# Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems

JONATHAN MACE, Max Planck Institute for Software Systems
RYAN ROELKE and RODRIGO FONSECA, Brown University

Monitoring and troubleshooting distributed systems is notoriously difficult; potential problems are complex, varied, and unpredictable. The monitoring and diagnosis tools commonly used today—logs, counters, and metrics—have two important limitations: what gets recorded is defined *a priori*, and the information is recorded in a component- or machine-centric way, making it extremely hard to correlate events that cross these boundaries. This article presents Pivot Tracing, a monitoring framework for distributed systems that addresses both limitations by combining dynamic instrumentation with a novel relational operator: the happened-before join. Pivot Tracing gives users, at runtime, the ability to define arbitrary metrics at one point of the system, while being able to select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries. We have implemented a prototype of Pivot Tracing for Java-based systems and evaluate it on a heterogeneous Hadoop cluster comprising HDFS, HBase, MapReduce, and YARN. We show that Pivot Tracing can effectively identify a diverse range of root causes such as software bugs, misconfiguration, and limping hardware. We show that Pivot Tracing is dynamic, extensible, and enables cross-tier analysis between inter-operating applications, with low execution overhead.

CCS Concepts: • **Computer systems organization → Cloud computing**; *n-tier architectures*; *Distributed architectures*; • **Software and its engineering → Interoperability**; *System administration*; *Software performance*; *Domain specific languages*;

Additional Key Words and Phrases: Distributed systems monitoring, end-to-end tracing

**ACM Reference format:**
Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Trans. Comput. Syst.* 35, 4, Article 11 (December 2018), 28 pages.
https://doi.org/10.1145/3208104

## 1 INTRODUCTION

Monitoring and troubleshooting distributed systems is hard. The potential problems are myriad: hardware and software failures, misconfigurations, hot spots, aggressive tenants, or even simply unrealistic user expectations. Despite the complex, varied, and unpredictable nature of these problems, most monitoring and diagnosis tools commonly used today—logs, counters, and metrics—have at least two fundamental limitations: what gets recorded is defined *a priori*, at development

This work was partially supported by NSF award #1452712 and by a Google Faculty Research Award.
Authors' addresses: J. Mace, Max Planck Institute for Software Systems, Campus E1 5, Saarbrücken 66123, Germany; email: jcmace@mpi-sws.org; R. Fonseca and R. Roelke, Brown University, 115 Waterman Street, Providence RI 02912, USA; email: rfonseca@cs.brown.edu.

0734-2071/2018/12-ART11 $15.00
https://doi.org/10.1145/3208104

or deployment time, and the information is captured in a component- or machine-centric way, making it extremely difficult to correlate events that cross these boundaries.

While there has been great progress in using machine learning techniques [60, 76, 78, 103] and static analysis [106, 107] to improve the quality of logs and their use in troubleshooting, they carry an inherent tradeoff between recall and overhead, as what gets logged must be defined *a priori*. Similarly, with monitoring, performance counters may be too coarse-grained [73]; and if a user requests additional metrics, a cost-benefit tug-of-war with the developers can ensue [19].

Dynamic instrumentation systems such as Fay [54] and DTrace [38] enable the diagnosis of unanticipated performance problems in production systems [37] by providing the ability to select, at runtime, which of a large number of tracepoints to activate. Both Fay and DTrace, however, are still limited when it comes to correlating events that cross address-space or OS-instance boundaries. This limitation is fundamental, as neither Fay nor DTrace can affect the monitored system to propagate the monitoring context across these boundaries.

In this article, we combine dynamic instrumentation with causal tracing techniques [39, 55, 93] to fundamentally increase the power and applicability of either technique. We present Pivot Tracing, a monitoring framework that gives operators and users, at runtime, the ability to obtain an arbitrary metric at one point of the system, while selecting, filtering, and grouping by events meaningful at other parts of the system, even when crossing component or machine boundaries.

Like Fay, Pivot Tracing models the monitoring and tracing of a system as high-level queries over a dynamic dataset of distributed events. Pivot Tracing exposes an API for specifying such queries and efficiently evaluates them across the distributed system, returning a streaming dataset of results.

The key contribution of Pivot Tracing is the "happened-before join" operator, $\overrightarrow{\bowtie}$, that enables queries to be contextualized by Lamport's happened-before relation, $\rightarrow$ [65]. Using $\overrightarrow{\bowtie}$, queries can group and filter events based on properties of any events that causally precede them in an execution.

To track the happened-before relation between events, Pivot Tracing borrows from causal tracing techniques, and utilizes a generic metadata propagation mechanism for passing partial query execution state along the execution path of each request. This enables inline evaluation of joins during request execution, drastically mitigating query overhead and avoiding the scalability issues of global evaluation.

Pivot Tracing takes inspiration from data cubes in the online analytical processing domain [56], and derives its name from spreadsheets' pivot tables and pivot charts [48], which can dynamically select values, functions, and grouping dimensions from an underlying dataset. Pivot Tracing is intended for use in both manual and automated diagnosis tasks, and to support both one-off queries for interactive debugging and standing queries for long-running system monitoring. It can serve as the foundation for the development of further diagnosis tools. Pivot Tracing queries impose truly no overhead when disabled and utilize dynamic instrumentation for runtime installation.

We have implemented a prototype of Pivot Tracing for Java-based systems and evaluate it on a heterogeneous Hadoop cluster comprising HDFS, HBase, MapReduce, and YARN. In our evaluation, we show that Pivot Tracing can effectively identify a diverse range of root causes such as software bugs, misconfiguration, and limping hardware. We show that Pivot Tracing is dynamic, extensible to new kinds of analysis, and enables cross-tier analysis between inter-operating applications with low execution overhead.

In summary, this article has the following contributions:

- Introduces the abstraction of the *happened-before join* ($\overrightarrow{\bowtie}$) for arbitrary event correlations;
- Presents an efficient query optimization strategy and implementation for $\overrightarrow{\bowtie}$ at runtime, using dynamic instrumentation and cross-component causal tracing;

—Presents a prototype implementation of Pivot Tracing in Java, applied to multiple components of the Hadoop stack;
—Evaluates the utility and flexibility of Pivot Tracing to diagnose real problems.

## 2 MOTIVATION

### 2.1 Pivot Tracing in Action

In this section, we motivate Pivot Tracing with a monitoring task on the Hadoop stack. Our goal here is to demonstrate some of what Pivot Tracing can do, and we leave details of its design, query language, and implementation to Section 3, Section 4, and Section 5, respectively.

Suppose we want to apportion the disk bandwidth usage across a cluster of eight machines simultaneously running HBase, Hadoop MapReduce, and direct HDFS clients. Section 6 has an overview of these components, but for now, it suffices to know that HBase, a database application, accesses data through HDFS, a distributed file system. MapReduce, in addition to accessing data through HDFS, also accesses the disk directly to perform external sorts and to shuffle data between tasks. We run the following client applications:

| | |
|---|---|
| FSread4m | Random closed-loop 4MB HDFS reads |
| FSread64m | Random closed-loop 64MB HDFS reads |
| Hget | 10kB row lookups in a large HBase table |
| Hscan | 4MB table scans of a large HBase table |
| MRsort10g | MapReduce sort job on 10GB of input data |
| MRsort100g | MapReduce sort job on 100GB of input data |

By default, the systems expose a few metrics for disk consumption, such as disk read throughput aggregated by each HDFS DataNode. To reproduce this metric with Pivot Tracing, we define a *tracepoint* for the `DataNodeMetrics` class, in HDFS, to intercept the `incrBytesRead(int delta)` method. A tracepoint is a location in the application source code where instrumentation can run, cf. Section 3. We then run the following query, in Pivot Tracing's LINQ-like query language [70]:

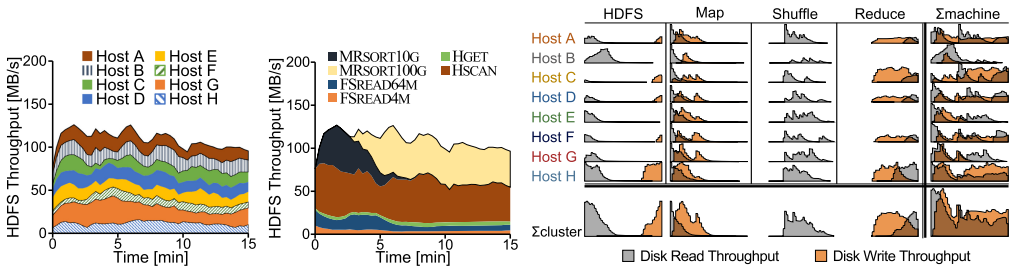
```
Q1: From incr In DataNodeMetrics.incrBytesRead
    GroupBy incr.host
    Select incr.host, SUM(incr.delta)
```

This query causes each machine to aggregate the `delta` argument each time `incrBytesRead` is invoked, grouping by the host name. Each machine reports its local aggregate every second, from which we produce the time series in Figure 1(a).

Things get more interesting, though, if we wish to measure the HDFS usage of each of our client applications. HDFS only has visibility of its direct clients, and thus an aggregate view of all HBase and all MapReduce clients. At best, applications must estimate throughput client side. With Pivot Tracing, we define tracepoints for the client protocols of HDFS (`DataTransferProtocol`), HBase (`ClientService`), and MapReduce (`ApplicationClientProtocol`), and use the name of the client process as the group by key for the query. Figure 1(b) shows the global HDFS read throughput of each client application, produced by the following query:

```
Q2: From incr In DataNodeMetrics.incrBytesRead
    Join cl In First(ClientProtocols) On cl -> incr
    GroupBy cl.procName
    Select cl.procName, SUM(incr.delta)
```

The `->` symbol indicates a happened-before join. Pivot Tracing's implementation will record the process name the first time the request passes through any client protocol method and propagate it




(a) HDFS DataNode throughput per machine from instrumented `DataNodeMetrics`.

(b) HDFS DataNode throughput grouped by high-level client application.

(c) Pivot table showing disk read and write sparklines for MRSORT10G. Rows group by host machine; columns group by source process. Bottom row and right column show totals, and bottom-right corner shows grand total.

Fig. 1. In this example, Pivot Tracing exposes a low-level HDFS metric grouped by client identifiers from other applications. Pivot Tracing can expose arbitrary metrics at one point of the system, while being able to select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries.

along the execution. Then, whenever the execution reaches `incrBytesRead` on a DataNode, Pivot Tracing will emit the bytes read or written, grouped by the recorded name. This query exposes information about client disk throughput that cannot currently be exposed by HDFS.

Figure 1(c) demonstrates the ability for Pivot Tracing to group metrics along arbitrary dimensions. It is generated by two queries similar to `Q2`, which instrument Java's `FileInputStream` and `FileOutputStream`, still joining with the client process name. We show the per-machine, per-application disk read and write throughput of MRSORT10G from the same experiment. This figure resembles a pivot table, where summing across rows yields per-machine totals, summing across columns yields per-system totals, and the bottom right corner shows the global totals. In this example, the client application presents a further dimension along which we could present statistics.

