Scrub: Online TroubleShooting for Large Mission-Critical Applications

Arjun Satish
Turn Inc.

Thomas Shiou
Turn Inc.

Chuck Zhang
Turn Inc.

Khaled Elmeleegy[*]
Oracle Cloud

Willy Zwaenepoel
EPFL

ABSTRACT

Scrub is a troubleshooting tool for distributed applications that operate under strict SLOs common in production environments. It allows users to formulate queries on events occurring during execution in order to assess the correctness of the application's operation.

Scrub has been in use for two years at Turn, where developers and users have relied on it to resolve numerous issues in its online advertisement bidding platform. This platform spans thousands of machines across the globe, serving several million bid requests per second, and dispensing many millions of dollars in advertising budgets.

Troubleshooting distributed applications is notoriously hard, and its difficulty is exacerbated by the presence of strict SLOs, which requires the troubleshooting tool to have only minimal impact on the hosts running the application. Furthermore, with large amounts of money at stake, users expect to be able to run frequent diagnostics and demand quick evaluation and remediation of any problems. These constraints have led to a number of design and implementation decisions, that go counter to conventional wisdom. In particular, Scrub supports only a restricted form of joins. Its query execution strategy eschews imposing any overhead on the application hosts. In particular, joins, group-by operations and aggregations are sent to a dedicated centralized facility. In terms of implementation, Scrub avoids the overhead and security concerns of dynamic instrumentation. Finally, at all levels of the system, accuracy is traded for minimal impact on the hosts.

We present the design and implementation of Scrub and contrast its choices to those made in earlier systems. We illustrate its power by describing a number of use cases, and we demonstrate its negligible overhead on the underlying application. On average, we observe a maximum CPU overhead of up to 2.5% on application hosts and a 1% increase in request latency. These overheads allow the advertisement bidding platform to operate well within its SLOs.

---

[*] This work was done when the author was at Turn Inc.

CCS CONCEPTS

• **Information systems** *Query languages*; • **Applied computing**; • **Computer systems organization** *Distributed architectures*; *Real-time systems*; *Dependable and fault-tolerant systems and networks*;

KEYWORDS

Scrub, Advertising, Mission Critical, Big Data, Query Processing, Troubleshooting, Debugging, Distributed Systems.

1 INTRODUCTION

Modern online applications, such as web search engines, social networks and online advertising platforms, serve billions of requests per day. To guarantee revenue streams, they must be always-on and respond to user requests within very tight SLOs. The complexity of these applications is staggering. Thousands of geographically distributed machines cooperate to maintain a very large internal state. This state is updated constantly, and the application itself is in constant flux, due to frequent new software rollouts.

Turn's online advertisement bidding platform is one example of such a complex system. Many thousands of machines serve millions of bid requests per second. New advertising campaigns, changes to existing ones, requests for bids on ad space, user clicks on ads all trigger updates to the state of the system. Bug fixes, new features, and introduction of new ad targeting models cause frequent new software rollouts. Within such a complex system problems occur all the time. New versions of the software often introduce bugs. Erroneous user input may lead to misconfiguration of advertising campaigns. With often millions of dollars at stake, problem resolution must be quick.

Scrub is a new troubleshooting tool for large mission-critical distributed applications, that we use in production on Turn's advertisement bidding system. Much like similar tools, Scrub allows users to specify SQL-like queries on the events in the system in order to assess its correct operation. Scrub is used by both developers and users, so the queries are quite diverse, and query load can at times be considerable. The key design goal of Scrub, and what distinguishes it from existing systems, is minimal interference with the application, even under high query load. Under no circumstance should the operation of Scrub cause the bidding platform to violate its SLO. Any monitoring inevitably puts some load on the machines where

the application runs, but Scrub strives to minimize any impact it may have on the application.

To this end we have designed a new query language that avoids costly operations that are seldom used, and trades, when necessary, accuracy for minimal impact. We also go counter to traditional query optimization, which optimizes for query execution time, typically by performing as much of the query as close to the data as possible [29]. Instead, our query execution strategy reduces as much as possible impact on the hosts. Most of the query execution and in particular all join, group-by and aggregation activity takes place in a dedicated centralized engine, ScrubCentral. The only query activity that takes place on the hosts is projection and selection, which serve to reduce the amount of data that needs to be sent by the hosts to ScrubCentral, thereby further reducing impact on the host.

The question then becomes whether under these constraints one can build a troubleshooting engine that remains sufficiently expressive to allow users to troubleshoot real problems, and is performant enough, in terms of throughput and latency, to allow expedient problem resolution. Our experience is that this is indeed possible. Our query language efficiently supports most SQL operators, and we illustrate its power by describing a number of use cases in which Scrub was used to quickly discover issues.

Our work was in large part motivated by the fact that logging [31, 37, 38, 43, 44, 46] – in practice is still the most common troubleshooting technique for distributed systems – is inadequate for our environment. With events containing hundreds of fields being produced at a very high rate, the amount of storage needed for logging quickly becomes prohibitive, leading to difficult questions about what to log and what not, with no good answers. Moreover, offline log analysis is computationally expensive and implies unacceptable delays in problem resolution.

Recognizing the issues with logging, a number of recent research works have developed online troubleshooting systems [9, 12–14, 22, 23, 30, 35, 39, 40]. The principal differences between Scrub and these other systems derive directly from differences in objectives. Scrub aggressively minimizes impact on the hosts, an objective that is essential in production systems.

The contributions of this paper are:

(1) The design and implementation of a troubleshooting tool for distributed applications that minimizes impact on the hosts running the application, and is therefore suitable for use in production environments with stringent SLOs.
(2) A query language and a query execution strategy that achieve the goal of low impact on the application hosts.
(3) The evaluation of Scrub in terms of expressiveness and performance.

The outline of the rest of this paper is as follows. Section 2 presents our design philosophy. Section 3 describes the Scrub query language. Section 4 shows how query execution is carried out in Scrub. Section 5 covers some implementation aspects. Section 6 compares the design and implementation decisions in Scrub to alternative strategies. Section 7 describes Turn's online advertisement bidding platform. Section 8 presents six use cases of Scrub at Turn. Section 9 explores some performance aspects of Scrub. Section 10 provides a summary of related work, and we conclude in Section 11.

2 DESIGN PHILOSOPHY

Like in earlier systems [22, 35], monitoring in Scrub takes the form of formulating and executing high-level queries over events defined and generated by the application under study. Scrub provides an API for the application developer to define and generate events, and supports a query language for the troubleshooter to express queries over these events.

Scrub is intended for use in mission-critical systems with very stringent SLOs, in which even minor disruption can cause the system to miss its SLO. As a result, Scrub's primary goal is to impose only minimal overhead on the running system, if need be at the expense of other, more common goals in troubleshooting systems, such as expressivity of the query language, query execution performance, or accuracy. This singular focus on minimizing impact on the monitored system is the primary differentiator between Scrub and earlier systems for troubleshooting, and is essential if the system is to be used in production environments with stringent performance demands.

While Scrub also strives for these other qualities such as query expressivity, performance and accuracy, it does so only to the extent that they do not interfere with its primary goal of not impacting the monitored system. Scrub's query language is sufficiently expressive to diagnose complex issues in large distributed systems, but constructs that may impose considerable overhead on the running system are discarded from the language. On the contrary, the query language incorporates facilities that allow Scrub to reduce its impact on the target system. Furthermore, as much money is at stake in these mission-critical systems, we want problem resolution to be expedient, and therefore Scrub strives for good query performance, but only to the extent that achieving good query performance does not have an adverse impact on the monitored system. Finally, query results must be sufficiently accurate to allow correct problem diagnosis, but Scrub allows a degree of inaccuracy. Our experience with Scrub is that any concessions we have made in terms of expressivity, performance and accuracy have had only minor consequences on its ability to support efficient troubleshooting.

In the next two sections we describe how this singular focus on minimal impact on the application hosts permeates the design of the query language and the query execution model.

3 QUERY LANGUAGE

Scrub allows troubleshooters to formulate queries over events defined and generated by the application that is being monitored. We first describe event definition and then the query language.

3.1 Events

An event in Scrub is an n-tuple of user-defined fields. In addition, Scrub annotates events with two system fields, a unique request identifier and a timestamp. The size of this metadata is bounded and is kept to the minimum necessary to support equi-joins and windowing.

The definition of an event takes two arguments: the event type (a string label), and a list of fields and their data types. Scrub supports fields of types: boolean, int, long, float, double, date/time, string, and homogeneous lists of these primitive types. In addition, Scrub

```
@ScrubType("bid")
public class ScrubBid
{
    @ScrubField("exchange_id")
    private final long exchange_id;

    @ScrubField("city")
    private final String city;

    @ScrubField("country")
    private final String country;

    @ScrubField("bid_price")
    private final double bid_price;

    @ScrubField("campaign_id")
    private final long campaign_id;

    // business logic
    // ...
}
```

Figure 1: Event type definition for bid event in the Turn bidding system.

also supports nested objects, e.g., XML encoded objects. Other data types can be added as the system evolves.

