Debugging Distributed Systems with Why-Across-Time Provenance

Technical Report. October 14, 2018.

Michael Whittaker
UC Berkeley
mjwhittaker@berkeley.edu

Cristina Teodoropol
UC Berkeley
ct@berkeley.edu

Peter Alvaro
UC Santa Cruz
palvaro@ucsc.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

ABSTRACT

Systematically reasoning about the fine-grained causes of events in a real-world distributed system is challenging. Causality, from the distributed systems literature, can be used to compute the causal history of an arbitrary event in a distributed system, but the event's causal history is an over-approximation of the true causes. Data provenance, from the database literature, precisely describes why a particular tuple appears in the output of a relational query, but data provenance is limited to the domain of static relational databases. In this paper, we present ***wat-provenance***: a novel form of provenance that provides the benefits of causality and data provenance. Given an arbitrary state machine, wat-provenance describes why the state machine produces a particular output when given a particular input. This enables system developers to reason about the causes of events in real-world distributed systems. We observe that automatically extracting the wat-provenance of a state machine is often infeasible. Fortunately, many distributed systems components have simple interfaces from which a developer can directly specify wat-provenance using a technique we call ***wat-provenance specifications***. Leveraging the theoretical foundations of wat-provenance, we implement a prototype distributed debugging framework called Watermelon.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267839

This version of the paper is a technical report that includes some additional information that is not present in our main publication. Additional text (like this) is annotated with a red bar along its left side.

CCS CONCEPTS

• **Information systems → Data provenance**; • **Theory of computation → Data provenance**; Formal languages and automata theory; • **Software and its engineering →** *Software testing and debugging*;

KEYWORDS

data provenance, state machines, distributed systems

ACM Reference Format:

Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M. Hellerstein. 2018. Debugging Distributed Systems with Why-Across-Time Provenance. In *Proceedings of ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 15 pages.
https://doi.org/10.1145/3267809.3267839

1 INTRODUCTION

Debugging distributed systems is hard. Traditional debugging techniques are poorly suited to distributed systems in which bugs arise across multiple nodes connected by an unreliable network that can drop, duplicate, and reorder messages. Distributed debugging tools exist but are in their infancy. They help tame *some* of the complexities of distributed debugging, but have limited applicability to real-world distributed systems that are made up of a large number of complex components. Consequently, developers perform ad-hoc root cause analysis to find the source of a bug, stitching together the logs of multiple concurrently executing nodes.

Worse, existing formalisms are also inadequate to reason about systematically debugging real-world distributed systems. For example, consider ***causality*** [25]. Causality is a

general-purpose formalism that specifies the causal history of a particular event in an arbitrary distributed system. However, causality is too general-purpose, as it fails to incorporate any semantics of the underlying distributed system [5]. As a consequence, the causal history of an event is an overapproximation of the cause of the event. It includes all the events that *might* have caused a particular event instead of the events that *actually do* cause it.

Alternatively, consider data provenance in the form of ***why-provenance*** [8, 11]. Given a relational database, a query issued against the database, and a tuple in the output of the query, why-provenance explains why the output tuple was produced. That is, why-provenance produces the input tuples that, if passed through the relational operators of the query, would produce the output tuple in question. In contrast to causality, data provenance heavily incorporates the semantics of relational databases and queries to describe *precisely* the cause of a particular output. However, why-provenance makes the critical assumption that the underlying relational database is static. It cannot handle the time-varying nature of stateful distributed systems. Moreover, data provenance is limited to the domain of relational data and cannot easily be applied to other system components (e.g., load balancers, file systems, coordination services, etc.).

In short, causality lacks a notion of data dependence, and data provenance lacks a notion of time. In this paper, we present ***wat-provenance*** (why-across-time provenance): a novel form of data provenance that unifies ideas from the two. Wat-provenance generalizes why-provenance from the domain of relational queries issued against a static database to the domain of arbitrary time-varying state machines in a distributed system. More specifically, given a deterministic state machine, the state machine's sequence of inputs, and a particular input to the state machine, wat-provenance formalizes *why* the state machine produces the output that it does. This description is the set of subsequences of the input trace that are necessary and sufficient to generate the output in question. Borrowing from causality, wat-provenance can be applied to time-varying state machines. Borrowing from why-provenance, wat-provenance incorporates state machine semantics to avoid overapproximating provenance.

After we define wat-provenance, we turn to the matter of computing it. We observe that automatically extracting the wat-provenance of a state machine is often infeasible. Computing the wat-provenance of a state machine is tantamount to inferring the state machine's data dependencies using a complex code analysis of the state machine's source code. This source code can be both large and complex which makes this code analysis intractable. Though automatically extracting the wat-provenance of an *arbitrary* state machine is difficult, many distributed systems components are designed with simple and minimalistic APIs. We can take advantage of this observation and sidestep the complexity of *extracting* the wat-provenance from the *implementation* of a state machine and instead *specify* the wat-provenance from the *interface* of a state machine. To this end, we propose ***wat-provenance specifications***: functions that directly encode the wat-provenance of a state machine using its interface instead of its implementation. We describe the provenance specifications of a number of widely used distributed systems components (e.g., Redis, Amazon S3, HDFS, Zookeeper) and find that in practice, they are often straightforward to implement.

Next, we present Watermelon: a prototype distributed debugging framework that leverages the formal foundations of wat-provenance and wat-provenance specifications. Watermelon includes a mechanism for developers to write wat-provenance specifications that are executed against the input traces of the components in a distributed system. We use Watermelon to measure the complexity of writing wat-provenance specifications and also compare Watermelon to existing distributed debugging techniques.

This paper presents the following contributions:

- We define wat-provenance: a formalism that extends notions of data provenance to the realm of state machines in a distributed system.
- We present wat-provenance specifications: a mechanism to compute the wat-provenance of distributed system components. We also describe a set of wat-provenance specifications for a number of widely used distributed systems components, illustrating that wat-provenance specifications can be straightforward to write.
- We implement a prototype distributed debugger called Watermelon that leverages the theoretical foundations of wat-provenance. We demonstrate that Watermelon makes it easier to identify the precise causes of events in a distributed system compared to existing debugging techniques.

2 BACKGROUND

2.1 Causality

As described by Lamport in [25], time is fundamental to our understanding of how events are ordered. It is clear that if an event occurs at 6:42, then it *happens before* another event that occurs at 6:45. Unfortunately, accurately measuring time in a distributed system is infeasible [32, 35, 36]. Clocks on different servers within a distributed system drift apart, so servers cannot agree on a single global notion of time, and thus they cannot agree on a single global *total order* of events that respects the real time ordering of events. However, as Lamport showed in [25], it *is* possible for servers to agree on a global *partial order* of events that respects the global

passage of time. This partial ordering of events also dictates which events can causally affect each other.

To make this partial ordering and notion of causality precise, we consider a set of single-threaded servers that communicate over the network. Every server a serially executes a sequence of events $a_1, a_2, a_3, \ldots$ where each event a_i represents the action of server a either (1) performing local computation, (2) sending a message to another server, or (3) receiving a message from another server.

The ***happens before*** relation $\to$ on events is the smallest transitive relation such that (1) if a_i, a_j are two distinct events within the same process, then a_i happens before a_j, and (2) if a_i and b_j are the sending and receiving of a message respectively, then a_i happens before b_j. The happens before relation is a partial order that formalizes our intuitive notion of which events can causally affect each other. An event a_i can only be caused by an event b_j that happens before it. The set $\{b_j \mid b_j \to a_i\}$ is called the ***causal history*** of a_i.

