Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems

By Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca

Abstract
Monitoring and troubleshooting distributed systems are 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. Pivot Tracing does not correlate cross-component events using expensive global aggregations, nor does it perform offline analysis. Instead, Pivot Tracing directly correlates events as they happen by piggybacking metadata alongside requests as they execute. This gives Pivot Tracing low runtime overhead—less than 1% for many cross-component monitoring queries.

1. INTRODUCTION
Monitoring and troubleshooting distributed systems are hard. The potential problems are myriad: hardware and software failures, misconfigurations, hot spots, aggressive tenants, or even simply unrealistic user expectations. Despite the complex and unpredictable nature of these problems, most of the 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 and static analysis to improve the quality of logs and their use in troubleshooting, logs carry an inherent tradeoff between recall and overhead, as what gets logged must be defined a priori.

Addressing this limitation, dynamic instrumentation systems such as Fay2 and DTrace4 enable the diagnosis of unanticipated performance problems in production systems1 by providing the ability to select, at runtime, which of a large number of tracepoints to activate. Dynamic instrumentation, however, is 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 present Pivot Tracing, a monitoring framework that combines dynamic instrumentation with causal tracing techniques8,23 to fundamentally increase the power and applicability of either technique. Pivot Tracing gives operators and users, at runtime, the ability to obtain an almost arbitrary metric at one point of the system, while selecting, filtering, and grouping by causally preceding events from other parts of the system, even when crossing component or machine boundaries. Pivot Tracing exposes these features by modeling system events as the tuples of a streaming, distributed data set. Users can write relational queries about system events using Pivot Tracing's LINQ-like query language. Pivot Tracing compiles queries into efficient instrumentation code and dynamically installs the code at the sources of events specified in the query, returning a streaming data set of results to the user.

The key contribution of Pivot Tracing is the “happened-before join” operator, →, that enables queries to be contextualized by Lamport’s happened-before relation, →.15 Using →, 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.

We have implemented and open-sourced a prototype of Pivot Tracing for Java-based systems, and instrumented a variety of distributed systems including HDFS, HBase, MapReduce, Tez, YARN, and Spark. In our full evaluation,16 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.

The original version of this paper was published in Proceedings of the 25th Symposium on Operating Systems Principles (2015), ACM. New York, NY, 378–393.
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 and implementation to Sections 3 and 4, respectively.

Suppose we are managing a cluster of eight machines and want to know how disk bandwidth is being used across the cluster. On these machines, we are simultaneously running clients with workloads in HBase, HDFS, and MapReduce. It suffices to know that HBase is a distributed database that 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. Figure 1 depicts this scenario along with the following client applications:

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

```
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 2a.

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 2b 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. PivotTracing’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 2c demonstrates the ability for Pivot Tracing to group metrics along arbitrary dimensions. It is generated...
by two queries similar to Q2 that instrument Java’s FileInput-
Stream and FileOutputStream, still joining with the client pro-
cess 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 pro-
cesses to the data access points. Pivot Tracing’s query opti-
mizer installs dynamic instrumentation where needed, and
determines when such propagation must occur to pro-
cess a query. The out-of-the-box metrics provided by HDFS,
HBase, and MapReduce cannot provide analyses like those pre-

duced here. Simple correlations—such as determining which
HDFS datanodes were read from by a high-level client applica-
tion—are not typically possible. Metrics are ad hoc between
systems; HDFS sums IO bytes, while HBase exposes opera-
tions 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 3 presents a high-level overview of how Pivot Tracing
enables queries such as Q2. We refer to the numbers in the fig-
ure (e.g., Q) in our description. Full support for Pivot Tracing
in a system requires two basic mechanisms: dynamic code
injection and causal metadata propagation.

Queries in Pivot Tracing refer to variables exposed by one
or more tracepoints—places in the system where Pivot
Tracing can insert instrumentation. Tracepoint defini-
tions are not part of the system code, but are rather instruc-
tions 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, and can refer
to arbitrary interface/method signature combinations.
Tracepoints are defined by someone with knowledge of the
system, maybe a developer or expert operator, and define the
vocabulary for queries (Q). 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 (Q), which get compiled to an intermediate represen-
tation called advice (Q). Advice uses a small instruction set
to process queries, and maps directly to code that local Pivot
Tracing agents install dynamically at relevant tracepoints (Q).
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 support-
ing joins between events that occur within and across pro-
cess, machine, and application boundaries. The efficient
implementation of the happened before join requires advice
in one tracepoint to send information along the execu-
tion path to advice in subsequent tracepoints. This is done
through a new baggage abstraction, which uses causal meta-
data propagation (Q). In query Q2, for example, cl.procName
is packed in the first invocation of the ClientProtocols tra-
cepoint, to be accessed when processing the incrBytesRead
tracepoint.

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

2.3. Monitoring and troubleshooting challenges

Pivot Tracing addresses two main challenges in monitor-
ing and troubleshooting. First, when the choice of what to
record about an execution is made a priori, there is an inher-
ent tradeoff between recall and overhead. Second, to diag-
nose many important problems one needs to correlate and
integrate data that crosses component, system, and machine
boundaries.

One size does not fit all. Problems in distributed systems
are complex, varied, and unpredictable. By default, the infor-
mation 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 discussed the
important need to “close the loop of operations back to de-
velopers.” According to Yuan et al., regarding diagnosing
failures, “(...) existing log messages contain too little infor-
mation. 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 requesting new met-
rics, changes to aggregation methods, or new breakdowns
of existing metrics. Many issues remain unresolved due to
developer pushback or inertia.

Eventually, applications may be updated to record more
information, but this has effects both in performance and
information overload. Users must pay the performance over-
heads of any systems that are enabled by default, regard-
less of their utility. For example, HBase SchemaMetrics
were introduced to aid developers, but all users of HBase
pay the 10% performance overhead they incur. “The HBase
user guide carries the following warning for users wishing
to integrate with Ganglia: “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 HBases.\footnote{The glut of recorded information presents a “needle-in-a-haystack” problem to users\textsuperscript{11}; while a system may expose information relevant to a problem, for example, 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.\textsuperscript{16}}

The glut of recorded information presents a “needle-in-a-haystack” problem to users\textsuperscript{11}; while a system may expose information relevant to a problem, for example, 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.\textsuperscript{16} Dynamic instrumentation frameworks such as Fay,\textsuperscript{7} DTrace,\textsuperscript{4} and SystemTap\textsuperscript{20} 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.\textsuperscript{3} 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\textsuperscript{9} 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\textsuperscript{16} 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 an 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 and studies of logging practices conclude that end-to-end tracing would be helpful in navigating the logging issues they outline.\textsuperscript{16} A variety of these mechanisms have also materialized in production systems, for example, Google’s Dapper,\textsuperscript{23} Apache’s HTrace,\textsuperscript{4} and Twitter’s Zipkin.\textsuperscript{24} These approaches can obtain richer information about particular executions than component-centric logs or metrics alone, and have found uses in troubleshooting, debugging, performance analysis and anomaly detection, for example. However, most of these systems record or reconstruct traces of execution for offline analysis, and thus share the problems above with the first challenge, concerning what to record.

### 3. Design

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

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

