Dynamic Causal Monitoring for Distributed Systems

Ryan Roelke
Brown University

Abstract

Monitoring and troubleshooting distributed systems is notoriously difficult; potential problems are complex, varied, and unpredictable. The *de-facto* monitoring and diagnosis tools at our disposal 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 report is an extended version of our full Pivot Tracing paper [68]. Pivot Tracing is 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 any inter-operating applications, with low execution overhead.

This report extends the original paper's discussion [68] of Pivot Tracing's implementation and provides further details on instrumenting and operating Pivot Tracing within distributed systems.

1 Introduction

Monitoring and troubleshooting distributed systems is notoriously difficult. 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, the *de-facto* monitoring and diagnosis tools at our disposal today – logs, counters, and metrics – have at least two fundamental limitations: what gets recorded is defined a priori, at development or deployment time, and the information is recorded in a component- or machine-centric way, making it extremely hard to correlate events that cross these boundaries.

While there has been great progress in using machine learning techniques [75, 97, 78, 60] and static analysis [100, 99] to improve the quality of logs and their use in troubleshooting, they are still limited to a one-size-fits all solution with an inherent tradeoff between precision and overhead. Likewise, with monitoring, performance counters may be too coarse-grained [74]; and if a user requests additional metrics, a cost-benefit tug-of-war with the developers can ensue [21].

Dynamic instrumentation systems such as Fay [51] 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.

In this paper we combine dynamic instrumentation with causal tracing techniques [52, 89, 39] 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 being able to select, filter, and group 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 a novel operator, the "happened-before join", $\overrightarrow{\bowtie}$, 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 [54], and 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 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. Pivot Tracing 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 any inter-operating applications with low execution overhead.

In summary, this paper 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.

This report extends the original paper's discussion [68] of Pivot Tracing's implementation details and requirements for deploying Pivot Tracing to a system.

2 Motivation

2.1 Pivot Tracing in Action: Preview

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 later sections.

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 5 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.

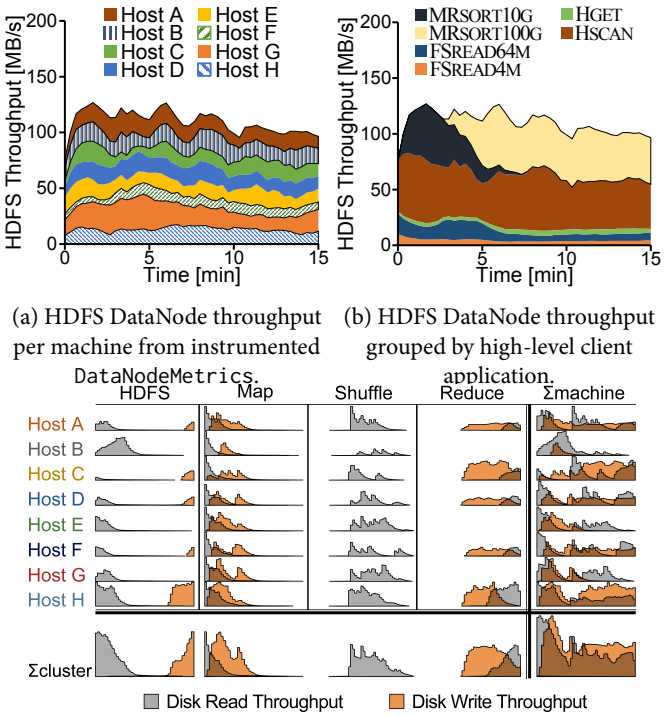
We run the following client applications:

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[1] for the `DataNodeMetrics` class to intercept the `incrBytesRead(int delta)` method, and we run the following query, in Pivot Tracing's LINQ-like query language [71]:

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

[1] A tracepoint is a location in the application source code where instrumentation can run, cf. §3.



(a) HDFS DataNode throughput per machine from instrumented

(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.

Figure 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.

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 1a.

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 1b 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 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 1c 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 Monitoring and Troubleshooting Challenges

Before we describe the design of Pivot Tracing, it is worth examining the challenges faced by traditional monitoring and troubleshooting techniques. Essentially, we group these challenges along two dimensions: first, when the choice of what to record about an execution is made *a priori*, there is an inherent tradeoff between precision and overhead. Second, solutions are tied to each individual component, which makes it hard to correlate and integrate cross-component data.

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 in [35] discuss the important need to "close the loop of operations back to developers". According to Yuan *et al.* [99], 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 [3, 4, 7, 8, 9, 17, 22]; to request changes to aggregation methods [10, 21, 23]; and to request new breakdowns of existing metrics [2, 5, 6, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 25]. Many issues remain unresolved due to developer pushback [20, 19, 17, 16, 12] or inertia [25, 23, 22, 18, 14, 8, 7, 5]. Even simple cases of misconfiguration are frequently unreported by error logs [98].

Eventually, applications are 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 [21]. The HBase user guide [1] carries the following warning for users users wishing to integrate with Ganglia [70]: *"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."*

In [81], the authors highlight the "needle-in-a-haystack" nature of diagnosis using logs; while a log may contain information relevant to a problem, extracting this information from a log requires system familiarity developed over a long period of time. VScope [95] 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. In [26], users complain that the entire state of the cluster is exposed via a single JSON endpoint and can become massive, even if a client only wants information for a subset of the state.

Dynamic instrumentation frameworks such as Fay [51], DTrace [38], and SystemTap [80] address these limitations, by allowing almost arbitrary instrumentation to be installed dynamically at runtime. These tools define many probes in the system, both at the kernel and user level, where they can insert instrumentation code, with near zero cost for inactive probes. This code can read, but not write, any state of the running system. They have proven extremely useful in the diagnosis of complex and subtle system problems [37]. They are limited, though, 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 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, [13] 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. [25] 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."*

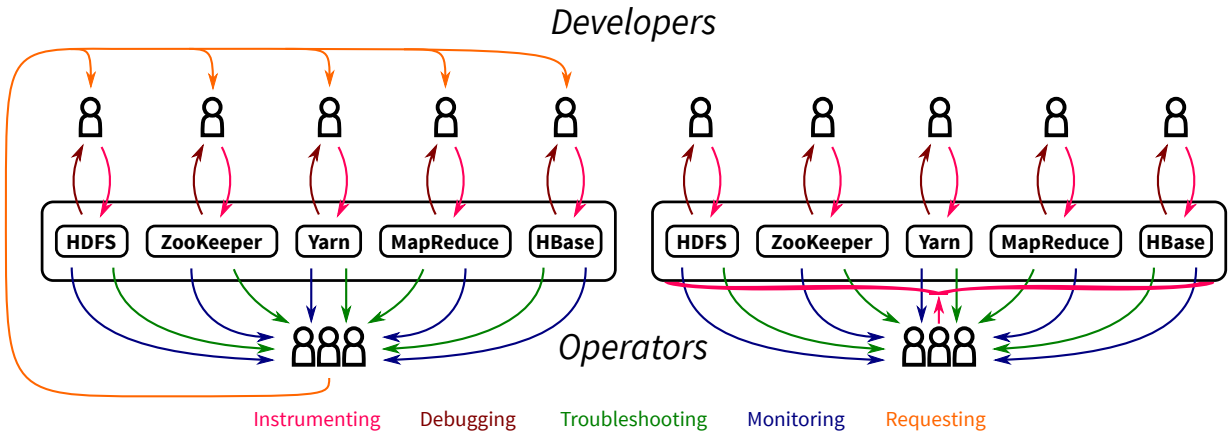


Figure 2: Developers instrument and debug single components of complex systems, while operators troubleshoot and monitor the entire system. On the left is the current state: operators must request additional instrumentation from developers. On the right is the ideal, where operators can instrument entire multi-component systems directly, without depending on developers.

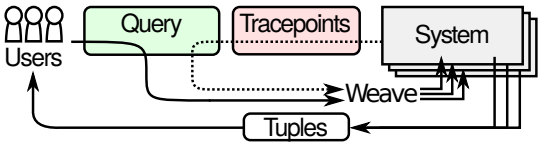


Figure 3: Interactions between Pivot Tracing components (§3).

Prior research has presented mechanisms to observe or infer the relationship between events [34, 45, 69, 87, 100, 52, 89, 31, 86, 32, 43, 85, 28]. Studies of logging practices conclude that end-to-end tracing would be helpful in navigating the logging issues they outline [81, 77]. A variety of these mechanisms have also materialized in production systems: for example, Google's Dapper [89], HBase's HTrace [58], Accumulo's Cloudtrace [27], and Twitter's Zipkin [91]. Overall, these approaches can obtain much 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 some of the problems above concerning what to record. Causal tracing enables coherent sampling [86, 89], which controls the overhead, but risks missing important information about rare but interesting events.

