Universal Context Propagation for Distributed System Instrumentation

Jonathan Mace, Rodrigo Fonseca
Brown University

Abstract

Many tools for analyzing distributed systems propagate contexts along the execution paths of requests, tasks, and jobs, in order to correlate events across process, component and machine boundaries. There is a wide range of existing and proposed uses for these tools, which we call cross-cutting tools, such as tracing, debugging, taint propagation, provenance, auditing, and resource management, but few of them get deployed pervasively in large systems. When they do, they are brittle, hard to evolve, and cannot coexist with each other. While they use very different context metadata, the way they propagate the information alongside execution is the same. Nevertheless, in existing tools, these aspects are deeply intertwined, causing most of these problems.

In this paper, we propose a layered architecture for cross-cutting tools that separates concerns of system developers and tool developers, enabling independent instrumentation of systems, and the deployment and evolution of multiple such tools. At the heart of this layering is a general underlying format, *baggage contexts*, that enables the complete decoupling of system instrumentation for context propagation from tool logic. Baggage contexts make propagation opaque and general, while still maintaining correctness of the metadata under arbitrary concurrency and different data types. We demonstrate the practicality of the architecture with implementations in Java and Go, porting of several existing cross-cutting tools, and instrumenting existing distributed systems with all of them.

1. Introduction

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, and have enabled a large number of tools, both in research and in practice [37, 42, 48, 57, 58]. Of these, the most widespread are tracing tools [31, 39, 74, 85], which can record causality between logs and events across components by propagating request and event IDs. Beyond tracing, tenant IDs enable coordinated scheduling decisions across components [51, 82]; latency measurements enable tools to adapt to bottlenecks or processing delays [33, 73]; user tokens enable tools for auditing security policies [2, 79, 85]; causal histories enable cross-system consistency checking [50]; and more. We refer to this broad class of tools as *cross-cutting tools*, to stress their use beyond recording traces.[1]

Despite their demonstrated usefulness, organizations report that they struggle to deploy cross-cutting tools. One fundamental reason is that they require modifications to source code that touch nearly every component of distributed systems. But this is not the only reason. To see why, we can break up cross-cutting tools into two orthogonal components: first is the instrumentation of the system components to propagate context; second is the tool logic itself. This logic – what the metadata is, such as IDs, tags, or timings, and when it changes – depends on the tool, while the context propagation – through threads, RPCs, queues, etc. – only depends on the structure of the instrumented system and its concurrency.

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.

In this paper, we present the design and implementation of baggage contexts, an intermediate representation for cross-cutting tool contexts that enables the complete decoupling of system instrumentation for context propagation from cross-cutting tool logic. Our design bridges two objectives. First, baggage contexts are 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

[1] Prior work has also used the term *meta-applications* [29].

EuroSys '18, April 23–26, 2018, Porto, Portugal.

ACM ISBN 978-1-4503-5584-1/18/04.
http://dx.doi.org/10.1145/3190508.3190526

	Concerns	*Tasks*	*Abstractions*
Cross-Cutting Tool Developers (*e.g.* Zipkin, Pivot Tracing)	Cross-cutting tool semantics, information communicated	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 (§3.2)
Tracing Plane Developers (This paper)	Context behavior under concurrent execution (*i.e.* correct merging)	Tracing plane internals: BDL compiler, underlying context format, nesting, multiplexing, trimming	
System Developers (*e.g.* HDFS, Spark)	Execution boundaries and concurrency (*e.g.* queues, threadpools, RPCs)	Instrumentation: modify systems to propagate contexts alongside executions	Baggage contexts, five propagation operations (§3.1)

Table 1: The Tracing Plane provides abstractions to separate the concerns of different developers.

remaining efficient and extensible. Second, system instrumentation that propagates baggage contexts is completely agnostic to the interpretation of any data therein and independent of the semantics of any cross-cutting tool. 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. §4 outlines the properties of baggage contexts that address this challenge and enable an expressive and efficient implementation: idempotent, commutative, and associative merge; lazy resolution; and order preservation.

To fully benefit from our 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, which we call the *Tracing Plane*, with baggage contexts serving as a narrow waist, to enforce this separation of concerns. Table 1 summarizes the concerns of each developer group, and the abstractions the Tracing Plane exposes to them.

System developers can instrument their systems to opaquely propagate baggage contexts, independent of any cross-cutting tool. All that is required at this level is that they instrument their code using a set of five propagation operations: Branch, Join, Trim, Serialize, and Deserialize. These effectively encode the system developer's knowledge of the concurrency structure of their system. Because it is simple and independent of any cross-cutting tool, this instrumentation can be done without committing to specific cross-cutting tools at development time, and the costs of instrumentation can be amortized among many tools.

Tool developers, on the other hand, should not be concerned with details of propagation, but focus on the tool's logic. We expose to them the abstraction of execution-flow scoped variables, *i.e.*, variables that "follow the execution", maintaining correct values across arbitrary concurrency patterns. These variables are specified through a simple interface definition language which we call Baggage Definition Language, or BDL, and are compiled to encode different concurrent data types onto the baggage context representation.

Baggage contexts serve as the compilation target for BDL types, encapsulate their correct behavior under concurrency, and bridge the gap between the high-level execution-flow scoped variables and the five propagation operations that system developers use. By making instrumentation for propagation independent of tool logic, it can be done once, at development time, independently for each component. It becomes easier to deploy and evolve cross-cutting tools, and for multiple cross-cutting tools to co-exist.

To demonstrate the applicability of baggage contexts as a general tracing context format, and of the effectiveness of the Tracing Plane abstractions built around them, we have implemented and open-sourced prototypes in Java and in Go. We have implemented revised and extended versions of OpenTracing [60], Pivot Tracing [53], Retro [51], X-Trace [39], and Zipkin [65], which use BDL to define and interact with baggage contexts, plus two new cross-cutting tools for resource management and critical path analysis. Using the five propagation operations above, we have instrumented several open-source distributed systems to propagate baggage contexts, including Hadoop, HDFS, YARN and Spark, and the Sock Shop Microservices Demo [95].