2.2 Data Provenance

Given a relational database instance I, a relational algebra query Q, and a tuple t in the output of the query, it is natural to ask why t appears in the output. For example, consider the relational database instance given in Figure 1 that describes users and friends in a social media application. And, consider the relational algebra query

$$Q \stackrel{\text{def}}{=} \pi_{\texttt{name}}(\sigma_{\texttt{friend1=ecodd}}(\texttt{Users} \bowtie_{\texttt{username=friend2}} \texttt{Friends}))$$

that returns the name of all of Edgar Codd's friends. Evaluating Q on I produces the tuple t = (Michael Jordan).

Users

username	name
ecodd	Edgar Codd
jumpman	Michael Jordan
mlpro	Michael Jordan

Friends

friend1	friend2
ecodd	jumpman
ecodd	mlpro

Figure 1: An example database instance

Intuitively, t = (Michael Jordan) is present in the output $Q(I)$ for two reasons: (1) the existence of the (ecodd, jumpman) and (jumpman, Michael Jordan) tuples and (2) the existence of the (ecodd, mlpro) and (mlpro, Michael Jordan) tuples. ***Why-provenance*** [8, 11] formalizes this intuition. The why-provenance[1] of a tuple t with respect to query Q and database instance I, denoted $\mathsf{Why}(Q, I, t)$, is a set $J_1, \ldots, J_n$ of subinstances of I where each subinstance $J_i \subseteq I$ suffices to produce t (i.e. $t \in Q(J_i)$). These subinstances are called ***witnesses of*** t, and a witness J_i is called a ***minimal witness of*** t if no proper subinstance of J_i is also a witness of t. The ***minimal why-provenance*** of t, denoted $\mathsf{MWhy}(Q, I, t)$, is the set of the minimal witnesses in $\mathsf{Why}(Q, I, t)$. It can be shown that $\mathsf{MWhy}(Q, I, t)$ is exactly the set of minimal witnesses of t [11].

Returning to our example above, the why-provenance of the (Michael Jordan) tuple is the set $\{J_1, J_2\}$ where

$$J_1 = \{(\text{ecodd}, \text{jumpman}), (\text{jumpman}, \text{Michael Jordan})\}$$
$$J_2 = \{(\text{ecodd}, \text{mlpro}), (\text{mlpro}, \text{Michael Jordan})\}$$

J_1 and J_2 are minimal witnesses, so the why-provenance and minimal why-provenance of t = (Michael Jordan) are the same.

2.3 State Machines

It is common to model servers—like key-value stores or relational databases—as deterministic state machines that repeatedly receive requests, update their state, and send replies [26, 37]. More precisely, a ***deterministic state machine*** $M = (S, s_0, \Sigma, \Lambda, \delta, \epsilon)$ consists of a (potentially infinite) set S of states, a start state $s_0 \in S$, an input alphabet Σ, an output alphabet Λ, a transition function $\delta : S \times \Sigma \to S$, and an output function $\epsilon : S \times \Sigma \to \Lambda$. A state machine M begins in state s_0 and repeatedly receives inputs $a \in \Sigma$. Upon receiving an input a, M transitions from state s to state $\delta(s, a)$ and outputs $\epsilon(s, a)$.

In our work, we need to reason about specific sub-inputs to a state machine, in the spirit of why-provenance, so we introduce some notation here. We refer to an ordered sequence of inputs received by a state machine as a ***trace*** $T = a_1 a_2 \ldots a_n \in \Sigma^*$. A ***subtrace*** T' of T is a subsequence $T' = a_{i_1} a_{i_2} \ldots a_{i_m}$ where $i_1, i_2, \ldots, i_m$ are distinct elements of $1, 2, \ldots, n$ in ascending order. Note that a subsequence does *not* have to be contiguous. For example, $T' = a_1 a_3$ is a subtrace of $T = a_1 a_2 a_3 a_4$. If T' is a subtrace of T, then a ***supertrace of*** T' ***in*** T is a subtrace of T that contains every element of T'. If T_1 and T_2 are two traces, we write the concatenation of T_1 and T_2 as $T_1 T_2$. Similarly, if $T \in \Sigma^*$ is a trace, and $a \in \Sigma$ is an input, we let Ta denote the trace produced by appending a to the end of T. An illustrative example of the definition of subtraces and supertraces is given in Figure 2.

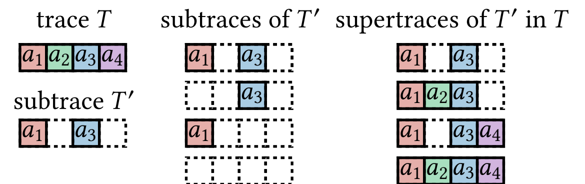

Figure 2: Subtraces and supertraces

δ takes in a state $s \in \Sigma$ and a single input $a \in \Sigma$. It is convenient to extend δ to a function $\delta^* : S \times \Sigma^* \to S$ that

[1] For a formal definition of why-provenance, we refer the reader to [11]. For our purposes, an informal understanding of why-provenance is sufficent.

takes in a state s and a trace $T \in \Sigma^*$ and outputs the state reached after sequentially executing the inputs in T starting in state s. Similarly, we can extend ϵ to a function $\epsilon : S \times \Sigma^+ \to \Lambda$ which takes in a state s and a non-empty trace $T = a_1 \dots a_n \in \Sigma^+$ and returns $\epsilon(\delta^*(s, a_1 a_2 \dots a_{n-1}), a_n)$: the final output produced from sequentially executing every input in T starting in state s.

3 WAT-PROVENANCE

3.1 A Motivating Example

It is difficult to reason precisely about the causes of events in a heterogeneous distributed system. To best understand why, let's look at a simple example. Consider the implementation of a Facebook-like social media application called ZardozBook, illustrated in Figure 3. ZardozBook users post status updates, and these updates are only viewable by their friends on the site. Users send requests to a load balancer that forwards the requests to one of three weakly consistent Redis-backed application servers: s_1, s_2, and s_3. These application servers store a cache of ZardozBook's data in Redis and periodically synchronize their caches with a centralized Postgres database.

Consider a scenario in which Ava, a ZardozBook user, has a falling out with Bob, a friend of hers on the site. Ava unfriends Bob and then posts the status "Bob is a big jerk!", thinking that Bob will not see the status because he is no longer her ZardozBook friend. Unfortunately, Bob later logs in to ZardozBook and sees Ava's mean comment!

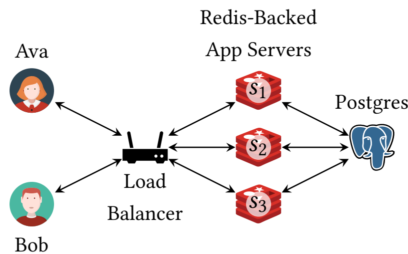

Figure 3: Social media application

Why did this happen? Here's an informal account. Ava's request to unfriend Bob was forwarded to application server s_1 by the load balancer. Then, Ava's request to post the mean comment about Bob was forwarded to s_2. s_2 then pushed the comment to the Postgres repository. s_3 then issued a SQL query to the Postgres repository, pulling the latest data into its Redis cache. In doing so, it pulled in Ava's mean comment. Finally, when Bob logged in, his request was forwarded to s_3 which returned the mean comment.

We argue that existing formalisms are inadequate for discovering this sequence of events. One possibility is to use causality, as described in Section 2. We could instrument our distributed system to record the causal history of every event that takes place in the system. Then, we could examine the causal history of Bob's request in an attempt to diagnose why Bob was seeing Ava's mean comment. Unfortunately, this is not helpful. The causal history of Bob's request includes *every* event that causally precedes it, whether or not the event is relevant. For example, the causal history of Bob's request would include every message received by server s_3 prior to Bob sending his request, even those that do not involve Ava and Bob. The problem is that causality is too coarse-grained. It fails to incorporate any notion of a system's semantics as a means to filter out irrelevant messages. Instead, it returns an overapproximation of all the events that *might* cause an event instead of the events that *do* cause an event.