Figure 1 shows the definition of a *bid* event type, corresponding to a bid response sent back to an online ad exchange in Turn's online bidding system. The definition uses Java annotations to declare the fields of an event type.

The application defines where in the code an event of a certain type can be generated by means of a Scrub API log() call.

3.2 Query Language

Scrub users write SQL-like queries, including selection, projection, join, grouping and aggregation operations. The query specifies the event types used in the query, and refers to the fields of those event types. The query may include a time window. Otherwise, a default window is used. Currently, only tumbling windows are supported, but Scrub can easily be extended to allow sliding windows. Scrub supports common aggregation functions such as *MIN*, *MAX*, *AVG*, *SUM*, and *COUNT*, and probabilistic aggregation functions such as *TOP-K*, using the space saving stream summary [36], and cardinality counts, such as *COUNT_DISTINCT*, using hyperloglog [27].

In addition to the above, Scrub uses additional constructs to express the following concepts:

- **Query span:** Unlike traditional streaming queries, which are everlasting, Scrub queries have a finite timespan, specified by the start and the duration keywords. Both have default values if no explicit values are given in the query. The timespan guards against users forgetting to end their queries after a troubleshooting session is finished, and avoids overloading the target system with queries no longer of interest.
- **Target hosts:** A Scrub query can specify the set of machines from which the query is to collect events. This set can include all machines, machines from a given list, or machines performing a certain service. Filters can be applied to a set, e.g., clients in the AdServers service that reside in the San Jose data center. Putting this construct in the language instead of, for instance, using a selection on the host name, allows Scrub to limit the execution of the query to the specified hosts, again reducing the load on the target system.
- **Sampling:** Two types of sampling are supported: sampling on the set of hosts, and sampling on the events on a given host. Both types of sampling can be used in combination with each other. Sampling reduces the load on the hosts in the target system if the query touches many events. The sampling rate is configurable, providing the possibility of trading accuracy for performance in a tunable fashion. Similar to ApproxHadoop [25], error bounds can be obtained through multi-stage sampling theory.

  For example, to compute an approximate sum, we randomly select n machines, and then randomly select m_i events from each chosen machine i. The sum is computed according to Equation 1, and the error bound according to Equations 2 and 3, where s_i^2 is the variance of readings at machine i,

$$\hat{\tau} = \frac{N}{n} \sum_{i=1}^{n} \left(\frac{M_i}{m_i} \sum_{j=1}^{m_j} v_{ij} \right) \pm \varepsilon \tag{1}$$

$$\varepsilon = t_{n-1, 1-\alpha/2} \sqrt{\widehat{Var}(\hat{\tau})} \tag{2}$$

$$\widehat{Var}(\hat{\tau}) = N(N-n)\frac{s_u^2}{n} + \frac{N}{n} \sum_{i=1}^{n} M_i(M_i - m_i)\frac{s_i^2}{m_i} \tag{3}$$

Figure 2 shows some example Scrub queries used in the Turn bidding system.

4 QUERY EXECUTION

Execution of a Scrub query can span thousands of machines in many data centers across the globe. Scrub's primary query optimization goal is minimizing impact on the hosts of the target system. To achieve this goal, Scrub departs from the conventional query optimization strategy of moving operations as close as possible to the data or to where the data is generated. In particular, the join, group-by and aggregation operations are carried out in a dedicated central facility, ScrubCentral, and not on the hosts. Only selection and projection happen on the host, because they reduce the amount of data to be sent to ScrubCentral.

Figure 3 shows the steps in the execution of a Scrub query. The user submits a query formulated in the Scrub query language to the Scrub query server. The server parses and validates the query, generates a unique query identifier, and then creates a number of query objects tagged with this unique query identifier. A query object representing the selection and projection operators is sent to the hosts involved in the query, where it activates data collection, including selection and projection. The resulting events are then sent to ScrubCentral. Another query object representing the join, group-by and aggregation operators is sent to ScrubCentral, where the final query result is computed. The numbers on the arrows of Figure 3 show the typical execution order.

```
• Select MAX(bid_price) from bid
  where exchange_id in (1, 2, 3, 4, 6, 9, 10)
  window 1m start 5s duration 10m;
• Select AVG(impression.cost)
  from bid, impression
  where bid.exchange_id = 10 and
          bid.request_id =
          impression.request_id
  @[Servers in (host1,host2,host3)] window 1m;
• Select exchange_id, country, COUNT(*)
  from bid
  where bid_price > 0.4
  @[Service = AdServers and
          SamplesPerServer = 0.1]
  group by exchange_id, country;
• Select COUNT_DISTINCT(user_id),
          AVG(bid_price)
  from bid
  where exchange_id in [10, 11]
  @[Service = AdServers and
          SampleServers = 0.1];
```

Figure 2: Example Scrub queries in the Turn bidding system.
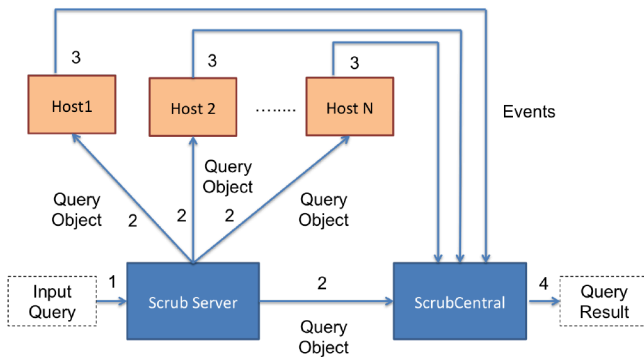

Figure 3: Steps in the Execution of a Scrub Query.

4.1 Data Collectors

When a query object arrives at a host that references fields of a particular event type, the collector starts collecting events of that type. Otherwise, the collector is inactive.

For each of these events, the collector first checks if it passes the selection criteria of any of its active queries. If so, the collector constructs a new tuple for it, including only the fields requested in the query. It adds the request identifier, the query identifier and a timestamp, and it sends the resulting tuple to ScrubCentral.

4.2 Joins

Joins are performed in ScrubCentral. Scrub supports equi-joins between events belonging to the same request, even if they are generated by different hosts. To this end, the Scrub data collectors annotate events with a request identifier, which is used as the join key.

The Scrub join is an instance of a stream join, as there is an infinite stream of events, relating to an infinite stream of requests. To limit resource consumption, stream joins are traditionally computed over (fixed) time windows and are often approximated [5–8, 10, 16, 19, 20]. This approach does not fit Scrub's use case. With joins over time windows, events relating to the same request could fall in different time windows. Scrub therefore uses an alternative strategy, relying on the fact that it only supports joins on the request identifier and taking advantage of the short-lived nature of request execution. Since requests are handled in a short time frame, all events relating to a specific request occur within that short time frame, allowing an aggressive expiration policy for the state used to implement the join.

Scrub joins are implemented as in-memory hash joins, with an in-memory hashtable for each active join query (see Figure 4 for an example). This hashtable is created when the corresponding query object for this join arrives at ScrubCentral.

An incoming event tuple is first routed to the appropriate hash table using the query identifier attached to it by the data collector on the host. Let us consider, for instance, a join on the request identifier between two streams of events P and Q, and let us assume an incoming event tuple belongs to P. The event tuple is hashed into the hash table based on its request identifier. If this event is the first with that request identifier to arrive for this query, the Time-To-Live (TTL) field of the hash table entry is set to a small value (measured in seconds), but still much larger than the normal request lifetime. The event tuple is then inserted in the set of events belonging to P attached to this hash table entry. When the TTL expires, the cross product of the event tuples recorded for P and Q is computed. Each resulting tuple is forwarded for grouping and aggregation. Finally, the hash table entry is garbage collected.

4.3 Aggregation

There is one aggregation table per query. We use the query identifier to direct an incoming tuple to the proper aggregation table. The group-by key is hashed to find the corresponding entry in the aggregation table, and then aggregation is performed.

Conform to its query optimization strategy, Scrub avoids aggregation at the hosts to reduce any impact on the application. Events can have hundreds or thousands of fields. Queries may have group-by keys with large cardinalities, resulting in large aggregation tables. Aggregation may therefore require a large amount of memory. Implementing it on the hosts could potentially compromise application performance, or, even worse, exhaust memory and threaten availability.