The main contributions of this work as as follows:

- Design of baggage context, a 'narrow waist' that decouples context propagation from cross-cutting tool logic. Baggage contexts are simple, yet they are expressive enough to represent many distributed data types and preserve their correct merge semantics.
- Proposal of a layered architecture that separates concerns of system developers and cross-cutting tool developers, enabling independent instrumentation of systems, and the deployment and evolution of multiple cross-cutting tools.
- Definition of a Baggage Definition Language (BDL), which compiles nested concurrent data types onto baggage contexts, allowing cross-cutting tool developers to program with simple 'execution-flow scoped variables'.
- Implementation of the Tracing Plane stack in Java and Go
- Demonstration of the practicality and efficiency of the Tracing Plane through implementing several cross-cutting tools and deploying on multiple existing distributed systems.

2. Background and Motivation

Monitoring, understanding, and enforcing distributed system behavior is notoriously difficult [37, 48, 53, 59]. Symptoms can manifest in components far removed from the root cause of problems [43, 44], making the tools and abstractions useful for standalone programs – execution stacks, thread IDs, thread-local variables, debuggers, profilers, and many more – ineffective or inadequate for distributed systems [42]. Inevitably, to address these challenges distributed systems need the ability to correlate events at one point of the system with events that are meaningful from other parts of the system.

This observation has been addressed in a growing body of work 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. There is a wide range of such tools, which we call *cross-cutting tools*, with diverse goals and, correspondingly, a rich set of data types. Many systems propagate activity and request IDs for use in debugging and profiling, anomaly detection, resource accounting, or resource management [39, 46, 51, 79, 85]. Tracing tools propagate sampling decisions to bias traced requests towards underrepresented request types [46]. Taint tracking and DIFC propagate security labels as the system executes, warning of or prohibiting policy violations [35, 55, 99]. User tokens enable tools for auditing security policies [2, 79, 85] and identifying business flows [79]. Data provenance systems propagate information about the lineage of data as different components manipulate it [34, 54]. Tools for end-to-end latency, path profiling, and critical path analysis propagate partial latency measurements and information about execution paths and graphs [28, 33, 73, 90]. Metric-gathering systems propagate labels and query state so that downstream components can select, group, and filter statistics [41, 53, 79]. Recent work has proposed cross-cutting tools that propagate causal histories to validate cross-system consistency [50]. 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.

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

2.1 Anatomy of a Cross-Cutting Tool

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 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.

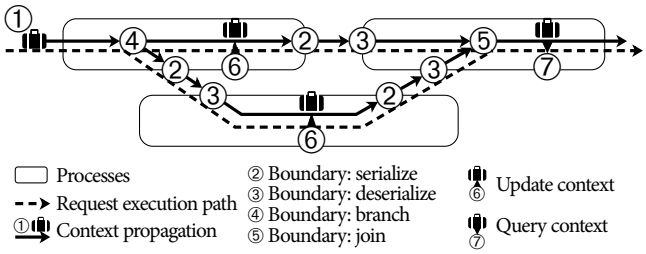


Fig. 1: Systems propagate the cross-cutting tool's context (①) along the execution path, including across process and machine boundaries (② ③ ④ ⑤). Cross-cutting tools update (⑥) and query (⑦) context values.

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.). A context comprises variables and data structures that the cross-cutting tool uses to observe and record causality. Examples include a single integer request ID [31, 51]; IDs and flags describing recent events [39, 63, 85]; and data structures like sets and maps [41, 53]. Context propagation requires small but non-trivial interventions within all distributed system components, to maintain and propagate contexts alongside executions. Examples include reading and writing contexts in RPC headers; storing in thread-local variables; and passing through thread pool queues and with continuations. This instrumentation is required at all execution boundaries, lest intermediary components discard context required later.

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.

Cross-Cutting Tool Logic. 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, distributed logging 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 [2, 85, 100]. 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 discussion.

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 request or tenant IDs [31, 51]. 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 [41, 53]; 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. In the general case we formalize contexts as state-based replicated objects with parallels to CRDTs [77], where communication between instances occurs only when their respective execution branches join (cf. §4.3).

2.2 Challenges

Developing and deploying cross-cutting tools on large distributed systems and across large organizations is challenging. One fundamental reason is the need for pervasive instrumentation to propagate contexts throughout all system comopnents. Researchers and practitioners consistently describe instrumentation as the most time consuming and difficult part of deploying a cross-cutting tool [32, 38, 39, 47, 76, 84]. 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 [85].

In many cases, missing or misaligned instrumentation will prematurely discard request contexts [12, 13, 16, 18, 26, 88]; on the other hand, failing to discard contexts enables them to linger and associate with the next request, causing *e.g.* distinct traces to merge [5] or resource consumption to be misattributed [51, 52]. Instrumentation effort also depends on system heterogeneity: the wider the variety of languages and platforms in use, the more effort required to do instrumentation [32, 49, 76].

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 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.

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. As shown in Table 1, the people who understand and care about these aspects are usually in different groups.

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, Zipkin omits merge rules [61, 62], leading to difficulties instrumenting queues [97], asynchronous executions [66, 67, 69] and capturing multiple-parent causality [68, 70]. HDFS and HBase developers encountered similar problems due to HTrace lacking rules to capture multiple-parent causality [17, 21, 23, 24]. 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 [25, 64, 89]. 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 [97], async [66, 67, 69], and several others [21, 23, 46, 68, 70].

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 [72]. An example of this is Pivot Tracing [53]; to propagate sets of tuples requires a complicated versioning scheme based on interval tree clocks [1].

