Distributed tracing is a core component of cloud and data-center systems, and provides visibility into their end-to-end runtime behavior. To reduce computational and storage overheads, most tracing frameworks do not keep all traces, but sample them uniformly at random. While effective at reducing overheads, uniform random sampling inevitably captures redundant, common-case execution traces, which are less useful for analysis and troubleshooting tasks.

In this work, we investigate biased trace sampling, whose goal is to capture qualitatively more diverse traces, by weighting sampling decisions towards edge-case code paths, infrequent request types, and anomalous events. We present two different sampling approaches, one of which is based on the trace comparison while the other relies on modeling the system’s behavior to infer the sampling probability of a trace. The first approach, SolutionA, uses a hierarchical clustering algorithm, that depends on pairwise comparison between traces, to derive the sampling probability of each trace. The other approach, SolutionB, uses the incoming stream of traces to build an unbiased low-dimensional model that approximates the system’s common-case behavior and then, to bias the sampling decisions towards traces that are poorly captured by such model.

We have implemented both approaches and integrated them with several open-source tracing systems, and evaluate them with traces from a range of open-source and production distributed systems. Our evaluation shows that both strategies effectively create a bias towards anomalous and outlier executions and are robust in case of noisy and heterogeneous traces. We also showed that SolutionB is more efficient and scalable, being more suitable for production environments.

1 Introduction

Distributed systems are continuously growing both in scale and complexity, reaching from hundreds to thousands of different components. Given the inherent heterogeneity and complexity of those environments, errors/faults become more likely to occur [28, 48]. These problems may involve multiple sources, such as software and hardware, may be due to faults or misconfiguration, may be contained in numerous system components or may be the result of interactions between various parts of the system. Such problems tend to be difficult to handle, varied and unpredictable [25].

End-to-end tracing has emerged as a valuable tool for distributed systems by recording, diagnosing and analyzing their execution across components. Distributed tracing is widely deployed in all major internet companies [18, 42], and there are several popular open-source variants such as OpenTracing [36], Jaeger [16], and Zipkin [46]. Those frameworks are useful for a range of end-to-end modeling [32], analysis [39], and troubleshooting tasks [12, 42], because they capture the full end-to-end flow of a request as it traverses the distributed system.

Capturing, processing, and storing traces is computationally expensive, especially for production workloads. To handle this, most tracing systems today do not trace everything; instead they sample traces uniformly at random, which reduces computational overheads proportional to the sampling rate. This sampling is done at the granularity of requests – either the end-to-end request trace is kept, or discarded. While uniform random sampling is effective at reducing overheads, it fails to take into account the utility of the traces it samples. Uniform random sampling will naturally capture more traces of common-case executions than it will of edge-case or anomalous executions, since ‘normal’ executions are far more prevalent in the workload. On the other hand, the most valuable traces for analysis and troubleshooting are traces of edge-case and anomalous executions.

In this paper, we investigate biased trace sampling, where traces are sampled proportional to how ‘interesting’ their content is. Biased trace sampling increases the overall utility of traces that are sampled, by reducing the prevalence of redundant common-case traces. However, biased trace sampling is not straightforward and we face several challenges. Sampling cannot rely on manually engineered features; this is brittle and time consuming for developers, and does not automatically adapt to new or unexpected cases. Sampling decisions must be fast and scalable, which is challenging given that traces are richly annotated graphs. Also, sampling decisions must be robust to handle noise and heterogeneity in the trace data: qualitatively similar traces often differ in subtle ways, due to transient issues like timing, and the detail and verbosity of traces vary because they combine information across many independently-instrumented components. Lastly, sampling techniques face strict operational requirements to be useful in practice: sampling decisions must be fast (on the order of milliseconds), techniques must scale to a large volume of traces, and they must operate online over a continuous stream.
of traces.

We addressed these challenges by proposing two different biased sampling approaches. The first approach, SolutionA, is based on pairwise comparison between traces. It performs a hierarchical clustering that organizes the traces and samples them in the inverse proportion of the size of the cluster that a trace belongs to, giving a higher weight to edge-case executions. The second approach, SolutionB, is a general-purpose trace sampler that automatically biases sampling decisions towards outlier and anomalous traces. SolutionB operates on a continuous stream of traces, and its computational cost is fixed with respect to both workload volume and sampling rate. The intuition behind SolutionB is to approximate the distributed system’s common-case behavior, and to sample new traces based on how well represented they are. The two approaches are significantly different. While SolutionA depends on previously sampled traces to make each sampling decision, SolutionB relies only on the model representing the system’s behavior. Therefore, SolutionB has fixed computational costs, regardless of the number of sampled traces. On the other hand, the computational costs for SolutionA are directly related to the number of traces to be sampled. Our results show that SolutionA is particularly efficient in identifying anomalies, however, SolutionB’s overall sample sets are more representative. In addition, SolutionB overheads are lower, which makes it a better option for production systems.

We have implemented both SolutionA and SolutionB in Python, and support sampling for X-Trace [12], Jaeger [16], and Zipkin [46] traces. To evaluate their ability to bias sampling decisions, we use a number of trace datasets from several different distributed systems, including the open-source systems HDFS [41], YARN [47], and Spark [49]; the Death- Star social network benchmark [14]; and production traces from a large internet company.

In summary, the main contributions of this paper are:

- We compare and contrast a sampling approach based on pairwise comparison between traces with another approach that relies on modeling the system’s behavior.
- We present SolutionA, a framework based on a hierarchical clustering algorithm, that relies on pairwise comparison between traces to bias sampling decisions toward infrequent traces.
- We describe the design of SolutionB, a framework that simultaneously models a system’s common-case behavior and makes sampling decisions for traces of that system.
- We implement SolutionA and SolutionB and integrate them with several open-source tracing frameworks.
- We demonstrate the feasibility of both SolutionA and SolutionB on a range of trace datasets including open-source and production distributed systems.
- We demonstrate that SolutionB meets the two main requirements that sampling frameworks need to be used in production systems: quality sampling decisions and constant computational cost.

## 2 Distributed Tracing Background

Traditional approaches for monitoring and debugging - *ad-hoc* analysis and per-machine logging - were designed for monolith systems, to observe the behavior of a single application instance, scaling poorly when used in complex distributed systems. Distributed tracing has emerged as a valuable tool for improving the dependability of these systems by recording, diagnosing and analyzing their execution across components. Tracing takes a request-centric view. It captures both the events that occur during request execution, as well as the causal ordering and concurrency of these events, according to the Lamport’s happens-before relation [21], and combine this information across all distributed components traversed by the request.

Distributed tracing is useful for diagnosing a range of correctness and performance problems in production systems, including diagnosing anomalous requests whose structure or timing deviates from the norm, diagnosing steady-state problems that manifest across many requests, identifying slow components and functions and modeling workloads and resource usage. Today, distributed tracing tools are deployed at all major internet companies [16, 18, 42]. Open-source distributed tracing tools are widely deployed, with notable examples including OpenTracing [36], Jaeger [16], and Zipkin [46].

### 2.1 A Primer on Traces

To better motivate the challenges of sampling traces, we begin with an overview of distributed traces. A common representation for requests recorded by tracing frameworks is an *execution graph*, a directed, acyclic graph that describes the path of a single request through components of a distributed system, composed by events and their relationship.

**Events** Events are the core building block of a trace. Events occur during a request’s execution, they are instantaneous in time, and they can be likened to logging statements. A request trace comprises all events generated during the request’s execution, from all processes and machines where the request executed. When events are generated, they are sent to the tracing framework’s backend; the backend receives all of the events and stitches them into a trace.

Events convey a variety of information. They carry useful diagnostic messages, indicate important control flow points in a program, capture timing information, log concurrent communication, report performance counters, log exceptions, and more. Events often carry auxiliary information such as timestamps, hostnames, process IDs, thread IDs, and so on. Traces can be arbitrarily large, depending on the amount and detail of instrumentation, and length of execution. For example, traces at Facebook contain several thousand events [18].

**Relationships** Events alone tell some of the story, but the most important information in a trace is the causal ordering
which illustrates the typical tracing pipeline. First, traces are generated on the critical path of requests in the distributed system. This incurs runtime overhead to generate trace data, and for propagating trace contexts alongside requests. For a single request, trace data will be spread across all components traversed by the request. Second, a pub-sub system such as Kafka [1] or Scribe [17] will route trace data to tracing backends, incurring network overheads. Third, tracing backends receive trace data and aggregate it in memory for a short period of time before then processing it. Tracing backends are shared-nothing and can easily be shared using the request ID contained in the trace data. Thus, despite originating from many disparate locations, trace data for any given request will consistently arrive at only a single backend instance. Processing a trace incurs computational costs for constructing its abstract representation and applying feature-extraction functions. There is no need for communication between backend instances – each trace is processed in isolation. Lastly, the trace data and the extracted features are forwarded to long-term storage, where they can later be queried and analyzed.