Figure 2 summarizes the challenges outlined in this section. Currently, developers control what gets instrumented in a system without an integrated view of all components. In the ideal case, operators should have tools that overcome the insufficient and isolated logging and monitoring mechanisms provided by components of a system. In the coming sections we discuss Pivot Tracing, which presents a leap forward to resolving these issues.

3 Design

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 2 motivate the following high-level design goals:

- Configure, implement, and install monitoring at runtime
- Low system overhead to enable "always on" monitoring
- Capture causality between events from multiple processes and applications

In this section we outline the fundamental concepts and mechanisms behind Pivot Tracing. We first give an overview of Pivot Tracing components, then go on to describe each component in more detail.

Pivot Tracing Overview

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 (σ)	Filter only tuples that match a predicate	`Where e.Size < 10`
Projection (Π)	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`

Table 1: Operations supported by the Pivot Tracing query language

Figure 3 shows the high-level interactions between Pivot Tracing components. Pivot Tracing models events as the tuples of a streaming, distributed dataset. Users submit relational queries over this dataset, which get compiled and installed in the system; query results are streamed back to the user. A query refers to variables exposed by one or more *tracepoints* — places in the system where Pivot Tracing can insert instrumentation. Each invocation of a tracepoint generates a tuple. Queries can select, filter, aggregate, and group tuples. We distinguish Pivot Tracing from prior work by supporting *joins* between events that occur within and across process, machine, and application boundaries.

To install a query, Pivot Tracing generates code implementing the query and *weaves* that code into the queried tracepoints. We discuss the generated code in §4.2 and the process of weaving in §4.3. After code is weaved, subsequent requests executing in the system invoke that code each time their execution reaches the tracepoint.

Tracepoints Tracepoints provide the system-level entry point for Pivot Tracing queries. A tracepoint typically corresponds to some event: a user submits a request; a low-level IO operation completes; an external RPC is invoked, etc. More specifically, a tracepoint identifies one or more locations in the system code where Pivot Tracing can install and run instrumentation, which is specified at runtime by operator queries and implemented in a restricted language which we describe in §4.2. Tracepoints export named variables that can be accessed by instrumentation. Figure 10 shows the specification of one of the tracepoints in `Q2` from §2. Besides declared exports, all tracepoints export a few variables by default: host, timestamp, process id, process name, and the tracepoint definition.

Whenever execution of the system reaches 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.

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 §2. Table 1 outlines the query operations supported by Pivot Tracing. Pivot Tracing supports several typical operations including projection (Π), selection (σ), 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 novel *happened-before join* query operator ($\overrightarrow{\bowtie}$).

Happened-before Joins A key contribution of Pivot Tracing is the novel 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.[2] If a and b are not part of 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 $b \not\rightarrow a$.

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 4 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.

Query `Q2` in §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.

[2] 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.

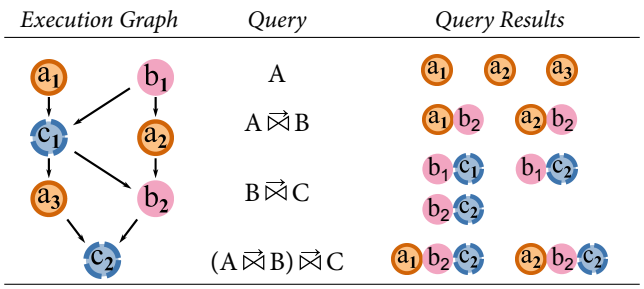


Figure 4: An example execution that triggers tracepoints A, B and C several times. We show several Pivot Tracing queries and the tuples that would result for each.

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 (§6). Pivot Tracing is designed to efficiently support happened-before joins, but does not optimize more general joins such as equijoins ($\bowtie$).

4 Pivot Tracing Fundamentals

We have implemented a prototype of Pivot Tracing for Java-based system. We categorize our prototype implementation into three pieces. To cross system components, we implement a generalization of end-to-end metadata propagation called *baggage* that allows different components of the system to interact with one another. To bridge high-level queries with low-level implementation, we introduce an intermediary representation for queries called *advice*. Finally, to address the problem of "one size fits all", we implement dynamic instrumentation for Java that allows us to insert code at runtime.

4.1 Baggage

A naïve implementation of happened-before join is expensive, as tuples must be aggregated across the cluster prior to performing the join. For example, temporal joins implemented by Magpie [33] are expensive for this reason.

In order to support Pivot Tracing, we need an efficient implementation of the happened-before join. To achieve an efficient implementation, Pivot Tracing borrows from techniques in end-to-end tracing, by generalizing causal metadata propagating found in systems such as X-Trace[52] and Dapper[89].

Pivot Tracing introduces the *baggage* abstraction to efficiently implement happened-before joins. Baggage is a per-request container for arbitrary key-value pairs that is propagated alongside a request as it traverses thread, application and machine boundaries. The execution of a request starts in a single thread, but can split into new threads, traverse process boundaries via the network, batch requests with other threads for a shared execution, or defer a subroutine to be executed by a different thread pulling it off a queue. In each of these cases, we need to maintain identifiers that identify an execution and its key-value container as it branches off from its original thread.

4.1.1 Implementation

Baggage is principally a container of key-value pairs; an application can store a key-value pair in the current execution's baggage, or query the baggage to retrieve all values previously stored under a specific key. Baggage maintains the invariant that values stored at one point in a request's execution are guaranteed to be available later on in the same execution unless they are specifically removed. Pivot Tracing provides a client library implementation of baggage. Table 2 outlines the API of this library, including functions to get and set key-value pairs, and to retrieve baggage for passing across execution boundaries.

Pivot Tracing uses a thread-local variable to store baggage instances. 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. At any point during the execution, we can query for the current value of the thread-local variable to receive the current baggage. Internally, baggage is a multimap pairing keys with lists of `ByteStrings`. Our baggage API, outlined in Table 2, restricts the legal operations processes can perform on the values contained in this multimap. **pack** stores an arbitrary value for propagation with this execution as it traverses thread or process boundaries; **unpack** retrieves values **pack**ed prior in the execution.

Pivot Tracing provides a serialized representation for baggage. The baggage API has methods for serializing and deserializing the current baggage or setting the current baggage from a serialized representation. Serialization supports

Method	*Description*
`pack(k, t...)`	Add a value to the multimap
`unpack(k)`	Retrieve all values from the multimap
`repack(k, t...)`	Set the values in the multimap
`serialize()`	Serialize the multimap to bytes
`deserialize(b)`	Set the multimap by deserializing from bytes
`split()`	Split the baggage for a branching execution
`join(b1, b2)`	Merge baggage from two joining executions

Table 2: Baggage API for Pivot Tracing Java implementation. API methods are static and only allow interaction with the current execution's baggage.

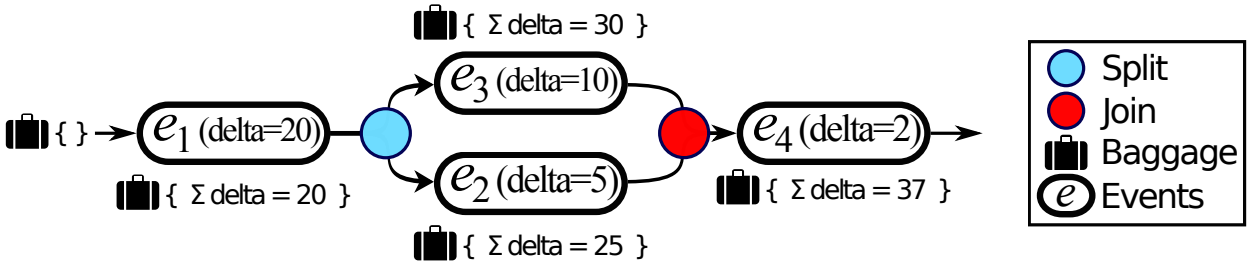


Figure 5: A sample execution aggregates a counter in the baggage, correctly computing the total even after events on both sides of the split execution.

propagating baggage alongside executions that traverse multiple processes or applications. When one application sends a message to a different process or application, it sends a serialized representation of its baggage together with that message. Our prototype specifies its serialization format using Protocol Buffers [53]. Baggage is lazily serialized and deserialized to minimize the overhead of propagating baggage through applications that do not actively participate in a query; baggage is deserialized only when an application attempts to **pack** or **unpack** tuples. Serialization costs are only incurred for modified baggage at network or application boundaries.

Splits, Joins, and Versioning In order to preserve the happened-before relation, Pivot Tracing must correctly handle executions that split and rejoin. When an execution branches, each branch must receive a copy of the execution's current baggage. Each branch now independently interacts with its own baggage. When several branches of an execution rejoin, the baggage from each joining branch must be merged. Figure 5 illustrates an example of this. Values accumulated in the baggage prior to the split must only be counted once during aggregation and upon merging.