5 IMPLEMENTATION

Figure 5 includes key implementation details. In particular, a metadata service stores information about hosts and event types. Zookeeper [28] is used to reliably deliver query objects to hosts and to ScrubCentral. Events sent out by the hosts are first captured in Kafka [2, 32], before they are joined and aggregated. The results are persisted in
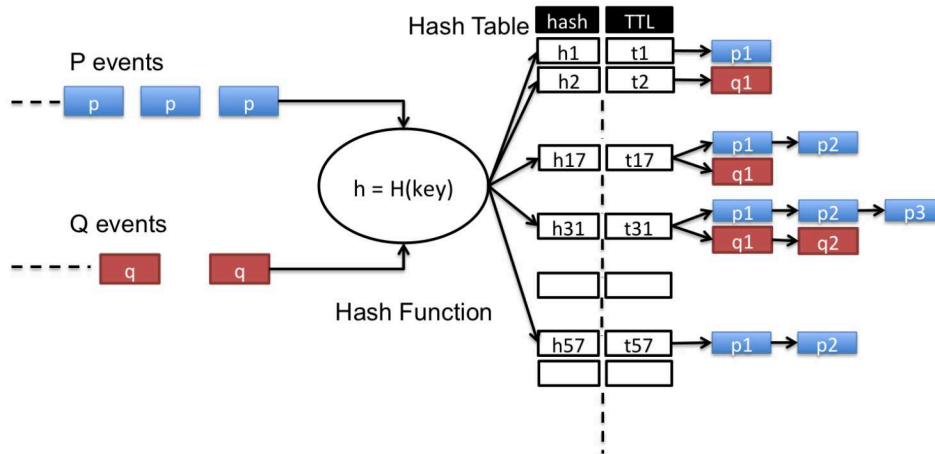
Figure 4: Implementation of hash join between two streams of events P and Q in Scrub.
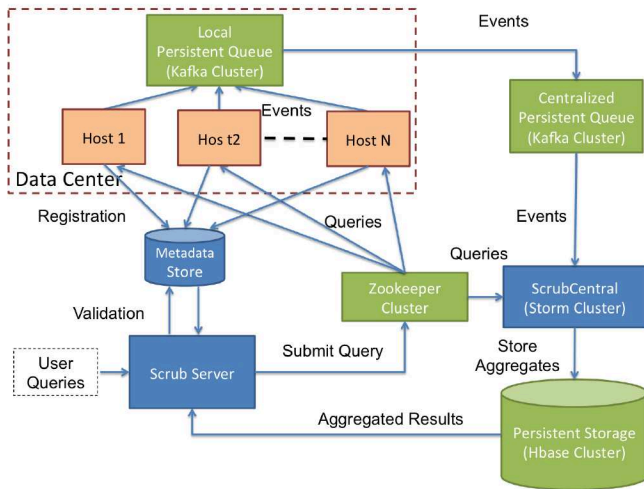

Figure 5: Scrub implementation components.

HBase [4]. We next discuss some relevant aspects of the implementation, focusing in particular on the scalability of ScrubCentral, and the tradeoff between accuracy and performance.

5.1   Metadata Service

To join the Scrub system, a host registers itself with Scrub. Registration creates a record for this host in the metadata service, including, e.g., its machine name, the service it belongs to, the data center it resides in, and attributes about the physical host such as CPU, memory, disk, etc. After a client has registered with Scrub, it can define the event types it uses. These event type definitions are also stored in the metadata service.

Metadata about hosts and event types is long-lived. Furthermore, it is queried by the Scrub server when a new query is submitted. Scrub uses the metadata to validate new queries (e.g., to verify that the fields specified in the query exist in the event types referenced) as

well as to identify target hosts for each query (e.g., all hosts belonging to a particular service). For these reasons, we use a conventional relational database for the metadata service.

5.2   Query Objects

Query objects are sent to the hosts to instruct the data collectors which data to collect. Similarly, they are needed at ScrubCentral to govern joins, grouping and aggregation. Query objects are ephemeral in nature and do not need to be queried. Hence, we simply rely on Zookeeper [28] to reliably forward them to their consumers.

5.3   Scaling ScrubCentral

5.3.1   Absorbing Incoming Events. A reliable input queue is required to absorb events arriving from the hosts. Scrub uses a Kafka cluster [2, 32] for this purpose. Kafka offers reliable, scalable, low-latency asynchronous messaging. The cluster is configured with enough buffering to absorb bursts of load until the downstream consumer, ScrubCentral in our case, can handle them. On the consuming side, ScrubCentral pulls new tuples from the Kafka cluster.

5.3.2   Scaling Joins and Aggregations. ScrubCentral is implemented as a distributed streaming Storm job [3]. Figure 6 shows its topology. The first stage, composed of a number of spout tasks, reads events from Kafka. The second stage, composed of a number of join-bolt tasks, implements the join operation. The final stage, composed of a number of aggregation-bolt tasks, performs grouping and aggregation, and writes the query results to HBase.

The spout tasks distribute the incoming tuples based on a hash of their request identifier, because all tuples with the same request identifier must be processed at the same join-bolt. Thus, every join-bolt in principle handles every query, but only for a subset of the request identifiers. Similarly, the join-bolts distribute the tuples to the aggregation-bolts based on a hash of the group-by key. All aggregation-bolts handle all queries, but only for a subset of the values of group-by key.

5.3.3   Result Storage. Query results are persisted to HBase [4, 15] for later viewing and analysis. The aggregation-bolts write their
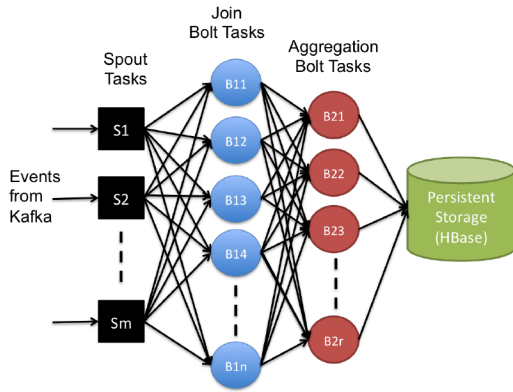
Figure 6: Topology of the Storm job implementing aggregations and joins in ScrubCentral.

output to HBase using a composite key, consisting of the query identifier, the timestamp and the group-by key, in that order. HBase's distributed index then makes these outputs appear as a time series, in which the elements are aggregates computed by the query over consecutive time windows.

5.4 Trading Accuracy

Scrub's use cases do not require exact results. Troubleshooters are mainly interested in outliers and in directional guidance about the behavior of different aspects of the system, as opposed to completely accurate results. Therefore, Scrub trades accuracy for minimal impact and performance at many levels.

First, on the hosts, application threads push events to a non-blocking ring-buffer queue. If the queue is full, events are dropped and hence application threads do not block. Scrub runs in a low-priority thread pool that drains the queue, evaluates the events against the corresponding query objects and sends them downstream to ScrubCentral. In the event of high application load, Scrub threads will yield the CPU to application threads due to their low priority. Hence, Scrub is not allowed to disrupt the application.

Second, in ScrubCentral, we disable Storm's exactly-once semantics, because it comes at a very high cost. Instead, we settle for at-most-once semantics, risking the loss of some in-flight events in return for higher throughput. Similar to what happens on the hosts, writes to HBase are asynchronous, using a ring buffer and dropping data when it becomes full.

Given the online nature of the system and the possibility of arbitrary failures, eliminating data loss entirely is virtually impossible. Moreover, in practice, failures are very rare. In our experience with the production system, events are dropped at the rate of 0.1%, mainly as a result of buffer overflows due to occasional large bursts of data. This low failure rate is consistent with what was reported previously in the literature [21]. If lower data loss rates are needed, buffer sizes can easily be extended to absorb larger load spikes.

6 DISCUSSION

In this section we highlight how our focus on minimal impact on the hosts has shaped the design of Scrub differently from that of

earlier systems. Differences exist in terms of query language, query execution strategy, and data collection method.

6.1 The Pivot Tracing "Baggage" Concept

Scrub only forwards a request identifier between different stages of execution of the same request. Other event fields useful for computing the result of a query are not forwarded along the request execution path and instead sent directly to ScrubCentral. In contrast, Pivot Tracing [35] allows an arbitrary set of attributes to be carried forward, "packed" into messages traveling along the execution path as so-called "baggage". While Pivot Tracing also supports the possibility of information being "emitted" in other ways than following the execution path, its focus is very much on the baggage approach.

Carrying arbitrary baggage allows additional functionality that is not supported in Scrub. While Scrub only supports an equi-join on the request identifier, Pivot Tracing also includes a happened-before (equi-) join on an arbitrary set of keys. Roughly speaking, a happened-before join allows two events a and b to be joined if a happens before b in the execution. For instance, if a has attributes A, B and C, and b has attributes B, C and D, then the happened-before join of a and b is A, B, C and D. A, B and C are carried as baggage as the request gets forwarded from a to b, so that they can be used in b to perform the join.