### 3 Sampling Overview

Sampling of traces is the focus of this paper, as it is the prevailing approach to reduce tracing overheads. Instead of tracing every request, the distributed tracing tools only capture and persist traces for a subset of requests to the system [42]. To ensure the captured data is useful, sampling decisions are coherent per request – a trace is either sampled in its entirety, capturing the full end-to-end execution, or not at all. Sampling is an effective way of reducing computational overheads; these overheads are only paid if a trace is sampled, so they can be easily decreased by reducing the sampling probability. In practice, sampling rates can be as low as 0.1% [42].

### 3.1 Head-Based Sampling

Head-based sampling, also known as upfront sampling, makes the sampling decisions immediately when a request enters the system. This approach avoids the runtime costs to generate trace data. This sampling is employed by the majority of existing industrial tracing systems, such as Google’s Dapper [42], Uber’s Jaeger [16] and Facebook’s Canopy [18].

The sampling decision must be made at the root of the trace and have relatively little information about the request to base the decision. However, there are different approaches to be used in this type of sampling, such as probabilistic sampling, in which the sampling decision is made based on a certain probability, or rate limiting sampling, in which the rate limiter ensures that only a fixed number of traces are sampled per given interval. Other approaches are adaptive sampling [16], that adapts the sampling probability based on the rate of data coming to the tracing backend, and context-sensitive sampling, which enables the use of a profile of requests to be sampled with higher rates.
storage costs will be spent with "uninteresting" traces. Using behavior. In a uniform random sampling, the chances of every one thousand requests, only one shows an anomalous request latency in a service is very high, which means that, for can be made.

Ideally, a tail-based sampler should not require developers to explicitly specify features on which to bias the sampling decision. We argue that it would be impossible to predict all possible useful features a priori, and since traces are richly structured, the space of possible features is extremely large. More pragmatically, the burden on developers to identify these features and write feature extraction functions is undesirable. Most importantly, it is undesirable for features to

However, since head-based sampling occurs prior to request execution, the sampling decision is uniformly at random, and the resulting data is simply a random subset of requests. Inevitably, the set of sampled traces contains mostly common-case execution paths, with a lot of overlap and redundant information. Conversely, uncommon edge-case executions may be missing entirely, despite containing useful information about anomalous behaviors and infrequently exercised code paths. This impacts the utility of the collected traces in subsequent analyses and investigations.

3.2 Tail-Based Sampling

Recent work in both industry [27] and academia [22] has questioned whether head-based sampling is a necessary optimization. Although the runtime overheads of distributed tracing are non-zero, they are not dominant compared to the subsequent trace processing, querying, and storage costs. The proposed alternative is tail-based sampling, where traces are captured for all requests, and the choice of whether to keep a trace is made after the request has completed. Tail-based sampling schemes pay the runtime costs of generating trace data, but in return the tracing data itself can be factored into the sampling decision. Tail-based sampling introduces the possibility of biased sampling schemes, where we only persist the most useful traces, and discard traces that carry no useful information (e.g., redundant traces of common-case execution paths). Figure 2 illustrates the pipeline for recording, aggregating, processing, and storing distributed traces, showing where in the pipeline different sampling decisions can be made.

Assuming a scenario where the 99.9th percentile of the request latency in a service is very high, which means that, for every one thousand requests, only one shows an anomalous behavior. In a uniform random sampling, the chances of sampling anomalous requests are very low and the majority of storage costs will be spent with "uninteresting" traces. Using tail-based sampling, we can sample qualitatively better traces for the same storage budget; or alternatively, to achieve the same utility of our sampled traces we need to sample fewer traces overall.

One important aspect of tail-based sampling is to define what constitutes an 'interesting' trace. The approach taken in prior work to define 'interesting' traces is manual feature engineering. For each trace, a handful of high-level features are calculated from traces. For example, request latency can be derived from traces as the timestamp delta between the request begin and the response being sent [18]. The sampler then uses these features to make its sampling decision. For example, if our use case is tail latency investigations, we might weigh the sampling probability proportional to the measured request latency. Trace sampling at Lightstep [27] takes this approach: traces are sampled in order to best explain the latency variation of services.

Manual feature engineering is suitable when we already have a good sense of the features that will correlate with traces being interesting. Latency is the clearest example of such a feature, as many anomalous executions or edge-case behaviors result in increased request latency. However, feature engineering is naturally limited to only those features that developers can predict will be interesting a priori, and those features that can be easily written as a feature extraction functions. For instance, at Facebook manually-derived features are a useful starting point for many investigations, but the authors acknowledge that manual features leave a large quantity of data unused from each trace [18].

Ideally, a tail-based sampler should not require developers to explicitly specify features on which to bias the sampling decision. We argue that it would be impossible to predict all possible useful features a priori, and since traces are richly structured, the space of possible features is extremely large. More pragmatically, the burden on developers to identify these features and write feature extraction functions is undesirable. Most importantly, it is undesirable for features to
be identified and incorporated only in response to a problem occurring, which is commonly how new features are added to trace processing pipelines [18].

We claim that the sampling decision should be made directly on the underlying trace data, and features should be learned, rather than engineered. However, each trace is a richly annotated, arbitrarily sized, directed acyclic graph, and there are a number of additional subtleties that make it difficult to work with this data. In this work, we present two different tail-based sampling approaches. SolutionA measures the similarity between traces, use hierarchical clustering to organize the traces and sample in the inverse proportion to the size of the cluster that a trace belongs to, to give higher weight to anomalous traces. SolutionB is a framework that models the system and makes sampling decisions based on these models.

3.3 Operational Requirements

Trace sampling is inherently online: samplers operate in a production setting, and they must make sampling decisions over a continuous stream of incoming traces. Sampling decisions must be fast, and the sampler should scale to large production workloads. Satisfying these constraints while avoiding feature engineering is difficult and has not been solved by prior work.

A common approach to trace sampling, and to related problems like trace clustering, is to adapt general-purpose machine learning and graph mining algorithms. However, while many techniques have a good conceptual fit, most algorithms are designed for an offline setting and have computational costs far in excess of acceptable production limits. For example, some initial work on trace sampling used graph kernels as a pairwise trace distance function, and applied off-the-shelf clustering algorithms [30]. While this approach yielded interesting offline clusters, the authors acknowledged that significant improvements were needed to be reasonable in an online setting.

Techniques for trace clustering have similar challenges. For example, to satisfy operational requirements, Magpie uses only an approximate string-edit-distance comparison metric, and a simple “nearest centroid” clustering algorithm [3]. This comes at a loss of utility, with the authors acknowledging that the approach is likely to deteriorate in larger systems with more complex request structures.

In all prior approaches, scalability remains a challenge. In all cases, the internal state grows in proportion to the number of traces seen or previously sampled. This is problematic if the computational cost of a sampling decision is proportional to the internal state, especially if that state is unbounded. For example, most prior approaches to both sampling and clustering compare incoming traces with the set of previously sampled traces [3, 30]. In terms of state, this requires persisting in memory the set of previously sampled traces; and in terms of computational cost, each successive sampling decision is proportional to the number of previously sampled traces.

To be feasible for a production setting, trace sampling must be fast and scalable. Ideally, the computational cost of each sampling decision should be constant, and the sampler should have a known and fixed memory overhead over time. These costs should be independent of workload volume, the number of traces previously seen, and the number of traces previously sampled.

4 Tail-based Sampling Goals

The goal of tail-based tracing samplers is to make biased sampling decisions and to be suitable for production tracing systems, satisfying the operational requirements of production distributed systems, described in subsection 3.3. For that, the sampler has to be able to sample interesting traces, in a timely manner and to be scalable.

Tail-based samplers require biased sampling decisions. The goal is to sample a representative set of traces. In this work, we investigate two approaches to formulating the trace sampling problem. First as representative. Second as a probabilistic model.

The representative sampling problem can be defined as:

Given a set of execution traces and a sampling budget s, select a subset of the traces that is maximally diverse (or minimally redundant).

Considering an ideal case, in which N total traces can be classified into a well-defined set of K clusters, with |Ci|, 1 ≤ i ≤ K, as the number of traces in each cluster and the sampling budget is sN = S traces, in which s is the sampling rate. We also assume that s is enough to capture at least one of each type of trace, i.e., sN ≥ K. Intuitively, we would like to have an equal representation of each cluster in our sample, except when there are not enough traces of a given cluster. A max-min fairness allocation of sampled traces per cluster provides precisely this [6]. With this allocation, for each cluster, one will have Si = min(|Ci|, σ), where σ ≥ S/K is such that ∑Si = S. If |Ci| ⩾ S/K ∀i, then σ = S/K, i.e., when there are enough traces in each cluster, all clusters get exactly the same “fair” share of S. It turns out that the max-min fair allocation is also the allocation with maximum entropy, as shown by Coluccia et al. [11], and this is in line with the problem definition.

