Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems

Jonathan Mace Ryan Roelke Rodrigo Fonseca
Brown University

Abstract

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

1. Introduction

Monitoring and troubleshooting distributed systems is hard. The potential problems are myriad: hardware and software failures, misconfigurations, hot spots, aggressive tenants, or even simply unrealistic user expectations. Despite the complex, varied, and unpredictable nature of these problems, most monitoring and diagnosis tools commonly used today – logs, counters, and metrics – have at least two fundamental limitations: what gets recorded is defined a priori, at development or deployment time, and the information is captured in a component- or machine-centric way, making it extremely difficult to correlate events that cross these boundaries.

While there has been great progress in using machine learning techniques [60, 74, 76, 95] and static analysis [97, 98] to improve the quality of logs and their use in troubleshooting, they carry an inherent tradeoff between recall and overhead, as what gets logged must be defined a priori. Similarly, with monitoring, performance counters may be too coarse-grained [73]; and if a user requests additional metrics, a cost-benefit tug of war with the developers can ensue [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. This limitation is fundamental, as neither Fay nor DTrace can affect the monitored system to propagate the monitoring context across these boundaries.

In this paper we combine dynamic instrumentation with causal tracing techniques [39, 52, 87] to fundamentally increase the power and applicability of either technique. We present Pivot Tracing, a monitoring framework that gives operators and users, at runtime, the ability to obtain an arbitrary metric at one point of the system, while selecting, filtering, and grouping by events meaningful at other parts of the system, even when crossing component or machine boundaries.

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

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

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10…$15.00.
http://dx.doi.org/10.1145/2815400.2815415

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

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

In summary, this 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.

2. Motivation

2.1 Pivot Tracing in Action

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

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

We run the following client applications:

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

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

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

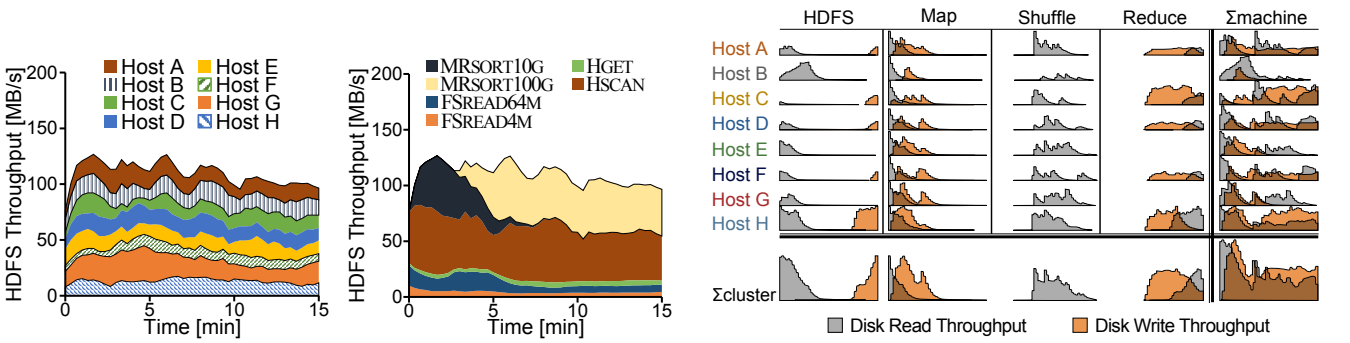
This query causes each machine to aggregate the `delta` argument each time `incrBytesRead` is invoked, grouping by the host name. Each machine reports its local aggregate every second, from which we produce the time series in Figure 1a.

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



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

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

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

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.

across columns yields per-system totals, and the bottom right corner shows the global totals. In this example, the client application presents a further dimension along which we could present statistics.

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

2.2 Pivot Tracing Overview

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

Queries in Pivot Tracing refer to variables exposed by one or more *tracepoints* – places in the system where Pivot Tracing can insert instrumentation. Tracepoint definitions are not part of the system code, but are rather instructions on where and how to change the system to obtain the exported identifiers. Tracepoints in Pivot Tracing are similar to pointcuts from aspect-oriented programming [62], and can refer to arbitrary interface/method signature combinations. Tracepoints are defined by someone with knowledge of the system, maybe a

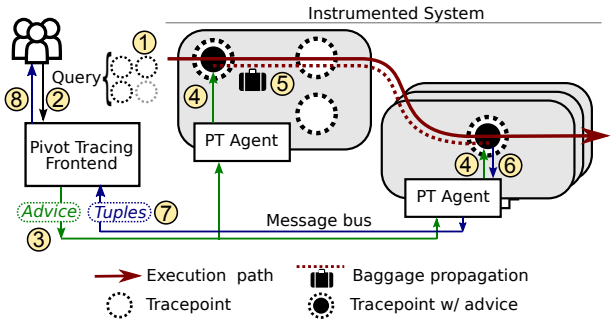


Figure 2: Pivot Tracing overview (§2.2)

developer or expert operator, and define the vocabulary for queries (①). They can be defined and installed at any point in time, and can be shared and disseminated.