Query `Q1` above is processed locally, while query `Q2` requires the propagation of information from client processes to the data access points. Pivot Tracing's query optimizer installs dynamic instrumentation where needed, and determines when such propagation must occur to process a query. The out-of-the box metrics provided by HDFS, HBase, and MapReduce cannot provide analyses like those presented here. Simple correlations—such as determining *which* HDFS datanodes were read from by a high-level client application—are not typically possible. Metrics are ad hoc between systems; HDFS sums IO bytes, while HBase exposes operations per second. There is very limited support for cross-tier analysis: MapReduce simply counts global HDFS input and output bytes; HBase does not explicitly relate HDFS metrics to HBase operations.

## 2.2 Pivot Tracing Overview

Figure 2 presents a high-level overview of how Pivot Tracing enables queries such as `Q2`. We refer to the numbers in the figure (e.g., ①) in our description. Full support for Pivot Tracing in a system requires two basic mechanisms: dynamic code injection and causal metadata propagation. While it is possible to have some of the benefits of Pivot Tracing without one of these (Section 8), for now we assume both are available.

Queries in Pivot Tracing refer to variables exposed by one or more *tracepoints*—places in the system where Pivot Tracing can insert instrumentation. Tracepoint definitions are not part of the

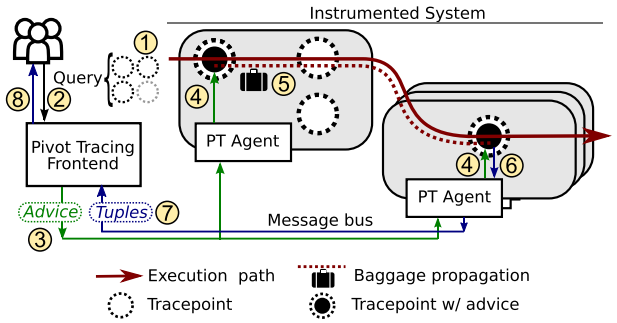



Fig. 2. Pivot Tracing overview (Section 2.2).

system code, but are rather instructions on where and how to change the system to obtain the exported identifiers. Tracepoints in Pivot Tracing are similar to pointcuts from aspect-oriented programming [62] and can refer to arbitrary interface/method signature combinations. Tracepoints are defined by someone with knowledge of the system, maybe a developer or expert operator, and define the vocabulary for queries (①). They can be defined and installed at any point in time and can be shared and disseminated.

Pivot Tracing models system events as tuples of a streaming, distributed dataset. Users submit relational queries over this dataset (②), which get compiled to an intermediate representation called *advice* (③). Advice uses a small instruction set to process queries, and maps directly to code that local Pivot Tracing agents install dynamically at relevant tracepoints (④). Later, requests executing in the system invoke the installed advice each time their execution reaches the tracepoint.

We distinguish Pivot Tracing from prior work by supporting *joins* between events that occur within and across process, machine, and application boundaries. The efficient implementation of the happened before join requires advice in one tracepoint to send information along the execution path to advice in subsequent tracepoints. This is done through a new *baggage* abstraction, which uses causal metadata propagation (⑤). In query Q2, for example, `cl.procName` is packed in the first invocation of the `ClientProtocols` tracepoint, to be accessed when processing the `incrBytesRead` tracepoint.

Advice in some tracepoints also emit tuples (⑥), which get aggregated locally and then finally streamed to the client over a message bus (⑦ and ⑧).

### 2.3 Monitoring and Troubleshooting Challenges

Pivot Tracing addresses two main challenges in monitoring and troubleshooting. First, when the choice of what to record about an execution is made *a priori*, there is an inherent tradeoff between recall and overhead. Second, to diagnose many important problems, one needs to correlate and integrate data that crosses component, system, and machine boundaries. In Section 7, we expand on our discussion of existing work relative to these challenges.

***One Size Does Not Fit All.*** Problems in distributed systems are complex, varied, and unpredictable. By default, the information required to diagnose an issue may not be reported by the system or contained in system logs. Current approaches tie logging and statistics mechanisms into the development path of products, where there is a mismatch between the expectations and incentives of the developer and the needs of operators and users. Panelists at 2011 workshop on managing large scale systems (SLAML'11) [36] discussed the important need to "close the loop of operations back to developers." According to Yuan et al. [106], regarding diagnosing failures, "*(. . . ) existing*

*log messages contain too little information. Despite their widespread use in failure diagnosis, it is still rare that log messages are systematically designed to support this function.*"

This mismatch can be observed in the many issues raised by users on Apache's issue trackers: to request new metrics [5, 7, 10–12, 20, 25]; to request changes to aggregation methods [13, 19, 21]; and to request new breakdowns of existing metrics [4, 8, 9, 14–19, 23, 24, 27, 28, 30]. Many issues remain unresolved due to developer pushback [15, 18, 24, 25, 28] or inertia [8, 10, 11, 20, 21, 23, 27, 30]. Even simple cases of misconfiguration are frequently unreported by error logs [105].

Eventually, applications may be updated to record more information, but this has effects both in performance and information overload. Users must pay the performance overheads of any systems that are enabled by default, regardless of their utility. For example, HBase SchemaMetrics were introduced to aid developers, but all users of HBase pay the 10% performance overhead they incur [19]. The HBase user guide [22] carries the following warning for users wishing to integrate with Ganglia [69]: "*By default, HBase emits a large number of metrics per region server. Ganglia may have difficulty processing all these metrics. Consider increasing the capacity of the Ganglia server or reducing the number of metrics emitted by HBase.*"

The glut of recorded information presents a "needle-in-a-haystack" problem to users [84]; while a system may expose information relevant to a problem, e.g., in a log, extracting this information requires system familiarity developed over a long period of time. For example, Mesos cluster state is exposed via a single JSON endpoint and can become massive, even if a client only wants information for a subset of the state [31].

Dynamic instrumentation frameworks such as Fay [54], DTrace [38], and SystemTap [83] address these limitations, by allowing almost arbitrary instrumentation to be installed dynamically at runtime, and have proven extremely useful in the diagnosis of complex and subtle system problems [37]. Because of their side-effect-free nature, however, they are limited in the extent to which probes may share information with each other. In Fay, only probes in the same address space can share information, while in DTrace, the scope is limited to a single operating system instance.

***Crossing Boundaries.*** This brings us to the second challenge Pivot Tracing addresses. In multi-tenant, multi-application stacks, the root cause and symptoms of an issue may appear in different processes, machines, and application tiers, and may be visible to different users. A user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. For example, HBASE-4145 [16] outlines how MapReduce lacks the ability to access HBase metrics on a per-task basis, and that the framework only returns aggregates across all tasks. MESOS-1949 [30] outlines how the executors for a task do not propagate failure information, so diagnosis can be difficult if an executor fails. In discussion, the developers note: "*The actually interesting / useful information is hidden in one of four or five different places, potentially spread across as many different machines. This leads to unpleasant and repetitive searching through logs looking for a clue to what went wrong. (…) There's a lot of information that is hidden in log files and is very hard to correlate.*"

Prior research has presented mechanisms to observe or infer the relationship between events (Section 7) and studies of logging practices conclude that end-to-end tracing would be helpful in navigating the logging issues they outline [77, 84]. A variety of these mechanisms have also materialized in production systems: for example, Google's Dapper [93], Apache's HTrace [29], Accumulo's Cloudtrace [3], and Twitter's Zipkin [96]. These approaches can obtain richer information about particular executions than component-centric logs or metrics alone, and have found uses in troubleshooting, debugging, performance analysis, and anomaly detection, for example. However, most of these systems record or reconstruct traces of execution for offline analysis, and thus share the problems above with the first challenge, concerning what to record.

## 3 DESIGN

We now detail the fundamental concepts and mechanisms behind Pivot Tracing. Pivot Tracing is a dynamic monitoring and tracing framework for distributed systems. At a high level, it aims to enable flexible runtime monitoring by correlating metrics and events from arbitrary points in the system. The challenges outlined in Section 2 motivate the following high-level design goals:

(1) Dynamically configure and install monitoring at runtime
(2) Low system overhead to enable "always on" monitoring
(3) Capture causality between events from multiple processes and applications

### 3.1 Tracepoints

Tracepoints provide the system-level entry point for Pivot Tracing queries. A tracepoint typically corresponds to some event: a client sends a request; a low-level IO operation completes; an external RPC is invoked, and so on.

A tracepoint identifies one or more locations in the system code where Pivot Tracing can install and run instrumentation. Tracepoints export named variables that can be accessed by instrumentation. Figure 5 shows the specification of one of the tracepoints in `Q2` from Section 2. Besides declared exports, all tracepoints export a few variables by default: host, timestamp, process ID, process name, and the tracepoint definition.

Tracepoints are only references to locations where Pivot Tracing can install instrumentation—they are not baked into the system and they do not require *a priori* modifications. Only at runtime, when a user submits a query, will Pivot Tracing compile and install monitoring code at tracepoints referenced by the query. Subsequently, when requests running in the system reach a tracepoint, any instrumentation configured for that tracepoint will be invoked, generating a tuple with its exported variables. These are then accessible to any instrumentation code installed at the tracepoint.

### 3.2 Query Language

Pivot Tracing enables users to express high-level queries about the variables exported by one or more tracepoints. We abstract tracepoint invocations as streaming datasets of tuples; Pivot Tracing queries are therefore relational queries across the tuples of several such datasets.

To express queries, Pivot Tracing provides a parser for LINQ-like text queries such as those outlined in Section 2. Table 1 outlines the query operations supported by Pivot Tracing. Pivot Tracing supports several typical operations including projection ($\Pi$), selection ($\sigma$), renaming ($\rho$), grouping (`G`), and aggregation (`A`). Pivot Tracing aggregators include `Count`, `Sum`, `Max`, `Min`, and `Average`. Pivot Tracing also defines the temporal filters `MostRecent`, `MostRecentN`, `First`, and `FirstN`, to take the 1 or `N` most or least recent events. Finally, Pivot Tracing introduces the *happened-before join* query operator ($\overrightarrow{\bowtie}$).

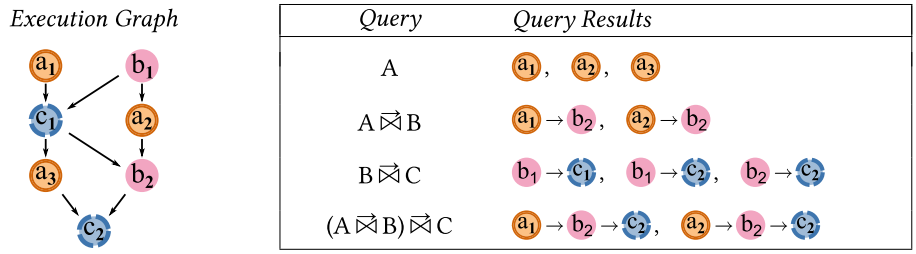
### 3.3 Happened-before Joins

A key contribution of Pivot Tracing is the happened-before join query operator. Happened-before join enables the tuples from two Pivot Tracing queries to be joined based on Lamport's happened before relation, $\rightarrow$ [65]. For events $a$ and $b$ occurring anywhere in the system, we say that $a$ happened before $b$ and write $a \rightarrow b$ if the occurrence of event $a$ causally preceded the occurrence of event $b$ and they occurred as part of the execution of the same request.[1] If $a$ and $b$ are not part of

[1]This definition does not capture all possible causality, including when events in the processing of one request could influence another, but could be extended if necessary.

