Online Reconstruction of Structural Information from Datacenter Logs

Zaheer Chothia John Liagouris Desislava Dimitrova Timothy Roscoe

Systems Group, Department of Computer Science, ETH Zürich *

firstname.lastname@inf.ethz.ch

Abstract

Well-run datacenter application architectures are heavily instrumented to provide detailed traces of messages and remote invocations. Reconstructing user sessions, call graphs, transaction trees, and other structural information from these messages, a process known as *sessionization*, is the foundation for a variety of diagnostic, profiling, and monitoring tasks essential to the operation of the datacenter.

We present the design and implementation of a system which processes log streams at gigabits per second and reconstructs user sessions comprising millions of transactions per second in real time with modest compute resources, while dealing with clock skew, message loss, and other real-world phenomena that make such a task challenging. Our system is based on the Timely Dataflow framework for low latency, data-parallel computation, and we demonstrate its utility with a number of use-cases and traces from a large, operational, mission-critical enterprise data center.

CCS Concepts • **Applied computing → Enterprise data management**; *Data centers*; *Enterprise computing infrastructures*

Keywords Sessionization; Trace Trees; Resource Attribution; Streaming Log Analytics; Data Parallelism

1. Introduction

This paper describes TS, a system for recovering structural information (sessions, spans, call graphs, transaction trees, etc.) from a large datacenter logging infrastructure in real time, with low latency, using only modest computing resources.

Reconstructing such information from traces is a crucial first step in many common datacenter management and diagnostic tasks, and application middleware is often instrumented to facilitate such recovery by generating appropriate log records. Such logs, however, are generally large (often terabytes of data per hour for a reasonably-sized datacenter), and so most existing approaches [19, 21, 24, 35] work by applying parallel computing techniques to offline logs.

TS processes complete traces online, in real time. This allows it to support all the functionality of existing offline analyses, but also much more: online operation is a fundamental requirement for continuous monitoring of a datacenter and rapid, interactive diagnosis of problems as they happen. A first contribution of this paper is to show that not only is this real-time functionality useful, but it is practical: we show in Section 5 that on a single machine, TS can process 500MB/s of log traffic from over a thousand log servers in a large, enterprise datacenter, and its architecture distributes naturally onto multiple machines with little overhead.

Reconstructing datacenter application sessions and transactions from individual log records is an aggregation problem over distributed streams, so a natural approach is to use a general-purpose stream processing engine. This turns out *not* to be practical: we show how a state-of-the-art stream processor with built-in operators for such aggregations (Flink [15]) can only process a small fraction of the traffic that TS can handle in real time, with the same CPU and memory resources, and generates results with 71× higher latency (1493 *msecs* vs 21 *msecs*) with respect to the best performance of both systems on a commodity machine.

The traditional high-performance alternative to general-purpose systems is custom code. However, the problem TS solves is sufficiently complex that engineering a solution from scratch would be a prohibitive engineering task. Instead, TS is built on *Timely Dataflow* [5, 36]. A further contribution of this paper is to show how, by exploiting this general framework, TS admits a simple, concise implementation in 1770 lines of code, while at the same time seamlessly integrating with management applications that exploit the session and transaction information generated by TS online: transaction clustering,

* This work was supported by Amadeus SA, Google, and the Swiss National Science Foundation.

EuroSys '17 April 23–26, 2017, Belgrade, Serbia

© 2017 ACM. ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/10.1145/3064176.3064195

call-graph pattern extraction, and others we demonstrate in Section 5.2.

Reconstruction of structural information from logs is only a small part of the broad problem of understanding the dynamics of a datacenter. However, we argue it is a fundamental building block for further analyses, and being able to perform such reconstruction efficiently is a *sine qua non* for more elaborate online monitoring and diagnostics. We elaborate on this argument, and discuss the problem itself in more detail, in the following section.

2. The Reconstruction Problem

We now define the problem that TS addresses in more detail. At a high level, modern enterprise datacenters run a range of application services on behalf of external clients, which submit *requests* to an application. Not only are individual applications distributed in nature, they also rely on a further range of shared services within the datacenter, which are also distributed, for example in a so-called "Service-oriented Architecture". Each request or user *session* therefore consists of a distributed call tree of invocations.

Understanding the dynamics of this request workload, and how it is serviced by the datacenter as a whole, is a foundation for most of the operations and management tasks in the datacenter. Data centers undergo constant evolution, with shifting workload patterns and infrastructure reconfiguration hundreds of times throughout the day. Operations teams are faced with the hurdle of understanding this complex and layered stack in a timely fashion – they require rich, fresh information about the state of the datacenter.

A concrete example with numbers will help to illustrate the general challenge. The real-world datacenter which inspired this work (and a snapshot of whose workload we use to evaluate TS in Section 5) is operated by a major provider of IT services to the travel industry. It consists of roughly 5500[1] physical machines (many running multiple virtual machines), supporting about 2500 application instances, about 13,000 service instances. Traditionally, this datacenter has serviced a workload from external clients (airlines, travel websites, etc.) which resembles an Online Transaction Processing (OLTP) model, though increasingly analytics (OLAP) queries are deployed as well. The system as a whole sees a few hundred thousand external requests each second, which translate into multiple internal RPC calls. The datacenter must cope with rapid workload changes (for example, due to natural events like hurricanes) and configuration changes: new services, capacity, etc.

2.1 Logging infrastructures

The basis for understanding the dynamics of a datacenter is logging, and so many datacenters instrument their applications (and middleware) with functionality to emit log records when messages are sent and/or received by each service or application process. The basic principle is to assign a unique ID to a request on its entry to the system and propagate (and add to) this metadata whenever the request is passed between components. This leads to records of this form:

```
Time: 2015/09/01 10:03:38.599859
Session ID: XKSHSKCBA53U088FXGE7LD8
Transaction ID: 26-3-11-5-1
```

A common task then is to relate all pieces of work done in individual components back to their originating request or tenant (*resource attribution*). By combining the correlators with other fields present in the logs, a detailed representation can be re-built which contains all activities in the workflow along with structure relating the individual facts.

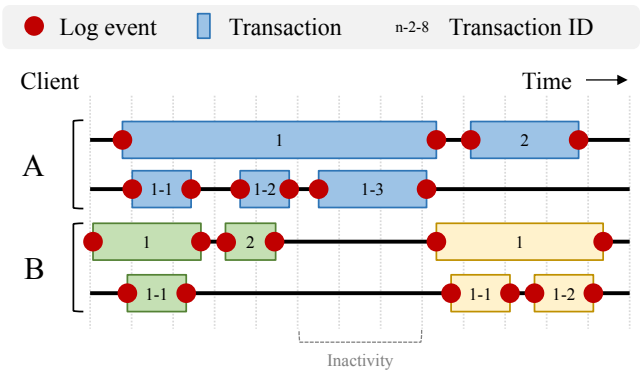


Figure 1: Hierarchical application tracing. Three sessions are shown, two from the same user.

Consider the example in Figure 1, where two clients are issuing requests. A transaction spans a range covering its constituent sub-requests. During a session, the client may invoke any number of services, each of these requests initiating a *root span* which can trigger an arbitrary number of additional nested spans. This leads to a natural hierarchical choice of log record identifiers which reflect the nesting. Red dots in the figure indicate log records that mark the start and end of individual spans, which are usually (but not always) fully contained within their parent.

Our example datacenter generates considerable logging traffic of this form – about 5Gb/s of network bandwidth on average, or about 50TB of log data per day. Log generation itself is spread over about 1300 distinct log servers.

2.2 Sessionization

The reconstruction process, known as *sessionization* [20, 25], transforms a (distributed) stream of these log records into a sequence of trees reflecting the workflow of each request and session. Several design choices exist as to how to do this, based on what assumptions the logging infrastructure makes [20, 25, 32], but broadly there are two key steps. First, log records are grouped into sessions using unique IDs, and sessions are marked as "closed" based, if necessary, on time-based windows of inactivity; and second, each closed session

[1] Figures are approximate for confidentiality reasons.

is converted to a *trace tree* which aggregates individual inter-application transactions or *spans* into a complete operation.