In addition to this added functionality, baggage allows Pivot Tracing to follow the more conventional query execution strategy of applying operators as close to the data as possible to optimize query performance. Unlike Scrub, where only selection and projection is performed on the host, and where joins, grouping and aggregation are performed in ScrubCentral, Pivot Tracing attempts to execute as much of the query as possible at the hosts. The baggage plays a key role in the Pivot Tracing approach, as it allows attributes of previous events in the execution to be carried forward to the next event.

Our experience in an environment where the hosts run under very tight SLOs is that the baggage approach exacts too much overhead on the hosts (see also Section 9), and one needs to be remove as much functionality as possible from the hosts. Scrub users have been able to live with the resulting restrictions imposed by the Scrub query language.

6.2 Dynamic Instrumentation

A number of systems, including Fay [22] and Pivot Tracing [35], use Dynamic Instrumentation (DI) to instantiate recording of events on the hosts. Scrub provides the hosts with a query object for each query relevant to the host, and the hosts interpret these query objects to collect data as part of a Scrub API log() call.

DI has the advantage that recording can be attached at runtime to any method in the program without any source modification. Scrub instead requires developers to indicate where in their code the recording must occur by a call to a log() method. Also, DI code can be compiled, while Scrub interprets the query object at runtime. Interpretation is slightly slower than executing compiled code, but in Section 9 we show that Scrub's overall overhead is negligible.

Although DI is more flexible, its overhead is prohibitive for production environments with tight SLOs. Java's HotSwap facility that enables DI requires a *safepoint*, pausing the entire JVM, often for
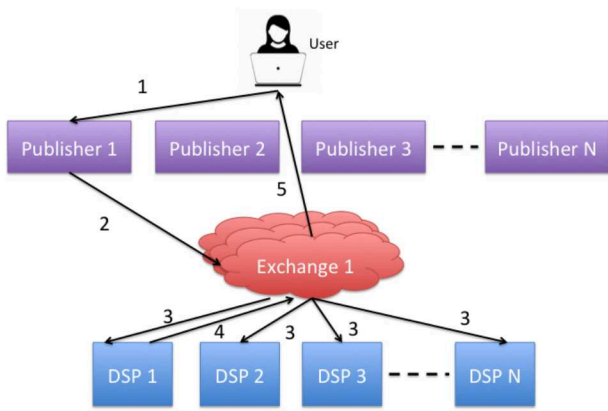
Figure 7: Simplified view of the Internet advertising ecosystem.

Figure 8: Overview of Turn's ad bidding platform and its Interaction with the Internet advertising ecosystem.

hundreds of milliseconds, which is unacceptable for Turn's mission-critical, latency-sensitive services. Scrub is heavily used at Turn by developers, operators and customers, leading to a heavy query load. If each query would lead to a HotSwap operation, the attendant overhead would cause an intolerable rate of SLO violations.

In addition to performance issues, DI also introduces security risks, as it allows for malicious code to be injected into a running production system. Various techniques have been suggested to mitigate these risks, including software fault isolation (XFI [22]), restrictions on branches, and timer interrupts to prevent injected code from running too long or monopolizing the machine by an infinite loop. The overhead of software fault isolation was judged too high for our environment. An experiment reported in Erlingsson et al. [22] shows one workload experiencing a factor of 13 slowdown due to using XFI. Catching infinite loops by using timer interrupts does not react quickly enough. For instance, the default MS Windows timer interrupt interval is 15 milliseconds, which pushes response latency well outside its SLO limits.

7 SCRUB AT TURN

We have used Scrub to troubleshoot Turn's ad bidding platform. First, we give a very simplified overview of the Real-Time Bidding (RTB) ecosystem for online advertising. Then, we give a high level overview of Turn's ad bidding platform. Finally, we explain Scrub's integration with this platform and its usage.

7.1 The Internet Advertising Ecosystem

Figure 7 provides a simplified picture of the Internet advertising ecosystem. The process of placing an ad on a web page involves three types of players. First, on the supply side, publishers (e.g., www.cnn.com) provide advertising slots during page views. Second, on the demand side, Demand Side Platforms (DSPs), such as Turn, act on behalf of advertisers and attempt to place ads in these slots. Finally, to match supply with demand, ad exchanges run an online auction for each ad slot. DSPs place bids in these auctions, the exchange chooses the winner, and the corresponding DSP gets to
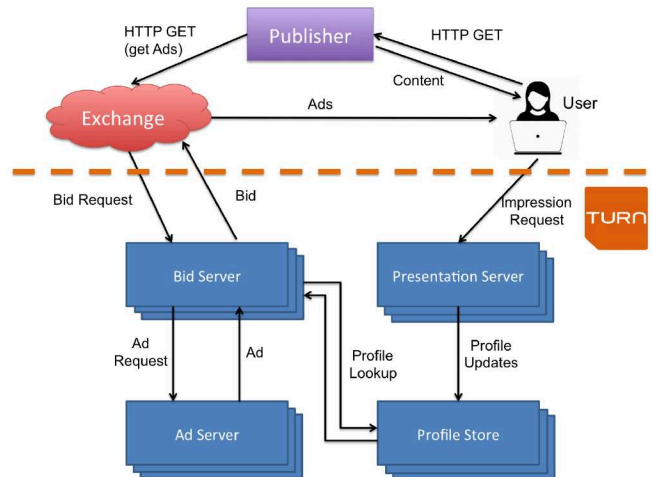
show its ad in that slot. The numbers on the arrows in Figure 7 show the typical processing sequence. In order to maintain good user experience, the whole process, from publication to display, needs to complete in about 100 milliseconds, including network latency. Many millions of these transactions are taking place every second, and billions of dollars of advertising budgets are involved.

7.2 Turn's Ad Bidding Platform

Figure 8 shows the high-level architecture of Turn's ad bidding platform.

BidServers interface with ad exchanges receiving bid requests for users viewing web pages. Using the user identifier in the bid request, a BidServer looks up the user profile in Turn's distributed ProfileStore. The profile provides user attributes like gender, age, location, etc., as well as browsing history, including previous page views, clicks, etc. After enriching the bid request with the user profile, the BidServer forwards it to an AdServer. The AdServer selects which ad to propose and sends it back to the BidServer, which in turn sends it back to the exchange in a bid response.

Advertisers launch ad campaigns, often composed of multiple *line items*. A line item specifies different images with different user targeting criteria (e.g., females living in California or males playing tennis), and possibly different constraints (e.g., maximum price per display, maximum number of displays per user per day). The AdServers perform ad selection, which proceeds in two phases: filtering and ranking. Filtering excludes the ads for which the user profile fails to match one or more criteria in the line item. For example, a male user does not qualify for an ad targeting only females. To make filtering as efficient as possible, filters that are highly selective or cheap to evaluate are checked first. In the ranking phase, line items passing the filtering process are included in an internal auction. The outcome of the auction is based on the predicted relevance of the ad to the user, as well as on the preconfigured advisory bid price for this line item. An actual bid price is then computed for the winner, based on the ad's relevance to the user and its advisory bid price. The ad and the bid price are then sent back

in the bid response to the exchange. The above transaction has to complete in under 20 milliseconds, so that the ad can be shown to the user in time.

Finally, when an ad is shown or a user interacts with it, an event is sent to Turn's PresentationServers, which record it the user's profile in the ProfileStore, and log it in Turn's data warehouse for subsequent analytics.

Scrub is integrated with the BidServers, the AdServers, the PresentationServers and the ProfileStore. Tens of Scrub event types are defined. We have already seen in Figure 1 the bid response event type generated at the BidServers. In the use cases described in Section 8 we use additional event types, such as auction and exclusion events, generated at the AdServers, and impression and click events, generated at the PresentationServers.

8 CASE STUDIES

8.1 Spam Detection

Spam is a serious problem in online advertising. A common example is bots faking ad views or clicks. DSPs try hard to protect their customers (advertisers) from such attacks. The challenge is to promptly identify the offending entities and shut them down.

In one particular incident, we suspected spam bid requests. A common spamming technique is to have bots simulating page views at high frequency, resulting in bid requests to show ads for these fake page views. We ran Scrub query in Figure 9 on one of the BidServers for 20 minutes, grouping bid requests by user identifier and counting the number of bid requests received from each user within tumbling windows of 10 seconds. Figure 10 visualizes the results, after some post-processing, in a three-dimensional plot. In the x-axis we have time, divided in 10-second windows, and in the y axis we have the logarithm of the number of bid requests received during that interval. The size of the dots reflects the number of users making that number of requests in the given window. There is a high density of large dots at one bid request per interval. In fact, in every time window, about half of the users issue a single bid request. Some users have multiple bid requests in the same time window, because many web pages show multiple ads. Nevertheless, the number of bid requests per user per window decreases exponentially. Moreover, most users issue a single batch of bid requests during the experiment's 20-minute duration, reflecting a single web page view. Some users have two batches, representing two page views, which remains consistent with human user behavior. Two users, however, exhibited a very abnormal pattern. One of these users is represented by the red triangles in Figure 10 and the other by black crosses. These users sent very large batches of bid requests at a high frequency. We concluded that these are bots, not human users. Consequently, we quickly blacklisted these users, stopping any ads from being served to them.

