SampleHST: Efficient On-the-Fly Selection of Distributed Traces

Alim Ul Gias*, Yicheng Gao†, Matthew Sheldon†, José A. Perusquía‡, Owen O'Brien§, Giuliano Casale†
*University of Westminster, Email: a.gias@westminster.ac.uk
†Imperial College London, Email: {y.gao20, matthew.sheldon20, g.casale}@imperial.ac.uk
‡Universidad Nacional Autónoma de México, Email: jose.perusquia@sigma.iimas.unam.mx
§Huawei Technologies (Ireland) Co., Ltd, Email: owen.obrien@huawei.com

***Abstract*—Since only a small number of traces generated from distributed tracing helps in troubleshooting, its storage requirement can be significantly reduced by biasing the selection towards anomalous traces. To aid in this scenario, we propose SampleHST, a novel approach to sample on-the-fly from a stream of traces in an unsupervised manner. SampleHST adjusts the storage quota of normal and anomalous traces depending on the size of its budget. Initially, it utilizes a forest of Half Space Trees (HSTs) for trace scoring. This is based on the distribution of the mass scores across the trees, which characterizes the probability of observing different traces. The mass distribution from HSTs is subsequently used to cluster the traces online leveraging a variant of the mean-shift algorithm. This trace-cluster association eventually drives the sampling decision. We have compared the performance of SampleHST with a recently suggested method using data from a cloud data center and demonstrated that SampleHST improves sampling performance up to by 9.5×.**

***Index Terms*—Distributed Tracing, Microservices, Anomaly Detection, Sampling.**

I. Introduction

Distributed tracing is tailored primarily to monitoring and profiling applications built with the microservice-based architecture [1]. In a microservice ecosystem, with the increase of services, the volume of the trace data, used for observability of application performance and reliability, increases significantly [2]. In a typical production setup, each server, hosting hundreds of microservices, generates several tens of gigabytes of trace data every day. Considering all the servers, the total daily generated data are in the order of several terabytes. Nevertheless, most of the traces do not report on application anomalies and thus there is little value in storing them all. The fraction that can be retained is constrained by a storage budget [3] and the problem we study is how to select the most interesting traces to help monitoring and diagnostics of microservices runtime behavior. This entails sampling a mix of traces that characterizes the overall user behavior but at the same time retaining a high relative ratio of anomalous traces.

To accommodate the storage budget, we need to deploy a sampling strategy. It is a common industry practice to use uniform sampling [3], which is also referred as *head-based* sampling. Under this strategy, the sampling decision is taken once the request for a service is received, leading to a lower hit rate of anomalous traces. To address this issue, it is increasingly preferred to use a *tail-based* sampling strategy [4], which can improve the selection accuracy as it takes the sampling decision after the response is served, *i.e.*, when the entire trace for the service call chain is available. This allows to reason on the information contained in the trace itself upon deciding whether to store it or not.

Ideally, a tail-based sampling strategy should be online and without any batch processing. This means that we must decide either to save or discard a trace on-the-fly rather than storing it temporarily for batch processing. Recently, researchers have proposed different tail-based sampling strategies based on unsupervised learning [3], [5], [6]. However, existing research faces multiple challenges such as difficulties in performing clustering due to high dimensionality of data, requirements of batch processing, low amplitude scores for anomalous traces, and no explicit consideration of the budget size. To address all these shortcomings, we propose a novel method, *SampleHST*. On one hand, SampleHST focuses on sampling only anomalous traces when the storage budget is comparatively lower than the fraction of expected anomalies. On the other hand, when the budget is higher, SampleHST samples both the normal and anomalous traces, with a bias towards anomalous ones. Such a bias is fair because it increases the representation of the anomalous traces, which are rare compared to normal ones, among the sampled traces. In other words, the bias allows representative sampling [3], [5].

SampleHST leverages a Bag-of-Words (BoW) model [7] as a count-based representation for each trace. By taking this representation as an input, we can generate a distribution of the mass values obtained from a forest of a tree-based classifier, namely Half Space Trees (HSTs) [8]. This distribution is then used to perform an online clustering of the traces based on an algorithm we have developed which is part of the mean-shift clustering algorithm family [9]. Once the clustering is complete, we decide to sample the trace based on its cluster association, *i.e.*, a trace is more likely to be sampled if it is associated with a cluster with low mass values as such clusters represent rarely observed traces.

We evaluate the performance of SampleHST, using data provided by a commercial cloud service operator and comparing the results with a recently proposed approach for point anomalies developed in [3]. For this production dataset, we see that SampleHST yields 2.3× to 9.5× better sampling performance in terms of precision, recall and F1-Score than prior work. When we consider representative sampling in a

high budget scenario, we see SampleHST is 1.6× fairer with respect to the Jain fairness index [10]. In summary, the key contributions are:

- A novel approach to sample distributed traces by forming clusters using the mass distribution of the traces obtained from Half Space Trees.
- An online clustering method, generalizing the mean shift algorithm [11], that considers non-spherical cluster shapes such as hyper-cubes and hyper-rectangles.
- Experiments using real-world data to compare the sampling performance of SampleHST with a recent tail-based sampling approach [3].

The rest of the paper is organized as follows. Section II presents the related work and motivation for developing SampleHST. Section III demonstrates how to model traces and detect anomalies. Section IV discusses how to transform anomaly detection processes to a sampling method. Section V and VI present the SampleHST clustering and sampling algorithms respectively. Section VII evaluates the sampling performance. Section VIII concludes the paper.

II. Background

A. Related Work

The first step of designing a sampler is to differentiate the anomalous traces from the normal ones. There have been many works on anomaly detection for microservices using their generated traces. The authors in [12], [13] learn from the patterns of call trees and request execution respectively to detect anomalies. Some studies [14]–[16] also consider deep learning based methods focusing on different aspects, *e.g.,* response times and causal relationships. However, these works do not consider our sampling scenario, *i.e.*, they only focus on anomaly detection but not on transforming the anomaly detection result to a sampling decision.

