Pivot Tracing
Dynamic Causal Monitoring for Distributed Systems

JONATHAN MACE, RYAN ROELKE, AND RODRIGO FONSECA

Jonathan Mace is a PhD student in computer science at Brown University, advised by Rodrigo Fonseca. His research interests include end-to-end tracing, runtime debugging, and resource management in distributed systems. jcmace@cs.brown.edu

Ryan Roelke received a master's degree in computer science from Brown University in 2015 and is currently a Software Engineer at HP Vertica. rroelke@cs.brown.edu

Rodrigo Fonseca is an Assistant Professor at Brown University's Computer Science Department. He holds a PhD from UC Berkeley, and prior to Brown was a visiting researcher at Yahoo! Research. He is broadly interested in networking, distributed systems, and operating systems. His research involves seeking better ways to build, operate, and diagnose distributed systems, including large-scale Internet systems, cloud computing, and mobile computing. He is currently working on dynamic tracing infrastructures for these systems, on new ways to leverage network programmability, and on better ways to manage energy usage in mobile devices. rfonseca@cs.brown.edu

Pivot Tracing is a monitoring framework for distributed systems that can seamlessly correlate statistics across applications, components, and machines at runtime without needing to change or redeploy system code. Users can define and install monitoring queries on-the-fly to collect arbitrary statistics from one point in the system while being able to select, filter, and group by events meaningful at other points in the system. 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—even across component and machine boundaries. This gives Pivot Tracing a very low runtime overhead—less than 1% for many cross-component monitoring queries.

Monitoring and Troubleshooting Distributed Systems

Problems in distributed systems are many and varied: component failures due to hardware errors, software bugs, and misconfiguration; unexpected overload behavior due to hot spots and aggressive tenants; or simply unrealistic user expectations. Due to designs such as fault-tolerance and load balancing, the root cause of an issue may not be immediately apparent from its symptoms. However, while troubleshooting distributed systems is inherently challenging, many of the monitoring and diagnosis tools used today share two fundamental limitations that further exacerbate the challenge.

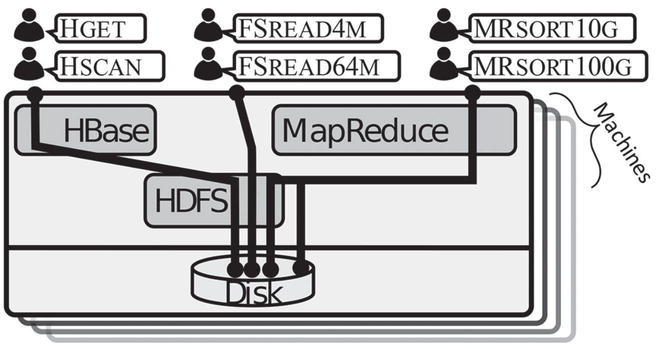
One Size Does Not Fit All

First, many tools only record information that is selected a priori at development or deployment time. Even though there has been great progress in using machine-learning techniques and static analysis to improve the quality of logs, they still carry an inherent tradeoff between recall and overhead. The choice of what to record must be made a priori, so inevitably the information needed to diagnose an issue might not be reported by the system. Even if a relevant event is captured in a log message, it can still contain too little information; similarly, performance counters may be too coarse grained or lack the desired filters or groupings.

On the other hand, if a system does expose information relevant to a problem, it is often buried under a mountain of other irrelevant information, presenting a "needle in a haystack" problem to users. Any time a user or developer patches a system to add more instrumentation, they contribute to this information overload. They also potentially add performance overheads for any monitoring that is enabled by default. Unsurprisingly, developers are resistant to adding additional metrics or groupings, as can be observed in a plethora of unresolved and rejected issues on Apache's issue trackers.

