(C) to SND(+C), as the former directly precedes the latter.
- A RCV has an incoming causal edge and an incoming remote sequencing edge from the SND vertex for the received message (lines 42-46). This is the case for RCV(+E) and RCV(+C) in Figure 13.
- A DRV vertex for a rule `A:-B,C,D` has an incoming causal edge for each precondition (B, C, and D) that leads to the vertex that produced the corresponding tuple (line 22); this can be an INS, a RCV, or another DRV (lines 25-30); in addition, a DRV vertex has an incoming sequencing edge from the chain of preceding events that happened after the last precondition was satisfied (the PREV-VERTEX call in line 33). For example, in the provenance from Figure 13, DRV(F) has a causal edge from its (only) precondition INS(G); in addition, a PREV-VERTEX($[2.5s, 3.5s]$,Y) finds the preceding event RCV(+C) that occurred between INS(G) ended and DRV(F) started.
- An INS vertex corresponds to the insertion of a base tuple, which cannot be explained further; thus, it has no incoming edges (line 35). This is true for INS(Z) and INS(G) in Figure 13.
- The edges for the negative "twins" of these vertices – UDRV and DEL – are analogous.

A.3 Properties

Given the "raw" temporal provenance $G(e', e, \mathcal{E})$ of a diagnostic query T-QUERY(e',e) in an execution trace $\mathcal{E}$, we say that $G(e', e, \mathcal{E})$ 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 algorithm for extracting such a subtrace, and then formally define these properties and their proofs.

Definition (Trace Extraction): *Given a temporal provenance $G(e', e, \mathcal{E})$, the trace $A(e', e, \mathcal{E})$ is extracted by running Algorithm 1 based on topological sort.*

Algorithm 1 converts the vertices in the provenance graph to events and then uses a topological ordering and timestamps to assemble the events into a trace. In particular, Line 13-33 implements the construction of one individual event, where the information of a rule evaluation (such as triggering event, conditions, and action) is extracted from vertices in $G(e', e, \mathcal{E})$: a DRV/UDRV/SND vertex and their children; a pair of RCV and SND vertices; or a INS/DEL vertex. In the algorithm, type(v), tuple(v), rule(v), startTime(v) and endTime(v) denote the vertex type, the tuple, the derivation rule, the start timestamp, and the end timestamp of the vertex v, respectively. For example, Algorithm 1 extracts the following trace from the provenance graph in Figure 13: $\langle$INS(Z)@X, DRV(C)@X, SND(+C)@X, INS(G)@Y, DRV(B)@X, RCV(+C)@Y,
DRV(F)@Y, DRV(E)@Y, SND(+E)@Y, RCV(+E)@X, DRV(A)@$X\rangle$.

We will show that the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ obtained from Algorithm 1 satisfies the following four properties.

Definition (Soundness): *A subtrace $\mathcal{A}$ extracted from $G(e', e, \mathcal{E})$ is sound if and only if it is a subtrace of some trace $\mathcal{E}'$ that is equivalent to $\mathcal{E}$, i.e., $\mathcal{A} \subseteq \mathcal{E}' \sim \mathcal{E}$.*

Intuitively, the soundness property means that $\mathcal{A}(e', e, \mathcal{E})$ must preserve all the happens-before relationships among events and the exact timestamps of events in the original execution trace obtained from running the NDlog program. Ideally, we would like $\mathcal{A}(e', e, \mathcal{E})$ to be a subtrace of $\mathcal{E}$, but without synchronized clocks, we cannot always order concurrent events on different nodes. However, for practical purposes $\mathcal{E}$ and $\mathcal{E}_0$ are indistinguishable: each node observes the same sequence of events in the same order and at the same times.

Definition (Completeness)*: A subtrace $\mathcal{A}$ extracted from $G(e', e, \mathcal{E})$ is complete if and only if it ends with the event e and e happens at the same time as in $\mathcal{E}$.*

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

Definition (Validity)*: A subtrace $\mathcal{A}$ extracted from $G(e', e, \mathcal{E})$ is valid if and only if, given the initial state S_0, for every event $d_i@N_i = (\tau_i, r_i, t_i, t'_i, c_i, \pm\tau'_i) \in \mathcal{A}$, the following holds:*

(a) there exists $d_j@N_j = (\tau_j, r_j, t_j, t'_j, c_j, \pm\tau'_j)$ that precedes $d_i@N_i$ in $\mathcal{A}$ such that $\tau_i = \tau'_j$;

(b) for all $\tau_k \in c_i$, we have $\tau_k \in S_{i-1}$, where $\mathcal{S}_0 \xrightarrow{d_1@n_1} \mathcal{S}_1 \xrightarrow{d_2@n_2} \dots \xrightarrow{d_{i-1}@n_{i-1}} \mathcal{S}_{i-1}$; and

(c) based on the conditions (a) and (b), consider the set of all events P_i such that $d_k@N_k \in P_i$ generates $\tau_k \in (c_i \cup \tau_i)$; denote $d_j@N_j$ as the latest event in P_i; if $N_j = N_i$ and $t'_j < t_i$, there must exist a set of events $\{d_p^1@N_i, ..., d_p^n@N_i\} \in \mathcal{A}$ such that: $t'_j = t_p^1$; ${t'}_p^m = t_p^{m+1}, 1 \le m < n$; and ${t'}_p^n = t_i$.

Intuitively, the validity property means that $\mathcal{A}(e', e, \mathcal{E})$ must correspond to a correct execution of the NDlog program both in terms of functionality and timing. Condition (a) states that any event that triggers a rule evaluation must be generated before the rule is evaluated. Condition (b) states that the preconditions of the rule evaluation must hold at the time of the rule evaluation. Finally, condition (c) requires that the evaluation is "work-conserving": the node cannot be idle when it is ready to compute a derivation.

Definition (Minimality)*: A subtrace $\mathcal{A}$ extracted from $G(e', e, \mathcal{E})$ is minimal iff there exists no trace $\mathcal{E}'$ such that: (a) there $\exists d_i@N_i$ where $d_i@N_i \in \mathcal{A}$ and $d_i@N_i \notin \mathcal{E}'$; (b) $\mathcal{E}'$ is valid, sound, and complete.*

Intuitively, minimality means that $\mathcal{A}(e', e, \mathcal{E})$ should not contain any events that are not necessary to reproduce e. If this property were omitted, $\mathcal{A}(e', e, \mathcal{E})$ could trivially output the complete trace $\mathcal{E}$.

A.4 Proofs

Lemma 1 *For any execution $\mathcal{E}$, and a temporal provenance query* T-QUERY*(e',e), the provenance graph $G(e', e, \mathcal{E})$ is acyclic.*

Proof. We first show that if there exists a cycle in $G(e', e, \mathcal{E})$, the cycle cannot include two vertices located on different nodes. Suppose there exists a cycle that contains two vertices v_1 and v_2 located on N_1 and N_2 respectively. Then the cycle must contain a least one pair of SND and RCV vertices in both the path from v_1 to v_2, and the path from v_2 and v_1. Each SND and RCV corresponds to a message communication which takes a positive amount of time. Therefore, traversing from v_1 along the cycle back to v_1 results in an absolute increment in the timestamp. This is a contradiction.