Duplicated Effort 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 [75] has three distributed databases based on BigTable's design [30] and each database implemented its own an ad-hoc tracing system: Accumulo [3] developed CloudTrace [8]; HBase [15] developed HTrace [19, 20]; and Cassandra [9] developed QueryTracing [11]. Accumulo developers wanted CloudTrace to extend to the underlying distributed file system HDFS [4]; however, HDFS developers opted for compatibility with HBase and deployed HTrace instead [22]. As a result, to get visibility of HDFS, Accumulo developers replaced CloudTrace with HTrace [7]. Migrating Accumulo to HTrace meant updating instrumentation at 471 source code locations [7]. Similarly extensive changes were required to deploy Zipkin in Cassandra [10, 14, 96] and Phoenix [27].

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 [62], or requiring system-level changes to capture missing relationships required by more expressive tools [6, 17, 24].

3. Interfaces

Many of the problems in §2.2 stem from the coupling between cross-cutting tools and instrumentation, 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: to be expressive enough to support a wide range of different data types and cross-cutting tools; and to enable opaque system-level instrumentation. Using baggage contexts, system instrumentation is a one-time task, and new cross-cutting tools can be deployed without effecting changes to instrumentation. This section first outlines the interfaces for 1) system developers and 2) cross-cutting tool developers, which baggage contexts must bridge. We then describe the design and implementation of baggage contexts in §4 and §5.

3.1 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 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 2 summarizes these operations.

3.2 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. §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 2 outlines BDL declarations for five cross-cutting tools

```
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;
}
```

Fig. 2: BDL declarations for cross-cutting tools used in our evaluation (see §6.1 for a description).

that we revisit in our evaluation (see §6.1 for a description). The BDL format is similar in style and primitives to protocol buffers [93]; it also provides sets, maps, nested data structures, and several more elaborate data types based on CRDTs (cf. §5.6). 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.

3.3 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 authors of this paper) 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 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 bound-

Operation	*Execution Boundary*	*Operation Description (§3.1)*	*Atom Layer Implementation (§5.1)*
BRANCH	Execution splits into multiple branches (*e.g.* threads, concurrent RPCs)	Derive a context for each branch	Duplicate the baggage context without modification
JOIN	Concurrent execution branches join	Combine multiple contexts into one	LEXMERGE incoming baggage contexts from the joining branches
SERIALIZE	Send a network request	Serialize context to wire format	Write baggage atoms in order, each length-prefixed by a varint
DESERIALIZE	Receive a network request	Deserialize context from wire format	Read baggage atoms in order, each length-prefixed by a varint
TRIM	Anywhere with size constraints	Impose a size constraint on a context	Discard atoms from the baggage tail then append trim marker

Table 2: Five propagation operations for system instrumentation (§3.1) and their implementation by the atom layer (§5.1).

aries 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.

Meanwhile, cross-cutting tool developers implement cross-cutting tool logic. 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) 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.

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.

4. 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. §5 describes concrete details of our implementation.

4.1 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 §3.1. 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 a tenant ID [51] or a trace ID [46]. 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 merge(A, merge(A, B)) = merge(A, B), and likewise merge(merge(A, B), B) = merge(A, B).

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.

4.2 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.

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 §4.1 apply recursively to nested objects. §5.2 describes our implementation of this encoding strategy.

4.3 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 behavior and provide strong eventual consistency [77]. In the literature, there are CRDT implementations for a range of data types including sets, maps, registers, counters, and graphs [78]. Furthermore, baggage contexts themselves fulfill the definition of a state-based CRDT [77], as they have an idempotent, associative, and commutative merge function.

4.4 Layering Summary

The final contribution of this work 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 1), and makes it easy to implement and support the different features.

Figure 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 (§3.1). The *cross-cutting layer* simplifies the development and deployment of cross-cutting tools, by providing BDL to specify cross-cutting tool contexts, and

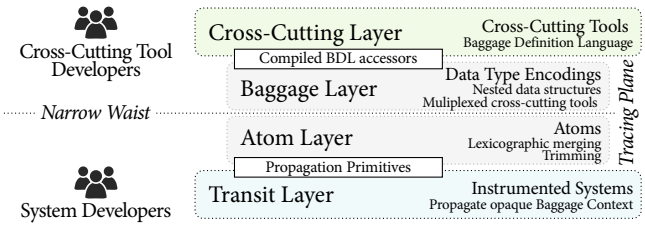


Fig. 3: Tracing Plane layered design.

BDL-compiled interfaces to access and manipulate data (§3.2). We bridge these layers with two internal layers. The atom layer corresponds to the core baggage context representation and implements the five propagation operations (§4.1 and §5.1). The baggage layer implements the encoding strategy for nested data structures and multiplexed cross-cutting tools (§4.2 and §5.2).

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 §5 are wholly encapsulated by the BDL compiler and baggage layer.

5. Implementation

We now present our implementation of baggage contexts. Our implementation is expressive, efficient, and enables 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. Zipkin is a open-source tool based on Google's Dapper [85]. 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 2). When spans end, Zipkin clients log them to a central database.

5.1 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 2 summarizes the five propagation operations for atoms. DESERIALIZE and SERIALIZE read and write atoms in order, varint-prefixed. BRANCH trivially duplicates atoms. TRIM truncates atoms and appends a special *trim marker*, which is just the zero-length atom; we discuss TRIM in §5.5. 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 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 4a‡). Notably however, if two atoms are found to be equal, only output them once. Figure 4b 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 4b†. Lexigoraphic merge satisfies the properties described in §4.1: it is idempotent, associative, commutative, lazy, and order preserving.

5.2 Atom Encodings (Cross-Cutting Layer)

As described in §4.4, 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[2] 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.* `00000000000000EA`. For ease of demonstration, we abbreviate data atoms by highlighting the `0` prefix and writing literal values, *e.g.* `0234`.