This might prompt us to try and apply ideas from data provenance. Unlike causality, why-provenance does incorporate system semantics to return the causes of a particular output of a query. As discussed above, why-provenance has two flaws. First is why-provenance's restriction to relational queries: while we might be able to use why-provenance to debug s_3's SQL query that was sent to the Postgres database, that is only one small piece of the puzzle. Understanding why Bob saw Ava's mean comment requires us to reason about messages that travel through our application servers, our Redis servers, and our load balancer. But, these are not relational databases, so we cannot apply why-provenance to them. Second and more fundamental is why-provenance's inability to reflect any notion of state change over time. In particular, the real "why" question we want to answer here is "why was the provenance of Bob's query unaffected by Ava's unfriend request?" The why-provenance of Bob's query has no answer to this question; it knows nothing about updates or the order in which they happen.

3.2 Defining Wat-Provenance

In isolation, causality and data provenance are both insufficient to diagnose why Bob saw Ava's mean comment. Understanding the root cause of this anomalous behavior requires us to reason about the ordering of events within the network (as with causality) *and* the precise data dependencies between different requests (as with data provenance). ***Wat-provenance*** unifies the benefits of causality and data provenance. Borrowing from causality, wat-provenance is a general-purpose mechanism that can be applied to arbitrary state machines. Borrowing from why-provenance, wat-provenance incorporates system semantics to produce the causes of an event, rather than a conservative overapproximation.

More formally, we consider a deterministic state machine $M = (S, s_0, \Sigma, \Lambda, \delta, \epsilon)$, a trace $T \in \Sigma^*$, and a particular input

$i \in \Sigma$. The state machine begins in state s_0 and executes T, transitioning to state $s_T = \delta^*(s_0, T)$. It then executes input i producing output $o = \epsilon^*(s_0, Ti) = \epsilon(s_T, i)$. Wat-provenance aims to formalize an intuitive notion of *why* the state machine M produces output o when given input i. We build this intuitive notion by way of a sequence of examples, ultimately leading to the definition of wat-provenance.

Example 3.1. Consider a key-value server state machine M with an input alphabet that consists of sets and gets to integer-valued variables that are initially 0. Consider the trace

$$T = \text{set}(x, 1); \text{set}(y, 2)$$

that consists of two requests: one that sets the value of x to 1 and one that sets the value of y to 2. Consider request $i = \text{get}(x)$ that requests the value of x. When M processes trace T and then input i, it expectedly outputs 1 (i.e. $o = \epsilon^*(s_0, Ti) = 1$).

Why did M output 1? In this trivial example, M returned 1 as the value of x because the first request in T set x's value to 1. More formally, the subtrace $T' = \text{set}(x, 1)$ of T suffices to generate the output of 1 (i.e. $\epsilon^*(s_0, T'i) = o$). The lesson here is that **the cause of an output o is a subtrace of the input that is sufficient to generate** o. We call such a subtrace a ***witness*** of o.

However, the entire trace T is also a witness. That is, T also suffices to generate an output of 1. But, the $\text{set}(y, 2)$ request is not relevant to our $\text{get}(x)$ request, so we shouldn't include it as a cause of our output. Thus, we revise our earlier observation; **the cause of an output o is a subtrace of the input that is sufficient to generate o and is also in some sense minimal**. We will define minimality more carefully below.

Example 3.2. Consider a state machine M that stores a set of boolean-valued variables that are initially false. Users can set variables to true or false and can request that M evaluate a formula over these variables. For example, consider the trace

$$T = \text{set}(a); \text{set}(b); \text{set}(c); \text{set}(d)$$

that sets variables a, b, c, d to true. Further, consider the input $i = \text{eval}((a \wedge d) \vee (b \wedge c))$ that requests M evaluate the formula $(a \wedge d) \vee (b \wedge c)$. $o = \epsilon^*(s_0, Ti)$ is expectedly true; a, b, c, d are all true, so the formula evaluates to true.

Why did M output true? Well, there are two reasons. The first is the subtrace $T_{ad} = \text{set}(a); \text{set}(d)$, and the second is the subtrace $T_{bc} = \text{set}(b); \text{set}(c)$. Both of these subtraces are witnesses of o, so we should include both in an explanation of our output. We again revise the lesson from our previous example; **the cause of an output o is a *set* of witnesses of** o.

Example 3.3. Consider again the state machine M from the previous example, and consider the trace

$$T = \text{set}(a); \text{set}(b); \text{set}(c)$$

and request $i = \text{eval}((a \wedge \neg b) \vee c)$. $o = \epsilon^*(s_0, Ti)$ is true. $a \wedge \neg b$ evaluates to false because $\neg b$ is false, but c is true, so $(a \wedge \neg b) \vee c$ is true. Why did M output true? Well, as we just explained $(a \wedge \neg b) \vee c$ is true solely because c is true. Thus, the subtrace $T_c = \text{set}(c)$ should be the only explanation. However, the subtrace $T_a = \text{set}(a)$ is also a witness! If M executes T_a and then i, M will output true.

This is certainly not what we want. $\text{set}(a)$ does not contribute to our output, so it should be excluded. The problem here is that the subtrace T_a does not include the $\text{set}(b)$ request that ultimately keeps the $\text{set}(a)$ request from satisfying the formula. From this, we see that in order for a witness T' to be a good explanation of a particular output o, **it must be that every supertrace of T' in T is also a witness of** o. $T_{ab} = \text{set}(a); \text{set}(b)$ is a supertrace of T_a in T, but T_{ab} does not suffice to generate o. Thus, a is not a valid witness.

Combining our lessons from Example 3.1, Example 3.2, and Example 3.3, we arrive at our definition of wat-provenance. Given a state machine M, an input trace T, an input i, and the corresponding output $o = \epsilon^*(s_0, Ti)$, we say that a subtrace T' of T is a ***witness*** of o if $\epsilon^*(s_0, T'i) = o$. We say that a witness T' of o is ***closed under supertrace in*** T if every supertrace of T' in T is also a witness of o. Let $\text{Wit}(M, T, i)$ be the set of witnesses of o that are closed under supertrace in T. The ***wat-provenance*** of input i with respect to M and T, abbreviated $\text{Wat}(M, T, i)$, is the set of minimal elements of $\text{Wit}(M, T, i)$. That is, $\text{Wat}(M, T, i)$ consists of every witness T' of o such that (1) T' is closed under supertrace in T, and (2) no proper subtrace of T' is also a witness of o that satisfies (1)[2]. Note that we formally define wat-provenance with respect to an input i, but colloquially discuss wat-provenance with respect to the corresponding output o.

3.3 Can I Get a Witness?

We now provide a few more simple examples involving wat-provenance to illustrate the definition. In Section 4, we describe the wat-provenance of realistic services such as Redis, Zookeeper, and S3.

[2] Note the subtlety that to find $\text{Wat}(M, T, i)$, we first list all the witnesses of o that are closed under supertrace and then remove the non-minimal elements. We do *not* list all the minimal witnesses and then remove the ones that are not closed under supertrace in T. Informally, wat-provenance is minimal(closed_under_supertrace(witnesses)), not closed_under_supertrace(minimal(witnesses)). These two are not the same. See Example 3.5, for example.

Example 3.4. Consider again the key-value server state machine from Example 3.1, the trace

$$T = a_1a_2a_3 = \text{set}(x, 1); \text{set}(x, 2); \text{set}(x, 1)$$

and the input $i = \text{get}(x)$. $o = \epsilon^*(s_0, Ti) = 1$. To compute $\text{Wat}(M, T, i)$ (the wat-provenance of o), we first compute $\text{Wit}(M, T, i)$: the witnesses of o that are closed under supertrace in T.