To demonstrate Scrub's effectiveness, we contrast using Scrub with the traditional way of tracing this problem using logging. Since queries are not known a priori, all data would need to be logged. Moving all this data over cross-continental links to a centralized location for analysis would be very costly, retaining it for any length of time even more so. To run the above query in batch mode on 20 minutes worth of data would require a large Hadoop cluster. The cost of doing so limits the number of queries that can be run in a given amount of time. While the query is running, the problem persists,

```
Select bid.user_id, COUNT(*)
from bid
@[Service in BidServers and Server = host1]
group by bid.user_id;
```

Figure 9: Query used to troubleshoot spam bots.

accumulating financial losses as a result. In contrast, with Scrub the problem as well as the offenders were detected very quickly, allowing for prompt corrective action. Moreover, only a small Scrub-Central cluster was needed to execute this query, making it very cost-effective.

8.2 Validating New Ad Exchanges

Over time new ad exchanges join the online advertising ecosystem. DSPs integrate with these new exchanges as they come up. After integration, the DSPs verify that the integration went well, by monitoring key metrics, such as the number of bid requests and impressions received and the amount of budget spent.

Figure 11 demonstrates a query used for this purpose. It counts the number of impressions per exchange. Since only statistical and not exact total information is required, the query samples 10% of the impression events in 10% of the PresentationServers in data center DC1.

Figure 12 shows the result of executing this query during a time interval in production when a new ad exchange came online. The x-axis shows time measured in seconds, and the y-axis shows the number of impressions served from four exchanges *A*, *B*, *C*, and *D* aggregated over 10-second windows. Exchange *A* was introduced at time 550. From that time on, we see a large number of impressions served by *D*, indicating a healthy integration.

This experiment demonstrates the effectiveness of Scrub in getting realtime results from the bidding platform, while in production. Even though the platform is distributed across the globe, Scrub was able to quickly validate the correctness of the integration with the new exchange.

8.3 A/B Testing of Ad Targeting Models

This experiment demonstrates the effectiveness of Scrub for A/B testing in production. Specifically, we ran a new ad targeting model *A* on a subset of machines, and used Scrub to measure its effectiveness against the incumbent model *B* running on the remaining machines. Ad targeting models try to target the right users for a particular ad, for instance seeking to optimize the Click Through Rate (CTR), while keeping the cost per impression constant. The CTR is defined as the fraction of clicks on an ad per impression. The cost per impression is usually measured by the industry-standard CPM value, the cost per thousand impressions. We ran the Scrub queries in Figures 13 and 14, each computing the daily average CPM and CTR values for a particular ad, with one query targeting the servers running model *A* and the other targeting the servers running model *B*. Figure 15a shows the measured CPM for both models, and Figure 15b shows the CTR. *B* achieved higher CTR than *A*, while keeping the CPM more or less the same, which is exactly what was desired.
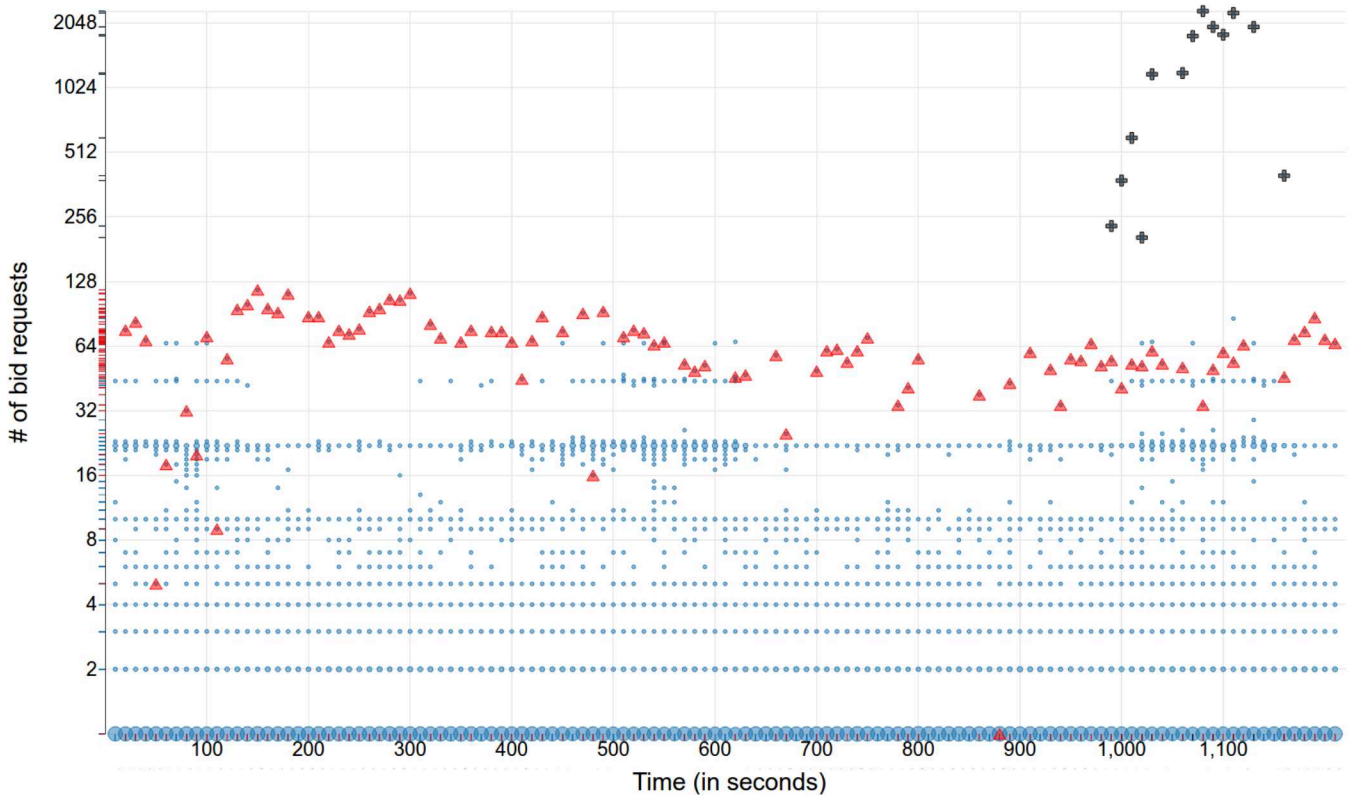
Figure 10: Identifying bots in the bid request Stream. Each dot in the figure represents a group of users making the same number of requests (shown on the logarithmic y-axis) during the same 10-second window (shown on the x-axis). The size of the dots is proportional to the number of users in each group. The red triangles and the black crosses each represent a group of one single user making a suspiciously large number of requests at high frequency. Hence, these users are identified as bots.

```
Select exchange_id,   COUNT(*)
from impressions
@[Service in PresentationServers
        and DataCenter = DC1
        and SampleServers = 0.1
        and SamplesPerServer = 0.1]
group by exchange_id;
```

Figure 11: Query used to validate the successful integration with a new ad exchange. A 10% sample of received impressions at 10% of the PresentationServers sufficed for this use case.
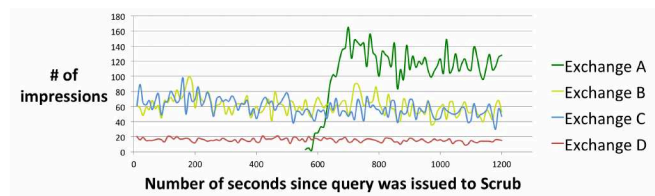

Figure 12: With the x-axis showing time measured in minutes, the y-axis represents the number of impressions aggregated over 10-second windows served from different exchanges, with exchange A added at time 550. This use case demonstrates that Scrub can detect a new exchange within a few seconds after its integration in Turn's platform.

8.4 Exclusions

As explained in Section 7, when a bid request is received, ads are excluded if the user profile does not match one or more criteria in the line item. The system has a list of about 100 predefined exclusions abstracting different types of filters, e.g., geography, gender, age, etc. For a line item that is often excluded from bid responses, it is often useful to understand the reasons for this repeated exclusion. This information can help tune the targeting criteria of the line item to achieve better performance. Similarly, when integrating with a new

exchange or targeting a new publisher, it is important to check that the distribution of exclusions is healthy.