[2]Zipkin is described in §3.2. Figure 2 outlines Zipkin's BDL declaration.

```
1:  procedure LexCmp(x, y)                          ▷ atom comparison †
2:      for i ∈ [0 . . . min(x.length, y.length)] do
3:          if x[i] ≠ y[i] then return x[i] − y[i]
4:      return x.length − y.length
5:  procedure LexMerge(a, b)                        ▷ baggage merge ‡
6:      j ← 0;  k ← 0;  out ← []
7:      while not a.IsEmpty and not b.IsEmpty do
8:          cmp ←LexCmp(a.Head, b.Head)
9:          if cmp < 0 then out.Push(a.Pop)
10:         else if cmp > 0 then out.Push(b.Pop)
11:         else
12:             a.Pop
13:             out.Push(b.Pop)
14:     out.PushAll(a, b)
15:     return out
```

(a)Lexicographic atom comparison and baggage merge pseudo-code.

	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

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

Fig. 4: A baggage context is an array of *atoms*. Baggage contexts are merged using lexicographic comparison.

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 $>_{lex}$ 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 $>_{lex}$ 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`.

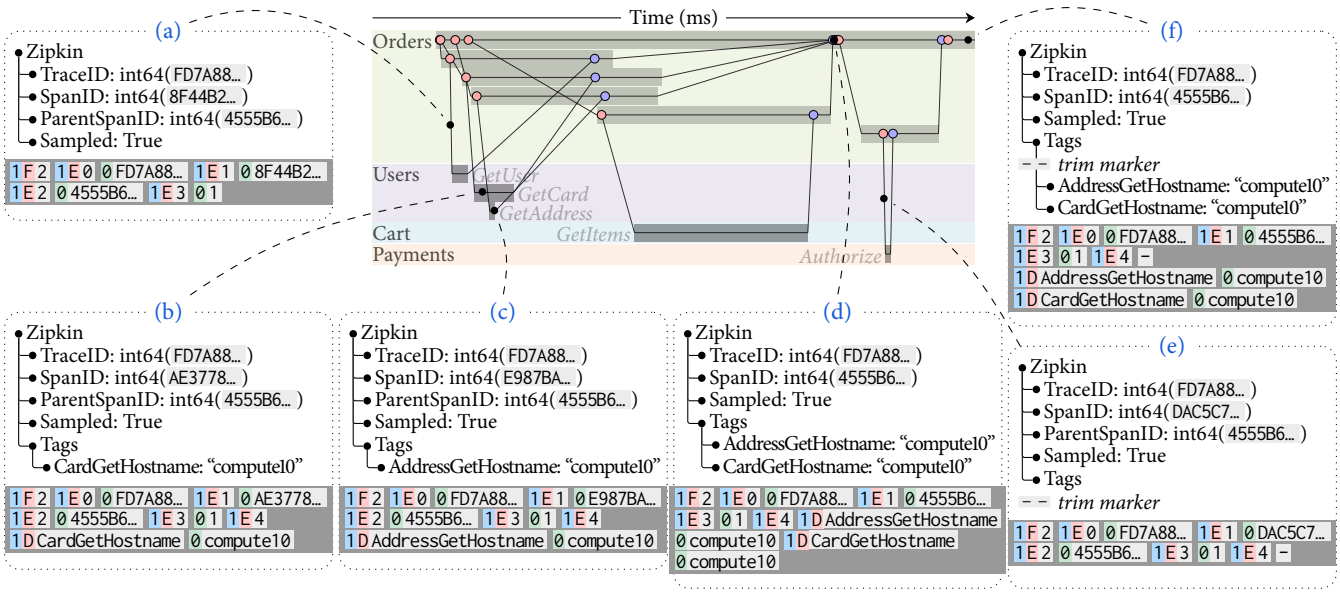


Fig. 5: Timeline of a request to order socks in the Sock Shop microservices demo [95], invoking Users, Cart, and Payments microservices. Rows represent threads; shaded bars represent time executing; lines indicate causality; and we highlight invocations of BRANCH (◉) and JOIN (◉). (b) and (c) illustrate 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.

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 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.

5.3 Example

To make these concepts concrete, we present an example using the Sock Shop microservices demo [95]. 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 [86] and Go Kit [91] 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 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. Figure 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.

5.4 Complex Data Types

Beyond the simple encodings described in §5.2, 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. §5.6). 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[3] present in the Go Zipkin implementation but not Java. For this demonstration, we modify the Go-based User microservice to add tags whenever GetAddress or GetCard is invoked.

[3] Zipkin tags are inspired by, and named after, Pivot Tracing's baggage [53]; we use the name tags instead of baggage to avoid overloaded terminology.

Maps encode each key-value pair as a header containing the key literal and a data atom containing the value. Figure 5b illustrates atoms after GetCard adds the tag `CardGetHostname=compute10`; similarly Figure 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 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 §6, but in general they are modest, particularly for fixed-width IDs. For example, excluding Tags, Zipkin's fields use 48 bytes.

5.5 Overflow

§3.1 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 [53,81]. 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 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.

5.6 Conflict-Free Replicated Data Types

As described in §4.3, 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 [78]. Current BDL data types with CRDT equivalents are counters, flags, sets, and maps. Furthermore, our baggage context implementation naturally provides sets and maps (cf. §5.4), 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 [78].

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 [53]. 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 [72]. 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.

6. Evaluation

§5.3 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)[83] and Spark [101] data analytics jobs running atop Hadoop YARN[94]. 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

6.1 Cross-Cutting Tools

In addition to Zipkin, described in §2, we implemented the following cross-cutting tools using the Tracing Plane. Figure 2 shows the BDL specification for each.

Retro Retro 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 [51]. For clarity, we configure Retro to only intercept disk API calls.

X-Trace X-Trace is an end-to-end tracing framework [39]; 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 is a dynamic instrumentation system for querying statistics about causally related events [53]. We reproduce Q7 from the paper, which propagates two pieces of information: the hostname of the client when initiating an HDFS operation; and the 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 (cf. Figure 2), to be flexible in experimenting with propagating different information.

6.2 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 §6.1.

6.2.1 Execution Patterns

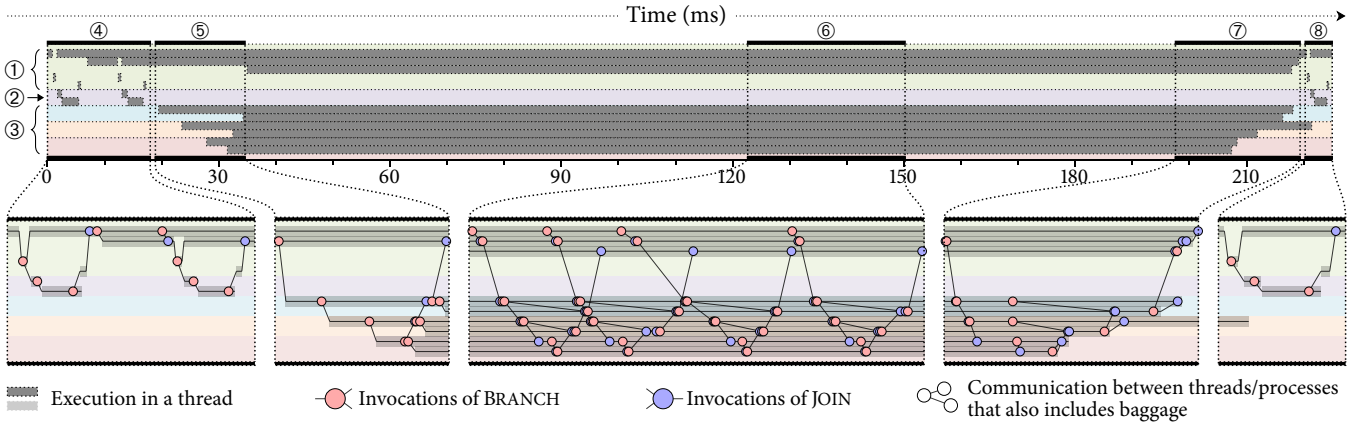
Figure 6a illustrates the end-to-end execution timeline of a 1MB HDFS write request, highlighting the BRANCH and JOIN behavior at several points. There are swimlanes for the client (①), NameNode (②), and three DataNodes (③). We highlight 5 phases: in ④, the client makes two RPCs to create the file and allocate a data block; in ⑤ the client sets up a streaming pipeline with three DataNodes; ⑥ zooms in on the streaming of 64kB 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 §5.3.

6.2.2 Cross-Cutting Tools

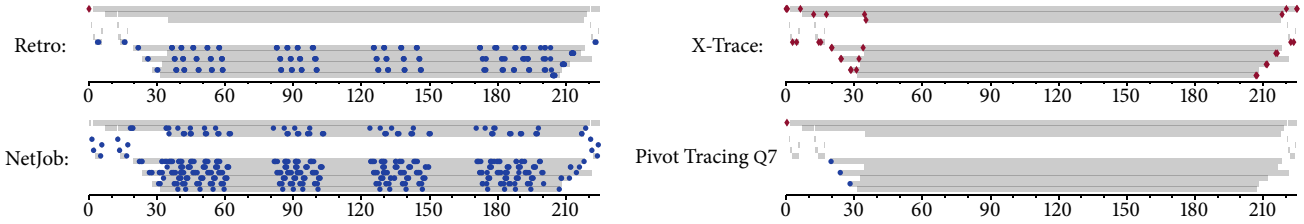
Figure 6b illustrates the places during request execution where each cross-cutting tool interacted with the request's baggage. Retro writes the tenant ID once and reads it on every disk operation. NetJob intercepts all network communication to check for a job ID, but because we were not running the request 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.

6.2.3 Baggage Overheads

Figure 7a 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

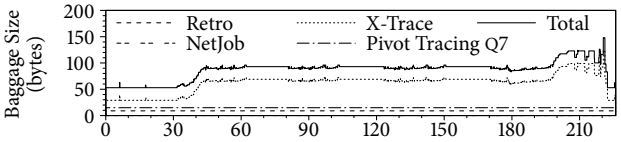


(a)Request timeline: rows represent threads; shading indicates execution; highlighted sections illustrate branch and join points during execution.

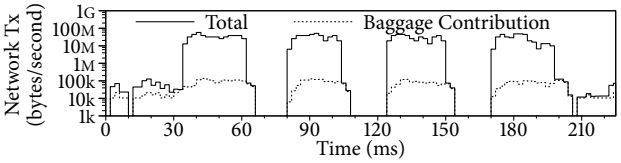


(b)Cross-cutting tools query (●) and update (◆) baggage values at various points during request execution.

Fig. 6: The end-to-end profile of a 1MB HDFS write request with four cross-cutting tools deployed. See §6.2 for a full description.



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



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

Fig. 7: Baggage overheads for Figure 6.

Figure 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 7b, 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.

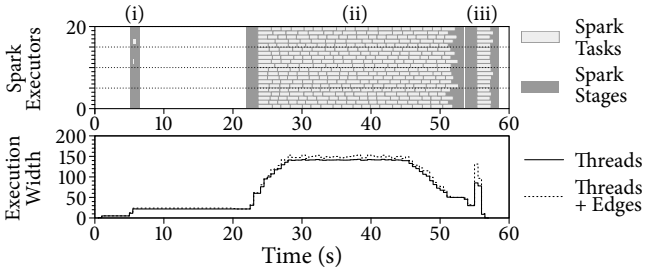
6.3 Cross-Cutting Tools at Scale

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 [40] 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 [92], selected by prior benchmarking work [36, 71], with the scale factor set to 100; that is, input data in HDFS is approximately 200GB uncompressed / 17GB compressed.

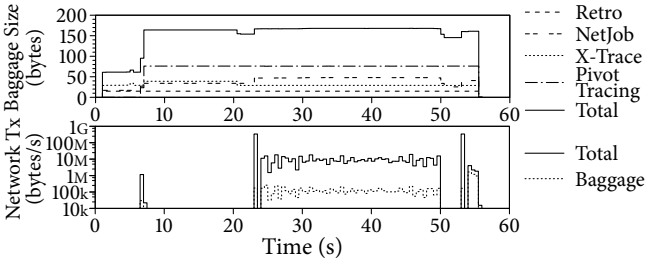
6.3.1 Overview of a Query

Figure 8a illustrates the execution of query 43 [45] (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.

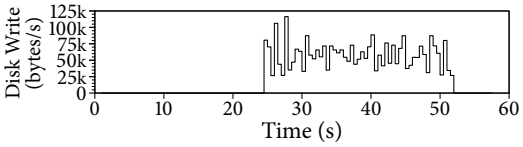
Figure 8b illustrates the average baggage size during query execution, which peaks at 167 bytes. Pivot Tracing Q7 and NetJob impose higher overheads than the HDFS request in §6.2: the job has many HDFS read requests, and Q7 updates the baggage in two places (client and NameNode) instead of the previous 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 Q43 execution (top) and execution width for the job across all system components (bottom). Edges account for in-flight network communication.



(b)Average baggage sizes for the cross-cutting tools during Q43 execution (top) and network overhead imposed (bottom).



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

Fig. 8: Execution of TPC-DS Q43 on a 25-node Spark cluster.

6.3.2 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. §5.6) 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 8c plots Spark's disk write over time; every task on every executor writes its output to disk. Figure 9a illustrates the growth in baggage over time. Once tasks start writing to disk (t = 25), the baggage 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 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 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.

6.3.3 Benchmark Results

Figure 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 they write to disk.

6.4 Experience Using Baggage Contexts

We found it easy to develop or update cross-cutting tools. 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.

In §6.3 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.

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.

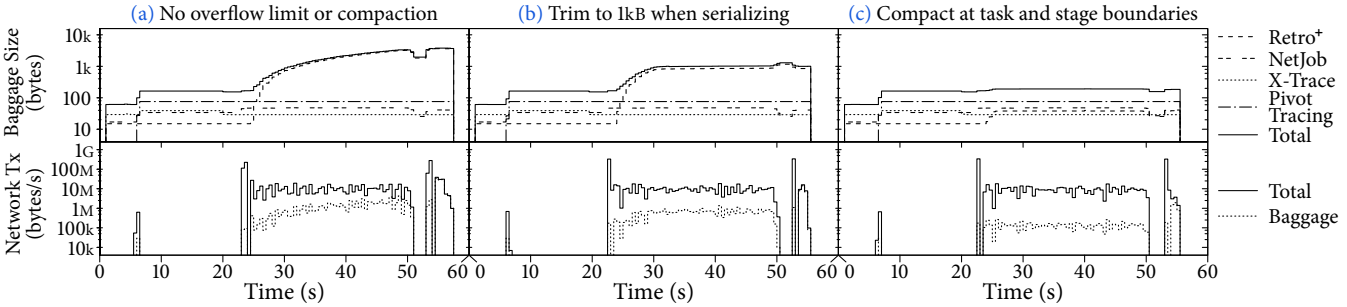


Fig. 9: Baggage size and network overhead for TPC-DS query 43 running on a 25-node Spark cluster. (a) Baggage for Retro+ grows to 4kB in size with no overflow configured; (b) Trimming baggage to 1kB reduces the overhead but sacrifices counter precision; (c) Compacting the counter at carefully chosen points during execution maintains correctness while substantially reducing size.

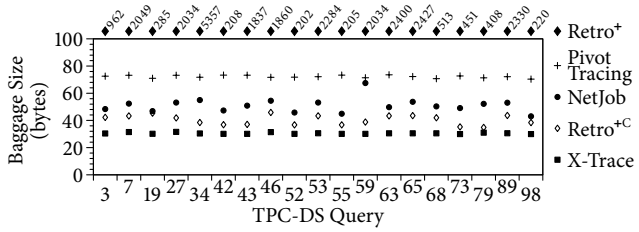


Fig. 10: Average baggage sizes for cross-cutting tools across 19 TPC-DS queries executed on a 25-node Spark cluster. Figures for Retro+ show with and without compaction.

Lazy resolution (§4.1) 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. §2.2).

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" [76]. However, baggage contexts could be propagated along other dimensions, such as through caches, using the same five propagation operations.

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.

7. Related Work

Throughout this paper we mention several cross-cutting tools and related instrumented distributed applications, and we complement the discussion here. Distributed systems tracing is useful for a range of tasks surrounding performance monitoring, anomaly detecting, and resource management; we refer to [76] for a detailed overview.

In the open-source community, Dapper's span model is the prevailing approach with numerous derivative implementations [20, 56, 65, 80, 87, 98]. As these tracing systems have matured, their community has encountered many of the challenges described in this paper [97], and recently motivated the OpenTracing effort to standardize the semantics of this class of cross-cutting tools [60]. Canopy [46] also identifies and addresses these challenges, by decoupling aspects of context propagation, instrumentation, and trace representation.

Pivot Tracing [53] introduced 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. In this work we generalize 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 [77, 78], and is inspired by the way in which IDLs such as protocol buffers [93] automate and simplify the tasks of marshalling, serializing, and transporting datastructures.

8. 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.

References

[1] Almeida, P. S., Baquero, C., and Fonte, V. Interval Tree Clocks: A Logical Clock for Dynamic Systems. In *12th International Conference On Principles Of Distributed Systems (OPODIS '08)*. (§2.2).

[2] Alshuqayran, N., Ali, N., and Evans, R. A Systematic Mapping Study in Microservice Architecture. In *9th IEEE International Conference on Service-Oriented Computing and Applications (SOCA '16)*. (§1, 2, and 2.1).

[3] Apache. Accumulo. Retrieved January 2017 from https://accumulo.apache.org/. (§2.2).

[4] Apache. ACCUMULO-1197: Pass Accumulo trace functionality through the DFSClient. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-1197. (§2.2).

[5] Apache. ACCUMULO-3725: Majc trace tacked onto minc trace. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-3725. (§2.2).

[6] Apache. ACCUMULO-3741: Reduce incompatibilities with htrace 3.2.0-incubating. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-3741. (§2.2).

[7] Apache. ACCUMULO-898: Look into replacing CloudTrace. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-898. (§2.2).

[8] Apache. Accumulo CloudTrace. Retrieved January 2017 from http://accumulo.apache.org/1.6/accumulo_user_manual.html#_tracing. (§2.2).

[9] Apache. Cassandra. Retrieved January 2017 from https://cassandra.apache.org/. (§2.2).

[10] Apache. CASSANDRA-10392: Allow Cassandra to trace to custom tracing implementations. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-10392. (§2.2).

[11] Apache. CASSANDRA-1123: Allow tracing query details. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-1123. (§2.2).

[12] 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. (§2.2).

[13] Apache. CASSANDRA-7657: Tracing doesn't finalize under load when it should. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-7657. (§2.2).

[14] Apache. CASSANDRA-8553: Add a key-value payload for third party usage. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-8553. (§2.2).

[15] Apache. HBase. Retrieved June 2016 from https://hbase.apache.org. (§2.2).

[16] Apache. HBASE-13077: BoundedCompletionService doesn't pass trace info to server. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-13077. (§2.2).

[17] 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. (§2.2).

[18] Apache. HBASE-15880: RpcClientImpl#tracedWriteRequest incorrectly closes HTrace span. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-14880. (§2.2).

[19] Apache. HBASE-6215: Per-request profiling. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-6215. (§2.2).

[20] Apache. HBASE-6449: Dapper like tracing. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-6449. (§2.2 and 7).

[21] 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. (§2.2).

[22] Apache. HDFS-5274: Add Tracing to HDFS. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-5274. (§2.2).

[23] Apache. HDFS-7054: Make DFSOutputStream tracing more fine-grained. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-7054. (§2.2).

[24] Apache. HDFS-9080: update htrace version to 4.0.1. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-9080. (§2.2).

[25] Apache. HDFS-9853: Ozone: Add container definitions. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-9853. (§2.2).

[26] 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. (§2.2).

[27] Apache. Phoenix 195: Zipkin. Retrieved January 2017 from https://github.com/apache/phoenix/pull/195. (§2.2).

[28] Chanda, A., Cox, A. L., and Zwaenepoel, W. Whodunit: Transactional Profiling for Multi-Tier Applications. In *2nd ACM European Conference on Computer Systems (EuroSys '07)*. (§2).

[29] Chanda, A., Elmeleegy, K., Cox, A. L., and Zwaenepoel, W. Causeway: Support for Controlling and Analyzing the Execution of Multi-tier Applications. In *6th ACM/IFIP/USENIX International Middleware Conference (Middleware '05)*. (§1).

[30] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*. (§2.2).

[31] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*. (§1 and 2.1).

[32] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. (§2.2).

[33] Chow, M., Veeraraghavan, K., Cafarella, M., and Flinn, J. DQBarge: Improving Data-Quality Tradeoffs in Large-Scale Internet Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. (§1 and 2).

[34] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium (Security '11)*. (§2).

[35] Enck, W., Gilbert, P., Chun, B.-G., Cox, L., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. (§2).

[36] Erickson, J., Kornacker, M., and Kumar, D. 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/. (§6.3).

[37] Esposito, C., Castiglione, A., and Choo, K.-K. R. Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing 3*, 5 (2016), 10–14. (§1 and 2).

[38] Fonseca, R., Freedman, M. J., and Porter, G. Experiences with Tracing Causality in Networked Services. In *2010 USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN '10)*. (§2.2).

[39] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*. (§1, 1, 2, 2.1, 2.2, and 6.1).

[40] Google. Compute Engine. Retrieved January 2017 from https://cloud.google.com/compute/. (§6.3).

[41] Google. gRPC/Census. Retrieved January 2017 from https://goo.gl/iEqlqH. (§2 and 2.1).

[42] Gould, O. 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/. (§1 and 2).

[43] Guo, Z., McDirmid, S., Yang, M., Zhuang, L., Zhang, P., Luo, Y., Bergan, T., Musuvathi, M., Zhang, Z., and Zhou, L. Failure Recovery: When the Cure Is Worse Than the Disease. In *14th USENIX Workshop on Hot Topics in Operating Systems (HotOS '13)*. (§2).

[44] Heath, M. 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/. (§2).

[45] Impala TPC-DS Kit. TPC-DS Query 43. Retrieved January 2017 from https://github.com/cloudera/impala-tpcds-kit/blob/c5d32ae55a5259dd081bf4546bb650b2a3d668de/queries/q43.sql. (§6.3.1).

[46] Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., Vekataraman, V., Veeraraghavan, K., and Song, Y. J. Canopy: An End-to-End Performance Tracing And Analysis System. In *26th ACM Symposium on Operating Systems Principles (SOSP '17)*. (§2, 2.2, 4.1, and 7).

[47] Karumuri, S. PinTrace: Distributed Tracing at Pinterest. (August 2016). Retrieved July 2017 from https://www.slideshare.net/mansu/pintrace-advanced-aws-meetup. (§2.2).

[48] Killalea, T. The Hidden Dividends of Microservices. *Communications of the ACM 59*, 8 (2016), 42–45. (§1 and 2).

[49] Lightstep. Lightstep. Retrieved January 2017 from http://lightstep.com. (§2.2).

[50] Loff, J., Porto, D., Baquero, C., Garcia, J., Preguiça, N., and Rodrigues, R. Transparent Cross-System Consistency. In *3rd International Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '17)*. (§1 and 2).

[51] Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. (§1, 1, 2, 2.1, 2.2, 4.1, and 6.1).

[52] Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. Towards General-Purpose Resource Management in Shared Cloud Services. In *10th USENIX Workshop on Hot Topics in System Dependability (HotDep '14)*. (§2.2).

[53] Mace, J., Roelke, R., and Fonseca, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*. (§1, 2, 2.1, 2.2, 3, 5.5, 5.6, 6.1, and 7).

[54] Muniswamy-Reddy, K.-K., Macko, P., and Seltzer, M. I. Provenance for the Cloud. In *8th USENIX Conference on File and Storage Technologies (FAST '10)*. (§2).

[55] Myers, A. C., and Liskov, B. A Decentralized Model for Information Flow Control. In *16th ACM Symposium on Operating Systems Principles (SOSP '97)*. (§2).

[56] Naver. Pinpoint. Retrieved January 2017 from https://github.com/naver/pinpoint. (§7).

[57] Netflix. Netflix Open Source Software. Retrieved January 2017 from http://netflix.github.io/. (§1).

[58] Newman, S. *Building Microservices*. O'Reilly Media, Inc., 2015. (§1).

[59] Oliner, A., Ganapathi, A., and Xu, W. Advances and Challenges in Log Analysis. *Communications of the ACM 55*, 2 (2012), 55–61. (§2).

[60] OpenTracing. OpenTracing. Retrieved January 2017 from http://opentracing.io/. (§1 and 7).

[61] OpenTracing. OpenTracing 28: Non-RPC Spans and Mapping to Multiple Parents. Retrieved January 2017 from https://github.com/opentracing/opentracing.io/issues/28. (§2.2).

[62] OpenTracing. Specification 5: Non-RPC Spans and Mapping to Multiple Parents. Retrieved February 2017 from https://github.com/opentracing/specification/issues/5. (§2.2).

[63] OpenZipkin. B3-Propagation. Retrieved January 2017 from https://github.com/openzipkin/b3-propagation. (§2.1).

[64] 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. (§2.2).

[65] OpenZipkin. Zipkin. Retrieved July 2017 from http://zipkin.io/. (§1 and 7).

[66] OpenZipkin. Zipkin 1189: Representing an asynchronous span in Zipkin. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1189. (§2.2).

[67] OpenZipkin. Zipkin 1243: Support async spans. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1243. (§2.2).

[68] OpenZipkin. Zipkin 1244: Multiple parents aka Linked traces. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1244. (§2.2).

[69] OpenZipkin. Zipkin 925: How to track async spans? Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/925. (§2.2).

[70] OpenZipkin. Zipkin 939: Zipkin v2 span model. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/939. (§2.2).

[71] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., and Chun, B.-G. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. (§6.3).

[72] Parker, D. S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B. J., Walton, E., Chow, J. M., Edwards, D., Kiser, S., and Kline, C. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, 3 (1983), 240–247. (§2.2 and 5.6).

[73] Ravindranath, L., Padhye, J., Mahajan, R., and Balakrishnan, H. Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. (§1 and 2).

[74] Reynolds, P., Killian, C. E., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: Detecting the Unexpected in Distributed Systems. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*. (§1).

[75] Roman, J. The Hadoop Ecosystem Table. Retrieved January 2017 from https://hadoopecosystemtable.github.io/. (§2.2).

[76] Sambasivan, R. R., Shafer, I., Mace, J., Sigelman, B. H., Fonseca, R., and Ganger, G. R. Principled Workflow-Centric Tracing of Distributed Systems. In *7th ACM Symposium on Cloud Computing (SOCC '16)*. (§2.2, 6.4, and 7).

[77] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '11)*. (§2.1, 4.3, and 7).

[78] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Technical Report, Inria–Centre Paris-Rocquencourt; INRIA, 2011. (§4.3, 5.6, and 7).

[79] Shkuro, Y. Baggage Propagation at Uber. (September 2017). Retrieved October 2017 from https://github.com/TraceContext/tracecontext-spec/issues/13#issuecomment-330094227. (§1 and 2).

[80] Shkuro, Y. Evolving Distributed Tracing at Uber Engineering. (February 2017). Retrieved July 2017 from https://eng.uber.com/distributed-tracing/. (§7).

[81] Shkuro, Y. Jaeger #373: Baggage Whitelisting. (September 2017). Retrieved October 2017 from https://github.com/jaegertracing/jaeger/issues/373. (§5.5).

[82] Shue, D., Freedman, M. J., and Shaikh, A. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. (§1).

[83] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. (§6).

[84] Sigelman, B. H. Towards Turnkey Distributed Tracing. (June 2016). Retrieved January 2017 from https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736. (§2.2).

[85] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report, Google, 2010. (§1, 2, 2.1, 2.2, and 5).

[86] Spring. Spring Cloud Sleuth. Retrieved October 2017 from http://projects.spring.io/spring-cloud/. (§5.3).

[87] Spring. Spring Cloud Sleuth. Retrieved January 2017 from http://cloud.spring.io/spring-cloud-sleuth/. (§7).

[88] 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. (§2.2).

[89] 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. (§2.2).

[90] Sun, H. General Baggage Model for End-to-End Tracing and Its Application on Critical Path Analysis. M.Sc. Thesis, Brown University, 2016. (§2).

[91] The Go Blog. Go kit: A toolkit for microservices. Retrieved October 2017 from https://gokit.io/. (§5.3).

[92] 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. (§6.3).

[93] Varda, K. Protocol Buffers: Google's Data Interchange Format. (July 2008). Retrieved January 2017 from https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html. (§3.2 and 7).

[94] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *4th ACM Symposium on Cloud Computing (SoCC '13)*. (§6).

[95] Weaveworks, and Container Solutions. Sock shop: A microservices demo application. Retrieved October 2017 from https://microservices-demo.github.io. (§1, 5, and 5.3).

[96] Wever, M. S. 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. (§2.2).

[97] Workgroup, D. T. Tracing Workshop. (February 2017). Retrieved February 2017 from https://goo.gl/2WKjhR. (§2.2 and 7).

[98] Wright, P. 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. (§7).

[99] Yan, L.-K., and Yin, H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *21st USENIX Security Symposium (Security '12)*. (§2).

[100] Yuri Shkuro, Uber. Personal Communication. (February 2017). (§2.1).

[101] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. (§6).