Table 1. Operations Supported by the Pivot Tracing Query Language

| *Operation* | *Description* | *Example* |
|---|---|---|
| From | Use input tuples from a set of tracepoints | `From e In RPCs` |
| Union ($\bigcup$) | Union events from multiple tracepoints | `From e In DataRPCs, ControlRPCs` |
| Selection ($\sigma$) | Filter only tuples that match a predicate | `Where e.Size < 10` |
| Rename ($\rho$) | Create a field using an expression | `Let SizeKB = e.Size / 1000` |
| Projection ($\Pi$) | Restrict tuples to a subset of fields | `Select e.User, e.Host` |
| Aggregation (A) | Aggregate tuples | `Select SUM(e.Cost)` |
| GroupBy (G) | Group tuples based on one or more fields | `GroupBy e.User` |
| GroupBy Aggregation (GA) | Aggregate tuples of a group | `Select e.User, SUM(e.Cost)` |
| Happened-Before Join ($\overrightarrow{\bowtie}$) | Happened-before join tuples from another query | `Join d In Disk On d -> e` |
| | Happened-before join a subset of tuples | `Join d In MostRecent(Disk) On d -> e` |




(a) A request that triggers tracepoints A, B and C several times during its execution. Each invocation of a tracepoint produces a tuple, *e.g.* tracepoint A generates $a_1$, $a_2$, $a_3$, etc.

(b) Queries and their corresponding results for the execution in Figure 3a. A query for a single tracepoint (*e.g.* A) gives all tuples produced by that tracepoint. A happened-before join between two tracepoints (*e.g.* A $\overrightarrow{\bowtie}$ B) gives all pairs of tuples satisfying that happened-before relationship.

Fig. 3. An example request execution graph and the results of running queries over that request.

the same execution, then $a \not\rightarrow b$; if the occurrence of $a$ did not lead to the occurrence of $b$, then $a \not\rightarrow b$ (e.g., they occur in two parallel threads of execution that do not communicate); and if $a \rightarrow b$, then $b \not\rightarrow a$.

In general, events occurring during a request's execution will form a *directed, acyclic graph* (DAG) under the happened-before relation. Figure 3 illustrates an example DAG. Events occurring concurrently in different threads, processes, or machines do not satisfy the happened-before relation (e.g., $a_1$ and $b_1$). However, when there is communication between concurrent components, that communication will establish the happened-before relationship with later events (e.g., $a_1 \rightarrow c_1$ and $b_1 \rightarrow c_1$).

The happened-before join operator enables queries about the relationships between events. For any two queries $Q_1$ and $Q_2$, the happened-before join $Q_1 \overrightarrow{\bowtie} Q_2$ produces tuples $t_1 t_2$ for all $t_1 \in Q_1$ and $t_2 \in Q_2$ such that $t_1 \rightarrow t_2$. That is, $Q_1$ produced $t_1$ before $Q_2$ produced tuple $t_2$ in the execution of the same request. Figure 3 shows an example execution triggering tracepoints A, B, and C several times, and outlines the tuples that would be produced for this execution by different queries.

Table 2. Primitive Operations Supported by Pivot Tracing Advice for Generating and Aggregating Tuples

| *Operation* | *Description* |
|---|---|
| Observe | Construct a tuple from variables exported by a tracepoint |
| Unpack | Retrieve one or more tuples from prior advice |
| Filter | Evaluate a predicate on all tuples |
| Pack | Make tuples available for use by later advice |
| Emit | Output a tuple for global aggregation |

Query Q2 in Section 2 demonstrates the use of happened-before join. In the query, tuples generated by the disk IO tracepoint `DataNodeMetrics.incrBytesRead` are joined to the first tuple generated by the `ClientProtocols` tracepoint.

Happened-before join substantially improves our ability to perform root cause analysis by giving us visibility into the relationships *between* events in the system. The happened-before relationship is fundamental to a number of prior approaches in root cause analysis (Section 7). Pivot Tracing is designed to efficiently support happened-before joins but does not optimize more general joins such as equijoins ($\bowtie$).

### 3.4 Advice

Pivot Tracing queries compile to an intermediate representation called *advice*. Advice specifies the operations to perform at each tracepoint used in a query, and eventually materializes as monitoring code installed at those tracepoints (Section 5). Advice has several operations for manipulating tuples through the tracepoint-exported variables and evaluating $\overrightarrow{\bowtie}$ on tuples produced by other advice at prior tracepoints in the execution.

Table 2 outlines the advice API. Observe creates a tuple from exported tracepoint variables. Unpack retrieves tuples generated by other advice at other tracepoints prior in the execution. Unpacked tuples can be joined to the observed tuple, i.e., if $t_o$ is observed and $t_{u1}$ and $t_{u2}$ are unpacked, then the resulting tuples are $t_o t_{u1}$ and $t_o t_{u2}$. Tuples created by this advice can be discarded (Filter), made available to advice at other tracepoints later in the execution (Pack), or output for global aggregation (Emit). Both Pack and Emit can group tuples based on matching fields, and perform simple aggregations such as `SUM` and `COUNT`. Pack also has the following special cases: `FIRST` packs the first tuple encountered and ignores subsequent tuples; `RECENT` packs only the most recent tuple, overwriting existing tuples. `FIRSTN` and `RECENTN` generalize this to `N` tuples. The advice API is expressive but restricted enough to provide some safety guarantees. In particular, advice code has no jumps or recursion and is guaranteed to terminate.

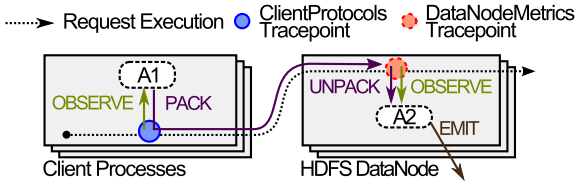
Figure 4 outlines the advice generated for query Q2 from Section 2 and illustrates how the advice and tracepoints interact with the execution of requests in the system. First, `A1` observes and packs a single valued tuple containing the process name. Then, when execution reaches the `DataNodeMetrics` tracepoint, `A2` unpacks the process name, observes the value of `delta`, then emits a joined tuple. Figure 4 shows how this advice and the tracepoints interact with the execution of requests in the system.

To compile a query to advice, we instantiate one advice specification for a `From` clause and add an Observe operation for the tracepoint variables used in the query. For each `Join` clause, we add an Unpack operation for the variables that originate from the joined query. We recursively generate advice for the joined query and append a Pack operation at the end of its advice for the variables that we unpacked. `Where` directly translates to a Filter operation. We add an Emit operation for the output variables of the query, restricted according to any `Select` clause. `Aggregate`,

```
A1: OBSERVE procName
    PACK-FIRST procName

A2: OBSERVE delta
    UNPACK procName
    EMIT procName, SUM(delta)
```

(a) Advice generated for Q2. A1 is dynamically installed at the ClientProtocols tracepoint, and A2 at the DataNodeMetrics tracepoint.




(b) When requests pass through ClientProtocols they invoke A1. A1 observes and packs procName to be carried with the request. When requests subsequently reach DataNodeMetrics, A2 unpacks procName, observes delta, and emits (procName, SUM(delta))

Fig. 4. Pivot Tracing evaluates query Q2 from Section 2 by compiling it into two *advice* specifications. Pivot Tracing dynamically installs the advice at the tracepoints referenced in the query.

GroupBy, and GroupByAggregate are all handled by Emit and Pack. Section 4 outlines several query rewriting optimizations for implementing $\overrightarrow{\bowtie}$.

Pivot Tracing *weaves* advice into tracepoints by: (1) loading code that implements the advice operations; (2) configuring the tracepoint to execute that code and pass its exported variables; (3) activating the necessary tracepoint at all locations in the system. Figure 5 outlines this process of weaving advice for Q2.

## 4 PIVOT TRACING OPTIMIZATIONS

In this section, we outline several optimizations that Pivot Tracing performs in order to support efficient evaluation of happened-before joins.

### 4.1 Baggage

The naïve evaluation strategy for happened-before joins is that of an equijoin ($\bowtie$) or $\theta$-join ($\bowtie_\theta$ [85]), requiring tuples to be aggregated globally across the cluster prior to evaluating the join. Temporal joins as implemented by Magpie [33], for example, are expensive because they implement this evaluation strategy (Section 7). Figure 7(a) illustrates this approach for happened-before joins.

Instead of a naïve global join, Pivot Tracing enables inexpensive happened-before joins by providing the *baggage* abstraction. Baggage is a per-request container for tuples that is propagated alongside a request as it traverses thread, application, and machine boundaries. Figure 6 illustrates baggage propagation for a request. Each branch of the request's execution maintains its own baggage instance. To propagate baggage, instances are copied when executions split into concurrent branches; merged when concurrent branches join; and included with all of the request's interprocess and inter-thread communication.[2] Pivot Tracing advice uses the Pack and Unpack operations to store and retrieve tuples from the current request's baggage.

Baggage explicitly follows the execution path of each request, and thereby observes all happened-before relationships of a request while it is executing. Tuples that are packed during a request's execution will follow the request from that point onward. Consequently, the presence of the tuples in baggage later during the request's execution imply a happened-before relationship.

Baggage is a generalization of end-to-end metadata propagation techniques outlined in prior work such as X-Trace [55] and Dapper [93]. Using baggage, Pivot Tracing efficiently evaluates

[2]Section 5.5 describes how baggage is propagated consistently through concurrent executions.

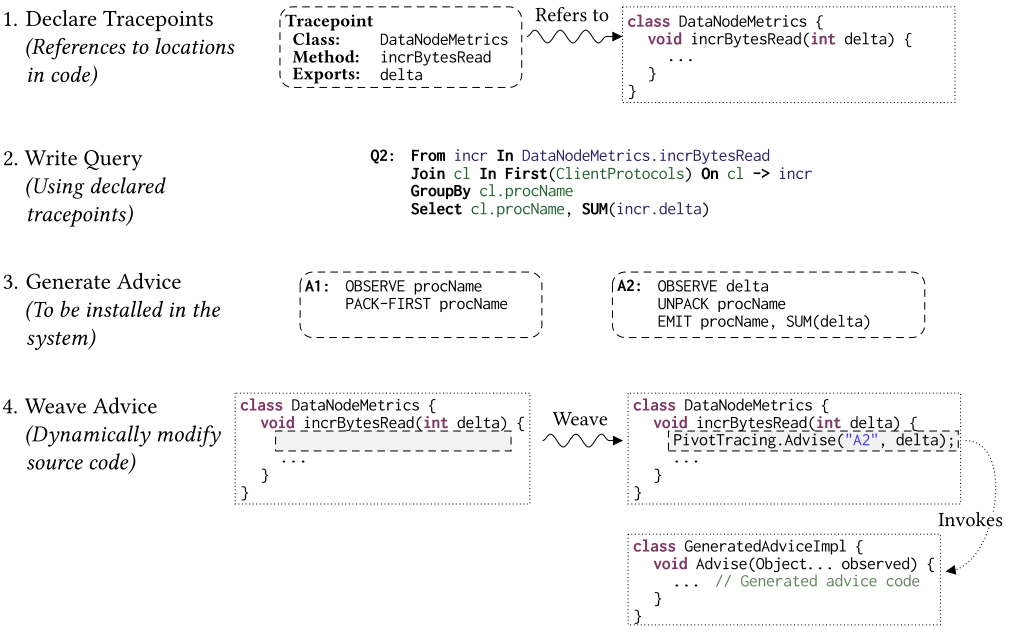



Fig. 5. Steps to install a Pivot Tracing query. Tracepoints are only references to locations in code and require no system-level modifications until queries are installed. Step 4 illustrates how we *weave* advice for Q2 at the `DataNodeMetrics` tracepoint (Q2 also weaves advice at `ClientProtocols`, not shown). Variables exported by the tracepoint (i.e., `delta`) are passed when the advice is invoked.