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

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

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

2.3 Monitoring and Troubleshooting Challenges

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

One size does not fit all Problems in distributed systems are complex, varied, and unpredictable. By default, the information required to diagnose an issue may not be reported by the system or contained in system logs. Current approaches tie logging and statistics mechanisms into the development path of products, where there is a mismatch between the expectations and incentives of the developer and the needs of operators and users. Panelists at SLAML [35] discussed the important need to "close the loop of operations back to developers". According to Yuan *et al.* [97], 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, 7–9, 17, 22]; to request changes to aggregation methods [10, 21, 23]; and to request new breakdowns of existing metrics [2, 5, 6, 11–16, 18–21, 25]. Many issues remain unresolved due to developer pushback [12, 16, 17, 19, 20] or inertia [5, 7, 8, 14, 18, 22, 23, 25]. Even simple cases of misconfiguration are frequently unreported by error logs [96].

Eventually, applications may be updated to record more information, but this has effects both in performance and information overload. Users must pay the performance overheads of any systems that are enabled by default, regardless of their utility. For example, HBase SchemaMetrics were introduced to aid developers, but all users of HBase pay the 10% performance overhead they incur [21]. The HBase user guide [1] carries the following warning for users wishing to integrate with Ganglia [69]: "By default, HBase emits a large number of metrics per region server. Ganglia may have difficulty processing all these metrics. Consider increasing the capacity of the Ganglia server or reducing the number of metrics emitted by HBase."

The glut of recorded information presents a "needle-in-a-haystack" problem to users [79]; while a system may expose information relevant to a problem, *e.g.* in a log, extracting this information requires system familiarity developed over a long period of time. For example, Mesos cluster state is exposed via a single JSON endpoint and can become massive, even if a client only wants information for a subset of the state [26].

Dynamic instrumentation frameworks such as Fay [51], DTrace [38], and SystemTap [78] address these limitations, by allowing almost arbitrary instrumentation to be installed dynamically at runtime, and have proven extremely useful in the diagnosis of complex and subtle system problems [37]. Because of their side-effect-free nature, however, they are limited in the extent to which probes may share information with each other. In Fay, only probes in the same address space can share information, while in DTrace the scope is limited to a single operating system instance.

Crossing Boundaries This brings us to the second challenge Pivot Tracing addresses. In multi-tenant, multi-application stacks, the root cause and symptoms of an issue may appear in different processes, machines, and application tiers, and may be visible to different users. A user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. For example, HBASE-4145 [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. MESOS-1949 [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."

Prior research has presented mechanisms to observe or infer the relationship between events (§7) and studies of logging practices conclude that end-to-end tracing would be helpful in navigating the logging issues they outline [75, 79]. A variety of these mechanisms have also materialized in production systems: for example, Google's Dapper [87], Apache's HTrace [58], Accumulo's Cloudtrace [27], and Twitter's Zipkin [89]. These approaches can obtain richer information about particular executions than component-centric logs or metrics alone, and have found uses in troubleshooting, debugging, performance analysis and anomaly detection, for example. However, most of these systems record or reconstruct traces of execution for offline analysis, and thus share the problems above with the first challenge, concerning what to record.

3. Design

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

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

Operation	*Description*	*Example*
From	Use input tuples from a set of tracepoints	`From e In RPCs`
Union (⋃)	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

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.

A tracepoint identifies one or more locations in the system code where Pivot Tracing can install and run instrumentation. Tracepoints export named variables that can be accessed by instrumentation. Figure 5 shows the specification of one of the tracepoints in Q2 from §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 *happened-before join* query operator ($\overrightarrow{\bowtie}$).

Happened-before Joins A key contribution of Pivot Tracing is the happened-before join query operator. Happened-before join enables the tuples from two Pivot Tracing queries to be joined based on Lamport's happened before relation, $\rightarrow$ [65]. For events a and b occurring anywhere in the system, we say that a happened before b and write $a \rightarrow b$ if the occurrence of event a causally preceded the occurrence of event b and they occurred as part of the execution of the same request.[1]

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

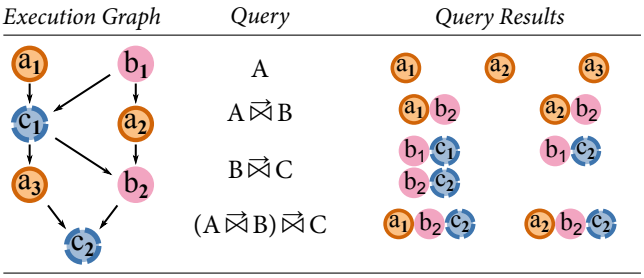


Figure 3: An example execution that triggers tracepoints A, B and C several times. We show several Pivot Tracing queries and the tuples that would result for each.

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 3 shows an example execution triggering tracepoints A, B, and C several times, and outlines the tuples that would be produced for this execution by different queries.

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.

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

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

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 2: Primitive operations supported by Pivot Tracing advice for generating and aggregating tuples as defined in §3.