An important design choice is when and how to declare a session "closed" (completed). *Time-oriented* sessionization declares the session closed once a given period has elapsed with no intervening session activity. In contrast, *navigation-oriented* sessionization relies on explicit termination records in the log. In theory, navigation-oriented approaches are easier to implement. In practice, a combination of both is used, since unreliable log servers cannot guarantee to presence of a termination record for every session.

Offline sessionization, using log files on disk, is for the most part a simple aggregation operation, concisely expressible in a MapReduce-like system (e.g. [31]): The map function hashes records based on session ID, and the reduce does the work of constructing the tree.

Online sessionization is harder. Data arrives continually, and the streaming computation must buffer log records, track inactivity windows for each session, and emit the reconstructed trace tree once the session is closed. Note that since an online sessionizer has an inherently limited horizon to look ahead in the log stream, it will produce different output to an offline sessionizer.

One example is with sessions that are prematurely terminated and later exhibit a renewal of activity after being idle; an initial trace tree will be constructed containing all activity accumulated until the inactivity period expires and a second session window will be re-opened with the later activity. As we observe, online sessionization results in fragmentation of the trace tree into multiple disjoint parts whereas offline grouping produces a single and complete user session.

2.3 Further challenges

In addition to data volume, online sessionization faces a number of other technical challenges (some of which are shared with the easier offline case).

Reordered logs: Logging servers typically buffer records and flush them in batches. This, and the use of many distributed log servers, results in log records being reordered in the stream. Assuming log servers have synchronized clocks (see below), this reordering can be bounded in time and buffering is used to restore the chronological record order.

Data burstiness: Batching also causes data to arrive in bursts, which can leave the processing CPUs idle in some periods but overloaded in others. This can be mitigated at the expense of further buffer space, and as we show in Section 5, memory footprint is a concern when using stream processing engines to perform sessionization.

Incomplete logs: Log records can go missing for a variety of reasons: failure of servers, lost packets, transient overload, and software bugs. The latter are endemic in a large enterprise with tens of thousands of software developers, and a constantly-evolving code base. Incomplete logs do not block the sessionization computation, but they result in trees with missing nodes which can introduce errors into downstream computations. In some cases, the missing nodes can be inferred – transaction ID of `2-10` implies there is a root transaction `2` and nine other siblings. In general, however, the correct policy for handling missing records (detectable or not) depends on the application.

In addition to these, there are further challenges that TS only partially addresses, or does not yet address.

Clock desynchronization: Messages may appear to be received before they were originally sent or parent transactions start after their children due to clock skew between different machines. In our particular use-case, such cases seem to be rare, but nevertheless do occur and cause anomalies in the output. In the current version of TS, we use timestamps taken using the local system clock on the producer, but we assume these clocks are synchronized. A natural extension is to incorporate a time resynchronization protocol (e.g. [45]) into the sessionization process to detect and correct for clocks which are not synchronized.

Very long sessions: A session can be reconstructed and closed only after all log messages belonging to the session are collected. It follows naturally that long sessions have higher memory resource requirements, which the system must handle without swapping.

An additional challenge is how to set appropriate inactivity timeouts for deciding when to close sessions. Too short a timeout splits sessions incorrectly, while too long a timeout results in increased latency for the session tree. This is a problem for any logging protocol (due to log record loss), but is particularly acute for those with no explicit end-of-session markers. One way to finesse this trade-off is a watermark scheme and incremental processing model as proposed by the Dataflow model in [6]. This would eliminate the waiting period to close a session and allow to inspect partially-reconstructed trees which, despite being incomplete, provide faster feedback on the session state.

3. TS Architecture

TS is designed to interface with existing datacenter logging infrastructure, and ingest logs in real time from a collection of *logging servers* which emit hierarchically-labelled log records along the lines of Dapper [49] and Magpie [10]. The tracing infrastructure and log collection are considered external to TS, which simply expects parallel streams of logs as input to the engine.

Each log record contains the local system time where it was generated, the identity of the machine or process, and an application-specific payload. As discussed in Section 2, we currently assume these timestamps are sufficiently synchronized to be taken at face value when reconstructing sessions.

It is common practice to decouple applications from logging by buffering traces locally on the producer and

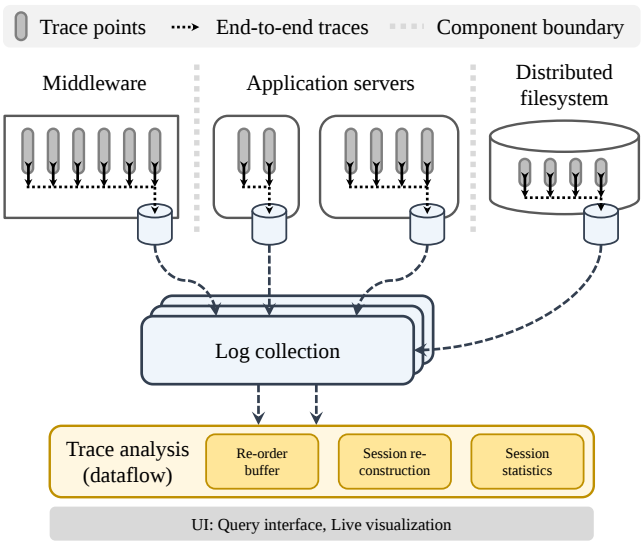


Figure 2: TS architecture

propagating log records to a set of dedicated servers in the background. Thus, we assume that all log records relevant to a given session arrive with bounded delay at TS, but can arrive re-ordered and from different logging servers.

TS is a logically centralized system, but parallelizes across multiple machines in a cluster. Figure 2 shows the main components of the architecture, and its context in the rest of the datacenter. We now discuss each component of Figure 2 in detail.

Re-order buffers: The re-order buffers (one for each TS "worker", in Timely Dataflow terminology) are responsible for buffering and sorting the input logs based on the indicated event time. Due to the distributed log sources, the arrival of an event to the system can be delayed, resulting in an arrival order that is different from the actual order the events occurred in the production system.

The size of the re-order buffer determines how tolerant TS is to the late arrival of log records. Anecdotally, our experience so far has been that when related events in real logging infrastructures arrive out of order, their arrival times at TS are actually quite tightly bounded[2], and so sessions can be reconstructed accurately using relatively small re-order buffers. In general, the larger the re-order buffers, the more accurate the results of sessionization, and we investigate this trade-off in Section 5.

Session reconstruction: During sessionization we need some indication of when an active session is closed so that the reconstructed trees can be emitted to allow reclamation of the system state. As we discussed in Section 2, the logs produced by datacenter applications record all session activities but TS cannot rely on an marker to denote the end of a session.

[2] Note that this is unrelated to skew in source clocks.

TS's "flush on inactivity" approach closes sessions after a chosen interval has elapsed during which no messages have been received. This imposes a fixed latency penalty on all session reconstructions (i.e. timeout is the norm rather than the exception). A large timeout will penalize short-lived sessions because they are only emitted after the full timeout has elapsed. The choice of the timeout is dictated by the longest interval between subsequent annotations of a session, or in other words the maximum inter-arrival time, and this is a property of the datacenter workload. In Section 5 we present statistics on session activity observed in a trace from our particular datacenter example. Whilst this penalty cannot be eliminated, the delay can be adjusted to be tolerable with a single runtime parameter.

An alternative approach would be to create multi-versioned sessions. In this scenario, new messages can arrive for a session at any time and changes are propagated downstream to subsequent calculations immediately. In principle, this may require recording the entire history of all sessions unless there is some way to deduce that a session is permanently closed. Moreover, permitting intermediate results which can be later retracted requires all downstream operators (subscribers) to support the same incremental computation model over changed inputs, for example recalculation using Differential Dataflow [37].