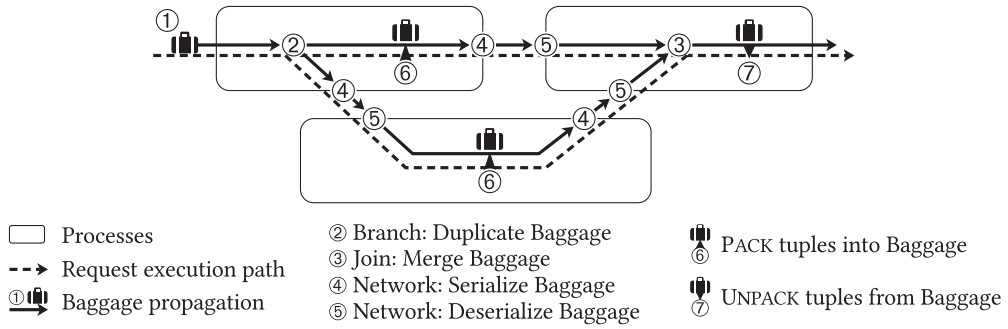
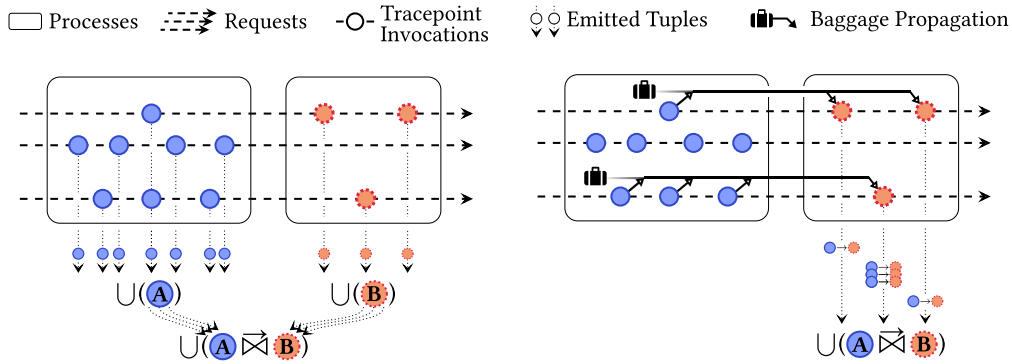



Fig. 6. To implement the happened-before join, systems propagate baggage (①) along the execution path of requests. Baggage is duplicated when requests split into concurrent branches (②); merged when concurrent branches join (③); and included in all inter-process communication (④ ⑤). Pivot Tracing queries PACK tuples into the baggage at some tracepoints (⑥) and UNPACK tuples at other tracepoints (⑦).

happened-before joins *in situ* during the execution of a request. Figure 7(b) shows the optimized query evaluation strategy to evaluate joins in-place during request execution.

### 4.2 Local Tuple Aggregation

One metric to assess the cost of a Pivot Tracing query is the number of tuples emitted for global aggregation. Although baggage carries partial query state at a per-request granularity, once




(a) Unoptimized query with all tuples collected centrally across the cluster prior to evaluation of $\overrightarrow{\bowtie}$.

(b) Optimized query that propagates tuples in baggage, enabling inline evaluation of $\overrightarrow{\bowtie}$.

Fig. 7. Illustration comparing the optimized and unoptimized evaluation of A $\overrightarrow{\bowtie}$ B. The optimized query evaluates $\overrightarrow{\bowtie}$ inline by propagating tuples in baggage, avoiding a costly global aggregation.

Table 3. Query Rewrite Rules for Happened-before Join between Two Queries P and Q

| *Query* | *Optimized Query* |
|---|---|
| $\Pi_{p,q}(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$ | $\Pi_p(\mathrm{P}) \overrightarrow{\bowtie} \Pi_q(\mathrm{Q})$ |
| $\sigma_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$ | $\sigma_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q}$ |
| $\sigma_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$ | $\mathrm{P} \overrightarrow{\bowtie} \sigma_q(\mathrm{Q})$ |
| $\mathtt{A}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$ | $\mathtt{Combine}_p(\mathtt{A}_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q})$ |

| *Query* | *Optimized Query* |
|---|---|
| $\mathtt{GA}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$ | $\mathtt{G}_p\mathtt{Combine}_p(\mathtt{GA}_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q})$ |
| $\mathtt{GA}_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$ | $\mathtt{G}_q\mathtt{Combine}_p(\mathrm{P} \overrightarrow{\bowtie} \mathtt{GA}_q(\mathrm{Q}))$ |
| $\mathtt{G}_p\mathtt{A}_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$ | $\mathtt{G}_p\mathtt{Combine}_q(\Pi_p(\mathrm{P}) \overrightarrow{\bowtie} \mathtt{A}_q(\mathrm{Q}))$ |
| $\mathtt{G}_q\mathtt{A}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$ | $\mathtt{G}_q\mathtt{Combine}_p(\mathtt{A}_p(\mathrm{P}) \overrightarrow{\bowtie} \Pi_q(\mathrm{Q}))$ |

Optimizations push operators as close as possible to source tuples, thereby reducing the tuples that must be propagated in baggage from P to Q. `Combine` refers to an aggregator's combiner function (*e.g.*, for `Count`, the combiner is `Sum`). See Table 1 for descriptions of query operators.

tuples have been emitted for global aggregation, Pivot Tracing collects and aggregates these tuples across all requests to produce the final query results. To reduce this cost, Pivot Tracing performs intermediate aggregation for queries containing `Aggregate` or `GroupByAggregate`. Pivot Tracing aggregates the emitted tuples within each process and reports results globally at a regular interval, e.g., once per second. Process-level aggregation substantially reduces traffic for emitted tuples; `Q2` from Section 2 is reduced from approximately 600 tuples per second to 6 tuples per second from each DataNode.

### 4.3 Optimizing Happened-Before Joins

A second cost metric for Pivot Tracing queries is the number of tuples packed during a request's execution and carried within the request's baggage. Pivot Tracing rewrites queries to minimize the number of tuples packed. Pivot Tracing pushes projection, selection, and aggregation terms as close as possible to source tracepoints. In Fay [54], the authors outlined query optimizations for merging streams of tuples, enabled because projection, selection, and aggregation are distributive. These optimizations also apply to Pivot Tracing and reduce the number of tuples emitted for global aggregation. To reduce the number of tuples transported in the baggage, Pivot Tracing adds further optimizations for happened-before joins, outlined in Table 3.

### 4.4 Cost of Baggage Propagation

Pivot Tracing does not inherently bound the number of packed tuples and potentially accumulates a new tuple for every tracepoint invocation. However, we liken this to database queries that inherently risk a full table scan—our optimizations mean that, in practice, this is an unlikely event. Several of Pivot Tracing's aggregation operators explicitly restrict the number of propagated tuples and, in our experience, queries only end up propagating aggregations, most-recent, or first tuples.

In cases where too many tuples are packed in the baggage, Pivot Tracing could revert to an alternative query plan, where all tuples are emitted instead of packed, and the baggage size is kept constant by storing only enough information to reconstruct the causality, *a la* X-Trace [55], Stardust [95], or Dapper [93]. To estimate the overhead of queries, Pivot Tracing can execute a modified version of the query to count tuples rather than aggregate them explicitly. This would provide live analysis similar to "explain" queries in the database domain.

## 5 IMPLEMENTATION

We have implemented a Pivot Tracing prototype in Java and applied Pivot Tracing to several open-source systems from the Hadoop ecosystem. Section 6 outlines our instrumentation of these systems. In this section, we describe the implementation of our prototype.

We opted to implement and evaluate Pivot Tracing in Java in order to make use of several existing open-source distributed systems written in this language. However, the components of Pivot Tracing generalize and are not restricted to Java—a query could span multiple systems written in different programming languages.

### 5.1 Pivot Tracing Agent

A Pivot Tracing agent thread runs in every Pivot Tracing-enabled process and awaits instruction via central pub/sub server to weave advice to tracepoints. Tuples emitted by advice are accumulated by the local Pivot Tracing agent, which performs partial aggregation of tuples according to their source query. Agents publish partial query results at a configurable interval—by default, one second.

### 5.2 Dynamic Instrumentation

Our prototype weaves advice at runtime, providing dynamic instrumentation similar to that of DTrace [38] and Fay [54]. Java version 1.5 onward supports dynamic method body rewriting via the `java.lang.instrument` package. The Pivot Tracing agent programmatically rewrites and reloads class bytecode from within the process using Javassist [44].

We can define new tracepoints at runtime and dynamically weave and unweave advice. To weave advice, we rewrite method bodies to add advice invocations at the locations defined by the tracepoint (cf. Figure 5). Our prototype supports tracepoints at the entry, exit, or exceptional return of any method. Tracepoints can also be inserted at specific line numbers.

To define a tracepoint, users specify a class name, method name, method signature, and weave location. Pivot Tracing also supports pattern matching, for example, all methods of an interface on a class. This feature is modeled after *pointcuts* from AspectJ [61]. Pivot Tracing supports instrumenting privileged classes (e.g., `FileInputStream` in Section 2) by providing an optional agent that can be placed on Java's boot classpath.

Pivot Tracing only makes system modifications when advice is woven into a tracepoint, so inactive tracepoints incur no overhead. Executions that do not trigger the tracepoint are unaffected

Table 4. Baggage API for Pivot Tracing Java Implementation

| *Method* | *Description* |
|---|---|
| `pack(q, t...)` | Pack tuples into the baggage for a query |
| `unpack(q)` | Retrieve all tuples for a query |
| `serialize()` | Serialize the baggage to bytes |
| `deserialize(b)` | Set the baggage by deserializing from bytes |
| `split()` | Split the baggage for a branching execution |
| `join(b1, b2)` | Merge baggage from two joining executions |

PACK operations store tuples in the baggage. API methods are static and only allow interaction with the current execution's baggage.

by Pivot Tracing. Pivot Tracing has a zero-probe effect: methods are unmodified by default, so tracepoints impose truly zero overhead until advice is woven into them.

### 5.3 Baggage

We provide an implementation of Baggage that stores per-request instances in thread-local variables. At the beginning of a request, we instantiate empty baggage in the thread-local variable; at the end of the request, we clear the baggage from the thread-local variable. The baggage API (Table 4) can get or set tuples for a query, and at any point in time, baggage can be retrieved for propagation to another thread or serialization onto the network. To support multiple queries simultaneously, queries are assigned unique IDs, and tuples are packed and unpacked based on this ID.