$T_3 = a_3 = \text{set}(x, 1)$ suffices to generate o and is closed under supertrace in T because the proper supertraces a_1a_3, a_2a_3, and $a_1a_2a_3$ all generate o. So, $T_3 \in \text{Wit}(M, T, i)$. By a similar line of reasoning, we also find that a_1a_3, a_2a_3, and $a_1a_2a_3$ are in $\text{Wit}(M, T, i)$.

The subtrace $T_1 = a_1 = \text{set}(x, 1)$ is also a witness of o, but it is not closed under supertrace in T because the supertrace $a_1a_2 = \text{set}(x, 1); \text{set}(x, 2)$ does not generate o. Thus, T_1 is not in $\text{Wit}(M, T, i)$ and therefore not in $\text{Wat}(M, T, i)$. Intuitively this is correct because it is the second $\text{set}(x, 1)$ command, not the first, that causes the value of x to ultimately be 1.

All other subtraces of T are not witnesses of o. Thus, $\text{Wit}(M, T, i) = \{a_3, a_1a_3, a_2a_3, a_1a_2a_3\}$ which has unique minimal element a_3. Thus, $\text{Wat}(M, T, i) = \{a_3\}$.

Example 3.5. Consider again the key-value server state machine from Example 3.1 and Example 3.4 with the input alphabet expanded to include additions and subtractions to a particular variable. Consider the trace

$$T = a_1a_2a_3a_4 = \text{set}(x, 42); \text{add}(x, 1); \text{add}(x, 2); \text{sub}(x, 3)$$

and the request $i = \text{get}(x)$. $o = \epsilon^*(s_0, Ti) = 42$.

Again, we compute $\text{Wit}(M, T, i)$. T suffices to generate o, and T does not have any proper supertraces in T, so it is trivially closed under supertrace in T. Thus, $T \in \text{Wit}(M, T, i)$. $T_1 = a_1 = \text{set}(x, 42)$ is the only other witness of o, but T_1 is not closed under supertrace in T. $T_{12} = \text{set}(x, 42); \text{add}(x, 1)$ is a supertrace of T_1 but does not generate o. Thus, $\text{Wit}(M, T, i) = \text{Wat}(M, T, i) = \{T\}$.

Note that we first compute the witnesses that are closed under supertrace in T and then remove the non-minimal elements. Imagine instead if we had first computed the minimal witnesses and then removed the elements that were not closed under supertrace in T. We would have found that the sole minimal witness of o was T_1. Then, we would have filtered out T_1 because, as we just saw, it is not closed under supertrace in T. This would leave us with an empty wat-provenance!

Example 3.6. Consider a relational database state machine M. The input alphabet of M includes commands to insert a tuple into M and to execute a relational algebra query against M. Initially, all relations are empty. Consider the trace

$$T = a_1a_2a_3 = \text{insert}(R, t); \text{insert}(R, u); \text{insert}(S, u)$$

that inserts tuple t into relation R, inserts tuple u into relation R, and inserts tuple u into relation S. Consider the request $i = \text{query}(R - S)$ that queries the set difference $R - S$ of R and S. $o = \epsilon^*(s_0, Ti) = \{t\}$ is the set of only the tuple t.

We first compute $\text{Wit}(M, T, i)$. There are only three witnesses of o: a_1, a_1a_3, and $a_1a_2a_3$. a_1 is not closed under supertrace in T because the supertrace a_1a_2 does not generate o. The other two traces, a_1a_3 and $a_1a_2a_3$, are closed under supertrace. Thus, $\text{Wit}(M, T, i) = \{a_1a_3, a_1a_2a_3\}$, and $\text{Wat}(M, T, i) = \{a_1a_3\}$.

3.4 Wat-Provenance Properties

Example 3.1 through Example 3.6 demonstrate that our definition of wat-provenance accurately models our intuition about data provenance for state machines. We now discuss how wat-provenance relates to why-provenance and causality.

CLAIM 1. *Wat-provenance subsumes why-provenance.*

Intuitively, wat-provenance generalizes why-provenance from relational databases to arbitrary state machines. We can formalize this intuition in the following way.

Let M be a general relational database state machine (first introduced in Example 3.6) that allows for the insertions of tuples and the execution of monotone relational queries (i.e. queries composed of the monotone relational algebra operators select, project, join and union). Let I be an arbitrary database instance, and let T be a trace which inserts every tuple in I. Let input $i = \text{in}(t, Q)$ be an input which returns a boolean that indicates whether tuple t is in the result of evaluating query Q on instance I. Then, viewing a subtrace T' as a subinstance $I' \subseteq I$, $\text{Wat}(M, T, i) = \text{MWhy}(Q, I, t)$.

Proving this fact is straightforward. If $t \notin Q(I)$, then $o = \epsilon^*(s_0, Ti)$ returns false and $\text{Wat}(Q, I, t)$ consists only of the empty trace, indicating that $\text{MWhy}(Q, I, t)$ is empty. Otherwise $t \in Q(I)$ and o returns true. Consider a witness $T' \in \text{Wat}(M, T, i)$. T' suffices to generate o, so the corresponding instance I' suffices to generate t. Moreover, because Q is monotone and T does not contain any deletions, every witness T' is closed under supertrace in T. Thus, T' (and hence I') is a minimal witness. The proof of the converse—showing that every $I' \in \text{MWhy}(Q, I, t)$ has a corresponding subtrace $T' \in \text{Wat}(M, I, t)$—is symmetric.

Unfortunately, wat-provenance's generality does not come for free. Given a query Q, tuple t, and instance I, it is possible to automatically compute $\text{MWhy}(Q, I, t)$ because queries are constructed from a fixed set of simple relational operators. As we discuss in Section 4, it is normally intractable to automatically compute the wat-provenance of a particular state machine because, unlike relational queries, these state machines can have arbitrarily complex semantics.

Claim 2. *Wat-provenance refines causality.*

While wat-provenance *generalizes* why-provenance, it *refines* causality in the following sense. Consider a state machine M that executes a trace T and then an input request i. The causal history of i includes every single request in T (and the causal history of every request in T) whether or not the request in T actually did influence the output of executing request i. Wat-provenance instead returns the subtraces of T that are actual causes.

Note that wat-provenance returns a set of witnesses that are local to a particular node. Wat-provenance does not return the causal history of these witnesses, which includes messages sent by other nodes in the distributed system. In Section 5, we see how to enrich wat-provenance with this information.

3.5 Limitations

To clarify the strengths of wat-provenance, we pause to discuss its limitations.

Wat-provenance is not GDB. Wat-provenance is a high-level debugging technique. It can be used to identify the necessary and sufficient inputs that cause a state machine to produce a particular output (the *why*), but unlike low-level debugging tools like GDB, it cannot describe the details of the code that actually produces a particular output (the *how*).

To make things concrete, consider again our motivating example from Section 3.1 in which two of Ava's requests were reordered, allowing Bob to erroneously see Ava's mean comment. We can use wat-provenance to debug scenarios like this one. These types of scenarios require us to reason about which messages affect other messages, to reason about how events are ordered with respect to one another, and to reason about how data is transferred across multiple machines across a span of time. On the other hand, wat-provenance cannot help us understand why, for example, a particular node is segfaulting; we need a lower-level tool like GDB for this.

Thus, we view wat-provenance and GDB-like debuggers as complementary. When debugging, a developer can use wat-provenance to trace the root cause of a bug across a system, narrowing their attention down to only the relevant inputs. Then, if needed, they can use a low-level debugger like GDB to discover the details of what's going wrong.