If all the vertices in the cycle are located on the same node, then we can order the vertices according to their associated timestamps (now all the timestamps are with respect to the same local clock). Such order corresponds to the precedence of events in the execution. As time always progresses forward, such cycle cannot exist in $G(e', e, \mathcal{E})$. □

Theorem 2 *$A(e', e, \mathcal{E})$ is sound.*

Proof. We need to show that a) all the events in $\mathcal{A}(e', e, \mathcal{E})$ also appear in $\mathcal{E}$ at the same time (and thus in any $\mathcal{E}_0 \sim \mathcal{E}$), and b) the local event ordering pertains on each node, that is, for any two events $d_1@N_i$ and $d_2@N_i$ in $\mathcal{A}(e', e, \mathcal{E})$ that are located on the same node N_i, $d_1@N_i$ precedes $d_2@N_i$ in $\mathcal{A}(e', e, \mathcal{E})$ iff $d_1@N_i$ precedes $d_2@N_i$ in $\mathcal{E}$.

Condition a. We perform a case analysis by considering the type of the root vertex of $G(e', e, \mathcal{E})$:

- DRV. According to Algorithm 1 (lines 13-22), an event $d_i@N_i$ is generated and included in $\mathcal{A}(e', e, \mathcal{E})$ for each DRV vertex (and its children) in the provenance graph $G(e', e, \mathcal{E})$. However, by construction, each DRV vertex v corresponds to an rule evaluation in $\mathcal{E}$. In our model, the rule evaluation is modeled as an event $d_j@N_j = (\tau_j, r_j, t_j, t'_j, \{c_j^1, ..., c_j^p\}, \pm\tau'_j)$, where τ_j is the trigger event, r_j and $[t_j, t'_j]$ are the rule used in and the time interval of the rule evaluation, c_j^k represents preconditions, and $\pm\tau'_j$ is the generated update. We need to show that $d_i@N_i$ is identical to $d_j@N_j$. This follows straightforwardly from the construction of $G(e', e, \mathcal{E})$: The RAW-QUERY(v) procedures generate a DRV vertex v by: (a) find a derivation event $d_j@N_j$ from $\mathcal{E}$, (b) add incoming edges from the trigger event (a vertex for τ_j), and (c) add incoming edges from the preconditions (vertices for $\{c_j^1, ..., c_j^p\}$). Algorithm 1 reverses this process and generates event $d_i@N_i$ from these information, which is extracted from $d_j@N_j$, and therefore $d_i@N_i = d_j@N_j$.
- RCV/INS/DEL/UDRV/SND Following the same argument for the DRV case above, we can prove that condition (a) holds.

Condition b. According to Algorithm 1 (specifically, Line 11), $d_1@N_i$ precedes $d_2@N_i$ in $\mathcal{A}(e', e, \mathcal{E})$, iff $d_2@N_i$ has a larger timestamp than $d_1@N_i$. However, $d_2@N_i$ is assigned a larger timestamp iff $d_1@N_i$ precedes $d_2@N_i$ in the actual execution $\mathcal{E}$. Note that events on different nodes may be reordered in $\mathcal{A}(e', e, \mathcal{E})$, but this is captured by the equivalence ($\sim$) relation. □

Theorem 3 *$A(e', e, \mathcal{E})$ is complete.*

Proof. We need to show that a) $\mathcal{A}(e', e, \mathcal{E})$ contains an event $d_i@N_i$ that generates e at the same time as in $\mathcal{E}$, and b) $d_i@N_i$ is the last event in $\mathcal{A}(e', e, \mathcal{E})$.

Condition a. By construction, the vertex for e has incoming edges from vertices representing the triggering event τ and all preconditions $c_1, ..., c_p$ (if any). Algorithm 1 (specifically, Lines 13-28) construct an event $(\tau, r, t, t', c, \pm e)$, where r and $[t, t']$ are the rule name and time interval encoded in the vertex. Note that the tuple τ as well as the timestamps t and t' are exactly the ones that are extracted from $\mathcal{E}$ (Algorithm 12).

Condition b. Now we have proved that some event $d_i@N_i$ that generates e must exist in $\mathcal{A}(e', e, \mathcal{E})$, we next show that $d_i@N_i$ is the last event in $\mathcal{A}(e', e, \mathcal{E})$. The provenance graph $G(e', e, \mathcal{E})$ is rooted by a vertex that corresponds to e. Since all other vertices in $G(e', e, \mathcal{E})$ have a directed path to the root vertex, the corresponding events must all be ordered before $d_i@N_i$, so $d_i@N_i$ must necessarily be the last event in the subtrace. □

Theorem 4 *$A(e', e, \mathcal{E})$ is valid.*

Proof. Lemma 1 shows that any provenance graph $G(e', e, \mathcal{E})$ is acyclic, and thus $G(e', e, \mathcal{E})$ has a well-defined height: the length of the longest path from any leaf to e. We prove validity using structural induction on the height of the provenance graph $G(e', e, \mathcal{E})$.

Base case: The height of $G(e', e, \mathcal{E})$ is one. In this case, e must be an insertion or deletion of a base tuple; $G(e', e, \mathcal{E})$ contains a single INS (or DEL) vertex that corresponds to the update of the base tuple. Therefore, $\mathcal{A}(e', e, \mathcal{E})$ consists of a single event and is trivially valid, because the event has neither a trigger nor any precondition (Algorithm 1 lines 29–33).

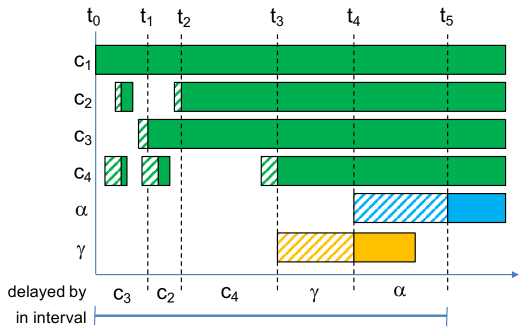


Figure 14: Illustration for the definition of direct and transitive delay. Shaded boxes represent intervals where a tuple was being derived, and solid boxes represent intervals where the tuple existed. The derivation is $\alpha : -c_1, c_2, c_3, c_4$, and the interval in question is $[t_0, t_5]$; γ is an unrelated tuple whose derivation just happened to be sequenced before that of α.

Induction case: Suppose the validity of the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ holds for any provenance graph with height less than k ($k \geq 1$). Consider a provenance graph $G(e', e, \mathcal{E})$ with height $k+1$. We perform a case analysis by considering the type of the root vertex of $G(e', e, \mathcal{E})$. For every event $d_i@N_i = (\tau_i, r_i, t_i, t'_i, c_i, \pm\tau'_i) \in \mathcal{A}(e', e, \mathcal{E})$, we prove that the three conditions in Definition A.3 hold.