**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, such as the name of a method. Since Pivot Tracing uses dynamic instrumentation to install queries, tracepoints do not need to be defined a priori, nor do they require a priori modification of system code; they are simply references to locations in the source code. A tracepoint is only materialized once a query is installed that references it. Tracepoints export named variables that can be accessed by instrumentation, such as method arguments or local variables, as well as several default variables: 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 Section 2. Table 1 outlines the query operations supported by Pivot Tracing.

<table>
<thead>
<tr>
<th>Query Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select (σ)</td>
<td>Selects tuples that satisfy a predicate</td>
</tr>
<tr>
<td>Project (Π)</td>
<td>Extracts a subset of the variables</td>
</tr>
<tr>
<td>Group (G)</td>
<td>Groups tuples by a key</td>
</tr>
<tr>
<td>Aggregation (A)</td>
<td>Performs an aggregation on the grouped tuples</td>
</tr>
</tbody>
</table>

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 (\(\rightarrow\)).

**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, \(\rightarrow\).\footnote{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.}

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.\footnote{If a} 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 \(a\) causal preceded the occurrence of event \(b\) and they occurred as part of the execution of the same request.\footnote{If a}
and $b$ are not part of the same execution, then $a \neq b$ if the occurrence of $a$ did not lead to the occurrence of $b$, then $a \neq b$ (e.g., they occur in two parallel threads of execution that do not communicate); and if $a \rightarrow b$ then $b \neq a$.

For any two queries $Q_1$ and $Q_2$, the happened-before join $Q_1 \triangleright Q_2$ produces tuples $t_i \triangleright t_j$ for all $t_i \in Q_1$ and $t_j \in Q_2$ such that $t_i \rightarrow t_j$. That is, if $Q_1$ produces $t_i$ before $Q_2$ produces tuple $t_j$ in the execution of the same figure. 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 $Q_2$ in Section 2 demonstrates the use of happened-before join. In the query, tuples generated by the disk IO tracepoint $DataNodeMetrics.incrBytesRead$ are joined to the first tuple generated by the $ClientProtocols$ tracepoint.

Happened-before join substantially improves our ability to perform root cause analysis by giving us visibility into the relationships between events in the system. The happened-before relationship is fundamental to a number of prior approaches in root cause analysis. \(^{16}\) Pivot Tracing is designed to efficiently support happened-before joins, but does not optimize more general joins such as equijoins ($\pi$).

**Advice.** Pivot Tracing queries compile to an intermediate representation called advice. Advice specifies the operations to perform at each tracepoint used in a query, and eventually materializes as monitoring code installed at those tracepoints (Section 4). Advice has several operations for manipulating tuples through the tracepoint-exported variables, and evaluating $\pi$ on tuples produced by other advice at prior tracepoints in the execution.

Table 2 outlines the advice API. OBSERVE creates a tuple from exported tracepoint variables. UNPACK retrieves tuples generated by other advice at other tracepoints prior in the execution. Unpacked tuples can be joined to the observed tuple, that is, if $t_i$ is observed and $t_k$ and $t_j$ are unpacked, then the resulting tuples are $t_i \triangleright t_k$ and $t_i \triangleright t_j$. Tuples created by this advice can be discarded (FILTER), made available to advise 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 $Q_2$ in Section 2 compiles to advice $A1$ and $A2$ for $ClientProtocols$ and $DataNodeMetrics$, respectively:

$A1 : OBSERVE \ procName \ A2 : OBSERVE \ delta$

PACK-FIRST $\ procName \ A2 : UNPACK \ procName \ A1 : EMIT \ procName, SUM(delta)$

Figure 5 shows how this advice and the tracepoints interact with the execution of requests in the system. First, when a request’s execution reaches $ClientProtocols$, $A1$ is invoked, which observes and packs a single valued tuple containing the process name. Then, when execution reaches $DataNodeMetrics$, $A2$ is invoked, which unpacks the process name, observes the value of $delta$, then emits a joined tuple.

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 generated for Q2 from Section 2: A1 observes and packs procName; A2 unpacks procName, observes delta, and emits (procName, SUM(delta)).

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.

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 and Dapper. Using baggage, Pivot Tracing efficiently evaluates happened-before joins in situ during the execution of a request.

Tuple aggregation and query optimization. To reduce the volume of emitted tuples, 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, for example, once per second. Process-level aggregation substantially reduces traffic for emitted tuples; Q2 from Section 2 is reduced from approximately 600 to 6 tuples per second from each DataNode. Pivot Tracing also rewrites queries to minimize the number of tuples that are packed during a request’s execution, using the same query rewriting rules described by Fay that push projection, selection, and aggregation terms as close as possible to source tracepoints. We extend these query rewriting rules to add further optimizations for happened-before joins.

4. IMPLEMENTATION

We have implemented a Pivot Tracing prototype in Java and applied Pivot Tracing to several open-source systems from the Hadoop ecosystem. Pivot Tracing source code and the instrumented systems are publicly available from the Pivot Tracing project website.

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 back to the user at a configurable interval—by default, 1 s.

Dynamic instrumentation. Our prototype weaves advice at runtime, providing dynamic instrumentation similar to that of DTrace and Fay. Java version 1.5 onwards supports dynamic method body rewriting via the java.lang.instrument package. The Pivot Tracing agent pro-grammatically rewrites and reloads class bytecode from within the process using Javassist. To weave advice, we rewrite method bodies to add advice invocations at the locations defined by the tracepoint. 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. Pivot Tracing supports instrumenting privileged classes (e.g., FileInputStream in Section 2) by providing an optional agent that can be placed on Java’s boot classpath.