The distributed tracing context is not so clear, because in a realistic setting it cannot be assumed that this clustering exists, since systems have multiple components that are loosely coupled, and which change independently. Even if it does, we do not know what the clustering is, or how many clusters there are, nor the probability distribution over the clusters. Therefore, we have the probabilistic sampling problem, which states that the sampling probability of a distributed trace is inversely proportional to how close the trace is to the standard
behavior of the system. That is, traces close to common-case executions should have low sampling probability, while traces that do not approximate to the system behavior should have a high sampling probability.

In this definition, we seek approximate solutions. Specifically, we seek a sampling function \( S_a \) that maintains an average sampling rate of \( \alpha \), and for each trace \( T \) calculates a biased sampling probability \( \rho \). In addition, \( S_a \) maintains state \( \Theta \) that is updated with each new trace.

\[ S_a(T, \Theta) \rightarrow \rho, \Theta' \]

These two different definitions of the problem lead to different solutions. The representative problem aims at an equal distribution for the different types of traces in the resulting sample set. Thus, it is necessary to organize the traces so that each group has equal share of the sampling budget. This leads to the need for pairwise comparisons between traces, which is what SolutionA does. It compares and organizes the traces in a binary tree to define the same sampling probability for each group of traces. The probabilistic problem determines that the sampling probability should be based on the approximation of the trace with respect to the system’s behavior. This statement leads to the need for an aggregated modeling of the system, just like SolutionB is designed for. SolutionB models the system’s behavior and infers the sampling probability based on approximations between the trace and the observed behavior of the system.

5 SolutionA: a pairwise comparison approach

This section describes the first proposed tail-based sampler, SolutionA, which focuses on sampling a representative set of traces. This sampler organizes distributed traces in a hierarchical clustering binary tree and, based on this tree, is able to infer the sampling probability for each execution. SolutionA relies on pairwise trace comparison to arrange the traces in the tree in such a way that similar traces are located on close branches, while the most different ones are in distant branches. This structure allows common-case executions to be close together and, consequently, having a small sampling probability, while edge-case executions are on separate branches and have a higher sampling probability. SolutionA can be used as an online sampler, that operates over a continuous stream of traces or in an offline manner, after all executions being received.

5.1 Hierarchical Clustering

The first step of SolutionA is to organize the traces in a hierarchical clustering binary tree, also called a dendrogram. While there are several approaches to hierarchical clustering, we adopt the PERCH algorithm (Purity Enhancing Rotations for Cluster Hierarchies) [20]. Perch is a non-greedy, incremental algorithm for hierarchical clustering that scales to both massive points and \( K \) clusters. This algorithm constructs a tree over data points that are incrementally inserted to the leaves, using rotation operations that maintain its dendrogram purity. Assuming the existence of an underlying clustering (separable data), purity roughly measures how close a dendrogram tree is to this ground truth clustering, by comparing, for each pair of nodes in the underlying cluster, how close they are in the tree. We refer to the original paper [20] for further details. When purity decreases, or when the tree becomes unbalanced, the algorithm will perform recursive rotations to restore these properties. The tree structure enables an efficient search that scales over large datasets and is often logarithmic.

In PERCH, each new trace is inserted next to its nearest neighbor, found using A* search. In order to reduce the computational cost of the algorithm, the authors use approximations based on bounding boxes. Each internal node of the tree maintains a bounding box that has all its leaves. The heuristic used in the A* search algorithm is the minimum distance to each node based on the bounding box. Thus, instead of comparing to all leaves of an internal node, they simply compare the new point to the bounding box, reducing the computational cost. In order to reduce even more the cost of the nearest neighbor search, it is possible to use the search algorithm with predefined beam width, that is, the algorithm only allows a limited number of nodes to be in the frontier at any time. That solution will not guarantee that the best solution will be found, but it reduces the cost of this operation.

Because of its node rotations, PERCH trees tend to be balanced, and this is helpful in our sampling strategy.

Algorithm 1 Dendrogram Sampling algorithm

Input: tree and \( n \), number of traces to be sampled. Output: \( s_{trace} \), set of sampled traces.

1: \( s_{trace} \leftarrow \text{set}() \)
2: \( c_{trace} \leftarrow 0 \)
3: while \( c_{trace} < n \) do
4: \( \text{node} \leftarrow \text{tree}\text{.root()} \)
5: while \( \text{node} \) is not leaf do
6: \( \text{node} \leftarrow \text{choose(node.children)} \) \( \triangleright \) Select a child randomly.
7: if \( \text{node} \) not in \( s_{trace} \) then
8: \( s_{trace}.\text{add(node)} \)
9: \( c_{trace} \leftarrow c_{trace} + 1 \)
10: return \( s_{trace} \)

5.2 Offline Sampling

Given the binary tree generated by the hierarchical clustering, our offline sampling strategy is straightforward, given in Algorithm 1: to produce a sample, start at the root of the tree and for each branch choose one side with 50% chance.
would also have an 80% chance to be sampling using the random strategy. With fewer nodes and, thus, shallower. These will tend to be balanced, making it deeper than the branches containing infrequent ones. Rare traces, different from other traces, will tend to be in separate branches, with fewer nodes and, thus, shallower. These will tend to be sampled with higher probability.

Repeat this without replacement until the desired number of samples is reached. All traces are in the leaves of the tree. For each individual trace, at depth $d$, the probability of being sampled is $2^{-d}$.

Under this strategy, in an ideal case with $K$ clusters, where $K$ is an integer and $K = 2^{k}$, and in which the first $k$ levels of the tree are balanced, each cluster will have the same probability $2^{-k}$ of having elements sampled, independent of the number of nodes in each cluster. If a cluster then has $C_i = 2^i$ traces, and again the subtree is balanced, each node will be at depth $k + \gamma$, and will have a probability $2^{-k} \times 2^{-\gamma}$ of being sampled, which equals $2^{-K}$ if multiplied by $C_i = 2^i$.

In practice, neither the number of clusters nor the number of nodes per cluster will be powers of 2, and the tree will not be perfectly balanced. This is a tradeoff in a binary tree that will make the sampling deviate from the ideal case. However, we found that this strategy, due to the properties of the clustering algorithm, seems to work well in practice. PERCH has two desirable properties: dendrogram purity, which strives to place similar nodes close in the tree, and the balancing, which tends to make subtrees balanced, provided this does not violate purity as well. Similar and frequent executions will likely be placed in the same branch, making it deeper than the branches containing infrequent ones. Rare traces, different from other traces, will tend to be in separate branches, with fewer nodes and, thus, shallower. These will tend to be sampled with higher probability.

Figure 3 shows the sampling probability for frequent and infrequent execution traces. The frequent ones, represented by the green points, are 80% of all traces and, in this case, would also have an 80% chance to be sampling using the random strategy. However, through our method, the probability decreases to 50%, since each of the 8 traces has $2^{-4}$ sampling probability ($2^{-4} \times 8 = 50\%$). The infrequent executions also have 50% chance of being sampled ($2^{-2} \times 2 = 50\%$), even representing only 20% of the total number of points.

5.3 Online Sampling: Sliding Tree

While we use the offline algorithm to evaluate the quality of the clustering, it is not very practical to have all traces and then sample them. In this section, we describe an online sampling approach, which we call the Sliding Tree algorithm. The name comes from an analogy with a sliding window, as we use the tree to store a fixed number of traces.

**Algorithm 2 Sliding Tree Algorithm**

**Input:** tree, $n$ trace

**Output:** tree

1. If $|tree| == n$ then
2. delete-unlikely-node(tree)
3. Insert new trace (tree, trace) $\triangleright$ Algorithm from [20].

With each new trace, the algorithm (listed in Algorithm 2) first checks to see if the tree has reached its maximum size. If so, it deletes an element that is among the ones with the lowest probability of being sampled. To do this, the algorithm starts at the root and follows the branch with the largest number of leaves, until it reaches a leaf. Ties are broken randomly. After this element is removed, the algorithm inserts the new element in its regular place, i.e., adjacent to its closest node already in the tree.

The use of this strategy makes it difficult to remove rare traces from the tree since they are likely to have high sampling probability. In contrast, frequent executions do not flood the tree, since when a new trace of this type is to be inserted, it simply replaces another one of this same type. This way, we can balance the tree with the different trace types, as will be shown in Section 7. This algorithm also decouples the insertion cost from the volume of traces, as the cost of insertion depends only on the (fixed) size of the tree, and not on the total number of traces.
5.4 SolutionA Workflow

Figure 4 outlines SolutionA’s workflow. SolutionA integrates a typical tracing backend. After each request ends, the tracing backend aggregate all events of the request, causally relating them, generating the final trace. When this trace is ready to be forwarded, it is sent to SolutionA.

Each graph consists of a set of vertices \( V = v_0, v_1, \ldots, v_n \) and a labeling function \( \ell: V \rightarrow \Sigma \) that assigns labels from an alphabet \( \Sigma \) to each vertex. Since those graphs represent request execution traces, we consider that for any nodes \( v_0 \in G_0 \) and \( v_1 \in G_1 \), \( \ell(v_0) = \ell(v_1) \), \( v_0 \) and \( v_1 \) are different occurrences of the same event. Commonly this means that the events were generated by the same line of code, though specifying event labels is a design decision at the time of instrumentation.