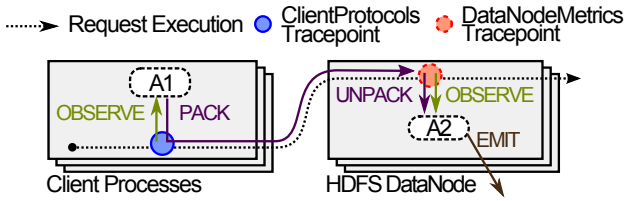


Figure 4: Advice generated for Q2 from §2: A1 observes and packs procName; A2 unpacks procName, observes delta, and emits (procName, SUM(delta))

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

Query Q2 in §2 compiles to advice A1 and A2 for ClientProtocols and DataNodeMetrics respectively:

```
A1: OBSERVE procName
    PACK-FIRST procName

A2: OBSERVE delta
    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 4 shows how this advice and the tracepoints interact with the execution of requests in the system.

To compile a query to advice, we instantiate one advice specification for a From clause and add an Observe operation for the tracepoint variables used in the query. For each Join clause, we add an Unpack operation for the variables that originate from the joined query. We recursively generate

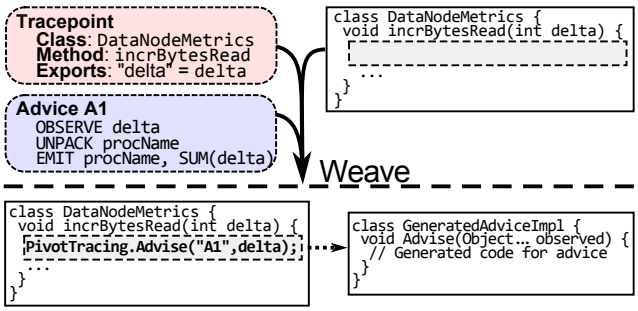


Figure 5: Advice for Q2 is *woven* at the DataNodeMetrics tracepoint. Variables exported by the tracepoint are passed when the advice is invoked.

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. §4 outlines several query rewriting optimizations for implementing $\overrightarrow{\bowtie}$.

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

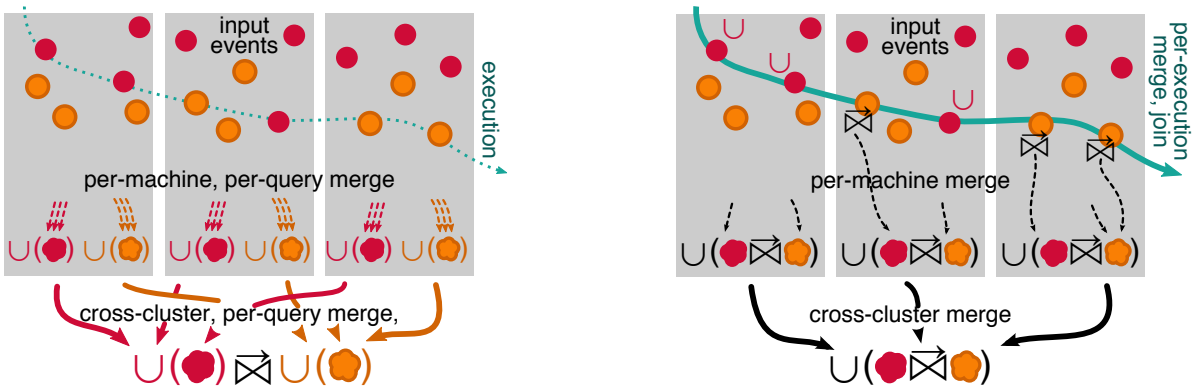
4. Pivot Tracing Optimizations

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

Unoptimized joins The naïve evaluation strategy for happened-before join is that of an equijoin ($\bowtie$) or θ-join ($\bowtie_\theta$ [80]), requiring tuples to be aggregated globally across the cluster prior to evaluating the join. Temporal joins as implemented by Magpie [32], for example, are expensive because they implement this evaluation strategy (§7). Figure 6a illustrates this approach for happened-before join.

Baggage Pivot Tracing enables inexpensive happened-before joins by providing the *baggage* abstraction. Baggage is a per-request container for tuples that is propagated alongside a request as it traverses thread, application and machine boundaries. Pack and Unpack store and retrieve tuples from the current request's baggage. Tuples follow the request's execution path and therefore explicitly capture the happened-before relationship.

Baggage is a generalization of end-to-end metadata propagation techniques outlined in prior work such as X-Trace [52] and Dapper [87]. Using baggage, Pivot Tracing efficiently evaluates happened-before joins *in situ* during the execution of a request. Figure 6b shows the optimized query evaluation strategy to evaluate joins in-place during request execution.



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

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

Figure 6: Optimization of ● ⋈ ●. The optimized query often produces substantially less tuple traffic than the unoptimized form.

Tuple Aggregation One metric to assess the cost of a Pivot Tracing query is the number of tuples emitted for global aggregation. To reduce this cost, Pivot Tracing performs intermediate aggregation for queries containing `Aggregate` or `GroupByAggregate`. Pivot Tracing aggregates the emitted tuples within each process and reports results globally at a regular interval, *e.g.*, once per second. Process-level aggregation substantially reduces traffic for emitted tuples; Q2 from §2 is reduced from approximately 600 tuples per second to 6 tuples per second from each DataNode.

Query Optimizations 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 Fay [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 in the baggage, Pivot Tracing adds further optimizations for happened-before joins, outlined in Table 3.

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

In cases where too many tuples are packed in the baggage, Pivot Tracing could revert to an alternative query plan, where all tuples are emitted instead of packed, and the baggage size is kept constant by storing only enough information to reconstruct the causality, *a la* X-Trace [52], Stardust [88], or Dapper [87]. 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.

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 3: 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`)

5. Implementation

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

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

Dynamic Instrumentation 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`

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

Table 4: Baggage API for Pivot Tracing Java implementation. PACK operations store tuples in the baggage. API methods are static and only allow interaction with the current execution's baggage.

package. The Pivot Tracing agent programmatically rewrites and reloads class bytecode from within the process using Javassist [44].

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

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