- DRV. We know that, by construction, the DRV vertex has an incoming edge from vertices representing the triggering event τ and all preconditions $c_1, ..., c_p$. By the induction hypothesis, Algorithm 1 outputs a valid trace $d_1@N_1, ..., d_j@N_j$ for the subgraph for the trigger event τ, where $d_j@N_j$ corresponds to the generation of τ (following the completeness property proved in Theorem 3). Because of the nature of Algorithm 1 (which is based on topological sort), $d_j@N_j$ must be ordered before $d_i@N_i$, which satisfies condition (a) in the definition of validity. For example, in the provenance graph from Figure 13, the trigger event INS(G) must precede the derived event DRV(F) in the extracted trace, because a causal edge exists from the former to the latter. Similarly, valid traces are generated for the updates that support the preconditions $c_1, ..., c_p$, which satisfies conditions (b). Condition (c) holds by construction: the original execution trace $\mathcal{E}$ is valid and must include a set of events $\{d_p^1@N_i, ..., d_p^n@N_i\}$ that satisfies condition (c); the PREV-VERTEX call in Figure 12 finds all these events because the call recursively find such events from $\mathcal{E}$ until the interval between the end of $d_j@N_j$ and the start of $d_i@N_i$ is fully exhausted (line 48); therefore, all events in $\{d_p^1@N_i, ..., d_p^n@N_i\}$ will be represented by vertices in the temporal provenance; the extraction algorithm merely reverses this process and reconstructs each of $\{d_p^1@N_i, ..., d_p^n@N_i\}$, while preserving their ordering and timestamps (following the soundness property proved in Theorem 2). Therefore, the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ is valid. For example, consider the DRV(F) event in the provenance from Figure 13: there is a gap of $[2.5s, 3.5s]$ between when its last precondition INS(G) completed and when its own derivation started; in the original execution, node Y must be busy during the gap, because it is work-conserving; in this case, Y was busy with handling RCV(+C); while constructing the vertex for DRV(F), the RAW-QUERY procedure calls PREV-VERTEX($[2.5s, 3.5s]$,Y), which finds the RCV(+C) event from the original trace and added a vertex to G; Algorithm 1 extracts events from G based on topological ordering, therefore, RCV(+C) will present in $\mathcal{A}$, after INS(G) and before DRV(F).
- RCV. We know that, by construction, the RCV vertex has an incoming edge from a SND vertex with the same tuple τ. By the induction hypothesis, Algorithm 1 outputs a valid trace $d_1@N_1, ..., d_j@N_j$ for the subgraph rooted at the SND vertex, where $d_j@N_j$ corresponds to the generation of τ (following the completeness property proved in Theorem 3). Because of the nature of Algorithm 1 (which is based on topological sort), $d_j@N_j$ must be ordered before $d_i@N_i$, which satisfies condition (a) in the definition of validity. A SND vertex have no preconditions, consequently, conditions (b) holds trivially. $d_i@N_i$ and $d_j@N_j$ happened on different nodes, which satisfies condition (c) trivially. Therefore, the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ is valid.
- UDRV/SND. Following the same argument for the DRV case above, the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ is valid.
- INS/DEL. This case cannot occur because INS and DEL have no preconditions, so the tree would have to have a height of one.

□

Theorem 5 *$A(e', e, \mathcal{E})$ is minimal.*

Proof. We prove the minimality property by induction on the syntactic structure of $\mathcal{A}(e', e, \mathcal{E})$: we show that an event $d_i@N_i \in \mathcal{A}(e', e, \mathcal{E})$ cannot be removed because it is necessary for some event $d_j@N_j$ appeared later in the trace. Suppose that $\mathcal{A}(e', e, \mathcal{E}) = d_1@N_1, ..., d_m@N_m$.

Base case. According to the completeness property (Theorem 3), the last event $d_m@N_m$ in $\mathcal{A}(e', e, \mathcal{E})$ generates e. Therefore the base case trivially holds, as the removal of $d_m@N_m$ breaks the completeness property.

Induction case. Suppose the last k events $d_{m-k+1}@N_{m-k+1}$, ..., $d_m@N_m(K >= 1)$ cannot be remove. We show that event $d_{m-k}@N_{m-k}$ cannot be removed as well: According to Algorithm 1, $d_{m-k}@N_{m-k}$ is constructed from a vertex v. v must have an outgoing edge to some other vertex in $G(e', e, \mathcal{E})$. Otherwise, v would not be included in $G(e', e, \mathcal{E})$ which is a subgraph rooted by e. Consider u as the first vertex on the path from v to the root of $G(e', e, \mathcal{E})$. According to Algorithm 1, an event $d_j@N_j$ is constructed from u and its children (if any). Given the edge from v to u, we know that $d_j@N_j$ depends on $d_{m-k}@N_{m-k}$, and that $d_{m-k}@N_{m-k}$ precedes $d_j@N_j$. By applying the induction hypothesis ($d_j@N_j$ cannot be removed from $\mathcal{A}(e', e, \mathcal{E})$), we can conclude that $d_{m-k}@N_{m-k}$ also cannot be removed. □

A.5 Delay annotations

In this section, we show that each vertex is annotated with the delay that it contributed. We first define what it means for a derivation to be directly "delayed" by one of its preconditions (Definition A.5), and then recursively extends this definition to transitive delays (Definition A.5). We continue by discussing several properties of the annotations computed by the algorithm from Figure 6 (Definition A.5, Lemmas 6-9). This allows us to further prove the first theorem which states that the algorithm from Figure 6 labels each vertex with the amount of (direct or transitive) delay that is contributed by the subtree that is rooted at that vertex (Theorem 10).

Definition (Direct delay): *Consider a derivation rule $\alpha : -c_1, c_2, \ldots, c_k$ and an interval $[t_0, t_5]$, such that α begins its derivation at $t_4 < t_5$ and finishes it at time t_5. We say that a precondition c_i* directly delays *the derivation of α during an interval $[t_x, t_y]$, $t_0 \leq t_x$, $t_y \leq t_4$, iff*

- *(a) c_i became true at t_y and remain true until t_4 (and was false before t_y); and*
- *(b) there either was some c_j, $i \neq j$, that delayed the derivation of α during some interval $[x, t_x)$; or there was no such c_j, and $t_x = t_0$.*

For convenience, we say that α itself delays its own derivation during $[t_4, t_5]$. Find the time $t_3 \leq t_4$ such that t_3 is the earliest time when all preconditions were true (and remained true until t4). If a tuple γ resides on the same node as α and the derivation of γ happened during $[t_x, t_y] \subseteq [t_3, t_4]$, we also say that γ directly delays the derivation of α.

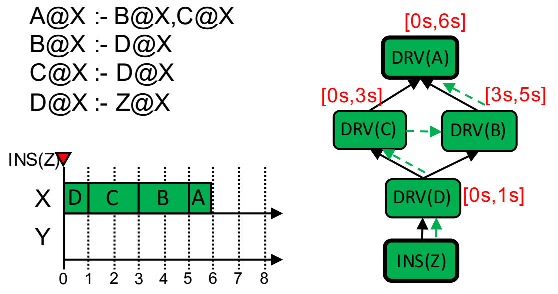


Figure 15: An example scenario, with NDlog rules at the top left, the timing of a concrete execution in the bottom left, and the resulting temporal provenance at the right. The query is T-QUERY(INS(Z), DRV(A)); the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is annotated with its annotation interval (Definition A.5).