The bidding platform exposes an event about each excluded line item, including the reason for its exclusion, i.e., which of its targeting criteria the user failed. To make the bid request handling as efficient as possible, only one exclusion is emitted for each excluded line item. Figure 17 gives a template for the Scrub queries that we often use to study the distribution of the reasons for a line item's exclusion.

```
Select 1000*AVG(impression.cost)
from impression
where impression.line_item_id = id
@[Servers in (list)];
```

Figure 13: Query template used to support A/B testing of ad targeting models by computing the CPM of different models. *id* **is the line item of interest, and** *list* **is a parameter used to select the set of machines running the desired model, i.e.,** *A* **or** *B***.**

```
Select COUNT(*)
from event
where event.line_item_id = id
@[Servers in (list)];
```

Figure 14: Query template used to support A/B testing of ad targeting models by computing the CTR of different models. *event* **is either clicks or impressions,** *id* **is the line item of interest, and** *list* **is a parameter used to select the set of machines running the desired model, i.e.,** *A* **or** *B***. The CTR is computed by dividing the count of clicks by the count of impressions.**

In this template, queries run at the BidServers and AdServers in DC1. The query template equi-joins events belonging to the same request, but one event type (bid) is generated at the BidServers and the other (exclusion) is generated at the AdServers. Selection on the set of fields *A* allows us to narrow down the results, For instance, selecting on *bid.exchange_id* gives us results for bids from a particular exchange. Figure 16 shows production results about the number of occurrences of line item exclusions for a particular exchange and a publisher. These distributions are then compared to corresponding distributions of well-behaved line items to identify anomalies.

This case study illustrates Scrub's scalability. At any given time, there are usually several tens of thousands of active line items. The vast majority of them do not pass the filtering process. Hence, every bid request produces tens of thousands of exclusions. If logging were used, it would result in a enormous data set. Similarly, if baggage propagation were used, the baggage would have to include all these exclusions and pass them from the AdServers to the BidServers. In contrast, Scrub queries the needed data on demand.

8.5 Line Item Cannibalization

After passing the filtering phase, line items go through an internal auction. There, using machine learning models, line items are assigned scores predicting how likely the user is to interact with their ad. Based on the scores as well as on a preconfigured advisory bid price for each line item, a winner is chosen and sent in the bid response. The bid price used in the bid response is based the advisory price, but adjusted depending on the score. Hence, in practice, the bid prices for a line item winning an internal auction move in a narrow band around the preconfigured advisory price.

If two line items, *A* and *B*, have similar targeting criteria, they are likely to pass the filtering phase together and compete in the auction.

If *A* has a significantly higher advisory bid price, its entire band of bid prices is likely to be higher than *B*'s entire price band. As a result, *A* ends up having precedence over *B*, "cannibalizing" it by preventing it from making bids and hence having a chance to show its ad. These conditions are hard to detect at campaign creation time as different line items may be created by different people. Moreover, even if the targeting criteria of two line items look different, they may act similarly, because the differences may be inconsequential for the user population in the bid requests. These situations need to be detected at runtime to make prompt corrective actions.

To give a concrete example, one advertiser reported that one of its line items λ was not serving ads, even though it had budget and fairly relaxed targeting criteria. After studying the exclusions to verify that it was not being excluded at the filtering phase, we ran the query in Figure 19 to investigate a possible cannibalization scenario.

An event of type auction is generated by the AdServers for every internal auction. An auction event includes the list of line item identifiers participating in the auction, each with its bid price. Impression is an event type generated by the PresentationServers, after the ad has been served to the user. Hence, it includes the identifier of the line item that won both the internal and external auctions. The query identifies line items winning at an internal auction, where λ is a participant. For each line item, the query computes the number of times it won and its average winning bid price. Figure 18 plots the output of this query running for an hour in production. Figure 18a shows the number of times a line item wins the auction, while Figure 18b gives the average winning bid prices for the corresponding line item. We noticed that λ advisory bid price was much lower than all winning bid prices in the auctions in which it participated, explaining its cannibalization. In response, we bumped up its advisory bid price, and immediately it started delivering ads.

This case study demonstrates Scrub's effectiveness in realtime troubleshooting as well as in scalability. It was critical to detect the problem promptly to allow the campaign to meet its goals in the desired time frame. In terms of scalability, logging auction events with information about all line items participating in the auction for every bid response would have been prohibitively expensive given the sheer volume of data it would entail.

8.6 Incorrectly Set Field

Contrary to the previous case studies that troubleshoot campaign performance, in this case a software developer was debugging a software problem. A Turn customer had configured a campaign to show ads with a maximum frequency of one ad per user per day. Using Turn's campaign reporting and analytics tools, the customer, however, noticed that some users received ads at higher frequencies.

Turn's platform records in the user's profile in the ProfileStore the number of times an ad has been served to this user. Since each bid request includes the user identifier, whenever a user is served an ad, the count for this ad for this user is incremented. This information is then used in the filtering phase for subsequent bid requests. When a new bid request is received, line items whose ads have met their frequency caps are filtered out. Since we had not made any changes in the code for maintaining these frequencies, we suspected that the problem resulted from erroneous input data. If, for instance, a
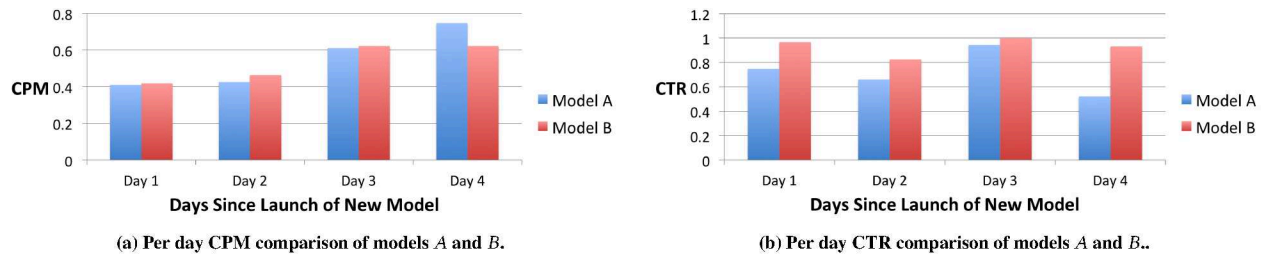
(a) Per day CPM comparison of models A and B.

(b) Per day CTR comparison of models A and B..

Figure 15: A/B testing of new ad targeting models. A new model B is expected to spend the same amount of money (CPM), but have a higher CTR than model A.

(a) Normalized histogram of exclusions for a new exchange.

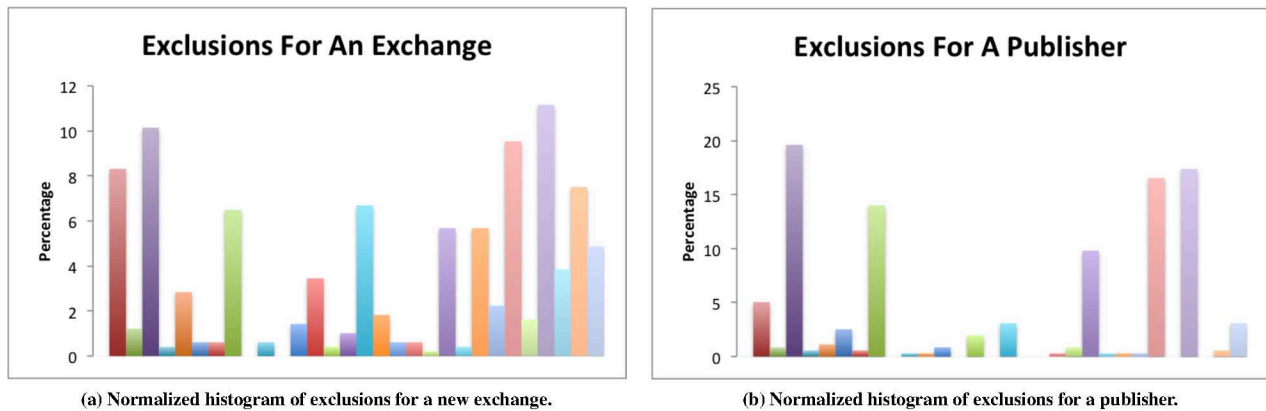(b) Normalized histogram of exclusions for a publisher.

Figure 16: Distribution of exclusions for an exchange and a publisher. Exclusion names are omitted for confidentiality, but they use the same color coding for each of the three experiments. Also, only exclusions with non-zero values in any of the experiments are shown.

```
Select exclusion.exclusion_id, COUNT(*)
from bid, exclusion
where exclusion.request_id = bid.request_id
        and A = α
@[Service in (AdServers, BidServers)
        and DataCenter = DC1]
group by exclusion.exclusion_id;
```

Figure 17: Query template for queries used to study exclusions.

bid request had the wrong user identifier, it would throw off the frequency capping logic.