Therefore, to build the node-based array, we count the number of times each event occurred in the trace and each entry of the array will represent one of those events. Once a trace is converted to an array, we can compare it to other traces using Euclidean distance as a metric.

Algorithm 3 Delete Unlikely Node

```plaintext
1: procedure DELETE-UNLIKELY-NODE(tree)
2:     node ← tree.root()
3:     while node is not leaf do
4:         node ← max(child.leaves for child in node.children)
5:     delete node
6:     update leaves property for all ancestors
```

Now, the next step is to insert it into the Sliding Tree. If the Sliding Tree has reached its maximum size, we first need to delete the node with the lowest sampling probability (as stated in Algorithm 3), and then insert the new trace. Figure 5 shows an actual example of a binary tree with maximum size = 100, in which 90% of the nodes are common-case executions and the other 10% are traces in which the execution failed. This is a clear example that the sampling probability for each type of trace reaches the ideal case.

The final step is to persist the traces in the tracing storage. There can be different policies to persist them. For example, if we desire a certain sampling rate \( p \), it is possible to persist all traces in the Sliding Tree after a predefined number of traces is received. Imagine we want \( p = 0.01 \). Then, we can keep a Sliding Tree with maximum size = 1,000 and after every 100,000 processed traces, the traces are stored. Another possibility is to persist traces after a certain period of time, like every day or every hour.

5.5 Implementation

We extended SolutionA to be used with X-Trace and Jaeger traces. However, SolutionA is extensible to different types of traces, provided we can compute a meaningful distance between two traces. We evaluated SolutionA and showed its results in section 7. However, first we present the design of the other sampling approach, SolutionB.

6 SolutionB: an approximation approach

This section describes SolutionB, a tail-based sampler that addresses the challenges described in section 4. SolutionB is an online sampler that operates over a continuous stream of traces coming from production distributed systems. The starting point for SolutionB’s design is the observation that collectively, the stream of traces in aggregate reflect the overall system and workload behavior, capturing the commonly-traversed paths, corner case executions, and distributions over paths and timings. Based on this observation, our intuition for SolutionB is to use the incoming stream of traces to construct...
and maintain an *unbiased* low-dimensional approximation of the distributed system’s common-case behavior. Then to sample traces, it is a matter of comparing each trace to the low-dimensional approximation, and biasing sampling decisions towards traces that are not well approximated.

### 6.1 Dimensionality Reduction

SolutionB constructs and maintains a low-dimensional approximation of the distributed system’s common-case behavior. The purpose of this low-dimensional approximation is simply to capture the probability that events occur in a particular order. However, our approximation must be robust to noise, and should condense highly correlated events and trace substructures, so explicitly enumerating paths and probabilities is an unsuitable approach.

Instead, to achieve our goal, we take inspiration from the area of neural language modeling [5], which we find shares many of the challenges described in this work. Of particular interest are techniques for constructing low-dimensional word and document embeddings from large corpuses of example text [23, 34]. These techniques share a similar goal of learning a probabilistic model for words appearing together in sentences, and also deal with challenges of noise and high dimensionality. The model used by SolutionB is an adaptation of the paragraph vector embedding [23].

Concretely, SolutionB uses a neural network that models the conditional probability of a label occurring given its immediate causal predecessors and successors. To incorporate structure, we decompose a trace into sequences of events following happened-before relationships. For each trace, we first extract all N-length paths, where a path is a sequence of events that follow happened-before relationships. We then map each event in the path to its corresponding label. The model then operates on these N-length label paths. For each path, we predict the middle label \( l_2 \) given the surrounding labels \( l_0 \ldots l_N \) (excluding \( l_2 \)) as input.

We implement the model using the same architecture as the distributed memory paragraph vector model, using concatenation [23]. The architecture is a 2-layer neural network illustrated in Figure 6. Inputs are one-hot encodings of \( l_0 \ldots l_N \) (excluding \( l_2 \)), and the output is a one-hot encoding of \( l_2 \). The first layer of the network transforms each input label to a \( P \)-dimensional vector. The second layer concatenates them then predicts the output label using a softmax classifier.

Key to the design of this architecture is the intermediate layer, where the choice of \( P \) is typically small. This forces the model to find approximate representations for its inputs, because it cannot represent the full joint probability distribution of inputs and outputs. Conceptually, this layer is similar to the hidden layer used by autoencoders. When used by language modeling, the weights of the intermediate layer can be used as low-dimensional representations of the corresponding words, and have a range of interesting properties [34]. In our case, \( P \) forces the model to approximate – specifically, the model maximizes the average log probability of a label given its surrounding labels, across all walks in all traces. By this definition, the model makes better predictions for paths that are seen more frequently in the input dataset (i.e., in the traces we train on).

### 6.2 Sampling Using Prediction Error

To make a sampling decision for a trace, SolutionB extracts all N-length paths from the trace, batches them together, and performs a single forward-pass of the model. Since traces are directed and acyclic, small values of \( N \) do not result in an intractable number of paths; Table 1 outlines statistics for datasets used in our experiments.

The output of the forward pass is both a prediction of events, and the prediction’s *loss*, i.e. the error between predicted labels and actual labels. A high loss means that the model was unable to predict the execution paths contained in the trace. Since the model is biased towards the common-case execution paths, a high loss implies that the trace contains edge-case execution paths that are not commonly exercised. This implies that the trace is more interesting, since it captures execution paths that are outside of the norm.

SolutionB uses this signal to determine whether to sample the trace – traces with higher loss are more interesting; we could not predict their execution paths well, so we want to sample them with higher probability.

### 6.3 Calculating Sampling Probability

After calculating the prediction loss for a trace, the next step is to map this loss to a sampling probability. To begin, SolutionB is parameterized with a target sampling rate \( \alpha \), e.g. to sample 1 out of every 100 traces, we set \( \alpha = 0.01 \). Translating prediction loss to a sampling probability is not straightforward, because it depends on the prediction loss of previously seen traces; i.e. if other traces seen recently had higher prediction loss, then this trace is qualitatively less interesting.
To address this, SolutionB tracks the loss of the $k$ most recently seen traces, and weights the sampling probability of the next trace based on the distribution of these prior $k$ losses. For the $k$ most recently seen traces and for the new trace, we calculate weights $w_i$ where:

$$w_i = \text{loss}_i - \min_{1 \leq j \leq k+1} \text{loss}_j$$

If all weights are equal (i.e. all losses are the same) then we sample the next trace uniformly at random with probability $\alpha$. Otherwise, we sample the next trace with probability:

$$\rho = \frac{w_{k+1}}{\sum_{j=1}^{k+1} w_j} \times (k+1) \times \alpha$$

The effect of this sampling scheme is that traces with the lowest loss have sampling probability of zero; traces with the highest loss have highest sampling probability; and for traces in between, the sampling probability increases linearly with the error. If zero sampling probability is undesirable, a constant term can be added to $\rho$, representing a minimum sampling probability.

### 6.4 SolutionB Workflow

Figure 7 outlines SolutionB’s sampling workflow. SolutionB integrates with a typical tracing backend. Typical tracing backends aggregate events in memory for a short period of time, before forwarding them for trace processing and storage [18]. When a trace is ready to be forwarded, it is sent to SolutionB to make a sampling decision. SolutionB begins by extracting all $N$-length paths from the trace and does a forward-pass of the model, as described in subsection 6.2. From this forward pass, we calculate a loss value for the trace.

Next, SolutionB performs a gradient descent backpropagation pass on the model to update the model weights to incorporate the trace that was just seen. SolutionB then calculates a sampling probability from the loss, as described in subsection 6.3. After calculating the sampling probability, SolutionB inserts the trace’s loss to the window of $k$ most recent losses and drops the oldest value.

Finally, SolutionB makes the sampling decision using the calculated sampling probability. If sampled, then the trace continues in the trace processing pipeline. If not sampled, the trace is now dropped.

### 6.5 Scalability

Part of SolutionB’s workflow is to perform a backpropagation pass on the model to update the model weights for every trace that is seen. This enables the model to capture common-case execution behavior, even if that behavior is then less likely to be sampled. This approach would fail if we only built the model using traces that are sampled, because it’s a lack of bias in the model that enables us to determine common cases in the incoming workload. As a consequence, the model weights change over time if the underlying distribution of traces changes (e.g., if the proportion of execution paths change; if new execution types are added; if the vocabulary of labels changes due to code updates, etc.).

SolutionB only keeps internal state for the model’s weights, and to track the $k$ most recently seen losses. The model comprises exactly $|\Sigma| \times P$ weights for the first layer, and $(N - 1) \times P \times |\Sigma|$ weights for the second layer, where $P$ is the size of the low-dimensional intermediate representation, $N$ is the path size we wish to use, and $\Sigma$ is the label vocabulary. The model does not maintain paragraph vectors, since each trace is seen once and only once. Thus the total size of $\Theta$ (cf. section 4) is constant with respect to the workload volume and number of sampled traces. In terms of computation, performing forward and backwards passes of the models constitute a fixed number of floating point multiplications, which has constant-time complexity.