Pivot Tracing only makes system modifications when advice is woven into a tracepoint, so inactive tracepoints incur no overhead. Executions that do not trigger the tracepoint are unaffected 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. Our implementation of baggage uses thread-local variables for storing per-request 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. The baggage API 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.

Hadoop instrumentation. Pivot Tracing relies on developers to implement Baggage propagation when a request crosses thread, process, or asynchronous execution boundaries. We have implemented this propagation in several open-source systems that are widely used in production today: HDFS, HBase, MapReduce, Tez, YARN, and Spark. 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 instrumentation to automatically modify common interfaces (Thread, Runnable, Callable, and Queue). Each system required between 50 and 200 lines of manual code modification. Once modified, these systems could support arbitrary Pivot Tracing queries without further modification.

5. EVALUATION

In this section, we evaluate Pivot Tracing with a case study in the context of the Hadoop Distributed File System (HDFS). HDFS is a distributed file system comprising a central NameNode process that manages filesystem metadata, and
multiple DataNode processes running across a cluster that store replicated file blocks. 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.11

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 (invoking GetBlockLocations), then selecting the closest replica as follows: (1) read a local replica, (2) read a rack-local replica, and (3) select a replica at random. We discovered a bug whereby rack-local replica selection always follows a global static ordering due to two conflicting behaviors: the HDFS client does not randomly select between replicas; and the HDFS NameNode does not randomize rack-local replicas returned to the client. The bug results in heavy load on some hosts and near zero load on others.

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

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 6a, 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 6b). 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(n)

Figure 6c 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/s, while host H has only 25 ops/s. This behavior was unexpected given that our stress test clients are supposed 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(n)

Figure 6a, despite identical machine specifications and setup.

---

**Figure 6. Pivot Tracing query results leading to our discovery of HDFS-6268.** Faulty replica selection logic led clients to prioritize the replicas hosted by particular DataNodes: host A was always preferred over other hosts if it held a replica; host D was always preferred, except if host A held a replica; etc. The increased load to host A DataNode reduced the throughput of co-located client A. (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).
This query counts the number of times each client reads each file. In Figure 6d, we plot the distribution of counts over a 5-min 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 6e shows that, for each client, replicas are near-uniformly distributed across DataNodes in the cluster. These results indicate that clients have an equal opportunity to read replicas from each DataNode, yet, our measurements in Figure 6c clearly show that they do not. To gain more insight into this inconsistency, our next query relates the results from Figure 6e to those from Figure 6c:

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 6f 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 6g 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. This should not have been the case; due to random replica selection, no host should have been preferred over any other host.

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.11

Application-level overhead. To estimate the impact of Pivot Tracing on application-level throughput and latency, we ran benchmarks from HiBench,12 YCSB,6 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: Read8kB reads 8kB from a file; Open opens a file for reading; Create creates a file for writing; Rename renames an existing file. Read8KB is a DataNode operation and the others are NameNode operations. We compared the end-to-end latency of requests in unmodified HDFS to HDFS modified in the following ways: (1) with Pivot Tracing enabled, (2) propagating baggage containing one tuple but no advice installed, (3) propagating baggage containing 60 tuples (≈1kB) but no advice installed, and (4) with queries Q3–Q7 installed.

Table 3 shows that the application-level overhead with PivotTracing enabled is at most 0.3%. This overhead includes the costs of empty 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 queries Q3–Q7, reflected by an overhead of just 0.3%.

6. DISCUSSION

Despite the advantages over logs and metrics for troubleshooting (Section 2), Pivot Tracing is not meant to replace all functions of logs, such as security auditing, forensics, or debugging.19

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 PivotTracing 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.

<table>
<thead>
<tr>
<th>Query</th>
<th>Read8k (%)</th>
<th>Open (%)</th>
<th>Create (%)</th>
<th>Rename (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmodified</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PivotTracing Enabled</td>
<td>0.3</td>
<td>0.3</td>
<td>&lt;0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>Baggage–1 Tuple</td>
<td>0.8</td>
<td>0.4</td>
<td>0.6</td>
<td>0.8</td>
</tr>
<tr>
<td>Baggage–60 Tuples</td>
<td>0.82</td>
<td>15.9</td>
<td>8.6</td>
<td>4.3</td>
</tr>
<tr>
<td>Queries Q3–Q7</td>
<td>1.5</td>
<td>4.0</td>
<td>6.0</td>
<td>0.3</td>
</tr>
</tbody>
</table>
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 that 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.

7. 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 interoperating 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

2. 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 Oper. Syst. Rev. 45, 3 (2011), 20–22. (Section 2.3).
3. Cantrill, B. Hidden in plain sight. ACM Queue 4, 1 (Feb. 2006), 26–36. (Sections 1 and 2.3).
18. 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 February 2015]. (Section 2.3).