Abstract of "A Universal Architecture for Cross-Cutting Tools in Distributed Systems" by Jonathan Mace, Ph.D., Brown University, May 2018.

Recent research has proposed a variety of cross-cutting tools to help monitor and troubleshoot end-to-end behaviors in distributed systems. However, most prior tools focus on data collection and aggregation, and treat analysis as a distinct step to be performed later, offline. This restricts the applicability of such tools to only doing post-facto analysis. However, this is not a fundamental limitation. Recent research has proposed tools that integrate analysis and decision-making at runtime, to directly enforce end-to-end behaviors and adapt to events.

In this thesis I present two new applications of cross-cutting tools to previously unexplored domains: resource management, and dynamic monitoring. Retro, a cross-cutting tool for resource management, provides end-to-end performance guarantees by propagating tenant identifiers with executions, and using them to attribute resource consumption and enforce throttling decisions. Pivot Tracing, a cross-cutting tool for dynamic monitoring, dynamically monitors metrics and contextualizes them based on properties deriving from arbitrary points in an end-to-end execution.

Retro and Pivot Tracing illustrate the potential breadth of cross-cutting tools in providing visibility and control over distributed system behaviors. From this, I identify and characterize the common challenges associated with developing and deploying cross-cutting tools. This motivates the design of baggage contexts, a general-purpose context that can be shared and reused by different cross-cutting tools. Baggage contexts abstract and encapsulate components that are otherwise duplicated by most cross-cutting tools, and decouples the design of tools into separate layers that can be addressed independently by different teams of developers.

The potential impact of a common architecture for cross-cutting tools is significant. It would enable more pervasive, more useful, and more diverse cross-cutting tools, and make it easier for developers to defer development-time decisions about which tools to deploy and support.

A Universal Architecture for Cross-Cutting Tools in Distributed Systems

by
Jonathan Mace
Sc.M., Brown University, 2014
M.Math.Comp., Oxford University, 2009

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2018

This dissertation by Jonathan Mace is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ________________ ____
Rodrigo Fonseca, Director

Recommended to the Graduate Council

Date ________________ ____
Maurice Herlihy, Reader

Date ________________ ____
Shriram Krishnamurthi, Reader

Approved by the Graduate Council

Date ________________ ____
Andrew G. Campbell
Dean of the Graduate School

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Thesis Statement Many tools for monitoring and enforcing distributed systems capture information about end-to-end executions by propagating in-band contexts. We characterize a broad class of such *cross-cutting tools*, and extend these ideas to new applications in resource management and dynamic monitoring We identify underlying commonalities in this class of tools, and propose an abstraction layering that simplifies their development, deployment, and reuse.

1.1 Cross-Cutting Challenges in Distributed Systems

Distributed systems represent some of the most interesting and successful computing applications in use today, from web search and social networks, to data analytics and large-scale machine learning, to loosely-coupled microservices, serverless lambdas, and the public cloud. The prevailing approach is to implement cloud systems as user-space applications running atop commodity PC hardware. System processes run across many different machines, communicate and co-ordinate over the local network, and implement important features such as fault tolerance at the application level, instead of in hardware. This approach leads to scalable and cost-effective systems [198], as system capacity can be increased by simply adding more machines, which scales linearly in cost.

However, it is notoriously difficult to understand, troubleshoot, and enforce cloud systems behaviors. The potential problems are myriad: hardware and software failures, misconfiguration, hot spots, aggressive tenants, or even simply unrealistic user expectations [6,132,145,166,198]. A single execution in the system — *e.g.*, an action such as loading a page on facebook.com, or a data analytics job — might entail complex executions spanning clients, networks, and distributed back-end services. Meanwhile, the symptoms of an issue can manifest in components far removed from the root cause, including in different processes, machines, and application tiers, and they may be visible to different users; transient factors are often to blame, symptoms are masked by fault tolerance mechanisms, and instead of an outright crash, systems experience more pernicious symptoms like sustained degraded performance [145,148]. When they materialize in production systems (as they have with all major cloud providers [145]) it leads to high-profile outages and a massive loss of revenue.

These challenges stem from the fact that distributed systems lack a central point of visibility and control over end-to-end executions. Each component only performs a narrow slice of work, and to execute a high-level task (such as a search query or an analytics job) entails a complex flow across multiple processes, machines, and the network. The tools and abstractions traditionally useful for diagnosing problems in standalone programs — *e.g.* the execution stack, thread IDs, thread-local variables, debuggers, profilers, and many more — are ineffective or inadequate in this setting because they cannot coherently reason about the system when executions cross boundaries between software components and machines [142,148]. Similarly, traditional, well-studied resource isolation and management techniques in the OS and hypervisor — *e.g.* myopic thread, process, network, and IO schedulers — are insufficient for distributed systems because they lack end-to-end visibility of requests, which contend for resources both within processes and across component and machine boundaries.

Consequently, most systems today do not embed monitoring or logging that can coherently reason about end-to-end executions. The tools and abstractions commonly used are ad-hoc and myopic; systems lack end-to-end resource management; and executions lose important context when they cross software component and machine boundaries. This makes it difficult to answer questions about causes of failures, uncover dependencies between components, understand performance or resource usage, and provide end-to-end performance guarantees or isolation between tenants. Many organizations face debugging, monitoring, and troubleshooting challenges after deploying distributed systems [6,132,145,166,198], and past systems have suffered cascading failures [10,72,145], slowdown [33,36,83,145,174], and even cluster-wide outages [10,33,145,178] as a result.

1.2 The Need for Cross-Cutting Tools

To address these challenges, we inevitably need to correlate and integrate data that crosses component, system, and machine boundaries, *i.e.* a user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. A growing body of research addresses this observation by maintaining the notion of a context that follows the execution of applications through events, queues, thread pools, files, caches, and messages between distributed system components. Contexts are a powerful mechanism for capturing causal relationships between events on the execution path at runtime, and have enabled a number of tools, both in research and in practice [132,142,166,197,198]. Of these, the most widespread are end-to-end tracing tools [111,135,227,244], which record causality between logs and events across components by propagating request and event IDs.

However, while end-to-end tracing is primarily concerned with aggregation of traces and offline analysis, a growing number of distributed system tools utilize context propagation for *online* tasks. For example, in recent research, tenant IDs enable coordinated scheduling decisions across components [241]; latency measurements enable tools to adapt to bottlenecks or processing delays [114,225]; user tokens enable tools for auditing security policies [6,237,244]; causal histories enable cross-system consistency checking [179]; and more. We refer to this broader class of tools as *cross-cutting tools*, to stress their use beyond recording traces.

Cross-cutting tools represent a compelling approach to tackling problems in observing and enforcing distributed system behaviors. However, it is an open question the extent of problems to which cross-cutting tools can apply. Furthermore, despite a growing proliferation of cross-cutting tools, there remain significant challenges to their development and pervasive, end-to-end deployment. We find that in practice, few cross-cutting tools get deployed pervasively in large systems. When they do, they are brittle, hard to evolve, and cannot coexist with each other.

1.3 Thesis Goals and Contributions

The goal of this thesis is to investigate the broad scope of monitoring and enforcement tasks that can be addressed by cross-cutting tools, and to address fundamental challenges affecting our ability to develop and deploy such tools. This thesis explores two new applications of cross-cutting tools to previously unexplored domains: resource management, and dynamic monitoring. Subsequently, to address challenges in the development and pervasive deployment of cross-cutting tools, we propose common abstractions to underpin their design. The contributions of this thesis are as follows:

Retro Retro (Chapter 4) is a cross-cutting tool for resource management, that provides guarantees to end-to-end workflows. Retro performs online resource management and, in real-time, attributes resource consumption to tenants and makes per-tenant scheduling decisions. A key challenge addressed by Retro is that end-to-end performance can be affected by resource congestion within any process visited by an execution. To address this, Retro adapts cross-cutting tool components to track tenant executions and resource usage in shared distributed systems. Retro is an example of an online, reactive cross-cutting tool that continually monitors and adjusts system parameters.

Pivot Tracing Pivot Tracing (Chapter 6) is a cross-cutting tool for dynamically monitoring metrics, and contextualizing them based on properties that derive from any point in an end-to-end execution. Pivot Tracing gives users the ability to obtain metrics from one point of the system while selecting, filtering, and grouping by events from other parts of the system. Pivot Tracing addresses two key challenges faced by distributed systems today: monitoring and logging decisions are made a priori, at development time; and existing monitoring and logging tools lack cross-component visibility. Pivot Tracing generalizes the context propagation seen in previous cross-cutting tools, and uses it to propagate partial query state alongside requests.

Baggage Contexts Together with other recent applications seen in the research literature, Retro and Pivot Tracing illustrate the potential breadth of cross-cutting tools in providing visibility and control over distributed system behaviors. From this, we identify and characterize the common challenges associated with developing and deploying cross-cutting tools (Chapter 7). This motivates the design of *baggage contexts* – general-purpose context propagation that can be shared and reused by different cross-cutting tools (Chapter 8). Baggage contexts abstract and encapsulate common context propagation components that are otherwise duplicated by most cross-cutting tools, and decouples the design of tools into separate layers that can be addressed independently by different teams of developers. The potential impact of a common architecture for cross-cutting tools is significant. It would enable more pervasive, more useful, and more diverse cross-cutting tools, and make it easier for developers to defer development-time decisions about which tools to deploy and support.

Chapter 2

Cross-Cutting Tools

2.1 Cloud Distributed Systems

Distributed systems underpin some of the most interesting and successful computing applications in use today, from web search and social networks, to data analytics and large-scale machine learning, to loosely-coupled microservices, serverless lambdas, and the public cloud. A typical cloud distributed system is not unlike the standalone software that runs on laptop or desktop PCs. It consists of several user-space processes, written in common programming languages like Java or C++. The processes run on off-the-shelf operating systems such as Linux, atop commodity PC hardware. The main difference compared to standalone software is that distributed system components communicate and co-ordinate with each other across the network, in order to execute their high-level tasks [136]. Since commodity PC hardware lacks the reliability guarantees previously provided by, *e.g.* mainframes, failure-tolerance mechanisms are typically implemented at the application level, within the distributed system software [120]. While this increases the complexity of the distributed system software, this approach is nonetheless an extremely cost-effective way of tackling the scalability challenges posed by internet-scale problems like web search. Today, this approach is the prevailing way of building cloud distributed systems [121,136].

A typical deployment of cloud systems will comprise not just one distributed system but many, inter-operating systems, often co-locating their processes atop the same physical hardware. For example, early work from Google presented individual systems providing storage [136], database [108], and processing [121], which together were combined to solve problems such as web indexing and search. Other examples include storage, configuration management, database, queueing, and co-ordination services, such as Azure Storage [103], Amazon Dynamo [122], HDFS [242], HBase [40], ZooKeeper [153], and many more. Similarly, the Hadoop ecosystem of open-source distributed systems comprises dozens of different systems, many of which are designed to compose with each other [228]. In turn, this has also prompted a wide variety of new architectures and frameworks, such as microservices architectures [197], function-as-a-service [9], and similar commercial offerings [8, 139, 191]. To be cost-effective, typical cloud deployments will host many different users and workloads concurrently, sharing the same machines, processes, and the network.

2.2 Cross-Cutting Executions

A fundamental difference between standalone programs and distributed systems is that distributed systems lack a central point of visibility and control. When we execute a program in a distributed system, each component of our distributed system will perform only a narrow slice of work. Each high-level task – such as a search query or an analytics job – will entail a complex flow across multiple processes, machines, and the network. At the same time, many of the processes visited by executions are *shared* processes, that handle executions of many different users simultaneously; these are multiplexed at the *application level*, within the processes of the distributed system [136, 242]. Given this, the scope of a single end-to-end execution will include many narrow slices of work, contributed by many different processes and machines. We refer to this as the *cross-cutting dimension* that is scoped to end-to-end executions. It cuts across processes, components, and machines, and includes all of the work done to complete a high-level, end-to-end task. In distributed systems, the cross-cutting dimension is orthogonal to machines, processes, and threads. It is an important dimension for reasoning about end-to-end executions, because the correct behavior and performance of an end-to-end execution can be influenced by events occurring anywhere in the cross-cutting dimension.

In this context, monitoring, understanding, and enforcing the behaviors of distributed systems is extremely challenging. Each cross-cutting execution can be influenced by a wide range of factors based on the processes it visits, other workloads present at the same time, differences in deployed software versions, and many more. Dynamic factors also influence performance, such as continuous deployment of new code, changing configurations, user-specific experiments, and datacenters with distinct characteristics [157]. The potential problems are myriad: hardware and software failures, misconfiguration, hot spots, aggressive tenants, or even simply unrealistic user expectations [6,132,145,166,198]. The symptoms of an issue can manifest in components far removed from the root cause, including in different processes, machines, and application tiers, and they may be visible to different users [145, 148]. Transient factors are often to blame; symptoms can be masked by fault tolerance mechanisms; and instead of an outright crash, systems experience more pernicious symptoms like sustained degraded performance [127, 145]. When problems materialize in production systems (as they have with all major cloud providers) it leads to high-profile outages and a massive loss of revenue.

2.3 Troubleshooting Across Boundaries

Because distributed system executions are inherently cross-cutting, to diagnose problems one often needs to correlate and integrate data that crosses component, system, and machine boundaries. That is, a user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. However, the tools and abstractions traditionally useful for diagnosing standalone programs — *e.g.* the execution stack, thread IDs, thread-local variables, debuggers, profilers, and many more — are ineffective or inadequate in this setting because they do not align with the cross-cutting dimension, and they cannot coherently reason about the system when executions cross boundaries between software components and machines [142, 148]. Similarly, traditional, well-studied resource isolation and management techniques in the OS and hypervisor — *e.g.* myopic thread, process, network, and IO schedulers — are insufficient for distributed systems because they too do not align with the cross-cutting dimension, and thereby lack end-to-end visibility of executions, which contend for resources both within processes and across component and machine boundaries.

Most systems today do not embed monitoring that can relate to other systems, lack end-to-end resource management, and lose important context when requests cross software component and machine boundaries. This makes it difficult to answer questions about causes of failures, uncover dependencies between components, understand performance or resource usage, and provide end-to-end performance guarantees or isolation between tenants. Many organizations face debugging, monitoring, and troubleshooting challenges after deploying distributed systems [6, 132, 145, 166, 198], and past systems have suffered cascading failures [10, 72, 145], slowdown [33, 36, 83, 145, 174], and even cluster-wide outages [10, 33, 145, 178] as a result. In describing this problem, a Twitter engineer writes *"the tried and true tools we're used to — configuration management, log processing, strace, tcpdump, etc — prove to be crude and dull instruments when applied to microservices"* [142]; for Hailo, *"rationalising the behaviour of each individual component may be simpler, but understanding the behaviour of the whole system, and ensuring correctness, is more difficult."* [148]. @Honest_Update is more candid: *"We replaced our monolith with micro services so that every outage could be more like a murder mystery."* [151].

2.4 End-to-End Tracing

To address these challenges, distributed systems inevitably need the ability to correlate events at one point of the system with events that are meaningful from other parts of the system. In the past decade, end-to-end tracing has emerged as a valuable tool for analyzing and diagnosing problems in distributed systems [230].

2.4.1 Overview

End-to-end tracing frameworks [230] record *traces* of end-to-end executions, by recording *events* (*i.e.*, logging statements) across multiple machines, and explicitly recording event timing, ordering, and causality (according to Lamport's happens before relation [171]). A common representation for the traces recorded by such frameworks is an *execution graph* — a directed, acyclic graph that describes the path of a single request through components of a distributed system [173]. An individual trace provides a description of a request as it traverses a distributed system – it captures the path of the request, performance costs incurred at the components visited, and timings between events. Traces incorporate a wide range of information from all components on the cross-cutting execution path of requests, from clients through backend services [157]. Causal relationships between events provide further information about the concurrency and dependencies of execution. Collectively, execution graphs can represent aggregate system behavior - capturing the commonly-traversed paths, corner case executions, and distributions over paths and timings.

The key component of tracing frameworks that enables capturing event causality is *metadata propagation*. To deploy an end-to-end tracing framework in a distributed system, developers must first modify their system to (i) generate events using the framework's APIs; and (ii) propagate *metadata* alongside the entire end-to-end execution. The latter of these – metadata propagation – is necessary for attributing events to executions and capturing relationships between events. Typically, the metadata includes a unique ID for the execution (*e.g.* a RequestID) as well as identifiers for the most recently generated events (*e.g.* a PreviousEventID). Each time an event is generated, the framework will annotate the event with these identifiers (and update the propagated identifiers for the most recently generated events). Backend components of the tracing framework (*e.g.*, databases) will receive and persist events generated across the distributed system. The attached identifiers enable the backend to group events related to the same requests, construct the graph of each trace, and provide interfaces for users to view and explore traces.

2.4.2 Implementations and Use Cases

End-to-end tracing frameworks exist both in academia [97,106,135,227,257] and in industry [128,157,161,244,245], with notable early examples including X-Trace [135] and Google's Dapper [244]. Recent commercial tracing offerings include Amazon's X-Ray [98] and Google's Stackdriver Trace [99], which provide cloud-hosted backends and instrumentation APIs to ingest trace data from users' applications. Popular open-source frameworks largely derive from Dapper [244], and include Zipkin [259], HTrace [62], Jaeger [238], and others [196,201,246,274]. As these tracing systems have matured, they have begun to converge and standardize the semantics of this class of tracing tools, with the recent OpenTracing effort [201].

Tracing frameworks have been used for a variety of purposes in monitoring and troubleshooting distributed system behaviors. This includes both manual analysis of traces, and automated approaches that reason about traces in aggregate. Examples include diagnosing anomalous requests whose structure or timing deviate from the norm [3,97,110,111,212,231]; diagnosing steady-state problems that manifest across many requests [135,227,231,244,257]; identifying slow components and functions [106,184,244]; modelling workloads and resource usage [96,97,184,257]; explaining structural and statistical differences between 'before' and 'after' traces [184,231]; reconciling observed system behavior with a description of the developer's expected behavior [227]; auditing security [244]; critical-path analysis [184, 212]; deriving high-level performance metrics [157]; recording resource consumption [257]; and optimizing client-server web requests [155,176]. Recent work has extended these techniques to continuous profiling and analysis [146,157,188–190,284].

Table 2.1 highlights use cases for tracing tools deployed in production at 26 companies. Examples range from debugging individual anomalous requests, to capturing performance, resource, and latency metrics, to correlating failures and behavioral clustering. Nonetheless, there remains a lack of consensus and standards on tools for understanding, monitoring, troubleshooting, and enforcing distributed system behaviors. A recent study identified security, performance, tracing, and monitoring as highly important yet overlooked areas [6]. Despite this proliferation of tracing tools there remain significant challenges to their development and pervasive, end-to-end deployment, and distributed tracing and logging is still described as *"the most wanted and missed tool in the micro-service world"* [281].

2.5 Cross-Cutting Tools

The primary use case of end-to-end tracing frameworks is *offline analysis*, with a distinct division between the system-level components of the tool for capturing traces at runtime, and the post-facto aggregation and trace analysis components. However, more recent research has considered a wider variety of *online* monitoring, diagnosis, and enforcement tasks, by adapting some of the concepts seen in end-to-end tracing and eschewing others. Instead of just offline analysis, these tools observe and analyze events in-band during executions, then make immediate decisions about actions to take, such as changing system or tool behaviors. We refer to this broad class of tools as *cross-cutting tools*, characterized by the fact that the tasks they perform align with cross-cutting executions. We include end-to-end tracing frameworks in this class of tools, but we emphasize that cross-cutting tools as a whole encompass a broader range of use cases than just recording traces.

Cross-cutting tools expand upon the ability of end-to-end tracing frameworks to correlate events at one point of the system with events that are meaningful from other parts of the system. Building upon the metadata propagation techniques used by end-to-end tracing frameworks, these tools maintain the notion of a context that follows the execution patterns of applications through events, queues, thread pools, files, caches, and messages between distributed system components. Context propagation entails that for every execution (*e.g.* request, task, job, etc.), the system forwards a context object alongside the execution, across all process, component, and machine boundaries, with metadata about the execution. Context propagation requires small but non-trivial interventions at the source-code level, within all distributed system components, to maintain and propagate contexts alongside executions. Contexts are a powerful mechanism for observing and capturing causal relationships between events on the execution path at runtime.

Beyond end-to-end tracing, cross-cutting tools have a diverse range of goals and applications. Correspondingly, different cross-cutting tools make use of contexts in different ways, to achieve different purposes. For example, many systems propagate tenant IDs for use in resource accounting and resource management [180, 241], and failure testing [149]. Energy tracking systems propagate activity IDs for attributing energy consumption to high-level activities [133]. Taint tracking and DIFC propagate security labels as the system executes, warning of or prohibiting policy violations [129, 193, 276]. User tokens enable tools for auditing security policies [6, 237, 244] and identifying business flows [237]. Data provenance systems propagate information about the lineage of data as different components manipulate it [123, 192]. Tools for end-to-end latency, path profiling, and critical path analysis propagate partial latency measurements and information about execution paths and graphs [106, 114, 225, 225, 253]. Metric-gathering systems propagate labels and query state so that downstream components can select, group, and filter statistics [141, 182, 237]. In the networking literature, tools for attributing latency in virtualized network stacks propagate timestamps with packets [254]. Recent work has proposed cross-cutting tools that propagate causal histories to validate cross-system consistency [179].

Other proposed use cases for cross-cutting tools include propagating fault instructions for chaos engineering, and markers for capacity planning [237].

Note that some of these tools use write-once contexts, while others constantly change and update the context data. Some tools like end-to-end tracing frameworks only record information, while others use context data to take actions at runtime. We broadly distinguish two different classes of cross-cutting tools. The first class encompasses the early tools presented in the literature – primarily end-to-end tracing frameworks – and we term these *first-generation* cross-cutting tools. First-generation tools separate the techniques for capturing traces at runtime, from the subsequent (offline) analysis step. We refer to the later class of tools – which perform adaptive or online tasks – as *second-generation* cross-cutting tools. Second-generation tools incorporate elements of analysis that were previously part of a post-facto stage, with the runtime techniques for observing events and causal relationships. This enables adaptive tasks that can immediately react to observed events; make analysis decisions on the fly; and avoid the computational overheads of aggregating verbose traces.

2.6 Goal: Second-Generation Cross-Cutting Tools

The central theme of this thesis is the following: to what extent can second-generation cross-cutting tools address the challenges of monitoring, understanding, and enforcing distributed system behaviors? To answer this question, this thesis will explore new applications of second-generation cross-cutting tools, and examine new abstractions to underpin their design.

The first goal of this thesis is to examine the application of cross-cutting tools to two new domains, with very different end-to-end goals. We present two cross-cutting tools. Retro (Chapter 4) is a resource management framework for providing end-to-end performance guarantees; it propagates tenant identifiers alongside requests, and uses them to attribute resource consumption to tenants and make per-tenant scheduling decisions. Pivot Tracing (Chapter 6) is a monitoring framework that gives users the ability to obtain metrics from one point of the system while selecting, filtering, and grouping by events from other parts of the system; it propagates partial query state alongside requests.

Together, these new applications illustrate the potential breadth of cross-cutting tools in providing visibility and control over distributed system behaviors. The second goal of this thesis is to identify and characterize the common challenges associated with developing and deploying cross-cutting tools (Chapter 7). This motivates the design of *baggage contexts* – general-purpose context propagation that can be shared and reused by different tracing tools (Chapter 8). Baggage contexts abstract and encapsulate common context propagation components that are otherwise duplicated by most cross-cutting tools, and decouples the design of tools into separate layers that can be addressed independently by different teams of developers. The potential impact of a common architecture for cross-cutting tools is significant. It would enable more pervasive, more useful, and more diverse cross-cutting tools, and make it easier for developers to defer development-time decisions about which tools to deploy and support.

2.7 Other Approaches

The focus of this thesis is abstractions for, and applications of, second-generation cross-cutting tools. As described, this work primarily extends prior research in the area of end-to-end tracing. However, this is not the only avenue of research in understanding, troubleshooting, and enforcing distributed system behaviors. In this section, we give a brief overview of the related work in other areas.

2.7.1 Alternatives to Context Propagation

Both first- and second-generation cross-cutting tools utilize context propagation in order to observe the relationships between events occurring in distributed systems. Context propagation requires small but non-trivial interventions at the source-code level, within all distributed system components, to maintain and propagate contexts alongside executions. Researchers and practitioners consistently describe instrumentation as the most time consuming and difficult part of deploying tracing frameworks [113, 134, 135, 162, 230, 243].

Consequently, prior work has considered alternative ways of establishing causal relationships between events. Alternative approaches include combining identifiers – *i.e.* call ID, IP address, etc.– present across multiple logging statements [96, 113, 163, 257, 275, 282]; inferring causality using machine learning and statistical techniques [101, 184, 200, 275]; and augmenting models with static source code analysis [275, 282, 283].

For example, Magpie [96, 97] 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. In a similar vein, Pip [227] expects developer-provided annotations at execution boundaries (*e.g.* to match up send and receive calls). Zhao *et al.* [282] captured this notion as the "Flow Reconstruction Principle", claiming that *"programmers will output sufficient information to logs so as to be able to reconstruct runtime execution flows after the fact"*.

These approaches avoid context propagation altogether; however, the use cases are limited to offline analysis of the information exposed by the system through logs, and they do not apply to online use cases such as scheduling and data quality trade-offs. It is also challenging to scale black-box analysis because inferring causal relationships is expensive; for example, computing a Facebook model from 1.3M traces took 2 hours for the Mystery Machine [113]. Furthermore, these approaches require verbose, broad-brush logging, whereas cross-cutting tools can target specific executions and coherently sample cross-cutting executions. Since the goal of this thesis is to explore new avenues in second-generation cross-cutting tools, we only consider approaches using context propagation. Of course, the trade-off is that context propagation requires explicit source-code instrumentation, and cannot treat the system as a black box.

2.7.2 Alternatives to Cross-Cutting Tools

Existing tools for troubleshooting distributed systems apply to a range of use cases and make use of a range of different input data. In addition to the cross-cutting tools described, they include tools that ingest existing per-process and per-component logs [163,194,275]; state-monitoring systems that track system-level metrics and performance counters [185]; and aggregation systems to collect and summarize application-level monitoring data [169,182,262]. Wang *et al.* provide a comprehensive overview of datacenter troubleshooting tools in [268].

Prior work in troubleshooting distributed systems has presented a variety of analysis techniques: semi-automatically honing in on root causes of performance anomalies [267]; identifying statistical anomalies [163]; online statistical outlier monitoring [94]; and analyzing critical path dependencies, slack, and speedup [113,213]. In most cases, operator-driven exploration is a prerequisite to most automated approaches in order to identify salient features [194]. In a recent analysis of Splunk usage, Alspaugh *et al.* noted that the use of statistical and machine learning inference techniques is *"relatively rare"* and that human inference is the key driver of analyses [7]. At Facebook, Canopy's primary use case is to support ad hoc high-level exploratory analysis, because problems arising in practice are difficult to anticipate and there are a wide range of potential features to explore [157]. Facebook developers made extensive use of statistical comparison techniques to find correlations between features and compare distributions, *e.g.* between application versions.

Beyond first-generation tracing tools, second-generation tools cover a much broader range of use cases, and in particular, these use cases include online and adaptive tasks. For the domains considered in this thesis, we defer discussion of alternative approaches to Chapter 3, Chapter 5 and Chapter 7, where we consider related work specific to each tool.

Company	*Services*	*Engineers*	*Tools*	*Use Cases*
Allegro	250+	500	Zipkin[†][*]	debugging; understanding service dependencies; network traffic analysis; latency monitoring
BBN Technologies	30+	60	Zipkin[†]	understand service dependencies; performance and latency monitoring
Coursera	15+	60	Zipkin[†][*]	dependency visualization; failure correlation and analysis
Etsy	—	200+	CrossStitch[†] [274]	latency monitoring; aggregated analysis
Facebook	—	—	Canopy [157]	mobile analysis; regression analysis
FINN.no	200	120	Zipkin[†]	
Google	—	—	Dapper [244], Census [141]	performance and resource monitoring; security auditing; root-cause analysis
Groupon	400+	1700	Zipkin[†]	performance improvements; architectural understanding; monitoring; SLA enforcement; anomaly detection; ad-hoc exploratory analysis
Hailo	200+	30	*In-House*[†] [148]	debugging; metric aggregation; architectural understanding; network traffic analysis; performance optimizations
Line	24+	200+	Brave, Zipkin[†]	latency monitoring; metrics monitoring
Lookout	15+	100	Zipkin[†]	statistics and metrics monitoring; deployment tooling; client whitelisting;
Lyft	—	—	zend[†] [168]	dependency analysis; latency analysis; mobile device correlations
Medidata Solutions	100	—	Zipkin[†]	system monitoring
Naver	100	2000	Naver Pinpoint[†]	architectural understanding; realtime monitoring; stacktrace sampling; batch analysis
Netflix	100+	1000+	Salp[†] [160]	dependency analysis; ad-hoc offline querying; realtime analysis; critical path analysis
Pinterest	—	—	PinTrace[†] [162]	latency analysis; architectural understanding; debugging; cost attribution; root-cause analysis
Prezi	50	100	Zipkin[†]	latency analysis; service dependency analysis
SmartThings	24+	35	Zipkin[†]	real-time analysis
SoundCloud	50	140	Zipkin[†]	architectural understanding, performance optimizations; latency analysis; batch analysis
Sourcegraph	—	—	Appdash[†]	debugging; performance and latency monitoring
Tracelytics	—	—	TraceView[§]	latency analysis; performance monitoring; realtime monitoring; metric aggregation
TomTom Maps	10+	100+	Brave, Zipkin[†]	statistical analysis and aggregation
Uber	2000+	2000+	Jaeger	architectural understanding, execution clustering; historical analysis; anomaly detection; inspect service dependencies; latency correlations; real-time aggregations
Yahoo	—	—	YTrace [161]	root-cause analysis, cascading failures
Yelp	300+	—	Zipkin[†] [2]	debugging; service dependency analysis; latency analysis
Zalando	100+	1000+	Zalando Tracer[§]	realtime and batch analysis
Zhihu	150+	80+	Zipkin[†]	architectural understanding; metric aggregation; dependency analysis; stack trace analysis; latency analysis

[*] version with extensions or modifications

[†] Dapper [244] derivative

[§] X-Trace [135] derivative

Table 2.1: Information about end-to-end tracing tools deployed at several companies. Information was gathered from the Distributed Tracing Workgroup [124]

Chapter 3

Resource Management in Distributed Systems

The first new domain for cross-cutting tools that we consider in this thesis is *resource management* in multi-tenant distributed systems. This chapter describes how resource management is inherently a cross-cutting concern, and motivates how cross-cutting tools are a compelling choice for managing resources. In the next chapter, we will present the design and implementation of Retro, a second-generation cross-cutting tool for resource management. To motivate Retro, this chapter describes the challenges inherent to managing resources in multi-tenant systems.

3.1 Background

3.1.1 Multi-Tenant Systems

Many important distributed systems and cloud services execute requests of multiple tenants simultaneously. These include storage, configuration management, database, queueing, and co-ordination services, such as Azure Storage [103], Amazon Dynamo [122], HDFS [242], ZooKeeper [153], and many more. Shared systems have clear advantages in terms of cost, efficiency, and scalability.

Key to shared systems is that they execute requests of multiple tenants simultaneously, *within the same shared processes*. Consider the Hadoop Distributed File System (HDFS) [242]. The HDFS NameNode process maintains metadata related to locations of blocks in HDFS. Users invoke various APIs on the NameNode to create, rename, or delete files, create or list directories, or look up file block locations. As in most shared systems, requests to the NameNode wait in an admission queue and are processed in FIFO order by a set of worker threads. In this setting tenant requests contend for resources, such as threads, CPU, disks, or even locks, from *within* the shared process.

3.1.2 Resource Management and Isolation

It is crucial to provide resource isolation in shared systems to ensure that a single tenant cannot get more than its intended share of resources, to prevent aggressive tenants or unpredictable workloads from causing starvation, high latencies, or reduced throughput for others. In such cases, aggressive tenants can overload the shared process and gain an unfair share of resources. In the extreme, this lack of isolation can lead to denial-of-service for well-behaved tenants and even system wide outages.

Systems in the past have suffered cascading failures [10, 145], slowdown [33, 36, 83, 145, 174], and even cluster-wide outages [10, 33, 145] due to aggressive tenants and insufficient resource isolation. For example, eBay Hadoop clusters regularly suffered denial of service attacks caused by heavy users overloading the shared HDFS NameNode [174, 175]. HDFS users report slowdown for a variety of reasons: poorly written jobs making many API calls [83]; unmanaged, aggressive background tasks making too many concurrent requests [72]; and computationally expensive APIs [36]. Impala [170] queries can fail on overloaded Kudu [178] clusters due to request timeouts and a lack of fair sharing [88]. Cloudstack users can hammer the shared management server, causing performance issues for other users or even crashes [33]. Guo et al. [145] describe examples where a lack of resource management causes failures that cascade into system-wide outages: a failure in Microsoft's datacenter where a background task spawned a large number of threads, overloading servers; overloaded servers not responding to heartbeats, triggering further data replication and overload; GMail and Skype outages where load balancing and front-end admission overloads individual nodes; and more. A 2015 Amazon DynamoDB outage [10] was caused by expensive API calls overloading a shared membership server, cascading to a system-wide outage.

Ideally, multi-tenant service providers should be able to implement resource management policies with various high-level goals – *e.g.*, admission control, fairness, guaranteed performance, or usage limits. These policies enable the provider to guarantee service-level objectives (SLOs) to a tenant, while simultaneously supporting other tenants with differing workload characteristics. Equally important, these policies can ensure that a tenant does not trigger a system-wide outage by adversarially or inadvertently starving essential background tasks of required resources.

3.1.3 Cross-Cutting Resource Management

Traditionally, resource management has been implemented using OS-level primitives at the granularity of processes or users (*e.g.*, cgroups [187]) or using hypervisors that provide similar isolation among virtual machines. There is also some progress in providing network performance guarantees to groups of VMs [219, 236].

However, when tenants compete inside a process, traditional and well-studied resource management techniques in the operating system and hypervisor are unsuitable for protecting tenants from each other. In this setting there is a mismatch in granularity between resource management and the existing mechanisms: on the one hand tenants share the same processes, thus sharing fine-grained resources within the processes (*e.g.* using the same data structures, thread pools, and locks); on the other hand, several processes spanning machines work on behalf of the same tenant. A single end-to-end execution will consume resources across multiple processes and machines, including fine-grained application-level resources within each process visited, as well as machine-level resources like disk, network, and CPU. This dimension of resource consumption is the same as the cross-cutting dimension.

The performance experienced by an end-to-end execution is, by definition, affected by every resource consumed along the cross-cutting execution path. To achieve our desired end-to-end resource management and performance guarantees, we cannot rely on traditional OS or hypervisor mechanisms, because they only align with processes, threads, or machines. Instead, we seek new mechanisms that orient with cross-cutting executions. This mismatch in granularity motivates our exploration of cross-cutting tools for resource management.

3.2 Hadoop Architecture

Before describing the challenges of resource management in multi-tenant distributed systems, we first give a high-level overview of Hadoop components. Though this work is presented in the context of the Hadoop stack, the results generalize to other distributed systems as well.

Figure 3.1 shows the relevant components of the Hadoop stack. HDFS [242], the distributed file system, consists of DataNodes (DN) that store replicated file blocks and run on each worker machine, and a NameNode (NN) that manages the filesystem metadata. Yarn [264] comprises a single ResourceManager (RM), which communicates with NodeManager (NM) processes on each worker. Hadoop MapReduce is an application of Yarn that runs its processes (application master and map and reduce tasks) inside Yarn containers managed by NodeManagers. HBase [40] is a data store running on top of HDFS that consists of RegionServers (RS) on all workers and an HBase Master, potentially co-located with the NameNode or Yarn. Finally, ZooKeeper [153] is a system for distributed coordination used by HBase.

MapReduce job input and output files are loaded from HDFS or HBase, but during the job's *shuffle* phase, intermediate output is written to local disk by mappers (bypassing HDFS) and then read and transferred by NodeManagers to reducers. Reading and writing to HDFS has the NameNode on the critical path to obtain block metadata. An HBase query executes on a particular RegionServer and reads/writes its data from one or many DataNodes.

In such deployment, a large number of processes share the hardware resources on worker nodes. Moreover, each process might concurrently execute requests on behalf of multiple users or background tasks.

3.3 Resource Management Challenges

Resource management shares the overarching challenges described in Chapter 2 that make it difficult to monitor, troubleshoot, and enforce distributed systems. This section elaborates on additional challenges specific to resource management.

Any resource can become a bottleneck Figure 3.2 demonstrates how the latency of an HDFS client can be adversely affected by other clients executing very different types of requests, contending for different resources.

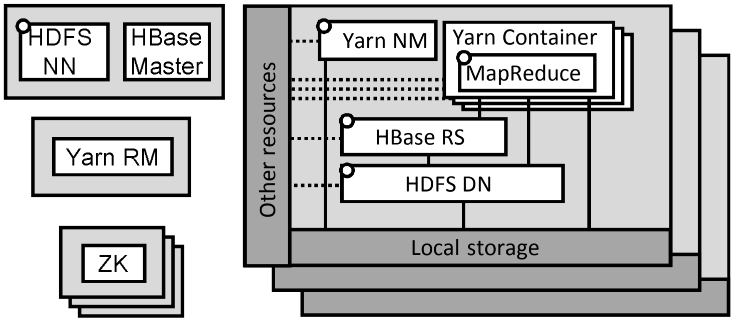


Figure 3.1: Typical deployment of HDFS, ZooKeeper, Yarn, MapReduce, and HBase in a cluster. Gray rectangles represent servers, white rectangles are processes, and white circles represent control points that we added. See text for details.

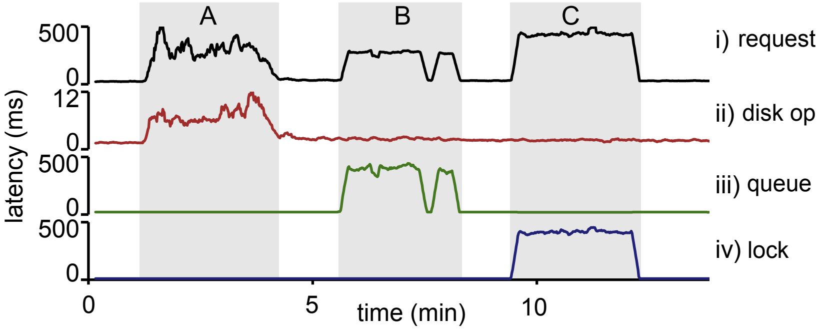
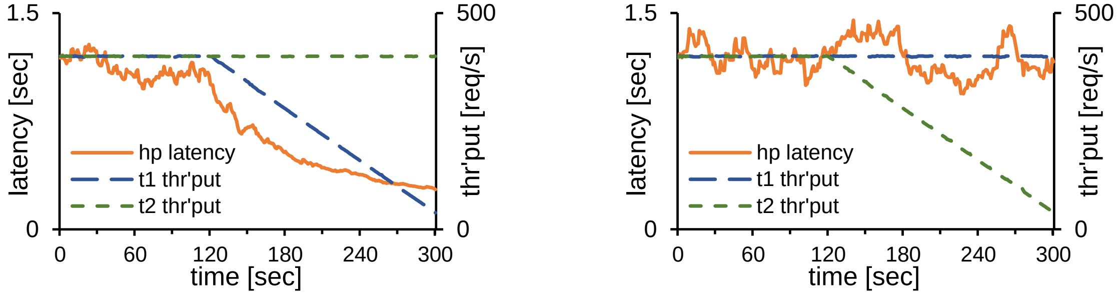


Figure 3.2: i) Average request latency for a client reading 8kB files from HDFS [242] is impacted by different workloads that A: replicate HDFS blocks; B: list large directories; and C: make new directories; these overload the disk, threadpool, and locks respectively. ii) latency of DataNode disk operations, iii) latency at NameNode RPC queue, iv) latency to acquire NameNode "NameSystem" lock.

In production, a Hadoop job that reads many small files can stress the storage system with disk seeks, as workload A in the figure, and impact all other workloads using the disks. Similarly, a workload that repeatedly resubmits a job that fails quickly puts a large load on the NameNode, like workload C, as it has to list all the files in the job input directories. In communication with Cloudera [11], they acknowledge several instances of aggressive tenants impacting the whole cluster, saying "anything you can imagine has probably been done by a user". Interviews with service operators at Microsoft confirm this observation.

The bottleneck resource in each of these instances varies from locks, thread pool queues, to the storage and the network. While it might be tempting to design throttling and scheduling policies based only on the primary APIs and resources, our experiments show that this would be incomplete. Thus, robust resource management *requires* a comprehensive accounting of all resources that clients can potentially bottleneck on, and consideration of all possible API calls.

Resource contention is often localized Distributed systems comprise multiple processes across many machines, and different tenants contribute different load to the system. Resource contention may be localized to a subset of machines or resources. Some tenants may not be accessing these machines or resources, while other



(a) t1 is throttled starting at $t = 120$. The latency of hp is initially high, but gradually decreases as t1 is throttled. This occurs because t1 creates bottlenecks on resources also consumed by hp.

(b) t2 is throttled starting at $t = 120$. The latency of hp remains high throughout the experiment, as t2 does not cause congestion on any of the resources consumed by hp.

Figure 3.3: The latency of a high-priority tenant, hp, is dependent on resource bottlenecks caused by background tenants t1 and t2. Only t1 is responsible for the resource bottlenecks, and only throtting t1 will alleviate the bottlenecks.

tenants may be consuming more than their fair share. If a goal of the system was to reduce contention on these resources, it would be inefficient and unfair to penalize all tenants equally when only a subset may be culpable.

Figure 3.3 demonstrates the effect in HDFS on the latency of a high priority tenant, when we manually throttle the request rates of two other tenants. The figure shows a high-priority tenant, t_{hp}, sending 4MB write requests, sharing the service with two low-priority tenants. Tenant t_1 submits 8kB random reads, while tenant t_2 lists files in a directory. When we separately throttle the request rates of the background tenants, we observe an effect on the latency of t_{hp} only when throttling t_1.

In the above example, if our goal was to decrease the latency of t_{hp}, we would only benefit from reducing t_1's request. A non-trivial system should be capable of *targeting* the cause of contention - the tenants, machines, and resources responsible.

Multiple granularities of resource sharing On the one hand, concurrently executing workflows share software resources, such as threadpools and locks, within a process, while on the other hand, resources, such as the disk on Hadoop worker nodes, are distributed across the system. The disk resource, for example, is accessed by DataNode, NodeManager, and mapper/reducer processes running across all workers. Systems have many entry points (*e.g.*, HBase, HDFS, or MapReduce API) and maintenance tasks are launched from inside the system. Finally, enforcing resource usage for long-running requests requires throttling inside the system, not just at the entry points.

Resource demands are hard to predict Many schedulers [138, 144, 256] need the *cost* of a request to be specified a priori, often in a multidimensional space representing the different resources. We argue that resource requirements estimated offline would be insufficient for a number of reasons.

First, the resources requested by a task could be influenced by one or more of the arguments of the API call. For example, the resource use of a write operation depends significantly on the size of the block and number of blocks in a file. While these parameters might be enough to estimate the number of bytes sent over the network and written to disk, or how many requests to the NN will be executed, the actual *rate* of resource consumption (*e.g.* the rate of transferring the bytes over the network) will depend on the overall throughput of the request and thus on the time spent in the other resources in the system.

Second, even if we could model resource requirements of some requests offline, the behavior of other API calls depends on the *state of the system*. For example, the cost of executing a read operation depends on the size of the file (which is not known when the request is submitted to the system) and on whether the data is cached in the OS/disk cache. Similarly, the cost of listing a directory will depend on the size of the directory. These costs can also change with every update to the system software.

Third, we may not know which specific machines may execute a request. For example, HDFS comprises multiple DNs, and a client has the option to read their data from multiple replicas. Given that congestion may be localized to one or more DNs, it would be impossible to predict which DN is selected.

Finally, while some operations intuitively execute a constant amount of work – such as the file rename API – and could thus be modeled using a fixed cost, their cost might change when errors are encountered. For example, when trying to rename a non-existent file, the NN returns early with an exception. A client continuously trying to rename a missing file thus imposes a significantly different load on the locks in the namenode.

Maintenance and failure recovery cause congestion Many distributed systems perform background tasks that are not directly triggered by tenant requests but compete for the same resources. For example, HDFS performs data replication after failures, asynchronous garbage collection after file deletion, and block movement for balancing DataNode load. In some cases, these background tasks can adversely affect the performance of foreground tasks. For example, HDFS-4183 [72] describes an example where a large number of files are abandoned without closing, triggering a storm of block recovery operations after the lease expiration interval one hour later, which overloads the NameNode. Guo et al. [145] describe a similar failure in Microsoft's datacenter where a background task spawned a large number of threads, overloading the servers. On the other hand, some of these tasks need to be protected *from* foreground tasks. For example, Guo et al. [145] describe a cascading failure resulting from overloaded servers not responding to heartbeats, triggering further data replication and further overload. Though background tasks are not directly triggered by tenant requests, they nonetheless have cross-cutting executions that involve multiple processes and machines, and can invoke other systems.

Resource management is nonexistent or noncomprehensive Systems like HDFS, ZooKeeper, and HBase do not contain any resource management policies. While Yarn allocates compute slots using a fair scheduler, it ignores network and disk, thus, an aggressive job can overload these resources. Interviews with service operators at Microsoft indicate that productions system often implement resource management policies that ignore important resources and use hardcoded thresholds. For example, a policy might assume that an `open()`

is 2x more expensive than `delete()`, while the actual usage varies widely based on parameters and system state, resulting in very inaccurate resource accounting. The policies are often tweaked manually, typically *after* causing performance issues or outages, or when the system or the workloads change. Writing the policies often requires intimate knowledge of the system and of the request resource profile, which may be impossible to know a priori.

3.4 Prior Approaches

When tenants compete inside a process, traditional and well-studied resource management techniques in the operating system and hypervisor are unsuitable for protecting tenants from each other due to a mismatch in the management granularity. Nonetheless, many shared distributed systems implement some variant of performance isolation, such as fair sharing [137,241,256], throttling aggressive tenants [103,242], and providing latency or throughput guarantees [122,256,265,266].

3.4.1 Multi-Resource Scheduling

Several research projects tackle multi-resource allocation, such as Cake [265], IOFlow [256], and SQLVM [195]. Cake provides isolation between low-latency and high-throughput tenants using HBase and HDFS. However, it treats HDFS as a single resource, and cannot target specific resource bottlenecks and workflows that overload these resources. IOFlow provides per-tenant guarantees for remote disk IO requests in datacenters but does not schedule other resources such as threadpools, CPU, and locks. SQLVM [195] provides isolation for CPU, disk IO, and memory for multiple relational databases deployed in a single machine, but does not deal with distributed scenarios.

In the data analytics domain, task schedulers such as Mesos [150], Yarn [264], or Sparrow [214] use an admission control approach to allocate individual tasks to machines. In these frameworks, each task passes through the scheduler before starting its execution, the scheduler can place it to an arbitrary machine in the cluster and after starting execution, the task is not scheduled any more. In typical distributed systems, requests do not pass through a single point of execution and routing of a request through the system is driven by complex internal logic. Finally, to achieve fine-grained control over resource consumption, requests have to be throttled *during* its execution, not only at the beginning. These frameworks thus do not directly apply to scheduling in general distributed systems. All of these approaches are confined to single systems, so only align with part, but not all, of the cross-cutting execution.

3.4.2 Ad-Hoc Approaches to Resource Isolation

In all cases observed, the enforcement mechanisms for high-level policies were manually implemented. For example, Cake [265] manually instruments the RPC entry points of HDFS and HBase to add queues and associates tenants based on an identifier from the HDFS RPC headers; IOFlow [256] modifies queues in key resources (*e.g.*, NIC, disk driver) on the data path; and Pisces [241] modifies the scheduling and queueing code of Membase and directly updates tenant weights at these queues.

Lack of visibility of actual resource bottlenecks leads to the ad-hoc selection of the metrics used for performance isolation. For example, Azure Storage [103] and Pisces [241] select only request rate and operation size as metrics. SQLVM [195] uses CPU, I/O, and memory as key resources. Cake [265] breaks HDFS requests into equal-sized chunks, then assumes disk as a bottleneck and uniform cost for each chunk.

Despite the potential for degraded quality of service and outages, providing intra-process fairness is challenging to the extent that Cassandra developers rejected a proposal for per-user queueing [30], responding

> *"As for multitenancy support - you aren't alone here, the request for it comes up relatively often. But I believe that it's something that should be designed from top to bottom, and it's going to be a major change encapsulating a lot of layers, and not from bottom up, via half measures."*

Twitter engineers share this sentiment in their proposal for a new database, Manhatten [232], for providing multi-tenancy and quality-of-service as first-class citizens:

> *"Supporting multi-tenancy was a key requirement from the beginning... Allowing multiple customers to use the same cluster increases the challenge of running our systems. We now must think about isolation, management of resources, capacity modeling with multiple customers, rate limiting, QoS, quotas, and more."*

Given the burden on application programmers, inevitably, many distributed systems do not provide isolation between tenants, or only utilize ad-hoc isolation mechanisms to address individual problems reported by users. For example, HDFS recently introduced priority queueing [39] to address the problem that *"any poorly written MapReduce job is a potential distributed denial-of-service attack,"* but this only provides coarse-grained throttling of aggressive users over long periods of time. CloudStack addressed denial-of-service attacks in release 4.1, adding manually configurable upper bounds for tenant request rates [32]. A recent HBase update [43] introduced rate limiting for operators to throttle aggressive users, but it relies on hard-coded thresholds, manual partitioning of request types, and lacks cost-based scheduling. In these examples, the developers identify multi-tenant fairness and isolation as an important, but difficult, and as-yet unsolved problem [30,88,232], describing the features as *"down payments on what will become more robust functionality in future releases."* [100]. These challenges illustrate how resource management is orthogonal to any individual system.

3.4.3 Other Dimensions of Resource Management

Banga and Druschel addressed the mismatch between OS abstractions and the needs of resource accounting with resource containers [95], which, albeit in a single machine, aggregate resource usage orthogonally to processes, threads, or users. Causeway [107] modifies Linux to propagate generic metadata when threads communicate, and uses this to build meta-applications that could include resource accounting. Whodunit [106] uses causal propagation to record timings between parts of a program, and provides a profile of where requests spent their time. Timecard [225] also propagates cumulative time information in the request path between a mobile web client and a server, and uses this in real time to speed up the processing of requests that are late; however Timecard is restricted to synchronous request-response applications.

Chapter 4

Retro: A Cross-Cutting Tool for Resource Management

Based on the challenges and principles outlined in Chapter 3, we argue that an efficient resource management tool will require detailed and timely tracking of the resources used by each tenant, along the entire cross-cutting execution path of requests. This motivates the design of Retro, a second-generation cross-cutting tool for resource tracking and throttling in distributed systems. In this chapter we describe the design and implementation of Retro, and demonstrate the feasibility of Retro in a complex Hadoop deployment.

4.1 Overview

Retro is a cross-cutting tool for resource management, whose core principle is to separate resource management *policies* from the *mechanisms* required to implement them. Retro enables system designers to state, verify, tune, and maintain management policies independent of the underlying system implementation. As in software defined networking, Retro policies execute in a *logically-centralized* controller with Retro mechanisms providing a global view of resource usage both within and across processes and machines. Retro captures resource measurements and enforces resource usage along the cross-cutting dimension.

The goal of Retro is to enable *targeted* policies that achieve desired performance guarantee or fairness goals by identifying and only throttling the tenants or system activities responsible for resource bottlenecks. Retro provides three abstractions to simplify the development of such policies. First, it groups all system activities – both tenant-generated requests and system-generated tasks – into individual *workflows*, which align with cross-cutting executions and form the units of resource management. Retro attributes the usage of a resource at any instant to some workflow in the system and aggregates measurements at the centralized controller. Second, Retro provides a *resource* abstraction that unifies arbitrary resources, such as physical storage, network, CPU, thread pools, and locks, enabling resource-agnostic policies. Each resource exposes two opaque performance metrics: *slowdown*, a measure of resource contention, and a per-workflow *load*, which attributes the resource usage to workflows. Finally, Retro creates *control points*, places in the system that implement resource scheduling mechanisms such as token buckets, fair schedulers, or priority queues. Each control point schedules requests locally, but is configured centrally by the policy.

Retro advocates *reactive* policies that dynamically respond to the current resource usage of workflows in the system, instead of relying on static models of future resource requirements. These policies continuously react to changes in resource bottlenecks and input workloads by making small adjustments directing the system towards a desired goal. Such a "hill climbing" approach enables policies that are robust to both changes in workload characteristics and nonlinear performance characteristics of underlying resources.

We evaluate Retro abstractions and design principles by implementing three policies: a reactive version of bottleneck resource fairness [137]; a reactive version of dominant resource fairness [138]; and a policy that enforces end-to-end latency targets for a subset of workflows. We use these policies on a Retro implementation for the Hadoop stack, comprising HDFS, Yarn, MapReduce, HBase and ZooKeeper. All three policies are concise (about 20 lines of code) and are agnostic of Hadoop internals. We experimentally demonstrate that these policies are robust and converge to desired performance goals for different types of workloads and bottlenecks.

The targeted and reactive policies of Retro rely on accurate, near real-time measurements of resource usage across all workflows and all resources in the system. Through a careful design of mostly-automatic instrumentation and aggregation of resource usage measurements our implementation of Retro for the Hadoop stack incurs latency and throughput overhead of 0.3% to 2%.

4.2 Design

The main goal of Retro is to enable simple, targeted, system-agnostic, and resource-agnostic resource-management polices for multi-tenant distributed systems. Based off the challenges described in Chapter 3, these policies should fundamentally deal with resources consumed by entire cross-cutting executions, and likewise apply throttling decisions uniformly for cross-cutting executions. Examples of such policies are: a) throttle aggressive tenants who are getting an unfair share of bottlenecked resources, b) shape workflows to provide end-to-end latency or throughput guarantees, or c) adjust resource allocation to either speed up or slow down certain maintenance or failure recovery tasks.

Retro addresses the resource management challenges described in §3.3 by separating the mechanisms of measurement and enforcement of resource usage from high-level, global resource management policies. It does this by using three unifying abstractions – *workflows*, *resources*, and *control points* – that enable logically centralized policies to be succinctly expressed and apply to a broad class of resources and systems.

4.2.1 Retro abstractions

Workflow Resource contention in a distributed system can be caused by a wide range of system activities. Retro treats each such activity as a first-class entity called a *workflow*. A workflow is a *set* of end-to-end requests, and for each request, includes all activity along the cross-cutting execution path. Workflows form the unit of resource measurement, attribution, and enforcement in Retro. For instance, a workflow might represent requests from the same user, various background activities (such as heartbeats, garbage collection, or data load balancing operations), or failure recovery operations (such as data replication). The aggregation of requests into a workflow is up to the system designer. For instance, one system might treat all background activities as one workflow but another might treat heartbeats as a distinct workflow from other activities, if the system designer decides to provide a different priority to heartbeats.

Each workflow has a unique workflow ID. To properly attribute resource usage to individual workflows, Retro propagates the workflow ID along the execution path of all requests, like context propagation used by end-to-end tracing tools (cf. §2.4). This causal propagation allows Retro to attribute the usage of a resource to a workflow at any point in the execution, whether within a shared process or across the network.

Resources As shown in Figure 3.2, a workflow can use different resources and contention in any one of these resources can reduce its end-to-end throughput or latency. A comprehensive resource management policy should be able to respond to contention in any resource – hardware or software – and attribute load to workflows using it. A key hypothesis of Retro is that resource management policies can and should treat all resources, from thread pools to locks to disk, uniformly under a common abstraction. Such a uniform-treatment allows one to state policies that respond to disk contention, say, in the same way as lock contention. Equally importantly, this allows gradually expanding the scope of resource-management to new resources without policy change. For instance, a storage service might start by throttling clients based on their network or disk usage. However, as the complexity of the service increases to include sophisticated metadata operations, the service can start throttling by CPU usage or lock-contention. On the other hand, the challenge in providing such a unifying abstraction is to capture the behavior of varied kinds of resources with different complex non-linear performance characteristics.

To overcome this challenge, Retro captures a resource's current *first-order* performance with two unitless metrics:

- **Slowdown** indicates how slow the resource is currently, compared to its baseline performance with no contention;
- **Load** is a per-workflow metric that determines who is responsible for the slowdown.

As a simple example, consider an abstract resource with an (unbounded) queue. Let $Q_{w,i}$ be the queueing time of the ith request from workflow w in a time interval and let $S_{w,i}$ be the time the resource takes to service that request. During this interval, the load by w is $\Sigma_i S_{w,i}$ and the slowdown is $\Sigma_{w,i} \frac{Q_{w,i}+S_{w,i}}{\Sigma_{w,i} S_{w,i}}$. Note, the denominator of the slowdown is the time taken to process the requests if the queue is empty throughout the interval.

The reactive policies in Retro allow these metrics to provide a linear approximation of the complex non-linear behavior. The policies continuously measure the resource metrics while making incremental resource allocation changes. Operating in such a feedback loop enables simple abstractions while reacting to nonlinearities in the underlying performance characteristics of the resource.

Resources in real systems are more complex than the simple queue above. Retro's goal is to hide the complexities of measuring the load and slowdown of different resources in *resource libraries* that are implemented once and reused across systems. §4.3.2 explains how our current implementation provides these abstractions for the resources of interest for our experiments.

An important implication of this abstraction is that it is not possible to query the capacity of a resource. Instead, a policy can treat a resource to have reached its capacity if the slowdown exceeds some fixed constant. Directly measuring true capacity is often not possible because of many request types supported (*e.g.* open, read, sync, etc. on a disk) and because of effects of caching or buffering, workflow demands do not compose linearly. Also, due to limping hardware [126], estimating the current operating capacity is next to impossible.

Control points To separate the low-level complexities of enforcing resource allocation throughout the distributed system, we introduce the *control point* abstraction. A control point is a point in the execution of a request where Retro can enforce the decisions of resource scheduling policies. Each control point executes locally, such as delaying requests of a workflow using a token bucket, but is configured centrally from a policy.

While a control point can be placed directly in front of a resource (such as a thread pool queue), it can more generally be located anywhere it is reasonable to sleep threads or delay requests, such as in HDFS threads sending and receiving data blocks. The location of control points should be selected by the system designer while keeping a few rules in mind. A control point should not be inserted where delaying a request can directly impact other workflows, such as when holding an exclusive lock. Conversely, some asynchronous design patterns (such as thread pools) present an opportunity to interpose control points, as it is unlikely that a request will hold critical resources yet potentially block for a long period of time.

Each logical control point has one or more instances. A point with a single instance is centralized, such as a point in front of the RPC queue in HDFS NameNode. Distributed points, such as in the DataNode or its clients, have many, potentially thousands of instances. Each instance measures the current, per-workflow throughput which is aggregated inside the controller.

To achieve fine-grained control, a request has to periodically pass through control points, otherwise, it could consume unbounded amount of resources. To illustrate this, consider a request in HBase that scans a large region, reading data from multiple store files in HDFS. If Retro only throttles the request at the RegionServer RPC queue, a policy has only one chance to stop the request; once it enters HBase, it can read an unbounded amount of data from HDFS and perform computationally expensive filters on the data server-side. By adding a point to the DataNode block sender, we can control the workflow at the granularity of 64kB HDFS data packets. More generally, the longer the period of time a request can execute without passing through a control point, the longer it will take any policy to react. This is similar to the dependence between the longest packet length L_{max} and the fairness guarantees provided by packet schedulers [216, 251].

4.2.2 Architecture

Figure 4.1 outlines the high-level architecture of Retro and its three main components. First, Retro has a measurement infrastructure that provides near-real-time resource usage information across all system resources and components, segmented by workload. Second, the logically centralized controller uses the resource library to translate raw measurements to the load and slowdown metrics, and provides them as input

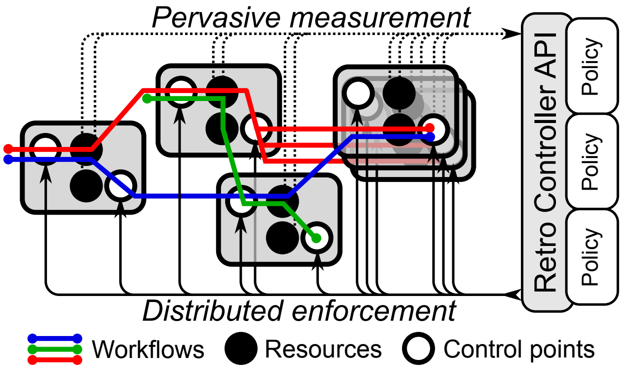


Figure 4.1: Retro architecture. Gray boxes are system components on the same or different machines. Workflows start at several points and reach multiple components. Intercepted resources (●) generate measurements that serve as inputs to policies. Policy decisions are enforced by control points (○).

to Retro policies. Third, Retro has a distributed, coordinated enforcement mechanism that consistently applies the decisions of the policies to control points in the system. We discuss the design of the controller in the following paragraphs. In §4.3 we describe the measurement and enforcement mechanisms in detail, and in §4.4 we present the implementation of three policies.

Logically centralized policies In current systems, resource management policies are hard-coded into the system implementation making it difficult to maintain as the system and policies evolve. A key design principle behind Retro is to separate the mechanisms (§4.3) from the policies (§4.4). Apart from making such policies easier to maintain, such a separation allows policies to be reused across different systems or extended with more resources.

Borrowing from the design of Software Defined Networks and IOFlow [256], Retro takes the separation a step further by logically centralizing its policies. This makes policies much easier to write and understand, as one does not have to worry about myopic local policies making conflicting decisions. In this light, we can view Retro as building a "control plane" for distributed systems, and providing a separation of concerns for policy writers and system developers and instrumenters.

Retro exposes to policies a simple API, shown in Table 4.1, that abstracts the complexity of individual resources and allows one to specify resource-agnostic scheduling policies, as demonstrated in §4.4. The first three functions in the table correspond to the three abstractions explained above. In addition, `latency(r,w)` returns the total time workflow `w` spent using resource `r`. `throughput(p,w)` measures the aggregate request rate of workflow `w` through a (potentially distributed) throttling point `p`, such as the entry point to the RS process. Finally, policies can affect the system through Retro's throttling mechanisms.

`workflows()`	list of workflows
`resources()`	list of resources
`points()`	list of throttling points
`load(r,w)`	load on `r` by workflow `w`
`slowdown(r)`	slowdown of resource `r`
`latency(r,w)`	total latency spent by `w` on `r`
`throughput(p,w)`	throughput of workflow `w` at point `p`
`get_rate(p,w)`	get the throttling rate of workflow `w` at point `p`
`set_rate(p,w,v)`	throttle workflow `w` at point `p` to `v`

Table 4.1: Retro API used by the scheduling policies. We omit auxiliary calls to set, for example, the reporting interval and smoothing parameters, as well as to obtain more details such as operation counts, etc.

4.3 Implementation

4.3.1 Per-workflow resource measurement

End-to-end ID propagation At the beginning of a request, Retro associates threads executing the request with the workflow by storing its ID in a thread local variable; when execution completes, Retro removes this association. While the developer has to manually propagate the workflow ID across RPCs or in batch operations, we use AspectJ to automatically propagate the workflow ID when using `Runnable`, `Callable`, `Thread`, or a `Queue`. This instrumentation effort is equivalent to that required by any other cross-cutting tool, as the execution boundaries at which to propagate contexts are invariant to the specific tool being deployed.

Aggregation and reporting When a resource is intercepted, Retro determines the workflow associated with the current thread, and increments in-memory counters that track the per-workflow resource use. These counters include the number of resource operations started and ended, total latency spent executing in the resource and any operation-specific statistics such as bytes read or queue time. When the workflow ID is not available, such as when parsing an RPC message from the network, the resource use is attributed to the *next* ID that is set on the current thread (*e.g.*, after extracting the workflow ID from the RPC message). Retro does not log or transmit individual trace events like the first-generation cross-cutting tools described in §2.4, but only aggregates counters in memory. A separate thread reads and reports the values of the counters to the central controller at a fixed interval, currently once per second. Reports are serialized using protocol buffers [263] and sent using ZeroMQ [4] pub-sub. The centralized controller aggregates reports by workflow ID and resource, smoothes out the values using exponential running average, and uses the resource library to compute resource load and slowdown.

Batching In some circumstances, a system might batch the requests of multiple workflows into a single request. HDFS NameNode, HBase RegionServers, and ZooKeeper each have a shared transaction log on the critical path of write requests. In these cases, we create a *batch workflow ID* to aggregate resource consumption of the batch task (e.g., the resources consumed when writing HBase transaction logs to HDFS). Constituent workflows report their relative contributions to the batch (e.g., serialized size of transaction) and the controller decomposes the resources consumed by the batch to the contributing workflows.

Automatic resource instrumentation using AspectJ Retro uses AspectJ [164] to automatically instrument all hardware resources and resources exposed through the Java standard library. Disk and network consumption is captured by intercepting constructor and method calls on file and network streams. CPU consumption is tracked during the time a thread is associated with a workflow. Locking is instrumented for all Java monitor locks and all implementers of the `Lock` interface, while thread pools are instrumented using Java's `Executors` framework. The only manual instrumentation required is for *application-level* resources created by the developer, such as custom queues, thread pools, or pipeline processing stages.

AspectJ is highly optimized and *weaves* the instrumentation with the source code when necessary without additional overheads. In order to avoid potentially expensive runtime checks to resolve virtual function calls, Retro instrumentation only intercepts constructors to return proxy objects that have instrumentation in place.

4.3.2 Resource library

Retro presents a unified framework that incorporates individual models for each type of resource. Management policies only make incremental changes to request rates allocated to individual workflows; for example, if the CPU is overloaded, a policy might reduce total load on the CPU by 5%. Therefore, as long as we correctly detect contention on a resource, iteratively reducing load on that resource will reduce the contention. Our models, thus, capture only the first-order impact of load on resource slowdown.

CPU We query the per-thread CPU cycle counter when setting and unsetting the workflow ID on a thread (using `QueryThreadCycleTime` in Windows and `clock_gettime` in Linux) to count the total number of CPU cycles spent by each workflow. The load of a workflow is thus proportional to its usage of CPU cycles. To estimate the slowdown, we divide the actual latency spent using CPU by the *optimal latency* of executing this many cycles at the CPU frequency. That is, on a core with a frequency of F cycles per second, a workflow that consumes f cycles in t seconds has slowdown:

$$\text{slowdown} = \frac{F \times t}{f} \tag{4.0.1}$$

Since part of the thread execution could be spent in synchronous IO operations, we only use CPU cycles and latency spent outside of these calls to compute CPU slowdown. If frequency scaling is enabled, we could use other existing performance counters to detect CPU contention [271].

Disk To estimate disk slowdown, we use a subset of disk IO operation types that we monitor, in particular, `read`s and `sync`s. For example, given a time interval with n `sync`s and b bytes written during these operations, we use a simple disk model that assumes a single seek with duration T_s for each `sync`, followed by data transfer at full disk bandwidth B. Denoting t to be the total time spent in `sync` operations, we derive slowdown using an estimate of the the optimal total latency:

$$\begin{aligned} \text{latency} &= nT_s + \frac{b}{B} \\ \text{slowdown} &= \frac{t}{\text{latency}} \end{aligned} \tag{4.0.2}$$

To deal with disk caching, buffering, and readahead, we only count as seeks the operations that took longer than a certain threshold, *e.g.*, 5ms. We use similar logic for reads and to estimate the load of each workflow.

Network The load of a workflow on a network link is proportional to the number of bytes transferred by that workflow. We ignore data sent over the loopback interface by checking the remote address when the connection is set up. We currently do not measure the actual network latency and thus estimate the network slowdown based on its utilization by treating it as a M/M/1 queue. Thus a link with utilization u has slowdown:

$$\text{slowdown} = 1 + \frac{u}{1-u} \tag{4.0.3}$$

It is feasible to extend Retro by encoding a model of the network (topology, bandwidths, and round trip times), and network flow parameters (source, destination, number of bytes), to estimate the network flow latency with no congestion [215]. Comparing this no-congestion estimate with measured latency could be used to compute network slowdown.

Thread pool The load of a workflow on a thread pool is proportional to the total amount of time it was using threads in this pool. Since we explicitly measure queuing time t_q and service time t_s of a thread pool operation, we can directly compute the slowdown:

$$\text{slowdown} = \frac{t_q + t_s}{t_s} \tag{4.0.4}$$

Write Locks A write lock behaves similarly to a thread pool with a single thread, and we explicitly measure the queuing time of a lock operation and the time the thread was holding the lock. Slowdown is thus the total latency of lock operation (from requesting the lock until release) divided by the time actually holding the lock.

Read-Write Locks Load of a read-write lock depends on the number of read and write operations, for how long they hold the lock, and the exact lock implementation. While there has been previous work on modeling locks using queues [156, 159, 221], none of them exactly match the `ReentrantReadWriteLock` used in HDFS. Instead, we approximate the capacity or throughput of a lock, $T(f, w, r)$, in a simple benchmark using three workflow parameters: fraction of write locks f, and average duration of write and read locks w and r. See Figure 4.2 for a subset of the measured values; notice that the throughput is nonlinear and non-monotonic.

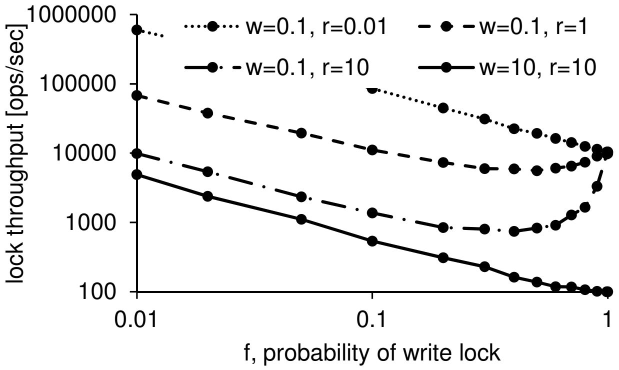


Figure 4.2: The throughput of Java ReentrantReadWriteLock (y-axis) as a function of three parameters: probability of a write lock operation (x-axis), average duration of read and write locks (see legend, time in milliseconds).

We use trilinear interpolation [158] to predict throughput for values not directly measured. Given a workflow with characteristic (f, w, r) and current lock throughput t, we estimate its load on the lock as:

$$\frac{t}{T(f,r,w)} \tag{4.0.5}$$

For example, a workflow making 1000 lock requests a second with its estimated max throughput of 5000 operations a second, would have a load of 0.2.

4.3.3 Coordinated throttling

Retro is designed to support multiple scheduling schemes, such as various queue schedulers or priority queues. In the current implementation of Retro, each control point is a *per-workflow distributed token bucket*. Threads can request tokens from the current workflow's token bucket, blocking until available. Queues can delay a request from being dequeued until sufficient tokens are available in the corresponding workflow's bucket. For a particular control point and workflow, a policy can set a rate limit R, which is then split (behind the scenes) across all point instances proportionally to the observed throughput. Retro keeps track of new control point instances coming and going – *e.g.*, mappers starting and finishing – and properly distributes the specified limit across them.

So long as each request executes a bounded amount of work, even using a single control point at the entrance to the system is enough for Retro to enforce usage of individual workflows. However, as described in §4.2.1, requests have to periodically pass through control points to guarantee fast convergence of allocation policies. Even without any control points in the system, each resource reports how many times it has been used by a particular workflow. For example, loading a single HDFS block of 64MB would result in approximately 1000 requests to the disk, each reading 64kB of data. These statistics help developers identify blocks of code where requests execute large amount of work and where adding control points helps break down execution and significantly improves convergence of control policies.

```
// identify slowest resource
S = r in resources() with max slowdown(r)
foreach w in workflows()
    demand[w] = load(S, w)
    capacity += $(1-\alpha)$*demand[w]

// determine fair allocation
fair = MaxMinFairness(capacity, demand)

// apply fair allocation
foreach w in workflows()
    if (slowdown(S) > T && fair[w] < demand[w])  // throttle
        factor = fair[w] / demand[w]
    else                                         // probe for demand
        factor = (1 + $\beta$)

    foreach p in points()
        set_rate(p, w, factor*get_rate(p, w))
```

Figure 4.3: BFair policy, see §4.4.1.

In the Hadoop stack, we added several points: in the HDFS NameNode and HBase RegionServer RPC queues, in the HDFS DataNode block sender and receiver, in the Yarn NodeManager, and in the MapReduce mappers when writing to the local disk. Each of these points has a number of instances equal to the number of processes of the particular type.

Notice that we do not need to throttle directly on resource R to enforce resource limits on R. Assume that a workflow is achieving throughput of N_p at point p and has load L_R on R. By setting a throttling rate of αN_p for all points, we will indirectly control the load on R to αL_R.

4.4 Policies

This section describes three targeted reactive resource-management policies that we used to evaluate Retro. Respectively, these policies enforce fairness on the bottleneck resource (§4.4.1), dominant-resource fairness (§4.4.2), and end-to-end latency SLOs (§4.4.3). All of these polices are system-agnostic, resource-agnostic, and can be concisely stated in a few lines of code. These are not the only policies that could be implemented on top of Retro; in fact, we believe that the Retro abstractions allow developers to write more complex policies that consider a combination of fairness and latency, together with other metrics, such as throughput, workflow priorities, or deadlines.

4.4.1 BFair policy

The BFair policy provides bottleneck fairness [137, 138]; *i.e.*, if a resource is overloaded, the policy reduces the total load on this resource while ensuring max-min fairness for workflows that use this resource. This policy can be used to throttle aggressive workflows or to provide DoS protection. It provides coarse-grained

```
1  // estimate resource demands based on measured usage
2  foreach w in workflows()
3      foreach r in resources()
4          demand[r,w] = $(1+\alpha)$*load(r,w)
5
6  // update capacity estimates based on current slowdown
7  capacities = current capacity estimates
8  foreach r in resources()
9      total_load = $\Sigma_w$load(r,w)
10     if(slowdown(r) > $T_r$)   // slowdown exceeds threshold, reduce estimate
11         capacities[r] = min(capacities[r], total_load);
12     else                      // probe for more capacity
13         capacities[r] = max((1+$\beta$)*capacities[r], total_load);
14
15 // determine fair allocation
16 share = DRF(demand, cap)
17
18 // apply fair allocation
19 foreach w in workflows()
20     if (share[w] < 1)
21         foreach p in points()
22             set_rate(p, w, share[w]*get_rate(p,w))
```

Figure 4.4: rDRF policy, see §4.4.2.

performance isolation, since workflows are guaranteed a fair-share of the bottlenecked resource.

The policy, described in Figure 4.3, first identifies the slowest resource `S` in the system according to the `slowdown` measure (Line 2). Then, the policy runs the max-min fairness algorithm with demands estimated by the current load of workflows (Line 4) and resource capacity estimated by the total demand reduced by $1-\alpha$ to relieve the bottleneck if any (Line 5).

The policy considers `S` to be bottlenecked if its slowdown is greater than a policy-specific threshold `T`. If this is the case and the fair share `fair[w]` of workflow `w` is smaller than its current load (Line 12), the policy throttles the rate by a factor of `fair[w] / demand[w]`. Here, the policy assumes a linear relationship between throughput at control points and the load on resources. If either the resource is not bottlenecked or if a workflow is not meeting its fair share (Line 14), the policy increases the throttling rate by a factor of $1+\beta$ to probe for more demand.

Notice that this policy performs *coordinated* throttling of the workflow across all the control points; by reducing the rate proportionally on each point, we quickly reduce the load of the workflow on all resources. Parameters α and β control how aggressively the policy reacts to overloaded resources and underutilized workflows respectively. Notice that this policy will throttle only if there is a bottleneck in the system; we can change the definition of a bottleneck using the parameter `T`.

4.4.2 rDRF policy

Dominant resource fairness (DRF) [138] is a multi-resource fairness algorithm with many desirable properties. The rDRF policy (Figure 4.4) calls the original DRF function at Line 16 which requires the current resource

```
// determine high-priority workflow that is missing latency target
foreach w in H
    miss(w) = latency(w) / target_latency(w)
h = w in H with max miss(w)

// compute low-priority workflow gradients
foreach l in L
    g[l] = $\Sigma_r$ (latency(h,r) * log(slowdown(r)) * load(r,l) / $\Sigma_w$ load(r,w))

// normalize gradients to use as weights
foreach l in L
    g[l] /= $\sum_k$g[k]

// throttle or relax low-priority workflows
foreach l in L
    if(miss(h) > 1)   // throttle
        factor = 1-$\alpha$*(miss(h)-1)*g[l]
    else              // relax
        factor = (1 + $\beta$)

    foreach p in points()
        set_rate(p, l, factor*get_rate(p, l))
```

Figure 4.5: LatencySLO policy, see §4.4.3.

demands and capacities of all resources. In a general distributed system, we cannot directly measure the *actual resource demand* of a workflow, but only its current load on a resource. A workflow might not be able to meet its demand due to bottlenecks in the system.

The rDRF policy overcomes this problem by being reactive: making incremental changes and reacting to how the system responds to these changes. At any instant, the policy conservatively assumes that each workflow can increase its current demand by a factor of α (Line 4). This increased allocation provides room for bottlenecked workflows to increase the load on resources.

Similarly, the policy uses the slowdown measure to estimate capacity. At Line 10, when the current slowdown exceeds a resource-specific threshold, the policy reduces its capacity estimate to the current load. On the other hand, if the slowdown is within the threshold (Line 12) and the current capacity estimate is lower than the current load, the policy increases the capacity estimate by a factor of β to probe for more capacity.

Given estimates of demand and capacity, the DRF() function returns `share[w]`, the fraction of `w`'s demand that was allocated based on dominant-resource fairness. If `share[w]`< 1, we throttle `w` at each point `p` proportionally to its current throughput at `p`.

4.4.3 LatencySLO policy

In the LatencySLO policy, we have a set of high-priority workflows H with a specified target latency SLO (service-level objective). Let L (low-priority) be the remaining workflows. The goal of the policy is to achieve the highest throughput for L, while meeting the latency targets for H. We assume the system has enough capacity to meet the SLOs for H in the absence of the workflows L; in other words, it is not necessary to throttle

H. To maximize throughput, we want to throttle workflows in L as little as possible; *e.g.*, if a workflow in L is not using an overloaded resource, it should not be throttled.

Consider a workflow h in H that is missing its target latency. If multiple such workflows exist, the policy choses the one with the maximum `miss` ratio (Line 4). Let t_w be the current request rate of workflow `w` and consider a possible change of this rate to $t_w \times f_w$. The resulting latency l_h of h is some (nonlinear) function of the relative workflow rates f_w of all workflows. The LatencySLO computes an approximate gradient of l_h with respect to f_w and uses the gradient to move the throttling rates in the right direction. Based on the system response, the policy repeats this process until all latency targets are met.

We derive an approximation of l_h which results in an intuitive throttling policy. Consider a resource r with a current slowdown of S_r, load $D_{w,r}$ for workflow w, and total load:

$$D_r = \sum_w D_{w,r} \tag{4.0.6}$$

If $L_{h,r}$ is the current latency of h at r, the baseline latency is $\frac{L_{h,r}}{S_r}$ when there is no load at r, by the definition of slowdown. We model the latency of h at r, $l_{h,r}$ as an exponential function of the load d_r that satisfies the current ($d_r = D_r$) and baseline ($d_r = 0$) latencies, and obtain:

$$l_{h,r} = L_{h,r} \times S_r^{\frac{d_r}{D_r}-1} \tag{4.0.7}$$

Finally, we model the latency of h as the sum of latencies across all resources in the system:

$$l_h = \sum_r l_{h,r} \tag{4.0.8}$$

Assuming that a fractional change in a workflow's request rate results in the same fractional change in its load on the resources, we have:

$$d_r = \sum_w D_{w,r} \times f_w \tag{4.0.9}$$

The gradient of $l_{h,r}$ with respect to f_w at $d_r = D_r$ is:

$$\frac{\partial l_{h,r}}{\partial f_w} = \frac{L_{h,r} \times \log S_r \times D_{w,r}}{D_r} \tag{4.0.10}$$

This is a very intuitive result: the impact of workflow w on the latency of h is high if it has a high resource share, $\frac{D_{w,r}}{D_r}$, on a resource with high slowdown, $\log S_r$, and where workflow h spends a lot of time, $L_{h,r}$.

Figure 4.5 uses this formula for the gradient calculation (Line 8). The policy throttles workflows in L based on the normalized gradients after dampening by a factor α to ensure that the policy only takes small steps. If all workflows in H meet their latency guarantees, the policy uses this opportunity to relax the throttling by a factor β.

4.5 Evaluation

In this section we evaluate Retro in the context of the Hadoop stack. We have instrumented five open-source systems – HDFS, Yarn, MapReduce, HBase, and ZooKeeper – that are widely used in production today. We use a wide variety of workflows, which are based on real-world traces, widely-used benchmarks, and other workloads known to cause resource overload in production systems. The cross-cutting executions in these workflows traverse multiple processes, machines, and systems, consuming resources in all of them.

Our evaluation shows that Retro addresses the challenges in §3.3 when applied simultaneously to all these stack components. In particular, we show that Retro:

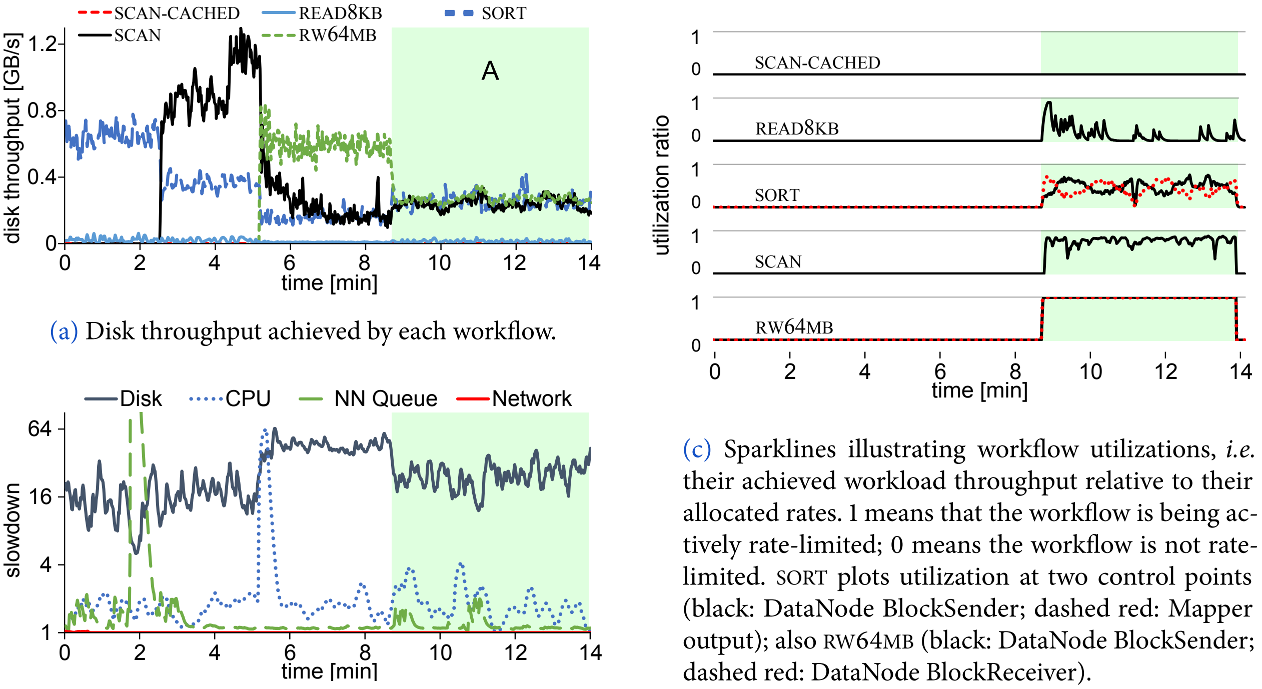
- applies coordinated throttling to achieve bottleneck and dominant resource fairness (§4.5.1 and §4.5.3);
- applies policies to application-level resources, resources shared between multiple processes, and resources with multiple instances across the cluster;
- guarantees end-to-end latency in the face of workloads contending on different resources, uniformly for client and system maintenance workflows (§4.5.2);
- is scalable and has very low developer and execution overhead (§4.5.4);
- throttles efficiently: it correctly detects bottlenecked resources and applies *targeted throttling* to the relevant workflows and control points.

We do not directly compare to other policies, since to our knowledge, no previous systems offer this rich source of per-workflow and per-resource data. Many of previous policies, such as Cake [265], could be directly implemented on top of Retro.

4.5.1 BFair in the Hadoop stack

In Figure 4.6, we demonstrate the BFair policy successfully throttling aggressive workflows without negatively affecting the throughput of other workflows. The three *major* workflows are: SORT, a MapReduce sort job; RW64MB, 100 HDFS clients reading and writing 64MB files with a 50/50 split; and SCAN, 100 HBase clients scanning large tables. These workflows bottleneck on the disk on the worker machines. The two *minor* workflows are: READ8KB, 32 clients reading 8kB files from HDFS; and SCAN-CACHED, 32 clients scanning tables in HBase that are mostly *cached* in the RegionServers. We perform this experiment on a 32-node deployment of Windows Azure virtual machines; one node runs the Retro controller, one node runs HDFS NameNode, Yarn, ZooKeeper, and HBase RegionServer, the other thirty are used as Hadoop workers. Each VM is a Standard_A4 instance with 8 cores, 14GB RAM and a 600GB data disk, connected by a 1Gbps network.

At the beginning of the experiment, we start READ8KB, SCAN-CACHED, and SORT together, and delay start of SCAN and RW64MB. Figure 4.6a shows the disk throughput achieved by each workflow; notice how the throughput changes as different workflows start, for example, throughput of SORT drops from 750MB/sec to 100MB/sec. Figure 4.6b shows the slowdown of a few different resources. Disk is the only constantly overloaded resource, reaching slowdown of up to 60. While slowdown of other resources also occasionally spikes, this happens only due to workload burstiness. In Figure 4.6c, we show sparklines of the workflow *utilization ratios*



(a) Disk throughput achieved by each workflow.

(b) Slowdown for four resources.

(c) Sparklines illustrating workflow utilizations, *i.e.* their achieved workload throughput relative to their allocated rates. 1 means that the workflow is being actively rate-limited; 0 means the workflow is not rate-limited. SORT plots utilization at two control points (black: DataNode BlockSender; dashed red: Mapper output); also RW64MB (black: DataNode BlockSender; dashed red: DataNode BlockReceiver).

Figure 4.6: BFAIR policy as described in §4.5.1. BFAIR is enabled in phase A with overload threshold T=25.

– the achieved throughput relative to the allocated rate at a particular control point. A ratio of 1 means that the workflow is being actively rate-limited; a ratio of 0 means that the workflow is never rate-limited. For SORT, we show ratios at two control points: the DN BlockSender (black, used by mapper to read data from the DN) and mapper output (dashed red, used by mapper to write its output to local disk). For RW64MB, we show ratios at two control points: the DN BlockSender (black, used to read data from HDFS DNs) and the DN BlockReceiver (dashed red, used to write data to HDFS DNs).

In phase A we enable the BFAIR with overload threshold T=25. Quickly, the disk throughput of the three major workflows equalizes at about 300MB/sec, thus achieving fairness on the bottlenecked resource. Also, the disk slowdown fluctuates at around 25 (navy blue line in the slowdown graph) because the policy starts throttling the major workflows.

The utilization ratio sparklines provide further insight. RW64MB is the most aggressive workflow and consequently it is fully throttled (ratio of 1) at all of the control points. While not as aggressive, SCAN is also throttled though less. Depending on the phase of the map-reduce computation, we throttle SORT while reading input (black) and/or when writing output (red dashed). Finally, as expected, the two minor workflows are not throttled as much, or at all, because the fairness allocates their full demand. Furthermore, SCAN-CACHED is completely unthrottled as it has no disk utilization.

These results highlight how Retro enables coordinated and targeted throttling of workloads. No other system we are aware of would achieve these results, as Retro coordinates the same resource through different control points – for example, disk is controlled not only by HDFS block transfer (used by SCAN, RW64MB,

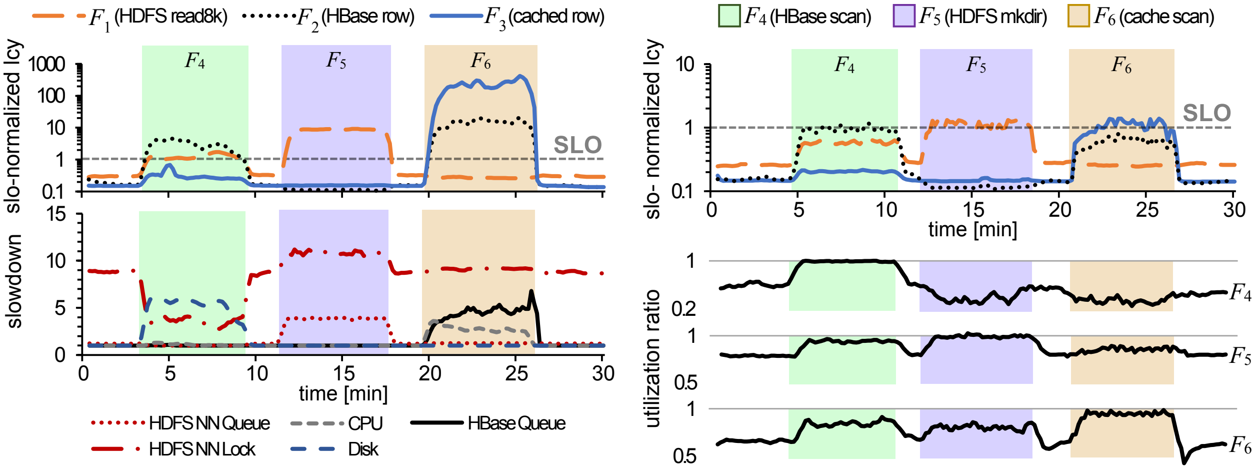


Figure 4.7: LatencySLO policy as described in §4.5.2. Top-left figure shows high priority workflow latencies without LatencySLO. Bottom-left figure shows resource slowdown during experiment. Top-right figure shows high priority workflow latencies with LatencySLO. Bottom-right sparklines show control point utilizations for background workflows.

READ8KB and the job input to SORT), but also by the SORT mapper output that accesses disk directly, bypassing HDFS. Retro only throttles the relevant workloads, leaving the small read and scan workloads mostly alone.

4.5.2 LatencySLO

We demonstrate that the LatencySLO policy can enforce a) end-to-end latency SLOs across multiple workflows and systems, and b) SLOs for both front-end clients and background tasks. We perform these experiments on an 8-node cluster; one node runs the Retro controller, one node runs HDFS NameNode, Yarn, ZK, and HBase Master, the other 6 are used as Hadoop workers and HBase RegionServers.

Enforcing multiple guarantees In this experiment we simultaneously enforce SLOs in HBase and HDFS for three high priority workflows with intermittently aggressive background workflows. The three high priority workflows are: F_1 randomly reads 8kB from HDFS with 500ms SLO, F_2 randomly reads 1 row from a small table cached by HBase with 25ms SLO, and F_3 randomly reads 1 row from a large HBase table with 250ms SLO. The background workflows are: F_4 submits 400-row HBase table scans, F_5 creates directories in HDFS, and F_6 submits 400-row HBase table scans of a cached HBase table.

Figure 4.7(top-left) demonstrates the request latencies of the three high priority workflows, normalized to their SLOs. During each of the three phases of the experiment, a background workflow temporarily increases its request rate, affecting the latency of the high priority workflows. In the first stage, F_4 increases its load and F_1 and F_2 miss their SLO. In the second stage, F_5 increases its load and F_1 misses its SLO by a factor of 10. In the last stage, F_6 increases its load and F_2 and F_3 miss their SLOs by factors of 10 and 500 respectively. Figure 4.7(bottom-left), shows the slowdown of different resources as the experiment progresses: at first F_4 table scans cause disk slowdown, then F_5 causes HDFS NameNode lock and NameNode queue slowdown, and

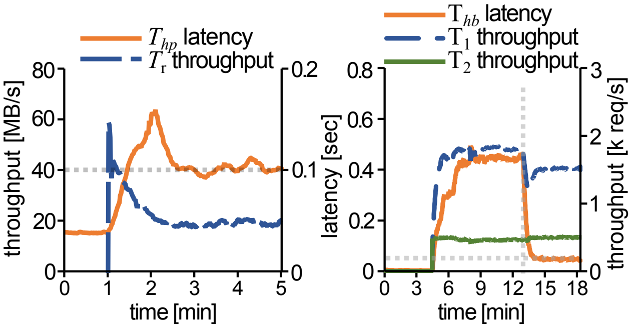


Figure 4.8: LatencySLO rate-limits replication to enforce a 100ms SLO for T_{hp} (left). LatencySLO enforces a 50ms latency for heartbeats (right).

finally F_6 causes CPU and HBase queue slowdown as its data is cached.

We repeat the experiment using LatencySLO to enforce the SLOs of F_1, F_2 and F_3. Figure 4.7(top-right) shows that the policy successfully maintains the SLOs by throttling the background workflows at a number of control points within HDFS and HBase. Figure 4.7(bottom-right) shows the sparklines of the workflow *utilization ratios* – the achieved throughput relative to the allocated rate at a particular control point, similar to Figure 4.6. We see that LatencySLO only rate-limits the background workflows during their specific overload phases.

These results highlight how LatencySLO selectively throttles workloads based on their contribution to the SLO violation. Retro can enforce SLOs for multiple workflows across software and hardware resources simultaneously.

Background workflows Thanks to the workflow abstraction, LatencySLO is equally applicable to providing guarantees for high priority background tasks, such as heartbeats, or to protecting high priority workflows from aggressive background tasks such as data replication.

Figure 4.8 (right) demonstrates the effect of two workflows T_1 and T_2 on the latency of datanode heartbeats, T_{hb}. The heartbeat latency increases from 4ms to about 450ms when T_1 and T_2 start renaming files and listing directories, respectively, causing increased load the HDFS namesystem lock. Whilst T_{hb} and T_2 only require read locks, T_1 requires write locks to update the filesystem, thus blocking heartbeats. When we start SLO enforcement at t=13, the policy identifies T_1 as the cause of slowdown, throttles it at the NameNode RPC queue, and achieves the heartbeat SLO.

Figure 4.8 (left), LatencySLO rate-limits low-priority background replication T_r, to provide guaranteed latency to high priority workflow T_{hp} submitting 8kB read requests with 100ms SLO. At t=1, we manually trigger replication of a large number of HDFS blocks; subsequently, LatencySLO rate-limits T_r. High-priority replication (single remaining replica) could use a separate workflow ID to avoid throttling.

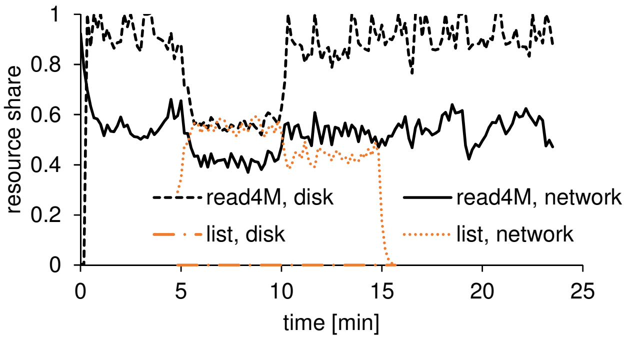


Figure 4.9: Resource share for experiment described in §4.5.3.

4.5.3 rDRF in HDFS

To demonstrate rDRF (Figure 4.9), we run an experiment with two workflows – READ4M with 50 clients reading 4MB files, and SORT with 5 clients listing 1000 files in a directory – accessing the HDFS cluster remotely sharing a 1Gbps network link. The dominant resource for READ4M is disk and for SORT it is the network, since it is reading large amounts of data from the memory of the NameNode.

We start READ4M at *t*=0 and add SORT at *t*=5, with sharing weights of 1. Between time 5 and 10, rDRF throttles READ4M to achieve equal dominant shares across both of these workflows (60% on disk and network). After increasing the weight of READ4M to 2 at *t*=10, the dominant shares change to 80% and 40%, respectively.

Despite knowing neither the demands of each workflow, nor the capacity of each resource, rDRF successfully allocates each workflow the fair share of its dominant resource. The experiment demonstrates how slowdown is viable as a proxy for resource capacity, and coupled with reactive policies, enables us to overcome some limitations of an existing resource fairness technique.

4.5.4 Overhead and scalability of Retro

Retro propagates a workflow ID (3 bytes) along the cross-cutting execution path of a request, incurring up to 80ns of overhead (see Figure 4.2) to serialize and deserialize when making network calls. The overhead to record a single resource operation is approximately 340ns, which includes intercepting the thread, recording timing, CPU cycle count (before and after the operation), and operation latency, and aggregating these into a per-workflow report.

To estimate the impact of Retro on throughput and end-to-end latency, we benchmark HDFS and HDFS instrumented with Retro using requests derived from the HDFS NNBench benchmark. See Figure 4.10 for throughput and end-to-end latency for five requests types. *Open* opens a file for reading; *Read* reads 8kB of data from a file; *Create* creates a file for writing; *Rename* renames an existing file and *Delete* deletes the file from the name system (and triggers an asynchronous block delete). Of the request types, *Read* is a DataNode operation and the others are NameNode operations. In all cases, latency increases by approximately 1-2%, and throughput drops by a similar 1-2%. Variance in latency and throughput increases slightly in HDFS instrumented with

Operation	Latency
Deserialize metadata	80ns
Read active metadata	9ns
Serialize metadata	46ns
Record use one resource operation	342ns

Table 4.2: Costs of Retro instrumentation

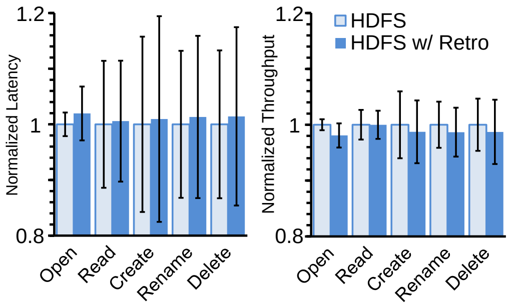


Figure 4.10: Normalized latency (left) and throughput (right) for HDFS NameNode benchmark operations along with error bars showing one standard deviation.

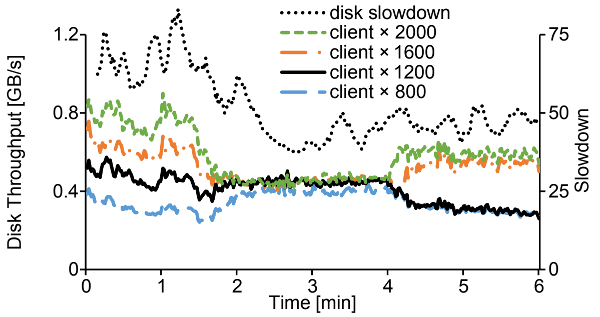


Figure 4.11: Retro's BFair policy on a 200-node cluster with four workflows and overloaded disks. BFair is enabled at t=1.5 with a target slowdown of 50; client weights are adjusted at t=4.

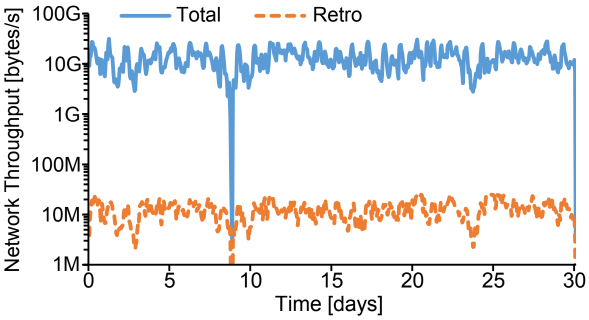


Figure 4.12: Total network throughput for a several-hundred node production Hadoop cluster and network throughput of Retro, from 1 month of traces. Retro's bandwidth requirements are on average 0.1% of the total throughput.

Retro. These overheads could be further significantly reduced by *sampling*, *i.e.*, tracing only a subset of requests or operations.

We evaluate Retro's ability to scale beyond the cluster sizes presented thus far with an 200-VM experiment on Windows Azure (Standard_A2 instances). Figure 4.11 shows slowdown and aggregate disk throughput for four workflows when BFair is activated (at t=1.5) and per-workflow weights are adjusted (at t=4). Each workflow ran a mix of 64MB HDFS reads and writes, with 800, 1200, 1600, and 2000 closed-loop clients respectively. Before the policy is activated we observe the expected imbalance in disk throughput caused by the differing number of clients in each workflow. When the policy is activated at t=1.5, the workflows quickly converge to an equal share of disk throughput, and the slowdown decreases to the target of 50. At t=4, two of the clients are given a weight of 2 and the policy quickly establishes the new fair share.

We evaluate the scalability of Retro's central controller in terms of its ability to process resource reports. In a benchmark where each report contains resource usage for 1000 workflows, the controller can process on the order of 10,000 reports per second. Assuming 10 resources per machine, the controller could thus support up to 1000 machines. In this setup, each machine would use about 600kB/sec of network bandwidth to send the reports. Figure 4.12 shows calculated network overhead that would be imposed by Retro on a production

Hadoop cluster comprising several hundred nodes over a period of a month. We calculate the network traffic that would be generated by Retro based on traces from this production cluster. The figure shows that Retro would account for an average of 0.1% of the network traffic present. Furthermore, since Retro aggregation only computes sums and averages, we can aggregate hierarchically (*e.g.* inside each machine and rack), further reducing the required network bandwidth and thereby supporting much larger deployments.

Whilst Retro requires manual developer intervention to propagate workflow IDs across network boundaries and to verify correct behavior of Retro's automatic instrumentation, our experience shows that this requires little work. For example, instrumenting each of the five systems required only between 50 and 200 lines of code; for example to handle RPC messages. Instrumenting resource operations happens automatically through AspectJ.

4.6 Discussion

Application-Level Measurement and Enforcement In Retro, we made the decision to implement both resource measurement and control points at the application level. While applying Retro in the OS, hypervisor, or device driver level could provide more accurate measurements and fine-granularity enforcement, our approach has the advantages of fast and pervasive deployment, and of not requiring specially built OS or drivers (we deployed Retro in both Windows and Linux environments). Retro's promising results indicate that OS's, and distributed systems in general, should provide mechanisms to facilitate the propagation of workflow IDs across their components.

Admission Control Retro's control points can throttle requests mid-execution. Initially this may seem counter-intuitive, as it increases the latency of executions artificially, and requests may have exclusive access to resources such as allocated memory. However, throttling requests mid-execution is necessary to handle arbitrary-length executions. Admission control is only appropriate for short requests with a known, bounded duration. For scenarios like long-running MapReduce jobs, admission control would be too coarse-grained an enforcement technique.

Application-Level Resources Retro is extensible to handle *custom resources*. For example, ZooKeeper uses a custom request processing pipeline, which is not part of Java's standard library. We treat ZooKeeper queues as custom resources and estimate their load and slowdown.

In some scenarios, the number of resources might be large, but they might be used infrequently. For example, in systems that perform *row-level locking*, the number of locks is equal to the number of rows in a table, which would prevent us from efficiently aggregating all resources. In systems with row-level locking we cannot treat each lock/row as an individual resource because the number of resources might be unbounded. To address this, one approach would be to use custom resources: we could define each data partition as a logical resource, which would significantly reduce the number of resources in the system. An alternative approach is to use sampling methods [116,183] that identify *heavy hitters* with low overhead.

Similarly, we currently group all resource instances of the same type into one logical resource, such as disks on all DataNodes, and thus do not efficiently handle the case when a single tenant is overloading a particular DataNode. However, we do have information available to identify these hot spots and throttle the offending tenants.

Limitations The current implementation of Retro has several limitations. First, some resources cannot be automatically *revoked* once a request has obtained them and have to be explicitly released by the system. For example, this applies to memory, sockets, or disk space. A developer could implement application-specific hooks that Retro could use to reclaim resources. Second, because the rates of distributed token buckets are updated only once a second, when workload is very variable, this might reduce the throughput of the system. Using different local schedulers, such as weighted fair queues [240] and reservations [144] would alleviate this problem.

4.7 Conclusion

Retro is a cross-cutting tool for implementing resource management policies in multi-tenant distributed systems. Retro tackles important challenges and provides useful abstractions that enable a separation between resource-management policies and mechanisms. In order to track the resources consumed by cross-cutting executions, Retro propagates a workflow ID alongside executions. The workflow ID is carried with executions including across process and machine boundaries. This enables Retro to attribute resource consumption to executions, and to make throttling decisions at the granularity of executions. Cross-cutting executions are an appealing dimension for resource management in distributed systems, because the overall performance experienced by a tenant is dependent upon all resources consumed along this dimension.

Writing policies requires low developer effort, and Retro is lightweight enough to be run in production. We demonstrate the applicability of Retro to key components of the Hadoop stack and develop and evaluate three targeted and reactive policies for achieving fairness and latency targets. These policies are system-agnostic, resource-agnostic, and uniformly treat all system activities, including background management tasks. To the best of our knowledge, Retro is the first framework to do so.

Chapter 5

Monitoring and Troubleshooting Distributed Systems

The second application domain we consider in this thesis is *online monitoring* in distributed systems. This chapter describes how a broad class of monitoring problems in distributed systems inherently cut across component boundaries, and need to relate information from multiple places in cross-cutting executions. In the next chapter, we will present the design and implementation of Pivot Tracing, a second-generation cross-cutting tool for dynamic monitoring. To motivate Pivot Tracing, this chapter describes two important limitations of tools today that make it difficult to monitor and troubleshoot cross-cutting executions.

5.1 Limitations of Current Approaches

Monitoring and troubleshooting distributed systems is notoriously difficult. The potential problems are myriad: hardware and software failures, misconfigurations, hot spots, aggressive tenants, or even simply unrealistic user expectations. Despite the complex, varied, and unpredictable nature of these problems, 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.

5.1.1 One Size Does Not Fit All

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 [102] discussed the important need to *"close the loop of operations back to developers"*. According to Yuan *et al.* [278], 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 [38, 42, 47–49, 65, 75]; to request changes to aggregation methods [50, 64, 67]; and to request new breakdowns of existing metrics [37, 45, 46, 57–60, 63, 64, 71, 73, 77, 80, 89]. Many issues remain unresolved due to developer pushback [58, 63, 73, 75, 80] or inertia [45, 47, 48, 65, 67, 71, 77, 89]. Even simple cases of misconfiguration are frequently unreported by error logs [277].

5.1.2 Costs of Monitoring

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 [64]. The HBase user guide [68] carries the following warning for users wishing to integrate with Ganglia [185]: *"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 [222]; 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 [90].

5.1.3 Dynamic Instrumentation

Dynamic instrumentation frameworks such as Fay [131], DTrace [105], and SystemTap [220] 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 [104]. 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. This limitation is fundamental, as neither Fay nor DTrace can affect the monitored system to propagate the monitoring context across these boundaries.

5.2 Cross-Component Monitoring

5.2.1 Crossing Boundaries

In multi-tenant, multi-application stacks, the root cause and symptoms of an issue may appear in different processes, machines, and application tiers, and may be visible to different users. A user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. For example, HBASE-4145 [59] 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 [89] 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.”*

5.2.2 Causal Tracing

Prior research, described in Chapter 2, 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 [199, 222]. A variety of these mechanisms have also materialized in production systems: for example, Google’s Dapper [244], Apache’s HTrace [85], Accumulo’s Cloudtrace [12], and Twitter’s Zipkin [259]. 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 outlined in §5.1 concerning what to record.

5.3 Other Tools and Techniques

5.3.1 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 [163, 194, 200, 275] and static analysis [283] to extract better information from logs; automatic enrichment of existing log statements to ease troubleshooting [278]; end-to-end tracing systems to capture the happened-before relationship between events [135, 244]; state-monitoring systems to track system-level resources and indicate the health of a cluster [185]; and aggregation systems to collect and summarize application-level monitoring data [169, 262]. Wang *et al.* provide a comprehensive overview of datacenter troubleshooting tools in [268]. These tools suffer from the aforementioned challenges of pre-defined information.

5.3.2 Troubleshooting and Root-Cause Diagnosis

Several offline techniques have been proposed to infer execution models from logs [101, 113, 184, 283] and diagnose performance problems [93, 167, 194, 231]. End-to-end tracing frameworks exist both in academia [97, 106, 135, 227, 257] and in industry [85, 128, 244, 245, 259] and have been used for a variety of purposes, including diagnosing anomalous requests whose structure or timing deviate from the norm [3, 97, 110, 111, 212, 231]; diagnosing steady-state problems that manifest across many requests [135, 227, 231, 244, 257]; identifying slow components and functions [106, 184, 244]; and modelling workloads and resource usage [96, 97, 184, 257]. Recent work has extended these techniques to continuous profiling and analysis [146, 188–190, 284].

VScope [267] 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 [229, 244] which controls overheads, sampling risks missing important information about rare but interesting events.

Chapter 6

Pivot Tracing: A Cross-Cutting Tool for Dynamic Causal Monitoring

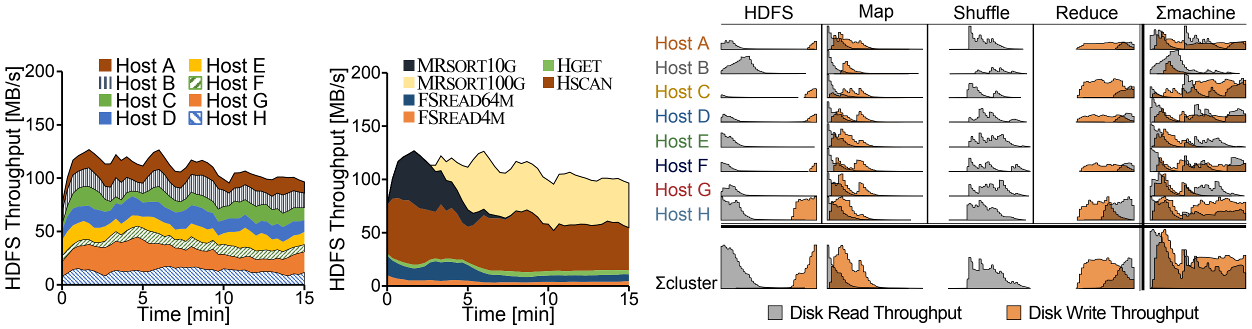
This chapter describes Pivot Tracing, a cross-cutting tool for monitoring distributed systems. Pivot Tracing addresses the challenges outlined in Chapter 5 by combining dynamic instrumentation with a novel relational operator: the happened-before join. Using the happened-before join, users of Pivot Tracing can express monitoring *queries* that relate information from multiple points on the cross-cutting execution path of requests. 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. In this chapter we describe the design, implementation, and evaluation of Pivot Tracing.

6.1 Overview

In Chapter 5 we outlined the challenges associated with monitoring and troubleshooting distributed systems. 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.

Pivot Tracing addresses these challenges and combines dynamic instrumentation with causal tracing techniques [106, 135, 244] to fundamentally increase the power and applicability of either technique. Like Fay [131], 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, $\rightarrow$ [171]. Using $\overrightarrow{\bowtie}$, queries can group and filter events based on properties of any events that causally precede them in an execution.



(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 6.1: 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.

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 cross-cutting execution path of each request. This enables inline evaluation of joins along the cross-cutting dimension, while requests execute, 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 [143], and derives its name from spreadsheets' pivot tables and pivot charts [117], 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 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.

6.1.1 Pivot Tracing in Action

In this section we motivate Pivot Tracing with a monitoring task on the Hadoop stack. The goal here is to demonstrate some of what Pivot Tracing can do; details of its design, query language, and implementation are introduced in §6.2, §6.3, and §6.4, 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. §6.5 has an overview of these components, but for now it suffices to know that HBase, a database application, accesses data through HDFS, a distributed file system. MapReduce, in addition to accessing data through HDFS, also accesses the disk directly to perform external sorts and to shuffle data between tasks. We run the following client applications:

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

By default, the systems expose a few metrics for disk consumption, such as disk read throughput aggregated by each HDFS DataNode. To reproduce this metric with Pivot Tracing, we define a *tracepoint* 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. §6.2. We then run the following query, in Pivot Tracing's LINQ-like query language [186]:

```
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 6.1a.

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 6.1b 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 6.1c demonstrates the ability for Pivot Tracing to group metrics along arbitrary dimensions. It is generated by two queries similar to `Q2` which instrument Java's `FileInputStream` and `FileOutputStream`, still joining with the client process name. We show the per-machine, per-application disk read and write throughput of MRSORT10G from the same experiment. This figure resembles a pivot table, where summing across rows yields per-machine totals, summing across columns yields per-system totals, and the bottom right corner shows the global totals. In this example, the client application presents a further dimension along which we could present statistics.

Query `Q1` above is processed locally, while query `Q2` requires the propagation of information from client processes to the data access points, with the cross-cutting execution. 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.

6.1.2 Design Summary

Figure 6.2 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 (§6.6), 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 [165], and can refer to arbitrary interface/method signature combinations. Tracepoints are defined by someone with knowledge of the system, maybe a developer or expert operator, and define the vocabulary for queries (①). They can be defined and installed at any point in time, and can be shared and disseminated.

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

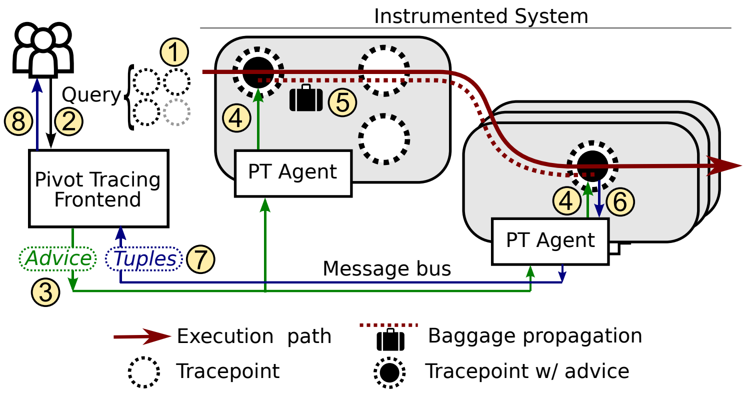


Figure 6.2: Pivot Tracing overview (§6.1.2)

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`
Rename (ρ)	Create a field using an expression	`Let SizeKB = e.Size / 1000`
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 (⋈)	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 6.1: Operations supported by the Pivot Tracing query language

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 context 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).

6.2 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

§5.1 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

6.2.1 Tracepoints

Tracepoints provide the system-level entry point for Pivot Tracing queries. A tracepoint typically corresponds to some event: a client sends a request; a low-level IO operation completes; an external RPC is invoked, 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 6.5 shows the specification of one of the tracepoints in Q2 from §6.1.1. Besides declared exports, all tracepoints export a few variables by default: host, timestamp, process id, process name, and the tracepoint definition.

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

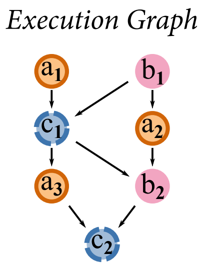
6.2.2 Query Language

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

To express queries, Pivot Tracing provides a parser for LINQ-like text queries such as those outlined in §6.1.1. Table 6.1 outlines the query operations supported by Pivot Tracing. Pivot Tracing supports several typical operations including projection (Π), selection (σ), renaming (ρ), 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 ($\overrightarrow{\bowtie}$).

6.2.3 Happened-before Joins

A key contribution of Pivot Tracing is the happened-before join query operator. Happened-before join enables the tuples from two Pivot Tracing queries to be joined based on Lamport's happened before relation, → [171]. 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 same



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

Query	*Query Results*
A	a_1, a_2, a_3
$A \overset{\rightarrow}{\bowtie} B$	$a_1 \rightarrow b_2$, $a_2 \rightarrow b_2$
$B \overset{\rightarrow}{\bowtie} C$	$b_1 \rightarrow c_1$, $b_1 \rightarrow c_2$, $b_2 \rightarrow c_2$
$(A \overset{\rightarrow}{\bowtie} B) \overset{\rightarrow}{\bowtie} C$	$a_1 \rightarrow b_2 \rightarrow c_2$, $a_2 \rightarrow b_2 \rightarrow c_2$

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

Figure 6.3: An example request execution graph and the results of running queries over that request.

cross-cutting execution.[1] 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 $b \not\rightarrow a$.

In general, events occurring during a request's execution will form a *directed, acyclic graph* (DAG) under the happened-before relation. Figure 6.3 illustrates an example DAG. Events occurring concurrently in different threads, processes, or machines, do not satisfy the happened-before relation (*e.g.*, a_1 and b_1). However, when there is communication between concurrent components, that communication will establish the happened-before relationship with later events (*e.g.*, $a_1 \rightarrow c_1$ and $b_1 \rightarrow c_1$).

The happened-before join operator enables queries about the relationships between events. For any two queries Q_1 and Q_2, the happened-before join $Q_1 \overset{\rightarrow}{\bowtie} Q_2$ produces tuples t_1t_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 6.3 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 §6.1.1 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 prior approaches in root cause analysis, including end-to-end tracing. Pivot Tracing is designed to efficiently support happened-before joins, but does not optimize more general joins such as equijoins ($\bowtie$).

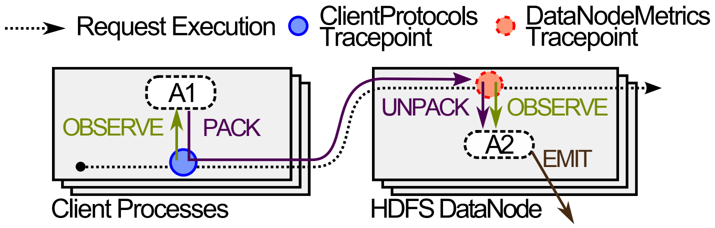
[1] This definition does not capture all possible causality, including when events in the processing of one request could influence another, but could be extended if necessary.

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 6.2: Primitive operations supported by Pivot Tracing advice for generating and aggregating tuples as defined in §6.2.

```
A1: OBSERVE procName
    PACK-FIRST procName

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



(a) Advice generated for Q2 from §6.1.1. A1 is dynamically installed at the ClientProtocols tracepoint, and A2 at the DataNodeMetrics tracepoint.

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

Figure 6.4: Pivot Tracing evaluates query Q2 from §6.1.1 by compiling it into two *advice* specifications. Pivot Tracing dynamically installs the advice at the tracepoints referenced in the query.

6.2.4 Advice

Pivot Tracing queries compile to an intermediate representation called *advice*. Advice specifies the operations to perform at each tracepoint used in a query, and eventually materializes as monitoring code installed at those tracepoints (§6.4). Advice has several operations for manipulating tuples through the tracepoint-exported variables, and evaluating $\bowtie$ on tuples produced by other advice at prior tracepoints in the execution.

Table 6.2 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.

By packing and unpacking tuples into the baggage, advice can join tuples based on the happened-before relation: if we are interested in emitting tuples only when A happened before B, a query can pack tuples at A and unpack tuples at B. Since baggage is explicitly propagated along the execution path of a request, this query directly evaluates the happened-before relation, as follows:

Theorem 1 Let e_1 be any event, and let t be a tuple observed with that event, *i.e.* $t = \textit{OBSERVE}(e_1)$. If we *PACK* t at e_1, then for all other events e_2 we get the following:

$$t \in \textit{UNPACK}(e_2) \iff e_1 \to e_2$$

Proof. Suppose $t \in \text{UNPACK}(e_2)$. Then the baggage at e_1 was propagated to e_2. By definition, if the baggage at e_1 is propagated to e_2 then e_2 is part of the same cross-cutting execution. Therefore $e_1 \to e_2$.

Now suppose $e_1 \to e_2$ and $t \notin \text{UNPACK}(e_2)$. Then the baggage at e_1 is not the same baggage as at e_2. By definition, if e_2 is part of the same execution as e_1 then the baggage at e_1 is propagated to e_2. Therefore e_2 is not part of the same execution so $e_1 \not\to e_2$. □

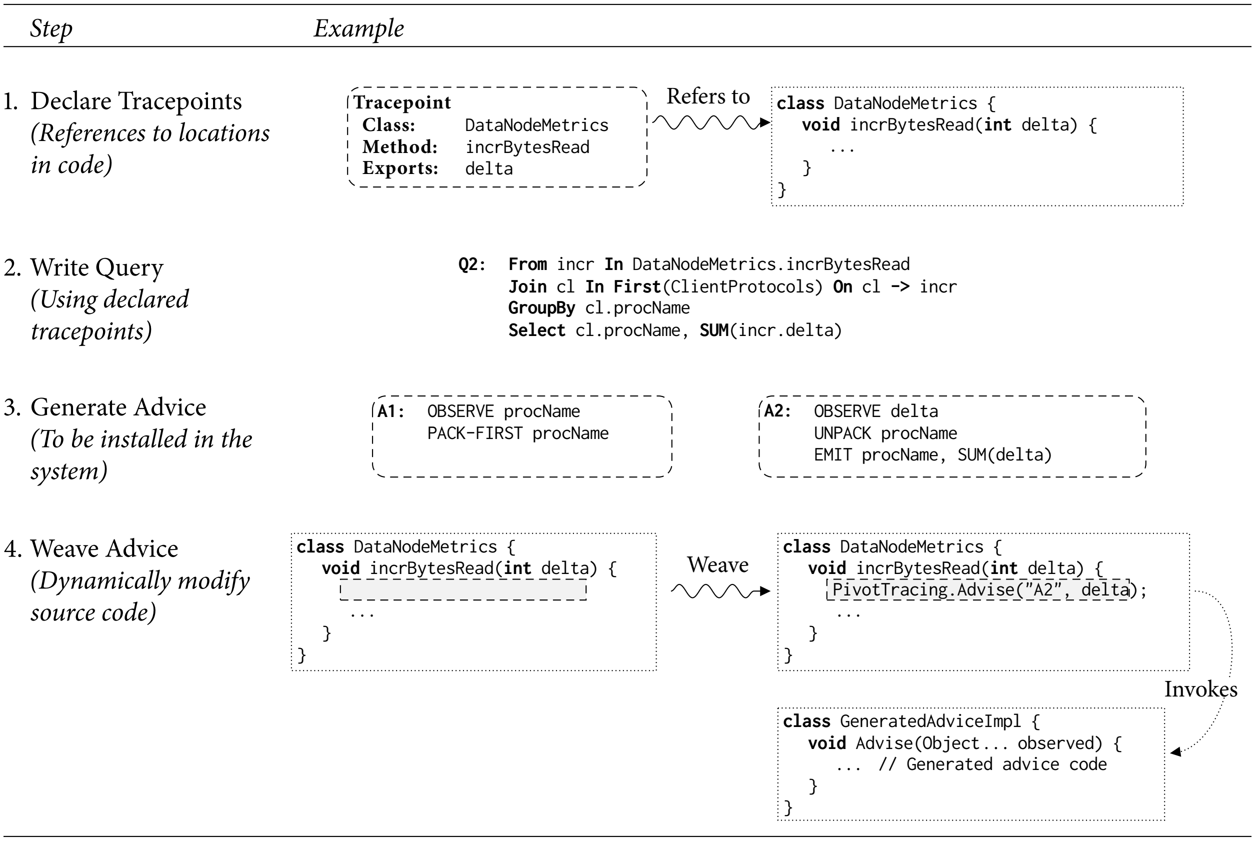
Example Figure 6.4 outlines the advice generated for query Q2 from §6.1.1, and illustrates how the advice and tracepoints interact with the execution of requests in the system. First, A1 observes and packs a single valued tuple containing the process name. Then, when execution reaches the DataNodeMetrics tracepoint, A2 unpacks the process name, observes the value of delta, then emits a joined tuple. Figure 6.4 shows how this advice and the tracepoints interact with the execution of requests in the system.

Compiling Queries to Advice To compile a query to advice, we instantiate one advice specification for a From clause and add an OBSERVE operation for the tracepoint variables used in the query. For each Join clause, we add an UNPACK operation for the variables that originate from the joined query. We recursively generate advice for the joined query, and append a PACK operation at the end of its advice for the variables that we unpacked. Where directly translates to a FILTER operation. We add an EMIT operation for the output variables of the query, restricted according to any Select clause. Aggregate, GroupBy, and GroupByAggregate are all handled by EMIT and PACK. §6.3 outlines several query rewriting optimizations for implementing $\bowtie$.

Installing Advice at Tracepoints 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 6.5 outlines this process of weaving advice for Q2.

6.3 Pivot Tracing Optimizations

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



Step
Example
1. Declare Tracepoints
(References to locations in code)
Refers to
2. Write Query
(Using declared tracepoints)
3. Generate Advice
(To be installed in the system)
4. Weave Advice
(Dynamically modify source code)
Weave
Invokes

```
Tracepoint
  Class:     DataNodeMetrics
  Method:    incrBytesRead
  Exports:   delta
```

```
class DataNodeMetrics {
  void incrBytesRead(int delta) {
    ...
  }
}
```

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

```
A1:  OBSERVE procName
     PACK-FIRST procName
```

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

```
class DataNodeMetrics {
  void incrBytesRead(int delta) {

    ...
  }
}
```

```
class DataNodeMetrics {
  void incrBytesRead(int delta) {
    PivotTracing.Advise("A2", delta);
    ...
  }
}
```

```
class GeneratedAdviceImpl {
  void Advise(Object... observed) {
    ... // Generated advice code
  }
}
```

Figure 6.5: Steps to install a Pivot Tracing query. Tracepoints are only references to locations in code, and require no system-level modifications until queries are installed. Step 4 illustrates how we *weave* advice for `Q2` at the `DataNodeMetrics` tracepoint (`Q2` also weaves advice at `ClientProtocols`, not shown). Variables exported by the tracepoint (*i.e.*, `delta`) are passed when the advice is invoked.

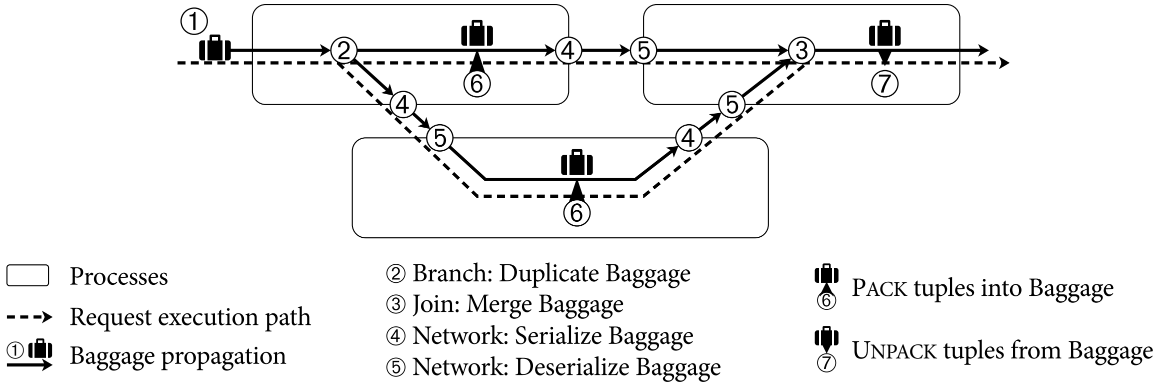

①
②
④
⑤
③
④
⑤
④
⑤
⑥
⑥
⑦
Processes
Request execution path
① Baggage propagation
② Branch: Duplicate Baggage
③ Join: Merge Baggage
④ Network: Serialize Baggage
⑤ Network: Deserialize Baggage
⑥ PACK tuples into Baggage
⑦ UNPACK tuples from Baggage

Figure 6.6: To implement the happened-before join, systems propagate baggage (①) along the execution path of requests. Baggage is duplicated when requests split into concurrent branches (②); merged when concurrent branches join (③); and included in all inter-process communication (④ ⑤). Pivot Tracing queries PACK tuples into the baggage at some tracepoints (⑥) and UNPACK tuples at other tracepoints (⑦).

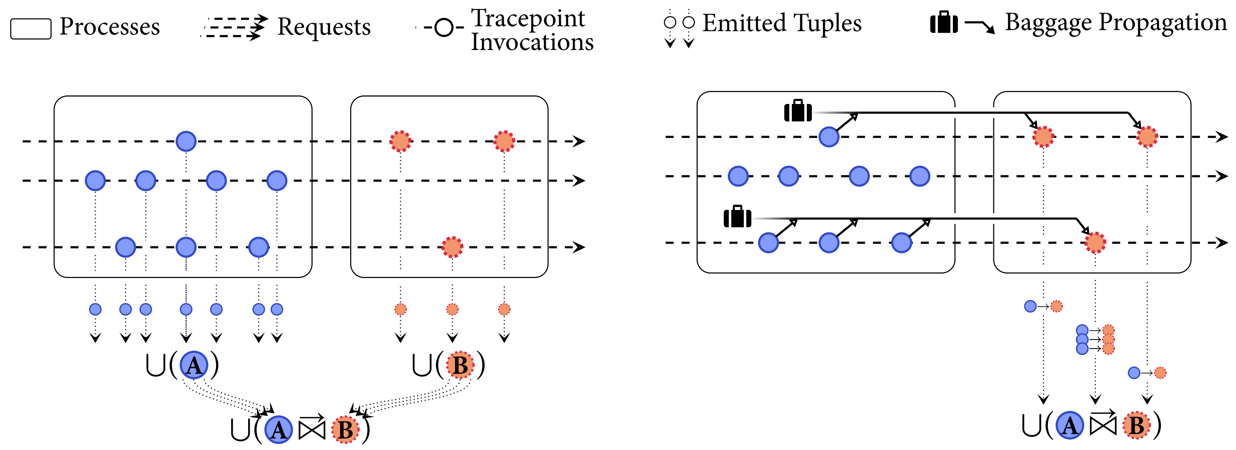
6.3.1 Baggage

The naïve evaluation strategy for happened-before join is that of an equijoin ($\bowtie$) or θ-join ($\bowtie_\theta$ [223]), requiring tuples to be aggregated globally across the cluster prior to evaluating the join. Temporal joins as implemented by Magpie [96], for example, are expensive because they implement this evaluation strategy (cf. §6.6). Figure 6.7a illustrates this approach for happened-before join.

Instead of a naïve global join, Pivot Tracing enables inexpensive happened-before joins by providing the *baggage* abstraction. Baggage is a per-request container for tuples that is propagated alongside a request as it traverses thread, application and machine boundaries. Figure 6.6 illustrates baggage propagation for a request. Each branch of the request's execution maintains its own baggage instance. To propagate baggage, instances are copied when executions split into concurrent branches; merged when concurrent branches join; and included with all of the request's inter-process and inter-thread communication. Pivot Tracing advice uses the PACK and UNPACK operations to store and retrieve tuples from the current request's baggage.

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

Baggage is a generalization of the metadata propagation used by end-to-end tracing frameworks described in Chapter 2. Using baggage, Pivot Tracing efficiently evaluates happened-before joins *in situ* during the execution of a request. Figure 6.7b shows the optimized query evaluation strategy to evaluate joins in-place during request execution.



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

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

Figure 6.7: Illustration comparing the optimized and unoptimized evaluation of Ⓐ $\overset{\rightarrow}{\bowtie}$ Ⓑ. The optimized query evaluates $\overset{\rightarrow}{\bowtie}$ inline by propagating tuples in baggage, avoiding a costly global aggregation.

6.3.2 Local Tuple Aggregation

One metric to assess the cost of a Pivot Tracing query is the number of tuples emitted for global aggregation. Although baggage carries partial query state at a per-request granularity, once tuples have been emitted for global aggregation, Pivot Tracing collects and aggregates these tuples across all requests to produce the final query results. To reduce this cost, Pivot Tracing performs intermediate aggregation for queries containing `Aggregate` or `GroupByAggregate`. Pivot Tracing aggregates the emitted tuples within each process and reports results globally at a regular interval, *e.g.*, once per second. Process-level aggregation substantially reduces traffic for emitted tuples; `Q2` from §6.1.1 is reduced from approximately 600 tuples per second to 6 tuples per second from each DataNode.

6.3.3 Optimizing Happened-Before Joins

A second cost metric for Pivot Tracing queries is the number of tuples packed during a request's execution and carried within the request's baggage. Pivot Tracing rewrites queries to minimize the number of tuples packed. Pivot Tracing pushes projection, selection, and aggregation terms as close as possible to source tracepoints. In Fay [131] 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 6.3.

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

Table 6.3: Query rewrite rules for happened-before join between two queries P and Q. Optimizations push operators as close as possible to source tuples, thereby reducing the tuples that must be propagated in baggage from P to Q. `Combine` refers to an aggregator's combiner function (*e.g.*, for `Count`, the combiner is `Sum`). See Table 6.1 for descriptions of query operators.

6.3.4 Cost of Baggage Propagation

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

In cases where too many tuples are packed in the baggage, Pivot Tracing could revert to an alternative query plan, where all tuples are emitted instead of packed, and the baggage size is kept constant by storing only enough information to reconstruct the causality, *a la* X-Trace [135], Stardust [257], or Dapper [244]. 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.

6.4 Implementation

We have implemented a Pivot Tracing prototype in Java and applied Pivot Tracing to several open-source systems from the Hadoop ecosystem. Section §6.5 outlines our instrumentation of these systems. In this section, we describe the implementation of our prototype.

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

6.4.1 Pivot Tracing Agent

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

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 6.4: 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.

6.4.2 Dynamic Instrumentation

Our prototype weaves advice at runtime, providing dynamic instrumentation similar to that of DTrace [105] and Fay [131]. Java version 1.5 onwards supports dynamic method body rewriting via the `java.lang.instrument` package. The Pivot Tracing agent programmatically rewrites and reloads class bytecode from within the process using Javassist [112].

We can define new tracepoints at runtime and dynamically weave and unweave advice. To weave advice, we rewrite method bodies to add advice invocations at the locations defined by the tracepoint (cf. Figure 6.5). Our prototype supports tracepoints at the entry, exit, or exceptional return of any method. Tracepoints can also be inserted at specific line numbers.

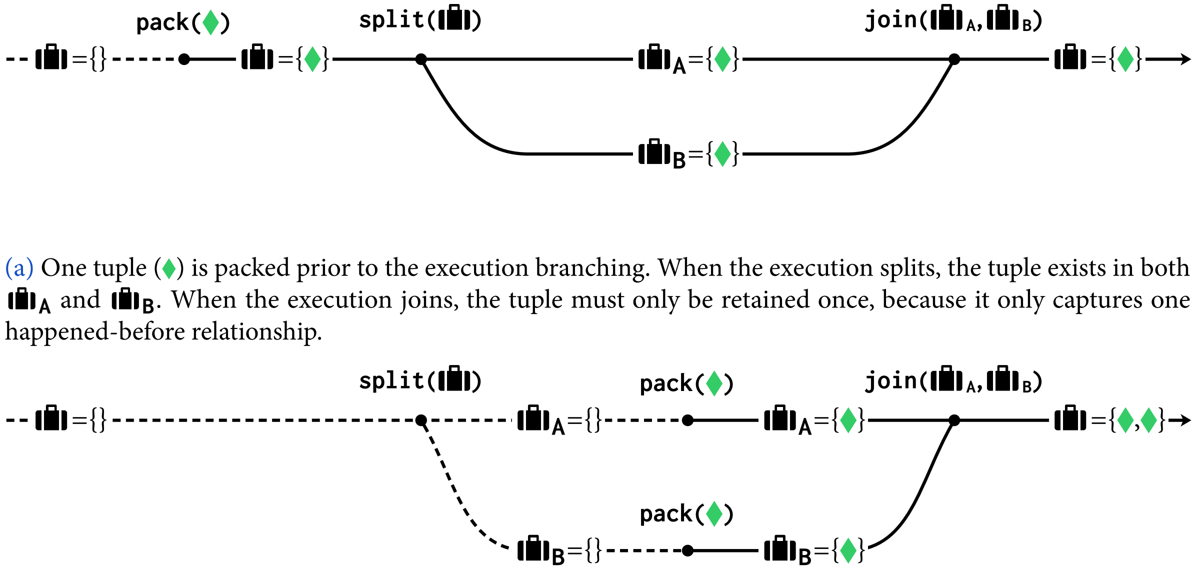
To define a tracepoint, users specify a class name, method name, method signature and weave location. Pivot Tracing also supports pattern matching, for example, all methods of an interface on a class. This feature is modeled after *pointcuts* from AspectJ [164]. Pivot Tracing supports instrumenting privileged classes (*e.g.*, `FileInputStream` in §6.1.1) 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.

6.4.3 Baggage

We provide an implementation of Baggage that stores per-request instances in thread-local variables. At the beginning of a request, we instantiate empty baggage in the thread-local variable; at the end of the request, we clear the baggage from the thread-local variable. The baggage API (Table 6.4) 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 [263]. 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



(a) One tuple (♦) is packed prior to the execution branching. When the execution splits, the tuple exists in both 💼$_A$ and 💼$_B$. When the execution joins, the tuple must only be retained once, because it only captures one happened-before relationship.

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

Figure 6.8: An execution that branches then joins. Baggage is split into 💼$_A$ and 💼$_B$, then later joined. In both examples, 💼$_A$ and 💼$_B$ contain the same tuples; however, the merge result is different in each case, because the tuples represent different happened-before relationships. Baggage must be consistently joined in order to correctly reflect the happened-before relationship represented by the tuples (cf. §6.4.5).

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 (§6.5) this entails adding a baggage field to existing application-level request contexts and RPC headers.

6.4.4 Materializing Advice

Tracepoints with woven advice invoke the `PivotTracing.Advise` method (cf. Figure 6.5), 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.

6.4.5 Baggage Consistency

In order to preserve the happened-before relation correctly within a request, Pivot Tracing must handle executions that branch and rejoin, as illustrated in Figure 6.6. Each branch of the request's execution maintains its own baggage instance. To propagate baggage, instances are copied when executions split into concurrent branches; merged when concurrent branches join; and included with all of the request's inter-process and inter-thread communication. Tuples packed by one execution branch are not visible to any other branch until the branches rejoin or communicate.

In executions that arbitrarily branch and join, we must be careful not to inadvertently duplicate tuples when merging baggage instances from concurrent execution branches. It is possible for both baggage instances to contain the same tuple derived from the same source event, as illustrated in Figure 6.8a. In this case our merge function must not naïvely duplicate the tuple – we must ensure that the output baggage only contains the tuple once, to correctly reflect that there is only one happened-before relationship. On the other hand, it is also possible for each execution branch to independently pack tuples with the same values, as illustrated in Figure 6.8b. In this case, we expect the merged baggage to contain both tuples, because they represent independent events and we must preserve both happened-before relationships.

To correctly preserve happened-before relationships for executions that arbitrarily branch and join, we implement baggage as a *set*: when a request branches, the baggage tuples are simply duplicated for each branch; when two or more branches join, we perform a *set union* on the tuples present in each baggage instance. To disambiguate between the two cases illustrated in Figure 6.8, we append a unique identifier to each tuple when it is initially packed. Consequently, when merging two baggage instances, set union will not duplicate tuples if they are the same tuple derived from the same source event; conversely, if two distinct events produced tuples with the same value, they will be differentiated by different IDs.

We extend this concept further to handle the query optimization rules described in §6.3.3, which push projection, selection, and aggregation terms as close as possible to source tuples. For some queries, these optimizations lead to advice that directly aggregates tuples in baggage, for example, by summing values immediately as tuples are packed, and propagating only the sum in baggage. However, for the same reason as illustrated in Figure 6.8, it is insufficient to only propagate a running total in baggage, as it can lead to double counting when merging baggage instances. Instead, we require some additional causality information to determine how to merge two sums.

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

Under this scheme a baggage instance resembles a version vector [218] with a variable number of components, for which bags correspond to components. In the original version of Pivot Tracing, we generated bag IDs using interval tree clocks [5], which provided a mechanism for splitting IDs when executions branch, and recombining IDs when branches rejoin. This scheme guaranteed unique bag IDs, but required maintaining

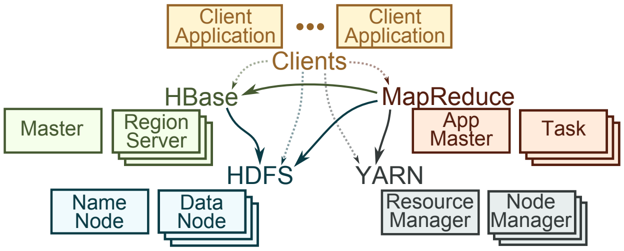


Figure 6.9: Interactions between systems. Each system comprises several processes on potentially many machines. Typical deployments often co-locate processes from several applications, *e.g.* DataNode, NodeManager, Task and RegionServer processes.

interval tree IDs even if nothing was propagated in the baggage. In our most recent Pivot Tracing implementation we have switched to random bag IDs, to eliminate the overhead of maintaining interval tree IDs. However, this now introduces a small chance of ID collisions, which can occur when two branches of the same request generate the same random ID concurrently. Our evaluation in §6.5 uses Pivot Tracing with the original interval tree clock scheme.

6.5 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 [172, 267]. Our evaluation shows that Pivot Tracing addresses the challenges in §5.1 when applied to these stack components. In particular, we show that Pivot Tracing:

- is dynamic and extensible to new kinds of analysis (§6.5.2)
- is scalable and has low developer and execution overhead (§6.5.3)
- enables cross-tier analysis between any inter-operating applications (§6.1.1, §6.5.2)
- captures event causality to successfully diagnose root causes (§6.5.1, §6.5.2)
- enables insightful analysis with even a very small number of tracepoints (§6.5.1)

Hadoop Overview We first give a high-level overview of Hadoop, before describing the necessary modifications to enable Pivot Tracing. Figure 6.9 shows the relevant components of the Hadoop stack.

HDFS [242] is a distributed file system that consists of several DataNodes that store replicated file blocks and a NameNode that manages the filesystem metadata.

HBase [40] is a non-relational database modeled after Google's Bigtable [109] that runs on top of HDFS and comprises a Master and several RegionServer processes.

Hadoop MapReduce is an implementation of the MapReduce programming model [121] 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 [264] 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 §6.4 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 [164] 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. The places in source code where we made these modifications mirror those made to propagate Retro's tenant ID (cf. §4.3.1).

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.

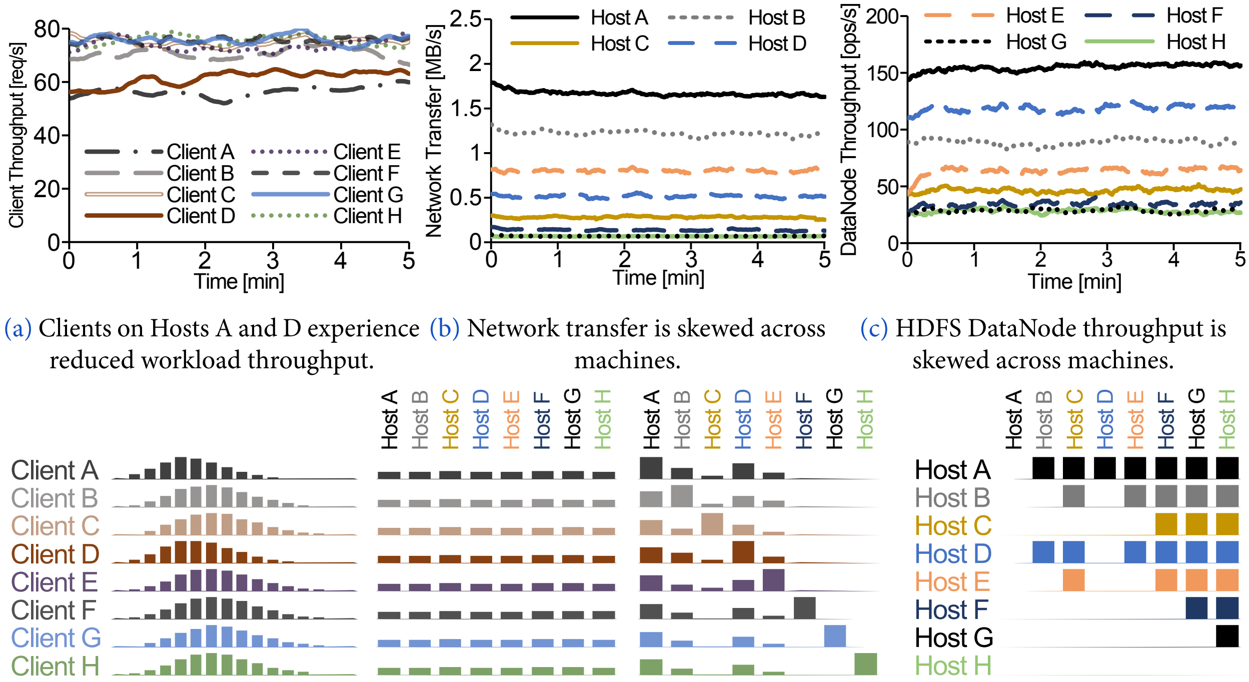
6.5.1 Case Study: HDFS Replica Selection Bug

In this section we describe our discovery of a replica selection bug in HDFS that resulted in uneven distribution of load to replicas. After identifying the bug, we found that it had been recently reported and subsequently fixed in an upcoming HDFS version [76].

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 6.10a, despite identical



(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 6.10: Pivot Tracing query results leading to our discovery of HDFS-6268 [76]. Faulty replica selection logic led clients to prioritize the replicas hosted by particular DataNodes (§6.5.1).

machine specifications and setup. We first checked machine level resource utilization on each host, which indicated substantial variation in the network throughput (Figure 6.10b). 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 DataNode.DataTransferProtocol
    GroupBy dnop.host
    Select dnop.host, COUNT
```

Figure 6.10c plots the results of this query, showing the HDFS request throughput on each DataNode. It shows that DataNodes on hosts A and D in particular have substantially higher request throughput than others – host A has on average 150 ops/sec, while host H has only 25 ops/sec. This behavior was unexpected given that our stress test clients are supposedly reading files uniformly at random. Our next query installs advice in the stress test clients and on the HDFS NameNode, to correlate each read request with the client that issued it:

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

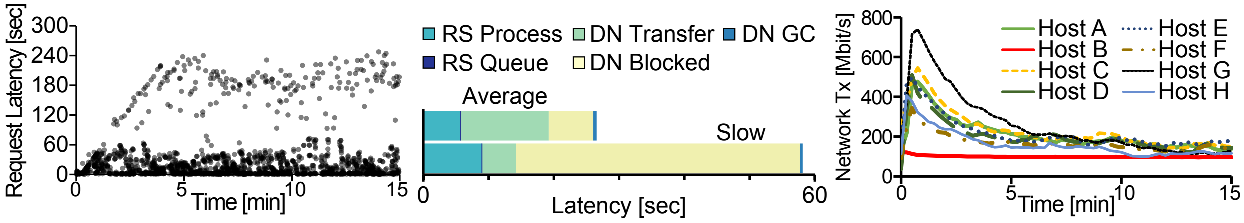
This query counts the number of times each client reads each file. In Figure 6.10d we plot the distribution of counts over a 5 minute period for clients from each host. The distributions all fit a normal distribution and indicate that all of the clients are reading files uniformly at random. The distribution of reads from clients on A and D are skewed left, consistent with their overall lower read throughput.

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

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

This query measures the frequency that each DataNode is hosting a replica for files being read. Figure 6.10e 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 6.10c clearly show that they do not. To gain more insight into this inconsistency, our next query relates the results from 6.10e and 6.10c:

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



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

Figure 6.11: (a) Observed request latencies for a closed-loop HBase workload experiencing occasional end-to-end latency spikes; (b) Average time in each component on average (top), and for slow requests (bottom, end-to-end latency > 30s); (c) Per-machine network throughput – a faulty network cable has downgraded Host B's link speed to 100Mbit, affecting entire cluster throughput.

This query measures the frequency that each client selects each DataNode for reading a replica. We plot the results in Figure 6.10f and see that the clients are clearly favoring particular DataNodes. The strong diagonal is consistent with HDFS client preference for locally-hosted replicas (39% of the time in this case). However, the expected behavior when there is not a local replica is to select a rack-local replica uniformly at random; clearly these results suggest that this was not happening.

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

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

This query correlates the DataNode that is selected with the other DataNodes also hosting a replica. We remove the interference from locally-hosted replicas by *filtering* only the requests that do a non-local read. Figure 6.10g 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 [76].

6.5.2 Diagnosing End-to-End Latency

Pivot Tracing can express queries about the time spent by a request across the components it traverses using the built-in `time` variable exported by each tracepoint. Advice can pack 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 JobCompletionEvents
    Join latencyMeasurement In Q8 On latencyMeasurement -> end
    Select job.id, AVERAGE(latencyMeasurement)
```

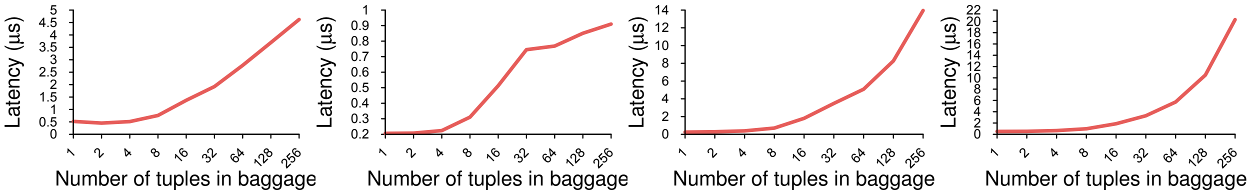
A query can measure latency in several components and propagate measurements in the baggage, reminiscent of transaction tracking in Timecard [225] and transactional profiling in Whodunit [106]. For example, during the development of Pivot Tracing we encountered an instance of network limplock [127,172], 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 6.11a). 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 6.11b). 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 6.11c).

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 [267]; and to diagnose an overloaded HDFS NameNode due to exclusive write locking as described in [180].

6.5.3 Overheads of Pivot Tracing

Baggage By default, Pivot Tracing propagates an empty baggage with a serialized size of 0 bytes. In the worst case Pivot Tracing may need to pack an unbounded number of tuples in the baggage, one for each tracepoint invoked. However, the optimizations in §6.3 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 presented in this chapter, `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 6.12 gives micro-benchmark results measuring the overhead of baggage API calls.



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

Figure 6.12: Latency micro-benchmark results for packing, unpacking, serializing, and deserializing randomly-generated 8-byte tuples.

Application-level Overhead To estimate the impact of Pivot Tracing on application-level throughput and latency, we ran benchmarks from HiBench [152], YCSB [115], and HDFS DFSIO and NNBench benchmarks. Many of these benchmarks bottleneck on network or disk and we noticed no significant performance change with Pivot Tracing enabled.

To measure the effect of Pivot Tracing on CPU bound requests, we stress tested HDFS using requests derived from the HDFS NNBench benchmark: Read8k reads 8kB from a file; Open opens a file for reading; Create creates a file for writing; Rename renames an existing file. Read8kB is a DataNode operation and the others are NameNode operations. We compared the end-to-end latency of requests in unmodified HDFS to HDFS modified in the following ways: 1) with Pivot Tracing enabled; 2) propagating baggage containing one tuple but no advice installed; 3) propagating baggage containing 60 tuples (≈1kB) but no advice installed; 4) with the advice from queries in §6.5.1 installed; 5) with the advice from queries in §6.5.2 installed.

Table 6.5 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 §6.5.1, but does trigger advice for the queries from §6.5.2. Overheads of 0.3% and 5.5% respectively reflect this difference.

Dynamic Instrumentation Some Java Virtual Machines (JVMs) only support the HotSwap feature when debugging mode is enabled, which in turn disables some compiler optimizations. For practical purposes, however, HotSpot JVM's full-speed debugging is sufficiently optimized that it is possible to run with debugging support always enabled [211]. 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.

	Benchmark			
	Read8k	Open	Create	Rename
Unmodified System	0%	0%	0%	0%
Pivot Tracing Enabled	0.3%	0.3%	<0.1%	0.2%
Baggage – 1 Tuple	0.8%	0.4%	0.6%	0.8%
Baggage – 60 Tuples	0.8%	15.9%	8.6%	4.1%
Queries – §6.5.1	1.5%	4.0%	6.0%	0.3%
Queries – §6.5.2	1.9%	14.3%	8.2%	5.5%

Table 6.5: Latency overheads for HDFS stress test with Pivot Tracing enabled, baggage propagation enabled, and full queries enabled, as described in §6.5.3

6.6 Discussion

Overheads 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.

Partial Deployment 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.

Similarly, a system that does not implement baggage can still utilize the other mechanisms of Pivot Tracing; in such a case the system resembles DTrace [105] or Fay [131]. Alternatively, kernel-level execution context propagation [107, 217, 226] can provide language-independent access to baggage variables.

Advice Safety 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 work.

Baggage Size 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. Pivot Tracing could also be extended to allow users to trade off between the performance overhead of queries and the desired results. Pivot Tracing can be easily extended to support custom aggregation, which would allow users to interchangeably select between coarse or fine grained aggregators.

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 [135], Stardust [257], or Dapper [244]. 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.

Scalability The experiments included in this thesis evaluate Pivot Tracing on an 8-node cluster. However, initial runs of the instrumented systems on a 200-node cluster with constant-size baggage being propagated showed negligible performance impact. By design, Pivot Tracing only imposes a constant amount of overhead with each request, as baggage is scoped to individual executions. As the size of a deployment grows, the baggage size propagated with each execution remains constant. The main scalability bottleneck for Pivot Tracing will eventually be its backend, which is responsible for aggregating output tuples. Sampling at the advice level is a further method of reducing overhead which we plan to investigate.

Platform and Language Independent 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.

Temporal Joins Like Fay [131], 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 [223] 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 [272]. 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 [135]. While Baggage enables Pivot Tracing to efficiently evaluate happened-before join, it is not necessary; Magpie [97] 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.

Online Analytics Processing Pivot Tracing derives its name from pivot tables and pivot charts [117] from spreadsheet programs due to their ability to dynamically select values, functions, and grouping dimensions from an underlying dataset. Pivot tables are simple versions of data cubes [143] in the online analytics processing domain. Streaming data cubes [147] perform similar real-time aggregation and optimization on incoming data streams.

Instrumentation Costs for Developers Pivot Tracing relies on developers to implement Baggage propagation when a request crosses thread, process, or asynchronous execution boundaries. In our experience (§6.5) this entails adding a baggage field to existing application-level request contexts and RPC headers.

A common criticism of systems that require causal propagation of metadata is the need to instrument the original systems [113]. We argue that the benefits outweigh the costs of doing so (§6.5), especially for new systems.

In the next chapter, we propose that a generalization of Pivot Tracing's baggage abstraction can be shared with other applications like those described in Chapter 2; then, system instrumentation would be a one-time task, reusable, and independent of any tracing system or use case, and deploying new tracing systems would be possible without having to revisit the underlying context propagation mechanism.

6.7 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 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.

Chapter 7

Developing and Deploying Cross-Cutting Tools

Building on our experiences with Retro (Chapter 4) and Pivot Tracing (Chapter 6), in this chapter we draw out similarities and differences between cross-cutting tools today. This chapter further describes the challenges faced today in actually *deploying* cross-cutting tools in production distributed systems. In the next chapter, we will argue that some of the core components of cross-cutting tools – that are responsible for these deployment challenges – can be abstracted into separate, shared components. Chapter 8 describes these abstractions in detail.

7.1 Metadata Propagation

Chapter 2 describes the challenges in monitoring and enforcing distributed system behaviors, and outlines how *cross-cutting tools* are an appealing approach to addressing these challenges. Chapter 4 and Chapter 6 demonstrated new use cases for cross-cutting tools, and illustrated how they can apply to a range of *online* monitoring and enforcement tasks. Often these tasks are tailored to specific concerns (*e.g.* resource management or metric gathering).

A key component of the cross-cutting tools considered in this thesis is *context propagation*. For example, Retro propagated a tenant ID, while Pivot Tracing propagated baggage containing tuples. Context propagation is fundamental to a broad range of distributed system monitoring, diagnosis, and debugging tasks. It means that for every execution (*e.g.* request, task, job, etc.), the system forwards a context object alongside the execution, across all process, component, and machine boundaries, with metadata about the execution. Contexts are a powerful mechanism for capturing causal relationships between events on the execution path at runtime.

7.2 Heterogeneous Data Types

Since cross-cutting tools have a diverse set of goals, correspondingly, cross-cutting tools propagate a wide range of different data types in their contexts. Depending on the tool and its goals, the data types in a context can vary widely. Many systems propagate activity and request IDs for use in debugging and profiling, anomaly detection, resource accounting, or resource management [135, 157, 180, 237, 244]. Tracing tools propagate

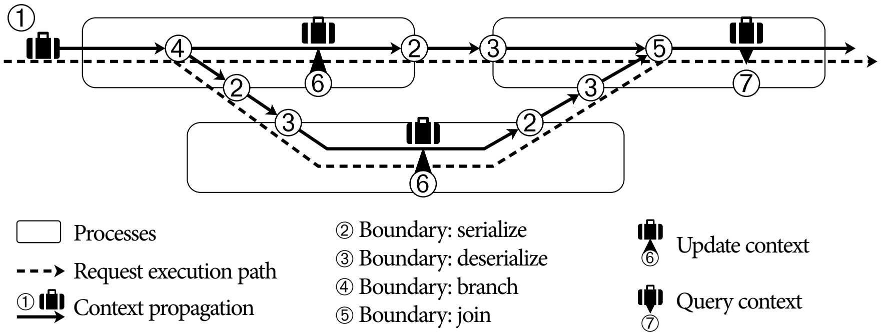


Figure 7.1: Systems propagate the cross-cutting tool's context (①) along the execution path, including across process and machine boundaries (② ③ ④ ⑤). Cross-cutting tools update (⑥) and query (⑦) context values.

sampling decisions to bias traced requests towards underrepresented request types [157]. Taint tracking and DIFC propagate security labels as the system executes, warning of or prohibiting policy violations [129,193,276]. User tokens enable tools for auditing security policies [6,237,244] and identifying business flows [237]. Data provenance systems propagate information about the lineage of data as different components manipulate it [123,192]. Tools for end-to-end latency, path profiling, and critical path analysis propagate partial latency measurements and information about execution paths and graphs [106,114,225,253]. Metric-gathering systems propagate labels and query state so that downstream components can select, group, and filter statistics [141, 182, 237]. Recent work has proposed cross-cutting tools that propagate causal histories to validate cross-system consistency [179]. Recent tools deployed in industry include propagating fault instructions for chaos engineering, and markers for capacity planning [237].

Note that some of these tools use write-once metadata, while others constantly change the metadata. Some tools only record information, while others use the metadata to take actions at runtime. In Chapter 2 we drew a distinction between first and second generation cross-cutting tools. Typically it is first-generation tools that use minimal, static metadata in order to correlate data for offline collection and analysis.

7.3 Anatomy of a Cross-Cutting Tool

Despite this proliferation of cross-cutting tools, there remain significant challenges to their development and pervasive, end-to-end deployment. In this section we detail the two main aspects of cross-cutting tools– propagation and logic – and in the next section, we describe how most of the problems with their development, evolution, and co-existence stem from the coupling between these aspects.

The cross-cutting tools we consider here have two largely orthogonal components: the system instrumentation for propagating contexts alongside executions, and the cross-cutting tool logic. Figure 7.1 illustrates these components' interaction with each other and with the instrumented system. We refer to the numbers in the figure (*e.g.* ①) in our description below.

7.3.1 Context Propagation

To correlate events across different parts of the system, all system components participate by propagating a *context* (①) along the end-to-end execution path of each request (or task, job, etc.).

Contexts A context comprises variables and data structures that the cross-cutting tool uses to observe and record causality. Examples include static IDs such as Retro's integer workflow ID (cf. §4.2.1); mutable IDs and flags describing recent events, as used by tracing tools (cf. §2.4); dynamic data structures like Pivot Tracing's baggage, comprising sets of tuples (cf. §6.3.1); and more (cf. §7.2).

Instrumentation Context propagation requires small but non-trivial interventions within all distributed system components, to maintain and propagate contexts alongside executions. Contexts must be serialized for inclusion with RPCs (②, ③); stored while a thread is doing work; attached to work items when queued to thread pools; duplicated when fanning out (④); merged when fanning in (⑤); etc. All distributed system components must be instrumented to propagate contexts, otherwise an intermediary component might discard context required by a later component. §4.3.1 and §6.4.3 describe this instrumentation task for Retro and Pivot Tracing respectively.

Propagation Rules Contexts need a small set of *propagation rules* to define behavior at execution boundaries. These include rules for serializing and deserializing (②, ③); for duplicating contexts when executions branch (④); and for merging contexts when concurrent branches join (⑤). Whereas the points in the execution where these operations happen depend on the instrumented system and its concurrency structure, the semantics of the operations, especially merge, depends on the cross-cutting tool at hand, as we describe next.

Instrumentation Scope Early research explored doing context propagation automatically in the operating system [107, 226]. However, the operating system's lack of visibility into application-level communication channels turned out to be a significant problem for today's distributed systems [235], which comprise a wide variety of applications, languages, and platforms. The prevailing strategy, which we employed with both Retro and Pivot Tracing, is to instead intervene at the application level, *i.e.* for system developers to directly modify source code to explicitly propagate context.

Happened-Before Relation Context propagation is intrinsically related to the *happened-before* relation [171], scoped to individual executions. Formally, for events a and b occurring anywhere in the system, including across system boundaries, 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 same execution.

7.3.2 Cross-Cutting Tool Logic

Using Contexts Cross-cutting tools interact with contexts intermittently during program execution, updating (⑥) and querying (⑦) values at relevant points. Unlike propagation, which requires pervasive instrumentation, the extent of intervention here depends on the tool. For example, end-to-end tracing scatters logging statements throughout components, whereas resource accounting adds counters to only very specific resource APIs. Some tools, such as security auditing, only deploy tool logic to a subset of components; *e.g.* to set the context's user ID in the front end, and to check the user ID once the request reaches the database [6,244,279]. Most cross-cutting tools also have background components, *e.g.* to collect logging statements or aggregate counters across many executions, but these are peripheral to our concerns in this chapter.

Propagation Rules The tool logic defines what data gets propagated with the execution, and the propagation rules for the data. These are trivial for simple contexts such as write-once IDs such as Retro's tenant ID. However for dynamic contexts these rules are more elaborate. For example, if a context comprises a set of tags then merging two contexts entails a set-union [141]; maintaining a simple integer counter requires care to avoid double-counting, or dropping changes made in one concurrent branch over another; and a critical path application might need to keep the maximum time accrued between two merging branches. §6.4.5 described how Pivot Tracing incorporates a versioning scheme in order to support consistent query operations that aggregate data. In the general case we formalize contexts as state-based replicated objects with parallels to CRDTs [233], where communication between instances occurs only when their respective execution branches join.[1] We discuss this correspondence in §8.3.3.

7.4 Deployment Challenges

Developing and deploying cross-cutting tools on large distributed systems and across large organizations is challenging. This section describes three broad challenges in developing and deploying cross-cutting tools. The first challenge is the need for pervasive instrumentation to propagate contexts throughout all system components. Pervasive instrumentation is time consuming and brittle, leading organizations to do instrumentation only once, if at all. Furthermore, instrumentation is usually deeply intertwined with a specific cross-cutting tool, which greatly increases the cognitive load of deploying cross-cutting tools. This ultimately makes the cost of changing existing tools, or deploying new ones, as high as that of the initial deployment.

This section cites numerous examples from the open-source distributed tracing community. As described in Chapter 2, distributed tracing tools are the most widespread of cross-cutting tools, and due to the relative maturity of this class of tools we readily find examples to illustrate the challenges described. However, we emphasize that these challenges apply more broadly to cross-cutting tools in general, and stress the use of cross-cutting tools beyond just recording traces.

[1] Out-of-band communication would violate the happened-before relation.

7.4.1 Pervasive Instrumentation

Researchers and practitioners consistently describe instrumentation as the most time consuming and difficult part of deploying a cross-cutting tool [113, 134, 135, 162, 230, 243], from early works like X-Trace: *"capturing causal connections presented the highest variability in difficulty"* [134, 135]; to recent efforts such as OpenTracing: *"deploying a distributed tracing system requires person-years of engineer effort, monkey-patched communication packages, and countless inconsistencies across platforms"* [243]. Instrumentation is a challenge because system modifications are dispersed across a wide range of disparate source code locations, making it brittle and easy to get wrong [244].

Completeness Developers can inadvertently omit infrequently exercised code paths such as background tasks, edge-case handling, and failure recovery; or only instrument specific request types. In Cassandra [22], missing instrumentation to propagate contexts with request retries caused traces to end prematurely after the first attempt [119], and missing instrumentation for paginated requests caused traces to end after the first page of results [118]; initial efforts in HDFS [74] only instrumented I/O operations, so developers had to later revisit instrumentation for metadata operations [79] and background tasks [69].

Correctness In many cases, missing or misaligned instrumentation will prematurely discard contexts [28, 29, 52, 56, 86, 248]; on the other hand, failing to discard contexts enables them to linger and associate with the next execution, causing *e.g.* distinct traces to merge [15] or misattributed resource consumption [180, 181].

Compatibility Aligning instrumentation between a system and its dependencies is also challenging; for example, Accumulo developers blamed HTrace's wrapper library for an instrumentation bug [87], discovered it was actually their own fault [14], and introduced new bugs attempting to fix it [18, 19]. Compatibility between tracing systems is the main focus of the recent OpenTracing initiative [201].

Maintenance Persistent problems may arise intermittently after an instrumented system is deployed. Unrelated code changes can break instrumentation if they fail to propagate contexts across new or modified execution boundaries; for example, a patch to Hadoop's inter-process communication [34] omitted HTrace context in the new call headers [35]; refactored components in Cassandra [23] failed to propagate request contexts across new execution boundaries [25, 26]; and in HBase, a patch to the write-ahead log led to gaps in traces due to instrumentation omissions [41]. When a system is only partially instrumented, the overlap between instrumented and uninstrumented code paths makes revisiting the instrumentation difficult. For example, Cassandra developers extended instrumentation to background tasks and found that *"It was more involved than I thought, partly because of heisenbugs and the trace state mysteriously not propagating"* [27].

Testing Testing instrumentation for end-to-end correctness is a further challenge because it touches so many components; the result is poor test coverage, complicated integration tests, broken tests across application versions, and a struggle to ensure that instrumentation remains correct [26, 44, 53, 54, 81].

Platform Support Some systems mitigate the instrumentation burden by propagating context in shared underlying communication and concurrency layers instead of directly at the application level [244]. Fonseca *et al.* proposed alleviating X-Trace instrumentation difficulties by modifying concurrency libraries or runtime environments [135]. Dapper [244] stated this as a design goal and mostly avoided application-level instrumentation by instead modifying Google's core threading, control flow, and RPC libraries. At Facebook, instrumenting core frameworks was useful for making tracing widely available and reducing the need for every system to maintain its own instrumentation [157]. Recent work has also used source and binary rewriting techniques to automatically instrument common execution patterns [155, 180–182].

Heterogeneous Systems However, this is not always feasible; extreme heterogeneity is commonplace, as many large organizations have to deal with legacy components and decentralized, disparate development teams [230]. For example, at Facebook, there are a wide variety of custom system designs, third party code, and other issues that limit use of those core frameworks [157]. Instrumentation is typically compiled into services as they are released, so even when tracing is integrated into core frameworks, a single trace may cross through multiple services each with their own version of instrumentation. Instrumentation effort also depends on system heterogeneity: the wider the variety of languages and platforms in use, the more effort required to do instrumentation [113, 177, 230]. For example, for systems like Facebook's, which comprise many language, middleware, and execution environments, adding instrumentation retroactively is a *"Herculean"* task [113].

Heterogeneous Use Cases On the other hand, at Facebook, different tracing systems were developed to address various single- and cross-system performance scenarios [157]. Each system was specialized for a specific use case and difficult to extend to other domains. For example, backend services had RPC call-tree instrumentation that was difficult to adapt to other execution models like event loops. Browser page-load tracing focused on joining data from the client and a single server, and had difficulty scaling to cover other backend systems. Mobile applications had standalone OS-provided tracing that lacked the ability to look at aggregate production data. In each of these cases, analysis over traces was confined to fixed workflows, with limited customization for per-domain features and slow iteration time for new features or analyses. This siloed approach to tracing led to several problems. It was difficult to get cross-system insights when the tools themselves didn't cross all systems, and it meant that engineers needed to understand multiple tracing tools and when to use and switch between them. Each tracing system evolved to meet specific needs and respond to gaps in other tracing systems, but no single system could handle the broad spectrum of issues that arise in practice. The type of analysis supported by each system was also fairly rigid and changes took days to deploy.

Conclusion The result of these complications is that instrumentation is done only once, if at all. This would be acceptable if such instrumentation could be reused by different cross-cutting tools, and if it would enable

	Developers	*Concerns*	*Tasks*
(a)	Cross-Cutting Tool Developers (*e.g.* Zipkin, Pivot Tracing)	Semantics of their cross-cutting tool, the APIs for using the tool, the information to communicate between API invocations	Specify cross-cutting tool logic; implement APIs; specify data types used by context
(b)	Context Format Developers	What is the correct behavior of the context under concurrent executions *i.e.* how to correctly merge multiple contexts	Implement context data format; implement logic for serialization and merging contexts.
(c)	System Developers (*e.g.* HDFS, Spark)	Execution boundaries and concurrency *e.g.* queues, threadpools, RPCs	Instrument systems to propagate contexts alongside executions

Table 7.1: Three groups are involved in developing and deploying cross-cutting tools; each group has a different set of concerns and is involved in different tasks.

the evolution of existing tools. In practice, however, instrumentation and propagation are deeply intertwined with a specific cross-cutting tool. As we discuss next, this greatly increases the cognitive load of deploying cross-cutting tools, and makes the cost of changing existing tools, or deploying new ones, as high as that of the initial deployment.

7.4.2 Cognitive Load

Because cross-cutting tools are coupled with context propagation, to perform any one task – system instrumentation, deploying tool logic, or even designing the tool itself – is cognitively challenging, because it requires tacit understanding of all components – the system being instrumented, the semantics of the tool, and context subtleties when executions branch and join. However, in practice the people who understand and care about these aspects are usually in different groups. Table 7.1 summarizes the concerns of each developer group: developers of the system (Table 7.1c) know its execution and concurrency structure; developers of cross-cutting tools (Table 7.1a) understand what data they need for their tool; and none of them should actually care about the complexities of representing concurrent data structures (Table 7.1b).

To propagate and interact with contexts, developers must pay attention to serialization, encoding schemes, and binary formats. They must also know the libraries (*e.g.* RPCs) and concurrency structures (*e.g.* event loops, futures, queues) used by the system, to determine boundaries for propagation. In practice, because they are usually deployed together, the semantics of the cross-cutting tool greatly affect instrumentation; not only does this entail understanding tool logic at instrumentation time, but also, by specializing instrumentation, precludes reuse of the instrumentation even by similar tools. For example, in HDFS, HBase, and Phoenix, instrumentation is hard-coded to HTrace contexts and rules [62, 74, 91]. For example, Zipkin omits merge rules [202, 204], leading to difficulties instrumenting queues [273], asynchronous executions [206, 207, 209] and capturing multiple-parent causality [208, 210]. HDFS and HBase developers encountered similar problems due to HTrace lacking rules to capture multiple-parent causality [55, 70, 78, 82]. When cross-cutting tools do not require instrumentation of all boundaries, it sows confusion among developers about whether to propagate contexts across those boundaries; the most common example being RPC response instrumentation [84, 205, 249].

Developers also struggle to instrument execution patterns when they do not fit into the tool's intended model. For example, Dapper's request-response span model is ill-suited for instrumenting streams, queues [273], async [206, 207, 209], and several others [70, 78, 157, 208, 210].

Because there are no pre-existing abstractions or implementations for context propagation, it is insufficient to simply state "propagate this data structure" in a setting where executions arbitrarily branch and join. Instead, something seemingly simple like an integer can require complex propagation rules in order to behave correctly and consistently; *e.g.* if it is used as a counter, its underlying implementation will more closely resemble a version vector [218]. An example of this is Pivot Tracing; to propagate sets of tuples requires a bag versioning scheme to avoid conflicts (cf. §6.4.5).

7.4.3 Duplicated Effort

Deploying tracing tools end-to-end is challenging because it requires coherent choices and participation across all system components. However, developers of different systems and components are often isolated from one another, causing them to make incompatible or conflicting choices about which tracing tools deploy; *e.g.* at Pinterest, *"enabling and deploying instrumentation across several services is like herding cats"* [162]. Amazon required extensive, co-ordinated effort to deploy X-Ray [98] across dependent systems to support tracing in Lambda [1, 9].

Deploying Tools The coupling between tool logic and propagation also makes any changes in tools entail revisiting the instrumentation to update variables, propagation logic, and cross-cutting tool API invocations. The same is true when deploying new tools, even if semantically similar. For example, the Hadoop ecosystem [228] has three distributed databases based on BigTable's design [109] and each database implemented its own an ad-hoc tracing system: Accumulo [12] developed CloudTrace [21]; HBase [40] developed HTrace [61, 62]; and Cassandra [22] developed QueryTracing [24]. Accumulo developers wanted CloudTrace to extend to the underlying distributed file system HDFS [13]; however, HDFS developers opted for compatibility with HBase and deployed HTrace instead [74]. As a result, to get visibility of HDFS, Accumulo developers replaced CloudTrace with HTrace [20]. Migrating Accumulo to HTrace meant updating instrumentation at 471 source code locations [20]. Similarly extensive changes were required to deploy Zipkin in Cassandra [23, 31, 270] and Phoenix [92]. We also observed this in our own experiences deploying, successively, X-Trace, Retro, and Pivot Tracing.

Updating Tools Different versions of the same tracing tool suffer similar problems if they change their context format or propagation logic [17, 27, 51, 247, 250]. Developers must be careful to consider compatibility implications when updating tools, especially if they wish to support tracing during upgrades, incremental upgrades, or mixed versions. For example, hard-coded serialization logic in Cassandra could cause out-of-bounds errors during deserialization if a newer version of the application extends the context definition [27]; similarly, clients of the tracing system Sleuth would crash when encountering unexpected context values [247, 250]. Accumulo developers were hesitant to change HTrace versions due to possible Hadoop conflicts [16, 17, 51], expressing frustration with the *"continued lack of concern in the HTrace project around tracing during upgrades"* [17]. The effort needed to update a tool's version is often the same as simply migrating to a different tool; for example, systems updating HTrace versions each required hundreds of changes in dozens of files [16, 66, 82]. Accumulo developers decried the *"continued lack of concern in the HTrace project around tracing during upgrades"* [17].

Mixing Tools Some tools alleviate these issues with ad-hoc compatibility shims, so that a system instrumented for a different tool can share contexts and talk to the same backends. This approach is fragile even for tools that ostensibly perform the same task. For example, open-source Dapper derivatives preserve causal relationships at different granularities, leading to "severe signal loss" when integrating with less expressive tools [204], or requiring system-level changes to capture missing relationships required by more expressive tools [16, 55, 82].

7.5 Related Work

7.5.1 Instrumenting Systems

Early research explored automatically propagating context in the operating system [107, 226]. However, the operating system lacks visibility into application-level communication channels, which leads to inaccurate instrumentation [235] and turns out to be a significant problem for today's distributed systems, which comprise a wide variety of applications, languages, and platforms. Consequently, the prevailing strategy is to instead intervene at the application level, *i.e.* by modifying source code in applications and libraries.

Many systems push instrumentation into shared underlying communication and concurrency layers instead of relying on application-level instrumentation. Fonseca *et al.* proposed instrumenting concurrency libraries or runtime environments to avoid the instrumentation burden they experienced with X-Trace [135]. Dapper [244] instrumented Google's core threading, control flow, and RPC library code, in order to minimize or eliminate the amount of application-level instrumentation needed. Several RPC frameworks follow this approach, such as Twitter's Finagle [260], Uber's tchannel [261], and Go kit [255]. Strongloop's Node runtime automatically traces Javascript continuations, HTTP requests, and calls to database backends [252]. Similarly, Spring Cloud Sleuth [246] automatically instruments Spring applications at common ingress and egress points.

Several prior systems have used binary and source rewriting techniques to automatically instrument common execution patterns. For example, with both Retro and Pivot Tracing, we pattern-match common interfaces and execution patterns and automatically instrument them using AspectJ. Similarly, Timecard [225] automatically instrumentation Windows Phone applications to propagate metadata, using binary rewriting techniques for Silverlight on the client side, and .NET on the server side(client side) and .NET (server side) [224]. WebPerf [155] extends Timecard's automatic instrumentation to include .NET async/await style programs. Domino [176] monkey-patches Javascript web applications on both the client and server side for node.js applications. APM companies Instana and AppDynamics deploy their tracing instrumentation using Java agents, which automatically modify Java bytecode at load time [125]; Naver's Pinpoint tracing framework takes the same approach [196].

Both of these techniques – framework instrumentation and automatic instrumentation – reduce the instrumentation required, but do not eliminate it entirely. For example, Prezi achieved approximately 80% coverage by instrumenting Django's HTTP, database, and AWS calls. Retro and Pivot Tracing required approximately 50-300 additional lines of source code modification to catch instrumentation cases not handled by automatic instrumentation. Dapper authors noted 40 C++ applications and 33 Java applications (out of thousands) that required some manual trace propagation, as well as some uninstrumented applications that used non-standard communication libraries.

OpenTracing is a recent effort in the open-source tracing community to address some of the challenges surrounding end-to-end deployment and reuse for tracing frameworks. However, OpenTracing is more narrowly focused on recording traces as span trees, like Dapper, and does not capture some causal relationships [202,204]; it also includes logging and span management, which are orthogonal to context propagation and instead fall under cross-cutting tool logic [203].

An alternative class of work in understanding and troubleshooting distributed systems focuses on black-box approaches, *i.e.* log analysis [96, 282]. This avoids context propagation altogether; however, its use cases are limited to offline analysis of the information exposed by the system through logs, and it does not apply to online use cases such as scheduling and data quality trade-offs.

7.5.2 Generic Contexts

SDI propagates generic context objects alongside requests on a single machine and defines a grammar of actions that occur at execution boundaries. SDI does not address merging of contexts, but does address some questions of security and other issues [226]. Causeway [107] introduced the notion of meta-applications and propose system-level instrumentation that can be used by many meta-applications. Causeway called execution boundaries 'transfer points', and supports registering callbacks at transfer points. As with SDI, Causeway does not address merging of contexts.

Subsequent to the publication of Pivot Tracing [182], OpenTracing incorporated the notion of key-value pairs that can be propagated alongside requests, also using the term baggage. OpenTracing does not specify how to resolve conflicts between baggage values when merging contexts, and the default behavior of the current implementation will arbitrarily select one value if there are conflicting keys.

7.6 The Need for Abstractions

Many of the problems described in this chapter stem from the coupling between cross-cutting tools and instrumentation, brittle deployments, and repeated developer effort. In fact, in all cross-cutting tools to date, system instrumentation and tool logic are deeply intertwined. This tight coupling requires teams deploying cross-cutting tools to have a deep understanding of both the tool and of all instrumented systems. It also makes any modification or evolution of a cross-cutting tool, or the deployment of new cross-cutting tools, as difficult as the initial deployment.

To address these challenges, we argue that system instrumentation should be a one-time task, reusable, independent of any tracing tool, and that developing, deploying, and updating tracing tools should be possible without having to revisit or consider the underlying context propagation mechanisms. To achieve this we advocate for abstractions to separate the concerns of tracing tool developers from those of system developers, to enable reusable general-purpose context propagation. In the next chapter we describe the design and implementation of our proposed abstractions.

Chapter 8

Baggage Contexts: A Universal Abstraction for Cross-Cutting Tools

Based on the challenges outlined in Chapter 7, we argue that system instrumentation for context propagation should be decoupled from cross-cutting tool logic, with the ultimate goal of system instrumentation being a one-time task. This motivates the design of *baggage contexts*, an intermediate representation for cross-cutting tool contexts. Baggage contexts extend and generalize the baggage concept introduced by Pivot Tracing in §6.3.1. Baggage contexts enable the complete decoupline of system instrumentation from cross-cutting tool logic, and using baggage contexts, new cross-cutting tools can be deployed without effecting changes to instrumentation. To fully benefit from this decoupling, we expose baggage contexts to system developers and tool developers through different abstractions that hide irrelevant aspects from each group. We propose a layered architecture called the *Tracing Plane*, with baggage contexts serving as a narrow waist, to enforce this separation of concerns. In this chapter we describe the design and implementation of baggage contexts. We also demonstrate and evaluate the applicability of baggage contexts as a general tracing context format, and the effectiveness of the Tracing Plane abstractions built around them.

8.1 Separation of Concerns

The problems described in Chapter 7 stem from the coupling between cross-cutting tools and instrumentation, which cause brittle deployments, and repeated developer effort. To avoid this, we separate the concerns of cross-cutting tool developers from those of system developers, by providing a general-purpose intermediate representation called *baggage contexts*. Baggage contexts have two goals:

- Baggage contexts must be capable of carrying a wide range of data types that cross-cutting tools use, such as primitive types, IDs, sets, maps, counters, nested data structures, and more, while remaining efficient and extensible.
- System instrumentation that propagates baggage contexts must be completely agnostic to the interpretation of any data therein and independent of the semantics of any cross-cutting tool.

Operation	*Execution Boundary*	*Operation Description*
BRANCH	Execution splits into multiple branches (*e.g.* threads, concurrent RPCs)	Derive a context for each branch
JOIN	Concurrent execution branches join	Combine multiple contexts into one
SERIALIZE	Send a network request	Serialize context to wire format
DESERIALIZE	Receive a network request	Deserialize context from wire format
TRIM	Anywhere with size constraints	Impose a size constraint on a context

Table 8.1: Five operations for propagating contexts used to instrument systems.

At first glance these objectives appear easily satisfied by existing data formats, such as plaintext dictionaries (as with HTTP headers), or structured serialization formats (like Protocol Buffers). However, a key challenge not addressed – which baggage contexts solve – is how to maintain correctness under arbitrary concurrency; specifically, merging baggage contexts correctly, according to the semantics of any data types therein, whenever two concurrent execution branches join.

8.2 Interfaces

Before describing the design of baggage contexts, we first outline the interfaces for 1) system developers and 2) cross-cutting tool developers, which baggage contexts must bridge.

8.2.1 Interface for System Instrumentation

The main concern for system developers is instrumenting systems to propagate context objects alongside requests as they execute. Our goal for system developers is to enable reusable instrumentation that is only done once, independent of any cross-cutting tools. To this end, it has two requirements. First, that baggage contexts be opaque, which decouples the instrumentation from the details of any specific cross-cutting tool. Second, that the instrumentation be pervasive: with opaque contexts, we have no idea which causal relationships a cross-cutting tool might want to preserve; consequently, it is necessary to instrument all execution boundaries.

To support this, system developers are not exposed to the internal representation of baggage contexts. Instead, baggage contexts are opaque objects that provide a minimal set of five propagation operations necessary for capturing system concurrency patterns and boundaries: BRANCH derives new context instances for when the execution splits into concurrent branches; a commutative JOIN merges multiple context instances when concurrent execution branches join; SERIALIZE and DESERIALIZE write and read a serialized baggage representation; and TRIM imposes a size constraint on baggage, *e.g.* if a system will propagate at most 1kB of context data. Table 8.1 summarizes these operations.

```
bag Zipkin {
  fixed64 traceID = 0
  fixed64 spanID = 1
  fixed64 parentSpanID = 2
  flag    sampled = 3
}
bag XTrace {
  fixed64 TaskID = 0;
  set<fixed64> ParentIDs = 1;
}
```

```
bag Retro {
  int32 TenantID = 0;
}
bag PivotTracing {
  map<string, set<bytes>> tuples = 0;
}
bag NetJob {
  map<string,string> Labels = 0;
}
```

Figure 8.1: BDL declarations for five cross-cutting tools. See §8.5.1 for a description of these tools..

8.2.2 Interface for Cross-Cutting Tools

Orthogonal to system developers, the main concern for cross-cutting tool developers is specifying the data types used by the cross-cutting tool. §7.2 describes the variety of different data types and use cases of cross-cutting tools.

To support this heterogeneity, we expose a rich library of concurrent data types expressed through an interface definition language called BDL, or Baggage Definition Language. Figure 8.1 outlines BDL declarations for five cross-cutting tools that we revisit in our evaluation (see §8.5.1 for a description). The BDL format is similar in style and primitives to protocol buffers [263]; it also provides sets, maps, nested data structures, and several more elaborate data types based on CRDTs (cf. §8.4.6). A declaration includes a name and one or more fields, with each field specifying a type, a name, and a numbered index. BDL declarations can be updated to add new fields and deprecate existing fields, without affecting backwards compatibility with systems that deployed old versions. The only requirement is that, once deployed, new fields cannot reuse indices of existing fields.

The goal of BDL is to separate the specification of *what* data is carried in a baggage context, from the implementation details of *how* the data is encoded. From a BDL specification, a BDL compiler will generate interfaces for tools to access and manipulate data within baggage context instances. For example, after compiling a bag declaration, tool code will access data inside baggage contexts with simple method calls, *e.g.* `zmd := zipkin.readFrom(bagCtx)`. Generated BDL interfaces access baggage contexts, interpret the data therein, and construct a corresponding object representation of the data. Callers can then read, update, and manipulate values, *e.g.* `zmd.SetTraceID(55)`.

From the perspective of cross-cutting tool developers, baggage contexts provide the abstraction of "execution-flow scoped variables", which, once set, follow the cross-cutting execution across all components. To automatically merge instances when execution branches join, all BDL data types encapsulate well-defined behavior for merging instances.

8.2.3 Example Baggage Context Usage

To clarify these concepts we give a brief description of how baggage contexts might be used, with an example focusing on *who* uses them and *when*.

First, baggage context developers (*e.g.*, the author of this thesis) implement baggage context libraries. The libraries are twofold: a propagation library for system developers that exposes an API with the five propagation operations; and a BDL compiler for cross-cutting tool developers that can compile BDL specifications.

Next, system developers (*e.g.*, developers of HDFS, HBase etc.) at development time instrument their systems using the propagation library, to propagate baggage contexts alongside executions. System developers treat baggage contexts as opaque objects. They identify the boundaries of execution – *e.g.* where threads are created, or RPCs are made – and at these boundaries, invoke propagation operations, *i.e.* to duplicate, merge, serialize, deserialize, or trim contexts. Developers also use techniques like thread-local storage to keep track of an execution's current baggage context instance. Overall, this instrumentation task requires no co-ordination with developers of other systems; it also requires no knowledge of any cross-cutting tools that may be later deployed.

Meanwhile, cross-cutting tool developers implement cross-cutting tool logic (*e.g.*, the APIs and backends for Retro, Pivot Tracing, etc.). They use BDL to specify the data types they wish to propagate, and compile the corresponding accessors. The public-facing APIs of their tool (*e.g.* that log trace events, record resource usage, or invoke advice) will include a baggage context instance as an argument. The internal API logic of their tool uses the compiled BDL accessors to observe and manipulate data within the passed baggage context parameter.

Finally, at some point in the future, a system operator decides to deploy a cross-cutting tool. They pick one or more parts of the system in which to deploy the tool. Perhaps this is localized to a single system, component, or function; or it could include disparate components, separated by several levels of indirection. In these chosen locations, they update the source code to add invocations of the cross-cutting tool API, *e.g.* to log trace events. For tools like Retro and Pivot Tracing, these modifications are trivial – for example Pivot Tracing uses an agent

The system operator redeploys the modified parts of their system, then runs a workload. Each execution initially has an empty baggage context. For each execution, the first invocation of the cross-cutting tool API will populate the request's baggage context with some data. This baggage context is carried with the execution, including over the network, within processes, and in particular, through all intermediate layers including those that were left unmodified. Later invocations of the cross-cutting tool API will read data from the baggage context that was written by that first invocation.

8.3 Baggage Context Design

Central to the separation of the two interfaces is an intermediate baggage context representation that is capable of preserving correct propagation behavior for a wide variety of data types. Baggage contexts hide these details from both the system instrumentation for propagating contexts, and from the cross-cutting tool code that uses BDL-generated APIs. Our baggage context design comprises two pieces. First, a core baggage context representation that provides several fundamental properties. Second, an efficient mapping of data types onto the core representation that includes nesting and multiplexing. §8.4 describes concrete details of our implementation.

8.3.1 Core Representation

The core baggage context representation factors out a minimal implementation of the five propagation operations for systems to propagate metadata, while maintaining correctness for arbitrary concurrency patterns. It does *not* include interpreting data types or understanding cross-cutting tools. It encapsulates a simple, concrete implementation of the five propagation operations described in §8.2.1. The most important of these operations to consider for correct propagation of contexts is the merging of contexts when two branches of computation join. To this end we derive the following properties:

Idempotent Merge Many cross-cutting tools have "write-once" contexts, such as Retro's workflow ID or a trace ID [157]. Since a context may be propagated through arbitrary invocations of BRANCH and JOIN, yet remain unchanged, both BRANCH and JOIN must be idempotent in order to constrain the size of a baggage context.

Lazy Resolution We often need to merge baggage contexts that contain different values, and resolve them using tool-specific logic (*e.g.* taking a max or min value). Since we do not interpret the values to merge, our default behavior is to keep both and resolve them later. We expect that, eventually, the relevant cross-cutting tool will access its data in the baggage context, and can then manually interpret and resolve the merge. Lazy resolution implies baggage contexts are collection-like and comprise multiple data elements, and that our merge function merges two collections.

Associative Merge For collection-like baggage contexts, an associative merge function is necessary for the same reason we require an idempotent merge: to constrain the size of a baggage context through arbitrary invocations of BRANCH and JOIN. That is, if we have two "write-once" contexts that remain unchanged, then it is natural to require:

$$\text{merge}(A, \text{merge}(A, B)) = \text{merge}(A, B) \tag{8.0.1}$$

and likewise, by also considering that merge is commutative:

$$\text{merge}(\text{merge}(A, B), B) = \text{merge}(A, B) \tag{8.0.2}$$

Order-Preserving Merge Our final property incorporates ordering into baggage contexts. A baggage context is an ordered collection of elements, but we place no restrictions on the actual interpretation, cardinality, or ordering of elements. Ordering elements implicitly gives us control over element priorities by manipulating their ordering. By extension, our merge function is priority-preserving, *i.e.* its output will preserve the relative order of elements from either input.

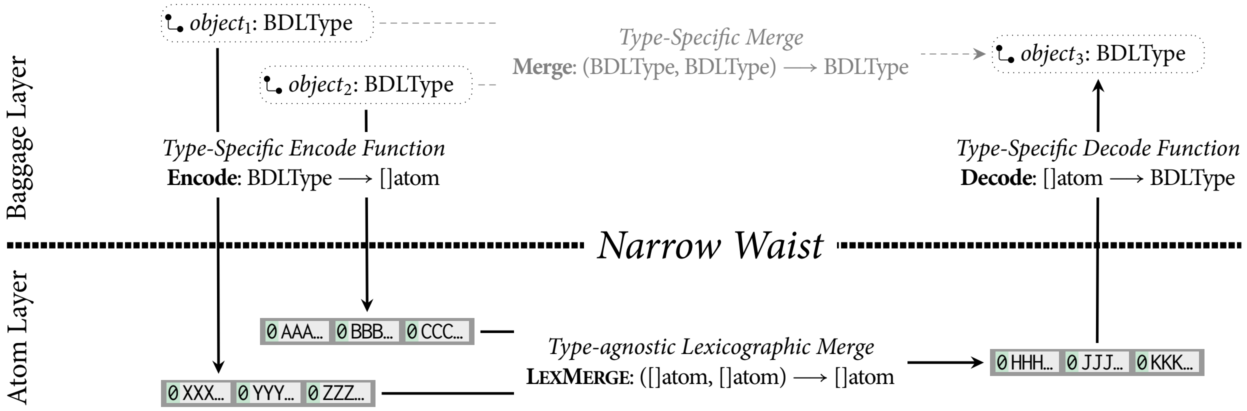


Figure 8.2: All BDL data types have **Encode** and **Decode** functions for encoding and decoding data type instances. For each BDL data type, **Encode** and **Decode** are carefully designed so that lexicographic merge (cf. §8.4.1) correctly preserves the type-specific merge behavior. This enables a separation into layers as described in §8.3.4

8.3.2 Representing Data Types

Using the core baggage context representation, we design encodings for a range of different data types that preserve the correct data type merge semantics through arbitrary invocations of Branch and Join. These data types are exposed through BDL. From a BDL specification, the BDL compiler will generate code that understands how to convert between data type instances and baggage context encodings. Some data types, like primitive types, only encode a single data element to represent their state. Other data types, such as sets and maps, encode their state using multiple data elements. Figure 8.2 illustrates the transformation of BDL data type instances to encoded instances and back again; for all BDL data types, the type-agnostic merge function, when applied to the encoded data type instances, correctly preserves the type-specific merge behavior.

All BDL data types adhere to a common encoding strategy that enables multiplexing of cross-cutting tools and arbitrary nesting of data. The encoding strategy manipulates the relative ordering of individual data elements to take advantage of the order-preserving merge. Nested objects, which might comprise many data elements, are laid out in a specific traversal order that persists through arbitrary invocations of Join. With this strategy, the properties described in §8.3.1 apply recursively to nested objects. §8.4.2 describes our implementation of this encoding strategy.

8.3.3 Conflict-Free Replicated Data Types

Context propagation has a direct analogy to replicated data structures. Replicated data structures comprise multiple independent object instances, and operations performed on one instance eventually propagate as updates to the other instances. In our setting, concurrent branches within an execution maintain their own baggage context instances, each branch interacts with its instance independently, and when two branches join their contexts must be merged. This analogy enables us to draw on a comprehensive body of existing work in conflict-free replicated data types (CRDTs), which are replicated data structures that have deterministic merge

	Developers	*Tasks*	*Abstractions*
(a)	Cross-Cutting Tool Developers (*e.g.* Zipkin, Pivot Tracing)	Cross-cutting tool logic: define context using BDL; use execution-flow scoped variables to implement tracing logic (e.g., spans, security, resource accounting)	Execution-flow scoped variables (§8.2.2)
(b)	Tracing Plane Developers (This chapter)	Tracing plane internals: BDL compiler, underlying context format, nesting, multiplexing, trimming	
(c)	System Developers (*e.g.* HDFS, Spark)	Instrumentation: modify systems to propagate contexts alongside executions	Baggage contexts, five propagation operations (§8.2.1)

Table 8.2: The Tracing Plane provides abstractions to separate the concerns of different developers. Each developer group performs a different task, corresponding to the tasks outlined in Table 7.1.

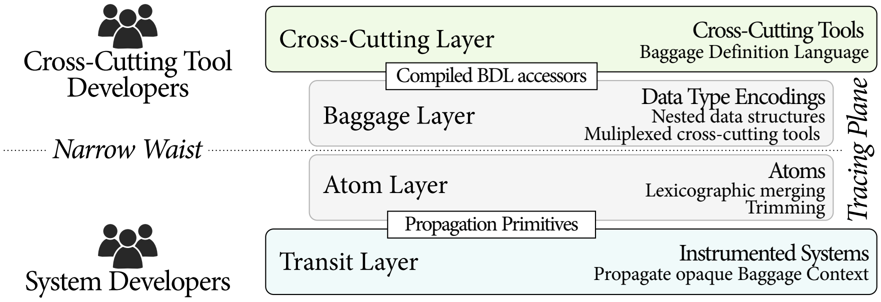


Figure 8.3: The Tracing Plane groups concepts into an abstraction layering that separates the concerns of different developer groups.

behavior and provide strong eventual consistency [233]. In the literature, there are CRDT implementations for a range of data types including sets, maps, registers, counters, and graphs [234]. Furthermore, baggage contexts themselves fulfill the definition of a state-based CRDT [233], as they have an idempotent, associative, and commutative merge function.

8.3.4 Layering Summary

The final component of our design is to group the concepts presented herein into an abstraction layering that we call the *Tracing Plane*. By separating components into distinct layers, the Tracing Plane separates concerns of system developers from those of cross-cutting tool developers (cf. Table 7.1), and makes it easy to implement and support the different features.

Figure 8.3 illustrates the Tracing Plane's layered design. The *transit layer* encapsulates the system-level instrumentation done by system developers, and aims to provide generic and reusable instrumentation (§8.2.1). The *cross-cutting layer* simplifies the development and deployment of cross-cutting tools, by providing BDL to specify cross-cutting tool contexts, and BDL-compiled interfaces to access and manipulate data (§8.2.2). We bridge these layers with two internal layers. The atom layer corresponds to the core baggage context representation and implements the five propagation operations (§8.3.1 and §8.4.1). The baggage layer implements the encoding strategy for nested data structures and multiplexed cross-cutting tools (§8.3.2 and §8.4.2).

Operation	*Atom Layer Implementation*
Branch	Duplicate the baggage context without modification
Join	LexMerge incoming baggage contexts from the joining branches
Serialize	Write baggage atoms in order, each length-prefixed by a varint
Deserialize	Read baggage atoms in order, each length-prefixed by a varint
Trim	Discard atoms from the baggage tail then append trim marker

Table 8.3: Atom layer implementation of the five propagation operations for system instrumentation.

Minimal Tracing Plane support does not require implementation of the full stack: a system needs to support the core baggage context representation to correctly propagate baggage. The atom layer thereby serves as the "narrow waist" of context propagation and provides a low barrier to entry; for example, our Go atom layer implementation has fewer than 100 LOC. However, though the core baggage context representation is designed to be simple and expressive, it is intended as a compilation target for BDL, and the mappings described in §8.4 are wholly encapsulated by the BDL compiler and baggage layer.

8.4 Implementation

We now present our implementation of baggage contexts. Baggage contexts are expressive, efficient, and enable multiple cross-cutting tools to co-exist without interference. Furthermore, it allows for controlling the size of the underlying baggage context.

Zipkin To aid our discussion, we briefly describe the cross-cutting tool Zipkin. Following the overview of end-to-end tracing tools given in §2.4, Zipkin is an open-source end-to-end tracing tool based on Google's Dapper [244]. For each execution it records the hierarchy of invoked RPCs as a tree of causally-related *spans*, with each span containing timing information and logged events. To relate the spans for each request, Zipkin generates a random TraceID at the start of the request, and propagates it with subsequent execution. To capture causal relationships between spans, Zipkin generates and propagates random IDs for the current span and parent span. Zipkin updates these IDs when logging new spans and reinstates parent IDs when closing spans. Zipkin also supports a sampling directive, and, more recently, tags that go in its context (we omit tags from Figure 8.1). When spans end, Zipkin clients log them to a central database.

8.4.1 Core Representation (Atom Layer)

Our core representation is based on two important concepts: *atoms* and *lexicographic merge*.

Atoms A baggage context instance is an array of zero or more *atoms*, where an atom is an arbitrary array of zero or more bytes. The ordering and interpretation of atoms within a baggage context is arbitrary, and different BDL data types write and interpret atoms in different ways.

Propagation Operations Table 8.3 summarizes the five propagation operations for atoms. Deserialize and Serialize read and write atoms in order, varint-prefixed. Branch trivially duplicates atoms. Trim truncates

```
procedure LexCmp($x$, $y$)                              ▷ Lexicographic atom comparison
    for $i \in [0 \ldots \min(x.\text{length}, y.\text{length})]$ do
        if $x[i] \neq y[i]$ then return $x[i] - y[i]$
    return $x.\text{length} - y.\text{length}$
```

(a) Lexicographic atom comparison pseudo-code.

```
procedure LexMerge($a$, $b$)                   ▷ Lexicographic merge of two atom arrays
    $j \leftarrow 0$;   $k \leftarrow 0$;   $out \leftarrow []$
    while not $a$.IsEmpty and not $b$.IsEmpty do
        $cmp \leftarrow$LexCmp($a$.Head, $b$.Head)
        if $cmp < 0$ then  $out$.Push($a$.Pop)
        else if $cmp > 0$ then  $out$.Push($b$.Pop)
        else
            $a$.Pop
            $out$.Push($b$.Pop)
    $out$.PushAll($a, b$)
    return $out$
```

(b) Pseudo-code of lexicographic merge. Two baggage contexts are merged by traversing their atoms in tandem, performing pairwise lexicographic atom comparisons.

	Baggage A	Baggage B	LexMerge (A, B)
	A0 5F1B		A0 5F1B
	A0 5F1B	A0 2B	A0 2B 5F1B
	A0 5F1B	A0 2B 77	A0 2B 5F1B 77
	A0 5F1B	A0 2B 5F1B	A0 2B 5F1B
	A0 5F1B	A0 5F1B0044	A0 5F1B 5F1B0044
	A0 5F1B	BB 2B	A0 5F1B BB 2B
†	A0 5F1B	BB 5F1B	A0 5F1B BB 5F1B

XX atom   XX YY baggage

(c) Lexicographic merge examples; atoms use hexadecimal notation.

Figure 8.4: A baggage context is an array of *atoms*. Baggage contexts are merged using lexicographic comparison.

atoms and appends a special *trim marker*, which is just the zero-length atom; we discuss Trim in §8.4.5. The most important operation is Join.

Lexicographic Order Join is based on the *lexicographic order* of atoms. Lexicographic order is a generalization of alphabetical order: to compare two atoms, compare them byte-by-byte from left-to-right until one byte is found to be less than the other, or one atom is found to be a prefix of the other (Figure 8.4a). For example, the following atoms are lexicographically ordered: 2B01 < 5E7744 < 5F < 5F01 < A0.

Lexicographic Merge To merge two baggage instances, Join performs a *lexicographic merge*, which is similar to the merge of merge-sort: traverse the input arrays in tandem; compare pairs of atoms; select and advance the lexicographically smaller atom each time (Figure 8.4b). Notably however, if two atoms are found to be equal, only output them once. Figure 8.4c illustrates lexicographic merge examples on various baggage contexts. Lexicographic merge does not sort atoms, nor requires the inputs to be sorted. The algorithm makes a single pass through the inputs, and repeated atoms *can* be output if they are not encountered together, *e.g.* 5F1B in Figure 8.4c†. Lexicographic merge satisfies the properties described in §8.3.1: it is idempotent, associative, commutative, lazy, and order preserving.

8.4.2 Atom Encodings (Cross-Cutting Layer)

As described in §8.3.4, the core atom representation and lexicographic merge are together sufficient for propagating opaque baggage contexts. We now describe the intermediate encoding scheme shared by all BDL data types that enables nesting and addressing, and provides isolation and multiplexing of fields and tools.

Atom Prefixes The first byte of each atom is its *prefix*, conceptually similar to IP packet headers. Prefixes serve two purposes: they encode information about the atom's type, and enable us to control the lexicographic order of different atom types regardless of their payload.

Data Atoms A data atom encodes the value of a field or struct and is prefixed by a 0 byte. For example, Zipkin[1] declares a TraceID field of type fixed64 (a 64 bit integer). BDL encodes primitives like fixed64 to one data atom; *e.g.* calling `zmd.SetTraceID(234)` will yield a data atom with the payload 234, *i.e.* `00000000000000000EA`. For ease of demonstration, we abbreviate data atoms by highlighting the `0` prefix and writing literal values, *e.g.* `0234`.

Header Atoms Header atoms address data atoms. Each header encodes one component of a fully qualified path name. For example, to address Zipkin.TraceID requires two header atoms, one each for Zipkin and TraceID. Header atoms encode the prefix byte as follows:

- The first bit of a header atom prefix is `1`, making all header atoms $>_{\text{lex}}$ all data atoms.
- The middle four bits of a header encode its depth in the path in *descending* order from 15 (the maximum depth) – *i.e.* 0→`F`, 1→`E`,…,15→`0`. For example, Zipkin (level 0) and TraceID (level 1) encode `F` and `E` respectively. With this encoding, headers at depth i are $>_{\text{lex}}$ headers at depth $> i$.
- The remaining prefix bits are incidental feature flags.

After the prefix byte, headers encode their component identifier. For a compact encoding we use positional indexes declared in BDL instead of literal identifiers, *e.g.* TraceID→`0`. Root identifiers, *e.g.* Zipkin, lack a positional index and instead use an implicit global mapping of root identifiers to indexes, similar to how TCP ports are mapped to common processes; our examples arbitrarily assign Zipkin to `2`. For ease of demonstration, we abbreviate header atoms by highlighting the leading `1` bit, the header's level, and its identifier, *e.g.* Zipkin.TraceID headers `F802`, `F000` abbreviate as `1F2`, `1E0`.

Atom Order With this encoding, BDL objects lay out their atoms as a pre-order depth-first traversal of the constituent fields. The traversal visits siblings in lexicographically-increasing order of identifier. For example, the complete encoding of Zipkin's TraceID is simply `1F2`, `1E0`, `0234`. If we also set Zipkin's SpanID, *e.g.*`zmd.SetSpanID(55)`, then the encoding is `1F2`, `1E0`, `0234`, `1E1`, `055`. TraceID precedes SpanID due to its lower BDL index. Note that the root Zipkin header `1F2` is not duplicated.

Merging The behavior of this encoding under lexicographic merge is fundamental to preserve BDL data types, and is a core contribution of this paper. Lexicographically merging atoms under this encoding conceptually

[1] Zipkin is described in §8.2.2 and Figure 8.1 outlines Zipkin's BDL declaration.

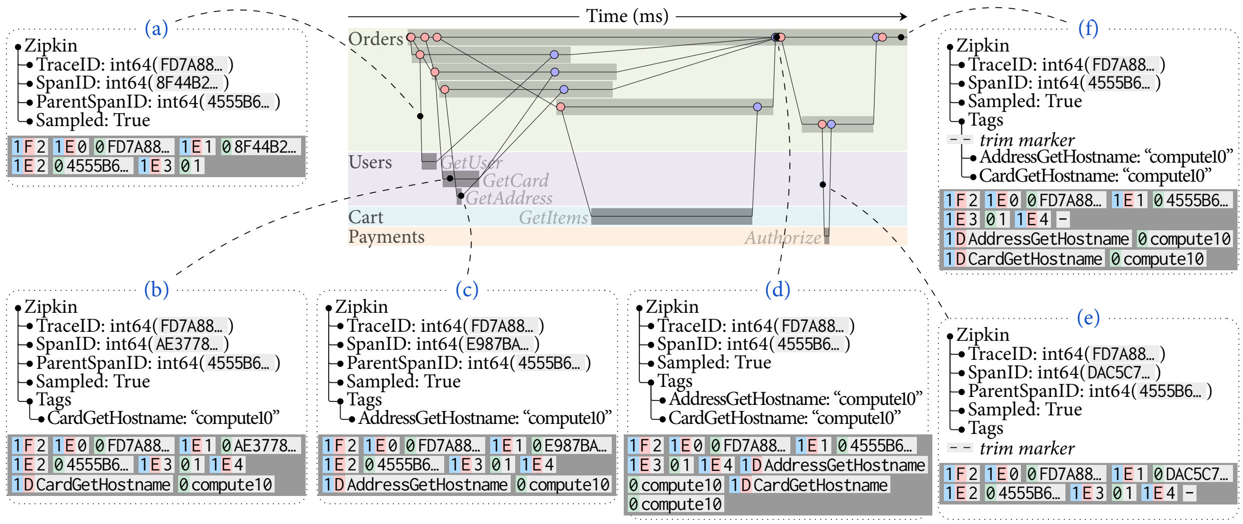

Time (ms)
Orders
Users
Cart
Payments
GetUser
GetCard
GetAddress
GetItems
Authorize
(a)
Zipkin
TraceID: int64(FD7A88...)
SpanID: int64(8F44B2...)
ParentSpanID: int64(4555B6...)
Sampled: True
1F2 1E0 0FD7A88... 1E1 08F44B2...
1E2 04555B6... 1E3 01
(b)
Zipkin
TraceID: int64(FD7A88...)
SpanID: int64(AE3778...)
ParentSpanID: int64(4555B6...)
Sampled: True
Tags
CardGetHostname: "compute10"
1F2 1E0 0FD7A88... 1E1 0AE3778...
1E2 04555B6... 1E3 01 1E4
1D CardGetHostname 0compute10
(c)
Zipkin
TraceID: int64(FD7A88...)
SpanID: int64(E987BA...)
ParentSpanID: int64(4555B6...)
Sampled: True
Tags
AddressGetHostname: "compute10"
1F2 1E0 0FD7A88... 1E1 0E987BA...
1E2 04555B6... 1E3 01 1E4
1D AddressGetHostname 0compute10
(d)
Zipkin
TraceID: int64(FD7A88...)
SpanID: int64(4555B6...)
Sampled: True
Tags
AddressGetHostname: "compute10"
CardGetHostname: "compute10"
1F2 1E0 0FD7A88... 1E1 04555B6...
1E3 01 1E4 1D AddressGetHostname
0compute10 1D CardGetHostname
0compute10
(e)
Zipkin
TraceID: int64(FD7A88...)
SpanID: int64(DAC5C7...)
ParentSpanID: int64(4555B6...)
Sampled: True
Tags
trim marker
1F2 1E0 0FD7A88... 1E1 0DAC5C7...
1E2 04555B6... 1E3 01 1E4 -
(f)
Zipkin
TraceID: int64(FD7A88...)
SpanID: int64(4555B6...)
Sampled: True
Tags
trim marker
AddressGetHostname: "compute10"
CardGetHostname: "compute10"
1F2 1E0 0FD7A88... 1E1 04555B6...
1E3 01 1E4 -
1D AddressGetHostname 0compute10
1D CardGetHostname 0compute10

Figure 8.5: Timeline of a request to order socks in the Sock Shop microservices demo [269], invoking Users, Cart, and Payments microservices. Rows represent threads; shaded bars represent time executing; lines indicate causality; and we highlight invocations of BRANCH (○) and JOIN (○). (b) and (c) illustrate concurrent execution branches setting different Tag values, that are later merged at (d). (e) illustrates atoms after imposing a size constraint of 60 bytes; (f) illustrates how the trim marker preserves the position of potentially lost atoms.

entails a tandem traversal of two trees. The traversal merges all data atoms at each node visited and proceeds jointly through both inputs in depth-first order. Nodes that exist in only one of the inputs are correctly inserted into the appropriate position in the output. Atoms present in both inputs are preserved once without being unnecessarily duplicated. This behavior enables BDL to nest and multiplex fields and tools.

To illustrate, suppose we had set Zipkin's SpanID and TraceID on separate baggage context instances, *i.e.* A = 1F2, 1E0, 0234 and B = 1F2, 1E1, 055. LEXMERGE of A and B correctly yields 1F2, 1E0, 0234, 1E1, 055 — it does not duplicate 1F2; it correctly positions siblings 1E0 and 1E1; and it preserves 0234 and 055 under the correct headers.

8.4.3 Example

To make these concepts concrete, we present an example using the Sock Shop microservices demo [269]. Sock shop comprises 13 microservices written in several languages, primarily Java and Go. We implemented baggage libraries in Java and Go; instrumented the Spring Cloud [246] and Go Kit [255] microservice frameworks; instrumented all Java and Go microservices; and migrated Java and Go Zipkin implementations to interface with baggage contexts instead of hard-coding identifiers.

Figure 8.5 illustrates the end-to-end execution of a request to place an order for socks, with calls to the Orders, Users, Cart and Payments microservices. Swimlanes represent threads; we label the microservices and APIs invoked; and highlight points during execution when the request branches and joins. Figure 8.5a illustrates the object and corresponding atom representation of baggage included in the GetUser call from Orders. The baggage includes values for Zipkin's TraceID, SpanID, ParentSpanID, and Sampled fields.

8.4.4 Complex Data Types

Beyond the simple encodings described in §8.4.2, BDL provides more elaborate data types with encodings comprising multiple atoms. For example, BDL encodes the set data type by encoding one header atom, then encoding a data atom per element and arranging data atoms in lexicographically-increasing order. Lexicographically merging two encoded sets performs the correct set union and outputs sorted data atoms due to the sorted inputs. Other data types supported by BDL include counters, maps, and CRDT variants (cf. §8.4.6). To demonstrate the BDL map type, we update Zipkin's BDL declaration with the following field:

```
map<string, string> tags = 4
```

Tags are a recent extension to Zipkin, present in the Go Zipkin implementation but not Java. Zipkin Tags are inspired by, and named after, Pivot Tracing's baggage; however, we use the name tags instead of baggage to avoid overloaded terminology. For this demonstration, we modify the Go-based User microservice to add tags whenever GetAddress or GetCard is invoked.

Maps encode each key-value pair as a header containing the key literal and a data atom containing the value. Figure 8.5b illustrates atoms after GetCard adds the tag `CardGetHostname=compute10`; similarly Figure 8.5c illustrates atoms after GetAddress adds `AddressGetHostname=compute10`. Note that GetCard and GetAddress are concurrent calls, so atoms in one execution branch will not be visible to the other, and vice-versa, until after the branches join.

Merging The desired merge behavior of a map is to union all keys and recursively merge values mapped under each key. With the above encoding of maps to atoms, lexicographic merge yields the correct behavior. Figure 8.5d illustrates this: after the concurrent GetCard and GetAddress calls return and the branches eventually join, lexicographic merge correctly preserves and orders both of the key-value mappings for Tags.

This example illustrates an important and powerful property of our encoding. The Orders microservice is written in Java, and its Zipkin implementation lacks support for tags. Previously, Zipkin would naïvely ignore and lose any tags propagated from the Go implementation. However, using baggage contexts, lexicographic merge correctly preserves and propagates tags, despite lacking knowledge of its existence or type. All BDL data types achieve this effect, which is extremely useful in environments like this where it is infeasible to redeploy all components for any new tool or update to existing tools. BDL declarations can be updated to add new fields and deprecate existing fields, without affecting backwards compatibility with systems that deployed old versions. The only requirement is that, once deployed, new fields cannot reuse indices of existing fields. In compiled objects, old versions ignore, but continue to propagate, fields they don't know about.

Optimizations Since most BDL fields have a small index (*i.e.*, TraceID=0, SpanID=1), we implement a special variable-length *lexvarint* encoding, similar to protocol buffer's varint, but with lexicographical comparison equivalent to the corresponding integer comparison. Using lexvarints, we can encode most headers in 2 bytes; 1 for the prefix and 1 for the field index. Lexvarints are the default integer type in BDL unless explicitly designated fixed-width (*e.g.*, int64 is a lexvarint; fixed64 is a fixed 8 bytes). We provide further lexvarint encodings for signed and unsigned integers, and ascending and descending order. We evaluate baggage encoded sizes in

§8.5, but in general they are modest, particularly for fixed-width IDs. For example, excluding Tags, Zipkin's fields use 48 bytes.

8.4.5 Overflow

§8.2.1 introduced TRIM, to enable system developers to impose size constraints on baggage contexts, which is often necessary to avoid excessive performance overheads from potentially large contexts [182, 239]. We call it *overflow* when we are forced to discard atoms as a result of TRIM. To handle overflow, we designate the zero-length atom to be a special *trim marker*. TRIM simply drops tail atoms until the size restriction is met, then appends the trim marker. The trim marker is lexicographically less than all other atoms. Consequently wherever it is inserted, it will maintain that exact position, even through subsequent branches and joins. Later, we can observe the position of the trim marker to infer whether atoms may have been dropped.

To illustrate, we modify the Orders microservice to impose an aggressive limit of 60 bytes when calling Payments. Consequently, baggage included with the Payments request overflows, and the tags added by the Users microservice are dropped (Figure 8.5e). Later, when Payments responds to Orders, the baggage containing the trim marker merges back with the sender's local baggage; however, the trim marker persists, marking the position where data may have been discarded. A corollary of TRIM is that the order of BDL fields also implies priority – higher index fields are dropped first by trimming. To ensure that BDL declarations can be updated to add higher priority fields, BDL supports both positive and negative indexed fields.

8.4.6 Conflict-Free Replicated Data Types

As described in §8.3.3, context propagation has a direct analogy to replicated data structures. In the literature, there are CRDT implementations for a range of data types including sets, maps, registers, counters, and graphs [234]. Current BDL data types with CRDT equivalents are counters, flags, sets, and maps. Furthermore, our baggage context implementation naturally provides sets and maps (cf. §8.4.4), corresponding to the add-only set and dictionary CRDTs; in the literature these form building blocks for many CRDTs and enable emulation of all others [234].

To make things concrete, we describe the implementation of an add-only counter using baggage, which mirrors the G-Counter CRDT. Counters are useful for cross-cutting tools, *e.g.* to measure resources consumed during execution [182]. However, it is insufficient to implement a counter by just propagating a single integer, because it can lead to concurrent updates or inadvertent double-counting when later merging baggage instances. Instead, the BDL counter type is similar to a version vector [218]. A counter comprises zero or more components. Each component has a random ID, and stores its value under a header with that ID. To increment, we either increment an existing component, or initialize a new component. To query, we sum up the values in all components. Execution branches reuse their own component ID, but do not share it when branching. Counters thus maintain 1 component for each concurrent branch of execution; this grows proportionally with execution width. When two baggage instances merge, lexicographic merge will recursively merge values under each component. If a component has multiple values (*i.e.*, the merging branches differed), we take the maximum value; this can be done lazily. Finally, counters supporting subtraction (*i.e.*, PN-Counters) are easily implemented by composing two add-only G-Counters, one for addition and one for subtraction.

8.5 Evaluation

§8.4.3 illustrated baggage contexts in the Sock Shop microservices demo environment. The remainder of our evaluation centers on four cross-cutting tools running simultaneously in an instrumented version of the Hadoop stack. We focus on requests to the Hadoop Distributed File System (HDFS) [242] and Spark [280] data analytics jobs running atop Hadoop YARN [264]. We run our experiments on a 25 node cluster. Our evaluation demonstrates that the Tracing Plane:

- supports a range of data types, hiding concurrency subtleties
- supports a variety of cross-cutting tools, deployed simultaneously
- propagates contexts through systems in different languages
- is robust to overflow and systems with size constraints
- makes it easy to update and deploy new tool versions
- is robust to mixed tool versions and black-box propagation

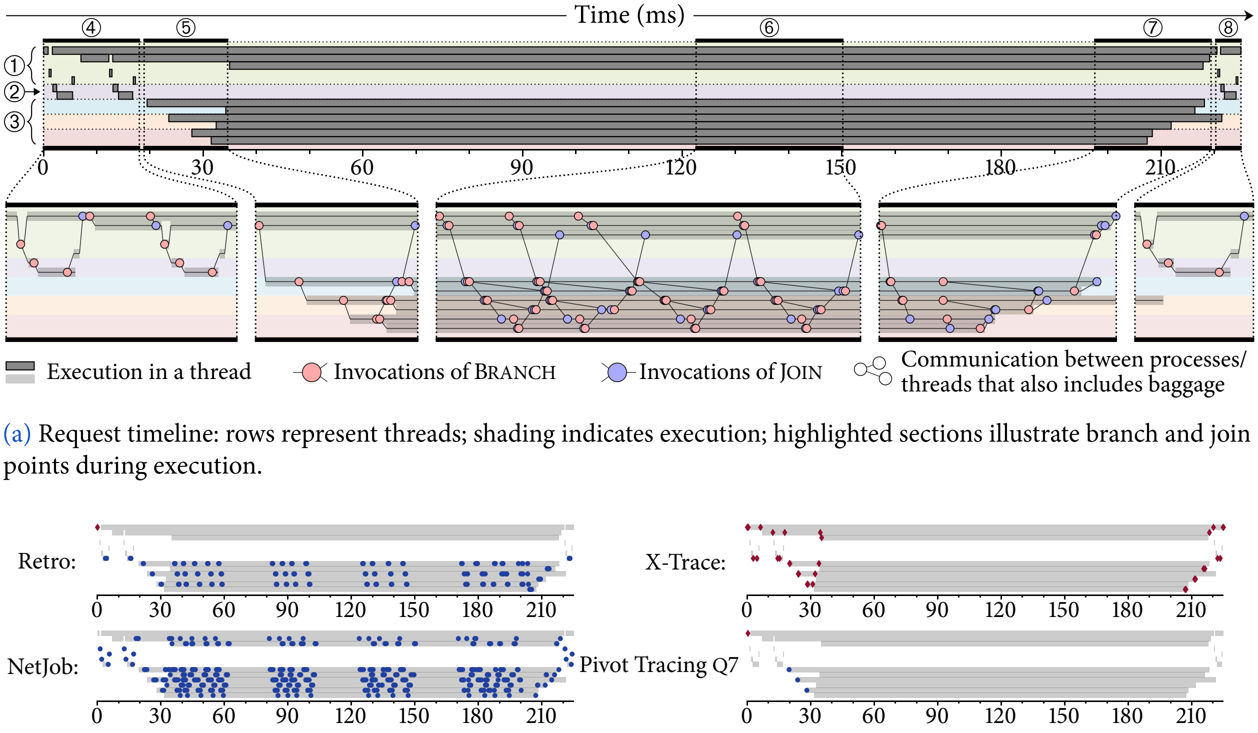
8.5.1 Cross-Cutting Tools

In addition to Zipkin, described in §8.4, we implemented several other cross-cutting tools using the Tracing Plane. Our evaluation also includes updated versions of Retro (Chapter 4) and Pivot Tracing (Chapter 6) which use baggage contexts to propagate workflow IDs and query tuples. Figure 8.1 shows the BDL specifications for all cross-cutting tools included in our evaluation. We briefly summarize these tools

Retro Retro, presented in Chapter 4, is a resource management framework that propagates a tenant ID alongside executions, intercepts API calls to resources (*e.g.*, disk, network, locks, etc.), and aggregates resource counters per tenant. For clarity in this section, we configure Retro to only intercept disk API calls.

X-Trace X-Trace [135] is an end-to-end tracing framework (cf. §2.4); during execution, X-Trace logs events, which are similar to logging statements. When X-Trace logs an event, it attaches three pieces of information: a *task ID* that is randomly generated at the beginning of execution; a randomly generated *event ID*; and *parent IDs* for the immediately preceding events. It then replaces the parent IDs in the baggage with the new event ID. We implement a slight variation of the original X-Trace: in our version, multiple parent events can accumulate when executions merge, so the size of X-Trace's baggage can grow if there are multiple merges and no new event is logged. We generate X-Trace events by overriding Java's log4j logging.

Pivot Tracing Pivot Tracing, presented in Chapter 6 is a dynamic instrumentation system for querying statistics about causally related events. We reproduce Q7 from Pivot Tracing's evaluation (cf. §6.5.1), which propagates two pieces of information: the hostname of the client when initiating an HDFS operation; and the



(a) Request timeline: rows represent threads; shading indicates execution; highlighted sections illustrate branch and join points during execution.

(b) Cross-cutting tools query (●) and update (♦) baggage values at various points during request execution.

Figure 8.6: The end-to-end profile of a 1MB HDFS write request with four cross-cutting tools deployed. See §8.5.2 for a full description.

locations of file replicas on the NameNode when looking up a file. When the request reaches a DataNode, it relates the DataNode's hostname with the two pieces of information, and emits a result tuple.

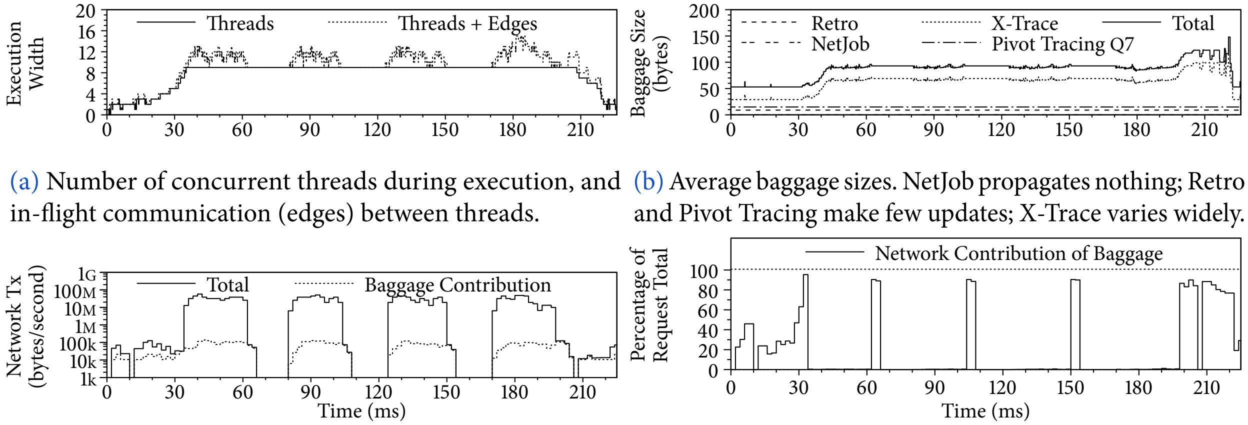
NetJob NetJob is a cross-cutting tool we are developing for monitoring network contention in data analytics jobs. NetJob propagates Hadoop and Spark job and task IDs alongside requests and combines it with network traffic statistics recorded on each node. Our current implementation of NetJob propagates these IDs in a map, to be flexible in experimenting with propagating different information.

8.5.2 Cross-Cutting Tools in Practice

We instrumented HDFS 2.7.2 with the Java tracing plane library and deployed the cross-cutting tools described in §8.5.1.

Execution Patterns

Figure 8.6a illustrates the end-to-end execution timeline of a 1MB HDFS write request, highlighting the BRANCH and JOIN behavior at several points. There are swimlanes for the client (①), NameNode (②), and three DataNodes (③). We highlight 5 phases: in ④, the client makes two RPCs to create the file and allocate a data block; in ⑤ the client sets up a streaming pipeline with three DataNodes; ⑥ zooms in on the streaming of 64kB



(a) Number of concurrent threads during execution, and in-flight communication (edges) between threads.

(b) Average baggage sizes. NetJob propagates nothing; Retro and Pivot Tracing make few updates; X-Trace varies widely.

(c) Total network throughput and baggage overhead (5ms buckets).

(d) Percentage of network traffic due to baggage; baggage imposes low overhead, but is proportionally expensive for small payloads, *e.g.* packet acks (t = 31, t = 61, etc.).

Figure 8.7: Baggage overheads for the 1MB HDFS write request illustrated in Figure 8.6.

application-level packets through the pipeline, in a rather complex pattern of branches and joins (there can be up to 80 unacknowledged packets in flight). After writing the file the client awaits confirmation that the data is synced to disk (⑦), then makes another NameNode RPC to close the file (⑧). This example illustrates how the request execution patterns of a seemingly simple API call can be quite complex, encompassing several models of execution; this contrasts with the comparatively simpler request-response microservices hierarchy in §8.4.3.

Cross-Cutting Tools

Figure 8.6b illustrates the places during request execution where each cross-cutting tool interacted with the request's baggage. Retro writes the tenant ID once and reads it on every disk operation. NetJob intercepts all network communication to check for a job ID, but because we were not running the request as part of a job, it never finds one and does nothing further. X-Trace generates events at several points during execution, which involves both querying and updating its IDs in the baggage. Finally, Pivot Tracing Q7 instruments three locations: the start of the request in the client; the return of `getBlockLocations` on the NameNode; and DataNode `DataTransferProtocol` operations. This example demonstrates how several cross-cutting tools coexist side-by-side using baggage, and how they vary widely in terms of where cross-cutting tool logic is invoked. All systems were instrumented once for propagation, and deploying the tools solely involved defining their BDL representation and using the accessor methods on the relevant variables, at the right points.

Baggage Overheads

Figure 8.7b shows a time series of the average baggage size during the request, and a break down for each cross-cutting tool. Retro and Pivot Tracing only updated values once, and imposed constant-sized overheads of 9 and 15 bytes respectively. NetJob never updated baggage values and imposed no overhead. X-Trace accessed baggage multiple times during execution, and the typical X-Trace overhead was 29 bytes to carry the TaskID and one ParentID. In several places X-Trace accumulated multiple ParentIDs (11 at most, using 153 bytes) due to repeated merges. Normally, X-Trace discards the previous ParentIDs each time it logs a new event; however, as Figure 8.6b illustrates, there is a long period where X-Trace logs no events and does not discard the IDs. Increasing the fidelity to trace-level messages mitigated this and we saw no more than 3 ParentIDs.

In Figure 8.7c, we plot the request's network throughput, and the cost of the baggage that is included in network requests. In aggregate across the request, baggage contributed 11kB of the request's total 3.18MB of network traffic (0.35%). The network utilization of the request varies over time, depending on the stage of execution; RPC communication at the beginning and end of the request has light network usage (t = 0 to 20); streaming data imposes the most overhead (t = 125 to 150); and there are periods of no network utilization while waiting for the client to fill up data buffers (t = 65 to 80). The network contribution of baggage is nearly constant; it is included in all network communication, regardless of payload sizes.

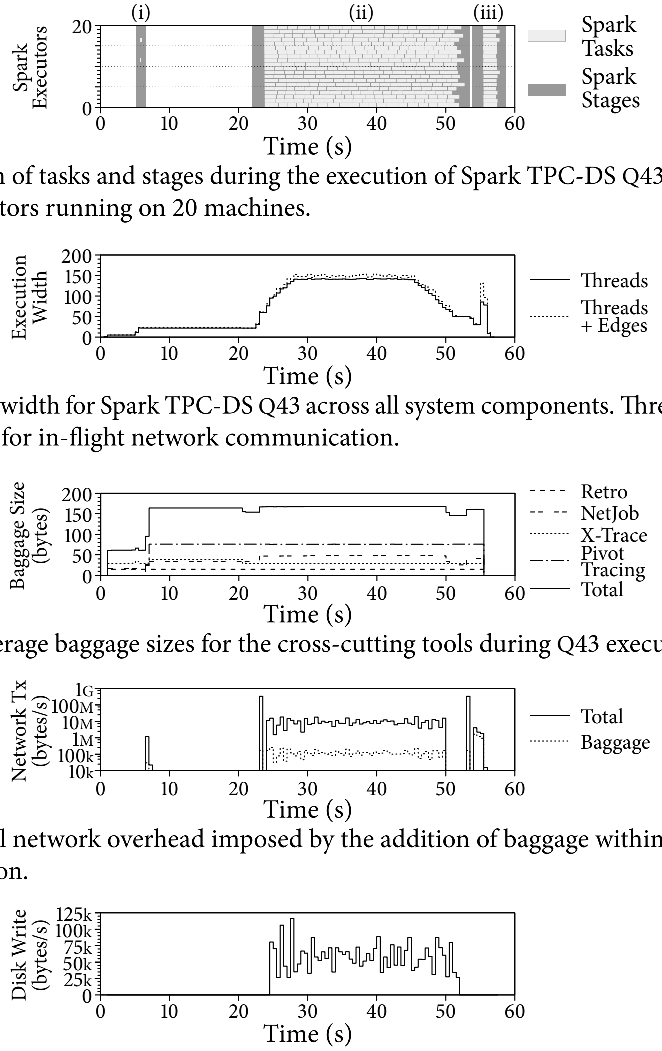
8.5.3 Cross-Cutting Tools at Scale

We now deploy the same set of cross-cutting tools in instrumented versions of Spark, YARN, and HDFS, running on a 25-node Google Compute Engine [140] cluster. Each node is an `n1-standard-4` instance with 4 cores and 15GB memory. Our workload comprises a subset of 19 TPC-DS queries [258], selected by prior benchmarking work [130, 213], with the scale factor set to 100; that is, input data in HDFS is approximately 200GB uncompressed / 17GB compressed.

Overview of a Query

Figure 8.8a illustrates the execution of query 43 [154] (Q43). Spark first acquires 20 *executors* – containers deployed in YARN that cache Spark data in memory and perform Spark computations. The query has three stages: (i) loading (small) metadata from HDFS; (ii) a parallel map stage that reads the tables into memory, filters them, and joins some small tables; (iii) a shuffle stage that combines the query results over the network. In stage (ii), each executor sets up multiple connections to HDFS to read input data, resulting in a large execution width.

Figure 8.8c illustrates the average baggage size during query execution, which peaks at 167 bytes. Pivot Tracing Q7 and NetJob impose higher overheads than the HDFS request in §8.5.2: the job has many HDFS read requests, and Q7 updates the baggage in two places (client and NameNode) instead of the previous one; and NetJob updates the baggage with stage and task IDs as the job runs. X-Trace overhead is lower because fewer parent IDs accumulate in baggage due to merges. Retro overhead is constant as before, since it only propagates a single tenant ID. In aggregate across the job, baggage accounts for a total of 1.0% of network throughput.



(a) Illustration of tasks and stages during the execution of Spark TPC-DS Q43 on a cluster with 20 executors running on 20 machines.

(b) Execution width for Spark TPC-DS Q43 across all system components. Threads + Edges also accounts for in-flight network communication.

(c) Average baggage sizes for the cross-cutting tools during Q43 execution.

(d) Additional network overhead imposed by the addition of baggage within all network communication.

(e) Spark executors spill intermediary output to disk. Retro+ instruments disk writes and increments a counter in baggage.

Figure 8.8: Execution of TPC-DS Q43 on a 25-node Spark cluster.

Developing a New Tool

So far, the demonstrated cross-cutting tools have nearly constant baggage overhead. However, data types such as counters, which are not used by these cross-cutting tools, (cf. §8.4.6) can potentially grow to be large, since they maintain a counter component for each execution branch in the worst case. To illustrate this, and how we can mitigate it, we develop an updated version of Retro called Retro+ which aggregates disk writes using an in-baggage counter. Retro+ modifies Retro's disk instrumentation to increment the counter, and extend Retro's BDL declaration with an additional field: `counter DiskWrites = 1`. Updating Retro+ took less than 10 minutes, and required no additional system-level modifications.

Figure 8.8e plots Spark's disk write over time; every task on every executor writes its output to disk. Figure 8.9a illustrates the growth in baggage over time. Once tasks start writing to disk (t = 25), the baggage

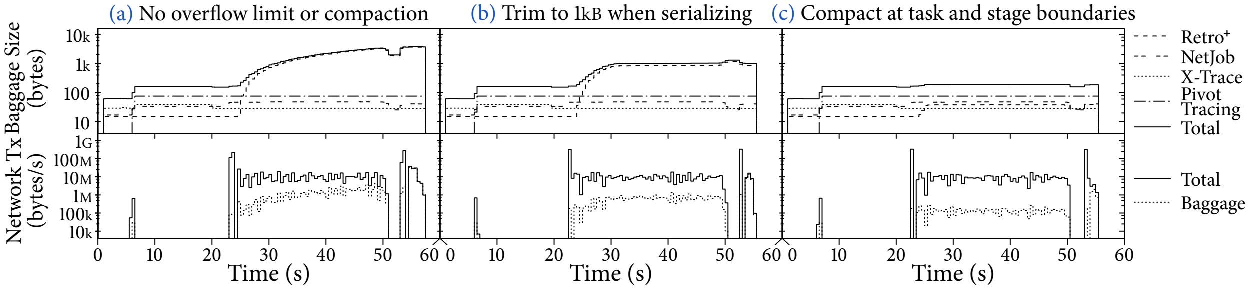

(a) No overflow limit or compaction (b) Trim to 1kB when serializing (c) Compact at task and stage boundaries

Figure 8.9: Baggage size and network overhead for TPC-DS query 43 running on a 25-node Spark cluster. (a) Baggage for Retro+ grows to 4kB in size with no overflow configured; (b) Trimming baggage to 1kB reduces the overhead but sacrifices counter precision; (c) Compacting the counter at carefully chosen points during execution maintains correctness while substantially reducing size.

size increases linearly, as each of the 362 tasks in stage (ii) requires a new counter component. At t = 50, the average baggage size across all execution branches is 3.7kB, which imposes up to 22% network overhead, and a total of 13% in aggregate for the job.

This experiment demonstrates the worst-case behavior of baggage – that it can grow proportional to execution width. This is, of course, dependent on the cross-cutting tool, and not inherent to baggage. We have two mechanisms to counter this. The first, system-level instrumentation can specify hard limits for baggage size by trimming. Figure 8.9b illustrates the growth in baggage size when we limit all baggage serialization to 1kB; at t = 30 we begin dropping atoms, which caps the network overhead to 7.4% for stage (ii) (5% in aggregate for the job). We configure baggage so that Retro+ appears after the other cross-cutting tools, so its atoms are dropped first; this preserves the other cross-cutting tools' baggage. Dropping Retro+'s counter components leads to the counter being inaccurate; by the end of the job, it has a 43% error.

Our second mechanism reduces baggage size and addresses counter error. At well-chosen points during execution (such as BSP barriers), we know that branches have completed. At these points, we can *compact* the baggage by collapsing the now-defunct counter components into a single component. Compaction is a special case of JOIN, but requires knowledge of baggage semantics (*e.g.*, knowing that bag 1 of Retro+ is a counter) and support for the operation by the data type. We updated our Spark instrumentation to compact baggage when tasks and stages complete – an additional two lines of code. Figure 8.9c illustrates baggage overheads with compaction enabled: baggage size peaks at only 186 bytes, which imposes at most 1.4% overhead during stage (ii) and a total of 1.1% in aggregate for the job. This occurs because at the end of each of the 362 tasks in stage (ii), the additional counter component it created is dropped and merged into some other existing counter component. Consequently, the number of components fluctuates between 2 and 4 during the job, overflow does not occur, and the counter value is correct at the end of the job.

Benchmark Results

Figure 8.10 repeats the Spark experiments for 19 TPC-DS queries, including results with and without compaction (Retro+C and Retro+ respectively). The queries each differ in the number of stages and tasks, and at what points

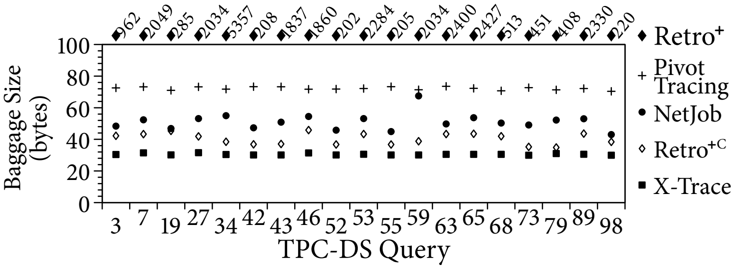


Figure 8.10: Average baggage sizes for cross-cutting tools across 19 TPC-DS queries executed on a 25-node Spark cluster. Figures for Retro$^+$ show with and without compaction.

they write to disk.

8.6 Discussion

In this section we discuss our experiences with using baggage contexts, relating both to the experiments presented in this chapter, and with cross-cutting tools more broadly. We also complement our discussion of the cross-cutting tools and related applications discussed throughout this chapter.

Developing and Updating Tools In general, we found the baggage context abstraction made it easy to develop and update new cross-cutting tools without having to revisit system-level instrumentation. For example, we added per-request logging levels to X-Trace, configuration flags in Retro, and a tool to record critical paths, all without touching the lower level system instrumentation.

Baggage Compaction In §8.5.3 we introduced a *compaction* operator. Compaction is a special case of JOIN, but we omit it from the transit layer API because, in order to compact a data type, it requires knowledge of tool and data type semantics (*i.e.*, knowledge that specific atoms correspond to a specific data type). This circumvents the separation of concerns achieved by our layering, in exchange for improved performance.

Datatype-Aware Overflow Similarly, we only implemented naive overflow that drops atoms from the end of baggage. However, if cross-cutting tool semantics are known (*i.e.*, BDL-generated code for the tool is deployed in a node), then we could implement a data type-aware overflow; for example, a counter could drop the components with the smallest values to minimize counter error.

Lazy Resolution Lazy resolution (§8.3.1) enables baggage contexts to carry essentially arbitrary data types, as cross-cutting tools can defer evaluation of custom merge functions until tool logic is invoked. However, this is only possible because merge is commutative. In some circumstances, tools might want a non-commutative merge operation. For example, in an environment where executions are RPC request-response trees, a tool might want to distinguish between baggage contexts of the caller and callee. We intentionally avoid supporting this use case, as it encourages a more restrictive execution model and commonly leads to brittle instrumentation (cf. §7.4).

Execution Patterns Our experiments demonstrated propagation in a variety of execution patterns, including nested RPCs, continuations, streams, and computation DAGs. Other environments, such as microservices architectures, are even more amenable to features such as compaction.

Instrumentation for Context Propagation Even with end-to-end baggage context propagation, successful cross-cutting tool deployment is still largely dependent on instrumentation decisions made by system developers. Errors in instrumentation can affect cross-cutting tools, as can mismatched expectations of what constitutes the extent of an execution. In this paper we advocate for instrumentation to propagate contexts along the end-to-end execution path of requests, *i.e.*, the "trigger-preserving slice" [230]. However, baggage contexts could be propagated along other dimensions, such as through caches, using the same five propagation operations.

Overheads The application-level overhead of deploying baggage was low, consistent with benchmarks from other papers. Overheads are more dependent on the cost of cross-cutting tool logic than the cost of baggage propagation. Baggage itself can be lazily serialized and deserialized, and branch and join can be efficiently implemented using tricks such as reference counting.

Cross-Language Compatibility BDL's wire specification of serialized atoms is language-independent, and minimal tracing plane support only requires the atom layer. The atom layer is simple and easy to implement, requiring less than 100 LOC in our Go implementation.

Datatype Proofs We have not yet provided formal definitions or proofs of the properties of lexicographic comparison and lexicographic merge, and leave it to future work to examine this in more detail. Particular results that we depend upon in this work include (i) proofs that lexicographic comparison is a total order on atoms; (ii) proofs that lexicographic merge is idempotent, commutative, and associative; (iii) proofs – for all BDL data types and for the overall baggage context representation – that lexicographic merge on their encoded representation correctly preserves the desired merge behavior for the data type (as illustrated in Figure 8.2); and (iv) proofs that baggage contexts themselves are a CRDT under lexicographic merge with the subsequence operator providing a least-upper-bound.

End-to-End Tracing Tools Baggage contexts are designed for arbitrary context data for a broad class of cross-cutting tools. Of these, end-to-end tracing frameworks are the most prevalent today, and these tools are useful for a range of tasks surrounding performance monitoring, anomaly detecting, and resource management. In the open-source community, Dapper's span model is the prevailing approach with numerous derivative implementations [62, 196, 238, 246, 259, 274]. Many popular open-source systems and service frameworks support tracing, including Twitter's Finagle [260], Uber's tchannel [261], and Go kit [255]. As these tracing systems have matured, their community has encountered many of the challenges described in this thesis [273], and recently motivated the OpenTracing effort to standardize the semantics of this class of cross-cutting tools [201]. Canopy [157] also identifies and addresses these challenges, by decoupling aspects of context propagation, instrumentation, and trace representation.

Beyond End-to-End Tracing Recent cross-cutting tools in the research literature tackled a variety of use cases, such as measuring and predicting critical path latency [114, 155, 225, 253], tracking energy consumption [133], understanding client-server interactions [155, 176, 225], making data quality trade-offs [114], attributing resource consumption [180, 182], failure testing [149] and others [106, 161, 161]. Other work has explored similar context propagation ideas at the granularity of network packets [254]. In some cases, authors have experienced limitations similar to those described in §7.4; *e.g.* Facebook authors could not extend existing TraceID propagation to record their desired causality [113]; nor could Google authors extend Dapper [184, 212].

Comparison to Pivot Tracing In Chapter 6 we introduced Pivot Tracing, which includes a related, but more restricted concept of baggage as a generic set of key-value pairs that follows execution. Pivot Tracing does not decouple cross-cutting tools from instrumentation, and its baggage is not order preserving, a requirement for lexicographic merge. This chapter generalizes baggage to a wide range of data types, and introduce abstractions that encapsulate baggage implementations from cross-cutting tools and system developers. Our notion of Baggage relies on the advances in concurrent data types [233, 234], and is inspired by the way in which IDLs such as protocol buffers [263] automate and simplify the tasks of marshalling, serializing, and transporting datastructures.

8.7 Conclusion

The Tracing Plane is a step forward towards truly pervasive instrumentation of distributed systems, addressing important roadblocks. At the system level, it increases the value of instrumenting a system – ideally at development time – as such instrumentation can be re-used by many tracing and related tools. It also makes the work of cross-cutting tool developers much easier, as they can focus on tool logic and data types, and ignore details of serialization, deserialization, propagation, and all of the subtleties of keeping data consistent in face of concurrency. The layered design brings in all the standard benefits of a strong separation of concerns, reuse, and independent evolution around a simple yet expressive narrow waist. While we have demonstrated the implementation of several cross-cutting tools on a number of instrumented systems, the Tracing Plane's ultimate success will be measured by the influence of its ideas in practice.

Chapter 9
Conclusions and Future Work

As systems get composed of increasing numbers of distributed components, understanding how they work and fail hinges on our tools to instrument their execution. While there are many very useful tools, there are important challenges preventing their truly pervasive application.

This thesis outlined several important distributed systems challenges that are prevalent in distributed systems today. These challenges predominantly affect the *end-to-end* behavior of executions, and we use the term *cross-cutting executions* to refer to this dimension of execution, and to this perspective on distributed system behavior. In this thesis we examined these challenges in detail, and characterized a broad class of cross-cutting tools designed to tackle cross-cutting challenges.

This thesis explored in greater detail two application domains where these challenges arise: resource management, and dynamic monitoring. To address these challenges, we designed and implemented two new cross-cutting tools, for each domain respectively.

Subsequent to this examination of two application domains, we generalized our experiences developing these tools, and observations of others from the research literature. We identified several key challenges to the development and pervasive deployment of cross-cutting tools in distributed systems. To address these challenges, we finally proposed common abstractions to underpin their designs.

9.1 Multi-Tenant Resource Management

The first application domain considered in this thesis is resource management in multi-tenant distributed systems. We presented Retro, a framework for implementing resource management policies in multi-tenant distributed systems. Retro tackles important challenges and provides key abstractions that enable a separation between resource-management policies and mechanisms. It requires low developer effort, and is lightweight enough to be run in production. Retro enables policies that are system-agnostic, resource-agnostic, and uniformly treat all system activities, including background management tasks. To the best of our knowledge, Retro is the first framework to do so.

9.2 Dynamic Causal Monitoring

The second application domain considered in this thesis is dynamic monitoring of distributed systems. We presented Pivot Tracing, a monitoring framework for distributed systems that combines dynamic instrumentation and causal tracing. Pivot Tracing is the first monitoring system that enables ad-hoc cross-component monitoring and querying. 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.

9.3 Universal Abstractions for Context Propagation

Building on our experiences with Retro and Pivot Tracing, the final contribution of this thesis generalizes our experiences developing these tools, and observations of others from the research literature Baggage contexts are a step forward towards truly pervasive instrumentation of distributed systems, addressing important roadblocks. At the system level, it increases the value of instrumenting a system – ideally at development time – as such instrumentation can be re-used by many tracing and related tools. It also makes the work of cross-cutting tool developers much easier, as they can focus on tool logic and data types, and ignore details of serialization, deserialization, propagation, and all of the subtleties of keeping data consistent in face of concurrency. The layered design brings in all the standard benefits of a strong separation of concerns, reuse, and independent evolution around a simple yet expressive narrow waist. While we have demonstrated the implementation of several cross-cutting tools on a number of instrumented systems, the Tracing Plane's ultimate success will be measured by the influence of its ideas in practice.

9.4 Future Work

Cloud and distributed systems are still a rapidly changing area. As we discover new designs for distributed systems, new use cases, and new requirements, we will also encounter new and interesting dimensions along which they can fail. There are several exciting future directions in this area. Here, we briefly outline future work in establishing cross-cutting tools in practice, and the common designs and conventions that might emerge over time.

9.4.1 Cross-Cutting Tools

This thesis presents two promising avenues for second-generation cross-cutting tools. Retro improves visibility and control over resource consumption in distributed systems, while Pivot Tracing improves our visibility of unanticipated monitoring problems and enables control over how metrics are grouped and aggregated. However, as we discover new designs for distributed systems, new use cases, and new requirements, we will also encounter new and interesting dimensions along which they can fail. There also remain several interesting applications for cross-cutting tools that are as-yet unexplored. This includes applications in detecting security violations and enforcing security policies; performing targeted chaos engineering, failure testing, and invariant verification; scriptable debugging; improving caching and scheduling heuristics; and many more. Furthermore, many of the traditional tools used for standalone software lack a distributed systems analogue, including fundamental tools such as profilers and stop-the-world debuggers. Distributed systems present a new setting for us to revisit these tools and techniques that are well understood for the single machine setting.

9.4.2 Abstractions for Cross-Cutting Tools

Deploying cross-cutting tools inherently requires coherent choices and participation across all system components. However, today there is little consensus on which tools, abstractions, and approaches to use. Developers of different components are often isolated from one another, causing them to make incompatible or conflicting choices about the cross-cutting tools to embed. Ideally it is not developers who should make these choices, but the operators who deploy the systems at runtime. An interesting future direction to pursue is the general-purpose mechanisms that developers can embed in the system at development time, to enable operators to dynamically deploy *any* cross-cutting tool at runtime. The work presented in this thesis on baggage contexts is a step towards this broader goal. A compelling way to explore this question is to take inspiration from software defined networking, and consider how cross-cutting tools can decouple system-wide enforcement mechanisms from tool-specific control logic. By disentangling control logic from enforcement, we can expose and exploit commonalities in the way different tools observe and manipulate system behaviors.

9.4.3 Automatic Instrumentation

Instrumentation has long been the biggest pain point of deploying cross-cutting tools. The work presented in this thesis reduces, but does not fully remove, the instrumentation burden associated with deploying cross-cutting tools. In the distributed tracing community, as well as the broader application performance monitoring (APM) industry, fully automatic instrumentation is a panacea: if the instrumentation burden can be completely removed, then it potentially enables black-box deployment of many cross-cutting tools. It remains an open question whether fully automatic instrumentation is possible. However, we remain cautiously optimistic, because we observe that instrumentation is challenging *not* because it is complex, but because it must be pervasive; most instrumentation is simple, repetitive, and characterized by only a handful of different techniques for capturing different models. Future work in this area can explore methods for observing or inferring execution models in systems, with the goal of reducing or eliminating the need for manual instrumentation. If successful, this would represent a significant advance for both the research community and industry.

9.4.4 Large-Scale Performance Analytics

First-generation tracing tools have addressed many of the challenges in capturing rich, end-to-end performance traces in large-scale systems. For the intrepid systems researcher, this presents a conundrum — we have been so focused on *how* to collect performance data from systems, that we don't know what to do with it once we have it. The second-generation cross-cutting tools presented in this thesis only perform rudimentary analysis, as do all other tools presented in the research literature. An exciting future research direction lies in advancing the techniques used to derive insights from cross-cutting performance data. This might entail applying techniques from statistics, data mining, and machine learning to large volumes of performance traces. There are a wide range of potential use cases, and they extend beyond just analysis. For example, traces can be used to find features early in an execution that are highly predictive of later performance; this insight can then be exploited by cross-cutting tools at runtime to make better scheduling decisions. One of the key challenges of analyzing performance traces is incorporating their structure, as a trace is conceptually a directed, acyclic graph (DAG) of events, with annotations (e.g. labels and metrics) on events and on edges; the most interesting features are often the structural relationships between events, such as their ordering and timing. This structure makes many off-the-shelf techniques computationally intractable given the number of potential features; it demands new approaches to visualizing, querying, and exploring traces based on structure; and at scale, it imposes new storage and processing constraints.

Bibliography

[1] Adrian Cockroft, Amazon Web Services. Personal Communication. (February 2017). Page 81.

[2] Prateek Agarwal. Distributed Tracing at Yelp. (April 2016). Retrieved January 2017 from `https://engineeringblog.yelp.com/2016/04/distributed-tracing-at-yelp.html`. Page 13.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Pages 8 and 46.

[4] Faruk Akgul. *ZeroMQ*. Packt Publishing, 2013. Page 28.

[5] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval Tree Clocks: A Logical Clock for Dynamic Systems. In *12th International Conference On Principles Of Distributed Systems (OPODIS '08)*. Page 63.

[6] Nuha Alshuqayran, Nour Ali, and Roger Evans. A Systematic Mapping Study in Microservice Architecture. In *9th IEEE International Conference on Service-Oriented Computing and Applications (SOCA '16)*. Pages 1, 2, 6, 7, 8, 9, 75, and 77.

[7] Sara Alspaugh, Bei Di Chen, Jessica Lin, Archana Ganapathi, Marti A Hearst, and Randy H Katz. Analyzing Log Analysis: An Empirical Study of User Log Mining. In *28th USENIX Large Installation System Administration Conference (LISA '14)*. Page 12.

[8] Amazon. Amazon Web Services. Retrieved January 2017 from `https://aws.amazon.com`. [Online; accessed January 2017]. Page 5.

[9] Amazon. AWS Lambda. Retrieved January 2017 from `https://aws.amazon.com/lambda`. Pages 5 and 81.

[10] Amazon. Summary of the Amazon DynamoDB Service Disruption. (September 2015). Retrieved June 2016 from `https://aws.amazon.com/message/5467D2/`. Pages 2, 7, and 15.

[11] Andrew Wang, Cloudera. Personal Communication. (March 2014). Page 17.

[12] Apache. Accumulo. Retrieved January 2017 from `https://accumulo.apache.org/`. Pages 46 and 81.

[13] Apache. ACCUMULO-1197: Pass Accumulo trace functionality through the DFSClient. Retrieved January 2017 from `https://issues.apache.org/jira/browse/ACCUMULO-1197`. Page 81.

[14] Apache. ACCUMULO-3507: NamingThreadFactory.newThread should not wrap runnable with TraceRunnable. Retrieved January 2017 from `https://issues.apache.org/jira/browse/ACCUMULO-3507`. Page 78.

[15] Apache. ACCUMULO-3725: Majc trace tacked onto minc trace. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-3725. Page 78.

[16] Apache. ACCUMULO-3741: Reduce incompatibilities with htrace 3.2.0-incubating. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-3741. Page 82.

[17] Apache. ACCUMULO-4171: Update to htrace-core4. https://issues.apache.org/jira/browse/ACCUMULO-4171. [Online; accessed January 2017]. Pages 81 and 82.

[18] Apache. ACCUMULO-4191: Tracing on client can sometimes lose "sendMutations" events. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-4191. Page 78.

[19] Apache. ACCUMULO-4192: Analyze Threading for Tracing correctness. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-4192. Page 78.

[20] Apache. ACCUMULO-898: Look into replacing CloudTrace. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-898. Page 81.

[21] Apache. Accumulo CloudTrace. Retrieved January 2017 from http://accumulo.apache.org/1.6/accumulo_user_manual.html#_tracing. Page 81.

[22] Apache. Cassandra. Retrieved January 2017 from https://cassandra.apache.org/. Pages 78 and 81.

[23] Apache. CASSANDRA-10392: Allow Cassandra to trace to custom tracing implementations. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-10392. Pages 78 and 81.

[24] Apache. CASSANDRA-1123: Allow tracing query details. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-1123. Page 81.

[25] Apache. CASSANDRA-11706: Tracing payload not passed through newSession(..). Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-11706. Page 78.

[26] Apache. CASSANDRA-12835: Tracing payload not passed from QueryMessage to tracing session. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-12835. Pages 78 and 79.

[27] Apache. CASSANDRA-5483: Repair tracing. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-5483. Pages 78 and 81.

[28] Apache. CASSANDRA-7644: Tracing does not log commitlog/memtable ops when the coordinator is a replica. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-7644. Page 78.

[29] Apache. CASSANDRA-7657: Tracing doesn't finalize under load when it should. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-7657. Page 78.

[30] Apache. CASSANDRA-8032: User based request scheduler. Retrieved June 2016 from https://issues.apache.org/jira/browse/CASSANDRA-8032. Page 21.

[31] Apache. CASSANDRA-8553: Add a key-value payload for third party usage. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-8553. Page 81.

[32] Apache. CLOUDSTACK-618: API request throttling to avoid malicious attacks on MS per account through frequent API request. Retrieved June 2016 from https://issues.apache.org/jira/browse/CLOUDSTACK-618. Page 21.

[33] Apache. Cloudstack API Request Throttling. Retrieved June 2016 from https://cwiki.apache.org/confluence/display/CLOUDSTACK/API+Request+Throttling. Pages 2, 7, and 15.

[34] Apache. HADOOP-13438: Optimize IPC server protobuf decoding. Retrieved January 2017 from https://issues.apache.org/jira/browse/HADOOP-13438. Page 78.

[35] Apache. HADOOP-13473: Tracing in IPC Server is broken. Retrieved January 2017 from https://issues.apache.org/jira/browse/HADOOP-13473. Page 78.

[36] Apache. HADOOP-3810: NameNode seems unstable on a cluster with little space left. Retrieved June 2016 from https://issues.apache.org/jira/browse/HADOOP-3810. Pages 2, 7, and 15.

[37] Apache. HADOOP-6599 Split RPC metrics into summary and detailed metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HADOOP-6599. Page 45.

[38] Apache. HADOOP-6859 Introduce additional statistics to FileSystem. Retrieved July 2017 from https://issues.apache.org/jira/browse/HADOOP-6859. Page 45.

[39] Apache. HADOOP-9640: RPC Congestion Control with FairCallQueue. Retrieved June 2016 from https://issues.apache.org/jira/browse/HADOOP-9640. Page 21.

[40] Apache. HBase. Retrieved June 2016 from https://hbase.apache.org. Pages 5, 16, 64, and 81.

[41] Apache. HBASE-11004: Extend traces through FSHLog#sync. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-11004. Page 78.

[42] Apache. HBASE-11559 Add dumping of DATA block usage to the BlockCache JSON report. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-11559. Page 45.

[43] Apache. HBASE-11598: Add simple RPC throttling. Retrieved June 2016 from https://issues.apache.org/jira/browse/HBASE-11598. Page 21.

[44] Apache. HBASE-12356: Rpc with region replica does not propagate tracing spans. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-12356. Page 79.

[45] Apache. HBASE-12364 API for query metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12364. Page 45.

[46] Apache. HBASE-12424 Finer grained logging and metrics for split transaction. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12424. Page 45.

[47] Apache. HBASE-12477 Add a flush failed metric. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12477. Page 45.

[48] Apache. HBASE-12494 Add metrics for blocked updates and delayed flushes. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12494. Page 45.

[49] Apache. HBASE-12496 A blockedRequestsCount metric. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12496. Page 45.

[50] Apache. HBASE-12574 Update replication metrics to not do so many map look ups. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12574. Page 45.

[51] Apache. HBASE-12938: Upgrade HTrace to a recent supportable incubating version. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-12938. Pages 81 and 82.

[52] Apache. HBASE-13077: BoundedCompletionService doesn't pass trace info to server. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-13077. Page 78.

[53] Apache. HBASE-13078: IntegrationTestSendTraceRequests is a noop. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-13078. Page 79.

[54] Apache. HBASE-13458: Create/expand unit test to exercise htrace instrumentation. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-13458. Page 79.

[55] Apache. HBASE-14451: Move on to htrace-4.0.1 (from htrace-3.2.0) and tell a couple of good trace stories. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-14451. Pages 80 and 82.

[56] Apache. HBASE-15880: RpcClientImpl#tracedWriteRequest incorrectly closes HTrace span. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-14880. Page 78.

[57] Apache. HBASE-2257 [stargate] multiuser mode. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-2257. Page 45.

[58] Apache. HBASE-4038 Hot Region : Write Diagnosis. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4038. Page 45.

[59] Apache. HBASE-4145 Provide metrics for hbase client. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4145. Page 45.

[60] Apache. HBASE-4219 Add Per-Column Family Metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4219. Page 45.

[61] Apache. HBASE-6215: Per-request profiling. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-6215. Page 81.

[62] Apache. HBASE-6449: Dapper like tracing. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-6449. Pages 8, 80, 81, and 105.

[63] Apache. HBASE-7958 Statistics per-column family per-region. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-7958. Page 45.

[64] Apache. HBASE-8370 Report data block cache hit rates apart from aggregate cache hit rates. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-8370. Page 45.

[65] Apache. HBASE-8868 add metric to report client shortcircuit reads. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-8868. Page 45.

[66] Apache. HBASE-9121: Update HTrace to 2.00 and add new example usage. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-9121. Page 82.

[67] Apache. HBASE-9722 need documentation to configure HBase to reduce metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-9722. Page 45.

[68] Apache. HBase Reference Guide. Retrieved July 2017 from https://hbase.apache.org/book.html. Page 45.

[69] Apache. HDFS-10174: Add HTrace support to the Balancer. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-10174. Page 78.

[70] Apache. HDFS-11622 TraceId hardcoded to 0 in DataStreamer, correlation between multiple spans is lost. Retrieved April 2017 from https://issues.apache.org/jira/browse/HDFS-11622. Pages 80 and 81.

[71] Apache. HDFS-4169 Add per-disk latency metrics to DataNode. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-4169. Page 45.

[72] Apache. HDFS-4183: Throttle block recovery. Retrieved June 2016 from https://issues.apache.org/jira/browse/HDFS-4183. Pages 2, 7, 15, and 19.

[73] Apache. HDFS-5253 Add requesting user's name to PathBasedCacheEntry. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-5253. Page 45.

[74] Apache. HDFS-5274: Add Tracing to HDFS. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-5274. Pages 78, 80, and 81.

[75] Apache. HDFS-6093 Expose more caching information for debugging by users. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6093. Page 45.

[76] Apache. HDFS-6268 Better sorting in NetworkTopology.pseudoSortByDistance when no local node is found. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6268. Pages x, 65, 66, and 68.

[77] Apache. HDFS-6292 Display HDFS per user and per group usage on webUI. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6292. Page 45.

[78] Apache. HDFS-7054: Make DFSOutputStream tracing more fine-grained. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-7054. Pages 80 and 81.

[79] Apache. HDFS-7189: Add trace spans for DFSClient metadata operations. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-7189. Page 78.

[80] Apache. HDFS-7390 Provide JMX metrics per storage type. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-7390. Page 45.

[81] Apache. HDFS-7963: Fix expected tracing spans in TestTracing along with HDFS-7054. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-7963. Page 79.

[82] Apache. HDFS-9080: update htrace version to 4.0.1. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-9080. Pages 80 and 82.

[83] Apache. HDFS-945: Make NameNode resilient to DoS attacks (malicious or otherwise). Retrieved June 2016 from https://issues.apache.org/jira/browse/HDFS-945. Pages 2, 7, and 15.

[84] Apache. HDFS-9853: Ozone: Add container definitions. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-9853. Page 80.

[85] Apache. HTrace. Retrieved January 2017 from http://htrace.incubator.apache.org/. Page 46.

[86] Apache. HTRACE-330: Add to Tracer, TRACE-level logging of push and pop of contexts to aid debugging "Can't close TraceScope..". Retrieved January 2017 from https://issues.apache.org/jira/browse/HTRACE-330. Page 78.

[87] Apache. HTRACE-5: Tracing never ends when using TraceRunnable in a thread pool. Retrieved January 2017 from https://issues.apache.org/jira/browse/HTRACE-5. Page 78.

[88] Apache. KUDU-1395: Scanner KeepAlive requests can get starved on an overloaded server. Retrieved June 2016 from https://issues.apache.org/jira/browse/KUDU-1395. Pages 15 and 21.

[89] Apache. MESOS-1949 All log messages from master, slave, executor, etc. should be collected on a per-task basis. Retrieved July 2017 from https://issues.apache.org/jira/browse/MESOS-1949. Page 45.

[90] Apache. MESOS-2157 Add /master/slaves and /master/frameworks/{framework}/tasks/{task} endpoints. Retrieved July 2017 from https://issues.apache.org/jira/browse/MESOS-2157. Page 45.

[91] Apache. PHOENIX-177: Collect usage and performance metrics. Retrieved January 2017 from https://issues.apache.org/jira/browse/PHOENIX-177. Page 80.

[92] Apache. Phoenix 195: Zipkin. Retrieved January 2017 from https://github.com/apache/phoenix/pull/195. Page 81.

[93] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. Page 46.

[94] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. MacroBase: Analytic Monitoring for the Internet of Things. *arXiv preprint arXiv:1603.00567*, 2016. Page 12.

[95] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*. Page 21.

[96] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*. Pages 8, 11, 46, 58, and 83.

[97] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online Modelling and Performance-Aware Systems. In *9th USENIX Workshop on Hot Topics in Operating Systems (HotOS '03)*. Pages 8, 11, 46, and 72.

[98] Jeff Barr. AWS X-Ray – See Inside of Your Distributed Application. (December 2016). Retrieved January 2017 from https://aws.amazon.com/blogs/aws/aws-x-ray-see-inside-of-your-distributed-application/. Pages 8 and 81.

[99] Dan Belcher. Introducing Google Stackdriver: unified monitoring and logging for GCP and AWS. (March 2016). Retrieved January 2017 from https://cloudplatform.googleblog.com/2016/03/Google-Stackdriver-integrated-monitoring-and-logging-for-hybrid-cloud.html. Page 8.

[100] Matteo Bertozzi. New in CDH 5.2: Improvements for Running Multiple Workloads on a Single HBase Cluster. (December 2014). Retrieved June 2016 from http://blog.cloudera.com/blog/2014/12/new-in-cdh-5-2-improvements-for-running-multiple-workloads-on-a-single-hbase-cluster/. Page 21.

[101] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *36th ACM International Conference on Software Engineering (ICSE '14)*. Pages 11 and 46.

[102] Peter Bodik. 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):20–22, 2011. Page 44.

[103] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. Pages 5, 14, 20, and 21.

[104] Bryan Cantrill. Hidden in Plain Sight. *ACM Queue*, 4(1):26–36, 2006. Page 45.

[105] Bryan Cantrill, Michael W Shapiro, and Adam H Leventhal. Dynamic Instrumentation of Production Systems. In *2004 USENIX Annual Technical Conference (ATC)*. Pages 45, 61, and 71.

[106] Anupam Chanda, Alan L Cox, and Willy Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. In *2nd ACM European Conference on Computer Systems (EuroSys '07)*. Pages 8, 9, 21, 46, 48, 69, 75, and 106.

[107] Anupam Chanda, Khaled Elmeleegy, Alan L Cox, and Willy Zwaenepoel. Causeway: Support for Controlling and Analyzing the Execution of Multi-tier Applications. In *6th ACM/IFIP/USENIX International Middleware Conference (Middleware '05)*. Pages 21, 71, 76, 82, and 83.

[108] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*. Page 5.

[109] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*. Pages 64 and 81.

[110] Mike Y Chen, Anthony Accardi, Emre Kiciman, David A Patterson, Armando Fox, and Eric A Brewer. Path-Based Failure and Evolution Management. In *1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*. Pages 8 and 46.

[111] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*. Pages 2, 8, and 46.

[112] Shigeru Chiba. Javassist: Java Bytecode Engineering Made Simple. *Java Developer's Journal*, 9(1), 2004. Page 61.

[113] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. Pages 11, 12, 46, 73, 78, 79, and 106.

[114] Michael Chow, Kaushik Veeraraghavan, Michael Cafarella, and Jason Flinn. DQBarge: Improving Data-Quality Tradeoffs in Large-Scale Internet Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. Pages 2, 9, 75, and 106.

[115] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC '10)*. Page 70.

[116] Graham Cormode and Shan Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005. Page 43.

[117] J. Couckuyt, P. Davies, and J.M. Cahill. Multiple chart user interface, 06 2005. US Patent US6906717 B2. Pages 49 and 72.

[118] Datastax. JAVA-794: Enable tracing accross multiple result pages. Retrieved January 2017 from `https://datastax-oss.atlassian.net/browse/JAVA-794`. Page 78.

[119] Datastax. JAVA-815: No tracing results when a RETRY happens. Retrieved January 2017 from `https://datastax-oss.atlassian.net/browse/JAVA-815`. Page 78.

[120] Jeff Dean. The Rise of Cloud Computing Systems. In *SOSP History Day 2015*. Page 5.

[121] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th USENIX Symposium on Operating System Design and Implementation (OSDI '04)*. Pages 5 and 64.

[122] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. Pages 5, 14, and 20.

[123] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium (Security '11)*. Pages 9 and 75.

[124] Distributed Tracing Workgroup. Distributed Tracing Workgroup. Retrieved January 2017 from `https://goo.gl/xs96fn`. Pages viii and 13.

[125] Distributed Tracing Workgroup. Shared Documents. Retrieved January 2017 from `https://goo.gl/8znW4w`. Page 83.

[126] Thanh Do, Haryadi S Gunawi, Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S Gunawi, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Case for Limping-Hardware Tolerant Clouds. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '13)*. Page 26.

[127] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *4th ACM Symposium on Cloud Computing (SoCC '13)*. Pages 6 and 69.

[128] Dynatrace. Dynatrace Application Monitoring. Retrieved July 2017 from `http://www.dynatrace.com`. Pages 8 and 46.

[129] William Enck, Peter Gilbert, Byung-Gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. Pages 9 and 75.

[130] Justin Erickson, Marcel Kornacker, and Dileep Kumar. New SQL Choices in the Apache Hadoop Ecosystem: Why Impala Continues to Lead. (May 2014). Retrieved January 2017 from `https://blog.cloudera.com/blog/2014/05/new-sql-choices-in-the-apache-hadoop-ecosystem-why-impala-continues-to-lead/`. Page 101.

[131] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. Pages 45, 48, 59, 61, 71, and 72.

[132] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing*, 3(5):10–14, 2016. Pages 1, 2, 6, and 7.

[133] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. Pages 9 and 106.

[134] Rodrigo Fonseca, Michael J Freedman, and George Porter. Experiences with Tracing Causality in Networked Services. In *2010 USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN '10)*. Pages 11 and 78.

[135] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*. Pages 2, 8, 11, 13, 46, 48, 60, 72, 74, 78, 79, 82, and 98.

[136] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Pages 5 and 6.

[137] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Pages 20, 24, and 32.

[138] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*. Pages 18, 24, 32, and 33.

[139] Google. Cloud Platform. Retrieved January 2017 from `https://cloud.google.com`. Page 5.

[140] Google. Compute Engine. Retrieved January 2017 from `https://cloud.google.com/compute/`. Page 101.

[141] Google. gRPC/Census. Retrieved January 2017 from `https://goo.gl/iEqlqH`. Pages 9, 13, 75, and 77.

[142] Oliver Gould. Real World Microservices: When Services Stop Playing Well and Start Getting Real. (May 2016). Retrieved July 2017 from `https://blog.buoyant.io/2016/05/04/real-world-microservices-when-services-stop-playing-well-and-start-getting-real/`. Pages 2 and 7.

[143] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997. Pages 49 and 72.

[144] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. Pages 18 and 43.

[145] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure Recovery: When the Cure Is Worse Than the Disease. In *14th USENIX Workshop on Hot Topics in Operating Systems (HotOS '13)*. Pages 1, 2, 6, 7, 15, and 19.

[146] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G^2: A Graph Processing System for Diagnosing Distributed Systems. In *2011 USENIX Annual Technical Conference (ATC)*. Pages 8 and 46.

[147] Jiawei Han, Yixin Chen, Guozhu Dong, Jian Pei, Benjamin W Wah, Jianyong Wang, and Y Dora Cai. Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005. Page 73.

[148] Matt Heath. A Journey into Microservices: Dealing with Complexity. (March 2015). Retrieved January 2017 from `http://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-3/`. Pages 1, 2, 6, 7, and 13.

[149] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. Gremlin: Systematic Resilience Testing of Microservices. In *36th IEEE International Conference on Distributed Computing Systems (ICDCS '16)*. Pages 9 and 106.

[150] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*. Page 20.

[151] @Honest_Update. Honest status page. (October 2015). Retrieved July 2017 from `https://twitter.com/honest_update/status/651897353889259520`. Page 7.

[152] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *26th IEEE International Conference on Data Engineering Workshops (ICDEW '10)*. Page 70.

[153] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (ATC)*. Pages 5, 14, and 16.

[154] Impala TPC-DS Kit. TPC-DS Query 43. Retrieved January 2017 from `https://github.com/cloudera/impala-tpcds-kit/blob/c5d32ae55a5259dd081bf4546bb650b2a3d668de/queries/q43.sql`. Page 101.

[155] Yurong Jiang, Lenin Ravindranath, Suman Nath, and Ramesh Govindan. WebPerf: Evaluating "What-If" Scenarios for Cloud-hosted Web Applications. In *2016 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Pages 8, 79, 82, and 106.

[156] Theodore Johnson. Approximate Analysis of Reader/Writer Queues. *IEEE Transactions on Software Engineering*, 21(3):209–218, 1995. Page 30.

[157] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and

Yee Jiun Song. Canopy: An End-to-End Performance Tracing And Analysis System. In *26th ACM Symposium on Operating Systems Principles (SOSP '17)*. Pages 6, 7, 8, 12, 13, 74, 75, 79, 81, 89, and 106.

[158] Henry R Kang. *Computational Color Technology*. SPIE Press Bellingham, 2006. Page 31.

[159] Sung-Il Kang and Heung-Kyu Lee. Analysis and Solution of Non-Preemptive Policies for Scheduling Readers and Writers. *SIGOPS Operating Systems Review*, 32(3):30–50, 1998. Page 30.

[160] Nitesh Kant. Distributed Tracing at Netflix. (July 2015). Retrieved January 2017 from `https://speakerdeck.com/niteshkant/distributed-tracing-at-netflix`. Page 13.

[161] Partha Kanuparthy, Yuchen Dai, Sudhir Pathak, Sambit Samal, Theophilus Benson, Mojgan Ghasemi, and PPS Narayan. YTrace: End-to-end Performance Diagnosis in Large Cloud and Content Providers. *arXiv preprint arXiv:1602.03273*, 2016. Pages 8, 13, and 106.

[162] Suman Karumuri. PinTrace: Distributed Tracing at Pinterest. (August 2016). Retrieved July 2017 from `https://www.slideshare.net/mansu/pintrace-advanced-aws-meetup`. Pages 11, 13, 78, and 81.

[163] Soila P. Kavulya, Scott Daniels, Kaustubh Joshi, Matti Hiltunen, Rajeev Gandhi, and Priya Narasimhan. Draco: Statistical Diagnosis of Chronic Problems in Large Distributed Systems. In *42nd IEEE/IFIP Conference on Dependable Systems and Networks (DSN '12)*. Pages 11, 12, and 46.

[164] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP '01)*. Pages 29, 61, and 65.

[165] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming (ECOOP '97)*. Page 51.

[166] Tom Killalea. The Hidden Dividends of Microservices. *Communications of the ACM*, 59(8):42–45, 2016. Pages 1, 2, 6, and 7.

[167] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. Root Cause Detection in a Service-Oriented Architecture. In *2013 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. Page 46.

[168] Matt Klein. Lyft's Envoy: From Monolith to Service Mesh. (January 2017). Retrieved January 2017 from `https://www.microservices.com/talks/lyfts-envoy-monolith-service-mesh-matt-klein/`. Page 13.

[169] Steven Y Ko, Praveen Yalagandula, Indranil Gupta, Vanish Talwar, Dejan Milojicic, and Subu Iyer. Moara: Flexible and Scalable Group-Based Querying System. In *9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08)*. Pages 12 and 46.

[170] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*. Page 15.

[171] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978. Pages 7, 48, 53, and 76.

[172] Brian Laub, Chengwei Wang, Karsten Schwan, and Chad Huneycutt. Towards Combining Online & Offline Management for Big Data Applications. In *11th USENIX International Conference on Autonomic Computing (ICAC '14)*. Pages 64 and 69.

[173] Jonathan Leavitt. End-to-End Tracing Models: Analysis and Unification. B.Sc. Thesis, Brown University, 2014. Page 7.

[174] Chris Li. eBay Tech Blog: Quality of Service in Hadoop. (August 2014). Retrieved June 2016 from `http://www.ebaytechblog.com/2014/08/21/quality-of-service-in-hadoop/`. Pages 2, 7, and 15.

[175] Chris Li. HDFS NameNode Denial of Service Resilience. Retrieved June 2016 from `https://issues.apache.org/jira/secure/attachment/12616864/NN-denial-of-service-updated-plan.pdf`. Page 15.

[176] Ding Li, James Mickens, Suman Nath, and Lenin Ravindranath. Domino: Understanding Wide-Area, Asynchronous Event Causality in Web Applications. In *6th ACM Symposium on Cloud Computing (SoCC '15)*. Pages 8, 82, and 106.

[177] Lightstep. Lightstep. Retrieved January 2017 from `http://lightstep.com`. Page 79.

[178] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, and Andrew Wang. Kudu: Storage for Fast Analytics on Fast Data. Retrieved June 2016 from `http://getkudu.io/kudu.pdf`. Pages 2, 7, and 15.

[179] João Loff, Daniel Porto, Carlos Baquero, João Garcia, Nuno Preguiça, and Rodrigo Rodrigues. Transparent Cross-System Consistency. In *3rd International Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '17)*. Pages 2, 9, and 75.

[180] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. Pages 9, 69, 74, 78, 79, and 106.

[181] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Towards General-Purpose Resource Management in Shared Cloud Services. In *10th USENIX Workshop on Hot Topics in System Dependability (HotDep '14)*. Pages 78 and 79.

[182] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*. Pages 9, 12, 75, 79, 83, 97, and 106.

[183] Gurmeet Singh Manku and Rajeev Motwani. Approximate Frequency Counts over Data Streams. In *28th International Conference on Very Large Data Bases (VLDB '02)*. Page 43.

[184] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. Modeling the Parallel Execution of Black-Box Services. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '11)*. Pages 8, 11, 46, and 106.

[185] Matthew L Massie, Brent N Chun, and David E Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004. Pages 12, 45, and 46.

[186] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *2006 ACM SIGMOD International Conference on Management of Data*. Page 50.

[187] Paul Menage. Control Groups. (2004). Retrieved July 2017 from `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`. Page 15.

[188] Haibo Mi, Huaimin Wang, Zhenbang Chen, and Yangfan Zhou. Automatic Detecting Performance Bugs in Cloud Computing Systems via Learning Latency Specification Model. In *8th IEEE International Symposium on Service Oriented System Engineering (SOSE '14)*, pages 302–307. IEEE. Pages 8 and 46.

[189] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael R Lyu, and Hua Cai. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013. Pages 8 and 46.

[190] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, Hua Cai, and Gang Yin. An Online Service-Oriented Performance Profiling Tool for Cloud Computing Systems. *Frontiers of Computer Science*, 7(3):431–445, 2013. Pages 8 and 46.

[191] Microsoft. Azure. Retrieved January 2017 from `https://azure.microsoft.com`. Page 5.

[192] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo I Seltzer. Provenance for the Cloud. In *8th USENIX Conference on File and Storage Technologies (FAST '10)*. Pages 9 and 75.

[193] Andrew C Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *16th ACM Symposium on Operating Systems Principles (SOSP '97)*. Pages 9 and 75.

[194] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. Pages 12 and 46.

[195] Vivek R. Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*. Pages 20 and 21.

[196] Naver. Pinpoint. Retrieved January 2017 from `https://github.com/naver/pinpoint`. Pages 8, 83, and 105.

[197] Netflix. Netflix Open Source Software. Retrieved January 2017 from `http://netflix.github.io/`. Pages 2 and 5.

[198] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015. Pages 1, 2, 6, and 7.

[199] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and Challenges in Log Analysis. *Communications of the ACM*, 55(2):55–61, 2012. Page 46.

[200] Adam J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken. Using Correlated Surprise to Infer Shared Influence. In *40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*. Pages 11 and 46.

[201] OpenTracing. OpenTracing. Retrieved January 2017 from `http://opentracing.io/`. Pages 8, 78, and 106.

[202] OpenTracing. OpenTracing 28: Non-RPC Spans and Mapping to Multiple Parents. Retrieved January 2017 from `https://github.com/opentracing/opentracing.io/issues/28`. Pages 80 and 83.

[203] OpenTracing. Specification 23: Standard(s) for in-process propagation. Retrieved February 2017 from `https://github.com/opentracing/specification/issues/23`. Page 83.

[204] OpenTracing. Specification 5: Non-RPC Spans and Mapping to Multiple Parents. Retrieved February 2017 from `https://github.com/opentracing/specification/issues/5`. Pages 80, 82, and 83.

[205] OpenZipkin. OpenZipkin 48: Would a common http response id header be helpful? Retrieved January 2017 from `https://github.com/openzipkin/openzipkin.github.io/issues/48`. Page 80.

[206] OpenZipkin. Zipkin 1189: Representing an asynchronous span in Zipkin. Retrieved January 2017 from `https://github.com/openzipkin/zipkin/issues/1189`. Pages 80 and 81.

[207] OpenZipkin. Zipkin 1243: Support async spans. Retrieved January 2017 from `https://github.com/openzipkin/zipkin/issues/1243`. Pages 80 and 81.

[208] OpenZipkin. Zipkin 1244: Multiple parents aka Linked traces. Retrieved January 2017 from `https://github.com/openzipkin/zipkin/issues/1244`. Pages 80 and 81.

[209] OpenZipkin. Zipkin 925: How to track async spans? Retrieved January 2017 from `https://github.com/openzipkin/zipkin/issues/925`. Pages 80 and 81.

[210] OpenZipkin. Zipkin 939: Zipkin v2 span model. Retrieved January 2017 from `https://github.com/openzipkin/zipkin/issues/939`. Pages 80 and 81.

[211] Oracle. The Java HotSpot Performance Engine Architecture. Retrieved March 2015 from `http://www.oracle.com/technetwork/java/whitepaper-135217.html`. Page 70.

[212] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. Diagnosing Latency in Multi-Tier Black-Box Services. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS '11)*. Pages 8, 46, and 106.

[213] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. Pages 12 and 101.

[214] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Page 20.

[215] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *1998 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Page 30.

[216] Abhay K Parekh and Robert G Gallagher. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994. Page 26.

[217] Insung Park and Ricky Buch. Event Tracing: Improve Debugging and Performance Tuning with ETW. (April 2007). Retrieved July 2017 from `http://download.microsoft.com/download/3/A/7/3A7FA450-1F33-41F7-9E6D-3AA95B5A6AEA/MSDNMagazineApril2007en-us.chm`. [Online; published April 2007; accessed July 2017]. Page 71.

[218] D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, (3):240–247, 1983. Pages 63, 81, and 97.

[219] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing The Network In Cloud Computing. In *2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Page 15.

[220] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating System Problems using Dynamic Instrumentation. In *2005 Ottawa Linux Symposium*. Page 45.

[221] Lindsey C. Puryear and Vidyadhar G. Kulkarni. Comparison of Stability and Queueing Times for Reader-Writer Queues. *Performance Evaluation*, 30(4):195–215, 1997. Page 30.

[222] Ariel Rabkin and Randy Howard Katz. How Hadoop Clusters Break. *IEEE Software*, 30(4):88–94, 2013. Pages 45 and 46.

[223] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000. Pages 58 and 72.

[224] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. Page 82.

[225] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Pages 2, 9, 22, 69, 75, 82, and 106.

[226] John Reumann and Kang G Shin. Stateful Distributed Interposition. *ACM Transactions on Computer Systems*, 22(1):1–48, 2004. Pages 71, 76, 82, and 83.

[227] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*. Pages 2, 8, 11, and 46.

[228] Javi Roman. The Hadoop Ecosystem Table. Retrieved January 2017 from `https://hadoopecosystemtable.github.io/`. Pages 5 and 81.

[229] Raja R. Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R. Ganger. So, you want to trace your distributed system? Key design insights from years of practical experience. Technical Report CMU-PDL-14-102, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA 15213-3890, April 2014. Page 47.

[230] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. Principled Workflow-Centric Tracing of Distributed Systems. In *7th ACM Symposium on Cloud Computing (SOCC '16)*. Pages 7, 11, 78, 79, and 105.

[231] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*. Pages 8 and 46.

[232] Peter Schuller. Manhattan, our real-time, multi-tenant distributed database for Twitter scale. (April 2014). Retrieved June 2016 from `https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.html`. Page 21.

[233] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '11)*. Pages 77, 91, and 106.

[234] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Technical Report, Inria–Centre Paris-Rocquencourt; INRIA, 2011. Pages 91, 97, and 106.

[235] Kai Shen and Meng Zhu. Best-Effort Request Labeling and Scheduling on Multicore Servers. Technical Report, University of Rochester, 2016. Pages 76 and 82.

[236] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10)*. Page 15.

[237] Yuri Shkuro. Baggage Propagation at Uber. (September 2017). Retrieved October 2017 from `https://github.com/TraceContext/tracecontext-spec/issues/13#issuecomment-330094227`. Pages 2, 9, 10, 74, and 75.

[238] Yuri Shkuro. Evolving Distributed Tracing at Uber Engineering. (February 2017). Retrieved July 2017 from `https://eng.uber.com/distributed-tracing/`. Pages 8 and 105.

[239] Yuri Shkuro. Jaeger #373: Baggage Whitelisting. (September 2017). Retrieved October 2017 from `https://github.com/jaegertracing/jaeger/issues/373`. Page 97.

[240] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996. Page 43.

[241] David Shue, Michael J Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. Pages 2, 9, 20, and 21.

[242] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. Pages ix, 5, 6, 14, 16, 17, 20, 64, and 98.

[243] Benjamin H Sigelman. Towards Turnkey Distributed Tracing. (June 2016). Retrieved January 2017 from `https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736`. Pages 11 and 78.

[244] 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. Technical Report, Google, 2010. Pages 2, 8, 9, 13, 46, 47, 48, 60, 72, 74, 75, 77, 78, 79, 82, and 92.

[245] SolarWinds. Traceview. `https://traceview.solarwinds.com/`. [Online; accessed July 2017]. Pages 8 and 46.

[246] Spring. Spring Cloud Sleuth. Retrieved January 2017 from `http://cloud.spring.io/spring-cloud-sleuth/`. Pages 8, 82, 95, and 105.

[247] Spring Cloud. Sleuth 306: Malformed X-B3 headers cause 500 error. Retrieved January 2017 from `https://github.com/spring-cloud/spring-cloud-sleuth/issues/306`. Page 81.

[248] Spring Cloud. Sleuth 410: Trace ID problem when using Spring ThreadPoolTaskExecutor. Retrieved January 2017 from `https://github.com/spring-cloud/spring-cloud-sleuth/issues/410`. Page 78.

[249] Spring Cloud. Sleuth 424: Not seeing traceids in the http response headers. Retrieved January 2017 from `https://github.com/spring-cloud/spring-cloud-sleuth/issues/424`. Page 80.

[250] Spring Cloud. Sleuth 425: Make Sleuth more robust in accepting invalid span headers. Retrieved January 2017 from `https://github.com/spring-cloud/spring-cloud-sleuth/issues/425`. Page 81.

[251] Dimitrios Stiliadis and Anujan Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, 1998. Page 26.

[252] StrongLoop Arc. Tracing. Retrieved January 2017 from `https://docs.strongloop.com/display/SLC/Tracing`. Page 82.

[253] Hongkai Sun. General Baggage Model for End-to-End Tracing and Its Application on Critical Path Analysis. M.Sc. Thesis, Brown University, 2016. Pages 9, 75, and 106.

[254] Kun Suo, Jia Rao, Luwei Cheng, and Francis Lau. Time Capsule: Tracing Packet Latency across Different Layers in Virtualized Systems. In *7th ACM Asia-Pacific Workshop on Systems (APSys '16)*. Pages 9 and 106.

[255] The Go Blog. Go kit: A toolkit for microservices. Retrieved October 2017 from `https://gokit.io`. Pages 82, 95, and 106.

[256] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-defined Storage Architecture. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Pages 18, 20, and 27.

[257] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: Tracking Activity in a Distributed Storage System. In *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. Pages 8, 11, 46, 60, and 72.

[258] Transaction Processing Performance Council. TPC Benchmark DS Version 2.4.0. (February 2017). Retrieved March 2017 from `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.4.0.pdf`. Page 101.

[259] Twitter. Zipkin. Retrieved July 2017 from `http://zipkin.io/`. Pages 8, 46, and 105.

[260] Twitter Open Source. Finagle: A Fault Tolerant, Protocol-Agnostic RPC System . Retrieved April 2018 from `https://twitter.github.io/finagle`. Pages 82 and 106.

[261] Uber Open Source. TChannel: Network multiplexing and framing protocol for RPC. Retrieved April 2018 from `https://github.com/uber/tchannel`. Pages 82 and 106.

[262] Robbert Van Renesse, Kenneth P Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology For Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003. Pages 12 and 46.

[263] Kenton Varda. Protocol Buffers: Google's Data Interchange Format. (July 2008). Retrieved January 2017 from `https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html`. Pages 28, 61, 87, and 106.

[264] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *4th ACM Symposium on Cloud Computing (SoCC '13)*. Pages 16, 20, 65, and 98.

[265] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *3rd ACM Symposium on Cloud Computing (SoCC '12)*. Pages 20, 21, and 36.

[266] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Ion Stoica, and Randy Katz. Sweet Storage SLOs with Frosting. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)*. Page 20.

[267] Chengwei Wang, Infantdani Abel Rayan, Greg Eisenhauer, Karsten Schwan, Vanish Talwar, Matthew Wolf, and Chad Huneycutt. VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications. In *13th ACM/IFIP/USENIX International Middleware Conference (Middleware '12)*. Pages 12, 47, 64, and 69.

[268] Chengwel Wang, Soila P Kavulya, Jiaqi Tan, Liting Hu, Mahendra Kutare, Mike Kasick, Karsten Schwan, Priya Narasimhan, and Rajeev Gandhi. Performance Troubleshooting in Data Centers: An Annotated Bibliography. *ACM SIGOPS Operating Systems Review*, 47(3):50–62, 2013. Pages 12 and 46.

[269] Weaveworks and Container Solutions. Sock shop: A microservices demo application. Retrieved October 2017 from `https://microservices-demo.github.io`. Pages xi and 95.

[270] Mick Semb Wever. Replacing Cassandra's tracing with Zipkin. (December 2015). Retrieved July 2017 from `http://thelastpickle.com/blog/2015/12/07/using-zipkin-for-full-stack-tracing-including-cassandra.html`. Page 81.

[271] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel Performance Counter Monitor – A better way to measure CPU utilization. (January 2017). Retrieved July 2017 from `http://intel.ly/1C23e67`. Page 29.

[272] Peter T. Wood. Query Languages for Graph Databases. *SIGMOD Record*, 41(1):50–60, 2012. Page 72.

[273] Distributed Tracing Workgroup. Tracing Workshop. (February 2017). Retrieved February 2017 from `https://goo.gl/2WKjhR`. Pages 80, 81, and 106.

[274] Paul Wright. CrossStitch: What Etsy Learned Building a Distributed Tracing System. (September 2014). Retrieved January 2017 from `https://www.slideshare.net/PaulWright9/crossstitch-what-etsy-learned-building-a-distributed-tracing-system-for-surge-conference-2014`. Pages 8, 13, and 105.

[275] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. Pages 11, 12, and 46.

[276] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *21st USENIX Security Symposium (Security '12)*. Pages 9 and 75.

[277] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 159–172. ACM. Page 45.

[278] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving Software Diagnosability via Log Enhancement. In *16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. Pages 44 and 46.

[279] Yuri Shkuro, Uber. Personal Communication. (February 2017). Page 77.

[280] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. Page 98.

[281] Denis Zaytsev. Distributed Tracing – The Most Wanted And Missed Tool In The Micro-Service World. (April 2016). Retrieved July 2017 from `https://medium.com/@denis.zaytsev/distributed-tracing-the-most-wanted-and-missed-tool-in-the-micro-service-world-c2f3d7549c47`. Page 9.

[282] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. Pages 11 and 83.

[283] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. Pages 11 and 46.

[284] Jingwen Zhou, Zhenbang Chen, Haibo Mi, and Ji Wang. MTracer: A Trace-Oriented Monitoring Framework for Medium-Scale Distributed Systems. In *8th IEEE International Symposium on Service Oriented System Engineering (SOSE '14)*. Pages 8 and 46.