Wat-provenance requires determinism. Wat-provenance is defined with respect to *deterministic* state machines, yet many pieces of code are nondeterministic. For example, the behavior of many weakly consistent distributed systems depend on the non-deterministic ordering of messages in the network. Similarly, some load balancers intentionally use randomness when deciding which machine should receive a particular request. Wat-provenance specifications cannot be applied to nondeterministic systems like these. We leave a generalization of wat-provenance to nondeterministic state machines as an interesting avenue for future work.

4 WAT-PROVENANCE SPECIFICATIONS

Now that we have defined wat-provenance, we turn to the matter of computing it.

4.1 Provenance Specifications

Automatically computing the wat-provenance for an *arbitrary* distributed system component, which we dub a ***black box***, is often intractable and sometimes impossible. Computing the wat-provenance of a black box requires that we analyze the black box's implementation to extract the relationships between the inputs and outputs of the black box. Because black box implementations can be large and complex, this program analysis is almost always intractable. Worse, we may not have access to the source code of the black box at all. For example, cloud services like Amazon S3 or Google Cloud Spanner have proprietary implementations. In this case, automatically computing wat-provenance is impossible.

Though automatically computing the wat-provenance for an *arbitrary* black box is intractable, we can take advantage of the fact that many real-world black boxes are far from arbitrary. Many black boxes have complex implementations but are designed with very simple interfaces. This allows us to sidestep the issue of *inferring* wat-provenance from an implementation and instead *specify* wat-provenance directly from an interface. That is, we can write a ***wat-provenance specification***: a function that—given a trace T and request i—directly returns the wat-provenance $\text{Wat}(M, T, i)$ for a black box modeled as state machine M.

For example, if we restrict our attention to the `get` and `set` API of Redis, then the wat-provenance specification of a `get` request is trivial: the wat-provenance of a `get` request for key k includes only the most recent `set` to k. Redis is implemented in over 50,000 lines of C. Analyzing this body of code and inferring the wat-provenance of a `get` request is infeasible using modern program analysis techniques. Wat-provenance specification avoids this issue entirely and instead specifies the wat-provenance in a single line of text.

Moreover, codifying this one-line wat-provenance specification is expectedly straightforward. Wat-provenance specifications are implemented as functions that take in a trace T and an input i and return the wat-provenance of i. Thus, wat-provenance specifications can be written in any programming language and can use any language features available. Wat-provenance specifications do not have to be written using any special domain specific language or using any restricted subset of a language.

As a simple example, we provide a Python implementation of the wat-provenance specification in Figure 4. The specification, get_prov, takes in a trace T and a get request i for key k. Redis requests are represented as objects of type Request with subclasses GetRequest and SetRequest. get_prov iterates through the trace in reverse order, looking for a set request to key k. If such a set request is found, get_prov returns it. Otherwise, get_prov returns an empty witness.

```
def get_prov(T: List[Request], i: GetRequest):
  for a in reversed(T):
    if (isinstance(a, SetRequest)
        and a.key == i.key):
      return {[a]}
  return {[]}
```

Figure 4: A Python implementation of a Redis get request wat-provenance specification

In Section 5, we present a prototype implementation of a system for writing wat-provenance specifications and describe the details of how the system collects traces. In this section, we omit these details and focus on the concepts behind wat-provenance specifications.

4.2 Examples

The wat-provenance specification of a Redis get request is particularly simple, but in our experience this simplicity is not completely uncommon. We now survey a variety of commonly used black boxes that have relatively simple wat-provenance specifications. Later, we discuss some black boxes for which writing wat-provenance specifications is more difficult.

Key-Value Stores. We have already seen a wat-provenance specification for the get and set API of Redis. We can easily extend our wat-provenance specification to handle more of Redis' API. For example, consider the operations append, decr, decrby, incr, and incrby that all modify the value associated with a particular key. With these operations present in a trace, the wat-provenance specification for a get request to key k now includes the most recent set to k and all subsequent modifying operations to key k.

Object Stores. We can also write wat-provenance specifications for storage systems that are more complex than key-value stores. For example, consider an object store like Amazon S3 where users can create, move, copy, list, and get buckets and objects. The wat-provenance specification for the get of an object o in bucket b includes the most recent creation of the bucket b and the most recent creation of o. If a bucket or object was created by a move, then the provenance also includes the provenance of the moved bucket or object. The wat-provenance of a request to list the contents of a bucket includes the most recent creation of the bucket, the most recent creation of every object in the bucket, and the deletion of any object that was previously in the bucket.

Distributed File Systems. We can specify the wat-provenance of a distributed file system like HDFS. A wat-provenance specification of a request to read a byte range from a file includes the most recent creation of the file, the most recent creation of the parent directories of the file, and the most recent writes that overlap with the requested byte range. If the file was created by moving another file, then the wat-provenance specification also includes the provenance of the file that was moved.

Coordination Services. Systems use coordination services like Apache Zookeeper [21] and Chubby [9] for leader election, mutual exclusion, etc. Take Zookeeper as an example. Zookeeper's API resembles that of a file system; users can create, delete, write, and read file-like objects called znodes. Though the implementation of Zookeeper and HDFS are radically different, their APIs (and thus their wat-provenance specifications) are similar. For example, the wat-provenance specification of a request to read a znode includes the most recent creation of the znode and the most recent creations of all ancestor znodes.

Load Balancers. Consider a load balancer, like HAProxy, that is balancing load across a set $s_1, \ldots, s_n$ of n servers. Periodically, a server s_i sends a heartbeat to the load balancer that includes s_i's average load for the last five minutes. Whenever the load balancer receives a message from a client, it forwards the message to the server s_i that is least loaded. Modelling the forwarding decision s_i as the output of the load balancer, the wat-provenance specification for the forwarding decision includes the most recent heartbeat message from the least-loaded server.

Stateless Services. A stateless service is a service that can be modelled as a state machine with a single state. Given a request, a stateless service always produces the same reply, no matter what other requests it has already serviced. For example, a web server serving a static website is stateless; it replies to all requests with the same website. Wat-provenance specifications of a stateless service are trivial. Requests are completely independent, so the wat-provenance of any request consists only of the empty witness.

4.3 Limitations

Though wat-provenance specifications are often simple to write, they are not a panacea. Here, we discuss some limitations of wat-provenance specifications.

Complex wat-provenance specifications. There are some black boxes for which writing a wat-provenance specification is inherently very difficult. For example, consider a state machine that implements an online support vector machine (SVM) [7, 19, 27]. Clients can either (a) submit training data to the state machine to update the model or (b) submit test data to the state machine for classification. The wat-provenance of a classification request includes only the support vectors of the model at the time of classification. Writing a wat-provenance specification to identify these support vectors is possible but very difficult. Such a wat-provenance specification would likely be as complicated as the state machine itself.

Buggy black boxes. We have thus far tacitly assumed that black boxes like Redis faithfully implement their advertised interfaces. However, if a black box is buggy and deviates from its expected behavior, then a wat-provenance specification will produce erroneous provenance. In other words, wat-provenance specifications are written with respect to the semantics of an abstract state machine. If a particular black box does not concretely implement these semantics, then the wat-provenance specification is incorrect.

5 WATERMELON

In this section, we present Watermelon: a prototype distributed debugging framework that leverages the theoretical foundations of wat-provenance and wat-provenance specifications.

Watermelon uses wat-provenance specifications to generate the provenance of data as it transits through the black box components of a distributed system. To write a wat-provenance specification for a black box, a developer must first wrap the black box in a Watermelon shim. A shim acts as proxy, intercepting all inbound requests sent to a black box and all outbound replies produced by a black box. Watermelon shims provide two key pieces of functionality.