Session-based statistics: Most of the work of reconstructing sessions, and indeed the subsequent statistical processing of such session information which we demonstrate in Section 5.2, is built from a basic library of operators that can be used to reconstruct transaction trees, gather histograms and percentiles for various session characteristics (e.g., session timespans, inactivity periods, etc.), perform transaction clustering, and mine communication patterns. All these tasks are based on the output of sessionization, and they could be easily composed to form more complex tasks [28, 42, 44, 52].

TS derives much of its flexibility, and performance, from using a common substrate for sessionization and subsequent statistical operators, namely Timely Dataflow.

Timely Dataflow [36] is a technique for executing distributed and data-parallel computations over streaming inputs, first implemented in the Naiad system [41]. It is based on the dataflow computational model: a program is represented as a directed, possibly-cyclic graph which reflects the sequence of operators forming the program's workflow. Operators are the nodes in the graph and consume inputs and produce new outputs; these are connected by channels (edges) along which data and control messages are exchanged. Timely Dataflow is not a contribution of this paper, but we describe it briefly here for completeness.

In contrast to many comparable systems (e.g. [31, 53]) that process arrivals in micro-batches, TS is purely *event-driven* (similarly to [33, 41, 50]), meaning that data can be ingested and processed on a record-at-a-time basis.

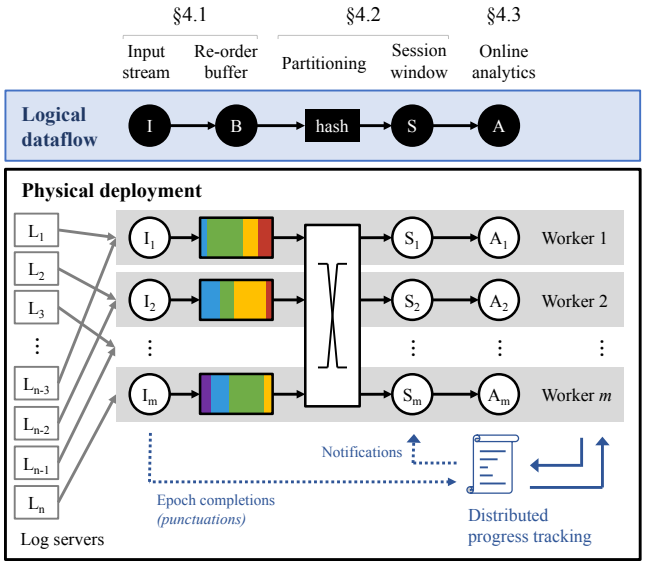


Figure 3: In Timely Dataflow's programming model a query is expressed by composing operators and channels into a logical graph (shown above). Once deployed, all participating workers instantiate a copy of the dataflow and are responsible for processing a partition of the streaming inputs (depicted below).

The dataflow modelling approach is a natural fit to streaming program design – a programmer chains together operators to construct the workflow. An illustration of the mapping between the workflow design and its execution as performed by Timely Dataflow is shown in Figure 3.

Expressing computations: A program built with Timely Dataflow is made up of two elements: an 'input' function that drives the input stream (read data, advance time) and the set of data-parallel operators which achieve the desired workflow (computation). In our case the workflow is the sessionization task as well as the computations for the session-based statistics. These connect to the input function which in turn interfaces with the log replayer. Let us look at how timely Dataflow supports the creation of workflows. There is a minimal set of default operators to filter and transform data. There is also a generic construct for unary- and binary-shaped operators and the built-in operators are written as a library without any special system privileges. For use cases requiring custom processing logic, as with sessionization, this can be achieved by supplying a user-defined functional logic, something we explain in more detail in the following section.

By building on top of a general-purpose dataflow model and using standardized windowing operators, simple tasks can be easily composed into more complex ones, resulting into larger applications. A reusable implementation of basic operators like sessionization plays an even more important role since it is a precursor step for most tasks which require the flow of requests in the datacenter and their context. Examples of such applications include anomaly detection, policy checking and identification of RPC deadlocks.

Runtime execution: In the context of multiple workers, the dataflow graph is replicated at each worker and logical exchange edges are realised by all-to-all channels between the workers. Data is moved along those edges encapsulated in messages tagged with a logical timestamp, usually an integer value reflecting its actual event time. Within a worker, the operators run on a single thread and share execution time with no pre-emption.

To support efficient analytics in real time, the whole data processing pipeline is *push-based*. The computation proceeds in steps and at each step operators are involved in round-robin fashion; new inputs are pushed all the way down to the operators, and results are produced as tuples flow in, without the need to manually issue queries or requests for updates. As result, the system allows distributed input of millions of records per second and produces outputs at millisecond-scale latency.

Progress tracking: A notable and distinguishing feature of Timely Dataflow is its distributed and asynchronous progress-tracking protocol [5]. This key component is responsible for bookkeeping of all outstanding messages and unexercised capabilities throughout the dataflow graph (i.e. timestamps which could potentially still emit outputs or pending notifications). What makes it special is that dataflow operators can consume and produce records without barriers or global coordination. Workers periodically exchange progress statements about local work and these are accumulated into a global view of progress. Once all messages and capabilities for a given epoch have been depleted, a *notification* signalling the completeness of an epoch is generated. Notifications allows data from different epochs to be in-flight concurrently, while operators can identify closed epochs.

4. System Implementation

In this section we detail the implementation choices and the flow of data in our system. At first we elaborate on how input streams are fed into the system, with a focus on the re-order buffer used for handling out-of-order events. We then explain the sessionization task as a data-parallel window operator. Finally, we describe how individual analytic tasks can be reused into more complex ones within our general-purpose dataflow approach.

4.1 Input Ingestion

Here we explain how arriving records are grouped by time and how completion of arrivals is signalled.

Time granularity. Epochs are logical timestamps used to track flow of data and communicate progress in the dataflow graph. An epoch is any partially-ordered coordinate (in our case, an integer) and is used to divide the input stream into disjoint time windows. The choice of how this coordinate

is defined has wide-reaching consequences on window operators, execution efficiency and synchronization between workers. For example, the amount of progress traffic grows in proportion to the number of outstanding epochs and, in addition, overly fine-grained epochs limit batching which can affect per-record processing costs [16]. For sessionization, session boundaries are determined by a fixed inactivity delay after the event and, for this reason, we use the original event timestamps as the reference base for our epochs. In the real trace we have, timestamps are captured with nanosecond-level precision, and this is too fine-grained as there are too many distinct timestamps which each need to be tracked individually. We therefore batch input records in windows of one second each, which results in around 1.3M records per epoch. We should note that our choice of epoch granularity primarily affects how often output tuples are materialized and that the high-resolution event timestamps are preserved within the records themselves for use in later analytics.

Streaming arrivals. Timely computations are executed across a set of workers and there is no designated master node. All workers participating in a computation can receive input in parallel. Workers alternate between reading data from an I/O source (e.g., a file or a network socket) and running computation steps, and both of these tasks are interleaved on a single worker thread. Data is ingested into the system using two functions from the programming interface:

- `give` which sends a single record into the dataflow at the current epoch.
- `advance_to` which signals completion of the current epoch. This issues a punctuation and allows the system to issue notifications which are used by the workers to determine that their input can no longer produce data at earlier timestamps.

These two functions correspond to the *data* and *control* paths respectively. Records (data) can be delivered to an operator for the current or any future epoch; notifications (control) arrive in strictly monotonic order. Input records are introduced sequentially (epoch-by-epoch) and processing of the current data must run to completion before proceeding. This has an impact on late arrivals because all data for an epoch needs to be buffered until it is complete before the next processing step can begin; we deal with this using a re-order buffer, as we will discuss in the following paragraph.

Re-order buffer. Our system has to cope with arrival of late records and we deal with using *buffer-and-reorder*, a standard approach in streaming systems. The idea is to stash all arriving records, wait a fixed interval (*slack*) and re-order any out-of-order records within this interval. The parameter *slack* refers to the upper bound on late arrivals and must not be confused with other definitions of slack, e.g., in the context of the critical path [29]. Records which arrive within this interval are ordered appropriately and excessively late records are simply discarded. The trade-off when choosing a value for slack is similar to the inactivity delay for sessions. Larger values add a fixed latency penalty because each record is delayed and there is a bigger memory footprint for buffering the input. We explore this trade-off in Section 5.