### 6.6 Implementation

We have implemented SolutionB in Python and integrated it with the X-Trace tracing framework. We used TensorFlow to implement SolutionB’s internal model. To use SolutionB, an instance is first initialized using input parameters $N$, $P$, $k$, and $\alpha$. Subsequently, sampling decisions are made through a call to `sample(T)`, which takes as input a trace $T$ and gives as output a boolean sampling decision. The input trace $T$ is simply a directed, acyclic graph where each node has a label $l$. For convenience, we implemented functions to convert X-Trace [12], Zipkin [46], and Jaeger [16] traces into this format.

Each call to `sample` performs the steps described in subsection 6.4, including backpropagation to update the internal model. SolutionB does not require a priori knowledge of the
label vocabulary \( \Sigma \); it extends the model weights as needed when it encounters previously unseen labels. By default, SolutionB uses paths of length 5 \((N=5)\), embedding dimension 10 \((P=10)\), a window of 50 \((k=50)\), and a learning rate of 0.01 for the model backpropagation pass.

7 Evaluation

In this section, we compare and contrast the two proposed samplers, SolutionA and SolutionB. Since the two approaches are structurally different, here we aim to understand how they behave under different circumstances. We evaluate its ability to bias sampling decisions towards edge-case and anomalous traces. Our evaluation aims to demonstrate characteristics like how the samplers distinguish anomalies from common-case executions, how it biases sampling decisions towards underrepresented execution types in mixed workloads and how it reacts to changing workload distributions, both when common executions become less common, and vice versa. We use traces from a variety of distributed systems, and consider a mix of synthetic and real-world workloads.

7.1 Datasets

In order to evaluate how the tail-based sampling approaches behave and to show different situations where each sampler is better suited, we use real datasets, including open-source system HDFS [41], YARN [47], Spark [49], the DeathStar social network benchmark [14]; and production traces from a large internet company.

**Hadoop Distributed File System** We have instrumented the Hadoop Distributed File System (HDFS) [41] with the X-Trace framework and deployed it on a 9-node cluster. To generate detailed traces, we override HDFS’s logging calls to emit X-Trace events. As a result, the generated traces can be large, and contain lots of noise and minor permutations of events. Our dataset comprises traces of the following requests: (1) 1MB file writes to HDFS; (2) reads of 1kB, 10kB, 100kB, 1MB, 10MB, and 100MB files from HDFS; (3) random reads of between 1kB and 100kB. The dataset comprises 70,966 traces in total.

**DeathStar Social Network Benchmark** We instrumented the DeathStar social network microservices benchmark [14] with X-Trace. We duplicated all logging calls to emit X-Trace events, and logged events at the start and end of all services. We deployed the benchmark on one machine and captured traces of 7 different API types (Register user, Follow user, Unfollow user, Compose post, Write timeline, Read timeline, Read user timeline). Internally, the benchmark comprises 36 microservices; each high-level API call invokes an overlapping subset of the services. In addition to datasets of regular workloads, we also captured traces of two classes of anomaly: one where we manually triggered exceptions in the internal microservices; and one arising accidentally from a configuration error in our deployment, causing docker containers to intermittently restart and services to be temporarily unavailable. The dataset comprises 15,148 traces in total.

**Production Traces** This dataset comprises 676 traces from a large internet company. The traces capture spans from a microservice architecture, and can be grouped into five different API types.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Traces</th>
<th>Avg. Nodes</th>
<th>Avg. Labels</th>
<th>Avg. Walks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>70,966</td>
<td>1,428</td>
<td>38</td>
<td>2.547</td>
</tr>
<tr>
<td>DeathStar</td>
<td>15,148</td>
<td>127</td>
<td>82</td>
<td>155</td>
</tr>
<tr>
<td>Production</td>
<td>676</td>
<td>71</td>
<td>56</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 1: Statistics for datasets used in experiments. Nodes, labels, and walks are averages for the traces in each dataset.

7.2 Anomalies and Outliers

In the first set of experiments, we aim to show the ability of the samplers to identify and sample anomalous and outlier executions.

7.2.1 HDFS Reads

In our first experiment we replay traces of HDFS read API calls, which randomly read between 1kB and 100kB of data from HDFS. We run a sequence of 1000 traces, and in five separate moments we insert a trace of an HDFS write API call, representing only 0.5% of the workload. Qualitatively, there is little overlap between read and write traces; some RPC events are shared, but most of the data pipelining events are mutually exclusive.

Figure 8: Sampled HDFS write traces for each approach varying the sampling rate.
We ran each experiment for each strategy 100 times, varying the sampling rate \( \alpha \) from 1% to 20%. Figure 8 shows how many HDFS write traces were sampled. As we can see, SolutionA is able to sample all 5 traces, independent of the sampling rate. In that case, we have two well-defined ‘classes’ of traces, thus the hierarchical clustering is able to perfectly separate each class. On the other hand, SolutionB can only sample all 5 anomalous traces for the 5% sampling rate and up, selecting around 1.6 traces when the sampling rate was 1%, an increase of 8 times from the actual distribution of traces.

SolutionB determines the sampling probability individually for each trace. Therefore, its main goal is to have a higher sampling probability for edge-case executions, as indicated by the probabilistic sampling problem. The more different a new trace is from what has been seen before, the higher its sampling probability will be. Figure 9, for example, plots SolutionB’s loss and sampling probability for an execution, where the target sampling rate was 1%. The figure clearly shows five spikes for each instance of a write. The average sampling probability for reads is 0.0084, with minor fluctuations from trace to trace owing to their internal variability. For the write traces, the sampling probability is significantly higher, averaging 0.3325.

Figure 9: SolutionB’s loss and sampling probability for a workload of 1000 HDFS read traces. We intermittently introduce 5 write traces, corresponding to the five spikes with high sampling probability.

7.2.2 Social Network Compose Post

We perform a similar experiment using ComposePost traces from the DeathStar social network benchmark. We also run a sequence of 1000 traces, with 5 anomalous executions. The anomalous traces are also calls to the ComposePost API, but internally we randomly triggered an exception in one of the internal services. Those anomalies represent only 0.5% of the workload, but unlike the previous experiment, there is qualitatively more overlap between normal and anomalous executions, since they are similar up to the point of the exception.

![Figure 10: Sampled Compose traces for each approach varying the sampling rate.](image)

As can be seen in Figure 10, when we ran SolutionA to sample anomalous ComposePost executions, considering a 1% sampling rate, we select 3.5 anomalous traces (which represents 35% of total samples). The sampling distribution is not perfect because the regular ComposePost executions are diverse among themselves, which end up resulting in more ‘classes’ of traces. However, when we increase the sampling rate to 2%, SolutionA is able to properly sample all 5 failure traces.

![Figure 11: SolutionB’s loss and sampling probability for ComposePost API calls. Five exception traces are introduced, corresponding to five spikes with high sampling probability.](image)

Unlike the HDFS traces, the regular ComposePost executions have more internal diversity and the anomalous executions are closer to regular traces. Thus, SolutionB can only sample all 5 anomalous traces above a 15% sampling rate. However, as mentioned before, and according to the probabilistic sampling problem, the main purpose of SolutionB is to increase the sampling probability of edge-case executions. As can be seen in Figure 11, which plots SolutionB’s loss and sampling probability, with a target sampling rate...
of $\alpha = 0.01$, all five anomalous traces have higher sampling probabilities between 0.12 and 0.28, while the regular traces fluctuate around 0.01. Therefore, despite the diversity between traces and the similarity between failure traces and the regular ones, SolutionB clearly identifies the anomalies and samples them with higher probability.

7.2.3 Social Network ReadUserTimeline

We perform a final experiment using ReadUserTimeline traces from the DeathStar social network benchmark. In that experiment, we compare normal traces to anomalous traces that we inadvertently encountered due to a Docker misconfiguration that intermittently restarted some of the microservices. We replay traces with a 90%-10% distribution of regular to failure traces, corresponding to the error ratio we experienced. We replay 1000 traces and record the loss and sampling probability.

Figure 12 shows the percentage of failure traces sampled, varying the sampling rate from 1% to 20%. The best result occurs when we use SolutionA with 1% rate ($\approx 25\%$ of anomalous traces sampled). However, when we increase the sampling rate, the percentage decrease to an average of 15%. Overall, SolutionB presents the best result, with an average of 17%, increasing the amount of sampled traces by 1.7, when compared to their actual distribution (10%).

7.3 Representative Sampling

Our next set of experiments evaluates the sampling strategies ability to generally bias towards underrepresented execution paths and request types for workloads in aggregate. We performed three different experiments, with different datasets and traces distribution.

7.3.1 Production Traces

We applied our sampling methods to a workload of 676 traces captured from a large internet company’s production system. Those traces comprise 5 different high-level API calls; internally, the requests share commonality in some of the services they call. To introduce more variability, we adjust the representation of each API type to 2%, 8%, 15%, 25%, and 50% respectively. Figure 13a compares the distribution of traces sampled by SolutionB and SolutionA and compares them to the original workload, as well as random sampling.