Figure 14 contains a brief illustration. c_3 directly delays the derivation of α during the interval $[t_0,t_1]$, because: (a) c_3 became true at t_1 and remained true until t_4; (b) t_0 was the start of the interval in question (the second case of condition (b)). c_2 directly delays the derivation of α during the interval $[t_1,t_2]$, because: (a) c_2 became true at t_2 and remained true until t_4; (b) c_3 delayed the derivation of α during $[t_0,t_1]$ (the first case of condition (b)). Similarly, c_4 delays the derivation of α during the interval $[t_2,t_3]$. γ delays the derivation of α during the interval $[t_3,t_4]$ because, during that interval, all preconditions were true and γ was derived on the same node as α. Finally, α delays the derivation of itself during $[t_4,t_5]$. We can now expand this definition to other derivations:

Definition (Transitive delay): *Consider two derivations $\alpha : -c_1,c_2,\ldots,c_k$ and $\beta : -d_1,d_2,\ldots,d_m$, and suppose β (directly or transitively) delays the derivation of α during an interval $[t_0,t_3]$. Then we say that a tuple d_i* transitively delays *the derivation of α during an interval $[t_1,t_2]$, $t_0 \le t_1$, $t_2 \le t_3$, iff d_i directly delays the derivation of β during $[t_1,t_2]$.*

We can think of the definition of transitive delay as recursively partitioning the interval $[t_0,t_5]$ into smaller intervals that are each associated with some lower-level derivation that caused delay to the top-level derivation of α.

Definition (Annotation interval): *We associate a vertex v in G with an annotation interval $I_v^\alpha = [t_s,t_e]$ for each call of the* ANNOTATE$(v,[t_s,t_e])$ *procedure in the algorithm in Figure 6.*

Figure 15 shows how the algorithm in Figure 6 would have assigned annotation intervals to an example temporal provenance graph. Before presenting the main theorem, we discuss a few properties of annotation intervals.

Lemma 6 *In the algorithm in Figure 6, each invocation of the* ANNOTATE$(v,[t_s,t_e])$ *procedure assigns a set of annotation intervals $\{I_{v^i}^\alpha\}$ to vertices $\{v^i\}$ such that $\bigcap_i I_{v^i}^\alpha = \emptyset$.*

Proof. This holds by construction. When v has no child, $\{I_{v^i}^\alpha\} = \emptyset$ and the condition holds trivially. When v has children: the first WHILE loop in the ANNOTATE procedure subdivides the interval between t_s and the end timestamp of the last precondition into annotation intervals for functional children (in lines 8–16); the second WHILE loop subdivides the interval between the end timestamp of the last precondition and $t_s(v)$ into annotation intervals for sequencing vertices (in lines 17–23); note that if a functional precondition v' is also connected via a sequencing edge to v, it is only handled by the first while loop, because $T = t_{end}(v') = t_{start}(v) = E$ after the first while loop finishes and the second while loop will not execute; therefore, all the generated annotation intervals within an ANNOTATE call are non-overlapping. □

This lemma states that the annotation intervals generated by recursive calls within the same ANNOTATE invocation do not overlap. For example, in Figure 15, the annotation intervals of the DRV(C) and DRV(B) vertices are both assigned by a recursive call on the DRV(A) vertex and thus do not overlap.

Lemma 7 *An annotation interval I_v^α of vertex v always ends at $t_e(v)$, where $t_e(v)$ is when the execution of v finishes or the end timestamp of v (Section 3.3).*

Proof. This holds by construction of the algorithm in Figure 6. In the first WHILE loop in the ANNOTATE procedure (in lines 8–16), the annotation interval associated with v always ends with $t_e(v)$. In the second WHILE loop (in lines 17–23), the annotation interval of the current vertex is $E = t_s(s)$, which is the start timestamp of the previous vertex connected via a sequencing edge; E is also the end timestamp of the current vertex, which follows from the construction of sequencing edges (PREV-VERTEX calls in the algorithm in Figure 12). □

This lemma states that the annotation interval ends when the actual execution finishes. For example, this holds for all annotation intervals in Figure 15.

Lemma 8 *Suppose a vertex v is associated with an annotation interval I_v^α, there exists a chain of ancestor vertices $v \to a_1 \to a_2 \to \ldots \to e$ ($\to$ represents an edge in G, and e is the root of G) such that for each a_i (including e) there exists an annotation interval $I_{a_i}^\alpha$ and $I_v^\alpha \subseteq I_{a_i}^\alpha$.*

Proof. This holds by construction of the algorithm in Figure 6. It follows from the recursive nature of ANNOTATE calls that the annotation interval of each vertex v is a subinterval of one annotation interval of one of its parents: in the first while loop (in lines 8–16), the ANNOTATE is called with an interval of $[T,t_{end}(v')]$, $t_s \le T$ and $t_{end}(v') \le t_{end}(v)$ because v' is a child of v; in the second while loop (in lines 17–23), the ANNOTATE is called with an interval of $[\text{MAX}(T,t_{start}(s)),E]$, $t_s \le T \le \text{MAX}(T,t_{start}(s))$ and $E \le t_{start}(v) \le t_e$ (Lemma 7). We can simply find the specified chain by following such parents recursively until reaching the root vertex e. As the annotation interval is initially I_v^α and is gradually extended as we climb the chain, $I_v^\alpha \subseteq I_{a_i}^\alpha$. □

For instance, consider the provenance from Figure 15, suppose $[0s,1s]$@DRV(D) represents that the DRV(D) vertex is associated with an annotation interval of $[0s,1s]$; the ancestor chain of $[0s,1s]$@DRV(D) would be $[0s,1s]$@DRV(D) $\to$ $[0s,3s]$@DRV(C) $\to$ $[0s,6s]$@DRV(A).

Lemma 9 *Each vertex v in G is associated with at most one annotation interval I_v^α, that is, each vertex v is annotated at most once by the algorithm in Figure 6.*

Proof. We prove by contradiction. Without loss of generality, suppose a vertex v is associated with two annotation intervals I_v^α and $I_v^{\alpha\prime}$. There must exist two corresponding ancestor chains (Lemma 8). We make two observation about the chains: (a) they cannot be identical, because an ancestor chain represents a unique stack of recursive ANNOTATE calls; by the nature of a single-rooted DAG, there cannot exist two stacks of recursive calls that visit the exact same sequence of vertices; (b) the two chains must share a common suffix, this holds trivially because both of the chains end at the root of G. Based on these observations, we can represent the two ancestor chains as $v \to \ldots \to a_i \to a_j \to \ldots$ and $v \to \ldots \to a_i' \to a_j \to \ldots$, where $a_i \neq a_i'$. It follows from Lemma 8 that $I_v^\alpha \subseteq I_{a_i}^\alpha$ and $I_v^{\alpha\prime} \subseteq I_{a_i'}^\alpha$. It follows from Lemma 7 that $[t_e(v)-\epsilon, t_e(v)] \subseteq I_v^\alpha$ and $[t_e(v)-\epsilon, t_e(v)] \subseteq I_v^{\alpha\prime}$, where ϵ is a small value. Therefore, $I_{a_i}^\alpha$ and $I_{a_i'}^\alpha$ overlap. This contradicts with Lemma 6, because $I_{a_i}^\alpha$ and $I_{a_i'}^\alpha$ and divided from $I_{a_j}^\alpha$ in the same ANNOTATE call and cannot overlap. □