Online re-ordering of the input has non-negligible cost but there are sorting algorithms, such as insertion sort and Timsort, which perform well and have linear complexity in this setting. In our implementation the re-order buffer adopts a similar approach to Pigeonhole sort and makes a single pass over the input while moving the records into a fixed number of buffers. The total number of buffers is equivalent to the slack interval and they are filled in circular discipline and re-used as timestamps advance. A record at time t is kept in the buffer slot at $t \,\%\, slack$. The re-order buffer keeps track of the smallest timestamp (*least*) seen so far and repeatedly reads from the input stream. Records with timestamp less than *least* are discarded, records in the interval $[least, least + slack]$ are appropriately slotted and once a record has been observed with timestamp greater than $least + slack$ all intervening buffers are flushed and emitted into the dataflow graph.

4.2 Sessionization as a Dataflow Operator

Sessionization can be expressed as a windowed group-by operator, which essentially groups records into buckets based on their session, and emits the entire bucket once a given inactivity period has elapsed. In the following we explain the use of dataflow primitives to efficiently implement this logic.

Data exchange. Our base framework, Timely Dataflow, has no built-in windowing mechanism, hence, we implemented sessionization as a unary data-parallel operator with a single input and a single output stream. The log dataset arrives from multiple sources and has sub-streams – one per log server in the real deployment – which are ingested by all workers of our system in parallel. Records are not partitioned by a specific attribute and an all-to-all shuffle is required before the custom operator logic such that the data is partitioned appropriately based on their session IDs. Timely Dataflow offers a facility to exchange data among workers based on Parallelization Contracts (PACTs) [7], a generalization of MapReduce. In our implementation we used the `Exchange` PACT, which is parameterized by a hash function that takes an element of the stream and returns a 64-bit integer indicating where to route the data modulo the number of available workers. Concretely, we have a fixed partitioning strategy and apply SipHash 2-4 to the session ID. An important distinction between our system and other approaches is the avoidance of synchronization; data exchange does not imply any logical barrier between the shuffle and computation phases, and workers do not need to coordinate with one another for these two operations.

Data processing. Sessionization is a stateful operator, meaning that arrival of new log records extends the corresponding session window by prolonging the inactivity delay. As records arrive, they are buffered and maintained in a series of indexed collections (essentially hash maps). Operator state is purely

worker-local and the backing collections are manually defined by the developer. Session state is stored in three collections: (i) one that stores messages organized by time, (ii) one that groups together all messages belonging to each session currently in-flight (i.e., for which the inactivity delay has not yet elapsed), and (iii) one containing all session IDs that may have expired by a given timestamp.

An operator is periodically scheduled to run, and the framework invokes the operator logic as shown in the pseudo-code below. From a programmer's perspective, records are delivered one-by-one and the flow of data is *pull-based* by means of an iterator (line **1**). However, our system internally adopts a *push-based* architecture where records are pushed to workers in small batches of 1024 records each, and are handed over to the operator in the form of flat vectors grouped by time. Each record is tagged with an epoch number and organized in separate buffers according to its (event) timestamp. The operator logic is responsible for maintaining any needed state and for segregating state from different epochs, but this is excluded from the pseudo-code for the ease of presentation.

Control plane. A session is marked as complete and all state is flushed once a fixed number of epochs have elapsed with no intervening activity for the session. Each worker receives exactly one notification for each in-flight epoch, and this is handled transparently by the Timely Dataflow framework. Notifications are delivered independently from data but in the same manner (line **6**). Note that pushing data to the output stream is triggered only in response to a notification (line **8**).

```
    // State initialization (done only once)
0.  state.initialize();
    // Consume input stream of batches
1.  for (epoch, data) in input.consume() {
2.    state.add(epoch,data);
3.    notificator.notify_at(epoch);
4.  }
    // Data processing -- Update worker state
5.  let sessions = state.update_active_sessions();
    // Control plane
6.  for epoch in notificator.consume() {
7.    for session in sessions.to_flush_at(epoch) {
        // Push sessions to the output stream
8.      ouput.give(session);
9.    }
10. }
```

Fault tolerance. A practical deployment must be resilient to failures and support automatic scaling in response to changing demand, as these are essential needs for online and long-running analytics. Although these features are currently lacking from our system, it is entirely viable to extend the underlying runtime as both features reduce to a common requirement of being able to migrate and take a consistent snapshot of the data streams and operator state. Timely Dataflow's predecessor system – Naiad [41] – features largely the same computational model, provides exactly-once guarantees on message delivery by taking periodic checkpoints and, as experiments with a comparable streaming acyclic graph application show, imposes low overhead with no discernible impact on median end-to-end latency and a tolerable degradation of 33% in throughput. The simple mechanism employed by Naiad pauses the dataflow computation and relies on upstream producers to backup emitted records until acknowledged (which imposes larger space requirements) and more recent work has reduced the cost of fault tolerance even further by taking snapshots asynchronously [14, 17, 26] and employing rollbacks [4] to selectively undo work. As fault tolerance is an orthogonal concern and can be transparently implemented within the streaming runtime, we defer this to future work.

4.3 Composition of Analytic Tasks

Our system can easily accommodate new ad hoc analytic tasks, and inherits a modular structure largely due to the computational model it builds upon. As the pseudo-code below shows, new operators are introduced by defining an extension method of the `Stream` type (lines **1-3**). Borrowing from ideas in functional programming, operators are generic with respect to the data types of their inputs and outputs, and can be parameterized by user logic. This is a notable departure from current practices for datacenter monitoring and analytics that use custom and tailor-made platforms specialized towards predefined tasks (cf. Section 6). A particular benefit of this approach is an open structure which permits code sharing and only specifies expected traits (functionality) without naming concrete data types.

```
1. trait Sessionize<S: Scope, R: Data> {
2.     fn sessionize(inactivity_delay: Duration)
       -> Stream<S, Session<R>>;
3. }
   // Re-use of results in several applications
4. computation.scoped::(|scope| {
5.    let (input, stream) = scope.new_input();
6.    let trees = stream.sessionize(INACTIVITY_LIMIT)
                        .construct_trace_trees();
      // Trace tree durations
7.    trees.filter(|t| t.messages.len() >= 2)
           .map(|t| min_max_time(t.messages))
           .histogram(|x| log_discretize(x));
      // Classify trace trees by structure
8.    trees.map(|t| t.signature())
           .topk(|x| x.clone(), 10);
      // Identify pairs of communicating services
9.    trees.flat_map(|t| service_call_patterns(t))
           .topk(|pairs| pairs.clone(), 10)
           .show_each_epoch();
10. });
```

In addition, the dataflow-based model allows for reusing simpler tasks like sessionization to concisely compose more complex tasks and, hence, to better utilize the available resources. Examples shown in the pseudo-code above include computing trace tree durations (line **7**), classifying trace

Trace duration	09:00:03 – 10:00:04 UTC
Trace size	223.3 GB (text, gzip-compressed) 305 bytes per record (mean)
Mean input rate	1.3 million events/sec 424.3 MB/sec spread across 1263 streams (replayed from 42 log servers)
Workload statistics	4′876′273′293 annotations 747′242′389 spans 103′382′086 root spans 99′508′175 trace trees

Table 1: Characteristics of the real-world event trace we use

trees based on their structure (line **8**), and identifying service communication patterns (line **9**). The implementation of these is concise and easily expressed in terms of data-parallel operations and we have built a re-usable library as part of TS which extends the Timely framework with Top-K ranking, histograms and CDFs. Whilst the code for these analyses are not included, we provide more details on the last two applications, along with an experimental evaluation in Section 5.2.

5. Experimental Evaluation

In this section we discuss the performance benefits of executing sessionization and various analytical tasks which depend on it within our system. The experimental evaluation relies for realism on an actual workload trace from one of the largest operational datacenters for the travel industry. We demonstrate that, where other general stream processing platforms lag behind, TS can comfortably keep up with high-volume distributed streams of events in real time. By real time we mean that, for each second of input data in event time (*epoch*), TS can perform the respective analytics in less than a second so that all processing tasks are completed before the next batch of input arrives.