Baggage is lazily serialized and deserialized using protocol buffers [98]. This minimizes the overhead of propagating baggage through applications that do not actively participate in a query, since baggage is only deserialized when an application attempts to pack or unpack tuples. Serialization costs are only incurred for modified baggage at network or application boundaries.

Pivot Tracing relies on developers to implement Baggage propagation when a request crosses thread, process, or asynchronous execution boundaries. In our experience (Section 6), this entails adding a baggage field to existing application-level request contexts and RPC headers.

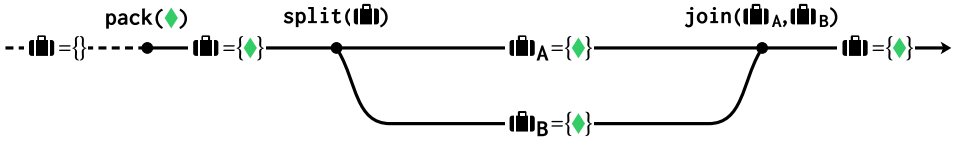
### 5.4 Materializing Advice

Tracepoints with woven advice invoke the `PivotTracing.Advise` method (cf. Figure 5), passing tracepoint variables as arguments. Tuples are implemented as `Object` arrays and there is a straightforward translation from advice to implementation: OBSERVE constructs a tuple from the advice arguments; UNPACK and PACK interact with the Baggage API; FILTER discards tuples that do not match a predicate; and EMIT forwards tuples to the process-local aggregator.

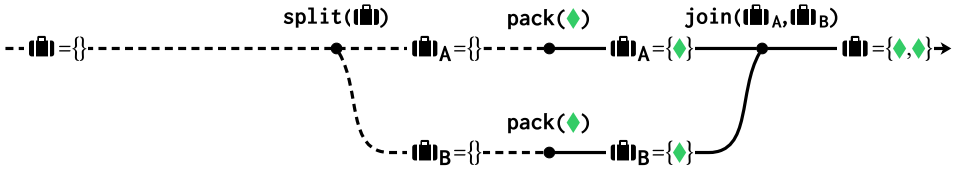
### 5.5 Baggage Consistency

In order to preserve the happened-before relation correctly within a request, Pivot Tracing must handle executions that branch and rejoin, as illustrated in Figure 6. Each branch of the request's execution maintains its own baggage instance. To propagate baggage, instances are copied when executions split into concurrent branches, merged when concurrent branches join, and included with all of the request's inter-process and inter-thread communication. Tuples packed by one execution branch are not visible to any other branch until the branches rejoin or communicate.

In executions that arbitrarily branch and join, we must be careful not to inadvertently duplicate tuples when merging baggage instances from concurrent execution branches. It is possible for both baggage instances to contain the same tuple derived from the same source event, as illustrated in




(a) One tuple (◆) is packed prior to the execution branching. When the execution splits, the tuple exists in both 🧳A and 🧳B. When the execution joins, the tuple must only be retained once, because it only captures one happened-before relationship.




(b) Baggage is empty when the execution branches. Both branches of execution encounter distinct events that pack a tuple; however, the tuples happen to have the same value (both ◆). When the execution joins, the two identical tuples must both be retained, because they capture different happened-before relationships.

Fig. 8. An execution that branches then joins. Baggage is split into 🧳A and 🧳B, then later joined. In both examples, 🧳A and 🧳B contain the same tuples; however, the merge result is different in each case because the tuples represent different happened-before relationships. Baggage must be consistently joined in order to correctly reflect the happened-before relationship represented by the tuples (cf. Section 5.5).

Figure 8(a). In this case, our merge function must not naïvely duplicate the tuple—we must ensure that the output baggage only contains the tuple once to correctly reflect that there is only one happened-before relationship. On the other hand, it is also possible for each execution branch to independently pack tuples with the same values, as illustrated in Figure 8(b). In this case, we expect the merged baggage to contain both tuples, because they represent independent events and we must preserve both happened-before relationships.

To correctly preserve happened-before relationships for executions that arbitrarily branch and join, we implement baggage as a *set*: when a request branches, the baggage tuples are simply duplicated for each branch; when two or more branches join, we perform a *set union* on the tuples present in each baggage instance. To disambiguate between the two cases illustrated in Figure 8, we append a unique identifier to each tuple when it is initially packed. Consequently, when merging two baggage instances, set union will not duplicate tuples if they are the same tuple derived from the same source event; conversely, if two distinct events produced tuples with the same value, they will be differentiated by different IDs.

We extend this concept further to handle the query optimization rules described in Section 4.3, which push projection, selection, and aggregation terms as close as possible to source tuples. For some queries, these optimizations lead to advice that directly aggregates tuples in baggage, for example, by summing values immediately as tuples are packed, and propagating only the sum in baggage. However, for the same reason as illustrated in Figure 8, it is insufficient to only propagate a running total in baggage, as it can lead to double counting when merging baggage instances. Instead, we require some additional causality information to determine how to merge two sums.

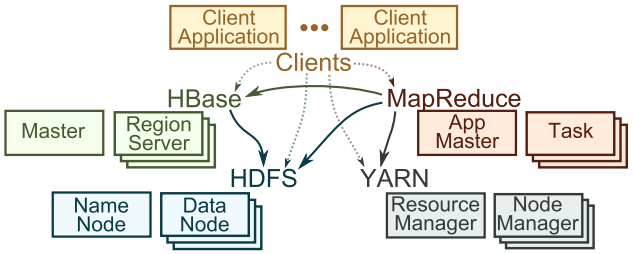



Fig. 9. Interactions between systems. Each system comprises several processes on potentially many machines. Typical deployments often co-locate processes from several applications, e.g., DataNode, NodeManager, and Task and RegionServer processes.

To handle this, we extend the previously described tuple ID scheme. Internal to a baggage instance, we maintain one or more *bags*, each of which has a unique bag ID and sequence number. We store tuples, aggregations, and the like within each bag naïvely without any additional embellishments. Each branch of execution considers one of the bags to be the "active" bag, and it packs and aggregates tuples into only that bag. After an execution branches, one of the branches continues using the previous active bag and increments the bag's sequence number; the other branch lazily instantiates a new bag the next time it needs to pack tuples. When multiple branches join, the bag with the highest sequence number for each bag ID is retained.

Under this scheme, a baggage instance resembles a version vector [82] with a variable number of components, for which bags correspond to components. In the original version of Pivot Tracing, we generated bag IDs using interval tree clocks [2], which provided a mechanism for splitting IDs when executions branch, and recombining IDs when branches rejoin. This scheme guaranteed unique bag IDs, but required maintaining interval tree IDs even if nothing was propagated in the baggage. In our most recent Pivot Tracing implementation, we have switched to random bag IDs to eliminate the overhead of maintaining interval tree IDs. However, this now introduces a small chance of ID collisions, which can occur when two branches of the same request generate the same random ID concurrently. Our evaluation in Section 6 uses Pivot Tracing with the original interval tree clock scheme.

## 6 EVALUATION

In this section, we evaluate Pivot Tracing in the context of the Hadoop stack. We have instrumented four open-source systems—HDFS, HBase, Hadoop MapReduce, and YARN—that are widely used in production today. We present several case studies where we used Pivot Tracing to successfully diagnose root causes, including real-world issues we encountered in our cluster and experiments presented in prior work [66, 101]. Our evaluation shows that Pivot Tracing addresses the challenges in Section 2 when applied to these stack components. In particular, we show that Pivot Tracing:

- —is dynamic and extensible to new kinds of analysis (Section 6.2)
- —is scalable and has low developer and execution overhead (Section 6.3)
- —enables cross-tier analysis between any inter-operating applications (Section 2, Section 6.2)
- —captures event causality to successfully diagnose root causes (Section 6.1, Section 6.2)
- —enables insightful analysis with even a very small number of tracepoints (Section 6.1)

***Hadoop Overview.*** We first give a high-level overview of Hadoop, before describing the necessary modifications to enable Pivot Tracing. Figure 9 shows the relevant components of the Hadoop stack.

HDFS [92] is a distributed file system that consists of several DataNodes that store replicated file blocks and a NameNode that manages the filesystem metadata.

HBase [6] is a non-relational database modeled after Google's Bigtable [41] that runs on top of HDFS and comprises a Master and several RegionServer processes.

Hadoop MapReduce is an implementation of the MapReduce programming model [49] for large-scale data processing that uses YARN containers to run map and reduce tasks. Each job runs an ApplicationMaster and several MapTask and ReduceTask containers.

YARN [99] is a container manager to run user-provided processes across the cluster. NodeManager processes run on each machine to manage local containers, and a centralized ResourceManager manages the overall cluster state and requests from users.

***Hadoop Instrumentation.*** In order to support Pivot Tracing in these systems, we made one-time modifications to propagate baggage along the execution path of requests. As described in Section 5, our prototype uses a thread-local variable to store baggage during execution, so the only required system modifications are to set and unset baggage at execution boundaries. To propagate baggage across remote procedure calls, we manually extended the protocol definitions of the systems. To propagate baggage across execution boundaries within individual processes, we implemented AspectJ [61] instrumentation to automatically modify common interfaces (`Thread`, `Runnable`, `Callable`, and `Queue`). Each system only required between 50 and 200 lines of manual code modification. Once modified, these systems could support arbitrary Pivot Tracing queries without further modification.

Our queries used tracepoints from both client and server RPC protocol implementations of the HDFS DataNode `DataTransferProtocol` and NameNode `ClientProtocol`. We also used tracepoints for piggybacking off existing metric collection mechanisms in each instrumented system, such as `DataNodeMetrics` and `RPCMetrics` in HDFS and `MetricsRegionServer` in HBase.

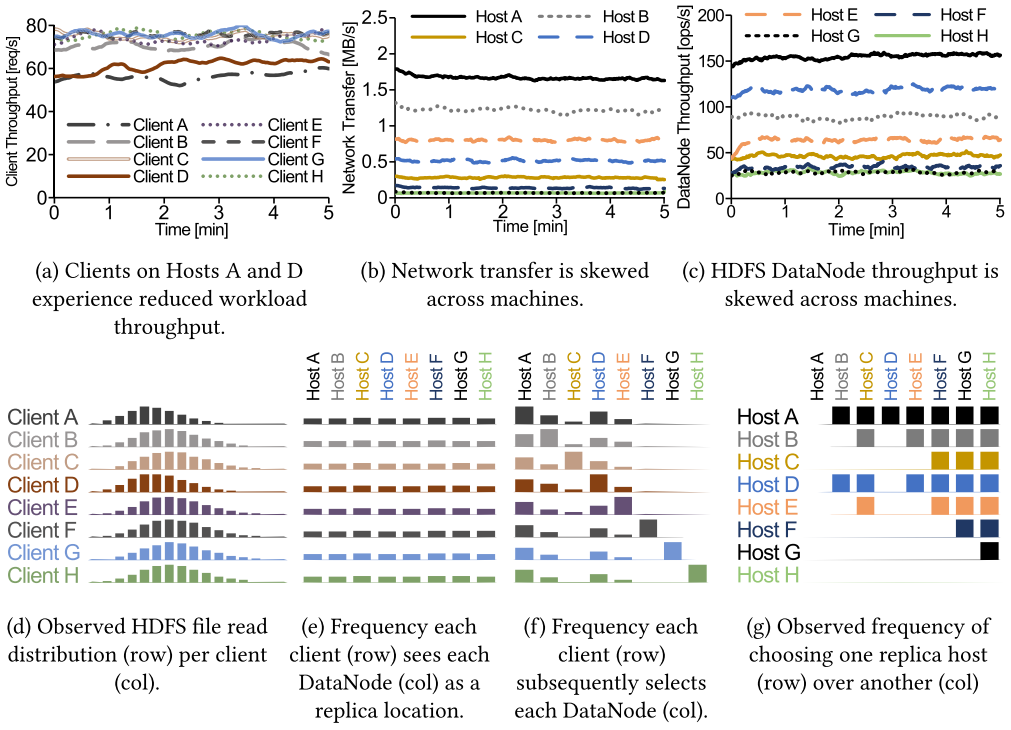
### 6.1 Case Study: HDFS Replica Selection Bug

In this section, we describe our discovery of a replica selection bug in HDFS that resulted in uneven distribution of load to replicas. After identifying the bug, we found that it had been recently reported and subsequently fixed in an upcoming HDFS version [26].

HDFS provides file redundancy by decomposing files into blocks and replicating each block onto several machines (typically three). A client can read any replica of a block and does so by first contacting the NameNode to find replica hosts (`GetBlockLocations`), then selecting the closest replica as follows: (1) read a local replica; (2) read a rack-local replica; (3) select a replica at random. We discovered a bug whereby rack-local replica selection always follows a global static ordering due to two conflicting behaviors: the HDFS client does not randomly select between replicas; and the HDFS NameNode does not randomize rack-local replicas returned to the client. The bug results in heavy load on some hosts and near zero load on others.

In this scenario, we ran 96 stress test clients on an HDFS cluster of eight DataNodes and one NameNode. Each machine has identical hardware specifications; 8 cores, 16GB RAM, and a 1Gbit network interface. On each host, we ran a process called StressTest that used an HDFS client to perform closed-loop random 8kB reads from a dataset of 10,000 128MB files with a replication factor of 3.

Our investigation of the bug began when we noticed that the stress test clients on hosts A and D had consistently lower request throughput than clients on other hosts, shown in Figure 10(a), despite identical machine specifications and setup. We first checked machine level resource utilization on each host, which indicated substantial variation in the network throughput (Figure 10(b)). We began our diagnosis with Pivot Tracing by first checking to see whether an imbalance in HDFS




(a) Clients on Hosts A and D experience reduced workload throughput.

(b) Network transfer is skewed across machines.

(c) HDFS DataNode throughput is skewed across machines.

(d) Observed HDFS file read distribution (row) per client (col).

(e) Frequency each client (row) sees each DataNode (col) as a replica location.

(f) Frequency each client (row) subsequently selects each DataNode (col).

(g) Observed frequency of choosing one replica host (row) over another (col)

Fig. 10. Pivot Tracing query results leading to our discovery of HDFS-6268 [26]. Faulty replica selection logic led clients to prioritize the replicas hosted by particular DataNodes (Section 6.1).

load was causing the variation in network throughput. The following query installs advice at a DataNode tracepoint that is invoked by each incoming RPC:

```
Q3: From dnop In DataNode.DataTransferProtocol
    GroupBy dnop.host
    Select dnop.host, COUNT
```

Figure 10(c) plots the results of this query, showing the HDFS request throughput on each DataNode. It shows that DataNodes on hosts A and D in particular have substantially higher request throughput than others—host A has on average 150ops/sec, while host H has only 25ops/sec. This behavior was unexpected given that our stress test clients are supposedly reading files uniformly at random. Our next query installs advice in the stress test clients and on the HDFS NameNode to correlate each read request with the client that issued it:

```
Q4: From getloc In NameNode.GetBlockLocations
    Join st In StressTestClient.DoNextOp On st -> getloc
    GroupBy st.host, getloc.src
    Select st.host, getloc.src, COUNT
```

This query counts the number of times each client reads each file. In Figure 10(d), we plot the distribution of counts over a 5 minute period for clients from each host. The distributions all fit a normal distribution and indicate that all of the clients are reading files uniformly at random. The distribution of reads from clients on A and D are skewed left, consistent with their overall lower read throughput.

Having confirmed the expected behavior of our stress test clients, we next checked to see whether the skewed datanode throughput was simply a result of skewed block placement across datanodes:

```
Q5: From getloc In NameNode.GetBlockLocations
    Join st In StressTestClient.DoNextOp On st -> getloc
    GroupBy st.host, getloc.replicas
    Select st.host, getloc.replicas, COUNT
```

This query measures the frequency that each DataNode is hosting a replica for files being read. Figure 10(e) shows that, for each client, replicas are near-uniformly distributed across DataNodes in the cluster. These results indicate that clients have an equal opportunity to read replicas from each DataNode, yet our measurements in Figure 10(c) clearly show that they do not. To gain more insight into this inconsistency, our next query relates the results from Figure 10(e) and (c):

```
Q6: From DNop In DataNode.DataTransferProtocol
    Join st In StressTestClient.DoNextOp On st -> DNop
    GroupBy st.host, DNop.host
    Select st.host, DNop.host, COUNT
```

This query measures the frequency that each client selects each DataNode for reading a replica. We plot the results in Figure 10(f) and see that the clients are clearly favoring particular DataNodes. The strong diagonal is consistent with HDFS client preference for locally-hosted replicas (39% of the time in this case). However, the expected behavior when there is not a local replica is to select a rack-local replica uniformly at random; clearly, these results suggest that this was not happening.

Our final diagnosis steps were as follows. First, we checked to see *which* replica was selected by HDFS clients from the locations returned by the NameNode. We found that clients always selected the first location returned by the NameNode. Second, we measured the conditional probabilities that DataNodes precede each other in the locations returned by the NameNode. We issued the following query for the latter:

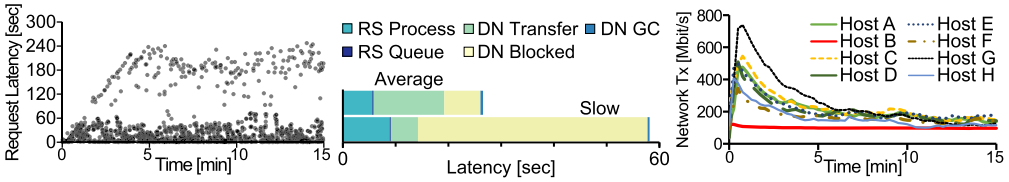
```
Q7: From DNop In DataNode.DataTransferProtocol
    Join getloc In NameNode.GetBlockLocations On getloc -> DNop
    Join st In StressTestClient.DoNextOp On st -> getloc
    Where st.host != DNop.host
    GroupBy DNop.host, getloc.replicas
    Select DNop.host, getloc.replicas, COUNT
```

This query correlates the DataNode that is selected with the other DataNodes also hosting a replica. We remove the interference from locally-hosted replicas by *filtering* only the requests that do a non-local read. Figure 10(g) shows that host A was *always* selected when it hosted a replica; host D was always selected except if host A was also a replica, and so on.

At this point in our analysis, we concluded that this behavior was quite likely to be a bug in HDFS. HDFS clients did not randomly select between replicas, and the HDFS NameNode did not randomize the rack-local replicas. We checked Apache's issue tracker and found that the bug had been recently reported and fixed in an upcoming version of HDFS [26].

### 6.2 Diagnosing End-to-End Latency

Pivot Tracing can express queries about the time spent by a request across the components it traverses using the built-in `time` variable exported by each tracepoint. Advice can pack the timestamp of any event then unpack it at a subsequent event, enabling comparison of timestamps between events. The following example query measures the latency between receiving a request and sending a response:




(a) HBase Request Latencies (b) Request latency decomposition. (c) Per-Machine Network Throughput

Fig. 11. (a) Observed request latencies for a closed-loop HBase workload experiencing occasional end-to-end latency spikes; (b) Average time in each component on average (top), and for slow requests (bottom, end-to-end latency > 30s); (c) Per-machine network throughput—a faulty network cable has downgraded Host B's link speed to 100Mbit, affecting entire cluster throughput.

```
Q8: From response In SendResponse
    Join request In MostRecent(ReceiveRequest) On request -> response
    Select response.time - request.time
```

When evaluating this query, **MostRecent** ensures we select only the most recent preceding ReceiveRequest event whenever SendResponse occurs. We can use latency measurement in more complicated queries. The following example query measures the average request latency experienced by Hadoop jobs:

```
Q9: From job In JobCompletionEvents
    Join latencyMeasurement In Q8 On latencyMeasurement -> end
    Select job.id, AVERAGE(latencyMeasurement)
```

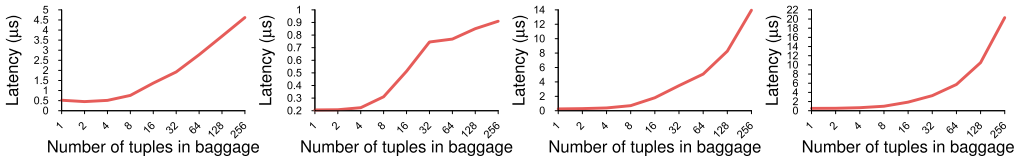
A query can measure latency in several components and propagate measurements in the baggage, reminiscent of transaction tracking in Timecard [86] and transactional profiling in Whodunit [39]. For example, during the development of Pivot Tracing, we encountered an instance of network limplock [51, 66], whereby a faulty network cable caused a network link downgrade from 1Gbit to 100Mbit. One HBase workload, in particular, would experience latency spikes in the requests hitting this bottleneck link (Figure 11(a)). To diagnose the issue, we decomposed requests into their per-component latency and compared anomalous requests (>30s end-to-end latency) to the average case (Figure 11(b)). This enabled us to identify the bottleneck source as time spent blocked on the network in the HDFS DataNode on Host B. We measured the latency and throughput experienced by all workloads at this component and were able to identify the uncharacteristically low throughput of Host B's network link (Figure 11(c)).

We have also replicated results in end-to-end latency diagnosis in the following other cases: to diagnose rogue garbage collection in HBase RegionServers as described in Ref. [101], and to diagnose an overloaded HDFS NameNode due to exclusive write locking as described in Ref. [67].

### 6.3 Overheads of Pivot Tracing

***Baggage.*** By default, Pivot Tracing propagates an empty baggage with a serialized size of 0 bytes. In the worst case, Pivot Tracing may need to pack an unbounded number of tuples in the baggage, one for each tracepoint invoked. However, the optimizations in Section 4 reduce the number of propagated tuples to one for `Aggregate`, one for `Recent`, $n$ for `GroupBy` with $n$ groups, and `N` for `RecentN`. Of the queries in this article, `Q7` propagates the largest baggage containing the stress test hostname and the location of all three file replicas (4 tuples, ≈137 bytes per request).

The size of serialized baggage is approximately linear in the number of packed tuples. The overhead to pack and unpack tuples from the baggage varies with the size of the baggage—Figure 12 gives micro-benchmark results measuring the overhead of baggage API calls.