Pivot Tracing only makes system modifications when advice is woven into a tracepoint, so inactive tracepoints incur no overhead. Executions that do not trigger the tracepoint are unaffected by Pivot Tracing. Pivot Tracing has a zero-probe effect: methods are unmodified by default, so tracepoints impose truly zero overhead until advice is woven into them.

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

Baggage is lazily serialized and deserialized using protocol buffers [53]. This minimizes the overhead of propagating baggage through applications that do not actively participate in a query, since baggage is only deserialized when an application attempts to pack or unpack tuples. Serialization costs are only incurred for modified baggage at network or application boundaries.

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

Branches and Versioning In order to preserve the happened-before relation correctly within a request, Pivot Tracing must handle executions that branch and rejoin. Tuples packed by one branch cannot be visible to any other branch until the branches rejoin.

To handle this, we implement a versioning scheme for baggage using *interval tree clocks* [29]. Internally, baggage actually maintains one or more versioned instances each with a globally unique interval tree ID. Only one of the versioned instances is considered 'active' for any branch of the execution.

Whenever an execution branches, the interval tree ID of the active instance is divided into two globally unique, non-overlapping IDs, and each branch is assigned one of these new IDs. Each side of the branch receives a copy of the baggage then creates a new active instance using its half of the divided ID.

When a tuple is packed, it is placed into the active instance for its branch. To unpack from baggage that has multiple instances, tuples are unpacked from each instance then combined according to query logic.

When two branches of an execution rejoin, a new active baggage instance is constructed by merging the contents of the active instances on each side of the branch. The ID of the new active baggage is assigned by joining the IDs of each side of the branch. The inactive instances from each branch are copied, and duplicates are discarded.

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

6. Evaluation

In this section we evaluate Pivot Tracing in the context of the Hadoop stack. We have instrumented four open-source systems – HDFS, HBase, Hadoop MapReduce, and YARN – that are widely used in production today. We present several case studies where we used Pivot Tracing to successfully diagnose root causes, including real-world issues we encountered in our cluster and experiments presented in prior work [66, 93]. 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 (§6.2)
- is scalable and has low developer and execution overhead (§6.3)
- enables cross-tier analysis between any inter-operating applications (§2, §6.2)

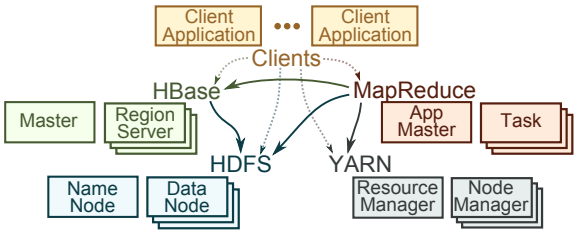


Figure 7: Interactions between systems. Each system comprises several processes on potentially many machines. Typical deployments often co-locate processes from several applications, *e.g.* DataNode, NodeManager, Task and RegionServer processes.

- captures event causality to successfully diagnose root causes (§6.1, §6.2)
- enables insightful analysis with even a very small number of tracepoints (§6.1)

Hadoop Overview We first give a high-level overview of Hadoop, before describing the necessary modifications to enable Pivot Tracing. Figure 7 shows the relevant components of the Hadoop stack.

HDFS [86] 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 [91] is a container manager to run user-provided processes across the cluster. NodeManager processes run on each machine to manage local containers, and a centralized ResourceManager manages the overall cluster state and requests from users.

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

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

6.1 Case Study: HDFS Replica Selection Bug

In this section we describe our discovery of a replica selection bug in HDFS that resulted in uneven distribution of load to replicas. After identifying the bug, we found that it had been recently reported and subsequently fixed in an upcoming HDFS version [24].

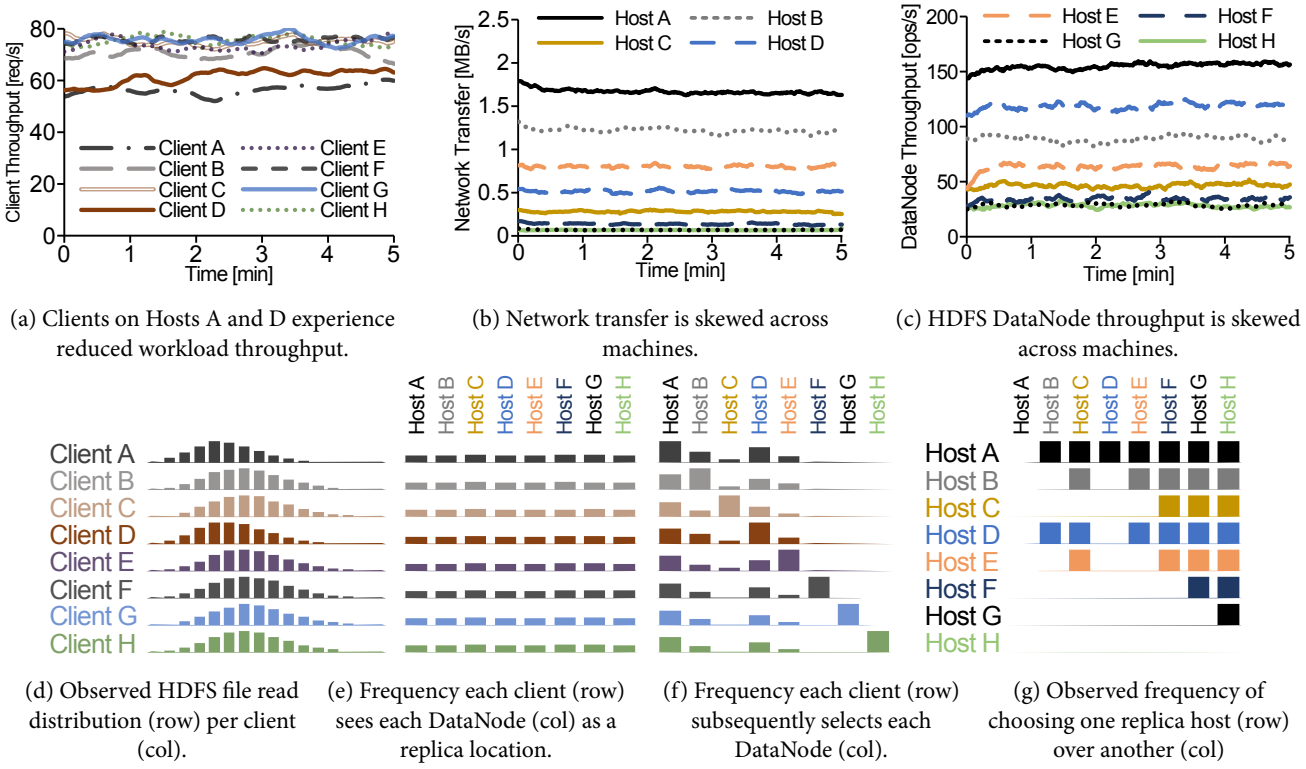
HDFS provides file redundancy by decomposing files into blocks and replicating each block onto several machines (typically 3). A client can read any replica of a block and does so by first contacting the NameNode to find replica hosts (`GetBlockLocations`), then selecting the closest replica as follows: 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 8a, despite identical machine specifications and setup. We first checked machine level resource utilization on each host, which indicated substantial variation in the network throughput (Figure 8b). 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 8c 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:



(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 8: 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 (§6.1).