SolutionB increases the representation of low-frequency APIs, while decreasing the representation of common-case APIs. By comparison, SolutionA only slightly increases the underrepresented APIs. To quantify this, Table 2 lists the mean-squared error compared to ideal distribution between all APIs (20% of sampled traces for each API, based on the representative trace problem). SolutionB has significantly lower error. SolutionA have higher error, but still do better than uniform random sampling.

<table>
<thead>
<tr>
<th>Data Source</th>
<th>SolutionB</th>
<th>SolutionA</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production</td>
<td>35.95</td>
<td>193.12</td>
<td>283.60</td>
</tr>
<tr>
<td>DeathStar</td>
<td>46.31</td>
<td>246.26</td>
<td>377.72</td>
</tr>
<tr>
<td>HDFS</td>
<td>119.91</td>
<td>404.22</td>
<td>218.28</td>
</tr>
</tbody>
</table>

Table 2: Mean-squared error of sampled clusters (cf. subsection 7.3) for SolutionB, SolutionA, compared to random sampling.

7.3.2 DeathStar Benchmark

We perform the same experiment for the 7 DeathStar-Bench API types totaling 14,000 traces, with the following distribution: 0.5% Register; 2% Unfollow; 2% Follow; 2.75% WriteHomeTimeline; 2.75% ComposePost; 45% Read-HomeTimeline; 45% ReadUserTimeline.

Figure 13b shows the distribution of traces sampled by each approach. Each framework increases the representation of low-frequency APIs, however, SolutionB has the biggest increase. For example, Register API calls represent only 0.5% of the actual distribution, but 5.2% of the traces sampled by SolutionB have this type, representing an increase of more than 10 times. The other API calls, except for ComposePost are pretty close to the balanced distribution. Both Read-HomeTimeline and ReadUserTimeline traces represent 45% of all actual executions, but have lower representation in SolutionB’s sampled set, with 15.53% and 13.94% of the traces, respectively. The ComposePost traces have a higher percentage in the distribution of all approaches (although is smaller for SolutionB). That happens because the set of ComposePost is the most diverse one, being the most complex API internally, with the highest internal execution variety.
Table 2 lists the mean-squared error of the clusters compared to an equal 14.3% division between the APIs. As can be seen, SolutionB has the smallest error, more than 5 times smaller than the second best sampling result (SolutionA).

### 7.3.3 HDFS

We lastly perform a similar experiment for HDFS. We include the following traces: 1MB write API calls; and 1kB, 10kB, 100kB, 1MB, 10MB, and 100MB read calls. The read calls have significant internal similarity, as increasing the read size primarily duplicates trace events that correspond to data chunk transfers. We include 1% writes, and 45%, 25%, 15%, 7.5%, 4.5%, and 2% of the reads respectively.

Figure 13c shows the distribution of traces sampled by SolutionB. SolutionB significantly increases the proportion of write traces that are sampled, to 25%. Fewer read API calls are sampled, but they are sampled approximately in proportion to their representation in the original dataset. This occurs because the traces comprise the same types of events (i.e., the same set of labels); just some events occur more frequently in the large read calls. Consequently, the large reads are interesting, but not too interesting, as the additional data transfer events are treated as noise. Despite the uneven clustering, this is a desirable result. The figure also shows the distribution of traces sampled using SolutionA; since this approach simply counts the co-occurrence of labels in each trace, it inevitably weights towards traces with higher node counts, which are the larger traces.

### 7.4 Change over Time

We next evaluate how the samplers adapt to change over time. We focus on two scenarios: when workload distributions change; and bootstrapping the samplers from scratch.

**Changing distributions** This experiment aims to understand how the samplers behave when the distribution of traces change. To evaluate that, we used 1MB HDFS write traces and 1MB HDFS read traces. Iterations of the experiment alternate between 90% write/10% read, followed by 10% write 90% read. We expect the sampler’s sampling probability to adapt to those workload changes, increasing the probability for the 10% class, and decreasing the probability for the 90% class.

Figure 14 shows the results obtained by SolutionB’s execution. It plots the average sampling probability for 9 iterations (with 1,000 traces each), with a target sampling probability of 0.01. The figure illustrates how the sampling probability alternates between low and high for the classes as they alternate between over- and under-represented.

![Figure 14: SolutionB’s results when changing distribution. Each iteration alternates the workload proportion 90%-10% and vice versa. SolutionB adapts to these changes, increasing the sampling probability of the underrepresented class each iteration.](image)
each. We used a 100-traces sliding tree. Since the sliding tree behavior was maintained throughout all 9 iterations, we show in Figure 15 only the first 1,000 traces (1st iteration). As we can see in it, as the tree reaches its maximum size, the class with more traces begins to decrease while the other one grows. After a while, the tree converges to a point where both classes have the same number of traces to be sampled, which means that both types have the same share in the sampling set. Even when the distribution is shifted, the tree continues with the same behavior, keeping the same amount of traces for each class. As the two types of traces are very different, the sliding tree can be perfectly balanced, separating them in different branches, which makes this behavior easier.

**Figure 15: SolutionA’s results when changing the trace distribution. Behavior was the same for all 9 iterations of the experiment.**

**Bootstrapping** It is also important to understand how the tail-based samplers behave when bootstrapping. To evaluate that, we initialize each sampler and simulate a continuous stream of 1,000 production traces being processed by them. 95% of those traces come from one API and the other 5% from a different one.

SolutionB is designed to operate in real-time and does not require pre-training or manual configuration before operation. Initially, SolutionB makes essentially random sampling decisions, as its internal model is randomly initialized. As soon as it starts to receive samples, it begins to learn its model of system behavior. Figure 16a plots SolutionB’s loss and sampling probability, starting randomly initialized. SolutionB quickly begins to differentiate common cases from edge cases, and the loss for common cases decreases faster than the loss for the edge-case traces. We repeated this experiment 100 times, varying the choice of API type and randomizing the trace order, and saw the same behavior each time.

SolutionA does not require any pre-training. Once the stream of traces starts to arrive, all traces are inserted into the tree until it reaches its maximum size. After that, for each new trace, we first delete the trace least likely to be sampled, and then add the new one to the tree. Figure 16b shows how the sliding tree behaves as the traces arrive. Just like in the previous experiment, new traces are inserted into the tree until it reaches its maximum size. After that, for each new trace to be inserted, it first deletes the one with the least sampling probability (or with the most frequent type of trace). That happens until the tree converges to a point in which both classes of traces have the same representation in the sliding tree, with equal distribution in the resulting sampling set.

**Figure 16: Bootstrap evaluation for SolutionB and SolutionA.**

**Figure 16a** shows how SolutionB requires no bootstrapping or pre-training.

**Figure 16b** shows how SolutionA’s results when starting with empty sliding tree.

**7.5 Importance of Structure**

Another experiment we performed is to understand the impact of the structure of traces on their sampling. As SolutionA takes into account the events composing a trace, and not how they are structured, this experiment is focused only on the evaluation of SolutionB.

In this experiment, we compare SolutionB to a similar scheme that fails to take into account trace structure. The goal of this experiment is to evaluate SolutionB’s effectiveness when using a linearized version of traces that discards internal concurrency information [3, 39]. We compare to SolutionB-Linear, a modified version of SolutionB in which we collapse a trace’s events into a single timestamp-ordered sequence. We then apply SolutionB to this sequential data, where paths are simply a sliding window over this sequence.

We apply SolutionB and SolutionB-Linear to a workload
of 1,000 traces comprising 90% 1kB HDFS reads and 10% 100kB HDFS reads. Figure 17 compares the loss and sampling probability for SolutionB and SolutionB-Linear. We observe that the loss for SolutionB-Linear has significantly more fluctuations than SolutionB, owing the fact that concurrent paths are superimposed on one another. The average sampling probabilities for 1kB and 100kB traces respectively are 0.010 and 0.010 for SolutionB-Linear, compared to 0.007 and 0.034 for SolutionB.

7.6 Overheads

A major challenge for tail-based samplers is scalability. Given the need to be used in production systems, the sampling decision must be made at the lowest possible cost. In this section, we evaluate how both proposed tail-based samplers behave regarding their computational cost.

To evaluate their sampling overhead, Figure 18 plots the distribution of sampling latencies for traces from our experiments. SolutionB’s sampling latencies vary between 20 and 300 ms, with most traces having latency around 50ms or less. For example, 90% of the production traces have a latency of 60ms or less, while 80% of the DeathStar traces had execution time of less than 45 milliseconds. Since HDFS have some larger traces (when reading large files), around 10% of the traces had latency large than 200 ms but around 60% of them have a latency of less than 50 ms. The factors that contribute to increased sampling latency are the size of the trace, and the number of unique labels in the trace. However, sampling latency is independent of the workload volume, and the number of previously sampled traces. On the other hand, SolutionA’s sampling overhead grows proportional to the number of sampled traces. Since SolutionA latencies grow arbitrarily, it is not possible to show a meaningful CDF, as presented for SolutionB. Therefore, Figure 18b shows the average latency for all traces for each sliding tree size. As the tree size increases, so do the sampling latency. For HDFS traces, for example, when we consider a tree with maximum size of 10, the sampling latency is around 8 ms. However, when we increase the tree size to 10,000 traces, the latency reaches around 125 ms. Although SolutionA’s latency is directly related to the amount of sampled traces, the size of each trace does not impact the sampling overhead as much as it does for SolutionB.