Crossing Boundaries

Second, many tools record information in a component- or machine-centric way, making it difficult to correlate events across these boundaries. Since today's datacenters typically host a wide variety of interoperating components and systems, the root cause and symptoms of an



HGET	10 kB row lookups in a large HBase table
HSCAN	4 MB table scans of a large HBase table
FSREAD4M	Random closed-loop 4 MB HDFS reads
FSREAD64M	Random closed-loop 64 MB HDFS reads
MRSORT10G	MapReduce sort job on 10 GB of input data
MRSORT100G	MapReduce sort job on 100 GB of input data

Figure 1: Six client workloads access the disks on eight cluster machines indirectly via HBase, a distributed database; HDFS, a distributed file system; and MapReduce, a data processing framework.

issue often appear in different processes, machines, and application tiers. A user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. To do this manually is cumbersome, and in many cases impossible, because it depends on sufficient execution context having been propagated across software component and machine boundaries.

Dynamic Instrumentation and Causal Tracing

Pivot Tracing overcomes these challenges by combining two key techniques: dynamic instrumentation and causal tracing. Dynamic instrumentation systems, such as DTrace [1], Fay [2], and SystemTap [6], let users defer until runtime their selection of information reported by the system. They allow almost arbitrary instrumentation to be added dynamically at runtime as needed, and have proven extremely useful in diagnosing complex and unanticipated system problems. Pivot Tracing also uses dynamic instrumentation, enabling users to specify new monitoring queries at runtime. Pivot Tracing queries are dynamically installed without the need to change or redeploy code.

Dynamic instrumentation alone does not address the challenge of correlating events from multiple components. To address this challenge, Pivot Tracing adapts techniques presented in the causal tracing literature by systems such as X-Trace [3] and Dapper [7]. These systems maintain a notion of context that follows an execution through events, queues, thread pools, files, caches, and messages between distributed system components. Likewise, Pivot Tracing propagates a tracing context alongside requests. Unlike end-to-end tracing, Pivot Tracing does not record or reconstruct traces of executions for offline analysis. Instead, its tracing context is a means for propagating a small amount of state directly along the execution path of requests, including when they cross component and machine boundaries.

Pivot Tracing

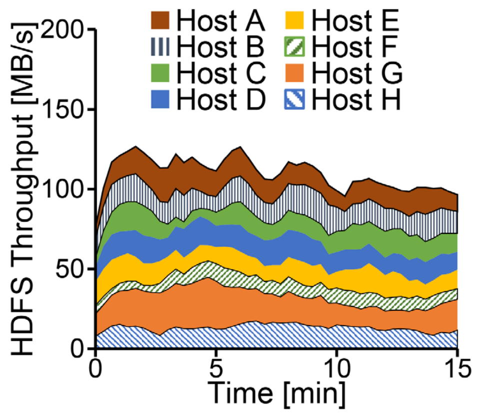
Pivot Tracing exposes these two features by modeling system events as the tuples of a streaming, distributed data set. Users can write relational queries about system events using Pivot Tracing's LINQ-like query language. Pivot Tracing compiles queries into instrumentation code and dynamically installs the code at the sources of events specified in the query. Each time one of the events occurs, the instrumentation code is also invoked.

Happened-Before Join

In order to reason about causality between events, Pivot Tracing introduces a new relational operator, the "happened-before join," $\overrightarrow{\bowtie}$, for joining tuples based on Lamport's happened-before relation [4]. 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. Using the happened-before join, users can write queries that group and filter events based on properties of events that causally precede them in an execution. Pivot Tracing evaluates the happened-before join by putting partial query state into the tracing contexts propagated alongside requests. This is an efficient way to evaluate the happened-before join, because it explicitly follows the happened-before relation. It drastically mitigates the overhead and scalability issues that would otherwise be required for correlating events globally.

Pivot Tracing in Action

To motivate Pivot Tracing's design and implementation, we present a brief example of Pivot Tracing with a monitoring task in the Hadoop stack. 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 several clients with workloads in HBase, MapReduce,



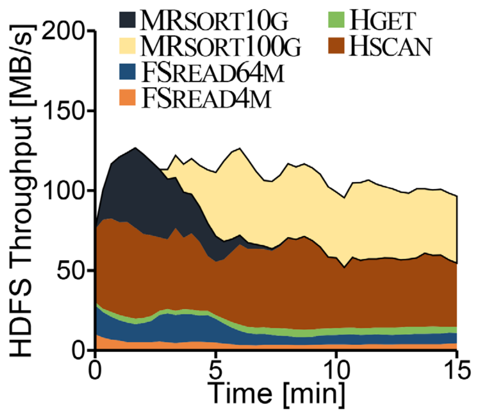


Figure 2: In this example, Pivot Tracing dynamically instruments HDFS to expose read throughput grouped by client identifiers from other applications.

and HDFS. 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.

By default, our distributed file system HDFS already tracks some disk consumption metrics, including disk read throughput aggregated on each of its DataNodes. To reproduce this metric with Pivot Tracing, we can define a *tracepoint* for the method `incrBytesRead(int delta)` in the `DataNodeMetrics` class in HDFS. A tracepoint is a location in the application source code where instrumentation can run. 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 if we wish to measure the HDFS usage of each of our client applications. HDFS only has visibility of its direct clients, and thus it only has 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. 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.

Design and Implementation

We opted to implement our Pivot Tracing prototype in Java in order to easily instrument the aforementioned open source distributed systems. However, the components of Pivot Tracing generalize and are not restricted to Java—a query can even span multiple systems written in different programming languages. Full support for Pivot Tracing in a system requires two basic mechanisms: dynamic code injection and causal metadata propagation. For full details of Pivot Tracing's design and implementation, we refer the reader to the full paper [5] and project Web site, http://pivottracing.io/.

Figure 3 presents a high-level overview of how Pivot Tracing enables queries such as Q2. We will refer to the numbers in the figure (e.g., ①) in our description.

Writing Queries

Queries in Pivot Tracing refer to variables exposed by one or more tracepoints (①)—places in the system where Pivot Tracing can insert instrumentation. Tracepoints export named variables that can be accessed by instrumentation. However, the definitions of tracepoints are not part of the system code but, rather, instructions on where and how Pivot Tracing can add instrumentation. Tracepoints in Pivot Tracing are similar to pointcuts from aspect-oriented programming and can refer to arbitrary interface/method signature combinations. Pivot Tracing's LINQ-like query language supports several typical operations including projection, selection, grouping, aggregation, and happened-before join.

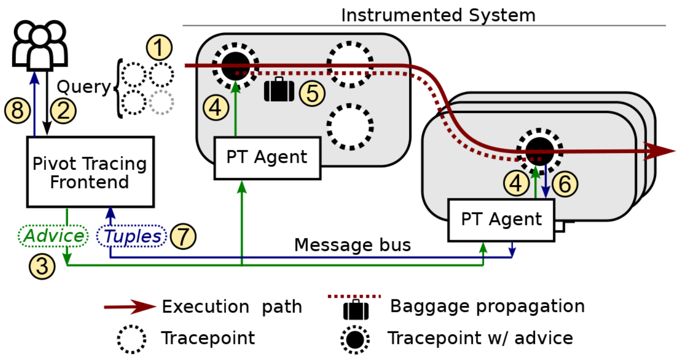


Figure 3: Pivot Tracing overview

Compiling Queries

Users submit queries to the Pivot Tracing front-end (②), which is responsible for optimizing queries using some simple static rewriting rules, pushing projection, selection, and aggregation as close as possible to the source tracepoints. The front-end then compiles queries into advice, an intermediate representation of the system-level instrumentation needed to evaluate the query. Advice specifies the operations to perform at each tracepoint used in a query.

Installing Queries

The Pivot Tracing front-end distributes advice to local Pivot Tracing agents running in each process (③). Pivot Tracing agents are responsible for dynamically instrumenting the running system so that advice is invoked at tracepoints. The agents *weave* advice into tracepoints (④) by: (1) generating 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. Later, requests executing in the system will invoke the installed advice every time their execution reaches the tracepoint.

Crossing Boundaries

In order to implement the happened-before join, advice invoked at one tracepoint needs to make information available to advice invoked at other tracepoints later in a request's execution. For example, in Q2, advice at the `ClientProtocols` tracepoint needs to make its `procName` available to later advice invoked at the `DataNodeMetrics` tracepoint. This is done through Pivot Tracing's *baggage* abstraction, which uses causal metadata propagation (⑤). Baggage is a per-request container for tuples that is propagated alongside a request as it traverses thread, application, and machine boundaries. At any point in time, advice can put tuples in the baggage of the current request, and retrieve tuples that were previously placed in the baggage by other advice.

Evaluating Queries

Advice uses a small instruction set to evaluate queries and maps directly to the code that local Pivot Tracing agents generate. Advice operations are as follows: advice can create a tuple from tracepoint-exported variables (Observe); filter tuples by a predicate (Filter); and output tuples for global aggregation (Emit). Advice can put tuples in the baggage (Pack) and retrieve tuples from the baggage (Unpack). Unpacked tuples are joined to the observed tuples (i.e., if t_0 is observed and t_{u1} and t_{u2} are unpacked, then the resulting tuples are t_0t_{u1} and t_0t_{u2}). Both Pack and Emit can group tuples based on matching fields and perform simple aggregations such as SUM and COUNT.

Query Results

Advice can emit tuples as output of a query using the Emit instruction (⑥). Pivot Tracing first aggregates emitted tuples locally within each process, then reports results globally at a regular interval, e.g., once per second (⑦). The Pivot Tracing front-end collects and forwards query results to the user (⑧). Process-level aggregation substantially reduces traffic for emitted tuples; Q2 is reduced from approximately 600 tuples per second to six tuples per second from host.

Pivot Tracing Example

Recall query Q2 from our earlier Hadoop example:

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

Q2 compiles to two advice specifications, A1 and A2, to be invoked at the `ClientProtocols` and `DataNodeMetrics` tracepoints, respectively:

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

When a request invokes any of the `ClientProtocols` methods, the instrumented code will invoke advice A1. The advice will observe the value of the `procName` variable and pack a tuple into the request's baggage, e.g., `<procName="HGet">`. The request will continue execution, carrying this tuple in its baggage. If the request subsequently invokes the `DataNodeMetrics.incrBytes-Read` method, the instrumented code will invoke advice A2. The advice will unpack the previously packed `procName` and observe the local value of the delta variable, e.g., `<delta=10>`. The advice will then join the unpacked procName with the observed delta and emit the result as output, e.g., `<procName="HGet", delta=10>`. The output tuple will be aggregated with other tuples in the process's Pivot Tracing agent and included in the next interval's query results.

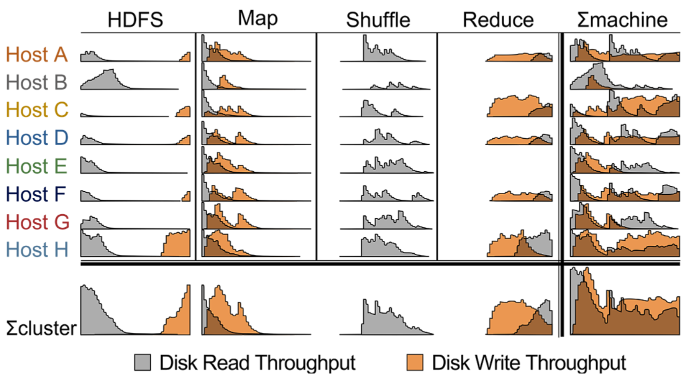


Figure 4: Pivot table showing disk read and write sparklines for MRsort10g. Rows group by host machine; columns group by source process. Bottom row and right column show totals, and bottom-right corner shows grand total.

Figure 4 gives a final demonstration of how Pivot Tracing can group metrics along arbitrary dimensions. It is generated by two queries similar to Q2 that instrument Java's `FileInputStream` and `FileOutputStream`, still joining with the client process name. We show the per-machine, per-application disk read and write throughput of `MRsort10g` from the same experiment. This figure resembles a pivot table, where summing across rows yields per-machine totals, summing across columns yields per-system totals, and the bottom-right corner shows the global totals. In this example, the client application presents a further dimension along which we could present statistics.

Summary

In this article we gave an overview of how Pivot Tracing can evaluate cross-component monitoring queries dynamically at runtime using a combination of dynamic instrumentation and causal tracing. For full details of Pivot Tracing's design and implementation, we refer the reader to the full paper [5] and project Web site. In our full evaluation, we present several case studies where we used Pivot Tracing to successfully diagnose root causes, including real-world issues we encountered in our cluster. We also evaluate the overheads imposed by Pivot Tracing, including the additional costs of invoking advice and the overheads of propagating tuples alongside requests at runtime. Of the examples presented in this article, Q2 only required the propagation of a single tuple per request, and imposed less than 1% overhead in terms of end-to-end latency on several application-level HDFS benchmarks.