```
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 8d 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 8e 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 8c clearly show that they do not. To gain more insight into this inconsistency, our next query relates the results from 8e and 8c:

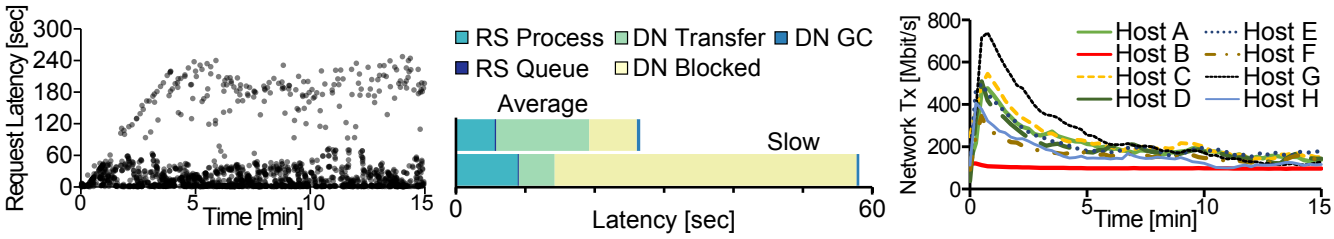
```
Q6: From DNop In DN.DataTransferProtocol
    Join st In StressTest.DoNextOp On st -> DNop
    GroupBy st.host, DNop.host
    Select st.host, DNop.host, COUNT
```

This query measures the frequency that each client selects each DataNode for reading a replica. We plot the results in Figure 8f 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 in-



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

Figure 9: (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.

terference from locally-hosted replicas by *filtering* only the requests that do a non-local read. Figure 8g 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].

6.2 Diagnosing End-to-End Latency

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

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

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

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

A query can measure latency in several components and propagate measurements in the baggage, reminiscent of transaction tracking in Timecard [81] 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 9a). 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 9b). 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 9c).

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

6.3 Overheads of Pivot Tracing

Baggage By default, Pivot Tracing propagates an empty baggage with a serialized size of 0 bytes. In the worst case Pivot Tracing may need to pack an unbounded number of tuples in the baggage, one for each tracepoint invoked. However, the optimizations in §4 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).

The size of serialized baggage is approximately linear in the number of packed tuples. The overhead to pack and unpack tuples from the baggage varies with the size of the baggage – Figure 10 gives micro-benchmark results measuring the overhead of baggage API calls.

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

To measure the effect of Pivot Tracing on CPU bound requests, we stress tested HDFS using requests derived from the HDFS NNBench benchmark: Read8k reads 8kB from a file; Open opens a file for reading; Create creates a file for writing; Rename renames an existing file. Read8kB is a DataNode operation and the others are NameNode operations. We compared the end-to-end latency of requests in unmodified HDFS