To find the source of this problem, we had to correlate the bid response events at the BidServers and the impression events at the PresentationServers. When an impression is served, the user's browser requests the ad media file via an HTTP GET request from Turn's PresentationServers. Turn's cookies are passed along with the GET request in the HTTP header. Turn records the user identifier in its cookie at the user's browser. Hence, it receives the user identifier, when an impression is served.

The query shown in Figure 20 counts the number of instances in which the user identifiers received from the exchange in the bid request is different from the user identifier stored in the cookie if the bid results in an ad from Turn being shown. To limit the amount of data generated and the impact on the system, this query selects for the problematic line item, λ, from the complaining advertiser.

The query produced non-zero counts, signaling mismatches. We inspected the offending bid requests, and found that they did not include a user identifier. We contacted the exchange and explained the bug, which allowed them to promptly fix it. This way the campaign was quickly put back on track. This case study again demonstrates Scrub's effective realtime debugging, but it also demonstrates Scrub's ability to correlate events occurring in different administrative domains, in this case Turn, the ad exchange and the users' web browsers.

9 EVALUATION

9.1 Performance Evaluation

The following experiments used test machines with 8-Core Xeon CPUs, 32 GB of RAM, and 1 Gb links, and running Java 8.
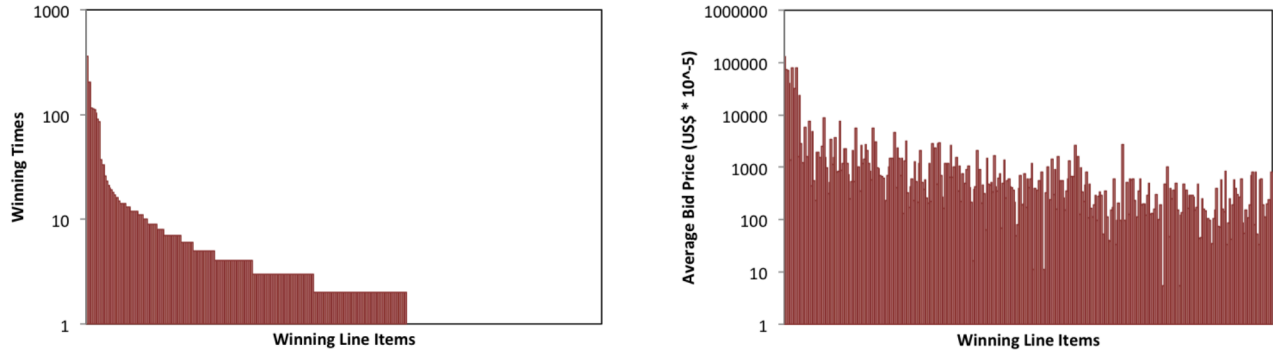
(a) Count of times each line item winning an internal auction, in which λ is a participant.

(b) Average bid price for each line item winning an internal auction, in which λ is a participant.

Figure 18: Line items cannibalizing λ. Each bar represents a line item. Line items have the same order in both graphs (y-axis in logarithmic scale.

```
Select impression.line_item_id, COUNT(*),
        AVG(auction.winning_line_item.bid_price)
from auction, impression
where auction.request_id = impression.request_id
        and λ in auction.line_items
        and λ != impression.line_item_id
@[Service in (AdServers, PresentationServers)
        and DataCenter = DC1]
group by impression.line_item_id;
```

Figure 19: Query used to troubleshoot the line item cannibalization problem.

```
Select impression.line_item_id, COUNT(*)
from bid, impression
where bid.request_id = impression.request_id
        and bid.user_id != impression.user_id
        and impression.line_item_id = λ
@[Service in (BidServers, PresentationServers)
        and DataCenter = DC1]
group by impression.line_item_id;
```

Figure 20: Query used to troubleshoot the problem of a field in the bid request incorrectly set by an ad exchange.

9.1.1 Overhead on the Hosts. We measure the CPU overhead on a production BidServer that handles about 10,000 requests per second. We collect bid events, and we vary the number of fields projected from 1 to 10. The resulting overhead ranges between 1.5% and 2.5%, which we consider to be acceptable.

We also study the overhead of collection on request latency. If no query is running against the logged event, *log()* just checks a condition and returns. Otherwise, it constructs a tuple and pushes it

Number of group-by keys	Number of aggregates	Maximum Throughput
0	1	405,658
1	1	361,285
2	1	300,232
3	1	282,711
4	1	271,601

Table 1: Maximum throughput in events per second for queries with varying number of group-by keys and a single aggregate.

into a ring buffer. We measure the distribution of *log()* latency at 10 hosts over a duration of one week. Median latency was under 50us, as most logged events have no queries running against them. The 99.9% latency is under 200us. Hence, Scrub's added overhead to request latency is negligible, and does not affect the ability of the system to satisfy its SLO.

9.1.2 Throughput of ScrubCentral. To study the maximum throughput Scrub can sustain, we use a single-node test cluster to run ScrubCentral. Obviously, the throughput decreases if there are more fields in the events to be processed, so we exercise ScrubCentral with different queries to vary the number of fields. We vary the total number of fields from 1 to 5, by increasing the number of group-by keys from 0 to 4, and keep the number of aggregates fixed at 1. Table 1 shows the resulting throughput, expressed in events per seconds. For one field per tuple, ScrubCentral can process over 400,000 events per second. With more fields per tuple, there is a modest drop in throughput.

Early projection and selection at the hosts dramatically reduces the number of fields in events arriving at ScrubCentral. In production, we have observed a median of 4 fields per event, with a maximum of 10, compared to the raw events, which can have hundreds of fields. Given these observations, a ScrubCentral cluster consisting of a handful of machines can handle Turn's traffic rates.
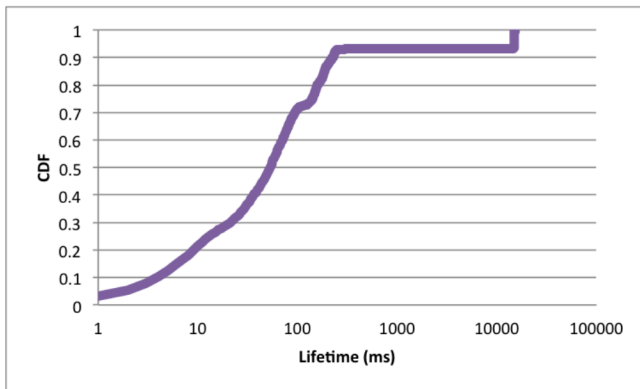
Figure 21: CDF of the lifetimes of entries in the hash table implementing Scrub's join (x axis on a logarithmic scale).

9.2 Lifetime of Join Hash Table Entries

To understand what TTL value is suitable for the hash tables implementing Scrub's join, we measure the distribution of the lifetimes of the hash table entries. The lifetime is defined as the time difference between the first and the last insertion of an event in a hash table entry. The join occurs when the TTL of an entry expires, which is increased to 30 seconds for this experiment.

Figure 21 shows the CDF for the lifetime distribution. The maximum value is under 15 seconds, and the 90th percentile is at 225ms. Consequently, a conservative TTL value of 20 seconds avoids almost all data loss, and produces an exact join in the overwhelming majority of cases. Even if a small fraction of events is lost, it is inconsequential, as our use cases usually look for directional guidance, as opposed to exact answers.

We elaborate on a couple of aspects of the lifetime CDF. First, some lifetimes are much larger than production request latencies. Some queries join impression events with bid events. An impression event is generated when an ad is served, which can be several seconds after when the bid request is received. Moreover, events go over transcontinental links from hosts to ScrubCentral in a different data center, introducing further latency. Second, there is a big jump in the curve at the 93 percentile, from 393 milliseconds to 14.8 seconds. This jump is the result of video ads. Impression of video ads takes longer, because the user needs to explicitly play the ad before it is counted as an impression event.

9.3 Comparison To Baggage Propagation

We study the overhead that Turn's workload would incur if it were to use the baggage approach [35]. We modified an AdServer and a BidServer to simulate baggage propagation by adding the baggage to the RPC parameters. The AdServer serializes the baggage and passes it back in the response to the BidServer, which deserializes and scans it. The bulk of the latency is due serialization and deserialization overhead. Network transmission adds only a small amount to the overall latency.

We use the query from Figure 17 from Section 8.4. The query is used to evaluate the health of the exclusion stage for a given ad, and it refers to the request identifier and the exclusion type. Thus,

Baggage Size (Tuples)	Mean Request Latency (ms)
0	20.3
10	20.9
100	26.8
1,000	78.7
10,000	523.3

Table 2: Mean request latencies in milliseconds with baggage propagation, when propagating exclusion events. Means are computed over 10,000 runs per data point. Each tuple carries 10 bytes.

each tuple only includes these two fields, for a total of 10 bytes per tuple. In the worst case, there can be an exclusion for each line item in the system. At Turn, there are currently tens of thousands of line items, and that number is growing. All these exclusions have to be included in the baggage.