First, Watermelon shims are responsible for recording the trace T of requests that are sent to a black box, as well as the corresponding replies produced by the black box. These traces are later used as the inputs to wat-provenance specifications. Currently, Watermelon shims persist traces in a relational database.

Second, Watermelon shims implement a simple distributed tracing service. Whenever a Watermelon shim receives a request, it records the address of the message's sender along with the request. Similarly, whenever a Watermelon shim sends a request, it records the address of the message's destination. This enables a developer to integrate the wat-provenance of multiple black boxes within a distributed system. To find the cause of a particular black box output, we invoke the black box's wat-provenance specification. The specification returns the set of witnesses that cause the output. Then, we can trace a request in a witness back to the black box that sent it and repeat the process, invoking the sender's wat-provenance specification to get a new set of witnesses.

After a user has written a black box's shim, they can write the black box's wat-provenance specification. Watermelon wat-provenance specifications are simple scripts written in a developer's choice of either SQL or Python. Given a particular black box request, a wat-provenance script computes the corresponding wat-provenance with respect to the black box's trace (which is persisted in a relational database by the black box's shim).

6 EVALUATION

In this section, we answer two questions: (1) How difficult is it to write wat-provenance specifications? and (2) How do wat-provenance and wat-provenance specifications compare to other debugging techniques? We answer question 1 in Section 6.1 and question 2 in Section 6.2.

6.1 Wat-Provenance Specifications

In Section 4, we argued that many commonly used distributed system components have relatively simple wat-provenance specifications. In this subsection, we substantiate our argument with concrete wat-provenance specifications for Redis, a subset of the POSIX file system API, Amazon S3, and Zookeeper. We implemented these wat-provenance specifications using Watermelon. Table 1 lists the language in which we wrote each provenance specification, the lines of code required to write each specification, the number of APIs supported by each specification, and the API supported by each specification.

We found it simple to write wat-provenance specifications for 17 of the 20 APIs. We found specifying three of the APIs slightly more challenging, but still relatively straightforward. First, specifying the read of a byte range in a file system required us to find the most recent write to each segment of the byte range. This required us to scan backwards through the trace, maintaining a disjoint set of byte ranges. Second, specifying the catting and listing of objects in Amazon S3 was complicated by the fact that objects could be moved across buckets. Thus, computing the wat-provenance required computing the transitive wat-provenance of objects that have been moved and copied.

As we discussed in Section 4.3, this does not mean that *all* black boxes have simple wat-provenance specifications, but it corroborates the claim that many commonly used black boxes do.

Table 1: Watermelon wat-provenance specifications

System	Language	LOC	Number of APIs	Supported API
Redis	SQL	30	9	get, set, del, append, incr, decr, incrby, decrby, strlen
POSIX	Python	88	2	reading and writing byte ranges
Amazon S3	Python	200	5	creating, copying, catting, removing, and listing objects and buckets
Zookeeper	SQL	70	4	creating, reading, writing, and listing znodes

6.2 Debugging with Wat-Provenance

How do wat-provenance and wat-provenance specifications compare to other debugging techniques? We answer this question by comparing Watermelon, our prototype wat-provenance debugger, against two other debugging techniques: SPADE [16] and `printf` debugging. We qualitatively evaluate the three debugging techniques using four metrics—ease of adoption, runtime overhead, high-level debugging support, and low-level debugging support—which we will explain momentarily. This qualitative analysis is summarized in Table 2.

6.2.1 SPADE. SPADE [16] is a framework that collects provenance information from arbitrary black boxes in a distributed system. SPADE collects provenance information from a variety of sources including operating system audit logs, network artifacts, LLVM instrumented applications, and applications dynamically instrumented for taint analysis. These techniques are the current state of the art in extracting the provenance of unmodified black boxes. For the sake of brevity, we will focus only on the provenance that SPADE collects from operating system audit logs and LLVM instrumented applications.

Ease of Adoption. Our first metric, ***ease of adoption***, is a measure of how difficult it is for a developer to set up the infrastructure required to use a particular debugging technique. Ease-of-adoption does not include the difficulty of actually debugging using a debugging technique; it only includes the difficulty of making a system amenable to the debugging technique. The ease of adoption of SPADE, for example, is very low. For SPADE to collect operating system audit logs, binaries can be run unmodified. To collect LLVM call graphs, binaries have to be compiled with LLVM instrumentation, but the code itself remains unchanged.

Runtime Overhead. Our second metric, ***runtime overhead***, is a measure of the performance overheads that a debugging technique imposes on a system. When SPADE collects operating system audit logs, it imposes a negligible amount of runtime overhead, as operating system audit logs are created whether or not SPADE is being used. LLVM instrumented binaries, on the other hand, run significantly slower than their uninstrumented counterparts. As a simple example, on an Amazon EC2 m5.xlarge instance, we measured that a single Redis client required 2.45 seconds to send 100,000 synchronous `PING` operations to a default configured Redis server running on the same machine. When we compiled the Redis server with LLVM instrumentation, the client required 29.76 seconds, an order of magnitude decrease in throughput.

High- and Low-Level Debugging Support. Our third and fourth metrics, ***high-level debugging support*** and ***low-level debugging support***, are measures of whether a debugging technique facilitates high-level and low-level debugging. As we described in Section 3.5, high-level debugging involves understanding which events in a distributed system cause each other, how events are ordered with respect to one another, etc. Conversely, low-level debugging involves reasoning about the details of how a particular program executes.

We ran SPADE—with audit logging and LLVM instrumentation enabled—against a trivial workload consisting of a single set and get to Redis. SPADE produced 1,087 audit log provenance entries and 1,118,764 LLVM instrumentation provenance entries. High-level debugging with either of these sources of provenance is difficult due to the provenance's size and detailed nature. Understanding the reported provenance requires either an understanding of how particular syscalls relate to Redis' source code (for audit logs) or a detailed understanding of Redis' implementation (for LLVM instrumentation). Low-level debugging with audit logs is challenging, because the audit logs lack information about the execution of the code being debugged (besides the syscalls that it makes). Low-level debugging with LLVM instrumentation is easier but still challenging due to the sheer volume of provenance information produced.

6.2.2 `printf` Debugging. "`printf` debugging" is the ad-hoc debugging technique in which a developer adds `printf` statements (or log statements) to the various components of a distributed system and then debugs the system by analyzing the resulting logs.

Ease of Adoption. The ease of implementing `printf` debugging depends on (a) how many log statements a developer wants to add and (b) the piece of code to which a developer

Table 2: A qualitative comparison of debugging techniques

Debugging Technique	Ease Of Adoption	Runtime Overheads	Supports High-Level Debugging	Supports Low-Level Debugging
SPADE (Audit Logs)	easy	low	no	some
SPADE (LLVM)	easy	high	no	yes
`printf` Debugging	easy–impossible	low	some	yes
Watermelon	easy–hard	medium	yes	no

wants to add log statements. It is relatively easy for a developer to add logging to a piece of code that they wrote because they know where and what to log. If the code is a complex piece of open source software (e.g. Apache Cassandra, Apache Zookeeper), then adding log statements requires understanding at least part of the software's implementation. If the code is closed-source (e.g. Cloud Spanner, Amazon Redshift), then adding log statements is impossible.

Runtime Overheads. The overheads of logging are typically negligible compared to heavier weight techniques like LLVM instrumentation or Watermelon shims.

High- and Low-Level Debugging Support. The ability to perform high- and low-level debugging with log statements is challenging to characterize. It varies from easy to difficult based on the complexity of the bug being debugged, the quantity and quality of logs, etc. Still, for high-level debugging, there is an unavoidable burden imposed by `printf` debugging. Developers have to make sure enough information is logged, collect logs from a number of machines (some of which might have crashed), filter out irrelevant entries from the potentially voluminous logs, and trace data across multiple logs.