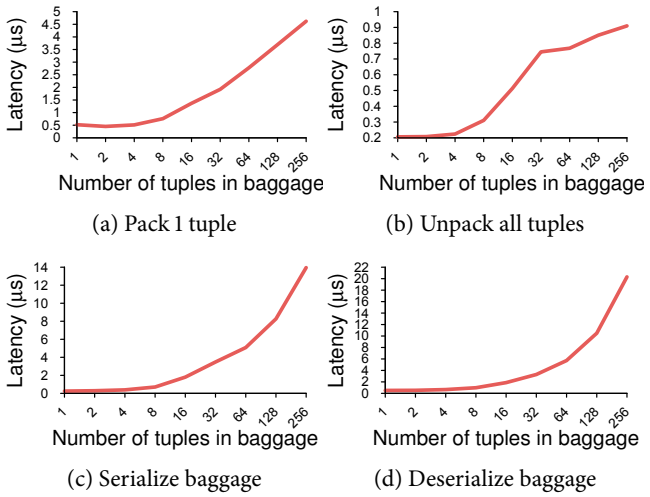


Figure 10: Latency micro-benchmark results for packing, unpacking, serializing, and deserializing randomly-generated 8-byte tuples.

to HDFS modified in the following ways: 1) with Pivot Tracing enabled; 2) propagating baggage containing one tuple but no advice installed; 3) propagating baggage containing 60 tuples (≈1kB) but no advice installed; 4) with the advice from queries in §6.1 installed; 5) with the advice from queries in §6.2 installed.

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

Dynamic Instrumentation 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.

7. Related Work

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

Pivot Tracing's dynamic instrumentation is modeled after aspect-oriented programming [62], and extends prior dynamic instrumentation systems [38, 51, 78] with causal information that crosses process and system boundaries.

	Read8k	Open	Create	Rename
Unmodified	0%	0%	0%	0%
PivotTracing Enabled	0.3%	0.3%	<0.1%	0.2%
Baggage – 1 Tuple	0.8%	0.4%	0.6%	0.8%
Baggage – 60 Tuples	0.82%	15.9%	8.6%	4.1%
Queries – §6.1	1.5%	4.0%	6.0%	0.3%
Queries – §6.2	1.9%	14.3%	8.2%	5.5%

Table 5: Latency overheads for HDFS stress test with Pivot Tracing enabled, baggage propagation enabled, and full queries enabled, as described in §6.3

Temporal Joins Like Fay [51], Pivot Tracing models the events of a distributed system as a stream of dynamically generated tuples belonging to a distributed database. Pivot Tracing's happened-before join is an example of a θ-join [80] where the condition is happened-before. Pivot Tracing's happened-before join is also an example of a special case of path queries in graph databases [94]. Differently from offline queries in a pre-stored graph, Pivot Tracing efficiently evaluates $\overrightarrow{\bowtie}$ at runtime.

Pivot Tracing captures causality between events by generalizing metadata propagation techniques proposed in prior work such as X-Trace [52]. While Baggage enables Pivot Tracing to efficiently evaluate happened-before join, it is not necessary; Magpie [33] demonstrated that under certain circumstances, causality between events can be inferred after the fact. Specifically, if 'start' and 'end' events exist to demarcate a request's execution on a thread, then we can infer causality between the intermediary events. Similarly we can also infer causality across boundaries, provided we can correlate 'send' and 'receive' events on both sides of the boundary (*e.g.*, with a unique identifier present in both events). Under these circumstances, Magpie evaluates queries that explicitly encode all causal boundaries and use temporal joins to extract the intermediary events.

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

Beyond Metrics and Logs A variety of tools have been proposed in the research literature to complement or extend application logs and performance counters. These include the use of machine learning [60, 74, 76, 95] and static analysis [98] to extract better information from logs; automatic enrichment of existing log statements to ease troubleshooting [97]; end-to-end tracing systems to capture the happened-before relationship between events [52, 87]; state-monitoring systems to track system-level resources and indicate the health of a cluster [69]; and aggregation systems to collect and summarize application-level monitoring data [64, 90]. Wang *et al.* provide a comprehensive overview of datacenter troubleshooting tools in [92]. 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, 45, 68, 98] and diagnose performance problems [31, 63, 74, 85]. End-to-end tracing frameworks exist both in academia [33, 39, 52, 83, 88] and in industry [30, 46, 58, 87, 89] and have been used for a variety of purposes, including diagnosing anomalous requests whose structure or timing deviate from the norm [28, 33, 42, 43, 77, 85]; diagnosing steady-state problems that manifest across many requests [52, 83, 85, 87, 88]; identifying slow components and functions [39, 68, 87]; and modelling workloads and resource usage [32, 33, 68, 88]. Recent work has extended these techniques to online profiling and analysis [55, 71–73, 99].

VScope [93] introduces a novel mechanism for honing in on root causes on a running system, but at the last hop defers to offline user analysis of debug-level logs, requiring the user to trawl through 500MB of logs which incur a 99.1% performance overhead to generate. While causal tracing enables coherent sampling [84, 87] which controls overheads, sampling risks missing important information about rare but interesting events.

8. 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 [75].

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

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

A common criticism of systems that require causal propagation of metadata is the need to instrument the original systems [45]. We argue that the benefits outweigh the costs of doing so (§6), 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, 40, 82] can provide language-independent access to baggage variables.

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

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 – a query can span multiple systems written in different programming languages due to Pivot Tracing's platform-independent baggage format and restricted set of advice operations. In particular, it would be an interesting exercise to integrate the happened-before join with Fay or DTrace.

9. Conclusion

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

Acknowledgments

We thank our shepherd Eddie Kohler and the anonymous SOSP reviewers for their invaluable discussions and suggestions. This work was partially supported by NSF award #1452712, as well as by a Google Faculty Research Award.