Table 2 shows the mean request latency, when varying the baggage size by increasing the number of tuples included. For each data point, latency is averaged over 10,000 requests. With baggage consisting of 1,000 tuples, latency grows by a factor of 3; with 10,000 tuples by a factor of 25. These overheads would break Turn's request latency SLO.

In contrast, Scrub introduces negligible overhead to the request latency for the same workload. Specifically, for 10,000 tuples Scrubs adds under 1 millisecond of overhead to the request processing time, because the only thing Scrub does on the request's critical path is to enqueue these tuples into a ring buffer.

10 RELATED WORK

Debugging distributed systems is an important and difficult problem. Hence, it received a lot of attention in the literature. For example, Causeway [13, 14], Whodunit [12], XTrace [23], Dapper [41], and The Mystery Machine [17] have focused on this problem. They all tried to track requests and their corresponding responses across different tiers of a distributed system. A key use case was to understand where time is spent processing different requests. Magpie [9], collects events generated by the operating system and the middleware and tries to capture the control flow and resource consumption of each and every request. All these systems relied on the offline postmortem processing of generated logs. Inspector gadget [39] focused on a special class of distributed applications, which is distributed data flows. It basically attempts to monitor and debug distributed batch data flows by instrumenting the processing engine to track data as it flows through the pipeline.

With the recent growing needs for big-data stream processing needs, multiple systems using scale-out architectures came out from both industry and academia [18, 33, 42, 45].

Approximate query processing is not new [1, 11, 24]. It mainly focused on aggregation queries and it exploits the observation that many queries can afford trading accuracy for performance. They rely on constructing data synopsis and then querying them for better performance. Online aggregation [26] also provides approximate answers to aggregation queries. However, as an online-aggregation query processes more data, the aggregation result is refined to finally arrive at the exact answer after the entire input data is processed.

None of this applies to streaming though. Conversely in Scrub, inputs are sampled, either by sampling the data sources or sampling records per source.

Kodiak [34] provides a platform to process high-dimensional, large-scale event data. Its primary use case was online advertising too. However, it was targeting offline analytics and hence relied on batch processing.

11 CONCLUSIONS

Scrub is an online troubleshooting tool for large-scale distributed applications that operate under tight SLOs common in production environments. This environment imposes on the troubleshooting tool the requirement that it only minimally impacts the hosts on which the application runs. This requirement is reflected in Scrub's design and implementation in a number of ways. Joins in its query language are restricted to equi-joins on a request identifier. Query execution is largely performed in a centralized entity, and not on the hosts. Where necessary, accuracy of the query results is traded for minimal impact on the hosts.

At the time of writing, Scrub has been in production use for two years with Turn's ad bidding platform. Given the large amounts of money at stake in this application, users demand quick problem detection and resolution. Offline analysis of logs is not an option in this environment.

This paper presents Scrub's architecture, explains its design choices, and describes its integration in Turn's ad bidding platform. We demonstrate its power by presenting a number of use cases where complex problems were quickly resolved with Scrub's help. Furthermore, we show that Scrub has negligible impact on the application hosts.

ACKNOWLEDGEMENTS

REFERENCES

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. The Aqua Approximate Query Answering System. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. ACM, New York, NY, USA, 574–576. https://doi.org/10.1145/304182.304581

[2] Apache [n. d.]. Apache Kafka: A high-throughput distributed messaging system. http://kafka.apache.org/. ([n. d.]).

[3] Apache [n. d.]. Apache Storm: A distributed realtime computation system. http://storm.apache.org/. ([n. d.]).

[4] Apache [n. d.]. HBase: the Hadoop database. http:///hbase.apache.org/. ([n. d.]).

[5] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. 2002. Characterizing Memory Requirements for Queries over Continuous Data Streams. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*. ACM, New York, NY, USA, 221–232. https://doi.org/10.1145/543613.543642

[6] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/342009.335420

[7] Ahmed M. Ayad and Jeffrey F. Naughton. 2004. Static Optimization of Conjunctive Queries with Sliding Windows over Infinite Streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, New York, NY, USA, 419–430. https://doi.org/10.1145/1007568.1007616

[8] Shivnath Babu and Jennifer Widom. 2001. Continuous Queries over Data Streams. *SIGMOD Rec.* 30, 3 (Sept. 2001), 109–120. https://doi.org/10.1145/603867.603884

[9] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251254.1251272

[10] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2002. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. VLDB Endowment, 215–226. http://dl.acm.org/citation.cfm?id=1287369.1287389

[11] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2000. Approximate Query Processing Using Wavelets. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 111–122. http://dl.acm.org/citation.cfm?id=645926.671851

[12] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. 2007. Whodunit: Transactional Profiling for Multi-tier Applications. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1272996.1273001

[13] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. 2005. Causeway: Operating System Support for Controlling and Analyzing the Execution of Distributed Programs. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10 (HOTOS'05)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251123.1251141

[14] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. 2005. Causeway: Support for Controlling and Analyzing the Execution of Multi-tier Applications. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware (Middleware '05)*. Springer-Verlag New York, Inc., New York, NY, USA, 42–59. http://dl.acm.org/citation.cfm?id=1515890.1515893

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Computer Systems* 26, 2 (2008).

[16] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 379–390. https://doi.org/10.1145/342009.335432

[17] MIchael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. 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)*. USENIX Association, Broomfield, CO, 217–231. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow

[18] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1855711.1855732

[19] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. 2003. Approximate Join Processing over Data Streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 40–51. https://doi.org/10.1145/872757.872765

[20] Amol Deshpande and Joseph M. Hellerstein. 2004. Lifting the Burden of History from Adaptive Query Processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 948–959. http://dl.acm.org/citation.cfm?id=1316689.1316771

[21] Khaled Elmeleegy. 2013. Piranha: Optimizing Short Jobs in Hadoop. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 985–996. http://dl.acm.org/citation.cfm?id=2536222.2536225

[22] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. 2011. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 311–326. https://doi.org/10.1145/2043556.2043585

[23] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. USENIX Association, Berkeley, CA, USA, 20–20. http://dl.acm.org/citation.cfm?id=1973430.1973450

[24] Minos N. Garofalakis and Phillip B. Gibbon. 2001. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 725–. http://dl.acm.org/citation.cfm?id=645927.672356

[25] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York,

NY, USA, 383–397. https://doi.org/10.1145/2694344.2694351

[26] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/253260.253291

[27] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. ACM, New York, NY, USA, 683–692. https://doi.org/10.1145/2452376.2452456

[28] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. 11–11.

[29] Yannis E. Ioannidis. 1996. Query Optimization. *ACM Comput. Surv.* 28, 1 (March 1996), 121–123. https://doi.org/10.1145/234313.234367

[30] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 344–360. https://doi.org/10.1145/2815400.2815412

[31] Soila P. Kavulya, Scott Daniels, Kaustubh Joshi, Matti Hiltunen, Rajeev Gandhi, and Priya Narasimhan. 2012. Draco: Statistical Diagnosis of Chronic Problems in Large Distributed Systems. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. http://dl.acm.org/citation.cfm?id=2354410.2355155

[32] J. Kreps, N. Narkhede, and J. Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*.

[33] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[34] Shaosu Liu, Bin Song, Sriharsha Gangam, Lawrence Lo, and Khaled Elmeleegy. 2016. Kodiak: Leveraging Materialized Views for Very Low-latency Analytics over High-dimensional Web-scale Data. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1269–1280. https://doi.org/10.14778/3007263.3007266

[35] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 378–393. https://doi.org/10.1145/2815400.2815415

[36] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT'05)*. Springer-Verlag, Berlin, Heidelberg, 398–412. https://doi.org/10.1007/978-3-540-30570-5_27

[37] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. 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)*. USENIX Association, Berkeley, CA, USA, 26–26. http://dl.acm.org/citation.cfm?id=2228298.2228334

[38] A. J. Oliner, A. V. Kulkarni, and A. Aiken. 2010. Using correlated surprise to infer shared influence. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 191–200. https://doi.org/10.1109/DSN.2010.5544921

[39] Christopher Olston and Benjamin Reed. 2011. Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 1221–1224. https://doi.org/10.1145/1989323.1989459

[40] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. 2005. Experiences with Pip: Finding Unexpected Behavior in Distributed Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/1095810.1118601

[41] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[42] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[43] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 117–132. https://doi.org/10.1145/1629575.1629587

[44] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1950365.1950369

[45] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[46] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 629–644. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhao