Saarland University

Faculty of Mathematics and Computer Science

Department of Computer Science

Master Thesis

Powering Accurate Aggregate Analysis with Representative Distributed Tracing

submitted by

Reyhaneh Karimipour

submitted

November 2022

Reviewers

Dr. Jonathan Mace

Prof. Dr. Deepak Garg

Abstract

Modern distributed systems make use of distributed tracing to aid in troubleshooting and aggregate analysis. Tracing a request can be computationally expensive, so to reduce tracing overheads, only a subset of requests will be traced. Tracing frameworks typically decide at random which requests to trace and thus a common assumption is that the traced requests are *representative* of the system's workload as a whole. However, in practice, this is not the case. Modern systems have many entry points, diverse execution paths, and combine multiple sampling policies. Overall, the set of requests that are traced is a combination of many different sampling decisions from many places. Although individual sampling decisions may be random, the overall composition of these decisions is *not* uniformly random. This discrepancy affects aggregate analysis, a common use case of distributed tracing. Aggregate queries across the recorded trace datasets assume a uniform random sample of traces, and consequently as we demonstrate they can produce arbitrarily incorrect results due to the non-uniformity of collected traces in practice. To address this challenge, we continue sampling traces non-uniformly, but we record additional metadata about sampling probabilities when making sampling decisions. Using this metadata, we can reduce or eliminate the bias in the sampled traces in order to get more accurate results for aggregate queries.

Acknowledgements

None of this work would have been possible without the constant support, guidance and feedback of my advisor Dr. Jonathan Mace. I am endlessly thankful for him being an excellent teacher, mentor, and advisor. I am proud of and grateful for my time working with him. Moreover, I would like to express my appreciation to Dr. Deepak Garg for agreeing to review this thesis. Finally, I would like to thank my friends for their continued patience and support throughout the whole process.

Contents

Chapter 1

Introduction

1.1 Motivation

Cloud and datacenter systems are often implemented as distributed systems composed of loosely coupled microservices. Developers of these systems use distributed tracing to record traces of the end-to-end execution of requests across all system components [1, 2]. Each trace includes events, logging statements, timing and ordering of events, the values of metrics such as latency or resources consumption, key-value attributes such as RPC, API names (e.g. "API": "read"), and other structured data that records the services and machines visited by the request. Overall, a trace forms a Directed Acyclic Graph (DAG) of such event data (Figure 1.1).

Figure 1.1: Trace visualisation

Aggregate Analysis. Distributed tracing has two key use cases. The first use case is troubleshooting of individual anomalous requests: a trace provides a detailed description of what happened during the execution of the request and help finding the root cause of the problem. A developer can use this detailed description to troubleshoot problems. The second use case is aggregate analysis which is the focus of this work. In aggregate analysis, a collection of traces are used to answer aggregate queries across a system's behavior as a whole. For example, a simple aggregate query is "Plot the latency distribution of the "read" API". Aggregate analysis entails querying datasets of traces that were recorded during the execution of requests; we elaborate further.

Queries. Although a trace is a richly detailed directed acyclic graph, the typical perspective for aggregate analysis is sparse tabular datasets [2]. To work with traces in this form, a backend processing stage called *feature extraction* converts a trace into a tabular form. For example, Figure 1.1 illustrates two simplified stylized traces, and Table 1.1 illustrates some features that have been pre-extracted from trace DAGs and stored in tabular form. Spans are circles and squares in Figure 1.1 that are related to each others by arrows that represent calls and transitions between them. One span of a trace corresponds to one row in the table. Columns correspond to key-value tags and annotated metrics (though more complicated processing is also possible [2]). Aggregate analysis of traces is essentially standard database queries over this tabular dataset. A common example is querying latency distributions and percentiles. Figure 1.2 illustrate an aggregate query for the previous example "Plot the latency distribution of the "read" API of service A". It shows the overall latency distribution or CDF of service A. However, there are more complicated queries that we can perform. Table 1.2 outlines the main query operations we cover. Note that joins are not typically supported.

Trace Sampling. The tracing system aims to capture information about as many requests as possible. However, in a system that serves many requests, tracing every request could be expensive and generate too much data. The computational costs include the cost of generating data, sending it over the network, and storing it in the backend. Trace sampling aims to reduce the amount of data that is recorded. It minimizes tracing costs by only capturing

Trace ID	Span ID	Parent ID	Service	API	Latency(ms)
0aec6cbb	9a6e3aa0	-	A	read	22.5
0aec6cbb	d9b01bb6	9a6e3aa0	B	read	6.5
0aec6cbb	0077ff9c	9a6e3aa0	C	write	14
0aec6cbb	67ef21d4	d9b01bb6 0077ff9c	D	write	4
16aa1a3a	3feda8e0	-	A	read	10.5
16aa1a3a	d569d6a3	3feda8e0	B	read	8.5
16aa1a3a	6ce38dec	d569d6a3	C	write	10
16aa1a3a	e2f0e515	d569d6a3	D	read	1

Table 1.1: Trace data

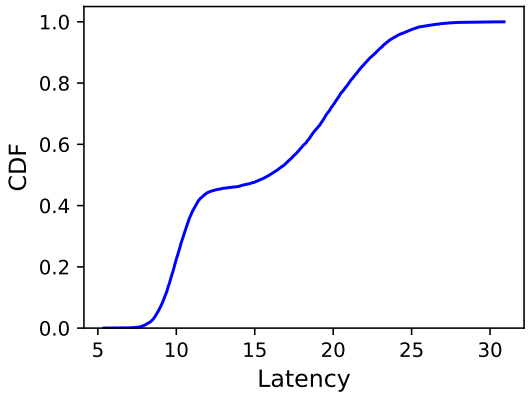


Figure 1.2: Latency distribution of read API for service A

traces for a small subset of requests; if a request is not sampled and therefore not traced, it is not recorded and the computational costs to generate, transfer over the network, centralize, and post-process traces do not exist. By its nature, trace sampling must occur *before* a request executes, and typically it is random. This is a common approach to trace sampling that is called head-based sampling. For each request, a random decision is made at some point (usually when it enters the system) as to record a trace or not. For those requests that were sampled, the tracing system will generate and record the tracing data. Aggregate queries are only executed against the sampled trace data, as trace data does not exist for requests that are not traced. When evaluating aggregate queries, developers assume that the set of randomly sampled requests is representative of all the existing requests. When the system decides to trace a request, *all* components visited by the request will also trace it. The request carries some metadata indicating that the sampling decision said "yes", thereby instructing subsequent services to record the trace.

In practice, few requests are traced – 1% by default in today's open-source frameworks [3], and as low as 0.001% in performance-conscious applications [4, 1, 2]. Thus, aggregate queries are only evaluated against the set of traces that were actually sampled – and *not* every request in the system – because naturally, trace data does not exist for requests that were not traced.

Trace sampling decisions are made uniformly at random, so the set of captured traces is assumed to be a uniform random sample of all requests to the system. Thus, latency distributions such as Figure 1.2 are expected to have low statistical error with respect to the system's actual latency distribution, since they are assumed to be calculated from a random sample of all requests.

Trace Datasets are Non-Uniform. In this thesis we tackle a fundamental limitation of existing trace system designs: are the results of aggregate analysis queries correctly? That is, does Figure 1.2 correctly show the actual latency distribution of our API read? It turns out that the answer is no: all existing tracing systems can produce *arbitrarily incorrect* results to aggregate analysis queries. We outline multiple reasons on why in practice the recorded trace datasets are *not* a uniform random sample of all the requests and consequently, aggregate analysis of sampled trace data yields results with arbitrarily large errors. For aggregate analysis queries, this can lead to query results with arbitrarily large errors. The main reason is that in large distributed systems, there is no central point of decision-making for sampling and no single uniform sampling policy. As a result, the set of captured traces is

Operation	Example
SAMPLE	SAMPLE random(0.01)
WHERE	WHERE API='read'
GROUPBY	GROUPBY API

Table 1.2: Supported query operations

influenced by many sampling decisions from many points in the system, along with heterogeneous code paths and workload fluctuations.

To demonstrate, we ran the DeathStar social network microservice benchmark [5] under a random sampling policy that is widespread in today's tracing systems. Figure 1.3 shows the real latency distribution of a particular service (*post-storage-service*) in the application. We also show the randomly sampled latency distribution for this service that is calculated from the recorded trace dataset in this figure. The measured latency distribution is substantially different from the actual latency distribution of this service. The root cause of this problem is *not* simply statistical variance. This problem arises because the trace dataset is a non-uniform sample of requests to the system while we assumed it is uniformly sampled – as all existing tracing systems do today – and hence we yield arbitrarily incorrect query results. This non-uniformity does not occur during data processing or querying, but during the *sampling* stage of the tracing pipeline, prior to any data being recorded. The application has several different entry points and code paths, and consequently, different requests are sampled with different sampling rates depending on their request types and the services they visit. We elaborate on this in Chapter 3.

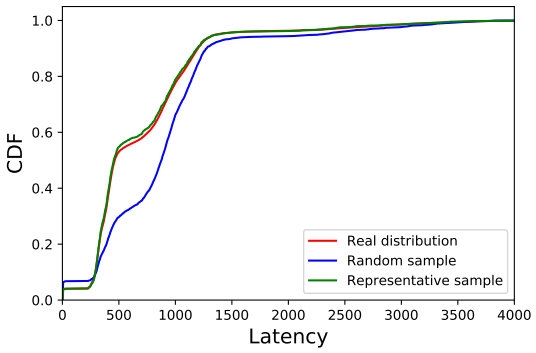


Figure 1.3: Actual latency distribution of a service from the DeathStar benchmark. Existing trace sampling regimes yield a non-uniform dataset and thus query results can have arbitrary errors.

1.2 Contribution

In the rest of this thesis, we examine the root cause of this problem and describe our preliminary approach to correcting this discrepancy. We propose a new consistent sampling approach, along with query-rewriting and dataset sub-sampling techniques to eliminate the bias in the sampled trace dataset. Figure 1.3 illustrates the latency distribution for the sampled data using our *representative sampling* approach after reducing the bias.

1.3 Thesis Overview

Chapter 2 provides necessary backgrounds on the topics of distributed tracing, trace sampling, and bloom filters. In that chapter, we also elaborate on some prior and related works and contributions to this thesis.

Chapter 3 motivates the problem of trace sampling further and provides some micro-benchmark results as well as some brief experiments on DeathStar microservices benchmark.

Chapter 4 explains the key ideas and design of our proposed trace sampling approach followed by an optimisation technique that is integrated with this project.

Chapter 5 elaborates the implementation of our sampling technique.

Chapter 6 reports the results of evaluating our work using different workloaads that are generated from DeathStar Microservices benchmaark [5] as well as Alibaba trace dataset [6].

Chapter 7 concludes our work and its final results and contribution.

Chapter 2

Background and Related Work

In this chapter, we explain some background concepts as well as related work that impact this work and are required to understand this thesis. Here is an outline of this chapter.

- First, we give background on distributed systems and microservices architecture which is a popular approach to building modern distributed systems.
- Next, we give necessary background on distributed tracing which is a tool for recording data during the execution of requests in a distributed system. We explain how tracing systems record this data and how the tracing backend performs feature extraction. We follow by explaining different sampling mechanisms implemented in today's tracing systems.
- We also elaborate on aggregate analysis as a use case of distributed tracing and the types of aggregate queries that we would support in this thesis.

2.1 Distributed systems

Cloud and datacenter systems are usually implemented as distributed systems composed of loosely coupled microservices. These systems are large and complex and they consist of many application services and tiers deployed across many machines and owned by multiple developers [1]. These machines could also be located on different networks and physical locations. Recent work on Alibaba cluster trace microservices reports an analysis on 20000+ microservices [6].

We review some properties of distributed systems.

- In a distributed system, each service performs a specific task; for example, while one service is client service and the point of interaction with a user, another service reads records of the database. These services could be located on different physical machines or networks.
- Requests enter a distributed system in many different ways and from different entry point services.
- Then, a request visits multiple services and during its execution invokes different services. There are multiple request types in the system and each request could follow a different path of services based on its type. For example, a request type might need to read a record in the database while the other one wants to modify it and enter a new record or change an existing one. These two request types follow different paths of services and different API calls to do their jobs.
- Requests can execute sequentially or concurrently. It depends on the service that they visit to allow concurrent executions.
- Different developers are responsible for different components of the system, and often developers do not have a coherent global view of the entire end-to-end application; instead, they only have a myopic local view of the part of the system that they are responsible for.
- Request rate as well as the overall workload in a distributed system can vary over time. Therefore, developers could see a variable request rates for their service over time.

- Distributed systems are not always fixed and they can change dynamically over time. A developer might have different views of the distributed system at different points in time.

2.2 Distributed tracing

Purpose of distributed tracing. There are many problems that can occur in a distributed system and distributed tracing can address them. Aggregate analysis, troubleshooting, and monitoring are key use cases of distributed tracing.

- One of the possible problems in a distributed system is correctness and timing. For example, developers might notice a high delay in a function that their service performs and they want to find its root cause. Another example is when developers want to diagnose requests that have high latency.
- To address such problems, it is important for the distributed tracing framework to monitor the end-to-end execution of requests [7]; because each service is only able to monitor its own local data while problems such as high latency are better diagnosed in an end-to-end request as a whole.
- Distributed tracing helps with troubleshooting end-to-end requests in a distributed system. The data collected by the distributed tracing framework could be used for finding uncommon and outlier requests, the root cause of anomalies in the system, and understanding the system's behavior in general [1, 8].

Distributed Tracing. A distributed tracing framework records the end-to-end journey of a request through the system during its execution. This process is called **tracing** and the data that is collected during the execution of aa request through tracing is called a **trace**. Developers need to instrument their system with a distributed tracing library to be able to benefit from distributed tracing. It means that they need to add the tracing library and some lines of code to the code base of the distributed system. Then, distributed tracing library collects data from different components of the system during the execution of requests to log data about services that were visited, APIs that were called, the time spent on each service, and functions that were performed. We elaborate on tracing procedure further, but first we explain what a trace is in more details:

- **Trace.** The rich detailed log of the execution of a request is called a trace. A trace consists of single or multiple components that are called **spans**.
- **Span.** A span represents a timed block of work that is a piece of the request workflow. It shows a function and some computation at a service that was visited by the requests. As a request traverses through multiple services, the spans of the trace of that request represent the visited services and computations inside them [7].

Trace Illustration. Figure 2.1 illustrates a trace. The x-axis shows time and each bar represents a span. Normally, there are relations between different spans which can be represented as parent-child relationships. When a service invokes another service, a span child is created. The edges in Figure 2.1 represent calls between parent and child spans. Spans can also be concurrent. Eventually, spans and their causal relationships form a trace. A trace is a directed acyclic graph where the nodes of the graph represent spans and edges represent calls and parent-child relations between them. Figure 2.2 illustrates two traces and their equivalent DAGs.

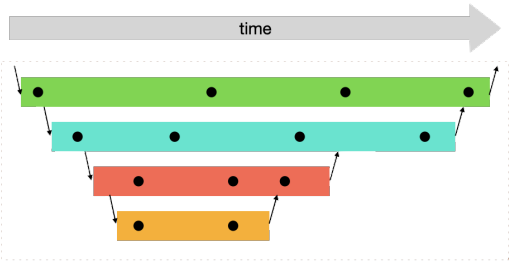


Figure 2.1: A trace, its spans and events as well as calls between them are depicted here.

Trace data. Every trace is assigned with a unique id so that its components could be identified throughout the system. The tracing system assigns a unique and usually randomly generated id to each request at the beginning of recording it. Each request carries the unique id as metadata throughout the system and when traversing through each service that it visits. Spans inside a single trace must share the same trace id to provide consistency. Developers of different components of the system need to cooperate to coherently propagate this unique trace id with the

Figure 2.2: Trace visualisation: A trace and its spans (right) can be depicted in the form of a directed acyclic graph (left).

request throughout the system to capture its end-to-end journey. Trace id is used to identify the spans and calls made by the same request and to make all these parts related together. In a trace some data about the execution of the request is recorded. This pertinent data inside a trace includes logging statements (events), but also structured data (spans).

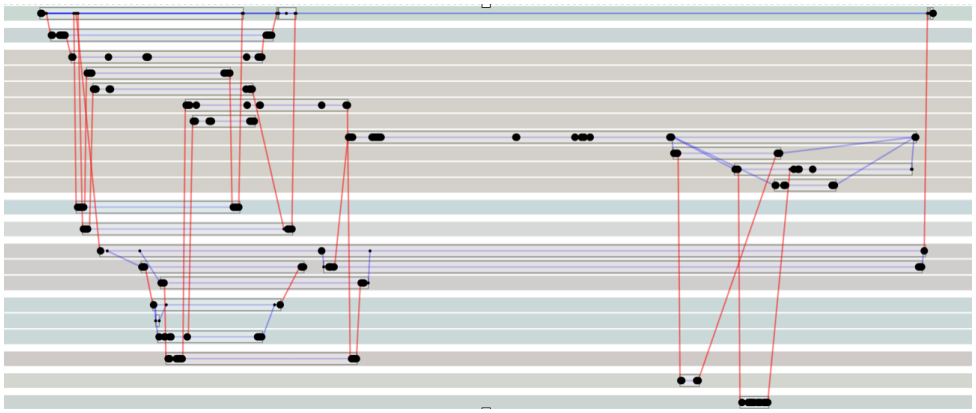
- **Spans.** represent a block of processing data. A span has a specific start and end time. A span has an identifier as well as a name to represent the function that it performs. It also records the name of its corresponding service and API or function that was executed by that span. For example, the name "UserService:Login" for a span represents the Login function being executed at the UserService. Spans have parent-child relationships and they always carry the name of their parent span if they exist; i.e. if a span is the root span, it does not have a parent. Spans of the same request share the same trace id that makes them identifiable to form a full trace. A span also records the name of the machines visited by the request, the values of metrics such as latency or resources consumed, and key-value tags such as the API name (e.g. "API":"read") and other attributes. The spans carry metadata about the request as a tag or other fields of data.
- **Logging Statements.** or **events** log information on the path of a request when it is passing through different components inside the application. There are causal relationships between events. A trace consists of events and the causal relationships between them. An event records trace id, span id, parent-child span relationship, and the name of the host machine and service. Logging statements can also carry key-value tags similar to spans and some numeric metrics.

Real Trace Representation. Different tracing systems choose different kinds of trace representation. In a system like Dapper [1] or Jaeger [9], each trace includes single or multiple spans. In some other systems, a trace is represented by events instead of spans. Canopy [2] and X-Trace [8] model traces in terms of events and their relations. In real distributed systems, traces can get very complex and consist of many spans and events. Figure 2.3 illustrates real traces captured by XTrace [8] and Jaeger [3] tracing systems.

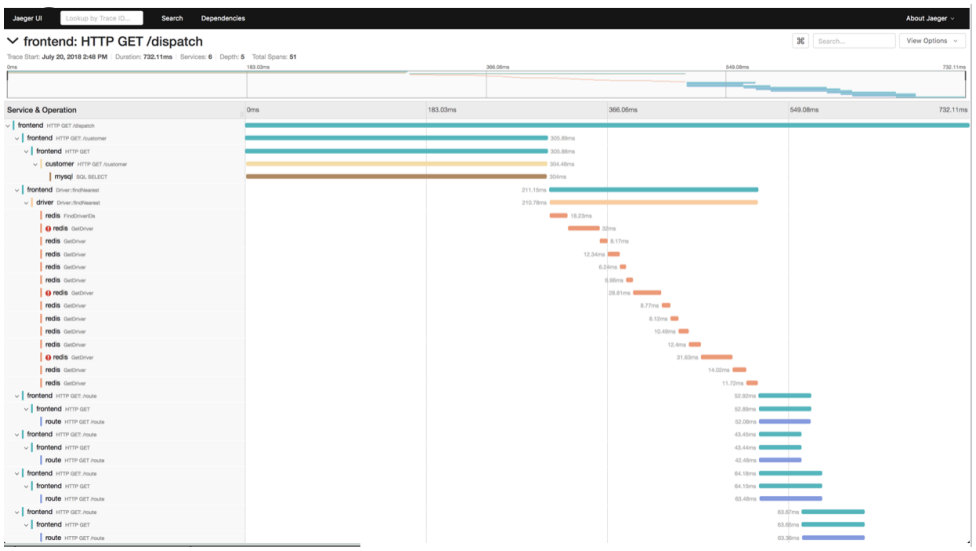
2.3 Tracing backend and feature extraction

Overview of how tracing systems work. Figure 2.4 illustrates an instrumented system with a tracing library. It shows how the tracing library records traces by collecting data through the execution of a request in different microservices. Eventually, the collected traces are sent to the tracing backend. We elaborate on backend processing later. Here, we explain an outline of the tracing procedure in an instrumented distributed system.

1. We instrument a distributed system using tracing libraries to be able to capture traces of requests with the help of the tracing system.
2. A request enters the system and the tracing system aims to record its data. First, the tracing system assigns a unique trace id that will be propagated through the system with this request.
3. As the request visits different services, some data such as the trace id, span id, start/end time of the span, and name of service is recorded by the tracing API.
4. Tracing API captures information about a request that will be recorded by the tracing library. This information includes the trace id and details about a span (id, service, timestamps, etc.) as well as key-value tags.
5. The tracing library sends the data collected from different components through the network to the tracing backend.



(a) A trace visualisation in XTrace showing events and transitions between them.



(b) A trace visualisation in Jaeger showing spans and transitions between them.

Figure 2.3: Illustration of traces in two different tracing systems: XTrace (top) and Jaeger (bottom).

6. The tracing backend processes the trace data and extracts features from the traces. In the tracing backend, trace ids are used to collect data about a single request among all the data that is captured and sent to the tracing backend.
7. Extracted trace id and trace features are stored in tabular database format in the tracing storage.

Tracing Backend. Requests traverse many different services in a distributed system where data about different parts of their executions are recorded by different services. To be able to construct an end-to-end trace of the request, we need to have the recorded segments of the request from every service and connect the pieces to each other. The tracing backend is responsible for collecting and processing this data into an end-to-end trace. After the tracing system generates trace data, the data from each service is sent to the tracing backend over the network. Trace data must be stored on storage in order for the developers to be able to access and process it. The tracing backend is responsible for processing and storing the trace data. This stage of tracing is necessary for two reasons:

- **Raw trace data is difficult to query.** Developers often use tracing to collect trace data for the purpose of aggregate analysis. However, querying raw trace data would be very slow. Prior work generates and processes 1.3 billion traces per day that is not feasible to analyze in an interactive time scale [2]. Therefore, the tracing backend needs to process the raw trace data in a format that is easier for evaluating aggregate queries.
- **Cross component queries.** Microservices can also measure some metrics such as latency; however, to measure these metrics, we need information from different components. For example, the latency of a service is determined by knowing its latency and its child services. Also, it is not possible to perform cross-component analysis without the tracing backend. For example: if we want to calculate the mean latency value of a service

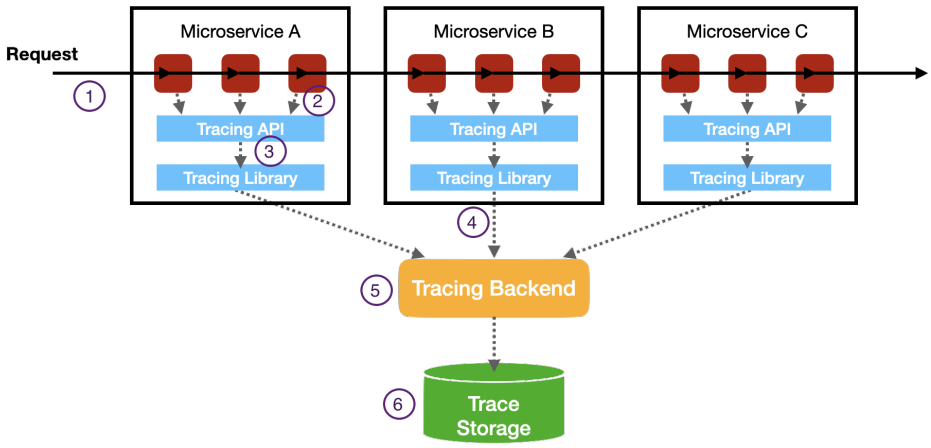


Figure 2.4: Tracing system records the journey of a requests through components of the system and send it to the tracing backend and finally store the data in databases.

and do a GROUPBY API, we need to collect data from other components that made different API calls to this service and calculate the latency of different APIS.

Feature extraction. The tracing backend performs feature extraction to extract information and features from the traces. These features include information about services that a trace visited, host machines, API calls, latency values, timestamps, spans of a request, any key-value tags, etc. that can be extracted from the recorded trace data. This information together with trace ids is stored in tabular format in a time-series database storage at the tracing backend. Figure 2.5 shows two traces in DAG format and Table 2.1 show the extracted features about these traces. Each row of the database represents a span of a trace. It includes trace id, span ids, span's parent id (if it exists), host service and API, and latency. It could also include a timestamp, host machine name, and any key value tags carried by the respective trace. Analysts (and/or developers/operators) could request some extra processing like service throughput, rate of API calls, latency percentiles, and so on.

Figure 2.5: Trace visualisation

2.4 Aggregate analysis

Supported aggregate queries. The focus of this work is distributed tracing for the purpose of aggregate analysis. Developers or operators of different services like to perform queries on requests and their extracted features [10]. These queries can be performed by different operators. It could be analysts who study the system, or developers evaluating their services. These aggregate queries are evaluated against the collected trace data and their extracted features in the tracing backend. Analysts assume that traces are recordings of all the requests; meaning they are either recordings of the complete set of requests or a uniform representative sample of all the requests. Therefore, they think that they are able to query across all recorded traces to understand and analyze the system's behavior as a whole. The aggregate queries fall into two categories:

- **Queries across one component.** These queries include questions like "plot latency distribution of service A", or "calculate the 95% latency of API read of service B". The developer needs information from one service or API to evaluate the query to measure some features of their service, analyze its performance and get statistics.

Trace ID	Span ID	Parent ID	Service	API	Latency(ms)
0aec6cbb	9a6e3aa0	-	A	write	169.5
0aec6cbb	d9b01bb6	9a6e3aa0	B	write	147
0aec6cbb	0077ff9c	d9b01bb6	C	read	24
0aec6cbb	67ef21d4	0077ff9c 0077ff9c	D	read	14
0aec6cbb	3ad6b001	d9b01bb6 0077ff9c	E	write	58
16aa1a3a	3feda8e0	-	B	read	31
16aa1a3a	d569d6a3	3feda8e0	E	read	8.5
16aa1a3a	6ce38dec	3feda8e0	C	read	13.5
16aa1a3a	f438aa94	6ce38dec	A	read	1.5
16aa1a3a	e2f0e515	6ce38dec	D	read	1

Table 2.1: Trace data

- **Cross component queries.** This type of analysis includes information about other services or requests that have relevant information to the service that the operator owns. For example, the operator of a service can query "callers of my service" or "average latency of my service grouped by API", which require information from other components (services/APIs) as well as the analyst's own service.

Query Evaluation

Analysts evaluate aggregate queries against the trace data that is extracted into a tabular format. The trace data is typically stored in a time series database. Storing trace data in a time series database will allow measuring numerous metrics whereas using a relational database has limitations [11, 12]. For this reason, prior work on performance monitoring stores data in a time series graph database [13].

Using a time series database allows us to perform operations on data such as filter, group by, and aggregation. Each of these operations could be performed based on some defined key-value tags or different data dimensions such as time. However, we do not support joins.

In this work, we allow users to define aggregate queries that address requests of a particular service, API, timestamp, etc. We explain what happens in the system when an analyst wants to evaluate a query:

1. Filter the database rows by column values in the database based on some metrics that are specified in the aggregate query such as service name/identifier, time interval, etc.
2. Perform group by if it exists in the query.
3. Calculate the query result such as average/percentile latency, latency distribution/CDF, or a list of values like mean latency and parent services.
4. Report the results in lists or graphs or a single value.

2.4.1 Types of aggregate queries

Here are the types of aggregate queries that we support and are common for analysis on a distributed system.

Average latency. Users can query average/mean latency of their service by using *avg(A)*.

Percentile latency. Users can query percentile latency of their service by using *percentile(A, p)*; for example, *percentile(A, 95)* calculates the 95% latency of service A.

Latency distribution. Users can query latency distribution of their service by using *latency(A)* to use it for purposes like plotting the latency distribution.

Group by API. Users can use groupby to group the requests of their service based on API and then run other queries on them; for example, *avg(groupby(A))* calculates the average latency of each API of service A.

Filter by tag. Users can filter their request based on a specific key-value tag to perform other aggregate queries on them. For example, *percentile(filter(host, M), 95)* calculates 95% latency of requests where their host was machine M.

2.4.1.1 Example aggregate queries

We elaborate on some example aggregate queries and explain how they are evaluated. Recall Figure 2.5 and Table 2.1; we explain how the following example queries are evaluated on them.

Plot latency distribution of service A. Service A appears in two different request types and has different latency values based on which API call is made to that service. Therefore, we need to get information from the recorded traces of both request types that are extracted and presented in tabular format in Table 2.1. Then, we filter rows by *service=A* and get latency values.

Calculate the 95% latency of service B group by API. This is similar to the previous example with an additional *group by* stage. After filtering rows by *service=B*, we group the rows by column API and then calculate the 95% latency.

Average throughout/load of service C. By filtering rows of the table using the filter *service=C*, we can calculate the number of spans that made a call to service C in order to determine the load/throughput of service C.

2.5 Trace sampling

Cost of Tracing. The cost of distributed tracing may come from different aspects and stages of tracing. First, there is computational cost on the critical path of a request to generate trace data (steps 1, 2, 3 in Figure 2.4). Second, tracing framework needs to transmit the trace data from the application layer to the tracing backend for analysis as quickly as possible which adds network cost (step 4 in Figure 2.4). Third, storage cost at the tracing backend. Whether the trace data is sent to the tracing backend, stored in the running application, or in a global repository, there is memory cost to temporarily keep the trace data for processing (step 5 in Figure 2.4). Furthermore, there is storage cost in the tracing backend (steps 5, 6 in Figure 2.4).

Reducing the Cost of Tracing. If the tracing framework records, stores, and transfers traces of all the requests to the backend, the cost of tracing would be very high and practically infeasible to afford. Facebook records 1.3 billion traces [2] and Google production generates more than one terabyte of trace data [1] per day which makes it impossible to record and store all of them. Tracing systems need to lower tracing costs by recording fewer traces since it is not possible for them to record traces of all the requests. As a solution, the tracing frameworks collect and process traces only for a small subset of requests. These traces are recordings of full requests which capture their end-to-end executions for those requests that where chosen to be traced.

Solution: sampling. Tracing systems use sampling to determine the subset of requests that they want to record. They make sampling decisions on whether to record a trace of a request or not. This decision often happens at the beginning of the request before it enters the system. The decision is also coherent; meaning that all components respect the sampling decision. Requests carry a flag that specifies whether they are sampled or not and each tracing component checks this flag prior to recording any data. Each component will record data about a request only if the flag says *sampled*. As a result, fewer traces will be recorded and the performance overhead will be reduced. If a request is not sampled, the overhead of tracing it is eliminated from all the tracing components shown in Figure 2.4.

As an example, dapper performs sampling in two steps. First, it samples 1/1024 traces. Then, it re-samples them and reduces the sampled traces size by a factor of 10. Dapper has reported to impose 1.5% throughput and a 16% latency overhead for a web search workload which degrades by sampling 1/1024 traces to 0.05% and 0.2% respectively [1].

Limitation. By using sampling, we do not get a trace of every request. We only record traces for a small subset of requests that were sampled. When developers access trace data and want to analyze them, they have the trace data for only a limited subset of requests and hence less data to work with compared to the no-sampling approach. This could reduce the accuracy and increase the error in their analysis, but that is a trade-off between accuracy and minimizing tracing costs.

2.5.1 Different sampling approaches

Tracing systems make sampling decisions in different places during a request. They either do it at the beginning or at the end of a request. Figure 2.6 illustrates these two approaches We will explain them in more details in the next section. We will also discuss the limitations of both approaches and investigate new sampling techniques.

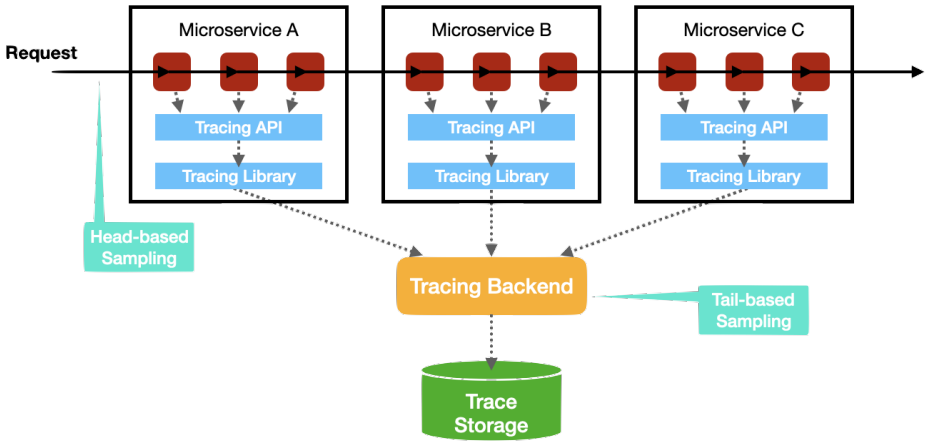


Figure 2.6: Head-based sampling makes sampling decisions at the beginning of a request prior to its entry into the system. Tail-based sampling makes sampling decisions at the tracing backend when all data about a request is collected.

2.5.1.1 Head-based sampling

Definition. With head-based sampling, the sampling decisions are made at the beginning of requests and usually uniformly at random. As Figure 2.6 illustrates, when a request enters the distributed system, the tracing system decides whether to record it or not before any more trace data is generated and recorded. If a request is sampled, all of its spans will be traced and recorded and if a request is not sampled, no data about it is generated and no tracing cost for it exists. Therefore, the cost of tracing will be decreased by recording only the sampled subset of requests.

Sampled flag. If a request is sampled, the tracing system adds a *sampled* flag to it which will be propagated with the request throughout the system. When the request visits different components of the system, the *sampled* flag shows them that the request is sampled and needs to be recorded. Alternatively, if it is *not sampled*, no data about it will be recorded. This results in capturing full traces that record the complete end-to-end path of the requests.

Sampling edge-case requests. Head-based sampling makes uniform random sampling decisions which means that it is more likely to record more common requests. However, users of tracing also care about edge-case and uncommon requests which could be the root cause of problems that were monitored in the system. But, at the beginning of a request, there is not enough information available about the requests that would show the request is an outlier and needs to be traced. Head-based sampling approach is sufficient in a system with little diversity in the traces or a system that wants to record common case requests.

Prior work. Early tracing systems like Dapper [1] make head-based sampling decisions that happen at the beginning of the requests and at the point that they enter the system. Recent work like Hindsight [4] enable sampling during the execution of the requests and not only at the beginning. Nevertheless, it allows recording full trace of each request.

2.5.1.2 Tail-based sampling

Definition. In tail-based sampling, sampling decisions are delayed until the execution of a request ends and all of its spans are available. The tracing system records the end-to-end path of all the requests and sends them to the tracing backend. Then, the tracing system makes sampling decisions on whether to store each trace or not. In tail-based sampling, the cost of generating trace data and sending it through the network to the backend exists for

all the requests. At the tracing backend, if a request is sampled, it will be stored in the trace storage and if it is not sampled, it will not be stored and the storage cost for that request does not exist.

More informed but costly sampling decisions. Making sampling decisions when all information about a request is available, results in more comprehensive decisions. If we are analyzing edge-case requests, this approach will be beneficial as we have more data available about the requests and we can trace the more interesting requests. For example, the tracing system can identify and sample requests with high latency. The trace data is stored in the memory for the time that the request is being executed and the trace sampling algorithm observes the whole trace data to decide if it should be sampled. Tail-based sampling allows the tracing system to record outlier and uncommon requests. However, as a trade-off, the cost of trace data generation and network exists for all the requests. In systems that use tail-based sampling, the bottleneck is normally storage and the tracing system must prevent the tracing backend from overloading.

Prior work. Sifter [14] is a tail-based sampler, mainly focused on capturing uncommon and outlier requests. Sifter uses neural networks to make unsupervised sampling decisions by biasing the sampling decisions towards outlier traces. Tail-based samplers like sifter and hindsight [4] aim to identify and record outliers. They tackle the problem of troubleshooting and edge-case analysis. This is an orthogonal use case to ours and actually makes the sampling result non-uniform. Biasing sampling decisions towards a set of traces results in non-uniformly sampled data.

Variants of tail-based sampling. Recent works suggest variants of tail-based sampling such as delaying sampling until half-way through the request instead of waiting for it to complete and record too much data.

As an example, developers of prior work, Jaeger, would like to delay the sampling decision only until more data about the request is available. Most of the Jaeger clients make sampling decisions at the beginning of the root span and only support re-sampling in case operation name changes. But users propose re-sampling due to possible changes in spans or API calls. Also they suggest sharing the sampling rate of a service so that it is also possible to make the sampling decision at a child span. This also helps prevent recording partial and incomplete traces especially if the sampling rate changes but does not propagate [15, 16]. Recent work like hindsight [4] addresses this problem.

2.5.1.3 Head-based sampling vs. tail-based sampling

Informed decision. A drawback of head-based sampling is its inability to make sampling decisions with respect to the system's behavior. Because the decision is made before observing some anomaly that might happen later in the trace. This problem does not exist in tail-based sampling as we make the sampling decision after observing the whole trace.

Trade-offs. There is also a trade-off between the computational and network costs and how useful the traces are. Head-based sampling is usually done within the application using built-in libraries. While in tail-based sampling, the sampling decision is made after the trace data has been generated and sent over the network to the tracing backend. This results in extra network overhead and also extra computational costs on the backend to receive trace data and process them.

2.5.1.4 Allowing to turn sampling on and off

Sampling decisions are usually made at the beginning of the requests and all the following components respect this sampling decision; meaning that if a sampling decision says "yes" all components that host this request will have to record it. However, in practice, some special cases could happen that contradicts with the nature of capturing the end-to-end path of a request. These cases are challenging and we address them in this thesis, but they are not our main problem.

- **Turning tracing off.** A downstream service might turn off sampling due to resource limitation when it cannot afford to trace more than a certain number of requests. For example: service C in Figure 2.5 might turn off tracing which results in capturing incomplete traces. A similar issue is also raised by Jaeger's developers that if a downstream has low throughput, sampler might fail to capture enough traces from it and hence not give a full view on the requests [17].
- **Turning tracing on.** An upstream service might decide not to trace a request (for example: due to tracing overhead or its sampling decision says "no") but a downstream service with higher tracing capacity or tolerance to overhead, decides to turn tracing on. In this case, it will capture partial traces as well. However, recent

work [4] offers a mechanism to enable downstream services to turn tracing on and still capture full traces of requests.

2.5.2 Sampling policies

Definition. Trace sampling is configurable in many different ways. For example, a sampler could be configured to perform random sampling or rate-limiting sampling with different sampling rates. The type of sampling along with the rate defined with it is called a sampling policy.

Many sampling policies exist. A sampling policy defines details about how to perform sampling on requests that belong to a service or any component of the distributed system. Distributed systems consist of many components that could be on different networks and owned by different developers. Developers are responsible for SLOs and system overhead, hence, they are allowed to define sampling policies based on the capacity and resource limitations of the component they own and the analysis they want to perform. In practice, each developer can define their own sampling policy independent of others. This ensures that each developer will get some traces of requests from their service. The sampling policy gets evaluated when requests visit their service. Therefore, there exist several unique sampling policies for different components of a distributed system. As the nature of distributed tracing is cross-component, one request might get evaluated in different places and by multiple sampling policies during its end-to-end execution. Tracing systems sample and record traces based on the defined sampling policies. The policies that get evaluated depend on the entry points for requests and services that requests visit.

Trace sampling is configurable. There are multiple ways of performing sampling; for example, different sampling rates or different places where the sampling decision can be made. The owners and developers of each service can define their own sampling policy or modify the existing policies. This issue has been raised in prior work on Jaeger to allow adaptive sampling [3]. Jaeger makes head-based sampling decisions. Developers of Jaeger would like to specify different sampling policies in a configuration file to allow sampling decisions that are made on a per-operation basis for spans as well as automatically updating changes in sampling policies [18, 19]. A similar issue is raised for tail-based sampling. Users of OpenTelemetry suggest extending tail-based sampling policies to allow traces to match complicated attributes instead of sampling a lot of traces with a default global sampling rate that may result in missing interesting traces. The solution is to provide more detailed sampling policies [20].

What is included in a sampling policy? Listing 2.1 shows an example sampling policy from a configuration file defined in Jaeger [3]. The sampling policy explains the sampler's type, rate, service, and request type that it is defined for. Developers of Jaeger also mentioned the need for custom fields in sampling policies or tag-based sampling [21] as well as different sampling strategies for different deployment regions and test vs. production [22, 23]. Developers of another prior work OpenTelemetry suggest defining a sampling probability per sampling policy, setting exclusion policies, matching all data in the span by providing some span properties such as service and operation name, and being able to disable any sampling policy [24].

```
{
    {
        "service": "A",
        "type": "probabilistic",
        "param": 0.1,
        "access_level": "developer",
        "duration": 60,
        "operation_policies": [
            {
                "operation": "op1",
                "type": "probabilistic",
                "param": 0.3,
                "access_level": "developer",
                "duration": 60
            }
        ]
    }
}
```

Listing 2.1: A simple sampling policy

Types of sampling policies

There are different types of sampling policies defined by developers which could follow different sampling strategies with their desired sampling rate.

- **Probabilistic sampling:** a fixed sampling probability is defined that indicates the percentage of requests that will be sampled. This is equivalent to tossing a coin with a defined probability. For example, a probabilistic sampler with a sampling probability of 0.1 samples 10% of the requests.
- **Rate-limiting sampling:** a fixed target rate is defined to sample a specific number of requests per time unit. For example, a rate of 10 requests per second could be defined for a rate-limiting sampler which means every second an average of 10 requests are sampled. Rate-limiting samplers could aim for a maximum or minimum rate and are usually implemented using a token or leaky bucket.
- **Combination of probabilistic and rate-limiting:** A sampling policy can define a combination of a probabilistic sampler with a sampling probability and a max/min rate-limiting sampler.
- **Stratified sampling:** Sampling policies could be stratified by request features. For example, request type or API. This kind of sampler uses tags that a request carries to identify its request type.

Sampling in practice

Trace sampling is implemented and used in today's open-source tracing systems. In practice, few requests are traced to reduce the tracing overhead; the number of sampled requests could be about 1% of all the requests by default in today's open-source tracing frameworks [3, 25, 26] or as low as 0.001% in performance-conscious applications [4, 1, 2].

Most of today's tracing frameworks use head-based sampling by default [1, 3, 27] but others offer sampling halfway through a request [4] or at the end of the execution of a request [14] so that it is possible to capture a full request. Developers could take this approach due to the low throughput of their service or if they aim to capture uncommon requests or a certain request type that causes a specific behavior.

Example: Jaeger's sampling. Jaeger is a tracing system that allows making sampling decisions throughout the system by applying multiple sampling policies. It enables users and developers to define sampling policies for different services and APIs. Users could also define sampling policies with a combination of random probabilistic and rate-limiting samplers. This means that different request types could get sampled at different services and with different sampling rates. Jaeger often samples non-uniform trace datasets. However, sampling policies are not necessarily the cause of this problem, but the way they are enforced in the system and are applied independently may lead sampled traces to being non-uniform.

Jaeger client library provides head-based sampling. When a new request arrives in the system, which does not contain any tracing information, Jaeger starts a new trace with a random ID and makes a sampling decision which will be propagated through the system with the request. With this approach, if a trace is sampled, all of its spans will be recorded and stored in the tracing backend as well.

Jaeger offers four types of sampling by specifying the **sampler.type** in the client sampling configuration.

- If the sampler's type is *const*, the Jaeger sampler either samples all the traces (when the **sampler.param** is *one*) or none of them (when it is *zero*).
- If the sampler's type is *probabilistic*, the sampler makes a random sampling decision for each trace with regard to the **sampler.param**.
- If the sampler's type is *rate-limiting*, the sampler makes sampling decisions with a constant rate with regard to the **sampler.param**.
- If the sampler's type is *remote*, the sampler makes sampling decisions according to the sampling strategy of the current operation.

2.5.3 Summary

Distributed systems are large and complex. There are multiple entry points to a distributed system for requests and they traverse different services during their execution. Tracing every request would be computationally expensive, therefore, we use trace sampling to minimize tracing costs by only tracing a small subset of requests. Services in a distributed system are run by different operators and developers. They can define their own sampling policies

independently of each other and they do not need to make coordination between them. As a request traverses through different services, it could get evaluated by different sampling policies during its execution. Finally, traces are sent to the tracing backend and converted to tabular trace datasets for aggregate analysis.

There are many developers that define sampling policies to minimize tracing costs. Developers assume that the set of recorded traces is representative of all the requests. For example, they assume that the 95% latency of recorded traces is the same as the 95% latency of all the requests in the system. However, this assumption is not correct. In the next chapter, we elaborate on the reasons why this is not the case.

Chapter 3

Challenges

In this chapter, we motivate the problem and explain the importance of the problem.

Intuition. One of the use cases of distributed tracing is aggregate analysis. Developers evaluate aggregate queries against the recorded traces of requests. As we explained in Chapter 2, trace sampling is introduced to minimize tracing costs by only recording traces for a small subset of requests. Developers assume that aggregate analysis of recorded traces is equivalent to aggregate analysis of all the requests. However, this assumption is not always correct. In an ideal setting, there exists a single point in the system which has visibility of all requests and all sampling decisions are made there. In this scenario, sampled requests are a random uniform sample of all requests. However, in a large real distributed system, there does not exist such a central point of control and visibility and sampling decisions can be made in different places and based on different sampling policies. This results in bias in the sampled data. In principle, aggregate queries evaluated against a randomly sampled subset of requests should yield the same result as they do (with slightly higher error) against all requests. However, in practice, the set of sampled requests is *not uniformly random*. We demonstrate in this section, the effect of sampling on aggregate analysis and the major factors contributing to this problem and illustrate the problem by presenting simple examples.

3.1 Effect of sampling on aggregate analysis

In this thesis, we discuss the negative effect of trace sampling on aggregate analysis. As explained before, the aggregate analysis aims to evaluate aggregate queries over all the existing requests in a distributed system. However, as it is not computationally feasible to record traces of all the requests and run aggregate analysis over all of them, we use trace sampling to pick a small subset of requests and only record traces of them for the purpose of aggregate analysis with tolerable overhead. This subset of sampled requests is assumed to be drawn uniformly at random from all the requests and therefore represent the system's behavior as a whole. A system analyst or a developer evaluates the aggregate queries with this assumption, expecting to get a correct result that is within a small margin of error. However, as we discuss in this chapter, this assumption is not always true. In reality, the dataset of sampled traces can be biased and the result of aggregate analysis over it could have arbitrarily large errors.

Recent work on a statistical approach. A recent work [28] identifies a similar problem to the problem we tackle in this thesis. However, they identify the problem that particularly arises from capturing partial traces. They propose a statistical solution to estimate the error in the analysis result so that they can undo the bias. We study a more general problem and try to approach the solution differently.

Example. We illustrate an example here to show how the non-uniformly sampled traces could cause bias in aggregate analysis.

Figure 4.4 illustrates two request types. These two request types enter the system at the same time. If we make sampling decisions for both request types at the two entry points A and D with a single sampling probability(40% in this example), we would get a sampled set of traces that are colored in green. At the analysis phase, we calculate the latency distribution of service B and plot it. Figure 3.3a illustrates the latency distribution of service B.

We repeat the experiment on the same requests but this time, we sample requests at service A with a probability of 20% and at service D with a probability of 60%. Figure 3.2 illustrates all the requests and the sampled requests are colored in green. We again calculate the latency distribution of service B and plot it which is shown in Figure 3.3b.

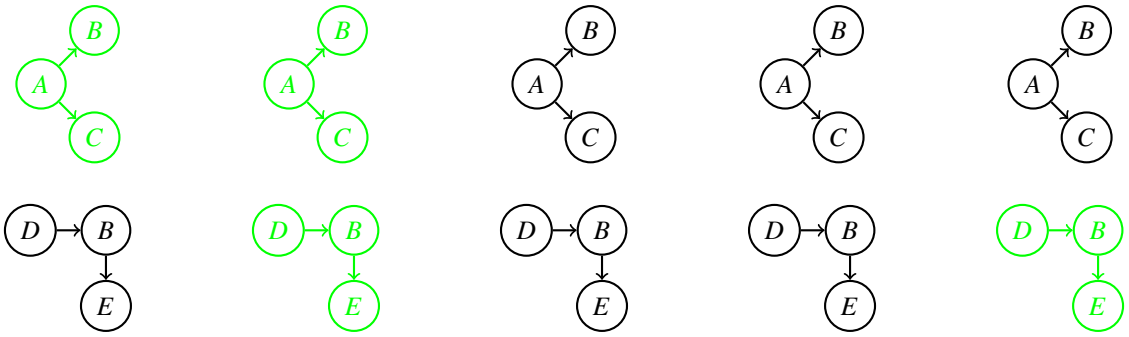


Figure 3.1: Uniformly sampled requests with a single sampling rate for all entry points: 40%. Sampled requests are specified with green color.

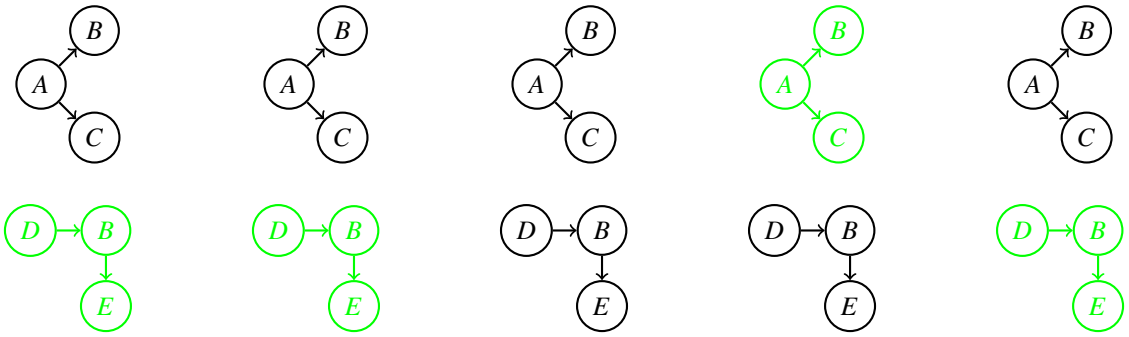
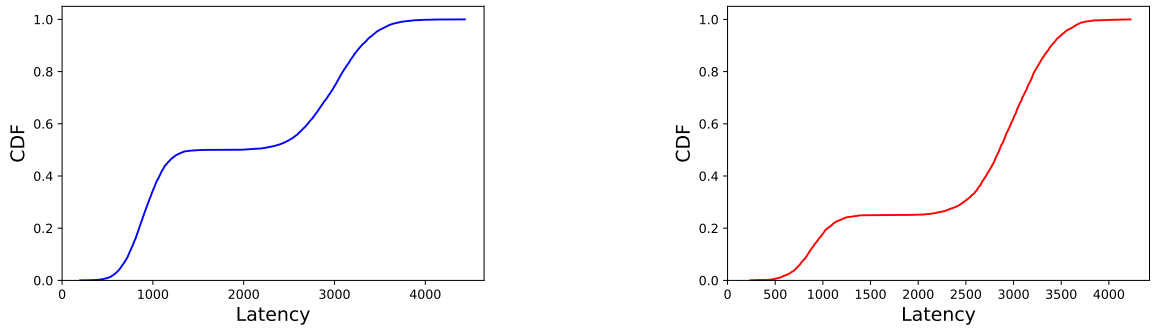


Figure 3.2: Sampled requests with different sampling rates at different entry points: 20% at A and 60% at D. Sampled requests are specified with green color.

By comparing two latency distribution graphs for service B in Figure 3.3, we can observe the large error caused by the bias in the trace data that comes from sampling the requests non-uniformly.



(a) Correct distribution

(b) Biased distribution

Figure 3.3: Latency distribution of service B under two different sampling policies: requests are sampled at all entry points with probability of 40% (left) requests are sampled at services A, D with probabilities of of 20%, 60% (right).

As illustrated in the above example, the set of sampled traces is not always a uniform representation of all the requests which causes large errors in aggregate analysis. We elaborate on the reasons behind this problem in this chapters.

3.2 Expectations vs. Reality

There are certain assumptions that there is a central point of sampling decision making and a unique policy that would sample and record traces of a uniform random sample of requests. Users evaluate aggregate queries against sampled trace dataset instead of all requests given this assumption, expecting to get similar results with low error.

As we demonstrate, there can be arbitrary biases in the sampled trace dataset. In this section, we elaborate further on how and why this bias appears.

3.2.1 Multiple Entry Points

This reason explains how head-based sampling does not sample requests uniformly at random. it also shows an example of traces entering the system from different entry points as well as a simple example and graph of expected vs sampled latency distributions.

Expectation: the system has a single entry point through which all requests flow.

Reality: there are multiple top-level APIs and gateways through which requests can enter the system.

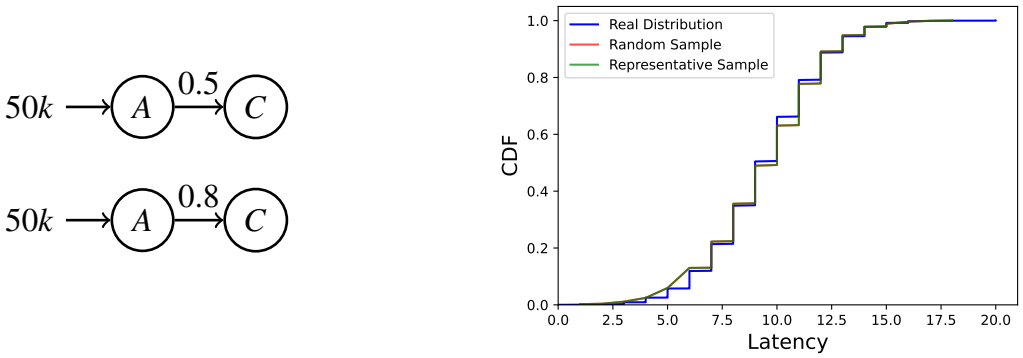
Sampling occurs at the beginning of requests, yet requests can enter the system in many different ways. A front-end API may be replicated to multiple machines; there may be many different front-end APIs, and requests can even originate internally in the system. Consequently, sampling decisions happen at all entry points, in many places in the system. Internally, requests that enter the system from different places may still arrive at the same child or leaf services. When we calculate aggregate statistics about those child services, we incorporate all traces regardless of the entry point.

If things were as expected. If there existed only one entry point where all the requests enter the system from there and get evaluated against sampling policies, we could have had a uniform view of all the requests and we could have successfully sampled a uniform random subset of them.

Figure 3.4a illustrates a system with one entry point A and two request types that call different APIs of service C. These request types enter the system at the same frequency. We define the following sampling policies for these services:

- service A: Probabilistic(0.01)
- service C: Probabilistic(0.1)

We calculate the real and sampled latency distribution of service C and plot them in Figure 3.4b in blue and red respectively. We also plotted the latency distribution of C that shows the result of our representative sampling approach in green. As Figure 3.4b illustrates, all of these distributions are similar with only a small error.



(a) A is the only entry point

(b) Latency distribution of service C

Figure 3.4: In the service setup (left) all the requests enter the system from one entry point (service A) and get evaluated against sampling policies of services A and C. The result of aggregate analysis is shown in the latency distribution of C (right).

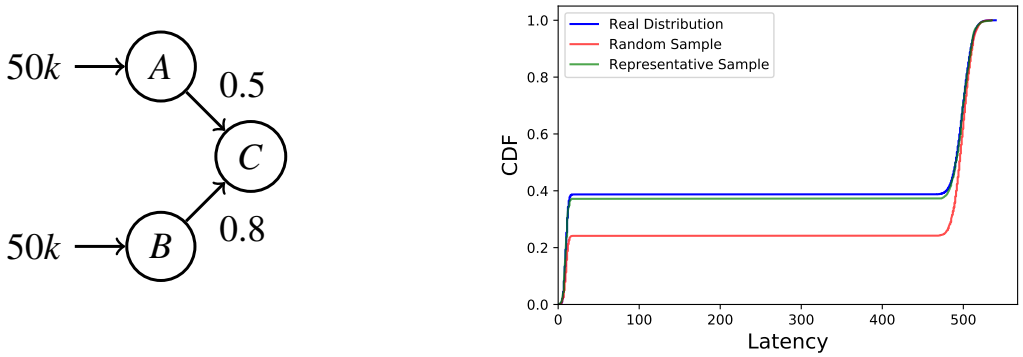
Entry points example in practice. In practice, however, requests enter the system at different locations and different sampling decisions are applied to them that could bias the requests. The sampled requests could be simply biased towards a request type with higher sampling rate or the request type that visits an entry point with higher throughput. This could happen when we have a low sampling rate for entry points with lower throughput and do not record enough traces of their requests.

In Figure 3.5a, requests enter the system through either service A or B and then potentially visit other services. Services A and B could potentially call different APIs of service C (50% and 80% of their API calls are made to

service C respectively) with different average latency values. We define the following sampling policies and discuss the result:

- service A: Probabilistic(0.01)
- service B: Probabilistic(0.1)
- service C: Probabilistic(0.05)

For such sampling policies where requests are sampled at service A with a lower sampling probability than B the sampling probability of the child service (service C) will affect the distribution of set of recorded traces. Figure 3.5b shows the real vs. sampled latency distribution of service C. The red graph indicates the random sampled latency distribution widespread in today's tracing systems and green shows the result of our representative approach that eliminates the bias.



(a) Both B and C are entry points

(b) Latency distribution of service C

Figure 3.5: In the service setup (left) requests enter the system from multiple entry points (services A and B) and get evaluated against different sampling policies. The result of aggregate analysis is shown in the latency distribution of C (right).

Special case: multiple callers. If a service has multiple callers or incoming requests from multiple origins/parent services, the set of its sampled requests are affected by the sampling policies and hence sampling decisions of all of its callers. This is basically the similar issue as in **multiple entry points**. In Figure 3.6, 1000 requests go through service A and then service C and 100 requests go through service B and then service C. So, service C is called by service A 10 times more than it is called by service B. However, if service A has a lower sampling rate than service B, for example if service A has a sampling rate of 0.01 and service B has a sampling rate of 0.1, service C receives 10 requests from services A and B each. These numbers are clearly not representative of the real throughput of these services and result in inaccurate aggregate analysis.

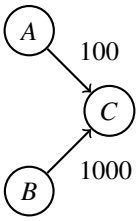


Figure 3.6: A service with multiple callers

3.2.2 Multiple Sampling Policies

This section explains how the result of aggregate analysis is affected by the limitation that services define on their tracing capacity and sampling rates. Example: a service with multiple parent services that each have different sampling policies (similar to the previous example.

Expectation: there exists a single global sampling policy at all services and components that make sampling decisions in a distributed system, e.g. "sample 1% of all requests".

Reality: systems have many components, developers, and operators. The components can have different tolerance or capacity for tracing overheads and operators can define different sampling policies based on their desire for more or fewer traces.

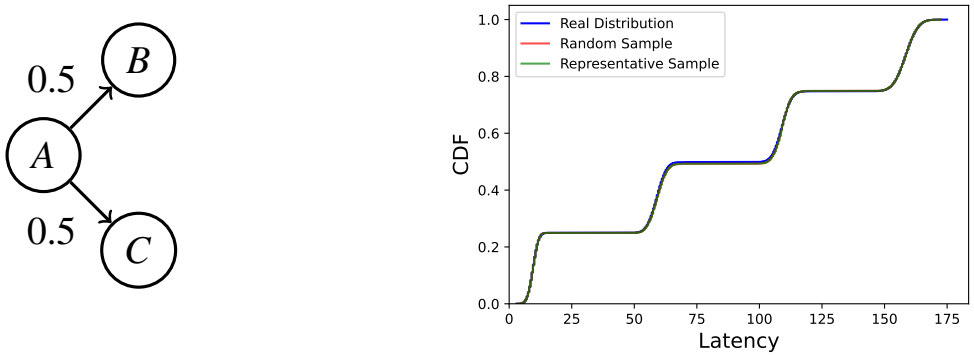
The sampling probability defined for each component is primarily driven by the desire to capture as many useful traces as possible while limiting the overall resource requirements and impact on the running system. However, there are many services and developers involved, who have different objectives and perspectives on what sampling rate is acceptable. Consequently, the sampling policy for one service (e.g. one entry point) may be more or less restrictive than another. If there was a single sampling policy throughout the system, for example, 1%, we would sample a uniform set of requests that correctly represent the system's behavior as a whole. However, as requests are sampled at different places with different sampling policies, the sampled requests could be biased towards the request type with a higher sampling rate.

Further to this reason, existing tracing systems support stratified sampling. When a request arrives at an under-utilized endpoint, the sampling probability may be higher, or there might be a minimum rate of sampled traces, to ensure that enough data about this request type is recorded. This result in seeing more traces from this entry point and hence getting incorrect aggregate query results.

Example: one sampling policy across the system. We elaborate further with an example. Figure 4.1a illustrates the configuration of three services. Requests enter the system through service A. They potentially call services B and C. It is equally likely for the requests to call neither service B nor service C, call only one of them, or call both of them. These requests are also sampled at the three services with the same sampling rate:

- service A: Probabilistic(0.1)
- service B: Probabilistic(0.1)
- service C: Probabilistic(0.1)

Figure 4.1b shows the real latency distribution of service A that is derived from all the requests. The real distribution clearly explains the transition rates between services. We also plot the sampled latency distribution of service A. We observe that the sampled latency distribution, in this case, is very close to the real latency distribution.



(a) Service A calls multiple child services. All services have the same sampling rate

(b) Latency distribution of service A

Figure 3.7: In the service setup (left) requests enter the system through service A and then potentially visit B and/or C. If all services sample requests with the same sampling probability, the sampled latency distribution will be the same as the real distribution (right).

Real example: different sampling policies across microservices. For the next example, recall the configuration depicted in Figure 4.1a. We change the experiment setup so that services A, B, and C make sampling decisions with different sampling rates. The following sampling policies are applied to services in this experiment:

- service A: Probabilistic(0.01)
- service B: Probabilistic(0.1)
- service C: Probabilistic(0.05)

Services B and C have similar throughput. If service C has a lower sampling rate than B, it will sample fewer requests than B. So, there will be more recorded traces that visited service B than those that visited service C, and the sampled latency distribution of service A as shown in Figure 3.8 will be biased towards that request type.

A similar issue can be observed in Figure 3.5. That example depicts a similar problem by defining different sampling policies at entry points and getting biased latency distribution at service C when using the common random sampling approach. The latency distribution is biased towards the request type going through service B that has a higher sampling rate.

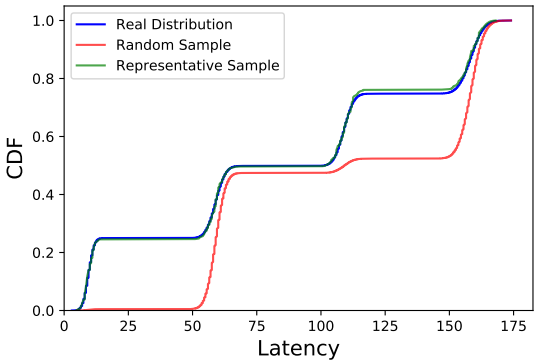
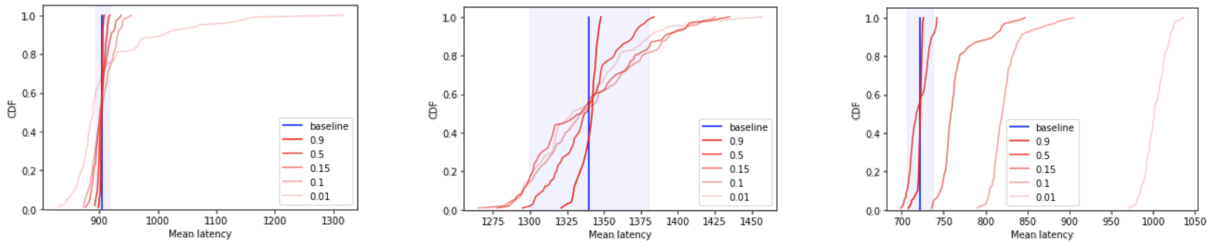


Figure 3.8: In the service setup in Figure 4.1a, if sampling probability of B is higher than C, sampled requests will be biased towards the request type that visits B.

A more advanced example: effect of changing sampling probability on mean latency. In this example, we show the effect of changing sampling probability of a service while keeping the rest of the sampling policies fixed, in a setup similar to Figure 3.6. We pick three DeathStar microservices [5] (*post-storage-service*, *home-timeline-service* and *compose-post-service*) to demonstrate this example. In this DSB workload, *post-storage-service* is called by *home-timeline-service* and *compose-post-service*. We define fixed sampling policies for *home-timeline-service* and *compose-post-service*. We change the sampling policy of *post-storage-service* so that it randomly samples requests with probability of 0.0 to 1.0 and repeat the experiment for each sampling policy. Figure 3.9 demonstrates how the mean latency of each service could change when we increase the sampling rate of the child service *post-storage-service*. In each graph, we plot the mean latency of all the requests of each service, as well as a 95% confidence interval around the baseline mean latency in blue. We repeat the experiment with different sampling policies and plot the distribution of mean latency in red. As we increase the sampling rate of *post-storage-service*, we observe how the mean latency changes in comparison to the baseline and the confidence interval. This result show that not only different sampling policies affect the uniformity of the captured traces, but also can introduce arbitrary large errors in the aggregate query results.



(a) Mean latency distribution of compose-post-service

(b) Mean latency distribution of home-timeline-service

(c) Mean latency distribution of post-storage-service

Figure 3.9: Mean latency distribution of three DeathStar microservices when increasing sampling rate of *post-storage-service*. The sampled mean latency for a sampling probability like 0.1 is not inside the confidence interval (right).

3.2.3 Partial traces

This section explains that sampling decisions might happen in places rather than the beginning of requests and at the entry points and elaborates on reasons behind it. Example: a service that refuses to trace requests or a service with child services with different sampling rates.

Expectation: sampling decisions are made at the very beginning of a request, and thus either an entire trace of a request is captured, or no trace is captured.

Reality: a request may traverse several services before or after a sampling decision is made.

Ideally, a sampling decision occurs at the entry to the system, and all subsequent services respect that sampling decision (by recording data, or not). This means when a new request enters the system, a sampling decision is made and if it is successful the end-to-end request will be traced. This would, ideally, yield complete end-to-end traces. However, in practice, this does not occur, for two reasons.

1. Some intermediate and leaf services may not be willing to trace the volume of requests that are sampled. For example, a highly optimized service such as a KV store may desire a much lower rate of sampled traces than the front-end. Such services are free to disregard sampling decisions and not trace requests.
2. A front-end service or API may choose not to sample a request, but an intermediate or leaf service may want a higher volume of sampled requests. This misses out on data from earlier services.

Although we describe sampling as occurring at entry points in reason 1, in practice sampling can be turned on (or off) at any point during a request. This is primarily driven by practical concerns, at a potential cost of utility. In reality, we can have partial traces by tracing the requests from the sampling point onwards, as well as using a system like Hindsight [4] to retrieve the lost data and trace the complete end-to-end requests.

Example: partial trace by turning tracing off. Figure 3.10 shows a partial trace that only recorded a part of the request. In this example, service C turns sampling off and refuses to capture traces due to resource limitations, and as a result, we cannot obtain the complete end-to-end path of this request. Also, in Figure 3.8, using recent work Hindsight [4], we can capture the end-to-end journey of a request if service B or service C decides to capture a trace that was refused by other services.

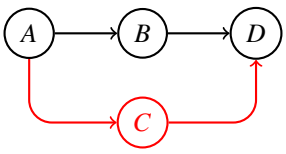


Figure 3.10: Traces are best effort: a service can disable tracing mid-way through a request (right).

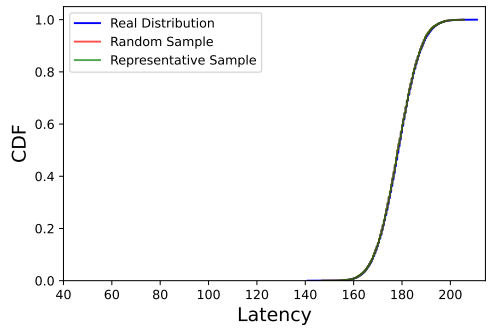
We elaborate more on this example. The mean latency values of services A, B, C, and D in Figure 3.10 are 50ms, 10ms, 100ms, and 30ms respectively. In two scenarios where service C turns sampling on and off, we calculate the latency distribution of service A and plot the result. If service C keeps tracing the requests, the latency distribution of service A would be as illustrated in Figure 3.11a. However, if service C turns sampling off and does not allow tracing requests, any data that would have been generated at service C will be lost and the sampled distribution of service A will be wrong as illustrated in the red graph in Figure 3.11b. We leverage the idea of Hindsight [4] in our sampling approach to be able to retrieve the lost trace data under such circumstances. Figure 3.11b illustrates the latency distribution of service A that is calculated by the sampled data through our proposed reprentative sampling approach that uses Hindsight [4] (green graph), would be correct and the same as the real distribution (blue graph).

3.2.4 Different types of samplers

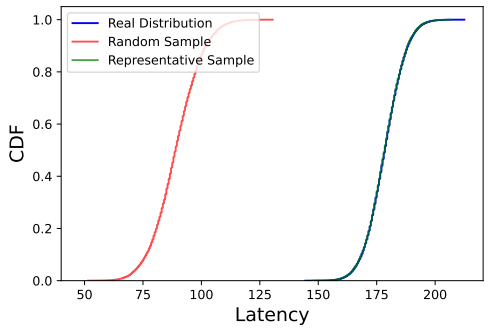
This section explains different types of samplers that we support in this thesis and how they could affect sampled data and aggregate analysis.

Expectation: all requests are sampled uniformly at random.

Reality: there are multiple types of sampling used by different systems. Some services have low throughput and a random sampler cannot capture enough requests from them. For such services, rate-based samplers are used that can guarantee an average rate of requests being sampled. In practice, there are multiple types of sampling used by different systems. For example, probabilistic sampling, rate-limiting sampling, and per-API sampling.



(a) Latency distribution of service A when C turns sampling on.



(b) Latency distribution of service A when C turns sampling off.

Figure 3.11: When service C allows sampling (left), real and sampled latency distributions of service A will be identical or close. When service C turns sampling off (right), trace data from service C will be lost (red graph), but using a system like Hindsight [4], the problem will be solved (blue and green graphs are identical).

Probabilistic sampling makes random sampling decisions with a certain probability. For example, for a random probabilistic sampler with the probability of 0.1 or 10%, an average of 10 out of 100 requests will be sampled.

Rate-limiting sampling makes sampling decisions using a leaky/token bucket and samples requests at a constant rate. For example, with a rate of 10 requests per second, an average of 10 requests per second will be sampled. This is mainly used by services that have low throughput and a probabilistic sampler cannot capture enough requests from them. Rate-limiting samplers can be defined as minimum or maximum rate-limiting.

- **Minimum-rate sampling** guarantees that a minimum number of requests will be traced over a given time interval. For example, with a min-rate of 10 requests per second, a minimum of 10 requests per second will be sampled. For aggregate analysis, the existence of a min-rate sampler can potentially over-represent the sampled request type if used in conjunction with a probabilistic sampler.
- **Maximum-rate sampling** is similar to minimum-rate but imposes an upper limit. This is useful if an over-abundance of traces is unnecessary. Like min-rate, a max-rate sampler will adjust over time based on the observed request rate. For aggregate analysis, the existence of a max-rate sampler can potentially under-represent the sampled request type.

Stratified, **per-API** or **tag-based sampling** makes sampling decisions based on some key-value pairs. Developers often define different sampling policies that only apply to specific APIs or requests that meet specific criteria. Per-API sampling is often used with minimum-rate sampling, to ensure that enough traces are recorded at all different APIs of a service. In general, tag-based sampling can customize sampling decisions for any key-value request attribute [21].

For example, a per-API sampler ensures that 10% of requests going through each API are sampled. Similarly, a tag-based sampler could define a higher sampling rate for the API with lower throughput.

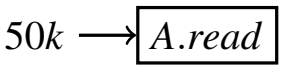
Consider the following example policies:

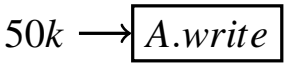
- service A – API read: Probabilistic(0.01)
- service A – API write: Probabilistic(0.1)

Figure 3.12a illustrates a scenario that the above sampling policies are applied. API read of service A has a request rate of 80k request per second and API write has a request rate of 20k request per second. Defining a higher sampling rate for API write, makes sure that enough traces of type write are recorded but it also results in sampled set of requests that is biased towards API write (red graph) in Figure 3.12b.

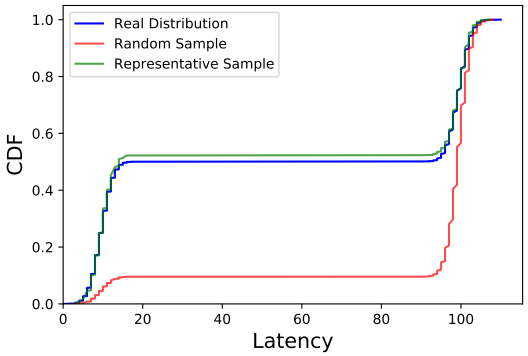
3.2.4.1 Combine different types of samplers: random and rate-limiting

Rate-limiting samplers are usually used in combination with a random sampler to ensure that a minimum rate of rare requests is sampled. However, this combination may result in biased sampled trace data when using the common sampling approach. Assume an arbitrary service B has two APIs *read* and *write* with different latency values l_r and l_w. The latency distribution of service B is illustrated as the expected distribution in Figure 3.13b. As illustrated in





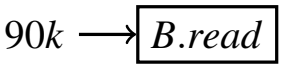
(a) API read of A is called as frequently as API write.

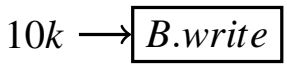


(b) Latency distribution of service A

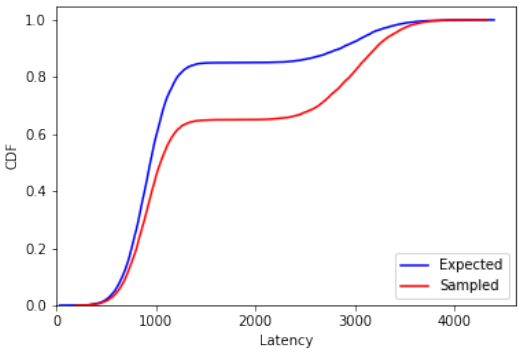
Figure 3.12: In the service setup (left) APIs read and write of service A are equally likely to be called by the requests. If API-based sampling policies are defined that is biased towards write, more requests of type write are sampled (right).

Figure 3.13a, API *write* is rarely called by other services, therefore, we would like to ensure sampling a fixed rate of *write* API calls in addition to randomly sampling all the calls to service B. In this case, sampled latency distribution of service B would look like the sampled distribution in Figure 3.13b which is not representative of the expected distribution. The reason is that we are sampling more *write* requests than *read* requests and the latency distribution is now biased towards *write* requests.





(a) API write of service B is rarely called compared to API read.



(b) Latency distribution of service B

Figure 3.13: In the service setup (left) API read of service A is called 90% of the time by the requests. If we use a random and a rate-limiting sampling policy combination where a maximum rate of API write requests are sampled, the latency distribution of B will be based towards API write (right).

3.2.5 Repeated evaluation of sampling policies

Multiple sampling decisions could be defined for a request type and therefore, the request gets evaluated multiple times.

Expectation: a sampling decision is made exactly once for each request.

Reality: a request can traverse multiple services, each time the sampling policy may be re-evaluated for the request. Alternatively, a request might match the conditions for several samplers configured for per-API or tag-based sampling. This can result in a non-uniform and elevated probability of a request being sampled, based on the number of times a sampling decision is made for a request. Such effect is shown in Figure 3.8.

Summary. In an idealized scenario, there is a single sampling rate that is uniformly applied to all requests at a single entry point, and if a request is sampled, it is traced in its entirety. If it is not sampled, no data about it will be

recorded. In practice, however,

- There are multiple entry points to the system.
- Sampling decisions are made at multiple points.
- There are many different sampling policies.
- Services can be affected by several sampling policies.
- Requests are not necessarily traced in their entirety.

3.3 Conclusion

This section explains the challenges to solving the representative trace sampling problem, meaning why it is difficult to solve it. What should we prevent in our solution and what should we pay attention to.

Evaluate queries. Queries including sampling policies are decoupled from datasets of traces. Sampling policies can affect each other and cannot be enforced into the system's implementation. However, there is an implicit assumption in current tracing systems that each of the sampling policies is independent of the others and can be directly and independently used to sample requests. However, in reality, they are strongly correlated and cannot be considered independently.

Wrong assumption about sampled traces. Similarly, as we explained in §3.2, analysts assume that the set of sampled traces is uniform and represents the system's behavior as a whole. They evaluate their aggregate queries against the biased trace datasets and get incorrect results. They might not even notice that the result of their query is wrong as they do not know anything about the system. They simply believe that they are evaluating their queries across all the requests or a uniform sampled subset of them.

Developers are free to define arbitrary queries. We allow developers to write arbitrary sampling policies and queries based on their own use cases. They are only responsible for defining their query and not applying it to the system. The queries are unknown, defined for a specific component, and independent from the rest of the components. It is our responsibility to translate the queries into a format that the tracing system understands, then apply every sampling policy independently, followed by combining the requests that were sampled by different sampling policies and eventually at the tracing backend evaluate the aggregate queries.

In the rest of this section, we talk about key elements of a solution to solving the trace sampling problem.

3.3.1 Straw Man Solutions

This subsection explains the challenges to finding a solution for the representative trace sampling problem and the straw man solutions to it. Mainly, we do not want to trace extra requests, have coordination between services nor enforce queries directly to services.

We first explain the possible straw man solutions to the trace sampling problem. Then, we explain the key features of our solution.

3.3.1.1 Straw man solution 1: Record all the requests

Assume a system analyst or a developer evaluates an aggregate query against the sampled trace dataset. How can they tell if the result is correct or not? A straw man solution is to trace all the requests and evaluate the query against all the requests to have the ground truth. Does this solution work?

- Advantage: The analyst can compare the result of their analysis with the ground truth as a reference to make sure it is correct.
- Disadvantage: Recording all the requests is expensive and is not practically possible due to resource limitations. This is the reason we introduced trace sampling in the first place.
- The analyst cannot expect the ground truth to be available and has to trust the sampled trace dataset. They will never know whether the result is correct nor how high the error is.

3.3.1.2 Straw man solution 2: Assign weights to sampled traces

We explained why sampled trace datasets could be biased towards some request types and how this phenomenon results in incorrect aggregate analysis of traces. Our next straw man solution to solving this problem is to put weights on the sampled traces in order to revert the bias. When a request gets sampled, the sampler adds its sampling rate as metadata to this request. In the tracing backend, if we have fewer or more traces from one request type, we will apply the weight to eliminate the bias. We explain why this solution does not work.

- The topology of the system is unknown and we cannot assume that it is static nor fixed. It could dynamically change over time. Throughput of services and transition calls between services could change over time as well. If service A calls service B in 100 of requests, we cannot necessarily assume that this is always the case for all other requests.
- Similarly, sampling policies can change or get updated and we cannot rely on the sampling rates that are propagated with the requests.
- Request rates and sampling rates both affect the aggregate result. We do not have prior information about the request rates. The traffic is unpredictable and can change or fluctuate over time. We cannot rely on the request rate at one point in time. We have to update the request rate constantly, which prevents us from any offline analysis.
- A request might get re-evaluated with different sampling policies and eventually get labeled with multiple sampling rates. We do not know which sampling rate to consider or we will have to communicate sampling policies between services and the tracing backend to clarify this. However, this requires a communication channel and coordination between services. This adds extra complexity to the system and we prefer each service to record and sample requests independently.

3.3.2 Key Elements of a Working Solution

In §3.2 we talked about challenges of trace sampling and in this section earlier, we explained why the straw man solutions do not work. In the following subsection, we summarize the key elements of our solution that is robust to those challenges.

- We do not want to record extra traces.
- Coordination between different components of the system should be prevented or minimized.
- Services make their own sampling decisions independent of other services and do not need to do calculations of sampling rates that are applied to their requests.
- Developers evaluate aggregate queries against the sampled traces and expect to get the same results as against all the requests.

In Chapter 4, we address these key elements in our solution and propose a novel approach to solve the representative trace samplling problem and prevent incorrect aggregate query results.

Chapter 4

Design

This chapter explains the design of our samplers and key ideas behind it. All algorithms and implementation details will be discussed in Chapter 5.

Intuition. So far we showed that tracing all the requests is expensive and not affordable due to resource limitations. Therefore, tracing systems sample a subset of requests to solve this problem. We aim to record an unbiased uniform sampled subset of requests to run aggregate queries on them.

Goals. In a large distributed system, achieving an unbiased sampled trace dataset is challenging due to the various reasons we explained in previous chapter. We summarize the reasons here as they are the basis of our design.

- A request might enter the system from different entry points and go through multiple services.
- There is not only one single sampling policy across the distributed system. Each service can have its own sampling policy.
- Requests go through multiple services and get evaluated against different sampling policies.

These challenges result in non-uniform sampled trace data which is not representative of all the requests. We suggest that it is possible to find a random uniform subset of requests in the sampled trace dataset. We need to find the maximal intersection of sampled traces by different samplers so that all samplers would potentially sample this maximal intersection.

We aim to reach the following goals in our approach:

- **Samplers are configured independently.** Service owners and developers should be able to configure samplers for their service independent of other services. They care about reducing tracing overhead and they should not need to know about the configuration of other services.
- **Unbiased sampled traces.** Developers and service owners should be able to assume that the sampled traces are unbiased. They need to know that if they evaluate aggregate queries against the sampled traces, the result would be representative of the system's behaviour as a whole.
- **No coordination.** As explained in §3.3, we do not want coordination between different sampling points because we want them to make their sampling decisions independent from each other without having to pay extra tracing costs for communication and sharing information between different sampling points.
- **Independent parent-based sampling decisions.** We let each sampler make its own sampling decision regardless of other services but we always respect the parent's sampling decision. If the parent service decided to sample a request, all child services have to trace that request as well.

Having these goals defined, we propose a summary of our approach:

- **Record a biased sampled dataset.** We record a biased dataset of traces from the requests based on the user-defined sampling policies. Each service records a set of traces based on its defined sampling policy, then we find a uniform subset of sampled traces from the set of sampled traces by different services. This uniform subset is the maximal intersection of sampling policies applied at different sampling points. To

find the uniform subset, we use a consistent hashing scheme for trace id generation and making sampling decisions.

- **Track metadata.** We track some additional metadata about sampled and not sampled requests that will help us find the uniform subset of sampled traces. This metadata includes the selectivity of each service, i.e. the smallest trace id among the requests that were not sampled at a particular service. The metadata will be extracted and recorded in a tabular form with the recorded traces.

- **Query evaluation.** When we evaluate an aggregate query, we re-write the query to evaluate it on a uniform subset of traces. This means that we need to filter traces using the metadata that we recorded to achieve an unbiased sampled trace dataset for evaluating each aggregate query.

4.1 Representative tracing approach

In this section, we explain two properties that we want to achieve in our design: selectivity and total ordering. We explain the consistent hashing scheme for trace id generation and how it is used for making sampling decisions in different types of samplers. We also explain the concept of selectivity threshold and how consistent hashing along with design of our samplers make it possible to track the selectivity threshold and find the uniformly sampled subset of traces.

We define two properties for our samplers that need to hold so that we can achieve representative tracing for aggregate analysis.

- **Selectivity.** We define a property for our samplers that we call selectivity. For every sampler A and B, either the set of traces sampled by sampler A is a subset of the set of traces sampled by sampler B or vice versa. Similarly, if sampler A samples trace X and sampler B samples trace Y, either sampler A samples trace Y as well or sampler B samples trace X. With this property, we say sampler A is more selective than sampler B if the set of traces sampled by sampler A is a subset of traces sampled by sampler B; i.e. any trace sampled by sampler A would also be sampled by sampler B.

- **Total order.** Using the common random sampling approach, selectivity is a partial order. It means that for two samplers A and B, neither sampler A is more selective than sampler B nor sampler B is more selective than sampler A. In our approach, we want a total order based on selectivity. This means that for every two samplers A and B, we need one of the following cases to happen: sampler A is more selective than sampler B, sampler B is more selective than sampler A, or both samplers are equally selective.

Examples. To make the above-mentioned properties more concrete, we explain some examples.

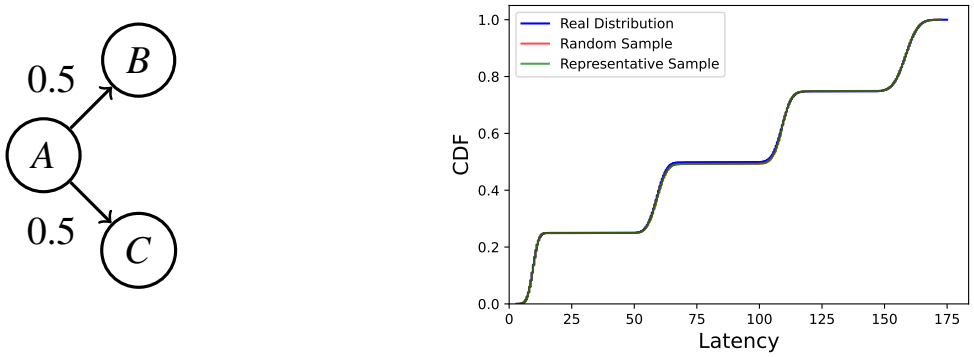
1. If sampler A samples requests with a probability of 1% and sampler B samples requests with a probability of 10%, any request sampled by A, would be sampled by B as well.
2. Consider the following samplers: Sampler A – random(10%), Sampler B – random(2%), Sampler C – random(5%), Sampler D – random(20%). We want these samplers to follow a total order based on selectivity such that: $B > C > A > D$.
3. Figure 4.1 depicts the services first introduced in Chapter 3. If all the requests were sampled with the same sampling policy, the set of sampled traces would have been a uniform sample of all the requests. However, samplers have different sampling rates and therefore different selectivity. For example, a sampler that samples 10% of the requests is less selective than one that samples 1%.

4.1.1 How do we design our samplers?

In this section, we elaborate further on how we design our samplers to achieve selectivity total ordering.

Most selective sampling policy is used to achieve a uniform sample. Recall the example from item 3 where all services share a single sampling policy. They are all equally selective and they will sample a uniform subset of requests. Similarly, if the above property about the sampler's selectivity stands, we can find a uniform subset of sampled requests based on the most selective sampling policy. If every service tracks its sampler's selectivity, we can find the overall maximum selectivity.

Single source of randomness: trace id. As explained in the previous examples, we cannot allow samplers to make random decisions and we need the decisions to be consistent. However, across the system and all samplers, we



(a) Service A calls multiple child services that all apply the same sampling rate

(b) Latency distribution of service A

Figure 4.1: In the service setup (left) requests enter the system through service A and then visit B and C. If all services sample requests with the same sampling probability, the sampled latency distribution will be the same as the real distribution (right).

need to sample a random subset of requests to record traces of them. Therefore, we put the randomness of sampling in the trace id generation phase rather than the sampling stage of the tracing pipeline.

At the beginning of a request when it enters the system, if it is sampled, the tracing system assigns a randomly generated trace id to it. This id is a large random integer number that is hashed using a consistent hash function in hexadecimal string format. The trace ids are randomly generated; this is the only source of randomness that exists in our approach. By introducing randomness only in the trace id generation, we allow building samplers that follow a selectivity total order that was explained above.

Random probabilistic sampling. Our probabilistic sampler makes sampling decisions based on the trace ids. Given the sampler's sampling probability, we can calculate a sampling threshold for trace ids and sample any trace id that is smaller than the threshold without involving more randomness. For example, if a sampler has a probability of 10%, it would sample any trace id that is smaller than the first 10th percentile of trace ids. Therefore, a sampler would either always sample a certain trace id or it will never sample it.

Consequently, if sampler A is less selective than sampler B, meaning that the sampling threshold of sampler B is smaller than the sampling threshold of sampler A, any trace id that is sampled by sampler B, would also be sampled by sampler A. This helps us achieve the property we explained earlier.

4.1.2 How do we evaluate aggregate queries?

In the previous subsection, we explained the properties that we want to achieve for our samplers and the key ideas for designing the samplers that allow them to have a selectivity total ordering. In this subsection, we explain how we evaluate aggregate queries.

Intuition. To evaluate an aggregate query that is defined over a number of services, we need to find the maximum selectivity of the sampler that impacted that service. This threshold will show us the requests that were not sampled by the samplers that are less selective. Then, we subsample the recorded traces using the maximum selectivity threshold to extract traces that would have been sampled by the sampler with the most selectivity. In the rest of this subsection, we elaborate further on the details of subsampling and query evaluation.

Services track the most selectivity they observe. A request visits multiple services during its execution and gets evaluated against multiple sampling policies by different samplers. We do not assume that we can track the selectivity of all the samplers that evaluated a request. Instead, we make services responsible for tracking the overall selectivity of their samplers that was observed by their requests.

1. Services observe all the trace ids assigned to the requests that visit them.
2. If a sampling decision at service A says *not sample*, service A will consider recording its trace id.

3. Service A always records the minimum trace id among the requests that were not sampled. We call this *selectivity threshold*.
4. The *selectivity threshold* specifies the selectivity observed by requests of service A.
5. When tracing of the requests is done, *selectivity threshold* will be extracted from each service for the purpose of aggregate analysis.

There are two options for tracking the selectivity threshold and it is only a design choice which one to choose.

- We can add the selectivity threshold as metadata to the requests that are sampled and recorded at each service and propagate it with the requests to the tracing backend.
- We can record the selectivity threshold at each service and send it out of band to the tracing backend directly.

Post-processing of selectivity thresholds. By tracking selectivity threshold at each service, we can find the most selectivity observed by requests of every service. However, to find the uniform subset of sampled requests, we need to find the most selectivity observed by all the requests. In the post-processing phase, we extract all the selectivity thresholds recorded by all services. Then, we find the minimum of all the selectivity thresholds. This determines the most selectivity observed by all the requests.

Evaluate aggregate queries against the uniform subset of sampled requests. When evaluating aggregate queries, we use selectivity threshold to find traces that can be used to form a uniformly sampled subset. When we query all the requests, we need to consider the minimum of all selectivity thresholds. This will give us the least selectivity observed by requests. Similarly, if we query requests of a particular service, we only need to use the selectivity threshold that was tracked at that service. Then, we filter traces with trace id smaller than the least selectivity threshold to obtain a uniform sampled subset of requests for aggregate query evaluation.

Why our approach works. Our approach is based on consistent trace id generation and sampling as well as tracking selectivity threshold. Making sampling decisions using our approach results in a uniform set of sampled requests. We justify why this works.

- A request is always or never sampled with a certain sampling rate. There is no randomness involved in sampling the requests.
- If a request is sampled at a very selective sampler, any less selective sampler would also sample it.
- The sampling threshold for a request type is implied by tracking the selectivity threshold.
- The sampler with the most selectivity, i.e. the smallest selectivity threshold tracked for a request type, specifies the threshold that any request with a trace id smaller than that would be sampled by any sampler.

Overview. We conclude this section by presenting an overview of our representative tracing approach. It is illustrated in Figure 4.2.

- Developers define sampling policies for their services.
- Requests enter the system and are assigned with a unique randomly generated trace id. Then, they flow through the system, visit different services and get evaluated against their sampling policies.
- All the sampling decisions are made based on trace ids with respect to a sampler's sampling threshold.
- Every service tracks the minimum trace id that visited it but was not sampled. This is selectivity threshold.
- When we process the recorded traces, we extract all the recorded selectivity thresholds.
- We re-write aggregate queries to filter traces for the query with respect to the corresponding selectivity threshold and only evaluate the queries against the traces that are filtered by selectivity threshold.

Properties. There are some properties introduced in our approach that we conclude this section with.

- We record only a small amount of data at each service to track the selectivity threshold. This data is tracked locally and does not require global knowledge of the system or coordination between different components.
- We are able to build different types of samplers that have a selectivity total ordering. The idea can be applied to probabilistic random samplers as well as max/min rate-limiting samplers.

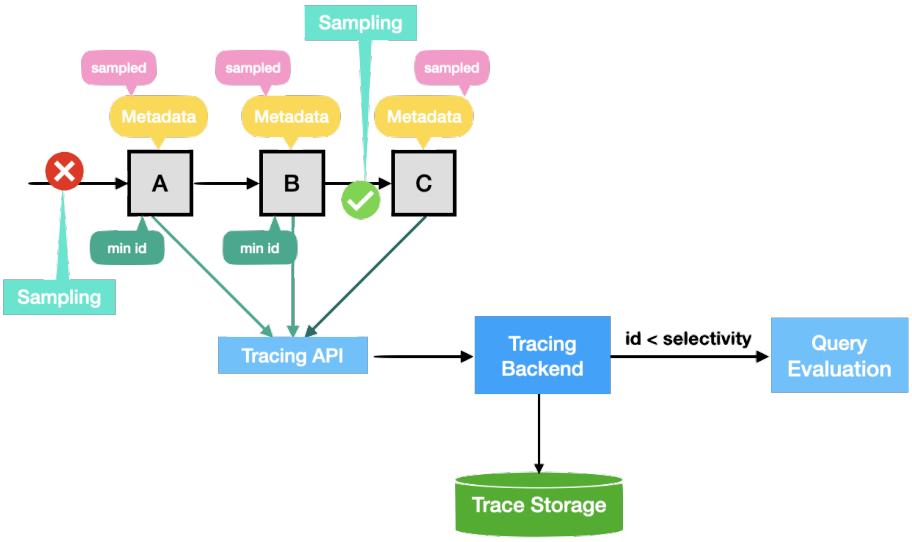


Figure 4.2: An illustration of a request flow in the distributed system, sampling decisions and query evaluation.

- Tracking the selectivity threshold shows us the maximum rate of the samplers that impacted each service. However, for a specific query, we only need to consider the selectivity threshold of the services related to that query.
- This approach allows us to additionally track the selectivity threshold for different attributes such as key-value tags recorded with requests and address queries such as filter and group by.
- One small drawback is that if a sampler has a selectivity threshold of 0, we will not have any traces for aggregate analysis. But it is still possible to get inaccurate results over the full set of recorded traces without using the selectivity threshold.

4.2 Samplers Design

In this section, we elaborate on the design of our samplers. We aim to build selective samplers with the properties that we explained in §4.1. We explain the design of probabilistic sampler, min rate-limiting sampler, and max rate-limiting sampler. These samplers are all designed based on an abstract sampler that offers some basic features.

4.2.1 Abstract Sampler

All our samplers are designed based on the key ideas of consistent trace id hashing and selectivity threshold tracking. We offer probabilistic sampling as well as max and min rate-limiting sampling that support these features.

4.2.2 Probabilistic Sampler

Key idea: sampling should not include randomness. A specific trace id will always be sampled or not under a single specific sampling probability.

Consistent sampling decisions. We would like to minimize the randomness included in the tracing pipeline; it means that we want to perform tracing (and sampling) such that the outcome is consistent and reproducible. To begin, we offer a sampling method that is also used by some current tracing systems [3, 29]. Our approach makes use of the randomness that comes in trace id generation to make random probabilistic decisions. The probabilistic sampler makes random sampling decisions based on the trace id. The sampling probability defined for a probabilistic sampler specifies a threshold for trace ids that will be sampled at that sampler. For example, if a probabilistic sampler, samples requests with a probability of 10%, any request with a trace id that was generated from the first 10% of the range of all posiible trace ids, would be sampled and any other request with a larger trace id (from the top 90%of the range of trace ids) will not be sampled.

Why does this work? Our probabilistic sampler either always samples a specific request because its trace id is

smaller than the sampling threshold or it never samples it because it is larger than the sampling threshold. In this design, sampling decisions are made at random, but we put the randomness in the trace id generation phase rather than the sampling stage. It means that we have consistent and reproducible sampling decisions and a specific trace id will always or never be sampled with a specific sampling probability regardless of when and where this sampling decision is made. Consequently, we will achieve the selectivity property. We explain further.

selectivity threshold. At a probabilistic sampler, we always track the smallest trace id that was not sampled which shows the selectivity threshold of that sampler. The selectivity threshold is very close to the sampling threshold of that sampler (only a bit larger). This selectivity threshold is obtained from the sampling probability and will be used to calculate a uniform subset of sampled traces.

Proof. Suppose we have two selective samplers A and B . Without loss of generality, let sampler A be more selective than sampler B. Any request X with a trace id smaller than the selectivity threshold of sampler A will always be sampled by sampler A. Also, as sampler A is more selective than sampler B, its selectivity threshold is smaller than the selectivity threshold of sampler B ($s_A < s_B$). So, request X would also be sampled by sampler B which keeps the sampling decisions consistent.

4.2.3 Rate-limiting Sampler

Key idea: sample traces with smaller trace ids first. We define a sampling threshold that we increase or decrease based on the sampling rate to bias rate-limiting sampler towards smaller trace ids.

Main idea. For some services or request types with lower traffic, probabilistic samplers often cannot record enough traces for analysis. A rate-limiting sampler is used mainly in combination with a probabilistic sampler to guarantee that enough traces are captured (in case the corresponding service has a low throughput). The rate-limiting sampler can be min or max rate-limiting which ensures tracing a minimum or maximum rate of requests respectively. Rate-limiting samplers often use token/leaky buckets to decide whether to sample a trace or not. This token bucket is bounded based on the rate of the sampler and whether it is min or max rate-limiting.

Selectivity threshold fails in rate-limiting. We would like to apply the approach of tracking the selectivity threshold in rate-limiting samplers as well to be able to find a uniform subset of sampled requests for aggregate analysis. However, a rate-limiting sampler will probably give us an extremely small selectivity threshold, because the requests are not sampled based on a sampling probability threshold in a rate-limiting sampler; they are sampled regardless of the trace ids and whenever the rate-limiting sampler has the capacity to sample requests.

Example. Assume the following sampling policies:

- Service A: $random(10\%)$
- Service A: $rate(10)$

If the request rate of service A is 20 requests per second, the probabilistic sampler ensures that on average we get 2 requests per second with trace ids below the selectivity threshold. However, rate(10) is not useful for determining the selectivity threshold. Rate-limiting samplers are implemented using a token bucket. In this example, 8 out of 10 sampled traces could be above the selectivity threshold and therefore not useful for evaluating aggregate queries. Hence, we need to modify the rate-limiting sampler in order to sample requests with lower trace ids with a higher chance.

Solution. To be able to utilize the selectivity threshold in a rate-limiting sampler, we make the rate-limiting sampler biased towards smaller trace ids. For this purpose, we add a built-in probabilistic sampler to the rate-limiting sampler with an initial sampling probability that will adapt to the rate of requests that are being sampled.

- When a sampling decision is made on whether to sample a request or not, both probabilistic and rate-limiting samplers make a separate sampling decision.
- Based on the individual sampling decisions made by probabilistic and rate-limiting samplers, a final decision will be made.
- Then, the sampling probability of the probabilistic sampler will be adjusted accordingly.
- If both probabilistic and rate-limiting samplers say "yes" to sampling a request, there is no adjustment required for the sampling probability.

- If only one of them says "no", it shows that the rate-limiting and probabilistic samplers are not aligned. Therefore, we have to adjust the probability used in the probabilistic sampler.

We elaborate more on how the above-mentioned cases are handled for each of the max and min rate-limiting samplers.

Proof. Assume we have a selective probabilistic sampler A and a min/max rate-limiting sampler B. The selectivity threshold of B will approximately be $s_B = r_B/throughput_B$ and for A will approximately be $s_A = p_A$. Without loss of generality, we assume that $s_A > s_B$. Requests that are sampled by B, have trace ids smaller than s_B. Therefore they would also be sampled by A since $s_A > s_B$.

4.2.3.1 Max-rate-limiting sampler

A max rate-limiting sampler aims to sample requests at a *maximum* threshold. It means that it is not allowed to exceed the sampling rate. As explained above, we add a probabilistic sampler to the rate-limiting sampler to bias its sampling decisions toward capturing traces with smaller ids. Here we explain how the sampling decision is made and the probability of the probabilistic sampler is adjusted.

- If the probabilistic sampler says "yes", the request will be sampled. The reason is that we want to sample smaller trace ids first while keeping the sampled request rate below the maximum rate defined for the max rate-limiting sampler.
- If the rate-limiting sampler said "no" to sampling this request, it means we have a higher sampling probability defined for the probabilistic sampler than the allowed rate from the rate-limiting sampler and we need to adjust the probabilistic sampler by decreasing its probability.
- If the decision of the probabilistic sampler is "no" and the decision of the rate-limiting sampler is "yes", it means that the probabilistic sampler is not sampling enough requests and we need to increase its probability so that it samples as many requests as the rate-limiting sampler expects.

This is the key idea behind our max rate-limiting sampler that allows the tracing system to sample traces with smaller ids and leverage the selectivity threshold idea.

4.2.3.2 Min-rate-limiting sampler

A min rate-limiting sampler aims to sample requests at a minimum threshold. It means that this sampler would record at least the rate specified for it. The design of the min rate-limiting sampler is similar to the max rate-limiting sampler. A built-in probabilistic sampler is defined to work with the min rate-limiting sampler in order to make it biased towards capturing requests with smaller trace ids. The idea is that because we want to ensure that a *minimum* rate of requests is sampled, we try to record requests with smaller trace ids with respect to the min-rate defined for the rate-limiting sampler. Here we explain how a min rate-limiting sampler works.

- If either the probabilistic or rate-limiting samplers agree to capture a request, it will be sampled.
- If the probabilistic sampler does not sample a request but the rate-liming does, we still trace that request but also increase the probability used in the probabilistic sampler to be aligned with the rate-limiting sampler. Because the probabilistic sampler saying "no" shows that its probability is not high enough to capture the minimum rate defined by the rate-limiting sampler.
- If the opposite scenario happens where the rate-limiting sampler does not sample but the probabilistic does, it indicates that the probability of the probabilistic sampler is too high and we need to decrease it.

Consequently, we can benefit from the selectivity threshold captured at a min rate-limiting sampler because it is inclined to capture requests with smaller trace ids with the help of a probabilistic sampler.

4.2.4 Feature Extraction and Post-processing

In this section, we talk about how to benefit from the selectivity threshold to get a uniform subset of sampled requests for the purpose of aggregate analysis when processing traces in the tracing backend.

Overview of design so far. We explained that the dataset of sampled traces is not a uniform random representation of the system's behavior as a whole. However, we suggest that there exists a uniform random subset of requests in the set of sampled traces. We introduced consistent trace id generation and sampling. We also suggested tracking some additional metadata while making sampling decisions about the minimum trace id that was not sampled at

each service. This metadata specifies the selectivity threshold that was observed at that service. Finally, we filter requests based on trace ids to obtain the set of requests with trace ids smaller than the selectivity threshold which would be a uniform random sampled subset of all the requests.

Extract selectivity threshold and the uniform sampled subset of requests. At the feature extraction phase at the tracing backend, information such as trace ids, which traces are sampled, and the selectivity threshold recorded by every service is transformed into tabular datasets. A selectivity threshold recorded at an arbitrary service A (s_A) shows that all requests with a trace id lower than the selectivity threshold were sampled at service A. To find a uniform random sample of all the requests that visited service A, we need to filter the recorded traces by their service and trace id. Any request that visited service A, and has a trace id smaller than s_A belongs to this uniform random sampled subset. Similarly, for any API, host, or other key-value tags, we can find such a uniformly random sampled subset of traces. If there are multiple tags or services mentioned in a query, we find the minimum of all the respective selectivity thresholds which corresponds to the filters that should be used for the query.

More accurate aggregate query results. We expect to get representative results by running aggregate queries on the extracted uniformly sampled trace dataset. For any filter mentioned in the query, we only consider the selectivity threshold related to that filter. If the query includes services A and B, we will consider the selectivity thresholds of services A and B. Similarly, if the query includes some tags or API names, we could track the selectivity thresholds for those values and extract them later for query evaluation. Note that we use less data than we initially recorded in this approach. Therefore, the aggregate query result might have small errors; however, it should be representative and more accurate than using a non-uniform sampled trace dataset.

Example. We define the following sampling policies for services in Figure 4.3.

- Service A: $random(1\%)$
- Service B: $random(10\%)$
- Service C: $random(5\%)$

Service C expects the same rate of requests calling it from services A and B. However, given the policies defined above, It will see more of the request type that visited service B than service A. Using the selectivity threshold, we ensure that service C has a uniform view of its callers A and B and it is possible to exclude the bias in the sampled trace data from the result of the data queries. The selectivity threshold will extract a uniform random sampled subset of requests as if both services A and B were sampling their requests at a sampling probability of 1%.

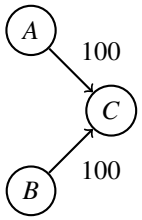


Figure 4.3: In this service setup, service C expects to see an equal number of requests coming from services A and B. In reality, its view of its callers is however affected by the sampling policies defined at those services.

Example. Recall the example from §3.1 that we repeat in Figure 4.4. Here we show that if we filter traces by selectivity threshold, we will get a better representation of requests than random sampling. We follow the same experiment where we sample requests at service A with a probability of 20% and at service D with a probability of 60%. Figure 4.5 illustrates the requests and sampled ones are colored in green. By filtering traces with a trace id smaller than the selectivity threshold, we will get the set of traces filled with the color blue. We calculate and plot the latency distribution of service B in the case of common random sampling and representative sampling where we apply the selectivity threshold. They are illustrated in Figure 4.6. Comparing these distributions, we can clearly show that the representative approach eliminates the bias significantly.

4.3 Optimization

This section explains tags, group by queries, and use of bloom filter to store selectivity thresholds over a time interval for optimization. Also, it explains how we define time intervals to clear bloom filters and start

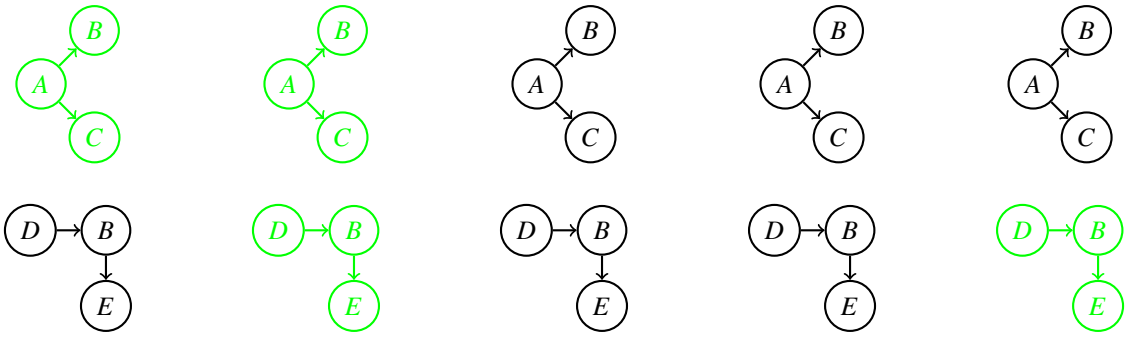


Figure 4.4: Uniformly sampled traces with a single sampling rate for all entry points (A and D): 40%

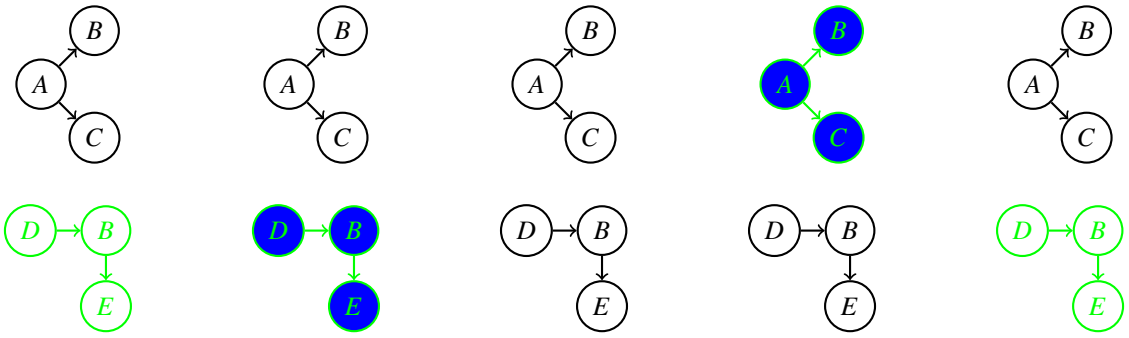


Figure 4.5: Sampled traces with different sampling rates at different entry points: 20% at A and 60% at D

monitoring new selectivity thresholds. Because otherwise, selectivity thresholds get smaller over time and would represent a very small trace dataset.

Intuition. Sampled trace datasets can be large and with various traces going through multiple services, tracking additional selectivity threshold data could bring unnecessary overhead. In this section, we explain tags and how they are used along with tracking selectivity thresholds. We also explain why selectivity threshold data can grow large and finally propose an optimization for tracking selectivity thresholds.

4.3.1 Tags

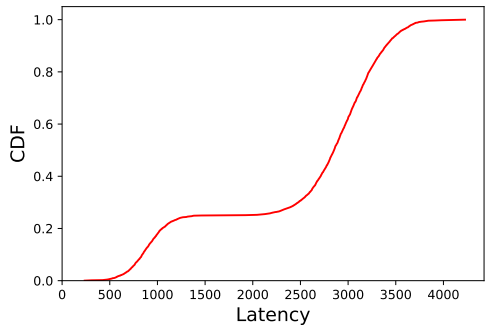
A trace can include multiple tags as part of its metadata. Tags are key-value pairs that can be added to a request at any point during its execution. For example, "API: read", "Host.Version: 2.4", etc. These tags could be addressed in queries. For example, "GROUPBY API" or "SAMPLE random (0.1) WHERE API='read'".

To be able to calculate the results of these queries, we need to track the selectivity threshold for every tag as well in each sampler. Instead of tracking one selectivity threshold per sampler, we need to have a table of selectivity thresholds per tag in each sampler and track the relevant selectivity threshold for each specific tag. However, as the number of tags increases, the size of this table can grow arbitrarily large. Storing and sending the data from this table to the tracing backend can introduce extra performance overhead to the system. We need a more efficient solution to store these selectivity thresholds.

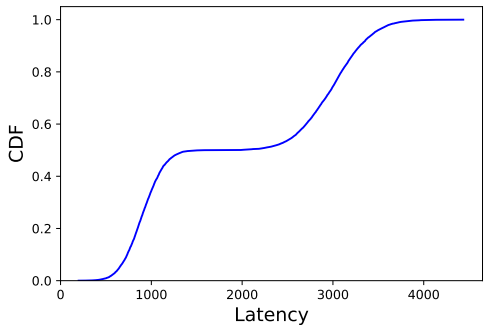
4.3.2 Bloom filter

To comply with resource limitations and prevent the high cost of storing tabular data of selectivity thresholds per tag across all requests, we propose a variant of bloom filters as a necessary optimization. We define a set of consistent hash functions for each sampler. The elements of the bloom filter are found by hashing the corresponding tag. The elements inserted in the bloom filter are selectivity thresholds. Using bloom filters will particularly allow us to track the selectivity threshold for any tag or filter carried by requests and indicated by the queries.

Insertion. For requests that are not sampled, we need to track the minimum trace id that was not sampled as part of tracking the selectivity threshold of a service. Instead of storing it as metadata for requests visiting that service, we insert it in the bloom filter. The index of the element that we insert this selectivity threshold in, is derived by hashing the tags carried by that request. So, this selectivity threshold could be inserted in the bloom filter multiple



(a) Random sample distribution



(b) Representative sample distribution

Figure 4.6: Latency distribution of service B calculated using randomly sampled traces (left) vs. representative sampled traces using the selectivity threshold (right).

times at different indices by different tags. Also, if a selectivity threshold already exists at an index in the bloom filter, we only overwrite it if it is larger than the new selectivity threshold that we wanted to insert. The insertion process is illustrated in Figure 4.7.

Lookup. As mentioned in the previous section, to find the uniform subset of sampled traces, we need to look for all selectivity thresholds corresponding with the particular service or tag that we are performing aggregate analysis on it. In a bloom filter, we just need to hash the tag or service name and look up the selectivity threshold stored in that index. Figure 4.7 illustrates how the tags are hashed and the corresponding indices are found.

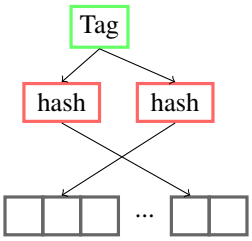


Figure 4.7: In a bloom filter, tags are hashed by multiple hash functions and the selectivity threshold is inserted at different indices calculated by the hash functions. Finding the corresponding indices works in the same way for lookup.

Improvements. In a large system containing large numbers of services, requests, and tags, we will have to store many selectivity thresholds. This increases the chance of a collision as well as overwriting selectivity thresholds with smaller amounts over time until we eventually store an extremely small selectivity threshold for some tags. Therefore, we need some key improvements in our design.

1. To prevent false results for lookup due to collision, we use multiple hash functions.
2. Selectivity thresholds that are inserted in the bloom filter, will get smaller over time. Using such small selectivity thresholds, make us run queries on limited sampled trace datasets and hence get not really precise results. As a solution, we extract bloom filter data periodically and send it to the tracing backend to prevent this problem. Then, we clear the bloom filter and insert new data in it.
3. We send the bloom filter data to the backend out of band and not as part of the trace data to prevent delays and to not make the request metadata larger than it needs to be.

4.4 Aggregate Queries

In this section, we talk about queries that we support, how we translate and re-write user-defined queries to DB queries, and calculate the results of these queries.

Focus of this thesis. The focus of this thesis is distributed tracing for the purpose of aggregate analysis. We allow

Trace ID	Span ID	Parent ID	Service	API	Latency(ms)
0aec6cbb	9a6e3aa0	-	A	read	22.5
0aec6cbb	d9b01bb6	9a6e3aa0	B	read	6.5
0aec6cbb	0077ff9c	9a6e3aa0	C	write	14
0aec6cbb	67ef21d4	d9b01bb6 0077ff9c	D	write	4
16aa1a3a	3feda8e0	-	A	read	10.5
16aa1a3a	d569d6a3	3feda8e0	B	read	8.5
16aa1a3a	6ce38dec	d569d6a3	C	write	10
16aa1a3a	e2f0e515	d569d6a3	D	read	1

Table 4.1: Trace data and features extracted into rows of the database. We run translated aggregate queries on these database rows.

users to define sampling policies and aggregate queries for their services. The aggregate queries are supposed to be performed on the sampled trace dataset. The queries that we support were listed earlier in §2.4.1: average latency, percentile latency, latency distribution, group by API, and Filter by tag.

Figure 4.8: Some arbitrary traces and services are illustrated in this figure. Some services could include different APIs that are called by different request types.

Translate queries. Tracing systems record the sampled requests and send their trace data to the tracing backend for processing. The tracing backend processing includes the feature extraction phase where the trace data is converted to tabular rows of a database. Figure 4.8 visualizes some sampled requests in DAG format and Table 4.1 illustrates their trace data and extracted features. To perform aggregate analysis, we need to translate the aggregate queries defined by users into database queries. Then, we run the database queries on the database rows of trace data. For example, *avg(A)* translates to the database query in Listing 4.1.

```
SELECT AVG(Latency)
FROM traces
WHERE Service = "A"
```

Listing 4.1: Translate an average latency query to a SQL database query

After getting the result of the aggregate query by running the equivalent database query, we could report it to the user or visualize it. For example. "plot latency distribution of service A" basically asks for *latency(A)* which translates to Listing 4.2 and returns latency CDF for all the rows that have the value of *Service="A"*. The result of this query is illustrated in Figure 4.9.

```
SELECT latency, SpanID
CUME_DIST () OVER (PARTITION BY SpanID ORDER BY Latency) AS CDF
FROM traces
WHERE Service = "A"
```

Listing 4.2: Translate latency distribution query to a SQL database query

Recall that we record the selectivity of samplers by tracking the selectivity threshold at each service. To evaluate the aggregate queries across a uniform subset of sampled traces, we need to filter traces that are in the format of rows in the trace database based on the selectivity threshold. Therefore, we need to extract the selectivity threshold from bloom filters. Table 4.2 illustrates the final table including selectivity thresholds.

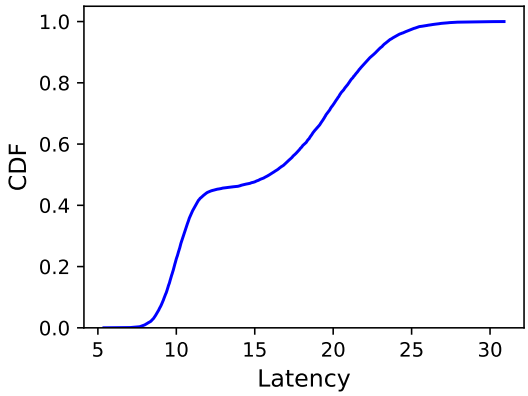


Figure 4.9: Latency distribution of service A

Trace ID	Span ID	Parent ID	Service	API	Latency(ms)	selectivityThreshold
0aec6cbb	9a6e3aa0	-	A	read	22.5	ccccccf
0aec6cbb	d9b01bb6	9a6e3aa0	B	read	6.5	d351800
0aec6cbb	0077ff9c	9a6e3aa0	C	write	14	ccc43c2d
0aec6cbb	67ef21d4	d9b01bb6 0077ff9c	D	write	4	abb324df
16aa1a3a	3feda8e0	-	A	read	10.5	ccccccf
16aa1a3a	d569d6a3	3feda8e0	B	read	8.5	d351800
16aa1a3a	6ce38dec	d569d6a3	C	write	10	ccc43c2d
16aa1a3a	e2f0e515	d569d6a3	D	read	1	abb324df

Table 4.2: Trace data and features extracted into rows of the database in addition to selectivity thresholds that are extracted from the bloom filter. Translated aggregate queries are run over this table.

In the example above, we look up the boom filter with the key *Service="A"* to extract the minimum selectivity threshold *ST* that corresponds with service A. Then we use this threshold for query translation. This approach eventually translates the query in Listing 4.2 to the following database query in Listing 4.3.

```
SELECT latency, TraceID, SpanID, Service
CUME_DIST () OVER (PARTITION BY SpanID ORDER BY Latency) AS CDF
FROM traces
WHERE Service = "A" AND TraceID < ST
```

Listing 4.3: Translate latency distribution query to a SQL database query filtered by selectivity threshold

Conclusion. We discussed the key ideas of our design which are based on sampling requests with smaller trace ids first and tracking selectivity thresholds based on the tags defined for requests and storing them in a bloom filter for each sampler. In Chapter 5, we will explain in more detail how samplers and bloom filters are implemented.

Chapter 5

Implementation

Intro. We discussed our design on a high level in Chapter 4 and explained the key ideas. In this chapter, we talk about the implementation of our design in detail and introduce our sampling algorithms and optimizations.

Programming language. We implemented this work including the trace generator and trace sampling simulator in Python3.

5.1 Trace IDs and Sampling

Consistent hashing. Each new request that enters the system is assigned a unique trace id that is randomly drawn from a fixed interval of integer numbers $[0, l)$. After a random integer number is drawn from this interval, it is hashed using a consistent hash function. The hashed value of the integer number would be assigned as a trace id. To make sampling decisions, we use the random sampling rate p at an arbitrary service X to specify a threshold of $p \times l$. This threshold specifies the range of trace ids that should be sampled at service X. To make a random sampling decision at service X for an arbitrary request, we look at the id and sample it only if the id is lower than the respective threshold. Therefore, this sampling decision would be consistent as it will always or never be sampled at this service with respect to the sampling threshold. As a result, the randomness comes only in the trace id generation and not in the sampling process. algorithm 1 explains how a random sampler makes sampling decisions.

Algorithm 1: Probabilistic sampler

```
1 begin
      Data: 0 ≤ trace_id ≤ l
2     if trace_id ≤ p × l then
3         return True
4     else
5         selectivity = min(selectivity, trace_id)
6         return False
```

5.2 Samplers

This section explains the implementation of our samplers and sampling algorithms. For each type of supported sampler, we present the algorithm and explain how sampling decisions are made and how the selectivity threshold is tracked.

5.2.1 Probabilistic Sampler

A probabilistic sampler is a uniform random sampler described in algorithm algorithm 1. Each probabilistic sampler has a sampling probability that is defined in a sampling policy. This sampling probability p is a floating-point number that must always be between 0.0 and 1.0. The sampling probability p is the base of sampling decision

making and defines a threshold for trace ids of the requests that will or will not be sampled. A probabilistic sampler with probability p always either samples a request or not based on its trace id. There is no randomness involved in making sampling decisions at a probabilistic sampler. As algorithm 1 explains, the requests with trace ids that are smaller than the threshold of $p \times l$ will be sampled and others not.

The selectivity threshold is also tracked in a probabilistic sampler. It gets updated every time a request enters the service that its trace id is larger than the sampling threshold and hence is not sampled. The sampler makes sure to keep the minimum trace id that was not sampled at all times. Using this consistent sampling method ensures that all traces with an id smaller than $p \times l$ and hence smaller than the selectivity threshold are sampled.

5.2.2 Rate-limiting sampler

We combine our rate-limiting sampler with a probabilistic sampler in order to bias the sampling decisions toward the requests with smaller trace ids. The probabilistic sampler starts sampling at a default sampling probability and then adjusts its probability based on the number of requests it has been sampling in comparison to what the rate-limiting sampler wants to sample. If the probabilistic sampler samples fewer requests than the goal rate of the rate-limiting sampler (meaning that the rate-limiting sampler would sample some requests where the probabilistic sampler says "no" to sampling them), the sampling probability needs to be increased. Similarly, if the probabilistic sampler samples more requests than the rate-limiting sampler aims to sample (meaning that the probabilistic sampler decides to sample some requests while the rate-limiting sampler does not agree and says "no"), the sampling probability needs to be decreased. We elaborate further on how this is implemented for each of the max and min rate-limiting samplers.

5.2.2.1 Max rate-limiting sampler

The number of sampled traces sampled by a max rate-limiting sampler must always be below the defined rate for the sampler. To be able to utilize the selectivity threshold, we define a probabilistic sampler with an initial parameter of 1.0 that will get updated each time a sampling decision is made. For each request, the sampling decision is successful if the probabilistic sampler says *yes*. To update the probabilistic parameter, we slightly increase the sampling probability if for a specific request the probabilistic sampler says *no* to sampling it but the rate-limiting sampler says *yes*. Also, we decrease the sampling probability if the rate-limiting sampler says *no* but the probabilistic sampler says *yes*. If both samplers say *no* to sampling a request, we do not update the sampling probability. The goal is for the sampling probability to converge to the selectivity threshold. To prevent high fluctuation in the sampling probability as a result of numerous increments and decrements, we suggest additive increments and multiplicative decrements. The additive and multiplicative parameters are derived experimentally. Algorithm 2 explains this procedure. Note that for those requests that are not sampled, the sampler function returns the value *False* and potentially updates the selectivity threshold to always track the minimum trace id that was not sampled.

Algorithm 2: Max rate-limiting sampler

```
1  begin
2      probability ← 1.0
3      increment ← 0.00001
4      decrement ← 0.98
5      if random(probability) and token_bucket.acquire() then
6          return True
7      if random(probability) and not token_bucket.acquire() then
8          probability ← probability * decrement
9          return True
10     if not random(probability) and token_bucket.acquire() then
11         probability ← probability + increment
12         selectivity = min(selectivity, trace_id)
13         return False
14     if not random(probability) and not token_bucket.acquire() then
15         selectivity = min(selectivity, trace_id)
16         return False
```

5.2.2.2 Min rate-limiting sampler

The number of sampled requests sampled by a min rate-limiting sampler is always above the defined rate for the sampler. A min rate-limiting sampler functions similarly to a max rate-limiting sampler with slight differences. A sampling decision by a min rate-limiting sampler is successful only if the rate-limiting sampler says *yes*. However, if the probabilistic sampler does not agree with this decision, we increase its sampling probability so that it samples more requests and gets closer to the goal rate defined by the rate-limiting sampler. If the probabilistic sampler says *yes* but the rate-limiting sampler says *no*, we decrease the probabilistic parameter because in this case, the probabilistic sampler is sampling more requests than is required by the rate-limiting sampler. Consequently, if neither samplers decide to sample a request, we do nothing. Similar to the max rate-limiting sampler, we implement additive increment and multiplicative decrement to avoid fluctuation in the sampling probability. The additive and multiplicative parameters are derived experimentally. Algorithm 3 explains this procedure. Similar to the max rate-limiting sampler, if a request is not sampled, the min rate-limiting sampler returns the value *False* and updates the selectivity threshold.

Algorithm 3: Min rate-limiting sampler

```
1  begin
2      probability ← 0.01
3      increment ← 0.01
4      decrement ← 0.97
5      if random(probability) and token_bucket.acquire() then
6          return True
7      if not random(probability) and token_bucket.acquire() then
8          probability ← probability + increment
9          return True
10     if random(probability) and not token_bucket.acquire() then
11         probability ← probability * decrement
12         selectivity = min(selectivity, trace_id)
13         return False
14     if not random(probability) and not token_bucket.acquire() then
15         selectivity = min(selectivity, trace_id)
16         return False
```

5.3 Optimization: Bloom Filter

Intuition. We implement a modified version of bloom filters and a set of consistent hash functions per sampler. This is explained in algorithm 4. The entries in this bloom filter are trace ids. We hash each tag, i.e. a key-value pair, to locate the index to insert the current entry. Then, we insert the currently observed selectivity threshold in the corresponding index of the bloom filter. We explain the operations of our variant of bloom filter in more detail in the following section.

Algorithm 4: Bloom filter initialization

```
1  begin
2      N ← 10000
3      false_positive ← 0.05
4      size ← −N * log(false_positive)/log(2)^2
5      hash_count = size/N * log(2)
6      for 0 ≤ i < size do
7          array[i] ← infinity
```

5.3.1 Insertion

When a request visits an arbitrary service X, if it does not get sampled at service X, we add its trace id to the bloom filter. We do this not only once for the service but for all the tags, i.e. key-value pairs, carried by that request. We hash each of the tags to find the index of that entry in the bloom filter. Then, we insert the trace id into the bloom filter for each hashed tag in its corresponding index. If an entry already exists in the calculated index of the bloom filter, we keep the minimum of the existing and the new entry. The insertion procedure is explained in detail in algorithm 5.

Algorithm 5: Bloom filter insertion

```
1 begin
2     for tag in trace.tags do
3         for i in hash_count do
4             index ← hash(tag, i)%size
5             if trace.id < array[index] then
6                 array[index] ← trace.id
```

5.3.2 Lookup

Finally, when we want to process the trace data and perform aggregate analysis, we need the selectivity threshold for the services and tags that the aggregate query refers to. We look up the hashed tag and/or service name mentioned in the query by each hash function, then we look up all the corresponding entries. Between all entries that match our query conditions, we choose the maximum of all of them. This should be the final selectivity threshold that we use in the post-processing phase for that particular query. The lookup procedure is explained in more detail in algorithm 6.

Algorithm 6: Bloom filter lookup

```
1 begin
2     item ← infinity
3     for tag in trace.tags do
4         for i in hash_count do
5             index ← hash(tag, i)%size
6             if id ≤ array[index] then
7                 id ← array[index]
8     return id
```

5.4 Trace generator and sampling simulator

5.4.1 Trace generator

Intro. We implemented a trace generator that generates datasets of traces based on the characteristics that we define for it.

Approach. We extracted information and statistics about DeathStar microservices [5] by analyzing some of their existing datasets from the **social network microservices**. We extracted the set of existing services, their latency stats, potential API calls, and transition rates between them.

To generate new datasets of traces, we specify a set of services from extracted DSB data, the number of traces that we need in the dataset, calls and transition rates between services, and generate the dataset accordingly. In the Chapter 6, we explain the configurations that we used to generate trace datasets and report the result of experiments that we ran over them.

5.4.2 Trace sampling simulator

We implemented the above-mentioned samplers in a trace sampling simulator. This simulator receives datasets of traces that are either generated by our trace generator, directly extracted from DeathStarBench [5], or Alibaba microservices [6] and replays the traces from the input dataset. Using the sampling simulator, we simulate sampling a set of requests and perform aggregate analysis over them. In Chapter 6, we explain the experiments that are evaluated using this trace sampling simulator and their results.

Chapter 6

Evaluation

Goals. In the experiments covered in this thesis, we focus on achieving the following goals:

1. **Correct (low error) aggregate query result.** We show that random sampling can result in incorrect aggregate query results and that our sampling algorithm is correct. We demonstrate that performing aggregate analysis on the sampled trace dataset using our approach yields accurate results with significantly lower error than the common random sampling that is widespread in today's tracing systems.
2. **Support enforcing multiple sampling policies.** We show that we can enforce multiple sampling policies defined by different users and service owners rather than a single universal sampling policy.
3. **Minimal performance overhead.** We show that using our sampling approach does not introduce high-performance overhead compared to random or no sampling.
4. **Scalability.** We show that our approach works on a simulated micro-benchmark (DSB) as well as a real system (Alibaba).

Challenges. We elaborate on some challenges in designing and performing sampling experiments:

1. **Multiple entry points.** Requests enter the distributed system from various entry points. The services at entry points can also have different sampling policies.
2. **Multiple sampling policies interfere with statistics.** Having multiple sampling policies defined by users that are actually correlated to each other are the source of incorrect results in evaluating aggregate queries. We want to allow users to enforce their desired policies independently and yet achieve correct aggregate query results.
3. **Workload variations.** Sampling policies, request rate, and workload proportion can change over time and we show that our approach is robust to this challenge.

Terms and definitions. In the rest of this section, we introduce some terms that we will be using in our experiments. Then, we elaborate on multiple experiments to show that we can achieve the goals that we mentioned above.

Sampling policies. We define a few default sampling policies in listings 6.1, 6.2, 6.3, 6.4 and use them later on for the experiments.

```
1 FROM * IN X
2     SAMPLE random(0.01)
```

Listing 6.1: Base policy

```
1 FROM * IN X
2     SAMPLE random(0.05)
```

Listing 6.2: Fair policy

```
FROM * IN X
    SAMPLE random(0.2)
```

Listing 6.3: High policy

```
FROM * IN X
    SAMPLE random(1.0)
```

Listing 6.4: Sample all policy

KS test. Kolmogorov–Smirnov test compares a sample distribution to a reference probability distribution or two sample distributions to each other. The test answers the question of whether the sample distribution is drawn from the reference distribution or alternatively whether the two sample distributions are drawn from the same reference distribution. This is the null hypothesis that the test accepts or rejects by calculating the distance between two distributions.

In our experiments, we use this test to compare distributions. For this purpose, we use the KS test function from the Python package *SciPy*. This function reports two outputs: **statistic** and **p-value**. The **statistic** is basically the maximum calculated distance between data points in two distributions and the smaller it is, the more similar the distributions are. Similarly, a larger **p-value** shows more similarity between the two distributions that are being compared by the KS test. In our experiments, we perform two-sample KS tests on expected distribution vs. randomly sampled distribution and expected distribution vs. representatively sampled distribution using slectivity threshold.

Mean Latency Approximation. For some of the experiments, we calculate a 95% confidence interval around the mean latency to compare random sampling to representative sampling (our approach) which uses the selectivity threshold. However, as discussed in detail before, the baseline *random* sample is not actually sampled uniformly at random, so we cannot correctly calculate the 95% confidence interval. Instead, for us to correctly calculate the 95% confidence interval, we repeat the experiment N times ($N = 100$ in most of our experiments), calculate the mean latency for each repetition, then manually calculate the 95% confidence interval of these N repetitions. (e.g. find the value x such that for 95% of our experiment repetitions, the calculated mean is within the true mean + x and true mean - x.) To prevent any confusion, from now on we use the term **Mean Latency Approximation** or **MLA** in this thesis instead of the term confidence interval for the approximated interval that we calculate with regard to mean latency.

Summary. We evaluated our sampling approach in different experiment setups. In §6.2, we demonstrate experiment results using DeathStarBench traces [5], varying the request throughput and API proportions. In §6.3, we show the results of another experiment using production Alibaba microservice traces [6].

6.1 Random Sampling with a Micro-benchmark

Goal

In this experiment, we show that random sampling can result in incorrect aggregate analysis.

Methodology

- We need a small representative system configuration where services have multiple parents/children. This system configuration could be generated by our simulator or we can use DeathStarBench.
- We define multiple sampling policies for services and change the parameters for different parent/child services.
- We perform random sampling to trace a subset of sampled requests.
- By performing aggregate analysis such as *calculate latency distribution* or *calculate average latency* on the whole dataset vs. randomly sampled dataset, we can observe how random sampling can give us results with high error.
- To compare real/expected to sampled latency distributions, we use KS test. We define a level of significance for KS test $\alpha = 0.05$ and show that under random sampling, the p-value does not fall between 0.95 and 1.0. Therefore, the sampled distribution is not representative of the real distribution.

Experiment setup

Intuition. In this experiment, we perform the common random sampling approach that is wide-spread in today's tracing systems over requests of DeathStar microservices benchmark. We show how different sampling policies for different services can affect each other. We cannot control how users define their sampling policies. We do not want to coordinate services and their sampling policies either. Therefore, depending on how sampling policies are defined, evaluating aggregate queries on sampled data could result in incorrect statistics. However, using our representative approach, the error in the aggregate results will be significantly reduced.

Services and calls. In this experiment we use social network microservices fro DeathStarBench [5] and mainly focus on two workloads from **compose post** and **user timeline**.

- **Compose post.** The requests of compose post, visit *compose-post-service* and then call *post-storage-service* in order to store the created post in storage.
- **User timeline.** The requests that read user timeline, first go through *home-timeline-service* and then call *post-storage-service* to read the posts from storage.

To simplify the experiment, we focus on the three mentioned services *compose-post-service*, *home-timeline-service*, and *post-storage-service*. They are configured similarly to the illustration in figure 6.1. Also, the latency distributions of these three services are shown in figures 6.2a, 6.2b, and 6.2c respectively.

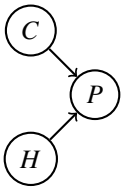


Figure 6.1: DeathStar microservices setup - social network microservices

Sampling policies. We need to define sampling policies for each service. We define generic probabilistic sampling policies for the above-mentioned services from DSB. Assume *compose-post-service* samples requests with a probability of $p_{compose}$, *post-storage-service* samples requests with a probability of $p_{storage}$, and *home-timeline-service* samples requests with a probability of $p_{timeline}$ uniformly at random.

Experiment result

Sampling result. Here we present the result of evaluating the experiment over datasets of randomly sampled requests. Even though the sampling policies are defined independently of each other, in reality, they are correlated and they can affect each other. Therefore, the result of aggregate analysis over a set of randomly sampled traces could be inaccurate with a large error. Here we look at different scenarios based on how the sampling policies are defined for each service.

Case 1: $p_{compose} = p_{timeline} = p_{storage}$

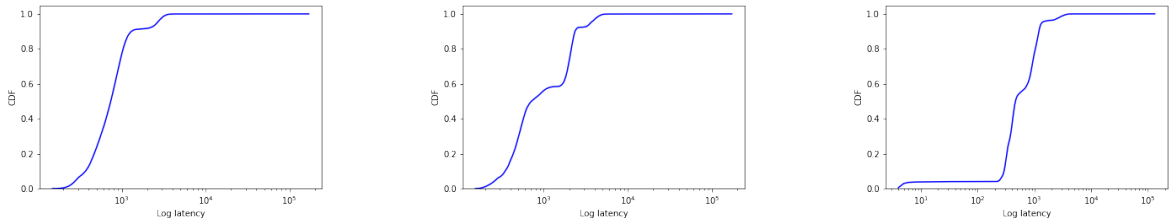
Requests of all these services will be equally likely to be sampled and the result will be similar to the expected latency distribution in figure 6.2.

Case 2: $p_{compose} = p_{timeline} \neq p_{storage}$

Requests of *compose-post-service* and *home-timeline-service* are sampled with the same sampling rate. This means that the number of calls from these services to *post-storage-service* in the sampled dataset is proportional to the original dataset. Therefore, whether *post-storage-service* samples additional requests or not, the sampled dataset will be uniform and representative of the original dataset. In this case, if we calculate the sampled latency distribution of any of the services, we will still get the same results as the expected latency distribution.

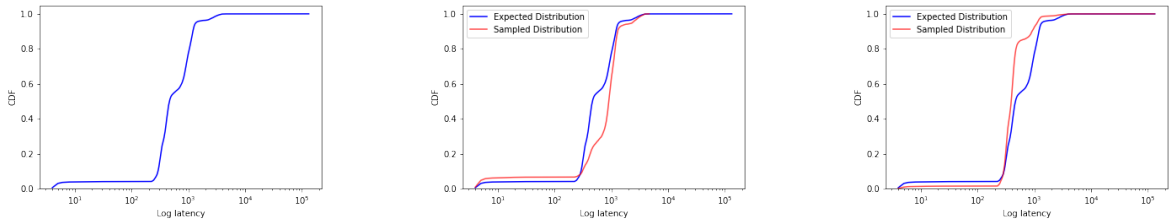
Case 3: $p_{compose} < p_{timeline} \leq p_{storage}$ **or** $p_{timeline} < p_{compose} \leq p_{storage}$

In this scenario, *post-storage-service* has the highest sampling rate among all services. This means that it will sample more requests coming from either *compose-post-service* or *home-timeline-service* regardless of how high each of



(a) compose-post-service (b) home-timeline-service (c) post-storage-service

Figure 6.2: Log scale latency distribution of three DSB services



(a) Expected distribution

(b) Sampled distribution ($p_{compose} < p_{timeline}$)

(c) Sampled distribution ($p_{timeline} < p_{compose}$)

Figure 6.3: Expected vs. sampled latency distribution of post-storage-services

their sampling rates was and how many requests were sampled by them initially. Also, because *post-storage-service* is performing sampling with a higher rate, it will have to sample the same rate of requests from *compose-post-service* or *home-timeline-service* which means, we will get a uniform sampled subset of requests of these two services. Therefore, regardless of how high the sampling rates of each of *compose-post-service* and *home-timeline-service* were, we will get the same sampled latency distribution for these services as we expect. The log latency distributions of all services are shown in figure 6.2.

Case 4: $p_{storage} \leq p_{compose} < p_{timeline}$ **or** $p_{storage} \leq p_{timeline} < p_{compose}$

If the sampling rate of one of *compose-post-service* or *home-timeline-service* is higher than the other one, *post-storage-service* receives more sampled requests from that service than expected. Therefore, its latency distribution will be biased toward requests that went through the service with a higher sampling rate. In such a case, the latency distribution of *post-storage-service* would be as illustrated in figures 6.3b or 6.3c.

If we use our representative sampling approach that tracks the selectivity threshold, we can fix this bias. By filtering traces using the selectivity threshold, we assume that all requests were sampled by $\min(p_{compose}, p_{timeline}$. It means we only use the traces with a trace id below this calculated selectivity threshold. This will be a scenario similar to case 1 or 2. Therefore, if we calculate the sampled latency distributions, we will get more accurate results compared to the expected distributions.

To prove this, we use **KS test** to compare the expected and randomly sampled distribution vs expected and representatively sampled distribution using selectivity threshold. Looking at the result of the KS tests, we will get lower *KS statistic* and higher *p-value* when using the representative sampling approach compared to only using random sampling.

Case 5: $p_{compose} < p_{storage} < p_{timeline}$ **or** $p_{timeline} < p_{storage} < p_{compose}$

This scenario is quite similar to case 4. The rate of sampled requests arriving at *post-storage-service* will be biased towards the request type that initially visited the service with a higher sampling rate. It means if $p_{compose} < p_{storage} < p_{timeline}$, we will get more traces from *home-timeline-service* and randomly sampled vs. expected latency distributions similar to figure 6.3b. If $p_{timeline} < p_{compose}$ we will get more traces from *compose-post-service* and therefore, randomly sampled vs. expected latency distributions similar to figure 6.3c.

Now if we switch to the representative sampling approach and filter traces by selectivity threshold, we would get more accurate results. At *compose-post-service*, the selectivity threshold would be around $p_{compose}$, and at

home-timeline-service, the selectivity threshold will be around $p_{timeline}$. However, at *post-storage-service*, the selectivity threshold would be equal to $min(p_{compose}, p_{timeline})$. If we filter sampled traces now and calculate their latency distribution, we will get an accurate sampled distribution that is similar to the distribution shown in figure 6.3a.

6.2 DeathStarBench

As explained in Chapter 5, we designed and implemented a simulator that extracts information such as latency distributions, throughput, transition rates, etc. from DeathStar microservices benchmark. This information lets us build an arbitrary topology of DSB services and run experiments on them. From the topology, we can generate new traces with arbitrary features. In the next two subsections, we define a topology to generate DSB traces for our experiments.

6.2.1 Varying Sampling Rate

In this experiment, we keep the topology of services fixed and change their sampling policies.

Experiment setup

Intuition. In this experiment, we change some of the sampling parameters at different steps and observe some statistics of sampled traces. We compare results of random sampling to our representative sampling approach that uses the selectivity threshold. We measure some statistics about the traces for aggregate analysis.

1. **KS test.** We run KS test for every variation of the experiment that uses a different sampling policy. We observe how the KS test results change based on the sampling rate and under what conditions we get more accurate results.
2. **Number of traces.** In our sampling approach, we aim to produce correct data and statistics by using fewer traces than a random sampler would do. (Recall from Chapter 4, how we use fewer traces by filtering recorded traces using the selectivity threshold and only evaluate aggregate queries on the subset of filtered traces.) We observe what percentage of sampled traces we actually use in our approach.
3. **Mean latency approximation.** Using fewer traces can of course affect the accuracy as we have less data available for analysis. We observe 95% mean latency approximation while the sampling rate changes and we use fewer traces.

Topology and setup. We picked three microservices *compose-post-service* (C), *home-timeline-service* (H), *post-storage-service* (P), and generated 200000 traces that match the service configuration illustrated in Figure 6.4.

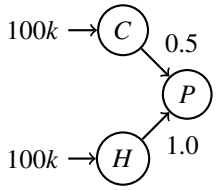
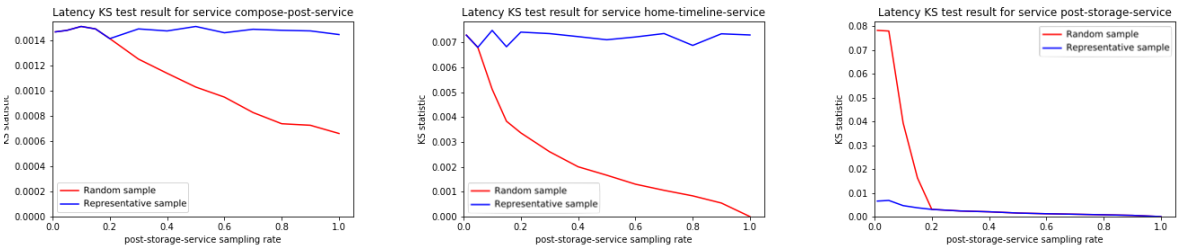


Figure 6.4: An arbitrary deathStar microservice setup defined for the experiment

Methodology. We define fixed sampling policies for *compose-post-service* and *home-timeline-service* ($p_{compose} = 5\%, p_{timeline} = 20\%$), and change the sampling rate of *post-storage-service* ($p_{storage}$) from 0% to 100%. We sample requests based on different sampling policies defined for *post-storage-service*, and evaluate a latency-related query on them.

Experiment Result

KS Test. We run KS test for sampled vs. expected latency distributions of each service. Figure 6.5 shows the KS test result of this experiment for each service. The x-axis indicates the sampling rate of *post-storage-service* and the y-axis shows the KS statistic. For each service, the blue graph shows the result of KS test for expected vs. sampled distribution and the red graph shows the result of KS test for expected vs. representatively sampled distribution.



(a) compose-post-service (b) home-timeline-service (c) post-storage-service

Figure 6.5: KS statistic of random sampled distribution vs. representative sampled distribution

Home-timeline-service. The selectivity threshold for this service is at the threshold of $p_{timeline}$ regardless of $p_{storage}$. When $p_{storage}$ goes above this threshold, there will be more trace data available (sampled and recorded through random sampling) and therefore, the sampled distribution gets closer to the expected distribution. This happens simply because there is more data points available and the accuracy increases, while the representative sampled distribution does not change since there is no increase in the number of traces for this approach. This does not mean that using the representative sampling is making the statistics worse. We perform this approach by tracking the selectivity threshold to get uniform and randomly sampled data that is not biased toward a specific request type. This approach works best when we have trace data collected from multiple services and not only one service.

Compose-post-service. The selectivity threshold for this service is at the threshold of $p_{compose}$. Similar to what we explained for *home-timeline-service*, as long as $p_{storage}$ is less than $p_{compose}$, we see the same results for sampled and expected KS statistic. When $p_{compose}$ goes above the selectivity threshold, sampled data gets better results simply because there will be more data points available to analyze.

Post-storage-service. As expected in the discussion in case 4 of the first experiment 6.1, the result of using the representative sampling is clearly more accurate in *post-storage-service* than random sampling. Also, as discussed in case 3 of the same experiment 6.1, when the sampling rate of this service ($p_{storage}$) goes above the threshold of $p_{timeline}$ (the larger sampling rate between *home-timeline-service* and *compose-post-service*), the KS statistic will be the same for randomly vs. representatively sampled data.

Number of traces used in our approach. Our goal is to get *correct* results to aggregate queries by using *fewer* sampled traces. However, using too few traces could lower the accuracy. To show this effect, we show in figure Figure 6.6, what percentage of sampled traces is used by each service using representative sampling based on the varying sampling rate of $p_{storage}$.

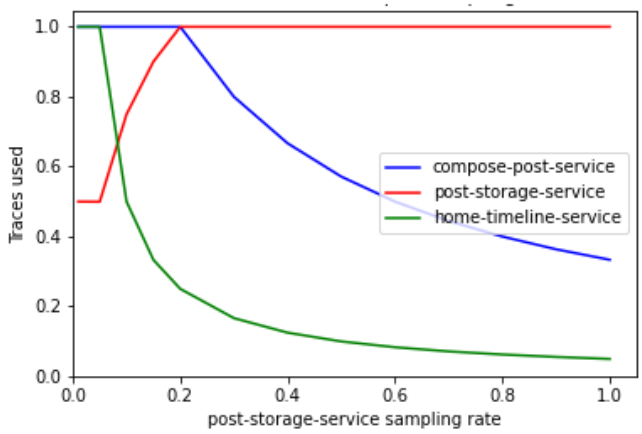
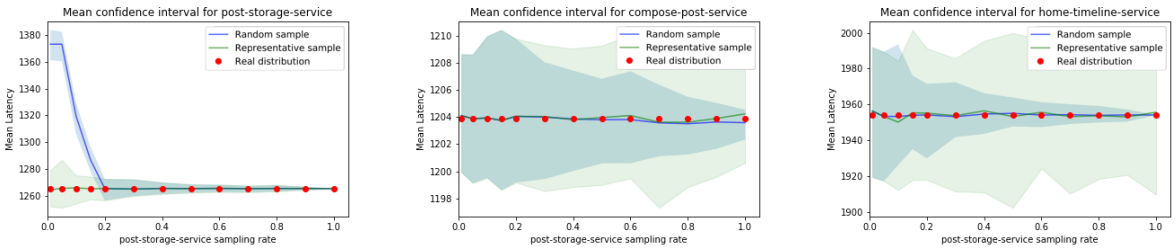


Figure 6.6: Percentage of traces used in representative sampling per *post-storage-service* sampling rate

Mean Latency Approximation. Now, we discuss the result of calculating the mean latency approximation of sampled traces for each service. We compare the randomly sampled traces to representatively sampled traces and justify how tracking the selectivity threshold plays an important role.



(a) post-storage-service (b) compose-post-service (c) home-timeline-service

Figure 6.7: 95% mean latency approximation of sampled traces

We run the experiment multiple times and calculate a 95% mean latency approximation around the real mean latency. We calculate the latency approximation intervals for both random and representative sampled trace datasets for comparison. We plot these mean latency approximations for varying sampling rates in *post-storage-service* and observe how it changes in Figure 6.7.

For *compose-post-service* and *home-timeline-service*, there is not a lot to discuss as the real mean latency always falls in the approximation intervals (see Figure 6.7b and Figure 6.7c for reference). However, we are particularly interested in the result of post-storage-service in Figure 6.7a, because the mean latency approximation calculated by randomly sampled data does not cover the real mean latency when the sampling rate is below the selectivity threshold. On the other hand, we do not see this incorrect result when we use the representative sampled data. We can see that the mean latency approximation for representative sampled data is sometimes larger than randomly sampled data due to using fewer traces. However, the sampled mean stays closer to the real mean latency which shows how more accurate our approach can be.

Note that we are using fewer sampled traces in the representative sampling approach compared to the random sampling approach and we are still getting more accurate results. However, we can observe that in Figure 6.7b, random sampling offers a slightly better approximation at sampling rates of 0.9 and 1.0 but the approximation still covers the real mean latency.

6.2.2 Varying Transition Rate

In this experiment, we define a fixed set of sampling policies for our services, but change the topology by changing the transition rate between two services and repeat the experiment.

Experiment setup

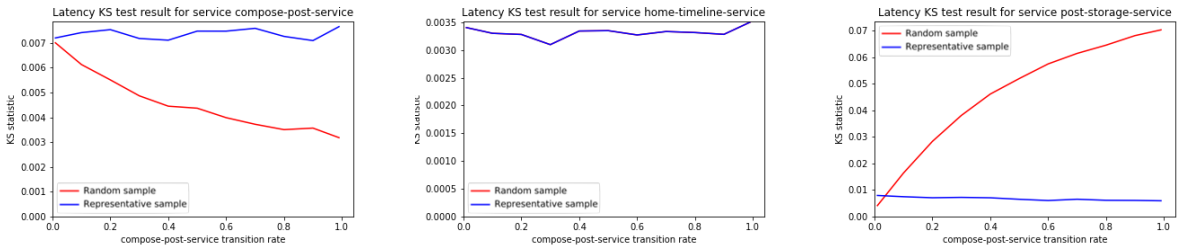
Intuition. In this experiment, we change the transition rate of one service (*compose-post-service*) at each repetition of the experiment. For each repetition, we measure some statistics about sampled traces. We compare the statistics for random sampling to results of our representative sampling approach that tracks the selectivity threshold.

Topology and setup. In this experiment, we use the same setup shown in Figure 6.4 and only vary the transition rate from *compose-post-service* to *post-storage-service* from 0.01 to 99%.

Methodology. We define fixed sampling policies for each of the three services. *Compose-post-service* ($p_{compose}$ follows the base policy 6.1), *home-timeline-service* ($p_{timeline}$ follows the high policy 6.3), and *post-storage-service* ($p_{storage}$ follows the fair policy 6.2). For each repetition of the experiment, we change the transition rate of *compose-post-service* from 0.01 to 0.99 and perform KS test. We will see how the latency distribution of *post-storage-service* changes by changing this transition rate.

Experiment result

KS Test. Figure 6.8 shows the result of running KS test in this experiment for each service. For each service, the blue graph shows the statistic calculated by the KS test for expected vs. sampled distribution. The red graph shows the statistic measured by KS test for expected vs. representativelly sampled distribution. The KS statistic is calculated for each repetition of the experiment for varying transition rate of *compose-post-service*.



(a) compose-post-service (b) home-timeline-service (c) post-storage-service

Figure 6.8: KS statistic of random sampled distribution vs. representative sampled distribution

Home-timeline-service. KS test result of randomly sampled is identical to the result of representative sampling approach because all the transition and sampling parameters of *home-timeline-service* are constant and not changing.

Compose-post-service. KS test result of randomly sampled traces is better than the representatively sampled traces. By increasing the transition rate of this service, more traces will be sampled by *post-storage-service* due to its higher sampling rate. Hence, when we calculate the latency distribution of the randomly sampled traces, we included more data points which makes the distribution more accurate. However, regardless of the change of transition rate, the selectivity threshold of this service does not change and stays at the threshold of *compose-post-service*'s sampling rate (0.01). Hence, the KS test statistic is not improved by changing the transition rate. It is important to note that both results are bellow 0.01 which is an acceptable range of KS statistic.

Post-storage-service. Using the representative sampling approach, KS test statistic is improved for this service, because more traces go through this service and hence more traces get sampled which gives us more data points for calculating the latency distribution.

Number of traces used in our approach. We show in Figure 6.9, what percentage of sampled traces is used in representative sampling approach based on the varying transition rate of *compose-post-service*. In the rest of this subsection, we explain how the mean latency approximation for the representative approach is more accurate than random sampling, even though we use fewer sampled traces for analysis.

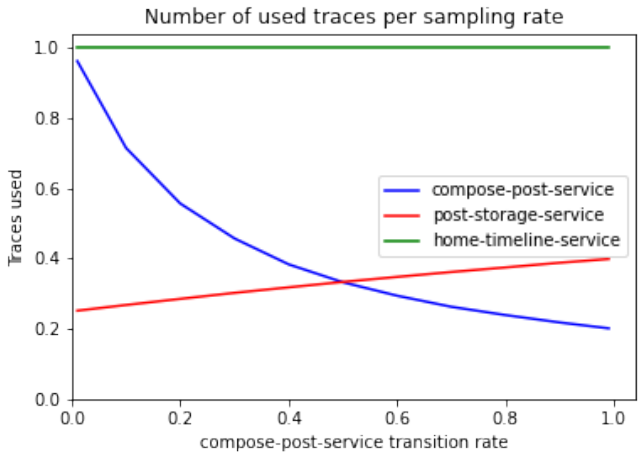
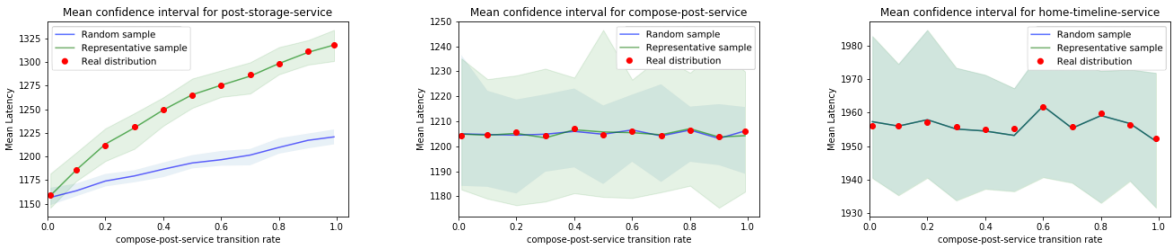


Figure 6.9: Percentage of traces used in representative sampling approach per *compose-post-service* transition rate

Mean Latency Approximation. We calculate a 95% mean latency approximation around the real mean latency of services for representatively sampled traces and compare it to the mean latency of randomly sampled traces. We plot these mean latency approximations for varying transition rates of *compose-post-service* and observe how it changes in Figure 6.10.

We are particularly interested in the result of post-storage-service, because the mean latency approximation calculated by randomly sampled data does not cover the real mean latency at all. However, for the representative sampling approach, the mean latency approximation always covers the real mean latency. We also can see in Figure 6.10a that



(a) post-storage-service (b) compose-post-service (c) home-timeline-service

Figure 6.10: 95% mean latency approximation of sampled traces

the larger the transition rate becomes, the farther the mean latency approximation gets from the real mean latency. Note that this is happening even though as we mentioned before, we are using fewer traces compared to the random sampling approach.

For the two other services, we do not have much to discuss as the mean latency approximations of random and representative sampling approaches are identical and both cover the real mean latency as well.

6.3 Alibaba Microservices

So far, we evaluated our approach using synthetic traces that we generated and also traces from DeathStarBench. To demonstrate that our approach also works on real trace datasets, we perform a similar experiment to §6.2.1 on Alibaba microservices trace dataset [6].

Experiment Setup

We picked a subset of Alibaba microservices traces [6] containing 788533 traces traversing 10910 microservices. We picked three microservices that we call A, B, and C. These three servicesare configured as illustrated in Figure 6.11 and we defined simple random sampling policies for them. We sample requests at service A uniformly at random with high sampling policy 6.3 and sample requests of service B with the base sampling policy 6.1. We also change the sampling rate at C from 0% to 100% and repeat the experiment. For each repetition of the experiment, we measure some statistics and present them in some graphs.

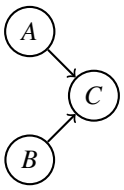
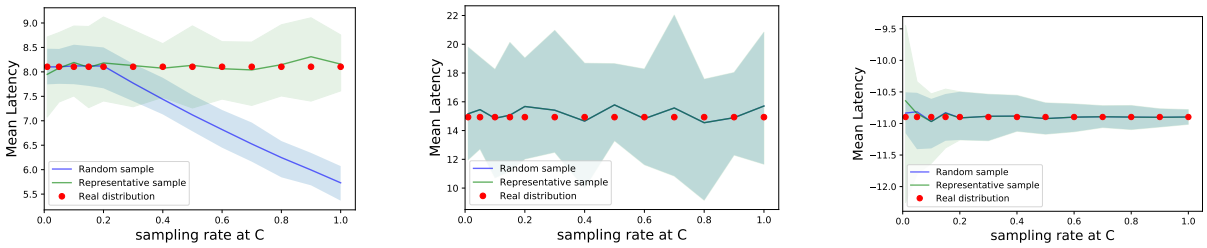


Figure 6.11: Alibaba service setup

Experiment Result

Mean latency approximation comparison. Figure 6.12 illustrates a 95% mean latency approximation around the real mean latency for each of the services A, B, and C. The mean latency approximation suggests an interval based on the sampled mean latency values that should cover the real mean latency. We calculate this approximation interval for repetitions of the experiment and for both randomly and representatively sampled traces and compare their results.

As we demonstrate in Figure 6.12a, the mean latency approximation does not cover the real mean latency of service A due to the overlap of multiple sampling policies and probably a bias in some sampled request types. More concretely, we see the effect of this bias when sampling probability at C goes above 20% which is the sampling probability of service A and its selectivity threshold. Recall Figure 6.11 where service A potentially calls service C. When service C begins to sample more requests from service A, the requests that only visited A and not C will be relatively under-sampled. Therefore, the set of sampled traces will be biased towards the request type that visits both A and C.



(a) Service A (b) Service B (c) Service C

Figure 6.12: 95% mean latency approximation of randomly and representatively sampled Alibaba traces.

For services B and C, however, we do not see the same problem and the mean latency approximations are calculated more accurately.

The error in mean latency approximation. As we explained earlier, we use only a subset of all sampled traces in our approach. As Figure 6.13 demonstrates, the number of traces sampled from service A that is used in aggregate query evaluation, decreases as the sampling rate of service C increases. This is the reason why we have a larger interval approximated for mean latency when we use our representative approach compared to the random sampling approach. Note that for analysis at service A we always use fewer sampled traces in our approach and always calculate a larger mean latency approximation interval as illustrated in Figure 6.12a.

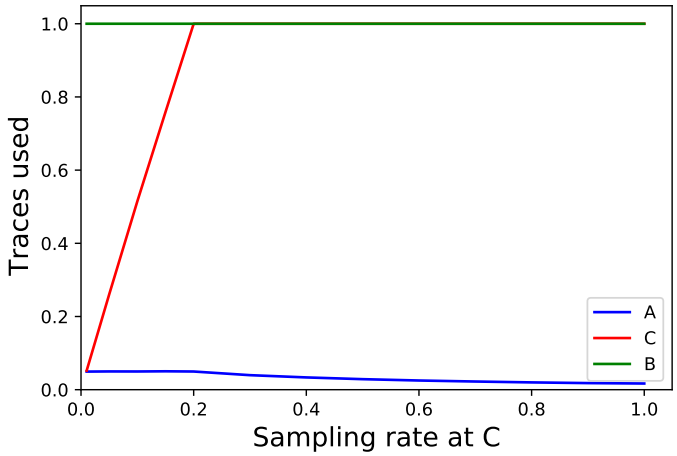


Figure 6.13: Percentage of sampled traces of each service that are actually used in our approach per service C sampling rate

On the other hand, looking at Figure 6.13, by changing the sampling rate of service C, we never sample more requests with the representative sampling approach compared to the random sampling approach. The red graph here that indicates percentage of traces used from service C, shows that after passing the selectivity threshold, service C uses all of the randomly sampled traces in representative approach as well. This justifies why in Figure 6.12c, we calculated a larger mean latency approximation when the sampling rate was below the selectivity threshold (20%). This is because below the selectivity threshold, fewer data points are available which means that the approximation would be a bit less accurate. However, the approximation interval always covers the real mean latency which is the main goal.

Finally, the calculated mean latency approximation for service B in Figure 6.12b is always the same for the representative vs. random sampling approach.

Chapter 7

Conclusion

Our preliminary results demonstrate the issue of aggregate analysis using simulated traces. The focus of our ongoing work is to translate our ideas into an online setting. In particular, we are tackling the following main challenges: first, an automatic query rewriting component that takes a user query and transparently rewrites the query to restrict the underlying dataset to an unbiased subset; second, efficient and representative sampler implementations that tracks *selectivity threshold* as metadata at runtime and report these to the tracing backend; and third, to tackle scalability, we are developing approximate *selectivity threshold* metadata tracking using bloom filter variants. Overall, we believe that our approach will transparently address the problem of inaccurate query results described in Chapter 3 and offers a practical approach in Chapter 4 to derive more accurate aggregate query results with lower error.

Bibliography

[1] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," tech. rep., Google, Inc., 2010.

[2] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), p. 34–50, Association for Computing Machinery, 2017.

[3] Jaeger, "Jaeger 1.26 Documentation: Sampling." Retrieved July 2022 from `https://www.jaegertracing.io/docs/1.36/sampling`, 2021.

[4] L. Zhang, V. Anand, Z. Xie, Y. Vigfusson, and J. Mace, "The benefit of hindsight: Tracing edge-cases in distributed systems," 2022.

[5] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for cloud and iot microservices," 2019.

[6] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 412–426, 2021.

[7] "Distributed tracing · open telemetry." https://lightstep.com/opentelemetry/tracing. Accessed: August 2022.

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," USENIX Symposium on Networked Systems Design and Implementation (NSDI '07), 2007.

[9] "Architecture · jaeger," https://www.jaegertracing.io/docs/1.20/architecture/. Accessed: November 2020.

[10] "Discussion for aggregated trace metrics (atm) · issue #2574 · jaegertracing/jaeger · github." https://github.com/jaegertracing/jaeger/issues/2574. Accessed: November 2020.

[11] L. Deri, S. Mainardi, and F. Fusco, "tsdb: A compressed database for time series," in *Traffic Monitoring and Analysis* (A. Pescapè, L. Salgarelli, and X. Dimitropoulos, eds.), (Berlin, Heidelberg), pp. 143–156, Springer Berlin Heidelberg, 2012.

[12] B. Agrawal, A. Chakravorty, C. Rong, and T. W. Wlodarczyk, "R2time: A framework to analyse open tsdb time-series data in hbase," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pp. 970–975, 2014.

[13] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed, "Scuba: Diving into data at facebook," *Proc. VLDB Endow.*, vol. 6, p. 1057–1067, aug 2013.

[14] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, (New York, NY, USA), p. 312–324, Association for Computing Machinery, 2019.

[15] "Delayed sampling · issue #1861 · jaegertracing/jaeger · github." https://github.com/jaegertracing/jaeger/issues/1861. Accessed: October 2020.

[16] "Rethink client metrics for sampled/unsampled spans/traces'#1862 · jaegertracing/jaeger · github." https://github.com/jaegertracing/jaeger/issues/1862. Accessed: November 2020.

[17] "Start sampling after a certain threshold · issue #2594 · jaegertracing/jaeger · github." https://github.com/jaegertracing/jaeger/issues/2594. Accessed: November 2020.

[18] "collector: reload sampling strategies on file content change#1058 · jaegertracing/jaeger · github." https://github.com/jaegertracing/jaeger/issues/1058. Accessed: November 2020.

[19] "Collector api to refresh sampling strategies · issue #2186 · jaegertracing/jaeger · github." https://github.com/jaegertracing/jaeger/issues/2186. Accessed: November 2020.

[20] "Support more tail-sampling scenarios#1978 · open-telemetry/opentelemetry-collector · github." https://github.com/open-telemetry/opentelemetry-collector/issues/1978. Accessed: November 2020.

[21] Y. S. sundarkvp, "Sampling by custom fields in the request· issue #260 · jaegertracing/jaeger · github," 2018.

[22] "support jaeger.tags when calling 'getsamplingstrategy'#1687 · jaegertracing/jaeger · github." https://github.com/jaegertracing/jaeger/issues/1687. Accessed: November 2020.

[23] "Remote sampling strategy support for agent level tag #1692 · jaegertracing/jaeger · github." https://github.com/jaegertracing/jaeger/issues/1692. Accessed: November 2020.

[24] "Enhance tail-sampling to have other processors' features to make it much more powerful'#1162 · open-telemetry/opentelemetry-collector · github." https://github.com/open-telemetry/opentelemetry-collector/issues/1162. Accessed: November 2020.

[25] "Zipkin." https://zipkin.io/. Accessed: July 2022.

[26] "Zipkin." https://istio.io/latest/docs/tasks/observability/distributed-tracing/zipkin/. Accessed: July 2022.

[27] "Zipkin." https://zipkin.io/brave/jdiff/5.11-to-5.12/brave-5.12.3/brave/sampler/Sampler.html. Accessed: July 2022.

[28] O. Ertl, "Estimation from partially sampled distributed traces," 2021.

[29] "Documentation · open telemetry." https://opentelemetry.io/docs/. Accessed: August 2022.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Saarbrücken, _____ _____

(Date/Datum) (Signature/Unterschrift)