To handle splits and joins, we implement a versioning scheme using *interval tree clocks* [29]. Each baggage instance now also points to a "parent" baggage from which it split off, and the thread-local variable references the bottom-most (youngest) baggage instance. **pack** only puts values into the youngest baggage, while **unpack** iterates through the entire chain, retrieving values stored in all versions. As a result, executions cannot modify any baggage inherited from a parent thread.

We split baggage by dividing the interval tree ID of the youngest baggage into two new globally unique, non-overlapping interval tree IDs [29]. We instantiate two new empty baggage instances, assign each one half of the divided ID, set the parent of the fresh baggage to be the current active baggage, and then update the thread-local reference to point to one of the new instances. **split** returns a reference to the other, so that manual instrumentation (described in §4.1.2) can set it to propagate through the other half of the split.

Our initial prototype assumes the common case where a forked execution joins only with the other end of the fork. With this assumption, we implement **join** by copying the contents of both youngest baggage instances into their shared parent, and set the thread-local reference to back to that parent. The interval tree IDs enforce this assumption. A future prototype will fully integrate interval tree clocks to remove this assumption.

4.1.2 Instrumentation

Pivot Tracing relies on developers to implement Baggage propagation when a request crosses thread, process, or asynchronous execution boundaries.

We instrumented a number of applications on the Hadoop stack including HDFS, MapReduce, Yarn. We discuss the

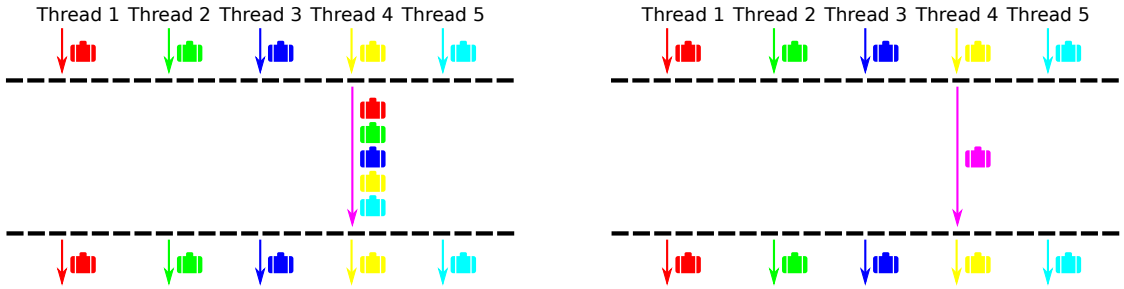
fruits of this instrumentation in Section §5, but here we describe the steps required to do so. Overall, instrumenting these applications to support Baggage required less than 200 lines of code per system.

Thread Boundaries Objects in Java can run in separate threads if they implement the `Runnable` interface. We used AspectJ [61], an aspect-oriented programming language extension of Java, to intercept `Runnable` instances to automatically inherit the **`split`** of the parent thread's baggage, and save a copy of its baggage to be joined with later on. Here we provide one of the aspects implementing this behavior:

```
1  public aspect InstrumentBaggage {
2      private interface PivotTracingRunnable {
3          public void setStartBaggage();
4          public Baggage getStartBaggage();
5          public void setEndBaggage();
6          public Baggage getEndBaggage();
7      }
8
9      private Baggage PivotTracingRunnable.startBaggage;
10     private Baggage PivotTracingRunnable.endBaggage;
11
12     public void PivotTracingRunnable.setStartBaggage() {
13         this.startBaggage = PivotTracing.getBaggage().split();
14     }
15     public Baggage PivotTracingRunnable.getStartBaggage() {
16         return this.startBaggage;
17     }
18     public void PivotTracingRunnable.setEndBaggage() {
19         this.endBaggage = PivotTracing.getBaggage();
20     }
21     public void PivotTracingRunnable.getEndBaggage() {
22         return this.endBaggage;
23     }
24
25     declare parents: Runnable+ implements PivotTracingRunnable;
26
27     before(PivotTracingRunnable r): this(r) && execution(void
           PivotTracingRunnable+.run(..)) {
28     }
29     after(PivotTracingRunnable r): this(r) && execution(void
           PivotTracingRunnable+.run(..)) {
30         r.setEndBaggage();
31     }
32
33     void around(PivotTracingRunnable r): this(r) && call(void
           PivotTracingRunnable+.run(..)) {
34         PivotTracing.setBaggage(r.getStartBaggage());
35         proceed(r);
36         r.setEndBaggage();
37     }
38
39     void around(Thread t): target(t) && call(void Thread+.join(..)) {
40         ThreadWrapper.join(t);
41         proceed(t);
42     }
43 }
```

In addition to these modifications, we also intercept `Thread` creation to call `setStartBaggage` on the target `Runnable`, and call `getEndBaggage` to extract the end baggage from a joined `Thread`, as in line 40 of the above aspect. The aspect implementing this behavior is similarly concise.

These aspects correctly instrument the simple but common fork-join pattern. However, there are other patterns of concurrent execution for which this instrumentation is either insufficient or incorrect. In such cases, manual instrumen-



(a) The shared execution merges and then distributes per-execution baggage upon completion.

(b) The shared execution starts its own baggage for the duration of its own execution, then reverts to pre-execution baggage upon completion.

Figure 6: Some distributed systems use shared execution patterns such as batching. In such cases, it is up to the system instrumenter to develop the appropriate semantics for baggage propagation.

tation is required to ensure that the current baggage is not split. This can be done by saving the current baggage, clearing the thread-local baggage, and then restoring the thread-local baggage from the saved copy after thread creation. Our implementation assumes that the created thread will not be joined with.

Background Tasks Background tasks are conceptually similar to user requests. Programs spawning background tasks should not split their baggage. Instead, when a background task begins, it should be initialized with fresh baggage since it is conceptually an independent execution.

Process Boundaries Instrumenting RPCs requires extending the target system's protocol definitions to include a field for storing serialized Baggage. Whenever an application invokes an RPC, it must also serialize the current baggage and set the protocol buffer's baggage field. In our instrumentation, this required only a few additional tweaks: increasing the maximum size of a protocol buffer, and adding code to copy the current baggage into the relevant field of the protocol buffer. Each of these modifications required a single line of code each.

Asynchronous Boundaries If an execution is deferred via a queue, we must modify the enqueued object that represents the deferred execution (*e.g.* an `RPCServer.Call` object in HDFS). We add a field to the object for storing the state of the baggage when the object is enqueued. Later, when some other thread dequeues the object to resume its deferred execution, that thread must immediately set its baggage to the baggage stored in the deferred execution context.

Shared Executions Some systems batch multiple concurrent requests to be handled by a single execution context. To properly propagate baggage in such cases, threads must save a copy of their baggage in their batched request to restore it upon returning from the batched request. However, the state of the baggage *during* the batched execution depends on the application and thus requires manual instrumentation. Figure 6 explains two possibilities.

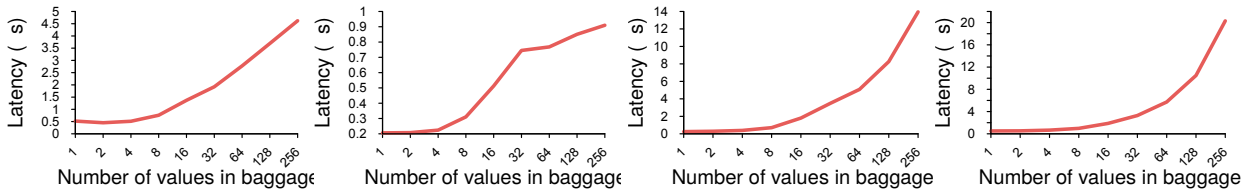
Callbacks To preserve baggage into a callback, we borrow techniques from continuation-local storage [76]. When an execution registers a callback, it also registers a copy of its baggage to be restored when the callback is invoked. Callback handlers save their baggage before invoking the callback, and restore it after returning from the callback.

4.1.3 Overheads

The size of serialized baggage containing k keys and at most v values per key is $O(kv)$, with an empty serialized size of 0 bytes. To estimate the latency of propagating baggage, we ran a series of both microbenchmarks and macrobenchmarks. Figure 7 depicts the latency of performing the in-line baggage operations. Table 3 displays the normalized overheads of propagating baggage containing 1 value and 60 values (≈ 1kB) respectively during benchmarks run on an instrumented Hadoop system. To measure these overheads, 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. Open suffers the greatest performance hit, with baggage propagation of 60 values incurring 15.9% overhead. But since this is a short CPU-bound request (involving a single read-only lookup), this is within reasonable expectations.