To the best of our knowledge, there are only a few research papers focusing on sampling anomalous traces generated by microservices. In [3], the authors propose a sampler based on a hierarchical clustering method PERCH [17]. Authors demonstrate that their method can achieve representative sampling, meaning equal share for both normal and anomalous traces. Such clustering methods can incur the curse of the data dimensionality [18] and they often require batch processing, which is not always supported under low latency requirements.

Sifter [5] avoids batch processing by taking sampling decisions trace-by-trace. It generates a sampling probability by utilizing the loss of training a neural network for a particular trace. A potential issue with loss-based methods is that anomalous traces may still have small probabilities overall, closer to 0 than to 1, allowing several anomalous traces to go unsampled. This problem is studied in recently proposed sampler, Sieve [6], which uses a threshold to first separate the anomalous traces and then amplify the sampling probability. This still leaves an open challenge regarding the optimal and automated choice of threshold.

B. Sampling performance

As a classification problem, it may be natural to study trace sampling performance in terms of F1-Score, as this strikes a balance between Precision and Recall. We however observe that this is not always an ideal performance criterion in the presence of budget constraints. For example, an abundant storage budget with few constraints is more appropriate to consider Recall, while a heavily constrained storage budget expects more from achieving high Precision. Summarizing, we set the following overall performance evaluation principles for trace sampling methods:

- For infrequent anomalous traces, where the prevalence of anomalies is less than the storage budget, the primary evaluation metric should be the Recall.
- For low storage budgets, where the prevalence of anomalies is greater than the storage budget, the primary evaluation metric should be the Precision.
- When sampling N traces from a collection of traces containing N anomalies, the primary evaluation metric should be the F1-Score.

C. Comparing State-of-the-Art Anomaly Detection Methods

Since anomaly detection is a key step for a sampling process, we here illustrate why off-the-shelf anomaly detection methods are not fit for purpose. We consider the following popular techniques: 1) local density estimate: *K-Nearest Neighbor* (KNN) and *Local Outlier Factor* (LOF), 2) tree-based classification: *Isolation Forest* and *Half Space Trees* (HST) [8], 3) boosting: *Lightweight Online Detection of Anomalies* (LODA) [19], and 4) neural network: *Deep Belief Net and One Class Support Vector Machine* (DBN+OCSVM) [20]. A notable advantage of using the tree-based methods is that they can work on one trace at a time, while the other methods, off-the-shelf, require batching.

To evaluate the performance of the above methods, we consider a production dataset from a cloud data center consisting of trace data spanning a week over a set of 14 microservices. As the trace is unlabelled, we identify $\sim 5\%$ point anomalies using the popular offline DBSCAN clustering algorithm, and evaluate the ability of the listed methods to obtain similar results. DBSCAN, being resource intensive, is not feasible in an online scenario such as distributed trace sampling, but is considered as a generally reliable technique in industry [21]. We use Matlab's native implementation of DBSCAN with $\epsilon = 2.5$ and $minpts = 5$, where ϵ indicates the size of the local neighborhood of the data points and $minpts$ indicates the minimum number of points per cluster. Once the traces are clustered, we regard the smallest clusters as anomalies, accounting for $\sim 5\%$ of the total traces.

The results of the experiment are presented in Table I. The dataset contains traces from six consecutive days with 77577 traces. For all the batch methods, we keep a similar batch size of 2000 traces. We see that HST is the best method with respect to F1-Score. This motivates further investigation in HST methods to address the problem under study. In addition, HST has other benefits from the perspective of a streaming

TABLE I
RESULTS OF DIFFERENT ANOMALY DETECTION METHODS ON THE PRODUCTION DATASET

	Isolation Forest	**KNN**	**LOF**	**LODA**	**DBN + OCSVM**	**HST**
Precision	0.73	0.77	0.73	0.62	0.47	0.94
Recall	0.72	0.72	0.72	0.60	0.97	0.70
F1-Score	**0.73**	**0.74**	**0.73**	**0.61**	**0.64**	**0.80**

platform. Due to the way HSTs are designed, for a particular trace, we only need to update a single mass value [22] per tree. To determine whether a trace is normal or anomalous, the mean mass value (m) of the HSTs, for that particular trace, is compared against a threshold. An HST only needs to query its already stored mass values, resulting in a very low computational footprint in the order of less than a millisecond per trace. This will reduce the time taken during the training, where we can only use the computing resource to update the mass values of the node. Due to all these benefits, the rest of the paper focuses on HST as a baseline classifier.

III. HALF SPACE TREES FOR ANOMALY DETECTION

Half Space Trees (HST) [8] are an ensemble of decision trees. The structure of the decision trees is a simple Binary Tree. Each HST has a depth d, and the corresponding binary tree will have $2^{d+1} - 1$ nodes. Each tree stores split points for a random subset of dimensions, and possibly multiple splits per dimension, together with a count of how many points are within the subspace defined by a path (a metric called *mass*). Mass is simply defined as a count of data points, thus it is easier to calculate than density measures used in other methods, *e.g.*, which require likelihood estimation. Normally, an ensemble of t Binary Trees is used, with identical depth h, which are independently trained on a data window w.

HSTs are particularly suitable for streaming data as its core processes - building the tree data structure and characterizing the data points using the mass values - are both lightweight [8]. In this study, we assume that such data points will be available of continuously arriving streams of spans generated in a cloud data center from a heterogeneous collection of microservices. A span is an immutable data structure that supplies the value of a collection of categorical and continuous variables at a particular point in time. The spans contain a *traceId*, based on which they can be grouped to form traces. We propose to abstract each trace as a document where the span properties are considered as words or terms. The document is subsequently converted to a bag of words [7].

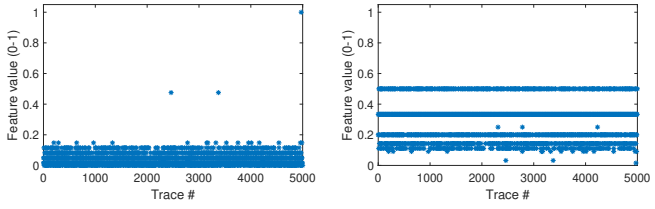
During the conversion, the span properties that are not relevant to performance and reliability analysis are ignored. We restrict our attention to discrete fields, some of which, *e.g.*, HTTP code, can be categorical *i.e.*, they have a fixed number of possible values. In addition, we do not explicitly address latency anomalies as they are often best studies with with anomaly detection based on continuous response time distribution estimators, which can be already done with specialized methods in the literature [23]–[25]. Alternatively, latencies can be discretized and considered as one of the features considered by our method. We represent each trace using a count vector $x = (x_1, \ldots, x_d, \ldots, x_D)$, where D is the number of different terms that have been seen across all the traces. For example, the HTTP code 200 is one term and a specific URL could be another one. Each dimension $x_d \geq 0$ is an integer value counting how many time a particular term appears in a trace. The resulting count data assures knowledge of the dimension D and the mappings of dimensions to terms. In a production implementation, such knowledge can be acquired from an initial monitoring period and periodically updated.

In production data, sparsity is frequently observed. Once the categorical properties of the spans are vectorized as count data, there are relatively few types of traces that occur repeatedly, thus the HST mass could accumulate within a small set of terminal nodes. This is confirmed from our production data where we observe that only 0.004% of the trace count vectors are unique. We thus focus on a variant of HST known as HS*-Trees (HS*T) [22], which aims to deal with the sparsity in the tree structure. In HS*T, nodes that have fewer than *SizeLimit* samples are not further expanded during the training phase. This reduces memory consumption and also the time to traverse the trees. Thus, we have opted for HS*T as our chosen HST variant. In the rest of the study, we use the term HST and HS*T interchangeably.

We incorporated two further modifications to HS*T. Firstly, we opted for depth-dependent split dimension. This means that when splitting a node, instead of using the normal procedure of picking a dimension at random, we require all nodes at the same depth level to use the same split dimension, which largely reduces memory usage since a single dimension is stored at each level. Secondly, as suggested in [8], we opted for a $[0, 1]$ workspace. This means that, the maximum and minimum values of the features are assumed by the HS*T to be 1.0 and 0.0, rather than in the min-max range observed in the data. This can simply be achieved with min-max scaling. However, an issue with such count data scaling is that outliers can often cluster the normal values at one end of the range, making the prediction particularly difficult for tree based methods since they rely on randomized partitioning of the input space, *i.e.*, random split points will be chosen in the segment [0,1] to branch the tree along a dimension. Therefore, if the points are all clustered in a small portion of the range [0,1] the HS*T will struggle to separate the samples along that dimension. To address this, we apply the following transformation in place of the min-max scaling

$$f(x) = \frac{1}{1 + g(x)}. \tag{1}$$

that allows us to control the stratification of the count data. We have found it sufficient to use $g(x) = x$ but we could also define, for example, $g(x) = \log(x)$ considering large values of x. Using (1), the large outliers will be squeezed near 0, therefore not suppressing the ability to resolve the normal values That are critical to HST training. We illustrate the impact of this transformation in Fig. 1 using 5000 randomly



(a) Min-Max Scaling (b) Transformation function f

Fig. 1. Comparing the scaled value of HTTP 200 code counts with min-max scaling and the transformation function f

chosen traces, where we scaled the feature corresponding to the frequency of HTTP code 200 in the trace.

As before, we used the production data from Section II-C to test these modifications. We consider each day as a window and use the first day to build the trees. We observe that the F1-Score improves from 0.8 to 0.97. This indicates that the changes aid in anomaly detection from the trace streams.

IV. Mass-based Clustering for Sampling

Although HSTs can help in classifying the anomalous traces, in reality we need to utilize this classification output in a sampling process. This process is complex because of the trade-off between sampling normal and anomalous traces. While sampling, the proportion of the storage budget and the expected percentage of anomalies should be taken into account. If the budget is lower than the anomaly percentage, the focus should be on sampling mostly the anomalous traces. The normal traces should gain more attention only when the budget is higher than the anomaly percentage. In addition, while sampling the anomalous traces, the target should be representative sampling from that group of traces *i.e.* sampling from different "groups" of traces fairly.

To achieve this, we propose to cluster the traces and decide whether to sample a trace or not based on its cluster association. However, when clustering in a high-dimensional space it is harder to achieve accurate density estimation [26], in addition to incurring a higher computational cost. This is expected in the normal behavior of our system, as our production data contains hundreds of features. Therefore, we propose a new approach considering the distribution of mass across the trees in the HS*T forest and selecting a mean mass score m and a low percentile of the mass score p. Low percentiles are expected to significantly differ from the mean when there is at least a subset of trees in the forest that identifies the trace as an anomaly. We refer to this method as *SampleHST* as we are using the the mass distribution of HST to perform sampling.

Since we want to use a low percentile (p) value along with the mean (m), we represent each trace with a unique pair (m, p) that will be used for clustering. The projection of the production traces from Section II-C in this 2-dimensional space is shown in Figure 2. The figure shows in different colors the clusters obtained by DBSCAN. It is clearly seen that the mass-based properties cluster the traces in distinct groups and the cluster centers are also appropriately detected using a

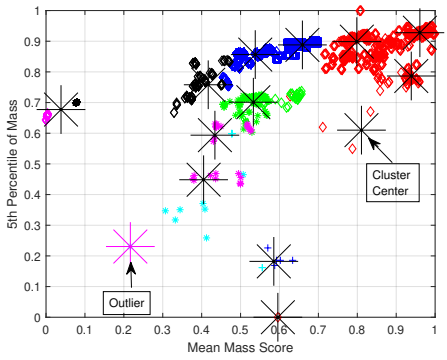


Fig. 2. The production trace plotted using the mass-based properties. The colors and marker shapes indicate the DBSCAN original clusters. The cluster centers are estimated with a baseline online clustering method.

baseline streaming clustering method [27]. Another potential benefit of using the low percentile value is a better separation of trace groups. As seen from Fig. 2, ignoring the percentile value will result in multiple trace groups being merged together, eventually affecting the sampling performance.