References

[1] Apache HBase Reference Guide. `http://hbase.apache.org/book.html`. [Online; accessed 25-Feb-2015]. (§2.3).

[2] HADOOP-6599 Split RPC metrics into summary and detailed metrics. `https://issues.apache.org/jira/browse/HADOOP-6599`. [Online; accessed 25-Feb-2015]. (§2.3).

[3] HADOOP-6859 Introduce additional statistics to FileSystem. `https://issues.apache.org/jira/browse/HADOOP-6859`. [Online; accessed 25-Feb-2015]. (§2.3).

[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]. (§2.3).

[5] HBASE-12364 API for query metrics. `https://issues.apache.org/jira/browse/HBASE-12364`. [Online; accessed 25-Feb-2015]. (§2.3).

[6] HBASE-12424 Finer grained logging and metrics for split transaction. `https://issues.apache.org/jira/browse/`

HBASE-12424. [Online; accessed 25-Feb-2015]. (§2.3).

[7] HBASE-12477 Add a flush failed metric. https://issues.apache.org/jira/browse/HBASE-12477. [Online; accessed 25-Feb-2015]. (§2.3).

[8] HBASE-12494 Add metrics for blocked updates and delayed flushes. https://issues.apache.org/jira/browse/HBASE-12494. [Online; accessed 25-Feb-2015]. (§2.3).

[9] HBASE-12496 A blockedRequestsCount metric. https://issues.apache.org/jira/browse/HBASE-12496. [Online; accessed 25-Feb-2015]. (§2.3).

[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]. (§2.3).

[11] HBASE-2257 [stargate] multiuser mode. https://issues.apache.org/jira/browse/HBASE-2257. [Online; accessed 25-Feb-2015]. (§2.3).

[12] HBASE-4038 Hot Region : Write Diagnosis. https://issues.apache.org/jira/browse/HBASE-4038. [Online; accessed 25-Feb-2015]. (§2.3).

[13] HBASE-4145 Provide metrics for hbase client. https://issues.apache.org/jira/browse/HBASE-4145. [Online; accessed 25-Feb-2015]. (§2.3).

[14] HBASE-4169 Add per-disk latency metrics to DataNode. https://issues.apache.org/jira/browse/HDFS-4169. [Online; accessed 25-Feb-2015]. (§2.3).

[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]. (§2.3).

[17] HBASE-6093 Expose more caching information for debugging by users. https://issues.apache.org/jira/browse/HDFS-6093. [Online; accessed 25-Feb-2015]. (§2.3).

[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]. (§2.3).

[19] HBASE-7390 Provide JMX metrics per storage type. https://issues.apache.org/jira/browse/HDFS-7390. [Online; accessed 25-Feb-2015]. (§2.3).

[20] HBASE-7958 Statistics per-column family per-region. https://issues.apache.org/jira/browse/HBASE-7958. [Online; accessed 25-Feb-2015]. (§2.3).

[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]. (§1 and 2.3).

[22] HBASE-8868 add metric to report client shortcircuit reads. https://issues.apache.org/jira/browse/HBASE-8868. [Online; accessed 25-Feb-2015]. (§2.3).

[23] HBASE-9722 need documentation to configure HBase to reduce metrics. https://issues.apache.org/jira/browse/HBASE-9722. [Online; accessed 25-Feb-2015]. (§2.3).

[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]. (§6.1, 8, and 6.1).

[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]. (§2.3).

[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]. (§2.3).

[27] Apache accumulo. http://accumulo.apache.org/. [Online; accessed March 2015]. (§2.3).

[28] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *SOSP* (New York, NY, USA, 2003), ACM Press. (§7).

[29] Almeida, P. S., Baquero, C., and Fonte, V. Interval tree clocks: A logical clock for dynamic systems. In *OPODIS* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 259–274. (§5).

[30] Appneta traceview. http://appneta.com. [Online; accessed July 2013]. (§7).

[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. (§7).

[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. (§4 and 7).

[33] Barham, P., Isaacs, R., Mortier, R., and Narayanan, D. Magpie: Online modelling and performance-aware systems. In *HotOS* (2003), vol. 9. (§7).

[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* (Hyderabad, India, June 4–6, 2014), pp. 468–479. (§7).

[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. (§2.3).

[36] Buch, I., and Park, R. Improve debugging and performance tuning with ETW. *MSDN Magazine* (2007). [Online; accessed 01-01-2012]. (§8).

[37] Cantrill, B. Hidden in plain sight. *ACM Queue 4*, 1 (Feb. 2006), 26–36. (§1 and 2.3).

[38] Cantrill, B., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *USENIX ATC* (2004), pp. 15–28. (§1, 2.3, 5, 7, and 8).

[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. (§1, 6.2, and 7).

[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 *Middleware* (November 2005), pp. 42–59. (§8).

[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. (§6).

[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). (§7).

[43] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN* (Washington, DC, USA, 2002), DSN '02, IEEE Computer Society, pp. 595–604. (§7).

[44] Chiba, S. Javassist: Java bytecode engineering made simple. *Java Developer's Journal 9*, 1 (2004). (§5).

[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 *OSDI* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 217–231. (§7 and 8).

[46] Compuware dynatrace purepath. `http://www.compuware.com`. [Online; accessed July 2013]. (§7).

[47] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *SOCC* (2010), ACM, pp. 143–154. (§6.3).