These lemmas allow us to formulate our main claim:

Theorem 10 *Suppose* T-QUERY(e', e) *returns* $G(e', e, \mathcal{E})$ *in some execution* $\mathcal{E}$*, and suppose a vertex* v *in* G *is annotated with a value* T *by the algorithm in Figure 6. Then* $T > 0$ *iff* v *directly or transitively delayed the derivation of* e *during an interval* $[t_1, t_2] \subseteq [\text{START}(e'), \text{FINISH}(e)]$ *and* $T = t_2 - t_1$*, and* $T = 0$ *otherwise.*

Proof. We begin by observing that the algorithm in Figure 6 labels each vertex at most once (Lemma 9). Therefore, we only need to show that any single invocation of the ANNOTATE procedure in Figure 6 correctly labels vertices with respect to Definition A.5.

Next, we observe that the ANNOTATE procedure in Figure 6 partitions the interval to explain into annotation intervals of other vertices in exactly the same way that the definition requires. Therefore, I_v^α is exactly the direct or transitive delay of v. We discuss the partition logic of the ANNOTATE procedure in more detail below.

The children of a DRV vertex in the provenance graph would be DRV, INS, or RCV vertices for its preconditions, and lines 8–16 iterate over these vertices in the order of their end times. (The original trace only records the preconditions of an event at the point when its derivation starts; thus, if a precondition had temporarily become true and then false again, the corresponding DRV vertex would not appear as children here.) The loop calls ANNOTATE on vertex v'. with a subinterval of $[t_s, t_e]$ that ends at the point where the precondition is fully derived, and starts either at t_s itself or the end of the previous interval. This subinterval is the annotation interval $I_{v'}^\alpha$ for v' (Definition A.5). Preconditions that were already true at t_s and remained true during the entire interval do not enter the IF block and thus do not generate a recursive call. The first WHILE loop exits with T set to the end time of the last precondition; the WHILE loop that follows it (in lines 17–23) subdivides any non-empty interval between the last satisfied precondition and the start of the derivation of v, just as the definition requires. Again, here each of the divided intervals is the annotation interval $I_{v'}^\alpha$ for another vertex v' (Definition A.5). In particular, noticed that recursive calls happen only for vertices that directly delayed v (and, hence, directly or transitively delayed the vertex in the original query).

Finally, we observe that each vertex v gets labeled with the length of I_v^α in line 7. The labeled value is also the amount of direct or transitive delay that v contributes, because we have proved above that the I_v^α is exactly the direct or transitive delay of v. For example, the intervals annotated beside vertices in Figure 15 are also their direct or transitive delay. □

A.6 Semantics of delay annotations

Although the definitions from Section A.5 do capture the intuitive notion of "delay", we want to reinforce this by formalizing another aspect of this concept: if a vertex v really did delay a derivation by some time T_v, then it should be possible to "speed up" the derivation by T (i.e., cause it to happen T_v units of time sooner) by reducing the duration of v by T_v. In other words, we should be able to construct a valid (hypothetical) trace that differs from the actual trace in that v takes less time, such that the hypothetical trace finishes T_v units of time earlier. (Note that the hypothetical trace might not be "realistic" in a practical sense because some of the events in it may take zero time, and thus be instantaneous; the goal is merely to demonstrate that v is really "responsible for" T_v units of delay.) For example, Figures 16 shows the steps of "speeding up" vertices based on their annotations ((a) $\rightarrow$ ... $\rightarrow$ (g)). This procedure shortens the overall (hypothetical) execution at each step and eventually eliminates any delay.

For this discussion, the annotation intervals that are computed by the algorithm in Figure 6 are not directly useful, because they describe the delay that was caused by an entire subgraph of the provenance. Hence, we first describe how we have derived a more fine-grain form of annotation, which describes the delay that is contributed by a vertex itself (Definition A.6). We then discuss two properties of the derived annotation (Lemmas 11-12). We continue by defining the procedure of "speeding up" an execution based on derived annotations (Definitions A.6-A.6). We conclude by presenting the main theorem which states that if there is a vertex v in a temporal provenance tree with a (derived) annotation of T, then it is possible to construct another valid (but hypothetical) execution in which v's finished time is reduced by T and in which the derivation finishes T units of time earlier (Definition A.6 and Theorem 13).

Definition (Speedup interval): *The speed interval* $I_v^\delta = [t_s, t_e]$ *of vertex* v *is the difference between* v*'s annotation interval, as computed by the algorithm from Figure 6, and the union of the annotation intervals of the vertices directly annotated by* v *(via the recursive calls in the* ANNOTATE *procedure).*

Intuitively, I_v^δ represents the interval during which the execution of v itself delays e. For example, in the provenance from Figure 16(a), red intervals represent annotation intervals and blue intervals represent speedup intervals. The speed up interval of DRV(A) is $[6s, 7s]$, which is the difference from its annotation interval ($[0s, 7s]$), and the union of the annotations intervals of DRV(B) and RCV(+E) ($[0s, 4s] \bigcup [4s, 6s]$). Speed up intervals have the following two properties:

Lemma 11 *The speedup interval* I_v^δ *of vertex* v *always ends at* $t_e(v)$*, where* $t_e(v)$ *is when the execution of* v *finishes or the end timestamp of* v *(Section 3.3).*

Proof. The annotation interval of v always ends at $t_e(v)$ (Lemma 7). We prove that that I_v^δ ends when I_v^α ends. If v has no child, $I_v^\delta = I_v^\alpha$; if v has children, the two WHILE loops in the algorithm in Figure 6 distribute the interval between t_s and $t_s(v)$ to other vertices via recursive ANNOTATE calls, and the remaining interval in $[t_s, t_e]$ is the speedup interval; in either case, I_v^δ ends when I_v^α ends, and therefore, I_v^δ ends at $t_e(v)$. □

Lemma 12 *Given the temporal provenance of* T-QUERY(e', e)*, consider the set of all speedup intervals* $\{I_{vi}^\delta\}$*: (a)* $\bigcap_i I_{vi}^\delta = \emptyset$*; (b)* $\bigcup_i I_{vi}^\delta = I_e^\alpha$.

Proof. In a temporal provenance graph, vertices with annotation intervals form a tree, because vertex v is annotated at most once (Lemma 9) by a parent of v. Consequently, vertices with speedup intervals form a tree (Definition A.6). We prove by structural induction on the height of the tree.

Base case: The height of the tree is one. Denote the root vertex as v. The set of speedup intervals has one element ($\{I_{vi}^\delta\} = I_v^\delta$), condition (a) holds. $I_v^\delta = I_v^\alpha$ because v has no child (Definition A.6), condition (b) holds.

Induction case: Suppose the conditions hold for trees (of vertices with annotation or speedup intervals) with height less than k ($k \geq 1$). Consider a tree with depth $k + 1$, rooted at vertex v. Without loss of generality, denote $T(v_1)$ and $T(v_2)$ as subtrees of v that have annotation intervals. Note that the speedup intervals of vertices in $T(v)$ must be a subinterval of the annotation interval of v, because: the speedup interval is simply the difference of the annotation interval of v and the annotation intervals of the children of v (Definition A.6); the annotation interval of v is a subinterval of the annotation interval of its parent in the tree (Lemma 8).

It follows from Definition A.6 that the speedup interval of v and that of $T(v_1)$ (or $T(v_2)$) cannot overlap. It follows from Lemma 6 that the speedup intervals of $T(v_1)$ and $T(v_2)$ cannot overlap. It follow from the induction hypothesis that the speedup intervals within $T(v_1)$ (or $T(v_2)$) cannot overlap. Therefore, condition (a) holds. It follows from Definition A.6 that condition (b) holds. □

Intuitively, the above two lemmas states that, given a temporal provenance that explains T-QUERY(e', e), the overall delayed interval – that is, $[t_s(e'), t_e(e)]$ – can be subdivided into a sequence of all speedup intervals $\{I_{v_i}^\delta\}$. In the example provenance from Figure 16(a), such a sequence of speedup intervals is $[0s, 2s]$@DRV(C), $[2s, 4s]$@DRV(B), $[4s, 5.5s]$@DRV(E), $[5.5s, 6s]$@RCV(+E), and $[6s, 7s]$@DRV(A).

Definition (Terminal event): *A vertex* v *is a terminal event if any of the following conditions holds:*

- *(a) if* v *is annotated,* v *ends at* $t_e(e)$*, and* $I_v^\delta = \emptyset$ *(a vertex is annoated if it is associated with an annotation interval in the original* G*);*
- *(b) if* v *is not annotated, on any path in* G *from* v *to* e*, select* u *as the first annotated vertex,* $I_u^\alpha = \emptyset$.

A@X :- B@X,E@Y
B@X :- C@X
E@Y :- C@X
C@X :- Z@X

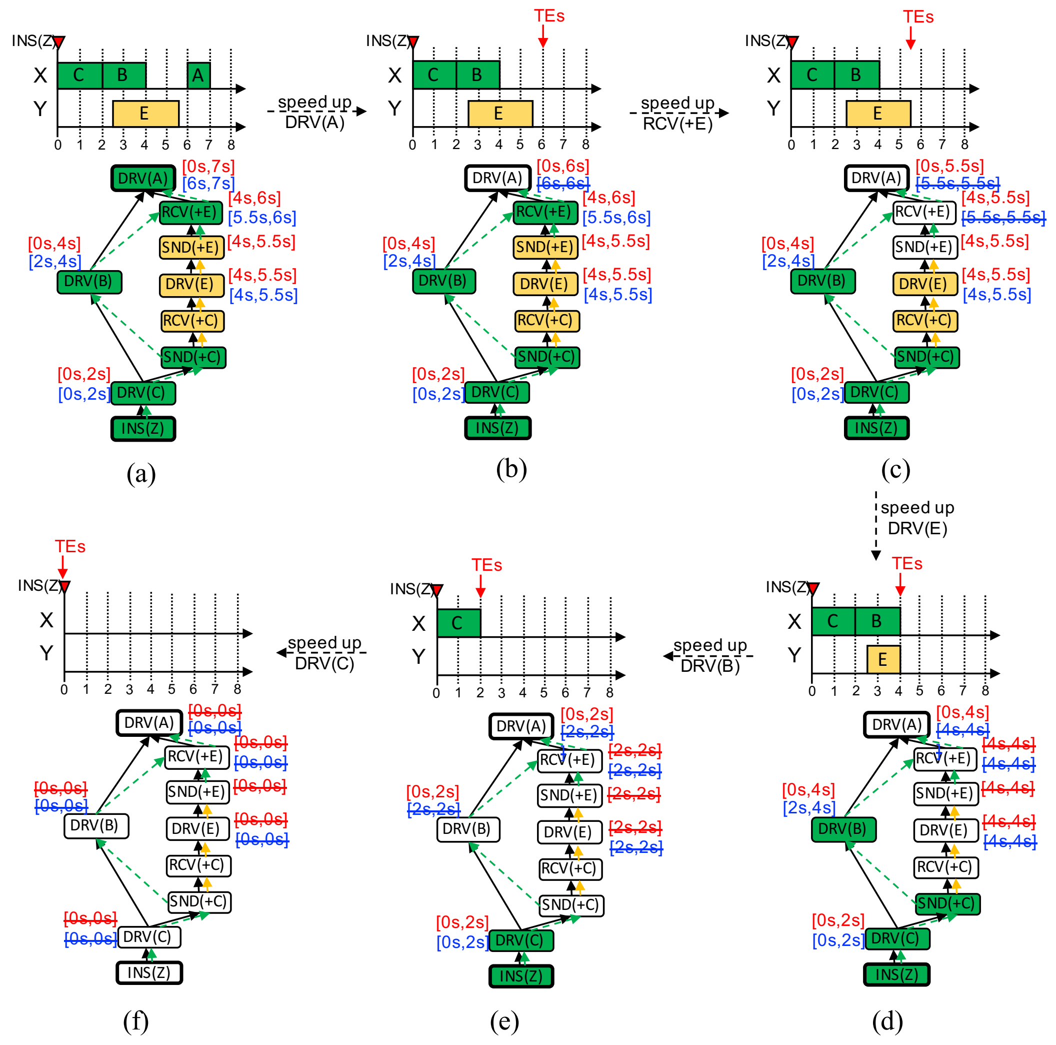


Figure 16: An example of "speeding up" temporal provenance using a series of transformations (Definition A.6). The NDlog rules are at the top left. Each sub-figure shows a step of the transformation ((a) $\rightarrow$ ... $\rightarrow$ (g)). In each sub-figure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all sub-figures; the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition A.5) and speed up interval (blue, Definition A.6). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertices are terminal events.

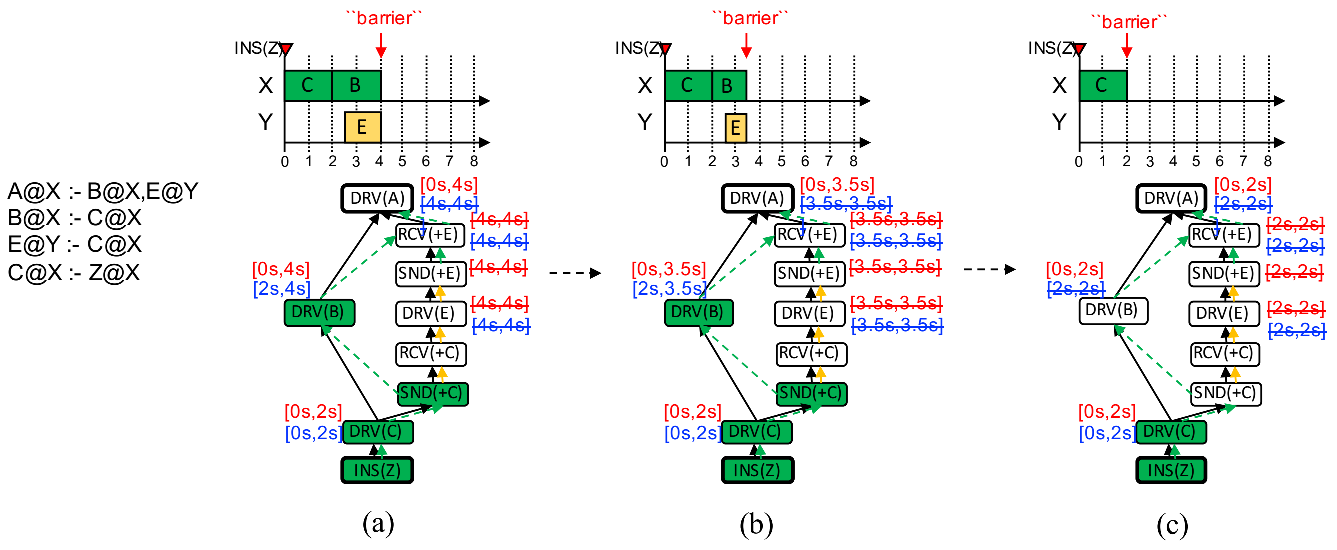


Figure 17: An example of "speeding up" temporal provenance using an annotated vertex DRV(B) (Definition A.6). The NDlog rules are at the left. In each sub-figure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all sub-figures; the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition A.5) and speed up interval (blue, Definition A.6). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertices are terminal events.

Intuitively, terminal events represent executions that no longer contribute any delay (in a hypothetical execution). Condition (a) describes an event that finishes at the end of the entire execution and that no longer contributes any delay. For example, $\mathrm{RCV}(+\mathrm{E})$ in Figure 16(c) is a terminal event: it was annotated in the original provenance (Figure 16(a)); it ends at $t = 5.5s$, which is the end timestamp of $\mathrm{DRV}(\mathrm{A})$; and its speedup interval is empty. Condition (b) describes an event that only belongs to subgraphs that no longer contribute any delay. For example, $\mathrm{RCV}(\mathrm{C})$ in Figure 16(e) is a terminal event: it was not annotated in the original provenance (Figure 16(a)); on its (only) path to $\mathrm{DRV}(\mathrm{A})$, the first annotated vertex is $\mathrm{DRV}(\mathrm{E})$, whose annotation interval is already empty ($[4s, 4s]$). Next, we describe steps to transform the original execution to hypothetical executions.

Definition (Speed up): *Given a vertex v in $G(e', e, \mathcal{E})$, where $t_e(v) = t_e(e)$ and $I^{\delta}_{v} > 0$, v speeds up G by I^{δ}_{v} using the following procedure. Consider a "barrier" t_b that moves on the timeline; it starts from the right boundary of I^{δ}_{v} and moves leftwards (and thus t_b becomes smaller); it stops when it reaches the left boundary of I^{δ}_{v}. For ease of exposition, we say that the "barrier" pushes a timestamp t when we set t to $\mathrm{MIN}(t, t_b)$. During its move, if the "barrier" encounters a vertex v_i that is either v or a terminal event, it transforms v_i by pushing these timestamps: (a) the starting timestamp (or the ending timestamp) of v_i, (b) the left boundary (or the right boundary) of $I^{\alpha}_{v_i}$ (if any); and (c) the left boundary (or the right boundary) of $I^{\delta}_{v_i}$ (if any).*

Intuitively, the "speed up" operation represents a transformation step that essentially "squeezes" a set of vertices to the left. Note that, while v speeds up G, only v itself and terminal events – vertices that no longer contribute any delay – are pushed leftwards. For example, Figure 17 shows the process of speeding up the provenance using $\mathrm{DRV}(\mathrm{B})$: the "barrier" starts from the right boundary of $I^{\delta}_{\mathrm{DRV(A)}}$ (Figure 17(a)); while it moves, the "barrier" pushes $\mathrm{DRV}(\mathrm{B})$ as well as terminal events $\mathrm{DRV}(\mathrm{E})$ and $\mathrm{RCV}(\mathrm{C})$ leftwards (Figure 17(b) shows the snapshot of $t_b = 3.5s$); the "barrier" stops at the left boundary of $I^{\delta}_{\mathrm{DRV(A)}}$ (Figure 17(c)).

Figure 16 shows the process of speeding up an entire provenance graph until it becomes instantaneous. Next, we briefly show the effect of each "speed up" operation:

- (a) $\rightarrow$ (b), $\mathrm{DRV}(\mathrm{A})$ speeds up G by $I^{\delta}_{\mathrm{DRV(A)}} = [6s, 7s]$: the execution is shortened to $[0s, 6s]$, $\mathrm{DRV}(\mathrm{A})$ becomes a terminal event;
- (b) $\rightarrow$ (c), $\mathrm{RCV}(+\mathrm{E})$ speeds up G by $I^{\delta}_{\mathrm{RCV(+E)}} = [5.5s, 6s]$: the execution is shortened to $[0s, 5.5s]$, $\mathrm{RCV}(+\mathrm{E})$ and $\mathrm{SND}(+\mathrm{E})$ become terminal events;
- (c) $\rightarrow$ (d), $\mathrm{DRV}(\mathrm{E})$ speeds up G by $I^{\delta}_{\mathrm{DRV(E)}} = [4s, 5.5s]$: the execution is shortened to $[0s, 4s]$, $\mathrm{DRV}(\mathrm{E})$ and $\mathrm{RCV}(+\mathrm{C})$ become terminal events;
- (d) $\rightarrow$ (e), $\mathrm{DRV}(\mathrm{B})$ speeds up G by $I^{\delta}_{\mathrm{DRV(B)}} = [2, 4s]$: the execution trace is shortened to $[0s, 2s]$, $\mathrm{DRV}(\mathrm{B})$ becomes terminal events;
- (e) $\rightarrow$ (f), $\mathrm{DRV}(\mathrm{C})$ speeds up G by $I^{\delta}_{\mathrm{DRV(C)}} = [0, 2s]$: the execution trace is shortened to $[0s, 0s]$, all events are now terminal events.

Definition (Well-annotated): *Consider an annotated temporal provenance graph $G(e', e, \mathcal{E})$. G is* well-annotated *iff either (a) $t_s(e') = t_e(e)$, that is, the entire execution is instantaneous; (b) we can transform G into another valid and well-annotated temporal provenance graph G' by locating an unique vertex v, where $t_e(v) = t_e(e)$ and $I^{\delta}_{v} > 0$, and speeding up G by v (Definition A.6).*

Theorem 13 *Temporal provenance is well-annotated.*

Proof. Consider the speedup intervals $\{I^{\delta}_{v_i}\}$ of G. It follows from Lemma 12 that $\{I^{\delta}_{v_i}\}$ do not overlap and unions to $[t_s(e'), t_e(e)]$. Therefore, we can sort intervals in $\{I^{\delta}_{v_i}\}$ by descending (ending) timestamp. At the ith step, we speed up G by v_i. We need to prove that: (a) each "speed up" operation pushes the timestamps of all events that ends during $I^{\delta}_{v_i}$; (b)

```
A@X :- B@X,E@Y
B@X :- C@X
E@Y :- C@X
C@X :- Z@X
```

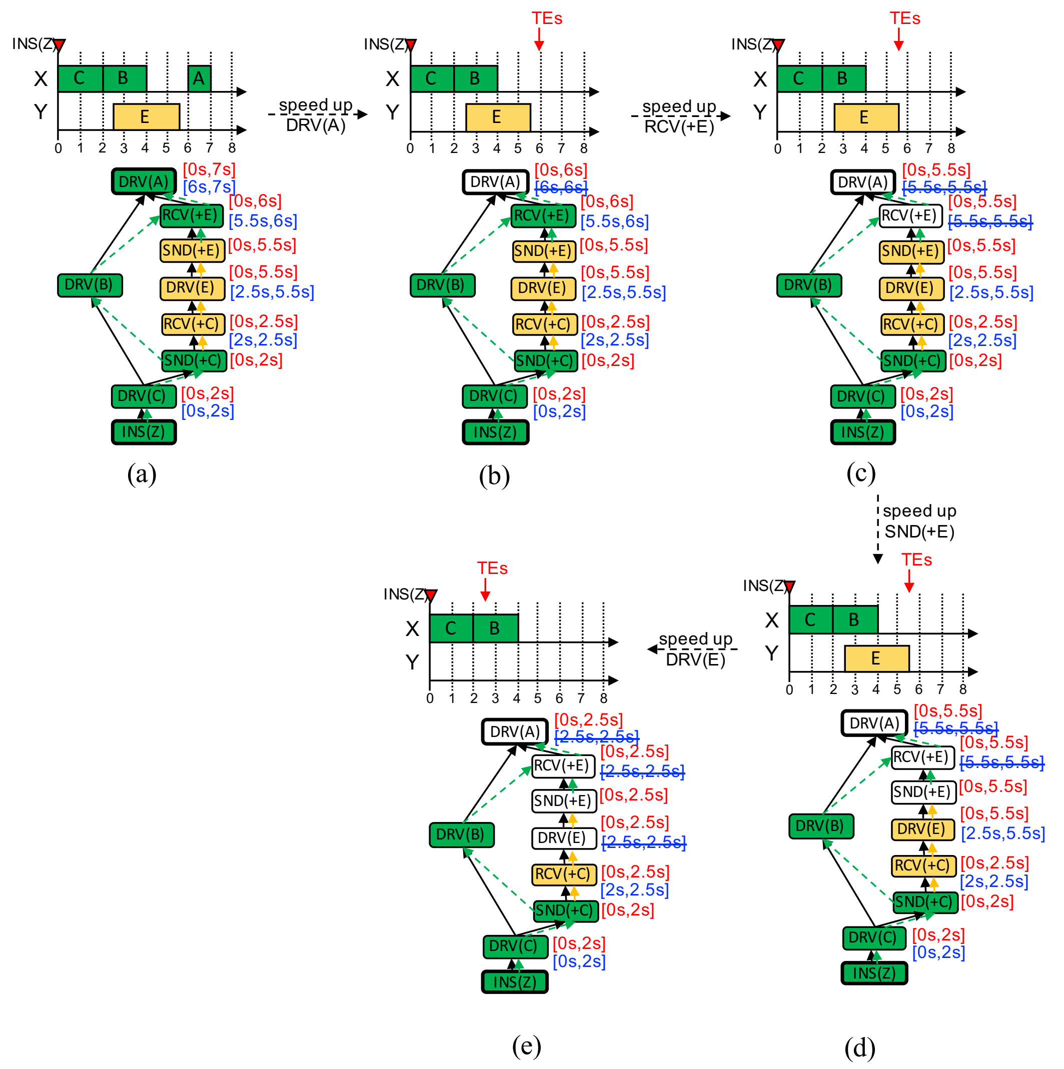


Figure 18: An example of "poorly annotated" temporal provenance. The NDlog rules are at the top left. Each sub-figure shows a step of the transformation ((a) → ... → (g)). In each sub-figure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all sub-figures; the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition A.5) and speed up interval (blue, Definition A.6). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertices are terminal events.

the length of the execution $[t_s(e'), t_e(e)]$ is reduced by the length of $I^{\delta}_{v_i}$; (c) the temporal provenance remains valid.

To prove condition (a), given any vertex v'_i that ends during $I^{\delta}_{v_i}$, we perform a case analysis of v'_i:

- $v'_i = v_i$: the timestamps of $v'_i = v_i$ is pushed, by the construction of Definition A.6.
- $v'_i \neq v_i$ and v'_i is annotated in the original provenance: v'_i must be a terminal event, and therefore, its timestamps is pushed. Because $I^{\delta}_{v'_i}$ ends when v'_i ends (Lemma 11); consequently, $I^{\delta}_{v'_i}$ must end during $I^{\delta}_{v_i}$; if $I^{\delta}_{v'_i}$ is not empty, $I^{\delta}_{v'_i}$ will overlap with $I^{\delta}_{v_i}$, which contradicts with Lemma 12.
- $v'_i \neq v_i$ and v'_i is not annotated in the original provenance: v'_i must be a terminal event, and therefore, its timestamps is pushed. Because, given any path from v'_i to e, consider the first annotated ancestor u and its child on the path w; if we assume that I^{α}_{u} is not empty when the "barrier" reaches the end of w, then I^{α}_{u} must start before the end of w; by construction of the algorithm from Figure 6, w must be annotated by u, which contradicts the fact that w is not annotated.

Condition (b) follows directly from the statement above: the execution is shortened by the length of $I^{\delta}_{v_i}$, because all events that end during $I^{\delta}_{v_i}$ are pushed leftwards until the left boundary of $I^{\delta}_{v_i}$.

Condition (c) holds because the "speed up" operation does not invert causality: if an event a caused another event b, it does not alter the ordering of a and b; nor does it delete any event. $\square$

Note that Definition A.6 weeds out some annotation approaches. For example, Figure 18 shows how a straw-man approach that associates the entire delay with the last precondition would have annotate the same provenance graph in Figure 16. The result is not well annotated: while DRV(E) speeds up G ((d) $\rightarrow$ (e)), another vertex DRV(B) becomes the bottleneck; however, DRV(B) cannot be pushed leftwards, because it is not a terminal event, that is, it has not been "sped up".

Theorem 13 suggests that the annotations on temporal provenance do correspond to the "potential for speedup" that one may intuitively associate with the concept of delay. This is useful, because, while the temporal provenance maybe gigantic and complex, operators can focus on vertices with annotations and gain a comprehensive understanding of the end-to-end delay, including potential operations to speed up.