4.1.4 Narrow Waist

Baggage generalizes end-to-end metadata propagation techniques outlined in prior work such as X-Trace[52] and Dapper[89]. This efficiently implements happened-before join. However, it is unlikely that this is the only use of baggage. Because the key/value store baggage provides is completely generic, applications instrumented to propagate baggage with intent orthogonal to Pivot Tracing can nonetheless support Pivot Tracing.



(a) Packing a value (b) Unpacking all values (c) Serialize baggage (d) Deserialize baggage

Figure 7: Latency micro-benchmark results for packing, unpacking, serializing, and deserializing randomly-generated 8-byte values.

	Read8k	Open	Create	Rename
Unmodified	0%	0%	0%	0%
Baggage – 1 Value	0.8%	0.4%	0.6%	0.8%
Baggage – 60 Values	0.82%	15.9%	8.6%	4.1%

Table 3: Latency overheads for HDFS stress test with baggage propagation enabled.

We liken this to devices implementing IP. IP, the so-called "narrow waist" of the internet, is the central layer of the network stack; it is implemented by the link layer, and provides an abstraction with its generic packet payload which encapsulates the transport layer protocols (e.g. TCP, UDP, etc.) that most application-level network protocols are layered upon. Any device that understands IP can participate in a network serving higher-level applications, without understanding what those applications are or even being aware of them.

Since it is generic, Baggage, like IP, could be a "narrow waist" for higher-level applications that require causality tracking such as tracing, taint checking, or resource management. Applications implementing baggage automatically support all other applications implementing and using baggage without understanding those applications or even being aware they exist. We leave further study of the applications of baggage to future work.

4.2 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 (§4.3). Advice has several operations for manipulating tuples through the tracepoint-exported variables, and evaluating $\bowtie$ on tuples produced by other advice at prior tracepoints in the execution.

Table 4 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, so it is guaranteed to terminate, and does not change the memory space of the instrumented system.

This small set of operations comprises the core operations on tuples understood by Pivot Tracing, whereas the query language provides convenience to system operators at the surface. To compile a query to advice, we instantiate one advice

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

Table 4: Primitive *advice* operations supported by Pivot Tracing to generate and aggregate tuples.

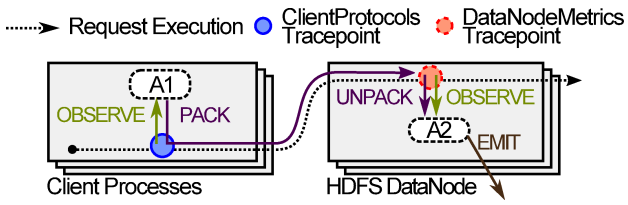


Figure 8: Advice generated for `Q2` from §2: `A1` observes and packs `procName`; `A2` unpacks `procName`, observes `delta`, and emits `(procName, SUM(delta))`.

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`, `GroupBy`, and `GroupByAggregate` are all handled by Emit and Pack.

For example, query `Q2` in §2 compiles to advice `A1` and `A2` for `Client Protocols` and `DataNodeMetrics` respectively:

```
A1:   OBSERVE procName          A2:   OBSERVE delta
      PACK-FIRST procName               UNPACK procName
                                        EMIT procName, SUM(delta)
```

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 8 shows how this advice and the tracepoints interact with the execution of requests in the system.

4.2.1 Implementation

We discuss our implementation of Observe and Filter in section §4.3. Pack and Unpack are built on top of the baggage API, and are merely thin wrappers over **pack** and **unpack**. To Emit, advice instances send tuples to a process-local aggregator, which collects all reported tuples and performs intermediate aggregation (implementing queries containing Aggregate or GroupByAggregate) via a pub/sub server at a regular, configurable interval – by default, one second.

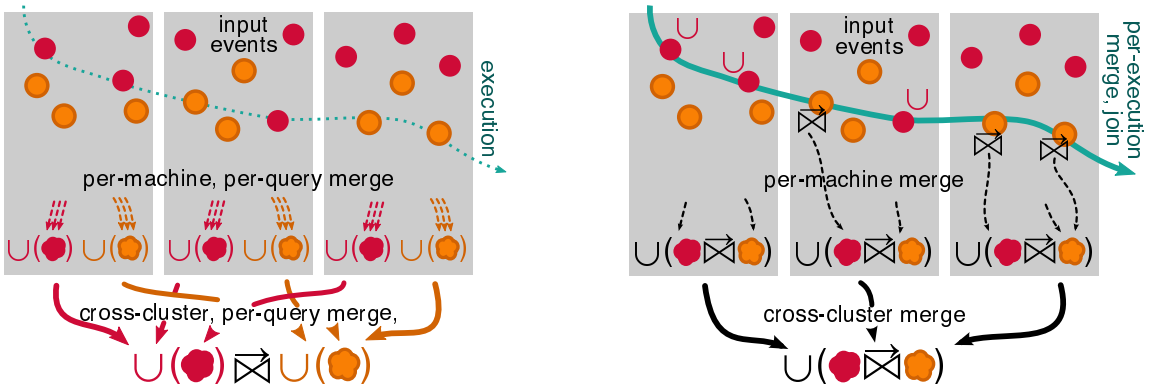
4.2.2 In Context

The Advice API is quite concise, but nonetheless suffices to satisfy many of Pivot Tracing's design goals. Observe provides an abstraction for monitoring a system, and due to its simplicity can be conveniently installed at runtime as outlined in §4.3. Pack allows data claimed from events in one process to be propagated to another. Unpack allows a process to determine what other events must have happened prior in the request's execution. This provides a simple implementation of the happened-before join.

By layering Pack and Unpack on top of baggage, Pivot Tracing efficiently evaluates happened-before joins *in situ* during the execution of a request. Figure 9a depicts the naïve join evaluation strategy that collects events and joins them globally. Figure 9b shows the optimized query evaluation strategy to evaluate joins in-place during request execution.

One metric to assess the cost of a Pivot Tracing query is the number of tuples emitted for global aggregation. The intermediate aggregation use in our Emit implementation substantially reduces traffic for emitted tuples; `Q2` from §2 is reduced from approximately 600 tuples per second to 6 tuples per second from each DataNode.

A second cost metric for Pivot Tracing queries is the number of tuples packed during a request's execution. 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 [51] 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 via Pack and Unpack, Pivot Tracing adds further optimizations for happened-before joins, outlined in Table 5.



(a) Unoptimized query with ⋈ evaluated centrally for the whole cluster.

(b) Optimized query with inline evaluation of ⋈ (↪).

Figure 9: Optimization of ●⋈●. The optimized query often produces substantially less tuple traffic than the unoptimized form.

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

Table 5: Query rewrite rules to join queries P and Q. We push operators as close as possible to source tuples; this reduces the number of tuples that must be propagated in the baggage from P to Q. `Combine` refers to an aggregator's combiner function (*e.g.*, for `Count`, the combiner is `Sum`).

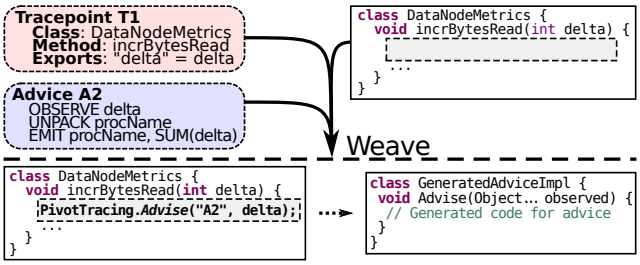


Figure 10: Advice for Q2 is *woven* at the `DataNodeMetrics` tracepoint. Variables exported by the tracepoint are passed when the advice is invoked.

4.3 Dynamic Instrumentation

In this section we describe how our Java prototype installs code implementing advice, an act we call *weaving*[3], into its target system at runtime.

At a high level, 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 10 depicts this process of weaving advice for A2.

4.3.1 Implementation

Agent A Pivot Tracing agent runs within every Pivot Tracing-enabled process and awaits instruction via a pub/sub server to define new tracepoints and advice, and weave advice into tracepoints. Users can specify new tracepoints, define new advice, instruct the agent to weave advice into a tracepoint, instruct the agent to forget about a tracepoint or advice, and instruct the agent to unweave advice from a tracepoint. Pivot Tracing supports pattern matching for more complex tracepoint definitions, for example, all methods of a class specified by a particular interface. This feature is modeled after *pointcuts* from AspectJ [61]. Pivot Tracing also supports instrumenting privileged classes (*e.g.*, `FileInputStream` in §2) by providing an optional agent that can be placed on Java's boot classpath.