Pivot Tracing is the first monitoring system to combine dynamic instrumentation with 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, and the overheads of evaluating the happened-before join are sufficiently low that we believe Pivot Tracing is suitable for production systems, both for high-level standing queries and for digging deeper when necessary. Ultimately, its power lies in the uniform and ubiquitous way in which it integrates monitoring of a heterogeneous distributed system.

References

[1] Bryan Cantrill, Michael W Shapiro, and Adam H Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, 2004, pp. 15–28.

[2] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz, "Fay: Extensible Distributed Tracing from Kernels to Clusters," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, 2012.

[3] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica, "X-Trace: A Pervasive Network Tracing Framework," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[4] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, 1978, pp. 558–565.

[5] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[6] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating System Problems Using Dynamic Instrumentation," in *Proceedings of the Ottawa Linux Symposium (OLS)*, 2005.

[7] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Research, 2010.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Doing research in the security field? Consider submitting your work to these upcoming events.

ASE '16: 2016 USENIX Workshop on Advances in Security Education

August 9, 2016, Austin, TX
Paper submissions due: May 3, 2016
www.usenix.org/ase16

The 2016 USENIX Advances in Security Education Workshop (ASE '16) is a new workshop, co-located with the 25th USENIX Security Symposium, designed to be a top-tier venue for cutting-edge research, best practices, and experimental curricula in computer security education.

CSET '16: 9th Workshop on Cyber Security Experimentation and Test

August 8, 2016, Austin, TX
Submissions due: May 3, 2016
www.usenix.org/cset16

The CSET workshop invites submissions on cyber security evaluation, experimentation, measurement, metrics, data, simulations, and testbeds.

SOUPS 2016: Twelfth Symposium on Usable Privacy and Security

June 22-24, 2016, Denver, CO
Poster submissions due: May 16, 2016
Lightning Talks and Demos early submissions due: May 16
www.usenix.org/soups2016

Posters: High-quality poster presentations are an integral part of SOUPS. We seek poster abstracts describing recent or ongoing research related to usable privacy and security. SOUPS will include a poster session in which authors will exhibit their posters. Accepted poster abstracts will be distributed to symposium participants and made available on the symposium Web site. Interactive demos alongside posters are welcome and encouraged. We also welcome authors of recent papers on usable privacy and security (2015 to 2016) to present their work at the SOUPS poster session.

Lightning Talks: A continuing feature of SOUPS is a session of 5-minute talks and 5- to 10-minute demos. These could include emerging hot topics, preliminary research results, practical problems encountered by end users or industry practitioners, a lesson learned, a research challenge that could benefit from feedback, a war story, ongoing research, a success, a failure, a future experiment, tips and tricks, a pitfall to avoid, exciting visualization, new user interface or interaction paradigm related to security and privacy. etc. Demo presentations should convey the main idea of the interface and one or more scenarios or use cases.

WOOT '16: 10th USENIX Workshop on Offensive Technologies

August 8-9, 2016, Austin, TX
Submissions due: May 17, 2016
www.usenix.org/woot16

The USENIX Workshop on Offensive Technologies (WOOT) aims to present a broad picture of offense and its contributions, bringing together researchers and practitioners in all areas of computer security. Offensive security has changed from a hobby to an industry. No longer an exercise for isolated enthusiasts, offensive security is today a large-scale operation managed by organized, capitalized actors. Meanwhile, the landscape has shifted: software used by millions is built by startups less than a year old, delivered on mobile phones and surveilled by national signals intelligence agencies.

FOCI '16: 6th USENIX Workshop on Free and Open Communications on the Internet

August 10, 2016, Austin, TX
Submissions due: May 19, 2016
www.usenix.org/foci16

The 6th USENIX Workshop on Free and Open Communications on the Internet (FOCI '16), to be held on August 8, 2016, seeks to bring together researchers and practitioners working on means to study, detect, or circumvent practices that inhibit free and open communication on the Internet.

www.usenix.org/cfp