We also evaluated the total time taken to execute 10,000 DeathStar follow traces. As can be seen in Figure 19a, until around 3,000 traces, the total execution time taken by SolutionA is smaller than SolutionB’s execution time. That
happens because the cost per trace for SolutionA is smaller when the sliding tree is also small, as showed in Figure 19b. However, as the sliding tree grows, the latency per trace also increases, while SolutionB’s latency remains constant. Consequently, the total execution time for SolutionA for all 10,000 traces is much larger than the one taken by SolutionB (approximately 3.5 times).

Through these experiments, we have shown that SolutionB is scalable and suitable for production distributed systems, since the sampling time is independent of the number of traces and the latency remains in the order of milliseconds. In contrast, SolutionA latency is directly dependent on the number of traces, which makes this approach better suited for situations where the sampling time is less important, or where the tree size can be acceptably bounded.

### 7.7 Clustering traces

Lastly, an interesting feature that both approaches have is the ability to cluster traces. This is an inherent characteristic to SolutionA, since its sampling probability is defined by applying a hierarchical clustering algorithm to the traces. SolutionB is also able to cluster traces. One interesting side-effect SolutionB has is the model used to represent the traces evaluated and can be used to cluster them.

In order to evaluate the clustering quality for each approach, we used HDFS and DeathStar traces. For HDFS traces, we have two different clusters, one containing 100 KB read call traces and the other one with 10 MB reads. For DeathStar traces, we used all 7 different types of traces from the SocialNetwork application: UserRead, ReadHome, WriteHome, Compose, Follow, Unfollow, Register.

![Figure 19: Latency behavior for each sampler over time.](image)

We applied offline SolutionA to the traces and outputted the dendrogram used to organize the traces in order to define their sampling probability. Due to visualization constraints, we used 500 HDFS traces (250 of each class) and 700 DeathStar traces (100 of each class). Figure 20 shows the dendrogram for HDFS traces. As can be seen, the two classes of traces are perfectly separated by the first branch of the tree. The same happens with DeathStar traces. Figure 21 presents its clustering. As can be seen, when we cut the dendrogram in a way that results in 7 different clusters, all traces are perfectly placed in the right group. Another advantage of SolutionA, when clustering the traces, is that, since it is a hierarchical approach, it is possible to identify similar sub-groups inside a cluster. When we manually analyze compose traces, for example, we can clearly see different characteristics between them. Thus, through the hierarchical clustering, we can get those groups.

![Figure 20: Resulting clustering of HDFS read traces (100KB and 10MB) obtained by applying SolutionA.](image)

As previously stated, SolutionB produces intermediate embeddings. Those embeddings are low-dimensional representation for each trace. We can apply a set of different machine learning algorithms on them, like clustering, for example. For this case, we applied Principal Component Analysis (PCA) to reduce the dimensionality of the embeddings and get a 2-dimensional representation [15]. Figure 22 plots this PCA

![Figure 21: Resulting clustering of DeathStar SocialNetwork traces obtained by applying SolutionA.](image)
projection of traces from HDFS and the DeathStar social network benchmark. As we can see in Figure 22a, there is a clear separation between the HDFS traces, despite both request types being read requests. Figure 22b also shows that DeathStar traces can be separated into different clusters. Based on this side-effect of SolutionB, another possible sampling algorithm might try to uniformly sample traces across different regions of the embedding space. However, in practice, we found that loss provided a much stronger signal for sampling, especially in cases of never-before-seen traces, for which loss will be very high, but the position in the embedding space may be arbitrary.

7.8 Artificial Datasets

To further understand how SolutionA and SolutionB behave, we created a set of experiments using synthetic trace workloads that we can more clearly characterize. Hence, it is possible, for example, to control how different distinct traces are and, with that, to understand how the sampling approaches behave given these differences. We can also generate traces with different structures and understand what impact it will have when trying to sample them. Finally, we can generate different types of traces and analyze the impact caused by the increase in the number of traces ‘types’ being observed.

7.8.1 Variation between traces

![Figure 23: Traces with 1% difference.](image)

In the first experiment, we aim to evaluate how the samplers behave regarding how different traces are. We used 1,000 traces from two different types. The first type is a linear trace with 100 nodes, composing the majority of traces in the experiment (99.5%). The second type of traces is also linear but they are \( x \)% different from the other trace, regarding their node labels. We vary \( x \) from 1 to 100%. Figure 23 shows an example with two linear traces, where the second one is 1% different from the first one.

We applied online SolutionA on this dataset, considering a sampling probability of 1%, i.e. to sample only 10 traces. For each \( x \)% difference, we ran SolutionA 10 times. All executions, for all differences, resulted in a perfect distribution of the sampling probability, with both normal and anomaly traces having 50% chance of being sampled (Figure 24b). That happens because all executions have a perfectly separated tree, where one branch has 5 traces of one type and the other branch has 5 traces of the other type, agreeing with the representative trace sampling behavior.

To evaluate SolutionB, we measure the average loss for all traces of each type. Figure 24a shows how the loss changes as we vary the difference between the traces. As we can see, when the traces are very similar (with only 1% difference), the losses are close to each other. However, the infrequent ones still have a higher loss, resulting in a higher sampling probability. The loss continues to increase as the difference percentage also increases, reaching its peak when the two types of traces are completely different. This shows that SolutionB is able to capture differences between traces and that the sampling probability is directly related to how different traces are, agreeing with the probabilistic sampling problem.

7.8.2 Impact of structure

Another experiment we made considers the structure of traces. We used traces with the same events but different structures. As can be seen in Figure 25, the first type of trace is a linear graph, where each event occurs right after the previous one. In the second one, we have two sets of events that occur in parallel with each other. We considered linear traces as the majority of traces and parallel traces as the infrequent ones.
Figure 24: Results for the experiments considering the variation of the degree of differentiation between traces.

Figure 25: Traces with same events but different structures. First one is linear while the other one has concurrent events.

Figure 26 shows the loss when we use SolutionB in that dataset. We can see that the loss (and sampling probability) of the parallel traces is much higher than that of the linear ones. Since SolutionB uses all N-length paths of a trace in the low dimensional model to infer the loss and, consequently, the sampling probability, the structure of a trace plays a big role in this sampling approach. In this case, the list of paths for each type of trace will be different, which makes SolutionB quite effective when applied to that dataset.

SolutionA does not do a good job when we consider those two types of traces. Since this method considers the number of occurrences of each event to compare the traces, and the two types have exactly the same events, the hierarchical clustering step is not able to organize traces separating the two types.

7.8.3 Difference in the number of classes

In the previous experiments, we considered only two different types of traces. Here, our goal is to understand the impact when we increment the number of different groups of traces. In this experiment, each type of trace is completely different from the other (regarding their events) and we vary the number of groups from 3 to 20. We ran each experiment 10 times, using 10,000 traces. Those traces are distributed in such a way that few groups have the majority of them, while the others have a small representation.

To evaluate each sampling approach, we measure the mean squared error between the ideal sampling distribution (based on the representative sampling problem [22]) and the resulting sampling set. Figure 27 shows the results for each approach and each number of groups.

SolutionB presents the best results, overall. Despite having a higher mean-squared error than SolutionA and to simply selecting the samples uniformly at random when there are only 3 types of traces, SolutionB reduces the error as the number of groups increases, reaching its best result when there are 20 groups, with a 1.84 error (20 and 10 times smaller than the random approach and SolutionA, respectively). An important feature of SolutionB is that it prioritizes less frequent traces. In these experiments, for example, in all cases, the group with the lowest representation was the one with the highest distribution of sampled traces. In order to better illustrate that, we depict the experiment with 14 different groups.