This mass-based clustering is at the core of our sampling approach. Once a trace is formed with its spans, to take a sampling decision, it is moved through two key components:

- **SampleHST Clustering**: Cluster the trace based on the its mass based properties.
- **SampleHST Controller**: Makes the sampling decision based on budget and trace-cluster association.

We discuss these components in details in the next sections.

V. SampleHST Clustering

SampleHST Clustering is primarily based on the underlying theory of *mean-shift analysis* [9] and the CEDAS algorithm [27] yielding a data-driven online approach that generalizes the hyper-sphere cluster shape commonly assumed in the literature to hyper-rectangles and hyper-cubes. Broadly speaking, our method receives the mass score of a trace in the form of a pair (m, p), which is generated using the HST mass distribution. Subsequently, the method aims to find the association of the new trace with an existing cluster, if the association condition is not met a new cluster is created and a signal is send. Furthermore, the method is able to remove clusters that have not received a new trace for a pre-defined period of time modulated by the *decay* and the *life (energy)* parameters and also merge clusters together whenever an overlapping occurs. These steps can be broadly grouped into two sets of tasks: *trace association* and *cluster management*. We now discuss the key aspects of these tasks.

A. Trace Association

1) Cluster Shape: A common assumption for online clustering algorithms for data streams is that the cluster shape is a hyper-sphere [9], [27]. In our case, the problem with such shapes is that they can lead to inaccurate partitioning of the traces because the normalized values of the unique pair (m, p) belong to the unit hyper-cube. To address this issue we consider instead an arithmetic average kernel whose support

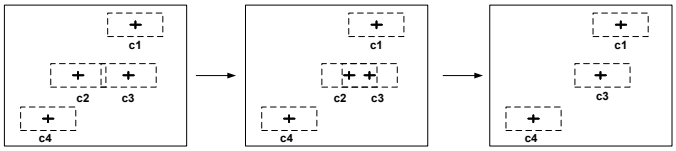


Fig. 3. Demonstrating cluster merging process. Initially, though there is an overlap between the boundary of cluster c2 and c3 they are not merged. Once their centroids overlap, they are merged into single cluster c3

is a hyper-rectangle [28]. Assuming d data dimensions, the kernel considered is presented in (2) for which we further show in Theorem 1 that the mean-shift property is achieved if the clustering bandwidth [29] is equal in all dimensions.

Theorem 1. *For the additive kernel defined as*

$$K_d(u_1, \dots u_d) = \begin{cases} \frac{3}{d2^{d+1}} \sum_{k=1}^{d} \left(1 - u_k^2\right) & \text{if } |u_k| < 1, \forall k \\ 0 & \text{otherwise}, \end{cases} \quad (2)$$

the mean-shift algorithm at each iteration shifts each sample with a value equal to the local mean if the support is given by a hyper-cube.

Due to space limitations, we present the proof in our online preprint[1].

2) Cluster Assignation: The assignment step requires a pre-defined clustering bandwidth. We define the bandwidth vector, $H = \{h \in \mathbb{R}^d | \forall i = 1, \dots, d,\ 0 < h_i \leq 1\}$, where each value $h_i \in H$ defines the Manhattan distance from the center to the boundary of the cluster in the i^{th} dimension. Now, if we define a vector of Manhattan distances between a cluster centroid and a new data point as $M = \{m \in \mathbb{R}^d | \forall i = 1, \dots, d, 0 \leq m_i \leq 1\}$, then if $\forall i\ m_i \leq h_i$, we assign the data point to that cluster. Otherwise, a new cluster is created with that point.

3) Centroid Update: Appropriately updating the cluster centroid is critical since *SampleHST* uses the centroid distance to decide the mapping of traces to clusters. In general it is preferable to update the centroid giving more importance to traces that are unequivocally within that cluster. This is the concept of *cluster kernel region* [27]. Given the clustering bandwidth vector $\mathbf{H}$, we can define the kernel region as the sub-space within a cluster with bandwidth $r\mathbf{H}$, where the scalar r quantifies the proportion of the cluster considered as the kernel region.

B. Cluster Management

1) Cluster Merging: To address the overlaps among clusters as they are indications of possibly inaccurate clustering, we opt for the policy that merges two clusters only when the centroid of one overlaps with the boundary of the other. This policy is less drastic than merging two clusters when their boundaries overlap because one distant point cannot shift the cluster center unless the cluster has a very few samples. An illustration of this policy is presented in Fig. 3.

[1]Available at: https://arxiv.org/abs/2210.04595

2) Cluster Removal: We need to regularly remove the clusters whose population have remained static for a while since they are unlikely to be relevant and might affect the sampling policy. We realize this by using the *decay* and *life (energy)* parameters for the clusters as in [27]. The life property is initially set to one and gradually reduced using the decay value, which is set as the average number of traces in the work cycles, defined as a sequence of consecutive periods where we received at least 1 trace, within the sampling window.

VI. SampleHST Controller

A. Overview

The SampleHST controller takes the sampling decision by utilizing the clustering method we have presented. The controller initially calculates the number of traces (s_w) that need to be sampled from the next sequence of w traces. We refer to this number as sampling limit and the sequence as a window. For a given budget τ, the sampling limit is defined as $s_w = \tau w$. The budget is held constant, therefore the sampling limit only varies with w over the runtime. The sampling process runs continuously according to Algorithm 1, using HST mass scores x_m. The algorithm expects a set of inputs that defines the size of the sampling window (w), the budget (τ), the total number of traces to be sampled in this window (s_w), the relative position of the current trace in the window ($w_i^{(p)}$), the number of traces that still remain to be sampled (s_r), the current clusters status (C), the clustering bandwidth vector (H) and the length of the system work cycle (β).

The algorithm initially performs a series of pre-processing on the received data. Subsequently, the locality of the trace, represented by its associated cluster index, is determined by SampleHST clustering. The final step is the sampling decision based on the inclusion of the trace in a set of prioritized clusters, which we refer as the *selection pool*. This step is skipped if the sampling target has already been reached. Since we already discussed the SampleHST clustering method, we now present the other key aspects of the controller.

B. Online Score Scaling

The first step in Algorithm 1 is to make the adjustments to the sampling window size estimate and the sampling target when the current window is larger than the expected window size. This is followed by log-transformation and min-max scaling of mass scores: $x_m^{(s)} = [\log_b(x_m) - \min(x_m^s)]/[\max(x_m) - \min(x_m)]$. It should be noted that before the log transformation, the mass scores are expected to be standardized. The SampleHST clustering method uses two mass scores, the mean and the 5^{th} percentile of the mass, to cluster the traces ($p = 0.05$). Since we are using HS*T, we use the mass value $m[l]2^l$, where $m[l]$ is the mass of the terminal node where the trace falls into and l is the depth of the corresponding tree node. To standardize the mass scores, we scale down the augmented mass using the maximum mass value possible, which is $w2^d$ where d is the tree depth and w is the number of observed traces.

Algorithm 1 Sampling Process

Require: massScores (x_m), budget (τ), idxPriWindow ($w_i^{(p)}$), windowSize (w), remainingTarget (s_r), windowTarget (s_w), clusters (C), bandwidth (H), workCycleLen (β)
Ensure: $decision$

```
if w_i^(p) > w then
    AdjustParameters()
end if
x_m^(log) = log_b(x_m)
x_m^(s) = ScaleScores(x_m^(log))
if HasMaxMinChanged() then
    ReScaleClusterCenters()
end if
(C, x_c) ← GetTraceLocality(x_m^(s), C, H, 1/β)
R = w_i^(p) / w
U = (s_w − s_r) / s_w
if s_r > 0 then
    decision = IsTraceInSelectionPool(C, x_c, τ, R, U)
end if
if decision then
    s_r = s_r − 1
end if
```

Once the mass scores are processed, it is checked that whether the minimum or maximum values change along with the new mass scores in the current sampling window. If this is the case, all the cluster centers are re-scaled. This is followed by clustering the trace and taking the sampling decision.

C. Sampling Decision

The sampling decision procedure needs to decide on-the-fly whether to sample a trace or not. If a new cluster is created by a trace the methods always sample it. For the case where the trace is associated with an existing cluster, we rely instead on generating a prioritized pool of clusters, which we refer as *selection pool* and use it to take the decision. This is done in three steps, which are described as follows.

1) Distance-based Cluster Ranking: The first step is to rank the clusters. Two methods of ranking were considered: size of the cluster and Euclidean distance from the origin. Cluster size is an obvious method of ranking, but since SampleHST creates and deletes clusters online, smaller clusters might not always represent less frequent traces. A cluster might be smaller but all of its traces can have high mass values. This means that the traces have hit HST nodes with a high mass count which indicates that these traces are quite frequent. In addition, the most interesting and possibly smallest clusters are likely to be near the origin, which represents a low mass region in the clustering place. Therefore, we chose Euclidean distance of the centroids to the origin $(0, 0)$ and if a cluster is closer to the origin, traces associated with it will be sampled first even if that cluster is not the smallest.

2) Selection Pool: Once the clusters are ranked, we decide how many of those will form the initial selection pool. Clusters are added according to the above ranking, starting with the one closest to the origin, until the threshold θ is reached. If two clusters are equidistant, the one created first is prioritized.

After creating the initial selection pool, we start the second phase by checking the actual value of the percentage total population in the selection pool denoted by $\hat{\theta}$. If the actual percentage is less than $\alpha\%$ of the budget, we add more clusters in the selection pool. The clusters are added depending of the magnitude M of the budget (τ) in comparison to $\hat{\theta}$. This is defined as $M = \left\lfloor (\tau - \hat{\theta})/\hat{\theta} + \frac{1}{2} \right\rfloor$. We then make M independent attempts to add the clusters in a probabilistic manner, where in the k^{th} attempt, the k^{th} closest cluster to the origin, which is not yet included in the selection pool, is chosen with a probability P^k. Here each attempt of being successful has the same probability $P = \max(\tau, S)$, where τ is the budget and S is the sampling eagerness defined as

$$S = R(1 - U). \tag{3}$$

This sampling eagerness is bounded between $[0, 1]$ and a high value indicates to sample more. It is defined in terms of the budget utilization (U), which is the ratio of number of sampled traces to the sampling limit, and the relative trace position in the current window (R), which is the ratio of the trace index in the current window to the sampling window size.

3) Decision Process: After the selection pool has been decided, we sample the new trace only if it is associated with any of the clusters in the pool. If that is the case, one of two paths may be followed. If the budget is greater than or equal to the actual percentage of population in the selection pool ($\tau \geq \hat{\theta}$), we sample the trace straightaway. Conversely, if the budget is less than the actual percentage, we follow the second path that takes a probabilistic sampling decision. This is to sample cautiously as we may have larger clusters in the selection pool containing common traces. In this path, we set the probability of sampling as

$$P_s = \begin{cases} \dfrac{\tau}{\hat{\theta}} & \text{if } \Gamma_c > \Gamma_\mu + k\Gamma_\sigma \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

Here we set the probability based on the cluster size. Firstly, if the size of the cluster (Γ_c), which is associated with the current trace, is greater than the sum of mean (Γ_μ) and k standard deviation (Γ_{sigma}) of the cluster size in selection pool, we set the sampling probability to $\tau/\hat{\theta}$. This means that, if there are N traces, the size of the selection pool will be $N\hat{\theta}$ and we would like to sample $N\tau$ traces from those in the selection pool. Secondly, if $\Gamma_c \leq \Gamma_\mu + k\Gamma_\sigma$, we set the sampling probability to 1. This means if the cluster is sufficiently small, we decide to sample the corresponding trace. The value of k is set using Chebyshev's inequality [30], which estimates the minimum percentage (V) of values within k standard deviation of the mean. For a given V, we can solve the inequality to determine the value of k. We notice that, this percentage V is related to the ratio of $\tau/\hat{\theta}$. Because,

if τ is much smaller than $\hat{\theta}$, we want to sample only if the associated cluster is smaller than the majority of the clusters. As the value of τ increases compared to $\hat{\theta}$, we can consider the larger clusters i.e., larger value of V. Thus, we consider $\hat{V} = \tau/\hat{\theta}$, where $\hat{V}$ is and estimate of the minimum percentage V, we can calculate the value of k using (5).

$$k = \sqrt{\frac{1}{1 - \frac{\tau}{\hat{\theta}}}} \equiv \sqrt{\frac{\hat{\theta}}{\hat{\theta} - \tau}}. \tag{5}$$

VII. Sampling Performance

A. Experimental Setup

To test SampleHST performance, we use a dataset provided by a cloud data centre composed of 77,577 traces. Each trace contains at least one span and the following four categorical features: *Service Name, URL, Process Id*, and *Node Id*. A span also contains the http return code and http method for the service invocation. The dataset includes 14 different services with four of them containing 98% of the spans; more than 50 different URLs with four accounting for 95% of the spans; more than 40 different Process Id's with 20 containing 91% of the spans; and 8 different node Id's with four containing 88% of the spans. The traces are represented as a count vector using the BoW model as detailed in Section III. Through this, we obtain 105 unique features. Ignoring timestamps, the 77,577 traces map to 308 unique traces.

To test the SampleHST robustness, we consider 5 cases with different storage budgets. First, since we have about 5% anomalies in our data, we include a case where the budget is 5%. The evaluation criteria for this case is the F1-Score. We have also chosen 3 smaller budgets (0.5%, 1% and 2%) where the evaluation criteria is precision. Finally, we also consider a high budget case of 10%, where the evaluation criteria is recall. We compare the results with two other samplers: uniform random sampler, implemented following the Head-based sampler in [5], and the PERCH-based method [3].

Since sampling methods such as [3], [5] focus on representative sampling, we also compare their fairness in terms the Jain index [10]. The index can be calculated using (6) where $X_i = \frac{T_i}{O_i}$. Here, for each cluster i, T_i is the number of traces sampled by a method and O_i is the optimal number of traces that should be sampled. This metric indicates what percentage of the groups are treated fairly. In our case, the groups are the clusters that we obtain offline from DBSCAN clustering. Note that, to calculate the index, we need to know the optimal number of traces that should be sampled. As we know the overall distribution of the traces among the groups and sampling budget, we calculate it offline using the max-min fair allocation approach [31].

$$\mathcal{J}(X_1, X_2, \ldots, X_n) = \frac{\left(\sum_{i=1}^{n} X_i\right)^2}{n \sum_{i=1}^{n} X_i^2} \quad X_i \geq 0 \tag{6}$$

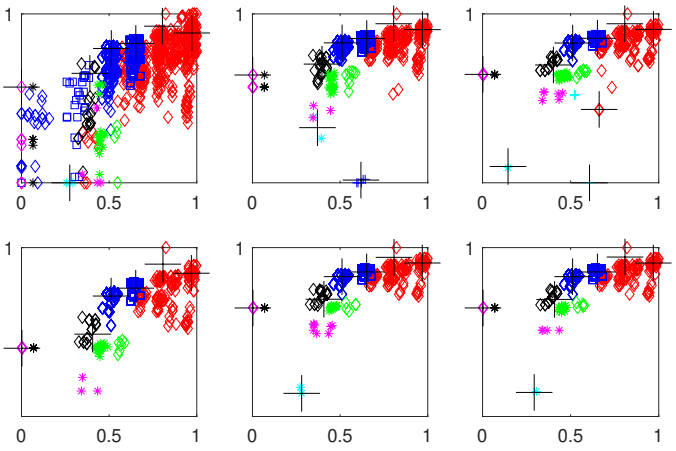


Fig. 4. Output of the SampleHST clustering algorithm. The X-axis and Y-axis represent mean and 5$^{\text{th}}$ percentile of mass respectively. The colored symbols represent different DBSCAN labels. The + signs are the cluster centers estimated by SampleHST clustering. The output is presented in 6 windows. As we move from left to right, we move towards the next window.

B. Results

SampleHST Clustering Operation. We begin by illustrating in Fig. 4 the operation of the SampleHST method. Since, this is an online clustering method, we divide the total time frame in six periods and show the clustering status for those periods. We immediately see that in the first window, the data points are less segregated. This is because of the online min-max scaling. In the initial period, the min-max values are not steady, which affects the data points as well. As we progress towards the end, we can see that the clusters are increasingly segregated. We also see that the number of clusters continue to change throughout these periods. The clusters around the top right corner remains stable, but the ones around the bottom left corner change their positions frequently as the top right clusters are of frequent traces whereas the bottom left ones are of the infrequent ones. The infrequent trace clusters decay quickly by not receiving traces in some work cycles.