Protocol Buffers Pivot Tracing represents tracepoints, advice, and *weaves* (state pairing woven advice with the woven tracepoint) internally (and externally) using an abstract protocol buffer [53] specification. Tracepoints are identified by a query- or operator-supplied unique ID, a class name, method name, method signature, and weave location (e.g. method entry, return, exceptional return, or a specific line number). Advice is identified by a query- or operator-supplied unique ID and the series of commands they perform. Weaves are specified by an advice ID and a tracepoint ID. Our Pivot Tracing agent tracks each of these structures within a map, using the ID as the key (or pair of IDs, in the case of weaves).

Materializing Advice Tracepoints with woven advice invoke the `PivotTracing.Advise` method (cf. Fig. 10), passing tracepoint variables as arguments. This method looks up the woven advice object and invokes each of its operations; OBSERVE constructs a tuple from the provided arguments. We implement FILTER by inserting additional code based on the woven advice's specification, inserting additional code that calls `PivotTracing.Advise` only if the desired conditions are met.

Weaving Advice Our prototype weaves advice at runtime, providing dynamic instrumentation similar to that of DTrace [38] and Fay [51]. Java version 1.5 onwards supports dynamic method body rewriting via the `java.lang.instrument` package. The Pivot Tracing agent programmatically rewrites and reloads class bytecode from within the proces using Javassist [44], which provides a convenient mechanism for compiling literal Java code to bytecode for insertion into a method body. For example, when weaving advice to a tracepoint at a method entry, we invoke the `insertBefore("PivotTracing.Advise(...);")` method on Javassist's method representation, and then reload the entire class using Javassist's `javassist.util.Hotswap` class.[4]

However, Javassist does not provide a simple means to *remove* inserted bytecode from arbitrary locations in a method. Providing a mechanism for removing queries is critical; otherwise Pivot Tracing is not truly dynamic. We resolve this problem by acquiring a fresh copy of the original bytecode for a class and re-computing the woven state of that class each time we weave. Javassist *does* provide a convenient means for doing this with its `javassist.ClassPool`. Classes pulled from a fresh `ClassPool` instance contain the original, uninstrumented definitions of that class.

[3] We borrow this term from aspect-oriented programming [62]. Pivot Tracing and this programming paradigm both combine, i.e. weave, together a *component language* (in the case of Pivot Tracing, the tracepoints of the target system) and an *aspect language* (our advice operations).

[4] This class provides no way to reload only a single method.

This leaves the step of determining which weaves must be re-weaved. To avoid additional space overhead, we do not map class names to weaves but instead algorithmically determine which weaves to re-weave. We model classes, tracepoints, advice, and weaves as a tripartite graph. One independent set of the graph is the set of advice; another is the set of tracepoints; and the third is the set of classes Pivot Tracing knows about (via tracepoints). An edge exists between an advice A and a tracepoint T if and only if there exists a weave specifying that A is woven into T. An edge exists between a tracepoint T and a class C if and only if T is defined over a method of C. When weaving T, we follow its edge to C and then reapply all weaves in the subtree rooted at C. This procedure assumes that the ordering of multiple advice woven into a single tracepoint does not matter. Advice only reads from an instrumented system's address space, so this assumption reduces to an assumption that each distinct advice woven to a tracepoint modifies different parts of the baggage. This assumption is reasonable under the expectation that each of those advice instances supports a distinct query. Figure 11 shows an example of such a graph.

The same procedure can be used to maintain the correct instrumentation state of all classes if an operator redefines a tracepoint or advice instance at runtime.

Unweaving Advice Because of the above algorithm, Pivot Tracing unweaves advice by simply removing the appropriate weaves from its internal data structures. When it recomputes the weave state of the affected classes, the target advice-tracepoint pair will not be included.

4.3.2 Instrumentation

In contrast to baggage, almost no developer overhead whatsoever is required to enable Pivot Tracing to install queries into an application.

```
1 public class MyApp {
2     public static void main(String[] args) {
3         PivotTracing.initialize();
4         ...
5     }
6 }
```

This starts the PivotTracing agent, which listens for queries via a pub/sub server.

We can use AspectJ [61] to generalize this to all main classes of a project:

```
1 public aspect StartPivotTracing {
2     before(): execution(public static void main(String[] args) {
3         PivotTracing.initialize();
4     }
5 }
```

4.3.3 Overheads

JVM HotSwap requires Java's debugging mode to be enabled, which causes some compiler optimizations to be disabled. For practical purposes, however, HotSpot JVM's full-speed debugging is sufficiently optimized that it is possible to run with debugging support always enabled [57]. 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.

Otherwise, 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 by Pivot Tracing. Pivot Tracing has a zero-probe [38] effect: methods are unmodified by default, so tracepoints impose truly zero overhead until advice is woven into them.

5 Pivot Tracing in Action

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 [95, 66]. Our evaluation shows that Pivot Tracing addresses the challenges in §2 when applied to these stack components. In particular, we show that Pivot Tracing:

- is dynamic and extensible to new kinds of analysis (§5.2)
- is scalable and has low developer and execution overhead (§5.3)

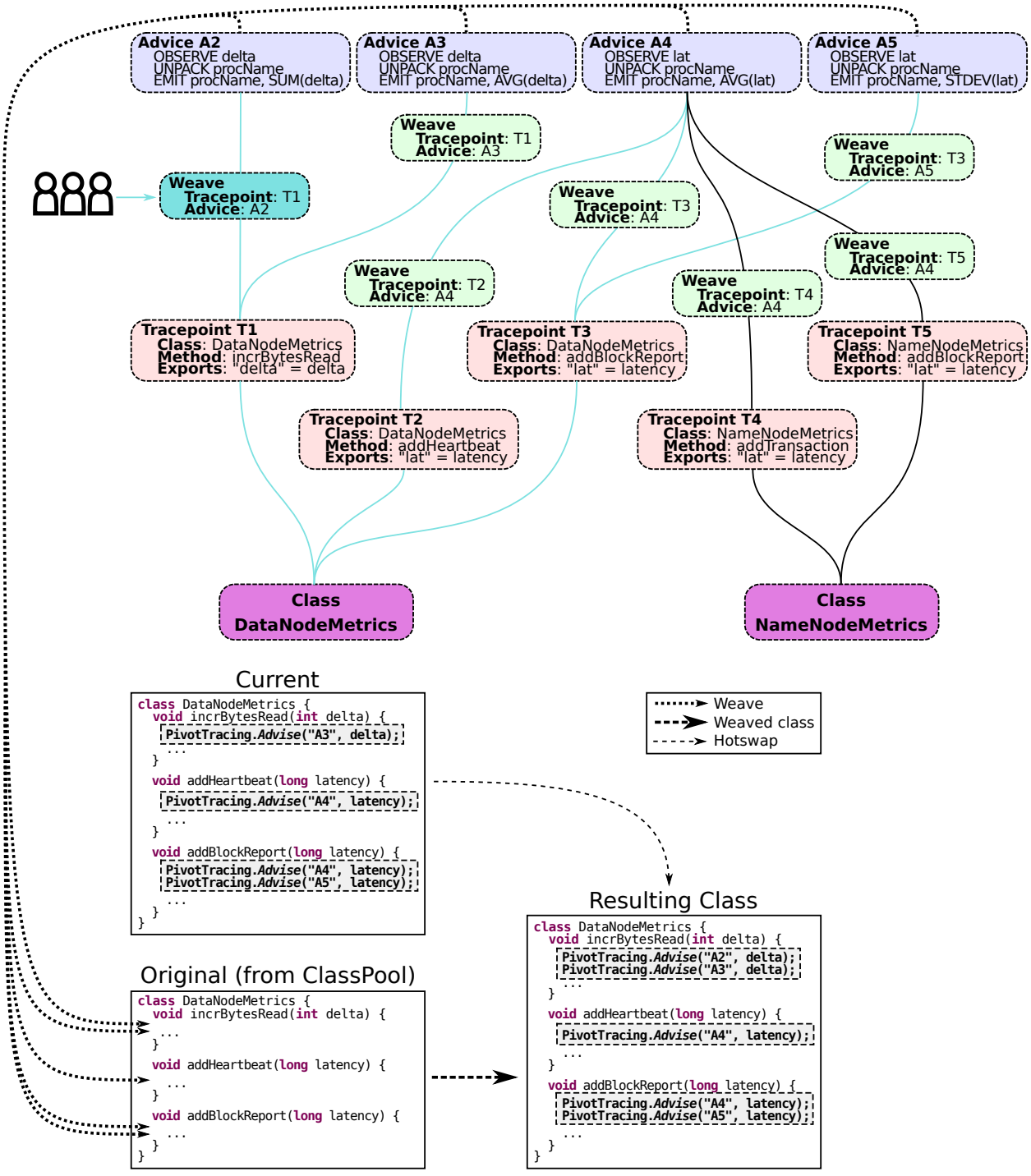


Figure 11: Operators instruct Pivot Tracing to weave advice A2 into tracepoint T1 in a system with the depicted tracepoints, advice (A1 from §4.2 omitted), and weaves. All advice in the cyan tree rooted at `DataNodeMetrics` must be re-weaved.