[48] Couckuyt, J., Davies, P., and Cahill, J. Multiple chart user interface, June 14 2005. US Patent 6,906,717. (§1).

[49] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113. (§6).

[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 *SOCC* (2013), ACM, p. 14. (§6.2).

[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. (§1, 2.3, 4, 5, 7, and 8).

[52] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *NSDI* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association. (§1, 4, 4, and 7).

[53] Google Protocol Buffers. `http://code.google.com/p/protobuf/`. (§5).

[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. (§1).

[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 ATC* (2011). (§7).

[56] Apache HBase. `http://hbase.apache.org`. [Online; accessed March 2015]. (§6).

[57] The Java HotSpot Performance Engine Architecture. `http://www.oracle.com/technetwork/java/whitepaper-135217.html`. [Online; accessed March 2015]. (§6.3).

[58] Apache HTrace. `http://htrace.incubator.apache.org/`. [Online; accessed March 2015]. (§2.3 and 7).

[59] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDEW* (2010), IEEE, pp. 41–51. (§6.3).

[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). (§1 and 7).

[61] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An Overview of AspectJ. In *ECOOP* (London, UK, UK, 2001), ECOOP '01, Springer-Verlag, pp. 327–353. (§5 and 6).

[62] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In *ECOOP* (June 1997), LNCS 1241, Springer-Verlag. (§2.2 and 7).

[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. (§7).

[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. (§7).

[65] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565. (§1 and 3).

[66] Laub, B., Wang, C., Schwan, K., and Huneycutt, C. Towards combining online & offline management for big data applications. In *ICAC* (Philadelphia, PA, June 2014), USENIX Association, pp. 121–127. (§6 and 6.2).

[67] Mace, J., Bodik, P., Musuvathi, M., and Fonseca, R. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI* (May 2015), USENIX Association. (§6.2).

[68] Mann, G., Sandler, M., Krushevskaja, D., Guha, S., and Even-Dar, E. Modeling the parallel execution of black-box services. *USENIX/HotCloud* (2011). (§7).

[69] 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. (§2.3 and 7).

[70] Meijer, E., Beckman, B., and Bierman, G. Linq: Reconciling object, relations and xml in the .net framework. In *SIGMOD* (New York, NY, USA, 2006), SIGMOD '06, ACM, pp. 706–706. (§2.1).

[71] Mi, H., Wang, H., Chen, Z., and Zhou, Y. Automatic detecting performance bugs in cloud computing systems via learning latency specification model. In *SOSE* (2014), IEEE, pp. 302–307. (§7).

[72] 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.

[73] 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. (§1 and 7).

[74] Nagaraj, K., Killian, C. E., and Neville, J. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI* (2012), pp. 353–366. (§1 and 7).

[75] Oliner, A., Ganapathi, A., and Xu, W. Advances and challenges in log analysis. *Communications of the ACM 55*, 2 (2012), 55–61. (§2.3 and 8).

[76] Oliner, A., Kulkarni, A., and Aiken, A. Using correlated surprise to infer shared influence. In *IEEE/IFIP Dependable Systems and Networks (DSN)* (June 2010), pp. 191–200. (§1 and 7).

[77] Ostrowski, K., Mann, G., and Sandler, M. Diagnosing latency in multi-tier black-box services. In *LADIS* (2011). (§7).

[78] 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). (§2.3 and 7).

[79] Rabkin, A., and Katz, R. H. How hadoop clusters break. *Software, IEEE 30*, 4 (2013), 88–94. (§2.3).

[80] Ramakrishnan, R., and Gehrke, J. *Database Management Systems*, 2nd ed. Osborne/McGraw-Hill, Berkeley, CA, USA, 2000. (§4 and 7).

[81] Ravindranath, L., Padhye, J., Mahajan, R., and Balakrishnan, H. Timecard: Controlling user-perceived delays in server-based mobile applications. In *SOSP* (2013), ACM, pp. 85–100. (§6.2).

[82] Reumann, J., and Shin, K. G. Stateful distributed interposition. *ACM Trans. Comput. Syst. 22*, 1 (2004), 1–48. (§8).

[83] 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* (Berkeley, CA, USA, 2006), USENIX Association. (§7).

[84] 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. (§7).

[85] 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). (§7).

[86] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop distributed file system. In *MSST* (2010), IEEE, pp. 1–10. (§6).

[87] 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). (§1, 2.3, 4, 4, and 7).

[88] 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. (§4 and 7).

[89] Twitter Zipkin. `http://twitter.github.io/zipkin/`. [Online; accessed March 2015]. (§2.3 and 7).

[90] 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. (§7).

[91] 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 *SOCC* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16. (§6).

[92] 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. (§7).

[93] 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. (§6, 6.2, and 7).

[94] Wood, P. T. Query languages for graph databases. *SIGMOD Rec. 41*, 1 (Apr. 2012), 50–60. (§7).

[95] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (New York, NY, USA, 2009), ACM, pp. 117–132. (§1 and 7).

[96] 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 *SOSP* (2011), ACM, pp. 159–172. (§2.3).

[97] 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). (§1, 2.3, and 7).

[98] 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 *OSDI* (2014). (§1 and 7).

[99] Zhou, J., Chen, Z., Mi, H., and Wang, J. Mtracer: a trace-oriented monitoring framework for medium-scale distributed systems. In *SOSE* (2014), IEEE, pp. 266–271. (§7).