6.2.3 *Watermelon.*

Ease of Adoption. To adopt Watermelon, a developer must first write wat-provenance specifications for each component in the system. As we discussed, many distributed system components have relatively simple wat-provenance specifications for which adopting Watermelon is relatively easy, while some components have very complex wat-provenance specifications that make Watermelon more challenging to adopt.

Runtime Overheads. Watermelon shims introduce a non-negligible amount of runtime overhead by intercepting network messages and periodically persisting traces. We measured the performance overheads of Watermelon shims on trivial workload in which a Redis client performs a series of `SET` operations against a Redis server. Watermelon shims decreased the system's throughput by 41.4%. While some of this performance degradation is an artifact of our prototype, some is inherent to shims.

High- and Low-Level Debugging Support. Watermelon's greatest strength is that it makes high-level debugging easy. Wat-provenance formalizes the intuitive notion of which inputs in a distributed system cause a particular output. Wat-provenance specifications allow a developer to codify these intuitions and automatically use them to filter irrelevant information when debugging. By refining causality, wat-provenance also allows developers to reason about the ordering of events in a distributed system without having to manually trace events through a set of logs. Conversely, Watermelon must be paired with another system to support low-level debugging.

As a simple experiment, we again ran our trivial workload consisting of a single set and get to Redis. Whereas SPADE produced 1,089 audit log provenance entries and 1,118,764 LLVM instrumentation provenance entries, Watermelon produced 8 provenance entries: the get and set request and response recorded on the client and the server.

7 ON WHITE BOXES

In Section 4, we saw how easy it is to write shims for a variety of common black box components. In return for this modest investment, Watermelon allows programmers to ask and answer debugging questions that span node boundaries and involve mutable state that changes over time. It is tempting to ask if we can take the idea further. If our components are written in a language amenable to provenance collection—if we work only with "white boxes"—can we infer their wat-provenance specifications automatically? Or, better still, can we obtain a form of provenance even richer than wat-provenance: one that explains not just *which* input tuples contribute to an output tuple, but also precisely *how*?

Unfortunately, automatically extracting wat-provenance remains elusive, even for state machines written in a restricted white-box language for provenance. Take for example provenance-enhanced distributed logic programming languages such as NDLog [30] and Dedalus [3] that store system state in relations and represent programs as collections of relational queries. Each exposes a form of why-provenance

that accounts for state that changes over time. In an NDLog program, effect-producing events—such as the addition, deletion, or update of records—are reified into the program's provenance graph. In Dedalus, logical time is reified into all of the tuples.

These custom representations of provenance in time, while not formalized, are similar in spirit to wat-provenance. It turns out, however, that they are *not* equivalent to wat-provenance—neither Dedalus nor NDLog provenance necessarily identify minimal witnesses. The underlying issue is the problem of ***negative provenance*** or the provenance of non-answers [10, 20], which remains an open issue in the database research community.

Consider the key-value state machine presented in Example 3.1, extended to support two new requests. The freeze request makes the existing set of keys and values immutable, while trunc deletes all of the keys and values (provided that freeze has not already been called). Now consider the trace

$$T = a_1 a_2 a_3 = \text{set}(x, 1); \text{freeze}; \text{trunc}$$

The wat-provenance of $i = \text{get}(x)$ is the singleton set $\{a_1 a_2\}$ consisting of the single witness $a_1 a_2$. But explaining why a_2 is part of this witness requires reasoning about negative provenance: x had the value 1 both because of a_1 (which inserted the value into the store) and because a_3 had *no effect* (due to the action of a_2, *without which* a_3 would have removed the effects of a_1!). If the state machine was written in a language such as NDLog or Dedalus, implementing it would require the use of logical negation to capture the reasoning that trunc applies only if freeze *does not exist* before it. (In fact, Example 3.4 through Example 3.6 all require nonmonotonic logic.) Explanations of the output to i would therefore require both positive and negative provenance. Explaining why a particular event *did not occur* is intractable in general, as the explanation may be infinitely large. Existing techniques for collecting negative provenance apply heuristics that over-approximate the why provenance of non-answers. For example, Dedalus provenance would determine that the provenance of i is $\{a_1 a_2 a_3\}$.

We could, of course, constrain the white-box programming language to rule out the complication of negative provenance. The language Dedalus$^+$ [31] is the positive fragment of Dedalus, in which negation is not permitted (except on base relations). Programs written in Dedalus$^+$ generate positive why-provenance graphs whose leaves correspond exactly with the wat-provenance of the given execution. Unfortunately, Dedalus$^+$ is not adequately expressive to implement arbitrary stateful distributed services. The CALM theorem [2, 4] shows that programs produce a single consistent output under all input orders if and only if those programs do not use negation in their logic. Therefore programs whose outputs are (by definition) dependent upon their input orders—such as mutable key/value stores, file systems and object stores, caches, etc—cannot be implemented in a logic language *without* using negation! Negative provenance is requirement for capturing the semantics of typical stateful services.

In short, fully automatic collection of wat-provenance for general-purpose systems seems tricky. As future work, we plan to continue efforts to design of an expressive programming language for which we can automatically extract wat-provenance with a minimum of programmer assistance.

8 RELATED WORK

Data Provenance. When discussing data provenance, we have focused primarily on why-provenance, as wat-provenance is a generalization of why-provenance. However, there are other forms of data provenance. How-provenance [17] formalizes not only why a particular output is produced by a relational query, but also how. Where-provenance [8] formalizes the set of input *attributes* that contribute to an output *attribute*. In [12], Cui et al. provide algorithms for computing the provenance of more generic data transformation operators that satisfy certain properties (e.g., homomorphisms, aggregators). In [42], Woodruff et al. use weak inverse functions to compute the provenance of generic operators. Why-, how-, and where-provenance are all limited to the domain of relational queries, and all five forms of provenance are limited to the domain of data transformations applied to static data. Wat-provenance is applicable to state machines with state that varies over time.

Black Box Provenance. RDataTracker [28], noWorkflow [33], and SPADE [16] are frameworks that attempt to record the provenance of data through an *arbitrary* black box using general purpose low-level provenance tracing techniques. RDataTracker and noWorkflow use reflection and runtime information to track the provenance of data through minimally annotated R and Python scripts. SPADE collects provenance information from operating system audit logs, network artifacts, LLVM instrumented applications, and applications dynamically instrumented for taint analysis. The benefit of these frameworks is their generality. However, their generality comes at the cost of verbosity. These provenance frameworks produce a large amount of low-level implementation-specific provenance information that can be challenging to interpret.

Data-Dependent Process Provenance. A data-dependent process (DDP) is a finite state machine that evaluates queries against a relational database to determine some transitions and uses external requests to trigger other transitions [13, 14]. In [13] and [14], Deutch et al. extend provenance semirings [17] to linear temporal logic formulas issued against

a DDP. Both DDP provenance and wat-provenance aim to extend traditional data provenance to state machines, but they do so in very different ways. DDP provenance is used to explain how particular execution traces may arise. Wat-provenance, on the other hand, aims to explain why a state machine generates a particular output with respect to a fixed trace of inputs.

Network Provenance. Network provenance was originally introduced in [47] as a generalization of data provenance to programs written in the extended relational language NDLog [30]. ExSPAN [47] and DistTape [46] are two accompanying implementations that record the network provenance of a distributed system implemented in NDLog. The network provenance of a piece of data is tied to the particular NDLog program that created it, whereas wat-provenance is defined with respect to abstract state machines instead of concrete implementations. As a result, network provenance provides finer grained debugging information than wat-provenance, but is applicable only to systems written in NDLog. Wat-provenance provides coarser grained debugging information, but is applicable to arbitrary state machines.