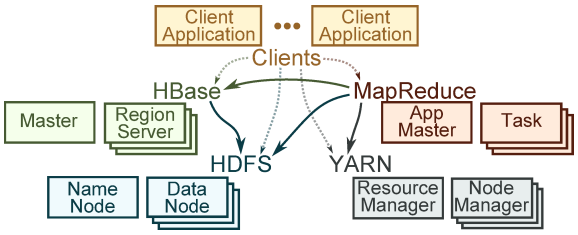


Figure 12: Interactions between systems. Each system comprises several processes on potentially many machines. Typical deployments often co-locate processes from several applications, *e.g.* DataNode, NodeManager, Task and RegionServer processes.

- enables cross-tier analysis between any inter-operating applications (§2, §5.2)
- captures event causality to successfully diagnose root causes (§5.1, §5.2)
- enables insightful analysis with even a very small number of tracepoints (§5.1)

Hadoop Overview We first give a high-level overview of Hadoop, before describing the necessary modifications to enable Pivot Tracing. Figure 12 shows the relevant components of the Hadoop stack.

HDFS [88] is a distributed file system that consists of several DataNodes that store replicated file blocks and a NameNode that manages the filesystem metadata.

HBase [56] 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 [93] 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 We modified these systems to propagate baggage along the execution path of requests. As described in §4.1.1 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, which we discussed in §4.1.2. Each system only required between 50 and 200 lines of manual code 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.

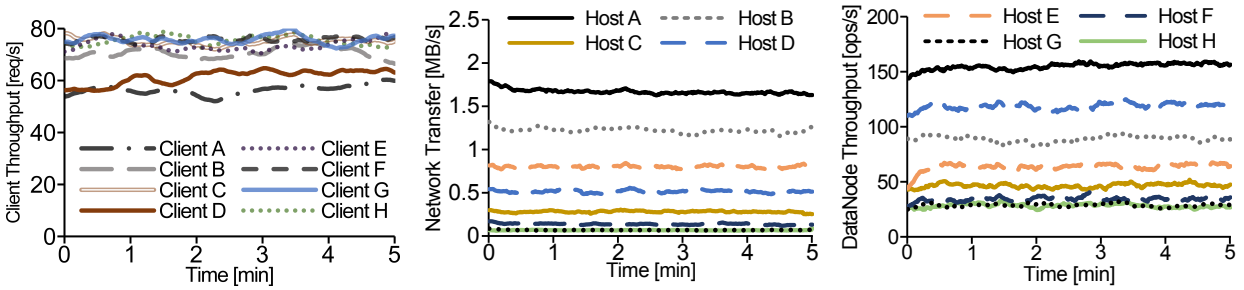
5.1 Case Study: HDFS Replica Selection Bug

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 [24].

HDFS provides file redundancy by decomposing files into blocks and replicating each block onto several machines (typically 3). 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 the some hosts and near zero load on others.

In this scenario we ran 96 stress test clients on an HDFS cluster of 8 DataNodes and 1 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.

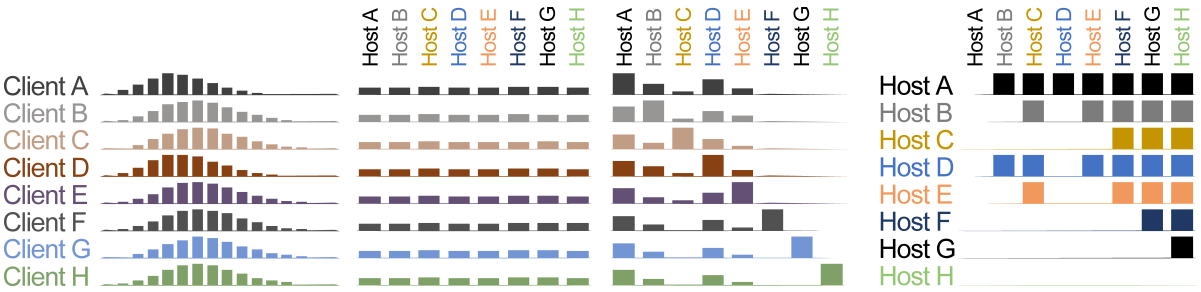
Our investigation of the bug began when we noticed that the stress test clients on hosts A and D had consistently lower request throughput than clients on other hosts, shown in Figure 13a, despite identical machine specifications



(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)

Figure 13: Pivot Tracing query results leading to our discovery of HDFS-6268 [24]. Faulty replica selection logic led clients to prioritize the replicas hosted by particular DataNodes (§5.1).

and setup. We first checked machine level resource utilization on each host, which indicated substantial variation in the network throughput (Figure 13b). We began our diagnosis with Pivot Tracing by first checking to see whether an imbalance in HDFS 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 DN.DataTransferProtocol
    GroupBy dnop.host
    Select dnop.host, COUNT
```

Figure 13c 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 150 ops/sec, while host H has only 25 ops/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 NN.GetBlockLocations
    Join st In StressTest.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 13d 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 NN.GetBlockLocations
    Join st In StressTest.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 13e 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 13c clearly show that they do not. To gain more insight into this inconsistency, our next query relates the results from 13e and 13c:

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

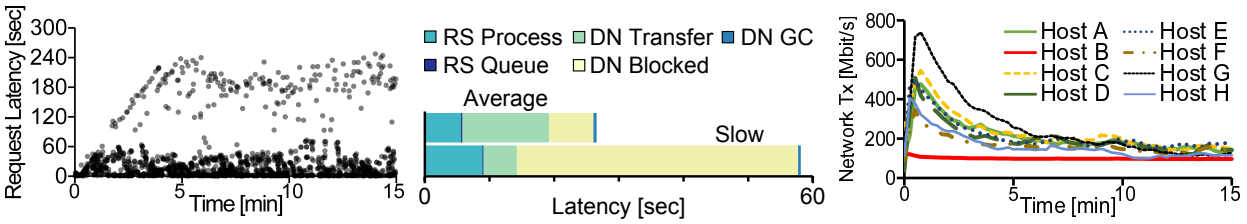
This query measures the frequency that each client selects each DataNode for reading a replica. We plot the results in Figure 13f and see that the clients are clearly favoring particular DataNodes. The strong diagonal is consistent with HDFS client preference for locally-hosted replicas (39% of the time 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 DN.DataTransferProtocol
    Join getloc In NN.GetBlockLocations
                   On getloc -> DNop
    Join st In StressTest.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 13g 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 [24].



(a) HBase Request Latencies (b) Latency Decomposition (c) Per-Machine Network Throughput

Figure 14: (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.

5.2 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 a timestamp at the start of a span of processing (*e.g.*, a function) and unpack it at the end of the span, represented in the following example query:

```
Q8: From end In Function_End
    Join begin In MostRecent(Function_Begin)
                 On begin -> end
    Select end.time – begin.time
```

A query can measure latency in several components and propagate measurements in the baggage, reminiscent of transaction tracking in Timecard [83] and transactional profiling in Whodunit [39].

For example, during the development of Pivot Tracing we encountered an instance of network limplock [50, 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 14a). 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 14b). 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 14c).

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 [95]; and to diagnose an overloaded HDFS NameNode due to exclusive write locking as described in [67].

5.3 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 §4.2.2 reduce the number of propagated tuples to 1 for `Aggregate`, 1 for `Recent`, n for `GroupBy` with n groups, and `N` for `RecentN`. Of the queries in this paper, `Q7` propagates the largest baggage containing the stress test hostname and the location of all 3 file replicas (4 tuples, ≈137 bytes per request).

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 impacts of our queries on CPU bound requests, we revisit the stress test described in §4.1.3, now comparing the following against unmodified HDFS: 1) HDFS with Pivot Tracing enabled; 2) HDFS instrumented with advice from the queries in §5.1 installed; and 3) HDFS with the advice from the queries in §5.2 installed.

Table 6 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. Again, we see the most noticeable overhead in Open and Create, very short CPU-bound requests comprising only a single read-only lookup. Rename does not trigger any advice for the queries from §5.1, but does trigger advice for the queries from §5.2. Overheads of 0.3% and 5.5% respectively reflect this difference.

	Read8k	Open	Create	Rename
Unmodified	0%	0%	0%	0%
PivotTracing Enabled	0.3%	0.3%	<0.1%	0.2%
Queries – §5.1	1.5%	4.0%	6.0%	0.3%
Queries – §5.2	1.9%	14.3%	8.2%	5.5%

Table 6: Latency overheads for HDFS stress test with Pivot Tracing enabled and full queries enabled, as described in §5.3.

6 Related Work

In §2 we described the challenges with troubleshooting tools that Pivot Tracing addresses, and several previous systems. We 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 [51, 38, 80] with causal information that crosses process and system boundaries. Pivot Tracing's happened-before join is an example of a θ-join [82] (where the condition is happened-before), and as a special case of path queries in graph databases [96]. Differently from offline queries in a pre-stored graph, Pivot Tracing efficiently evaluates $\overrightarrow{\bowtie}$ at runtime.

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 [75, 97, 78, 60] and static analysis [100] to extract better information from logs; automatic enrichment of existing log statements to ease troubleshooting [99]; end-to-end tracing systems to capture the happened-before relationship between events [52, 89]; state-monitoring systems to track system-level resources and indicate the health of a cluster [70]; and aggregation systems to collect and summarize application-level monitoring data [92, 64]. Wang *et al.* provide a comprehensive overview of datacenter troubleshooting tools in [94]. These tools suffer from the challenges of pre-defined information outlined in §2.

Troubleshooting and Root-Cause Diagnosis Several offline techniques have been proposed to infer execution models from logs [34, 69, 45, 100] and diagnose performance problems [75, 87, 63]. End-to-end tracing frameworks exist both in academia [33, 39, 52, 85, 90] and in industry [89, 58, 30, 91, 46] and have been used for a variety of purposes, including diagnosing anomalous requests whose structure or timing deviate from the norm [33, 42, 87, 79]; diagnosing steady-state problems problems that manifest across many requests [85, 90, 52, 89, 87]; identifying slow components and functions [39, 89, 69]; and modelling workloads and resource usage [33, 69, 90]. Recent work has extended these techniques to online profiling and analysis [101, 73, 74, 72, 55].

7 Discussion

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

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 (§5), especially for new systems. 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 [51]. Kernel-level execution context propagation [36, 84, 40] can provide language-independent access to baggage variables.

Even though we evaluated Pivot Tracing on an 8-node cluster in this paper, 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. In particular, it would be an interesting exercise to integrate the happened-before join with Fay or DTrace.

We motivated Pivot Tracing with system operators in mind, but it has several applications that could assist system developers as well. User applications could be instrumented to allow developers to collect specific statistics in real

time that would assist with debugging the system and developing updates or new features. Applications with a slow redeployment cycle would especially benefit from dynamic instrumentation.

7.1 Misuse

Section §4.2 describes the simple algorithm for compiling queries to advice, and the simplicity of our advice operations; section §5 demonstrates (albeit transitively) the power of those operations. While users are restricted to these 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. A comprehensive security analysis is beyond the scope of this paper, but we do note that certain simple safeguards could be taken against misuse. Digital signatures might be used to enforce that only trusted administrators can define tracepoints, advice, and specify what to weave; tools in static analysis might be used to ensure that injected code does not modify system state or risk unbounded recursion.

The temporal filters `FIRST`, `RECENT`, `FIRSTN`, and `RECENTN` all provide a mechanism for explicitly bounding the size of the baggage. However, there is no inherent bound, and a poorly-constructed query would potentially accumulate 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. In our experience, even without the temporal filters 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 [52], Stardust [90], or Dapper [89]. 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.

It is likely the case that system operators do not need the full expressiveness that Pivot Tracing offers. User studies conducted with Pivot Tracing in a contained, non-production cluster could help system developers learn what kinds of queries operators need their applications to support. Support for those queries by default rather than dynamically mitigates this security concern.

8 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] Apache HBase Reference Guide. http://hbase.apache.org/book.html. [Online; accessed 25-Feb-2015].

[2] HADOOP-6599 Split RPC metrics into summary and detailed metrics. https://issues.apache.org/jira/browse/HADOOP-6599. [Online; accessed 25-Feb-2015].

[3] HADOOP-6859 Introduce additional statistics to FileSystem. https://issues.apache.org/jira/browse/HADOOP-6859. [Online; accessed 25-Feb-2015].

[4] HBASE-11559 Add dumping of DATA block usage to the BlockCache JSON report. https://issues.apache.org/jira/browse/HBASE-11559. [Online; accessed 25-Feb-2015].

[5] HBASE-12364 API for query metrics. https://issues.apache.org/jira/browse/HBASE-12364. [Online; accessed 25-Feb-2015].

[6] HBASE-12424 Finer grained logging and metrics for split transaction. https://issues.apache.org/jira/browse/HBASE-12424. [Online; accessed 25-Feb-2015].

[7] HBASE-12477 Add a flush failed metric. https://issues.apache.org/jira/browse/HBASE-12477. [Online; accessed 25-Feb-2015].

[8] HBASE-12494 Add metrics for blocked updates and delayed flushes. https://issues.apache.org/jira/browse/HBASE-12494. [Online; accessed 25-Feb-2015].

[9] HBASE-12496 A blockedRequestsCount metric. https://issues.apache.org/jira/browse/HBASE-12496. [Online; accessed 25-Feb-2015].

[10] HBASE-12574 Update replication metrics to not do so many map look ups. https://issues.apache.org/jira/browse/HBASE-12574. [Online; accessed 25-Feb-2015].

[11] HBASE-2257 [stargate] multiuser mode. https://issues.apache.org/jira/browse/HBASE-2257. [Online; accessed 25-Feb-2015].

[12] HBASE-4038 Hot Region : Write Diagnosis. https://issues.apache.org/jira/browse/HBASE-4038. [Online; accessed 25-Feb-2015].

[13] HBASE-4145 Provide metrics for hbase client. https://issues.apache.org/jira/browse/HBASE-4145. [Online; accessed 25-Feb-2015].

[14] HBASE-4169 Add per-disk latency metrics to DataNode. https://issues.apache.org/jira/browse/HDFS-4169. [Online; accessed 25-Feb-2015].

[15] HBASE-4219 Add Per-Column Family Metrics. https://issues.apache.org/jira/browse/HBASE-4219. [Online; accessed 25-Feb-2015].

[16] HBASE-5253 Add requesting user's name to PathBasedCacheEntry. https://issues.apache.org/jira/browse/HDFS-5253. [Online; accessed 25-Feb-2015].

[17] HBASE-6093 Expose more caching information for debugging by users. https://issues.apache.org/jira/browse/HDFS-6093. [Online; accessed 25-Feb-2015].

[18] HBASE-6292 Display HDFS per user and per group usage on webUI. https://issues.apache.org/jira/browse/HDFS-6292. [Online; accessed 25-Feb-2015].

[19] HBASE-7390 Provide JMX metrics per storage type. https://issues.apache.org/jira/browse/HDFS-7390. [Online; accessed 25-Feb-2015].

[20] HBASE-7958 Statistics per-column family per-region. https://issues.apache.org/jira/browse/HBASE-7958. [Online; accessed 25-Feb-2015].

[21] HBASE-8370 Report data block cache hit rates apart from aggregate cache hit rates. https://issues.apache.org/jira/browse/HBASE-8370. [Online; accessed 25-Feb-2015].

[22] HBASE-8868 add metric to report client shortcircuit reads. https://issues.apache.org/jira/browse/HBASE-8868. [Online; accessed 25-Feb-2015].

[23] HBASE-9722 need documentation to configure HBase to reduce metrics. https://issues.apache.org/jira/browse/HBASE-9722. [Online; accessed 25-Feb-2015].

[24] HDFS-6268 Better sorting in NetworkTopology.pseudoSortByDistance when no local node is found. https://issues.apache.org/jira/browse/HDFS-6268. [Online; accessed 25-Feb-2015].