Workload characteristics. The trace we use is taken over the span of an hour on a work day. The characteristics of the trace are shown in Table 1. We briefly discuss aspects relevant to the computation requirements for TS. We start with a few generic observations – the mean input rate of the trace is 1.3 million records/sec and it remains relatively constant. This would be also the rate at which TS will have to operate in order to precess events in real time. The 1263 streams we refer to are logging processes and we talk about them in detail when we discuss the logging pipeline in a follow-up section. What is perhaps interesting to observe is that there are about 99.5 million trace trees in the trace but 103 million root spans, meaning that multiple user requests can be grouped into the same session. Note that a root span is initiated by an end-user connecting to the datacenter. Also from the table we could deduce that there are 7.5 spans per tree on average and that each tree will see 49 message exchanges on average. Theses figures should be read cautiously since we observed strong variation in the number of transactions per tree. We do not go into details of the statistical characterisation of the trace in this paper because we focus on TS being able to cope with the real-time use of the trace for sessionization and related tasks.

Two important properties of the dataset are (i) the total duration of user sessions, and (ii) the time between successive messages. These properties dictate the amount of state that needs to be tracked while correlating messages during sessionization and thus guide memory requirements and buffer management. Moreover, the first property also guides the choice of the inactivity timer when deciding to close a session. The vast majority (roughly 95%) of all root spans are short-lived and have a total lifetime of less than 2 seconds. Only in 0.24% of cases root spans remain dormant for more than a minute, although sessions can last up to a full hour and extend beyond the trace boundary. Similarly, in 99.5% of cases, the longest interval between subsequent messages belonging to a single root span does not exceed 12.3 milliseconds. Unlike the total duration of sessions, inter-arrival time does not have a long tail, and we observe that root spans receive activity at least once per second.

As a final comment, the short duration of user sessions is also reflected in their size with respect to the number of service invocations they involve: most trace trees include only a single or a few services (Figure 4). Service invocations can be either local to a server process or result in execution of a transaction on a remote machine; this distinction does not play a role for session reconstruction because our solution works independent of component boundaries, provided transaction identifiers are labelled according to their hierarchy within the trace. The figure showing small number of service invocations is typical of enterprise workloads in a service-oriented architecture where the decomposition of an application is broad when compared against a micro-services approach. Notwithstanding, cloud workloads are also a valid use case for structural reconstruction because the finer breakdown of services only exacerbates the log reconstruction problem and raises the need for better resource accounting.

Logging pipeline and its simulation. Whilst our prototype is intended for deployment in the datacenter, we do not have access to a live stream of log events. Instead, we have access to a log trace generated by the logging infrastructure at the datacenter we look at. In the trace, events are recorded by a distributed middleware which serves as the message broker between application instances. Each middleware replica generates log events each time a message is received, sent or being processed and contains a common header (timestamp, remote endpoint, tracing IDs) and several application-specific

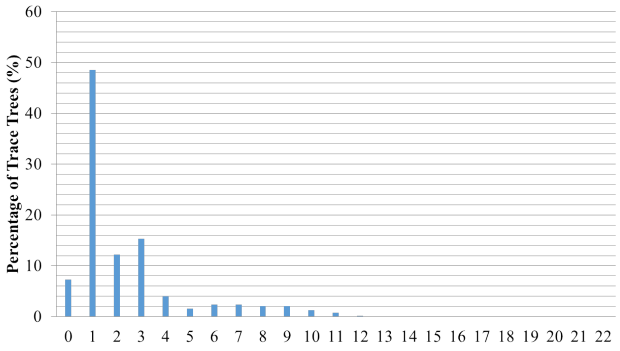


Figure 4: Histogram reflecting the number of service invocations in trace trees

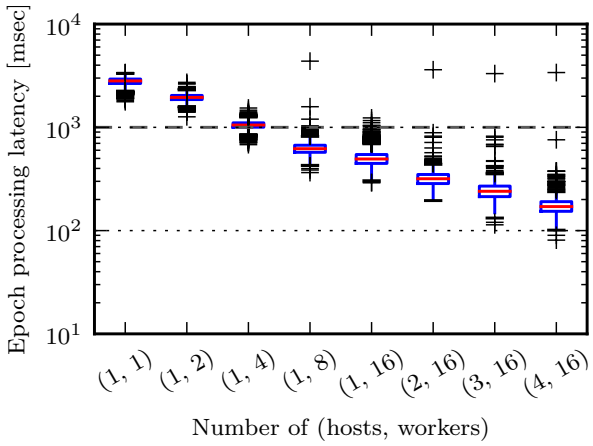


Figure 5: Box-and-whisker plot showing the latency per epoch (1 *sec*) of log data for sessionization on our system (timely-*x*) using *x* number of workers. In this experiment we process the whole trace with 1263 input streams (replayed from 42 simulated log servers mapped over 4 replayer nodes).

fields. Log events are being immediately propagated to a set of 42 log servers, each of which may be running multiple logging processes accumulating to a total of 1263 processes. Upon arrival at the log server the log messages are written to disc and each server will periodically compress outdate messages into log files. These files are what comprise the log trace we use as input.

To realistically represent the behaviour of the actual logging pipepline, we preserve the full fidelity of the trace and reproduce both the timings of each log event and simulate the same number of log processes as in the real deployment (cf. Table 1). We replay the *entire* trace from the log files which are stored on disc and due to the limited size of our evaluation cluster we re-map the log streams across a smaller four-node cluster. This effectively increases the log output at each replayer instance but without altering the total rate of log emission compared against the original logging pipeline. The functionality that does that is the *replayer module*, which runs as an external data source. Based on the original event time, the replayer groups log events into epochs and at the begin of a new epoch the corresponding events are emitted in their original text format over a TCP socket.

The replayer uses the same number of workers as the main computation. The simulated log servers are assigned to the available workers of our system in a round robin fashion. Since the number of workers is a varying parameter in our experiments, different number of log servers are mapped to a worker depending on the setup. In the experiments, we do not account the cost of replay (since this is external to our system), however, all latencies we report include the time each processing task needs to read data from the input buffers.

Experimental setup. Our system is built upon the open-source release of Timely Dataflow [36] (v. 0.1.15) compiled with Rust 1.14. All experiments are conducted on a small cluster with four identical machines, each one having two sockets with Intel Xeon E5-2650 (16 physical cores, 32 with Hyper-threading), 64 GB RAM, running Linux (Debian 7.8 "wheezy"). The cluster has a private 10 Gb Ethernet network. The workload trace is stored in gzip-compressed archives on a local SAN attached via iSCSI which has 60 x 3 TB disks in a RAID 6 configuration.

Prior to each experiment we present in the following, we performed a warm-up run to make sure the page cache is primed. This is to ensure that timings for repeated executions are consistent, and to simulate a real logging pipeline where disks are used for persistent storage of archives but recent log records are likely to remain in memory for a short time frame.

5.1 Online Sessionization

In this section we evaluate the performance of our system in terms of latency and memory consumption for the basic sessionization task described in Section 4.2. We also compare our system with a state-of-the-art streaming engine.