(a) Pack 1 tuple (b) Unpack all tuples (c) Serialize baggage (d) Deserialize baggage

Fig. 12. Latency micro-benchmark results for packing, unpacking, serializing, and deserializing randomly-generated 8-byte tuples.

Table 5. Latency Overheads for HDFS Stress Test with Pivot Tracing Enabled, Baggage Propagation Enabled, and Full Queries Enabled, as Described in Section 6.3

| | *Benchmark* | | | |
|---|---|---|---|---|
| | Read8k | Open | Create | Rename |
| Unmodified System | 0% | 0% | 0% | 0% |
| Pivot Tracing Enabled | 0.3% | 0.3% | <0.1% | 0.2% |
| Baggage—1 Tuple | 0.8% | 0.4% | 0.6% | 0.8% |
| Baggage—60 Tuples | 0.82% | 15.9% | 8.6% | 4.1% |
| Queries—Section 6.1 | 1.5% | 4.0% | 6.0% | 0.3% |
| Queries—Section 6.2 | 1.9% | 14.3% | 8.2% | 5.5% |

***Application-level Overhead.*** To estimate the impact of Pivot Tracing on application-level throughput and latency, we ran benchmarks from HiBench [59], YCSB [47], and HDFS DFSIO and NNBench benchmarks. Many of these benchmarks bottleneck on network or disk, and we noticed no significant performance change with Pivot Tracing enabled.

To measure the effect of Pivot Tracing on CPU bound requests, we stress tested HDFS using requests derived from the HDFS NNBench benchmark: Read8k reads 8kB from a file; Open opens a file for reading; Create creates a file for writing; Rename renames an existing file. Read8kB is a DataNode operation and the others are NameNode operations. We compared the end-to-end latency of requests in unmodified HDFS to HDFS modified in the following ways: (1) with Pivot Tracing enabled; (2) propagating baggage containing one tuple but no advice installed; (3) propagating baggage containing 60 tuples (≈1kB) but no advice installed; (4) with the advice from queries in Section 6.1 installed; (5) with the advice from queries in Section 6.2 installed.

Table 5 shows that the application-level overhead with Pivot Tracing enabled is at most 0.3%. This overhead includes the costs of baggage propagation within HDFS, baggage serialization in RPC calls, and to run Java in debugging mode. The most noticeable overheads are incurred when propagating 60 tuples in the baggage, incurring 15.9% overhead for Open. Since this is a short CPU-bound request (involving a single read-only lookup), 16% is within reasonable expectations. Rename does not trigger any advice for the queries from Section 6.1, but does trigger advice for the queries from Section 6.2. Overheads of 0.3% and 5.5% respectively reflect this difference.

***Dynamic Instrumentation.*** Some Java Virtual Machines (JVMs) only support the HotSwap feature when debugging mode is enabled, which in turn disables some compiler optimizations. For practical purposes, however, HotSpot JVM's full-speed debugging is sufficiently optimized so that it is possible to run with debugging support always enabled [79]. Our HDFS throughput experiments above measured only a small overhead between debugging enabled and disabled. Reloading a class with woven advice has a one-time cost of approximately 100ms, depending on the size of the class being reloaded.

## 7 RELATED WORK

In Section 2, we described the challenges with troubleshooting tools that Pivot Tracing addresses, and complement the discussion on related work here.

Pivot Tracing's dynamic instrumentation is modeled after aspect-oriented programming [62] and extends prior dynamic instrumentation systems [38, 54, 83] with causal information that crosses process and system boundaries.

***Temporal Joins.*** Like Fay [54], Pivot Tracing models the events of a distributed system as a stream of dynamically generated tuples belonging to a distributed database. Pivot Tracing's happened-before join is an example of a $\theta$-join [85] where the condition is happened-before. Pivot Tracing's happened-before join is also an example of a special case of path queries in graph databases [102]. Differently from offline queries in a pre-stored graph, Pivot Tracing efficiently evaluates $\bowtie$ at runtime.

Pivot Tracing captures causality between events by generalizing metadata propagation techniques proposed in prior work such as X-Trace [55]. While Baggage enables Pivot Tracing to efficiently evaluate happened-before join, it is not necessary; Magpie [34] demonstrated that under certain circumstances, causality between events can be inferred after the fact. Specifically, if "start" and "end" events exist to demarcate a request's execution on a thread, then we can infer causality between the intermediary events. Similarly, we can also infer causality across boundaries, provided we can correlate "send" and "receive" events on both sides of the boundary (e.g., with a unique identifier present in both events). Under these circumstances, Magpie evaluates queries that explicitly encode all causal boundaries and use temporal joins to extract the intermediary events.

By extension, for any Pivot Tracing query with happened-before join, there is an equivalent query that explicitly encodes causal boundaries and uses only temporal joins. However, such a query could not take advantage of the optimizations outlined in this article and necessitates global evaluation.

***Beyond Metrics and Logs.*** A variety of tools have been proposed in the research literature to complement or extend application logs and performance counters. These include the use of machine learning [60, 76, 78, 103] and static analysis [107] to extract better information from logs; automatic enrichment of existing log statements to ease troubleshooting [106]; end-to-end tracing systems to capture the happened-before relationship between events [55, 93]; state-monitoring systems to track system-level resources and indicate the health of a cluster [69]; and aggregation systems to collect and summarize application-level monitoring data [64, 97]. Wang et al. provide a comprehensive overview of datacenter troubleshooting tools in Ref. [100]. These tools suffer from the challenges of pre-defined information outlined in Section 2.

***Troubleshooting and Root-Cause Diagnosis.*** Several offline techniques have been proposed to infer execution models from logs [35, 45, 68, 107] and diagnose performance problems [32, 63, 76, 90]. End-to-end tracing frameworks exist both in academia [34, 39, 55, 88, 95] and in industry [29, 52, 93, 94, 96] and have been used for a variety of purposes, including diagnosing anomalous requests whose structure or timing deviate from the norm [1, 34, 42, 43, 80, 90]; diagnosing steady-state problems that manifest across many requests [55, 88, 90, 93, 95]; identifying slow components and functions [39, 68, 93]; and modeling workloads and resource usage [33, 34, 68, 95]. Recent work has extended these techniques to online profiling and analysis [57, 71–73, 108].

VScope [101] introduces a novel mechanism for honing in on root causes on a running system, but, at the last hop, defers to offline user analysis of debug-level logs, requiring the user to trawl through 500MB of logs, which incur a 99.1% performance overhead to generate. While causal

tracing enables coherent sampling [89, 93], which controls overheads, sampling risks missing important information about rare but interesting events.

***Context Propagation.*** Several use cases have been demonstrated in the research literature that require context propagation between system components. Many systems propagate activity and request IDs for use in debugging, profiling, and end-to-end tracing [55, 93]. Taint tracking and DIFC maintain and propagate security labels as the system executes, warning of or prohibiting policy violations [53, 75, 104]. Data provenance systems propagate information about the lineage of data as different components manipulate it [50, 74]. Tools for end-to-end latency, path profiling, and critical path analysis propagate partial latency measurements and information about execution paths and graphs, in order to adapt to bottlenecks or processing delays [39, 46, 86]. Tenant IDs enable coordinated scheduling decisions across components [67, 91]. Pivot Tracing's baggage abstraction generalizes the contexts used by systems such as these, by providing an opaque container for queries to dynamically propagate tuples along the execution path of requests.

***Online Analytics Processing.*** Pivot Tracing derives its name from pivot tables and pivot charts [48] from spreadsheet programs due to their ability to dynamically select values, functions, and grouping dimensions from an underlying dataset. Pivot tables are simple versions of data cubes [56] in the online analytics processing domain. Streaming data cubes [58] perform similar real-time aggregation and optimization on incoming data streams.

## 8 DISCUSSION

Despite the advantages over logs and metrics for troubleshooting (Section 2), Pivot Tracing is not meant to replace all functions of logs, such as security auditing, forensics, or debugging [77].

Pivot Tracing is designed to have similar per-query overheads to the metrics currently exposed by systems today. It is feasible for a system to have several Pivot Tracing queries on by default; these could be sensible defaults provided by developers, or custom queries installed by users to address their specific needs. We leave it to future work to explore the use of Pivot Tracing for automatic problem detection and exploration.

Dynamic instrumentation is not a requirement to utilize Pivot Tracing. By default, a system could hard-code a set of predefined tracepoints. Without dynamic instrumentation, the user is restricted to those tracepoints; adding new tracepoints remains tied to the development and build cycles of the system. Inactive tracepoints would also incur at least the cost of a conditional check, instead of our current zero cost.

A common criticism of systems that require causal propagation of metadata is the need to instrument the original systems [45]. We argue that the benefits outweigh the costs of doing so (Section 6), especially for new systems. We also propose that the baggage abstraction, or a generalization, could be shared with other applications like those described in Section 7; then, system instrumentation would be a one-time task, reusable, and independent of any tracing system or use case, and deploying new tracing systems would be possible without having to revisit the underlying context propagation mechanism. Nonetheless, a system that does not implement baggage can still utilize the other mechanisms of Pivot Tracing; in such a case, the system resembles DTrace [38] or Fay [54]. Alternatively, kernel-level execution context propagation [40, 81, 87] can provide language-independent access to baggage variables.

While users are restricted to advice comprised of Pivot Tracing primitives, Pivot Tracing does not guarantee that its queries will be side-effect free, due to the way exported variables from tracepoints are currently defined. We can enforce that only trusted administrators define tracepoints and require that advice be signed for installation, but a comprehensive security analysis, including complete sanitization of tracepoint code is beyond the scope of this article.

Even though we evaluated Pivot Tracing on an 8-node cluster in this article, initial runs of the instrumented systems on a 200-node cluster with constant-size baggage being propagated showed negligible performance impact. It is ongoing work to evaluate the scalability of Pivot Tracing to larger clusters and more complex queries. Sampling at the advice level is a further method of reducing overhead, which we plan to investigate.

We opted to implement Pivot Tracing in Java in order to easily instrument several popular open-source distributed systems written in this language. However, the components of Pivot Tracing generalize and are not restricted to Java—a query can span multiple systems written in different programming languages due to Pivot Tracing's platform-independent baggage format and restricted set of advice operations. In particular, it would be an interesting exercise to integrate the happened-before join with Fay or DTrace.

## 9 CONCLUSION

Pivot Tracing is the first monitoring system to combine dynamic instrumentation and causal tracing. Its novel happened-before join operator fundamentally increases the expressive power of dynamic instrumentation and the applicability of causal tracing. Pivot Tracing enables cross-tier analysis between any inter-operating applications, with low execution overhead. Ultimately, its power lies in the uniform and ubiquitous way in which it integrates monitoring of a heterogeneous distributed system.

## REFERENCES


[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*.

[2] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. 2008. Interval tree clocks: A logical clock for dynamic systems. In *12th International Conference on Principles of Distributed Systems (OPODIS'08)*.

[3] Apache. 2017. Accumulo. Retrieved January 2017 from https://accumulo.apache.org/.

[4] Apache. 2017. HADOOP-6599 Split RPC metrics into summary and detailed metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HADOOP-6599.

[5] Apache. 2017. HADOOP-6859 Introduce additional statistics to FileSystem. Retrieved July 2017 from https://issues.apache.org/jira/browse/HADOOP-6859.