[25] MESOS-1949 All log messages from master, slave, executor, etc. should be collected on a per-task basis. https://issues.apache.org/jira/browse/MESOS-1949. [Online; accessed 25-Feb-2015].

[26] MESOS-2157 Add /master/slaves and /master/frameworks/{framework}/tasks/{task} endpoints. https://issues.apache.org/jira/browse/MESOS-2157. [Online; accessed 25-Feb-2015].

[27] Apache accumulo. http://accumulo.apache.org/. Last accessed March 2015.

[28] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *Proc. SOSP '03* (New York, NY, USA, 2003), ACM Press.

[29] Almeida, P. S., Baquero, C., and Fonte, V. Interval tree clocks: A logical clock for dynamic systems. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 259–274.

[30] Appneta traceview. http://appneta.com. July, 2013.

[31] Attariyan, M., Chow, M., and Flinn, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI* (2012), pp. 307–320.

[32] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using magpie for request extraction and workload modelling. In *OSDI* (2004), vol. 4, pp. 18–18.

[33] Barham, P., Isaacs, R., Mortier, R., and Narayanan, D. Magpie: Online modelling and performance-aware systems. In *HotOS* (2003), vol. 9.

[34] Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. Inferring models of concurrent systems from logs of their behavior with CSight. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India, June 4–6, 2014), pp. 468–479.

[35] Bodik, P. 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.

[36] Buch, I., and Park, R. Improve debugging and performance tuning with etw. *MSDN Magazine,[Online],[Accessed: 01.01. 2012], Avaliable from: http://msdn. microsoft. com/en-us/magazine/cc163437. aspx* (2007).

[37] Cantrill, B. Hidden in plain sight. *Queue 4*, 1 (Feb. 2006), 26–36.

[38] Cantrill, B., Shapiro, M. W., Leventhal, A. H., et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track* (2004), pp. 15–28.

[39] Chanda, A., Cox, A. L., and Zwaenepoel, W. Whodunit: Transactional profiling for multi-tier applications. *ACM SIGOPS Operating Systems Review 41*, 3 (2007), 17–30.

[40] Chanda, A., Elmeleegy, K., Cox, A. L., and Zwaenepoel, W. Causeway: System support for controlling and analyzing the execution of multi-tier applications. In *Proc. Middleware 2005* (November 2005), pp. 42–59.

[41] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[42] Chen, M. Y., Accardi, A., Kiciman, E., Patterson, D. A., Fox, A., and Brewer, E. A. Path-based failure and evolution management. In *NSDI* (2004).

[43] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2002), DSN '02, IEEE Computer Society, pp. 595–604.

[44] Chiba, S. Javassist: Java bytecode engineering made simple. *Java DeveloperâĂŹs Journal 9*, 1 (2004).

[45] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 217–231.

[46] Compuware dynatrace purepath. `http://www.compuware.com`. Accessed July, 2013.

[47] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[48] Couckuyt, J., Davies, P., and Cahill, J. Multiple chart user interface, June 14 2005. US Patent 6,906,717.

[49] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[50] Do, T., Hao, M., Leesatapornwongsa, T., Patana-anake, T., and Gunawi, H. S. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 14.

[51] Erlingsson, Ú., Peinado, M., Peter, S., Budiu, M., and Mainar-Ruiz, G. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS) 30*, 4 (2012), 13.

[52] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association.

[53] Google Protocol Buffers. `http://code.google.com/p/protobuf/`.

[54] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1*, 1 (1997), 29–53.

[55] Guo, Z., Zhou, D., Lin, H., Yang, M., Long, F., Deng, C., Liu, C., and Zhou, L. G2: A graph processing system for diagnosing distributed systems. In *USENIX Annual Technical Conference* (2011).

[56] HBase. `http://hbase.apache.org`.

[57] The Java HotSpot Performance Engine Architecture. `http://www.oracle.com/technetwork/java/whitepaper-135217.html`.

[58] Apache HTrace. `http://htrace.incubator.apache.org/`. Last accessed March 2015.

[59] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on* (2010), IEEE, pp. 41–51.

[60] Kavulya, S. P., Daniels, S., Joshi, K., Hiltunen, M., Gandhi, R., and Narasimhan, P. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)* (June 2012).

[61] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (London, UK, UK, 2001), ECOOP '01, Springer-Verlag, pp. 327–353.

[62] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (June 1997), LNCS 1241, Springer-Verlag.

[63] Kim, M., Sumbaly, R., and Shah, S. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review 41*, 1 (2013), 93–104.

[64] Ko, S. Y., Yalagandula, P., Gupta, I., Talwar, V., Milojicic, D., and Iyer, S. Moara: flexible and scalable group-based querying system. In *Middleware 2008*. Springer, 2008, pp. 408–428.

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

[66] Laub, B., Wang, C., Schwan, K., and Huneycutt, C. Towards combining online & offline management for big data applications. In *11th International Conference on Autonomic Computing (ICAC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 121–127.

[67] Mace, J., Bodik, P., Musuvathi, M., and Fonseca, R. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI '15: Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation* (May 2015), USENIX Association.

[68] Mace, J., Roelke, R., and Fonseca, R. Pivot tracing: Dynamic causal monitoring for distributed systems.

[69] Mann, G., Sandler, M., Krushevskaja, D., Guha, S., and Even-Dar, E. Modeling the parallel execution of black-box services. *USENIX/HotCloud* (2011).

[70] Massie, M. L., Chun, B. N., and Culler, D. E. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 30*, 7 (2004), 817–840.

[71] Meijer, E., Beckman, B., and Bierman, G. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2006), SIGMOD '06, ACM, pp. 706–706.

[72] Mi, H., Wang, H., Chen, Z., and Zhou, Y. Automatic detecting performance bugs in cloud computing systems via learning latency specification model. In *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on* (2014), IEEE, pp. 302–307.

[73] Mi, H., Wang, H., Zhou, Y., Lyu, M. R., and Cai, H. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems 24*, 6 (2013), 1245–1255.

[74] Mi, H., Wang, H., Zhou, Y., Lyu, M. R.-T., Cai, H., and Yin, G. An online service-oriented performance profiling tool for cloud computing systems. *Frontiers of Computer Science 7*, 3 (2013), 431–445.

[75] Nagaraj, K., Killian, C. E., and Neville, J. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI* (2012), pp. 353–366.

[76] Continuation-local storage. `https://github.com/othiym23/node-continuation-local-storage`. Last accessed March 2015.

[77] Oliner, A., Ganapathi, A., and Xu, W. Advances and challenges in log analysis. *Communications of the ACM 55*, 2 (2012), 55–61.

[78] Oliner, A., Kulkarni, A., and Aiken, A. Using correlated surprise to infer shared influence. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on* (June 2010), pp. 191–200.

[79] Ostrowski, K., Mann, G., and Sandler, M. Diagnosing latency in multi-tier black-box services. In *LADIS* (2011).

[80] Prasad, V., Cohen, W., Eigler, F. C., Hunt, M., Keniston, J., and Chen, B. Locating system problems using dynamic instrumentation. In *Ottawa Linux Symposium (OLS)* (2005).

[81] Rabkin, A., and Katz, R. H. How hadoop clusters break. *Software, IEEE 30*, 4 (2013), 88–94.

[82] Ramakrishnan, R., and Gehrke, J. *Database Management Systems*, 2nd ed. Osborne/McGraw-Hill, Berkeley, CA, USA, 2000.

[83] Ravindranath, L., Padhye, J., Mahajan, R., and Balakrishnan, H. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 85–100.

[84] Reumann, J., and Shin, K. G. Stateful distributed interposition. *ACM Trans. Comput. Syst. 22*, 1 (2004), 1–48.

[85] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2006), USENIX Association.

[86] Sambasivan, R. R., Fonseca, R., Shafer, I., and Ganger, G. R. So, you want to trace your distributed system? Key design insights from years of practical experience. Tech. Rep. CMU-PDL-14-102, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA 15213-3890, April 2014.

[87] Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. Diagnosing performance changes by comparing request flows. In *NSDI* (2011).

[88] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (2010), IEEE, pp. 1–10.

[89] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. *Google research* (2010).

[90] Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. Stardust: tracking activity in a distributed storage system. *SIGMETRICS Perform. Eval. Rev. 34*, 1 (2006), 3–14.

[91] Twitter Zipkin. `http://twitter.github.io/zipkin/`. Last accessed March 2015.

[92] Van Renesse, R., Birman, K. P., and Vogels, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS) 21*, 2 (2003), 164–206.

[93] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16.

[94] Wang, C., Kavulya, S. P., Tan, J., Hu, L., Kutare, M., Kasick, M., Schwan, K., Narasimhan, P., and Gandhi, R. Performance troubleshooting in data centers: an annotated bibliography? *ACM SIGOPS Operating Systems Review 47*, 3 (2013), 50–62.

[95] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., Talwar, V., Wolf, M., and Huneycutt, C. Vscope: middleware for troubleshooting time-sensitive data center applications. In *Middleware 2012*. Springer, 2012, pp. 121–141.

[96] Wood, P. T. Query languages for graph databases. *SIGMOD Rec. 41*, 1 (Apr. 2012), 50–60.

[97] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 117–132.

[98] Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 159–172.

[99] Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. Improving software diagnosability via log enhancement. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems* (March 2011).

[100] Zhao, X., Zhang, Y., Lion, D., Faizan, M., Luo, Y., Yuan, D., and Stumm, M. lprof: A nonintrusive request flow profiler for distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (2014).

[101] Zhou, J., Chen, Z., Mi, H., and Wang, J. Mtracer: a trace-oriented monitoring framework for medium-scale distributed systems. In *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on* (2014), IEEE, pp. 266–271.