Table 3 shows the actual distribution between groups (that
when the number of classes is a power of two. Even when the
would be achieved if using a random strategy), SolutionA and
7.14% of the total traces. As can be seen, SolutionB has the
different approaches for solving this problem, SolutionA and
of a system. Based on those definitions, we proposed two
portional to how close the trace is from the normal behavior
sampling budget for each type of trace, while the
nition refers to the need of giving an equal share of the sam-
tracing sampling problem. The
approaches are only part of the story. Useful features often
are available, such as end-to-end request latency. By definition,
these features exist because an engineer previously found
them useful for some particular problem, and outlier values
offer a strong signal that a trace should be sampled. When
this is a case, simple and computationally efficient statistical

<table>
<thead>
<tr>
<th>Gr.1</th>
<th>Gr.2</th>
<th>Gr.3</th>
<th>Gr.4</th>
<th>Gr.5</th>
<th>Gr.6</th>
<th>Gr.7</th>
<th>Gr.8</th>
<th>Gr.9</th>
<th>Gr.10</th>
<th>Gr.11</th>
<th>Gr.12</th>
<th>Gr.13</th>
<th>Gr.14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>30</td>
<td>12.5</td>
<td>12.5</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3.5</td>
<td>3</td>
<td>2.5</td>
<td>2</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>SolutionA</td>
<td>30.2</td>
<td>14.6</td>
<td>10</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>3.4</td>
<td>3.5</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>SolutionB</td>
<td>2.0</td>
<td>7</td>
<td>45</td>
<td>6.45</td>
<td>9</td>
<td>56</td>
<td>7.67</td>
<td>8.89</td>
<td>8.56</td>
<td>8.78</td>
<td>9.89</td>
<td>9</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 3: Mean-squared error for the experiment considering 14 distinct groups.

Figure 27: Mean-squared error for different number of groups

would be achieved if using a random strategy), SolutionA and
SolutionB sampling distributions. In this case, considering the
representative sampling problem, each group should have
7.14% of the total traces. As can be seen, SolutionB has the
closest to ideal values for all groups.

Although SolutionA presents a larger error than SolutionB,
it achieves better results than the random strategy for all dif-
fferent numbers of groups. The better result is achieved when
the test set is composed of 8 groups. Since the sampling prob-
abilities are based on a binary tree, it is expected better results
when the number of classes is a power of two. Even when the
error is close to the random one, the resulting sampling set is
better. In the experiment with 14 groups (Table 3), SolutionA
and the random sampling errors are close (51.74 and 62.37,
respectively). However, the distribution for the SolutionA
is better. For example, the most infrequent group has only
0.5% of all traces. Using SolutionA, its representativeness
increases 5.2 times (2.6%). Moreover, almost all other groups
are closer to the ideal value than the random distribution.

8 Discussion

In this work, we presented two different definitions for the
tracing sampling problem. The representative sampling defi-

The first approach, SolutionA, relies on pair-wise com-
parison between traces, performs hierarchical clustering to
organize the traces and sample in the inverse proportion to
the size of the cluster that a trace belongs to, given a higher
weight to edge-case executions. SolutionA presented itself as
an effective approach, as we showed in the previous results.
In particular, in situations where it was necessary to identify
anomalies, or very distinct groups, this technique proved to
be very effective. Another advantage of this technique is that
the resulting sampling set is already grouped based on their
similarities. Therefore, an a posteriori analysis of the sampled
traces becomes easier given their organization in the resulting
binary tree.

The other approach, SolutionB, is a general-purpose trace
sampler that automatically biases sampling decisions towards
outlier and anomalous traces. SolutionB operates on a con-
tinuous stream of traces, and its computational cost is fixed
with respect to both workload volume and sampling rate. The
intuition behind SolutionB is to approximate the distributed
system’s common-case behavior, and to sample new traces
based on how well represented they are.

Although SolutionA has its strengths, SolutionB presents
as a better option. As stated earlier, the two main goals of
a tail-based sampler are the ability to make good sampling
decisions (resulting in the most representative set) and to be
scalable. Despite not being as good as SolutionA in the ex-
periments related to the anomaly detection, overall SolutionB
was more effective regarding the resulting sampling set. Be-
sides that, SolutionB has low-memory requirements and the
computational cost required to perform the sampling deci-
sions is not related to workload volumes such as the amount
of traces seen or sampled. Therefore, SolutionB is the best
option to be applied in production environments.

Another important characteristic of both tail-based sam-
pling approaches is that they are based on the structure of the
traces themselves. The main advantage of approaches like this
over manual feature engineering is that it no longer requires
explicit features. We believe that differences in high-level
metrics can always be explained by differences in the timing
and ordering of events in the underlying traces, so it is here
that sampling should be done. Nonetheless, the presented ap-
proaches are only part of the story. Useful features often are
available, such as end-to-end request latency. By definition,
these features exist because an engineer previously found
them useful for some particular problem, and outlier values
offer a strong signal that a trace should be sampled. When
this is a case, simple and computationally efficient statistical
techniques will do a good job of sampling. Our approaches are not intended to replace this use case, but to handle the case when engineered features do not capture differences between traces.

In practice, instead of a single global sampling policy, it is often desirable to specify separate sampling policies for different stratified populations. SolutionB can extend to this scenario in a straightforward way: subpopulations share the same model, but maintain a separate window of losses, each parameterized with their own sampling rate $\alpha$.

9 Related work

Traditional approaches to debugging and profiling end-to-end requests – ad-hoc analysis and per-machine logging - scale poorly in large-scale distributed systems. Causal end-to-end tracing has emerged as a valuable tool for improving the dependability of distributed systems by recording, diagnosing and analyzing their execution across components, recording their causality according to Lamport’s happens before relation [21]. A number of such frameworks exist both in academia [4, 7, 13, 38, 44] and in industry [2, 26, 36, 43, 46], and have been used for a variety of purposes, including diagnosing anomalous requests [4, 8, 37, 40]; diagnosing steady-state problems that manifest across many requests [13, 38, 40, 43, 44]; identifying slow components and functions [7, 33, 43]; and modeling workloads and resource usage [4, 33, 44].

Several prior works sought to extract value from end-to-end traces in aggregate, through clustering, classification, or anomaly detection, and similar techniques could be considered for the trace sampling problem. Pairwise graph comparison often appears as a recurring primitive [3, 9, 22, 30, 32, 39]. However, many of the approaches only consider approximated versions of traces, such as string-edit-distance of linearized graphs [3, 39], event counting [22], or grammars [9].

Magpie [4] correlates events generated by the operating system, middleware, and applications, and infers causal relations between events to produce execution graphs. It compares traces using a string-edit-distance metric on flattened execution graphs, and uses it as a basis for execution clustering. This linearized graphs end up discarding structural and temporal information which degrades the results of this approach. Like Magpie, Spectroscope [40] also linearize graphs using string-edit distance. This framework diagnoses performance changes by comparing sets of before- and after- traces, based on the request structures and/or timings. It assumes that a similar workload was run before and after the performance change, and that the performance change manifests as a change in distribution over the request structures and/or request timings. To diagnose a change, Spectroscope compares the distributions of service completion times for graphs that are topologically identical, and compares structural differences between executions using string-edit-distance. SolutionA also relies on trace comparison, based on the counting of events from each trace, to apply hierarchical clustering and infer a sampling probability for each execution.

On the other hand, SolutionB’s approach takes advantage of recent results in the area of neural language modeling [5]. In particular, approaches from this research area excel at tackling high dimensionality, inferring relationships between proximate words, and handling variable length structured input [23, 34]. We are not the first to identify an analogy between natural language and traces. Pinpoint generates a probabilistic context-free grammar from paths in order to diagnose anomalies to infer which components cause system failures [9] which is an intuitive way to model how the system generates events. It performs anomaly detection in which new traces are compared to the generated model to determine the probability with which it would have been generated by the grammar. Although it might work for some cases, it might not do well in scenarios where metrics must be represented in the training set to be considered in real time; changes such as software upgrades require the model to be retrained; and the learned model represents a superset of observed paths.

SolutionB’s approach to trace sampling is a form of online machine learning. Many online learning algorithms try to mitigate the fact that models bias towards more recently seen training samples. For our use case, we explicitly want to adapt changes in workload; but it remains open how to exactly quantify this rate of change. In general, the phenomenon of temporally varying data is called concept drift [45], and advances in this area may yield solutions that can apply to SolutionB. SolutionB takes advantage of a concept called class imbalance, which affects many machine learning models which give better predictions for more common training examples. Common techniques to mitigate class imbalance include subsampling frequently seen samples [34]; we did not use or evaluate these techniques.

Outside of distributed tracing, prior work has also found structure useful for the metrics emitted by a system [19, 31], and for logs [29]. Complementary research also looks into placing logging statements in a way that maximizes differentiating code paths [50], and extracting structure and meaning from logging messages [10, 35].

10 Conclusion

End-to-end tracing frameworks applied to modern distributed system generates high volumes of very rich data. The use of tail-based tracing sampling can improve the usefulness of this data by reducing the amount of data while still preserving interesting and diverse traces, even if very infrequent. In order to solve this problem, we presented SolutionA, a technique that relies on the use of a hierarchical clustering algorithm to derive the sampling probability of each trace. We also presented SolutionB, a general-purpose trace sampler that builds a low-dimensional unbiased model of common-case system behaviors. Using the model of common-case
behaviors, SolutionB exploits prediction error as a signal for identifying edge-case and anomalous traces, and biases its sampling decision towards these traces. In our evaluation we showed that SolutionA is very effective when dealing with anomalous traces. We also showed that SolutionB, overall, presented better results, identifying and biasing its sampling decisions towards edge case and underrepresented trace types, while having low memory requirements, making sampling decisions in the order of milliseconds.

References


[18] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O’Neill, Kian Win Ong,


[38] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In