[6] Apache. 2016. HBase. Retrieved June 2016 from https://hbase.apache.org.

[7] Apache. 2017. HBASE-11559 Add dumping of DATA block usage to the BlockCache JSON report. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-11559.

[8] Apache. 2017. HBASE-12364 API for query metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12364.

[9] Apache. 2017. HBASE-12424 Finer grained logging and metrics for split transaction. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12424.

[10] Apache. 2017. HBASE-12477 Add a flush failed metric. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12477.

[11] Apache. 2017. HBASE-12494 Add metrics for blocked updates and delayed flushes. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12494.

[12] Apache. 2017. HBASE-12496 A blockedRequestsCount metric. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12496.

[13] Apache. 2017. HBASE-12574 Update replication metrics to not do so many map look ups. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12574.

[14] Apache. 2017. HBASE-2257 [stargate] multiuser mode. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-2257.

[15] Apache. 2017. HBASE-4038 Hot Region : Write Diagnosis. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4038.

[16] Apache. 2017. HBASE-4145 Provide metrics for hbase client. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4145.

[17] Apache. 2017. HBASE-4219 Add Per-Column Family Metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4219.
[18] Apache. 2017. HBASE-7958 Statistics per-column family per-region. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-7958.
[19] Apache. 2017. HBASE-8370 Report data block cache hit rates apart from aggregate cache hit rates. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-8370.
[20] Apache. 2017. HBASE-8868 add metric to report client shortcircuit reads. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-8868.
[21] Apache. 2017. HBASE-9722 need documentation to configure HBase to reduce metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-9722.
[22] Apache. 2017. HBase Reference Guide. Retrieved July 2017 from https://hbase.apache.org/book.html.
[23] Apache. 2017. HDFS-4169 Add per-disk latency metrics to DataNode. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-4169.
[24] Apache. 2017. HDFS-5253 Add requesting user's name to PathBasedCacheEntry. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-5253.
[25] Apache. 2017. HDFS-6093 Expose more caching information for debugging by users. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6093.
[26] Apache. 2017. HDFS-6268 Better sorting in NetworkTopology.pseudoSortByDistance when no local node is found. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6268.
[27] Apache. 2017. HDFS-6292 Display HDFS per user and per group usage on webUI. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6292.
[28] Apache. 2017. HDFS-7390 Provide JMX metrics per storage type. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-7390.
[29] Apache. 2017. HTrace. Retrieved January 2017 from http://htrace.incubator.apache.org/.
[30] Apache. 2017. MESOS-1949 All log messages from master, slave, executor, etc. should be collected on a per-task basis. Retrieved July 2017 from https://issues.apache.org/jira/browse/MESOS-1949.
[31] Apache. 2017. MESOS-2157 Add /master/slaves and /master/frameworks/{framework}/tasks/{task} endpoints. Retrieved July 2017 from https://issues.apache.org/jira/browse/MESOS-2157.
[32] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*.
[33] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using magpie for request extraction and workload modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*.
[34] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online modelling and performance-aware systems. In *9th USENIX Workshop on Hot Topics in Operating Systems (HotOS'03)*.
[35] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *36th ACM International Conference on Software Engineering (ICSE'14)*.
[36] Peter Bodik. 2011. Overview of the workshop of managing large-scale systems via the analysis of system logs and the application of machine learning techniques (SLAML'11). *SIGOPS Operating Systems Review* 45, 3 (2011), 20–22.
[37] Bryan Cantrill. 2006. Hidden in plain sight. *ACM Queue* 4, 1 (2006), 26–36.
[38] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic instrumentation of production systems. In *2004 USENIX Annual Technical Conference (ATC)*.
[39] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. 2007. Whodunit: Transactional profiling for multi-tier applications. In *2nd ACM European Conference on Computer Systems (EuroSys'07)*.
[40] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. 2005. Causeway: Support for controlling and analyzing the execution of multi-tier applications. In *6th ACM/IFIP/USENIX International Middleware Conference (Middleware'05)*.
[41] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*.
[42] Mike Y. Chen, Anthony Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. 2004. Path-based failure and evolution management. In *1st USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*.
[43] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'02)*.

[44] Shigeru Chiba. 2004. Javassist: Java bytecode engineering made simple. *Java Developer's Journal* 9, 1 (2004).
[45] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.
[46] Michael Chow, Kaushik Veeraraghavan, Michael Cafarella, and Jason Flinn. 2016. DQBarge: Improving data-quality tradeoffs in large-scale internet services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.
[47] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC'10)*.
[48] J. Couckuyt, P. Davies, and J. M. Cahill. 2005. Multiple chart user interface. US Patent US6906717 B2.
[49] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*.
[50] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. 2011. QUIRE: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium (Security'11)*.
[51] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. 2013. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *4th ACM Symposium on Cloud Computing (SoCC'13)*.
[52] Dynatrace. 2017. Dynatrace Application Monitoring. Retrieved July 2017 from http://www.dynatrace.com.
[53] William Enck, Peter Gilbert, Byung-Gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.
[54] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. 2011. Fay: Extensible distributed tracing from kernels to clusters. In *23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.
[55] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*.
[56] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery* 1, 1 (1997), 29–53.
[57] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. 2011. $G^2$: A graph processing system for diagnosing distributed systems. In *2011 USENIX Annual Technical Conference (ATC)*.
[58] Jiawei Han, Yixin Chen, Guozhu Dong, Jian Pei, Benjamin W. Wah, Jianyong Wang, and Y. Dora Cai. 2005. Stream cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases* 18, 2 (2005), 173–197.
[59] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *26th IEEE International Conference on Data Engineering Workshops (ICDEW'10)*.
[60] Soila P. Kavulya, Scott Daniels, Kaustubh Joshi, Matti Hiltunen, Rajeev Gandhi, and Priya Narasimhan. 2012. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *42nd IEEE/IFIP Conference on Dependable Systems and Networks (DSN'12)*.
[61] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP'01)*.
[62] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*.
[63] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. 2013. Root cause detection in a service-oriented architecture. In *2013 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.
[64] Steven Y. Ko, Praveen Yalagandula, Indranil Gupta, Vanish Talwar, Dejan Milojicic, and Subu Iyer. 2008. Moara: Flexible and scalable group-based querying system. In *9th ACM/IFIP/USENIX International Conference on Middleware (Middleware'08)*.
[65] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
[66] Brian Laub, Chengwei Wang, Karsten Schwan, and Chad Huneycutt. 2014. Towards combining online & offline management for big data applications. In *11th USENIX International Conference on Autonomic Computing (ICAC'14)*.
[67] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*.

[68] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. 2011. Modeling the parallel execution of black-box services. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'11)*.

[69] Matthew L. Massie, Brent N. Chun, and David E. Culler. 2004. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840.

[70] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling object, relations and XML in the .NET framework. In *2006 ACM SIGMOD International Conference on Management of Data*.

[71] Haibo Mi, Huaimin Wang, Zhenbang Chen, and Yangfan Zhou. 2014. Automatic detecting performance bugs in cloud computing systems via learning latency specification model. In *8th IEEE International Symposium on Service Oriented System Engineering (SOSE'14)*. IEEE, 302–307.

[72] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael R. Lyu, and Hua Cai. 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255.

[73] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, Hua Cai, and Gang Yin. 2013. An online service-oriented performance profiling tool for cloud computing systems. *Frontiers of Computer Science* 7, 3 (2013), 431–445.

[74] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo I. Seltzer. 2010. Provenance for the cloud. In *8th USENIX Conference on File and Storage Technologies (FAST'10)*.

[75] Andrew C. Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *16th ACM Symposium on Operating Systems Principles (SOSP'97)*.

[76] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*.

[77] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *Commun. ACM* 55, 2 (2012), 55–61.

[78] Adam J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken. 2010. Using correlated surprise to infer shared influence. In *40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*.

[79] Oracle. 2015. The Java HotSpot Performance Engine Architecture. Retrieved March 2015 from http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[80] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. 2011. Diagnosing latency in multi-tier black-box services. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS'11)*.

[81] Insung Park and Ricky Buch. 2007. Event Tracing: Improve Debugging and Performance Tuning with ETW. Retrieved July 2017 from http://download.microsoft.com/download/3/A/7/3A7FA450-1F33-41F7-9E6D-3AA95B5A6AEA/MSDNMagazineApril2007en-us.chm.

[82] D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering* 3 (1983), 240–247.

[83] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. 2005. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*.

[84] Ariel Rabkin and Randy Howard Katz. 2013. How Hadoop clusters break. *IEEE Software* 30, 4 (2013), 88–94.

[85] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems* (2nd ed.). Osborne/McGraw-Hill, Berkeley, CA.

[86] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. 2013. Timecard: Controlling user-perceived delays in server-based mobile applications. In *24th ACM Symposium on Operating Systems Principles (SOSP'13)*.

[87] John Reumann and Kang G. Shin. 2004. Stateful distributed interposition. *ACM Transactions on Computer Systems* 22, 1 (2004), 1–48.

[88] Patrick Reynolds, Charles Edwin Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. 2006. Pip: Detecting the unexpected in distributed systems. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*.

[89] Raja R. Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R. Ganger. 2014. *So, You Want to Trace Your Distributed System? Key Design Insights from Years of Practical Experience*. Technical Report CMU-PDL-14-102. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

[90] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*.

[91] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance isolation and fairness for multi-tenant cloud storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*.

[92] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop distributed file system. *26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*.
[93] 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. Google.
[94] SolarWinds. 2017. Traceview. Retrieved July 2017 from https://traceview.solarwinds.com/.
[95] 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 *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.
[96] Twitter. 2017. Zipkin. Retrieved July 2017 from http://zipkin.io/.
[97] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems* 21, 2 (2003), 164–206.
[98] Kenton Varda. 2008. Protocol Buffers: Google's Data Interchange Format. Retrieved January 2017 from https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html.
[99] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet another resource negotiator. In *4th ACM Symposium on Cloud Computing (SoCC'13)*.
[100] Chengwel Wang, Soila P. Kavulya, Jiaqi Tan, Liting Hu, Mahendra Kutare, Mike Kasick, Karsten Schwan, Priya Narasimhan, and Rajeev Gandhi. 2013. Performance troubleshooting in data centers: An annotated bibliography. *ACM SIGOPS Operating Systems Review* 47, 3 (2013), 50–62.
[101] Chengwei Wang, Infantdani Abel Rayan, Greg Eisenhauer, Karsten Schwan, Vanish Talwar, Matthew Wolf, and Chad Huneycutt. 2012. VScope: Middleware for troubleshooting time-sensitive data center applications. In *13th ACM/IFIP/USENIX International Middleware Conference (Middleware'12)*.
[102] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.
[103] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP'09)*.
[104] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium (Security'12)*.
[105] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, 159–172.
[106] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving software diagnosability via log enhancement. In *16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.
[107] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.
[108] Jingwen Zhou, Zhenbang Chen, Haibo Mi, and Ji Wang. 2014. MTracer: A trace-oriented monitoring framework for medium-scale distributed systems. In *8th IEEE International Symposium on Service Oriented System Engineering (SOSE'14)*.

Received August 2017; revised January 2018; accepted April 2018