Latency (full log rate). We start with evaluating the cost of sessionization, being the cornerstone for higher-order analytical tasks. The experiments logic is as follows: measure the time required to reconstruct user sessions for each epoch (1 *sec*) of incoming log data; do that for the whole trace of Table 1. To provide accurate measurements, we instrumented our system with high resolution timers (`time::precise_time_ns()` in Rust), and we measured the sessionization time as the interval between (i) the first time an epoch is observed, and (ii) the time a *punctuation* is delivered by the system, confirming that the epoch is over (cf. Section 3). We vary the number of worker threads and report the results in Figure 5. Each box in Figure 5 contains the first quartile (bottom line), median (red line) and third-quartile (top line), whiskers extend until 1.5 times the interquartile range and outliers are plotted as individual points. As a general comment, our system is able to perform sessionization

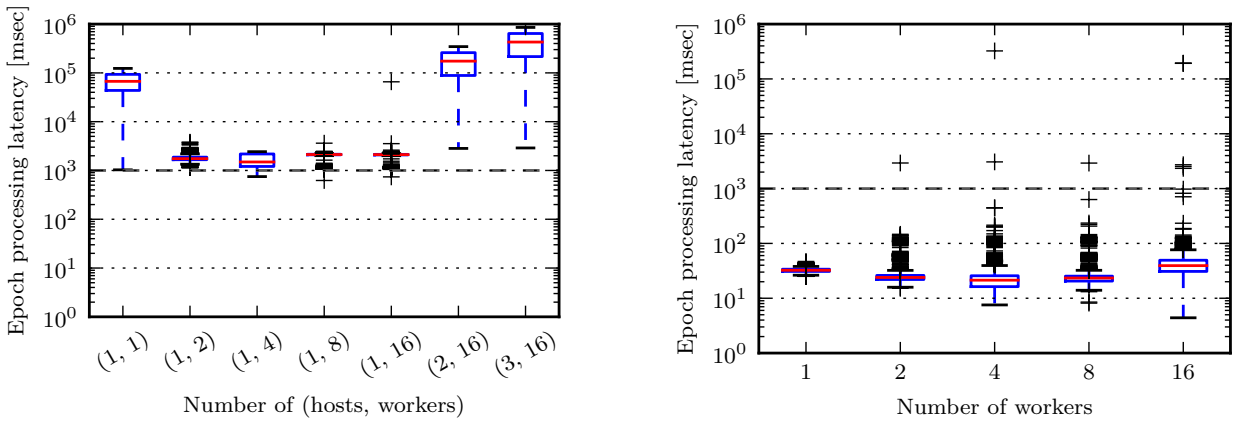
in real time (i.e., in less than a second) when using 8 or more workers.

Comparison with other systems (reduced log rate). Previously, we made the argument the TS can deliver better performance than a general-purpose streaming engine. In this evaluation scenario we conduct the experiments to back our claims. For comparison we selected Apache Flink [33] – a state-of-the-art steaming engine – which has a flexible design for windowing over data streams and which already accommodates the inactivity windows needed for sessionization as a built-in primitive. In order to make a fair comparison, we disabled checkpointing and instrumented Flink similarly to our system taking advantage of the fact that it already tags tuples with their event time ('epoch') and tracks their progress. Our latency measurements rely on the delivery of watermarks and records the time interval between when the first tuple for a given epoch is observed and later once a punctuation is delivered, confirming that no elements with that epoch will arrive in the future. In this way, the processing time is derived by sampling two timestamps from the high-resolution, monotonic clock source provided by the runtime environment, namely `System.nanoTime()` in Java. In addition, we provided a custom data source that ingests data from a TCP socket in parallel across all streaming tasks, disabled checkpointing and also configured Flink with event time characteristics so that its notion of time matches exactly the one used in our system.

A key finding of the experiment is that Flink failed when processing the complete input stream (prior experiment) due to an excessive accumulation of log records in operator queues. Specifically, its processing rate fell behind the input rate and finally went out of memory. Monitoring revealed a high level of backpressure at the data source, i.e., more than half of the stack traces samples were blocked on internal method calls. To still report a comparison, we evaluated both systems against a reduced input dataset by replaying only one of the 42 log servers (with 37 log streams and a mean input rate of 6.9 MB/s). Figure 6 depicts the results of this experiment for each epoch (1 *sec*) of the reduced trace. Boxes in Figure 6 have the same meaning as in Figure 5. We observe a significant difference in the processing delay when comparing both systems on an identical workload. In its best configuration (single host, four workers), Flink spent on average 2.1 seconds (±1.1 s) for processing a single epoch of streaming logs whereas our system (with 16 workers) took only 26 milliseconds (±53 ms). Moreover, our system can make better use of parallelism and both the median and 90-percentile latency scale linearly until 8 cores. After 8 workers (there is one worker per core), load imbalance stalls progress due to the cluster-wide synchronization once per epoch and the costs for progress tracking in Timely Dataflow increasingly dominate the computation. We plan to investigate these issues more fully as part of future work. It is also worth mentioning that the choice of Rust as an implementation language has a considerable impact on the memory footprint of sessionization; in our system, the peak resident set size remained stable and reached a peak of 203 MB while Flink's heap rose above 7.5 GB and required considerable tuning.

Varying the number of workers. In these experiments we evaluate the performance of our system when executing the sessionization task on a single multi-core machine of our cluster. The objectives here are (i) to provide detailed latency measurements when varying the number of worker threads, and (ii) to show the fraction of time spent in reading data from the input buffers. While in the previous experiment we handled the full log, here, to improve readability results are shown only for the first 5 minutes of the trace in Figure 7. Initially, the epoch processing latency gradually rises and stabilizes after roughly a minute (Figure 7a); this is explained by a corresponding increase in the log arrival rate caused by the trace collection process. The horizontal dotted lines in both plots of Figure 7 show the upper bound to meet real time processing for one epoch of log data. Note that configurations with fewer than 16 worker threads cannot process the trace within one second (Figure 7a). Moreover, the breakdown in Figure 7b shows that a sizeable fraction of execution time is spent reading input (on average 41.1%). Overall, these experiments demonstrate that our system can process logs in real-time with only the modest resources of a single modern multi-core machine.

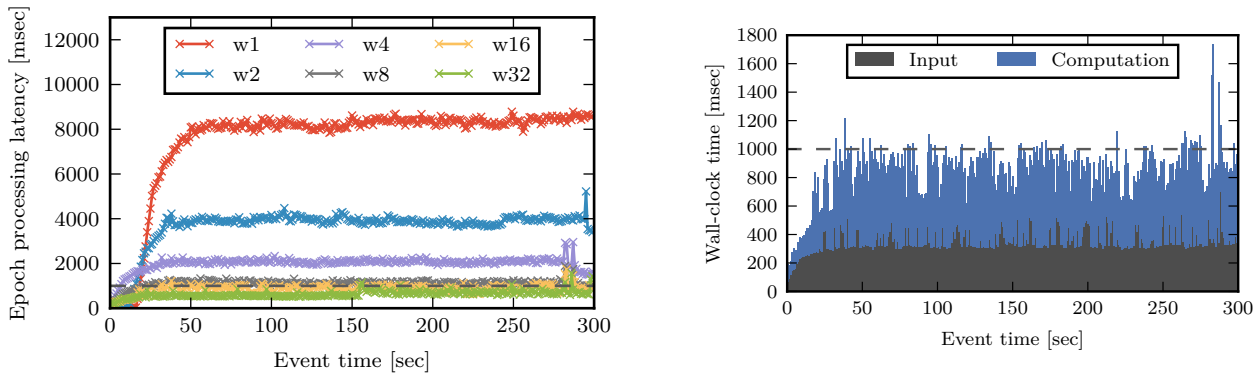
Memory footprint. As discussed in Section 2.3, one of the challenges in a real tracing infrastructure is that log records arrive in a non-deterministic order. In the common case, logs are not delayed arbitrarily and arrive within a limited time frame; taking the difference between the timestamps of consecutive log records that are out-of-order, we observed a median difference of 0.69 ms with the 90-th, 99-th, 99.9-th and 99.99-th percentile values being 4.5 ms, 17 ms, 32.5 ms, 1.2 sec respectively. The most delayed log record we observed arrived many minutes (485 sec) late. In Section 4.1 we described how we deal with late inputs, i.e., by buffering and re-ordering records at the input source, an approach that introduces a fixed latency penalty. However, the characteristics of the real trace indicate that, whilst out-of-order records are a problem to contend with, they can be handled by simply setting an upper bound of several seconds. Here we investigate the impact of this choice on our system's memory footprint. Figure 8 shows the total resident set size of the sessionization process on a single machine of our cluster as we vary the re-order window size in number of epochs. As expected, this grows linearly because a larger window buffers proportionally more input data. Each second of input data from our trace adds 571 MB on average. All data resides in main memory, and the total physical RAM becomes the limiting factor (when the window size is set to 110 epochs).