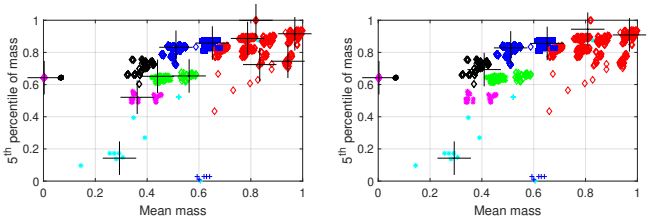
Comparative experiments. We now compare the performance of SampleHST against the uniformly random and PERCH-based methods. In Table II we see that SampleHST with a bandwidth of $h = 0.1$ is the best method across all budgets, with the uniform random sampler performing the worst. We also see that the PERCH-based method does not perform significantly better with respect the precision, recall and F1-Score. From the fairness perspective, the PERCH-based method scores much higher than the random sampler, but still it cannot outperform SampleHST. The results show that even though the PERCH-based method can achieve better Jain score in low budgets, it is not precise in sampling the anomalous traces as made evident by the precision score.

As we mentioned earlier, identifying anomalous traces is difficult for clustering methods due to the high number of dimensions of the input data, as in the present case with 105 dimensions. SampleHST, on the other hand, eliminates this problem by using the mass scores, which are low dimensional.

We now focus on the case with high budget (10%). Firstly,

TABLE II
PERFORMANCE OF DIFFERENT SAMPLERS WITH DIFFERENT BUDGET

		0.5%	1%	2%	5%	10%
Uniform	**J**	0.10	0.10	0.11	0.13	0.18
	P	**0.05**	**0.04**	**0.06**	0.05	0.05
	R	0.01	0.01	0.03	0.05	**0.10**
	F1	0.01	0.01	0.04	**0.05**	0.06
PERCH-based	**J**	0.32	0.24	0.32	0.47	0.56
	P	**0.41**	**0.18**	**0.13**	0.11	0.09
	R	0.03	0.03	0.04	0.09	**0.15**
	F1	0.05	0.04	0.07	**0.10**	0.11
SampleHST	**J**	0.40	0.59	0.72	0.75	0.88
	P	**0.84**	**0.83**	**0.86**	0.92	0.80
	R	0.10	0.18	0.37	0.91	**0.94**
	F1	0.17	0.30	0.52	**0.92**	0.87



(a) $h_1 = h_2$ (b) $h_1 \neq h_2$

Fig. 5. Comparing clusters with equal and unequal clustering bandwidth

we see that SampleHST easily outperforms the PERCH-based method considering the primary evaluation criteria recall. Secondly, when we consider representative sampling, we see that the Jain score produced by SampleHST is 1.6× better than the PERCH-based method. The reason for SampleHST performing better is as follows. The primary objective of SampleHST is to sample as much as anomalous traces possible. In high budget cases, it only shifts focus towards normal traces when the primary objective is fulfilled. Anomalous traces can create many groups, each with a small size, whereas normal traces create a small number of large groups. This is indeed the case with the production data. As a result, when SampleHST samples most of the traces from anomalous groups, it satisfies the demands of majority of the groups, making it more fair which is reflected in the Jain score.

SampleHST with Hyper-Rectangles. The mass scores work as anomaly signals to the SampleHST, which are not always likely to be equally strong in all clustering dimensions. In such cases, the traces may not be segregated ideally in that dimension. This is not a problem as long as we can separate anomalous traces from normal ones. However, if the bandwidth in that dimension is small, we can have multiple clusters in a particular region in the clustering hyper-plane, which represents traces of similar types. Thus rather than using a small clustering bandwidth in that dimension, as illustrated in Fig. 5, we can chose a large one to remove clusters containing similar traces, allowing a more precise clustering. In other words, we can opt for hyper-rectangles, with unequal clustering bandwidths in each dimension, instead of hyper-cubes. When we observe the clustering status, as presented in Fig. 5, indeed with hyper-rectangles there are less number of clusters in the top right corner, that represents normal

TABLE III
PERFORMANCE OF SAMPLEHST CONSIDERING HYPER-RECTANGLES

	Jain	Precision	Recall	F1-Score
0.05, 0.1	0.76	0.90	0.91	0.91
0.05, 0.2	0.75	0.91	0.91	0.91
0.05, 0.3	0.74	0.93	0.91	0.92
0.1, 0.2	0.74	0.94	0.91	0.92
0.1, 0.3	0.73	0.97	0.92	0.95

TABLE IV
SAMPLING RESULTS WITH HYPER-CUBES AND HYPER-RECTANGLES

	$h = 0.1$				$[h_1, h_2] = [0.1, 0.3]$			
	J	**P**	**R**	**F1**	**J**	**P**	**R**	**F1**
0.5%	0.40	**0.84**	0.10	0.17	0.41	**0.94**	0.10	0.18
1%	0.59	**0.83**	0.18	0.30	0.50	**0.95**	0.21	0.34
2%	0.72	**0.86**	0.37	0.52	0.47	**0.96**	0.41	0.58
5%	0.75	0.92	0.91	**0.92**	0.73	0.97	0.92	**0.95**
10%	0.88	0.80	**0.94**	0.87	0.88	0.79	**0.94**	0.86

traces. Having less number of traces reduces the probability of sampling from normal groups, which is essential in low and moderate budget cases. This is also reflected in the sampling performance. In Table III we present the results, for the 5% budget case and for different sizes of hyper-rectangles. From these results, we can appreciate that the F1-Score for bandwidth $[0.1, 0.3]$ reaches 0.95, which is higher than the one we achieved for hyper-cubes presented in Table II. Moreover, and considering the hyper-rectangle $[0.1, 0.3]$ as our baseline we can see in Table IV that the hyper-rectangles approach yields significantly better results in the metrics considered. In particular for low-budget scenarios we achieve on average an improvement of $1.12\times$ with respect to hyper-cubes.

VIII. CONCLUSION AND FUTURE WORK

In this paper we propose a novel sampling method for distributed tracing namely *SampleHST*. The objective of SampleHST is to take its sampling decision based on the proportion of sampling budget and the fraction of expected anomalous traces. If the budget is lower, the priority is to sample the anomalous traces. On the other hand, when the budget higher, the normal traces are sampled as well. This sampling process is based on an online clustering mechanism. The traces are first clustered using their mass scores generated using a forest of HST. After that, if the budget permits, the sampling decisions are taken based on the association of a trace with a cluster, where the clusters more likely to contain anomalous traces are prioritized. Our experiments, that considers production data from a cloud data center, show that SampleHST by far outperforms the recent approach targeting point anomalies.

A possible line of future research direction could be integrating the continuous trace properties, like the response time, to identify also the latency anomalies in an integrated approach.

ACKNOWLEDGMENTS

This research has received funding by Huawei Technologies (Ireland) Co., Ltd.

REFERENCES

[1] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[2] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," in *Proc. of ESEC/FSE*. ACM, 2020, pp. 1387–1397.

[3] P. Las-Casas, J. Mace, D. Guedes, and R. Fonseca, "Weighted sampling of execution traces: Capturing more needles and less hay," in *Proc. of SoCC*. ACM, 2018, pp. 326–332.

[4] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[5] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proc. of SoCC*. ACM, 2019, pp. 312–324.

[6] Z. Huang, P. Chen, G. Yu, H. Chen, and Z. Zheng, "Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems," in *Proc. of ICWS*. IEEE, 2021, pp. 436–446.

[7] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: a statistical framework," *Intl. Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 43–52, 2010.

[8] S. C. Tan, K. M. Ting, and T. F. Liu, "Fast anomaly detection for streaming data," in *Proc. of IJCAI*, 2011.

[9] R. D. Baruah and P. Angelov, "Evolving local means method for clustering of streaming data," in *Proc. of FUZZ-IEEE*. IEEE, 2012, pp. 1–8.

[10] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System," *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[11] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Trans. on Information Theory*, vol. 21, no. 1, pp. 32–40, 1975.

[12] T. Wang, W. Zhang, J. Xu, and Z. Gu, "Workflow-aware automatic fault diagnosis for microservice-based applications with statistics," *IEEE Trans. on Network and Service Management*, vol. 17, no. 4, pp. 2350–2363, 2020.

[13] Y. Zuo, Y. Wu, G. Min, C. Huang, and K. Pei, "An intelligent anomaly detection scheme for micro-services architectures with temporal and spatial data analysis," *IEEE Trans. on Cognitive Communications and Networking*, vol. 6, no. 2, pp. 548–561, 2020.

[14] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection and classification using distributed tracing and deep learning," in *Proc. of CCGRID*. IEEE, 2019, pp. 241–250.

[15] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *Proc. of CLOUD*. IEEE, 2019, pp. 179–186.

[16] J. Bogatinovski, S. Nedelkoski, J. Cardoso, and O. Kao, "Self-supervised anomaly detection from distributed traces," in *Proc. of UCC*. IEEE, 2020, pp. 342–347.

[17] A. Kobren, N. Monath, A. Krishnamurthy, and A. McCallum, "A hierarchical algorithm for extreme clustering," in *Proc. of SIGKDD*, 2017, pp. 255–264.

[18] A. Zimek, E. Schubert, and H.-P. Kriegel, "A Survey on Unsupervised Outlier Detection in High-Dimensional Numerical Data," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 5, no. 5, pp. 363–387, 2012.

[19] T. Pevnỳ, "Loda: Lightweight on-line detector of anomalies," *Machine Learning*, vol. 102, no. 2, pp. 275–304, 2016.

[20] S. M. Erfani, S. Rajasegarar, S. Karunasekera, and C. Leckie, "High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning," *Pattern Recognition*, vol. 58, pp. 121–134, 2016.

[21] P. Fisher-Ogden, G. Burrell, C. Sanden, and C. Rioux, "Tracking down the Villains: Outlier Detection at Netflix," https://netflixtechblog.com/tracking-down-the-villains-outlier-detection-at-netflix-40360b31732, 2015, Accessed: 2023-01-25.

[22] K. M. Ting, G.-T. Zhou, F. T. Liu, and S. C. Tan, "Mass estimation," *Machine Learning*, vol. 90, no. 1, pp. 127–160, 2013.

[23] R. Li, M. Du, Z. Wang, H. Chang, S. Mukherjee, and E. Eide, "LongTale: Toward Automatic Performance Anomaly Explanation in Microservices," in *Proc. of ICPE*, 2022, pp. 5–16.

[24] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," in *Proc. of NOMS*. IEEE, 2020, pp. 1–9.

[25] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei, "Localizing Failure Root Causes in a Microservice through Causality Inference," in *Proc. of IWQoS*. IEEE, 2020, pp. 1–10.

[26] K. Fukunaga, *Introduction to Statistical Pattern Recognition*. Elsevier, 2013.

[27] R. Hyde, P. Angelov, and A. R. MacKenzie, "Fully online clustering of evolving data streams into arbitrarily shaped clusters," *Information Sciences*, vol. 382, pp. 96–114, 2017.

[28] N. Langrené and X. Warin, "Fast and stable multivariate kernel density estimation by fast sum updating," *Journal of Computational and Graphical Statistics*, vol. 28, no. 3, pp. 596–608, 2019.

[29] M. P. Wand and M. C. Jones, *Kernel smoothing*. CRC press, 1994.

[30] W. Feller, *An Introduction to Probability Theory and its Applications, vol 2*. John Wiley & Sons, 2008.

[31] J. Jaffe, "Bottleneck flow control," *IEEE Trans. on Communications*, vol. 29, no. 7, pp. 954–962, 1981.