Scientific Workflow Systems. Scientific workflow systems like Taverna [41], Kepler [1], Pegasus [24], and Swift [43] can be used to structure complex computational tasks as a graph, composing tasks by connecting the outputs of one task to the inputs of another. These systems are limited to the workflow languages that they support, and these languages are not typically used to build distributed systems. They also assume that the inputs to a workflow are static. Wat-provenance formalizes provenance for arbitrary distributed systems composed of time-varying state machines.

DISC Provenance. Data intensive scalable computing (DISC) frameworks like Apache Hadoop [38] and Apache Spark [44] can execute data-parallel programs on massive amounts of data both efficiently and with fault tolerance. Work on GMRW provenance [22] and systems like BigDebug [18], Titian [23], RAMP [34], and Newt [29] augment DISC frameworks with data provenance. Unlike wat-provenance, DISC provenance is restricted to the domain of data intensive computations over static data and cannot be applied to other distributed systems components like storage systems, coordination services, load balancers, etc. DISC provenance frameworks take advantage of operator interfaces (e.g., map, filter, aggregate) to construct lineage of arbitrary operator implementations similar to how wat-provenance specifications are written against high-level interfaces instead of implementations.

Distributed Tracing Tools. Distributed tracing tools like Dapper [39], X-trace [15], Magpie [6], Stardust [40], and lprof [45] allow programmers to trace messages through large and complex distributed systems that span multiple nodes and administrative domains. The fundamental distinction between distributed traces and wat-provenance is that distributed traces do not carry historical information that link prior inputs to present outputs. For example, traces may show you which clients contacted a key-value store and when, but they will not show you which requests wrote the values that later requests read back. Also note that lprof, unlike other tracing tools, constructs traces by analyzing systems' existing logs. In the future, we plan to explore whether we can use a similar technique to construct traces.

9 CONCLUSION

This paper identified inadequacies in existing formalisms used to reason about the causes of events in distributed systems. Causality overapproximates the true cause of an event, and data provenance is restricted to the domain of static (typically relational) data. We then presented wat-provenance: a novel form of provenance that generalizes why-provenance and refines causality. We then discussed how to sidestep the complexity of automatic wat-provenance extraction with wat-provenance specifications written against the high-level API of a black box.

ACKNOWLEDGMENTS

We thank Lennart Oldenburg, Sanjay Krishnan, Alvin Cheung, and Anthony Tan for fruitful discussion and feedback. We also thank our shepherd, Ken Yocum, for his helpful insights and guidance. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, National Science Foundation Grant No. 1652368, and gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft, Scotiabank, Splunk and VMware.

REFERENCES

[1] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. 2006. Provenance collection support in the kepler scientific workflow system. In *International Provenance and Annotation Workshop*. Springer, 118–132.
[2] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. 249–260.
[3] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in time and space. In *Datalog Reloaded*. Springer, 262–281.
[4] Tom J. Ameloot, Frank Neven, and Jan Van Den Bussche. 2013. Relational Transducers for Declarative Networking. *J. ACM* 60, 2 (2013).
[5] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2012. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM.
[6] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-aware Systems.. In *HotOS*. 85–90.

[7] Antoine Bordes and Léon Bottou. [n. d.]. The Huller: a simple and efficient online SVM. ([n. d.]).

[8] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *International conference on database theory*. Springer, 316–330.

[9] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 335–350.

[10] Adriane Chapman and H. V. Jagadish. 2009. Why Not?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. ACM, 523–534.

[11] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. 2009. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases* 1, 4 (2009), 379–474.

[12] Yingwei Cui and Jennifer Widom. 2003. Lineage tracing for general data warehouse transformations. *The VLDB Journal-The International Journal on Very Large Data Bases* 12, 1 (2003), 41–58.

[13] Daniel Deutch, Yuval Moskovitch, and Val Tannen. 2014. A provenance framework for data-dependent process analysis. *Proceedings of the VLDB Endowment* 7, 6 (2014), 457–468.

[14] Daniel Deutch, Yuval Moskovitch, and Val Tannen. 2015. Provenance-based analysis of data-centric processes. *The VLDB Journal* 24, 4 (2015), 583–607.

[15] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association.

[16] Ashish Gehani and Dawood Tariq. 2012. SPADE: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 101–120.

[17] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 31–40.

[18] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 784–795.

[19] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. 1998. Support vector machines. *IEEE Intelligent Systems and their applications* 13, 4 (1998), 18–28.

[20] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. 2008. On the Provenance of Non-answers to Queries over Extracted Data. *Proc. VLDB Endow.* 1, 1 (2008), 736–747.

[21] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX annual technical conference*, Vol. 8. Boston, MA, USA.

[22] Robert Ikeda, Hyunjung Park, and Jennifer Widom. [n. d.]. Provenance for Generalized Map and Reduce Workflows. In *CIDR 2011*. Stanford InfoLab. http://ilpubs.stanford.edu:8090/985/

[23] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data provenance support in spark. *Proceedings of the VLDB Endowment* 9, 3 (2015), 216–227.

[24] Jihie Kim, Ewa Deelman, Yolanda Gil, Gaurang Mehta, and Varun Ratnakar. 2008. Provenance trails in the wings/pegasus system. *Concurrency and Computation: Practice and Experience* 20, 5 (2008), 587–597.

[25] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[26] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.

[27] Pavel Laskov, Christian Gehl, Stefan Krüger, and Klaus-Robert Müller. 2006. Incremental support vector learning: Analysis, implementation and applications. *Journal of machine learning research* 7, Sep (2006), 1909–1936.

[28] Barbara S Lerner and Emery R Boose. 2014. Collecting provenance in an interactive scripting environment. In *Workshop on the Theory and Practice of Provenance (TaPP), Cologne, Germany*.

[29] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable lineage capture for debugging disc analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 17.

[30] Boon Thau Loo and Joseph M Hellerstein. 2006. *The design and implementation of declarative networks*. Ph.D. Dissertation. University of California, Berkeley.

[31] William R Marczak, Peter Alvaro, Neil Conway, Joseph M Hellerstein, and David Maier. 2012. Confluence analysis for distributed programs: A model-theoretic approach. In *Datalog in Academia and Industry*. Springer, 135–147.

[32] Keith Ansel Marzullo. 1984. Maintaining the time in a distributed system: an example of a loosely-coupled distributed service (synchronization, fault-tolerance, debugging). (1984).

[33] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2014. noWorkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*. Springer, 71–83.

[34] Hyunjung Park, Robert Ikeda, and Jennifer Widom. 2011. Ramp: A system for capturing and tracing provenance in mapreduce workflows. (2011).

[35] Amritha Sampath and C Tripti. 2012. Synchronization in distributed systems. In *Advances in Computing and Information Technology*. Springer, 417–424.

[36] Ulrich Schmid. 2000. Orthogonal accuracy clock synchronization. *Chicago Journal of Theoretical Computer Science* 3, 2000 (2000), 3–77.

[37] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[38] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 1–10.

[39] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report. Technical report, Google, Inc.

[40] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. 2006. Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 34. ACM, 3–14.

[41] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. 2013. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic acids research* 41, W1 (2013).

[42] Allison Woodruff and Michael Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In *Data Engineering, 1997. Proceedings. 13th International Conference on*. IEEE, 91–102.

[43] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. 2013. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM*

International Symposium on. IEEE, 95–102.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010).

[45] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *OSDI*, Vol. 14. 629–644.

[46] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. 2012. Distributed time-aware provenance. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 49–60.

[47] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 615–626.