(a) Flink

(b) TS

Figure 6: Comparison of TS against state-of-the-art. Box-and-whisker plot showing the latency per epoch (1 *sec*) of log data for sessionization with varying number of workers. The input streams in this experiment were reduced to 37 out of 1263 in total, so that Flink could keep up with the processing.



(a) Latency when varying the number of workers

(b) Fraction of time reading input (with 16 workers)

Figure 7: Latency of sessionization with breakdown of processing costs for each epoch of log data (1 *sec*) during the first 5 minutes of the trace. These experiments are run on a single host with both the replayer and TS sharing the resources. The simulated inputs correspond to the entire datacenter trace with its 1263 parallel log streams and 42 log servers.

5.2 Online Data Summaries

To demonstrate the merits of our approach, this section discusses analytic tasks we conducted on top of the sessionization output. As mentioned in Section 4.3, the composition of analytic tasks allow us to reuse the output of simpler tasks into more complex ones and to easily extend the system with new modules. Examples of first-level statistics are the duration of sessions, bandwidth incurred, and number of services invoked per trace tree. Their composition enables deeper analytics, including latency inference [35], communication pattern mining, service dependency extraction, and critical path analysis [19], among others. First, a cautionary note - the applications just mentioned are more sensitive to poor data quality and incomplete traces (Section 2.3). For example, skewed timestamps break causality and result in false dependencies whereas missing logs fragment the trace trees and cause short dependency chains.

Online trace tree clustering. The classification of trace trees based on their structure provides valuable high-level feedback on the performance of the application layer in the data center. To support such a classification in real time, we have implemented a dataflow whose task is to first create light-weight signatures of trace trees, as produced by sessionization, then count the number of occurrences of each such signature, and finally output the k most frequent signatures per epoch. A tree signature amounts to a vector whose elements correspond to the number of outgoing edges of the nodes in the trace tree. We focus on the outgoing node degree since real traces typically have a single incoming edge, the one that triggered the current span, and multiple outgoing

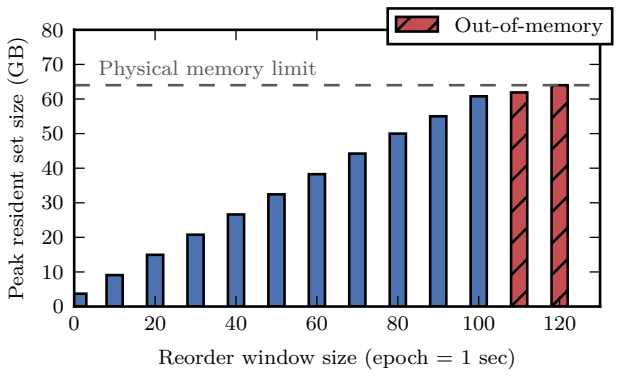


Figure 8: Total memory used by our system when varying the size of the re-order buffers (in number of epochs). The bigger the re-order buffers, the more tolerant the system is to late record arrivals.

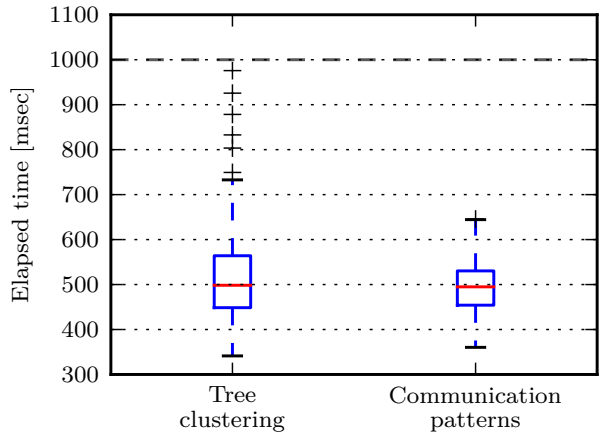


Figure 9: Measured latency per epoch (1 *sec*) of log data to conduct two different analytic tasks on the output of sessionization, including the latency of sessionization. The top-10 trace tree signatures and pairs of communicating services are updated in real time (<1 *sec*).

edges, leading to subsequent spans. Performance results for online structure classification of trace trees are shown on the left of Figure 9.

Inferring communication patterns. Here we showcase how we can efficiently extract the top-k most frequent pairs of communicating services per epoch of log data. Such summaries are useful as they may suggest the need to optimise the placement of replicas for specific pairs of services that communicate heavily. The task is performed by traversing the trace trees (as produced by sessionization) in a breadth-first fashion, track the pairs of communicating services, and output the k most frequent ones per epoch. Performance results for this task on the whole trace are shown on the right of Figure 9.

6. Related Work

The work we present in this paper spans multiple boundaries across the datacenter analytics stack, and this section provides an overview of the related works divided into three main categories: (i) systems for datacenter analytics, (ii) frameworks for tracing distributed systems, and (iii) modern engines for data-parallel computation. In each class of works, we describe the core features of the existing approaches, and we also highlight how our system advances or complements the current practices.

Datacenter Analytics. Datacenter log analysis has received considerable attention in both academia and industry over the last years [43, 51]. One of the most prominent systems in the area is Splunk [2], a full-fledged platform that goes beyond traditional monitoring tools, like Ganglia [47] and Nagios [1], by collecting and processing logs from multiple levels of the software and hardware stack of the datacenter. Splunk supports batch processing of archived data as well as real-time processing of streaming data with the use of `Forwarders` (reminiscent of the `Replayer` module we used in Section 5). In contrast to our system, Splunk does not adopt a *push-based* evaluation strategy along the whole data processing pipeline. Changes in the logs are pushed down to the `Indexers` but the actual data analytics are *pull-based*; to update a data summary, an incremental query must be issued, making online processing quite cumbersome. In addition, although Splunk's Search Processing Language (SPL) [13] offers a wide range of built-in functions (including SQL-like operators, clustering, and rule mining algorithms, among others), it is unclear how it can be used to express more complex tasks like those we describe in Section 5.2. To the best of our knowledge, such graph-based analytics have been addressed so far either in an offline setting [8, 9, 11, 18, 19, 23, 24, 27, 35, 48, 54] or in the context of online special-purpose systems [30, 38–40, 46, 55].

Distributed System Tracing. End-to-end tracing frameworks are vital components for the comprehensive performance analysis of complex datacenters. DTrace [12] was the first to introduce the idea of dynamic instrumentation, which was later adopted in systems like Fay [22]. Other popular tracing frameworks are Google's Dapper [49], and its open-source version Zipkin [3]. PinPoint [18], and X-Trace [23] provide mechanisms for offline log processing whereas Magpie [10], MTracer [55] and PivotTracing [34] are targeted to real-time analytics. Our work does not make any assumptions on the instrumentation of the individual systems in the datacenter, hence, it can be easily combined with any of the existing tracing frameworks. In fact, the logs we used in Section 5 were generated with custom tracing techniques

implemented by our industry partners, and whose details were only partially disclosed to us.

Streaming Engines. The powerful computational model of Timely Dataflow serves as a good basis for various data processing tasks, including complex graph-based analytics like those in section 5.2, and ML-based diagnostics like those in [28, 42, 44, 52]. Three popular data-parallel systems that also meet the requirements we listed in Section 3 are Spark Streaming [53], Flink [15], and Storm [50]. These systems have rather general computation models, they support streaming inputs from multiple sources, and they can also operate according to the event timestamps (event vs system time). Technically, the tasks we perform in Section 5 can be implemented on top of these systems, however, we are not aware of any public results regarding their integration into a real logging pipeline, other than those we presented for Flink in Section 5.

7. Conclusions

TS maintains and updates user sessions in real-time for an entire data center with modest resource requirements and processing latency in the range of tens of milliseconds. We exploit the comprehensive instrumentation already present in data center applications to reconstruct user sessions, communication dependencies and trace tree clusters online.

Whilst these computations involve window operations and graph traversals and go beyond the typical relational operators used in log processing, we demonstrate how such computations can be expressed in data-parallel fashion using a dataflow model, executed efficiently using Timely Dataflow, and can significantly outperform a state-of-the-art general stream processor in both memory usage and result latency. We demonstrate the use of TS using traces from from a large, operational production datacenter.

Timely Dataflow currently lacks two features desirable in a system for online processing of real-time logs: dynamic scaling and fault tolerance. These both reduce to a common requirement to migrate or take a consistent distributed snapshot of the dataflow state and streaming inputs. Existing work on recovery which could adopted, for instance the asynchronous snapshot mechanism in Flink [14], or (closer to Timely's core model) rollbacks [4] to selectively undo work.

While we expect this additional functionality to impact the performance of TS somewhat, we argue that TS still represents a far more efficient point in the design space for datacenter diagnostic foundations than existing general-purpose stream-processing systems.

Acknowledgments

Many people contributed towards the work described in this paper. The authors appreciate the fruitful collaboration and assistance of Frank McSherry and would like to generously thank Amadeus Data Processing GmbH, particularly Dénes Vadász, Pierre-Jean Demaret, and Dietmar Fauser for providing access to the production workload logs, for thoroughly detailing the ESB architecture and for offering technical guidance throughout the project. We are further indebted and express our gratitude to the anonymous reviewers and our shepherd Dilma Da Silva for their valuable and thoughtful feedback which helped shape the final version of this paper. This work is supported in part by the Swiss National Science Foundation under grant number 159537, by a Google Research Award (2016), and additionally by a generous gift from Amadeus IT Group.

References

[1] Nagios. URL https://www.nagios.org.

[2] Splunk. URL https://www.splunk.com/.

[3] Zipkin. URL http://zipkin.io/.

[4] M. Abadi and M. Isard. Timely rollback: Specification and verification. In *7th NASA Formal Methods Symposium*, NFM '15, pages 19–34, Apr. 2015. doi: 10.1007/978-3-319-17524-9_3.

[5] M. Abadi and M. Isard. Timely dataflow: A model. In *Formal Techniques for Distributed Objects, Components, and Systems: 35th IFIP WG 6.1 International Conference*, FORTE '15, pages 131–145, June 2015. doi: 10.1007/978-3-319-19195-9_9.

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, Aug. 2015. doi: 10.14778/2824032.2824076.

[7] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke. MapReduce and PACT - comparing data parallel programming models. In *Proceedings of the 14th Conference on Database Systems for Business, Technology, and Web (BTW)*, BTW 2011, pages 25–44, 2011. ISBN 978-3-88579-274-1.

[8] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, 2012.

[9] P. Bahl, R. Chandra, A. G. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Kyoto, Japan, August 27-31, 2007*, pages 13–24, 2007.

[10] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation*, OSDI'04, page 259–272, Dec. 2004.

[11] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their

behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, 2014. ISBN 978-1-4503-2756-5.

[12] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, 2004.

[13] D. Carasso. *Exploring Splunk*. CIT, 2012.

[14] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.

[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.

[16] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 838–849, 2003.

[17] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985. doi: 10.1145/214451.214456.

[18] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 595–604, 2002.

[19] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, 2014.

[20] R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Knowledge and Information Systems*, 1:5–32, 1999. doi: 10.1007/BF03325089.

[21] J. Dai, J. Huang, S. Huang, B. Huang, and Y. Liu. Hitune: Dataflow-based performance analysis for big data cloud. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIX ATC'11, pages 7–7, 2011.

[22] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.*, 30(4):13:1–13:35, Nov. 2012. ISSN 0734-2071. doi: 10.1145/2382553.2382555.

[23] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.*, 2007.

[24] Z. Guo, D. Zhou, H. Lin, M. Yang, F. Long, C. Deng, C. Liu, and L. Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 27–27, 2011.

[25] P. Huntington, D. Nicholas, and H. R. Jamali. Website usage metrics: A re-assessment of session data. *Information Processing and Management*, 44(1):358–372, Jan. 2008. doi: 10.1016/j.ipm.2007.03.003.

[26] M. Isard and M. Abadi. Falkirk wheel: Rollback recovery for dataflow systems. *CoRR*, abs/1503.08877, 2015.

[27] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*, pages 243–254, 2009.

[28] S. Kavulya, S. Daniels, K. R. Joshi, M. A. Hiltunen, R. Gandhi, and P. Narasimhan. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*, pages 1–12, 2012.

[29] J. E. Kelley, Jr and M. R. Walker. Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 160–173, 1959.

[30] M. Kim, R. Sumbaly, and S. Shah. Root cause detection in a service-oriented architecture. *SIGMETRICS Perform. Eval. Rev.*, 41(1):93–104, 2013.

[31] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 985–996, 2011. doi: 10.1145/1989323.1989426.

[32] B. Li, Y. Diao, and P. Shenoy. Supporting scalable analytics with latency constraints. *Proc. VLDB Endow.*, 8(11):1166–1177, July 2015. doi: 10.14778/2809974.2809979.

[33] B. Lohrmann, D. Warneke, and O. Kao. Nephele streaming: Stream processing under QoS constraints at scale. *Cluster Computing*, 17(1):61–78, Mar. 2014. doi: 10.1007/s10586-013-0281-8.

[34] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393, 2015. ISBN 978-1-4503-3834-9.

[35] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-Dar. Modeling the parallel execution of black-box services. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 20–20, 2011.

[36] F. McSherry. A modular implementation of timely dataflow in rust. URL https://github.com/frankmcsherry/timely-dataflow.

[37] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research*, CIDR'13, Jan. 2013.

[38] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 24(6):1245–1255, 2013.

[39] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, H. Cai, and G. Yin. An online service-oriented performance profiling tool for cloud

computing systems. *Frontiers of Computer Science*, 7(3):431–445, 2013.

[40] H. Mi, H. Wang, Z. Chen, and Y. Zhou. Automatic detecting performance bugs in cloud computing systems via learning latency specification model. In *Proceedings of the 2014 IEEE 8th International Symposium on Service Oriented System Engineering*, SOSE '14, pages 302–307, 2014.

[41] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, 2013. doi: 10.1145/2517349.2522738.

[42] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 26–26, 2012.

[43] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, 2012.

[44] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pages 191–200, 2010.

[45] B. Poirier, R. Roy, and M. Dagenais. Accurate offline synchronization of distributed traces using kernel-level events. *SIGOPS Oper. Syst. Rev.*, 44(3):75–87, Aug. 2010. ISSN 0163-5980. doi: 10.1145/1842733.1842747.

[46] I. Rish, M. Brodie, N. Odintsova, S. Ma, and G. Grabarnik. Real-time problem determination in distributed systems using active probing. In *Managing Next Generation Convergence Networks and Services, IEEE/IFIP Network Operations and Management Symposium, NOMS 2004, Seoul, Korea, 19-23 April 2004, Proceedings*, pages 133–146, 2004.

[47] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide area cluster monitoring with ganglia. In *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003), 1-4 December 2003, Kowloon, Hong Kong, China*, page 289, 2003.

[48] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 43–56, 2011.

[49] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010. URL http://research.google.com/archive/papers/dapper-2010-1.pdf.

[50] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156, 2014.

[51] C. Wang, S. Kavulya, J. Tan, L. Hu, M. Kutare, M. P. Kasick, K. Schwan, P. Narasimhan, and R. Gandhi. Performance troubleshooting in data centers: an annotated bibliography? *Operating Systems Review*, 47(3):50–62, 2013. doi: 10.1145/2553070.2553079.

[52] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, 2009.

[53] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, 2013. doi: 10.1145/2517349.2522737.

[54] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. Lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 629–644, 2014.

[55] J. Zhou, Z. Chen, H. Mi, and J. Wang. Mtracer: A trace-oriented monitoring framework for medium-scale distributed systems. In *8th IEEE International Symposium on Service Oriented System Engineering, SOSE 2014, Oxford, United Kingdom, April 7-11, 2014*, pages 266–271, 2014.