Constructing Workflow-centric Traces in Close to Real Time for the Hadoop File System

Neel Bhalla

Lexington High School, Lexington, MA 02420

Adviser: Prof. Raja Sambasivan, Tufts University

Research performed for the 2019-2020 MIT CS PRIMES Research Program

(Jan 2019 – June 2020)

ABSTRACT

Diagnosing problems in large scale systems using cloud based distributed services is a challenging problem. Workflow-centric tracing captures the workflow (work done to process requests) and dependency graph of causally-related events among the components of a distributed system. But, constructing traces has historically been performed offline in batch fashion, so trace data is not immediately available to engineers for their diagnosis efforts. In this work, we present an approach based on graph abstraction and streaming framework to construct workflow-centric traces in near real time for the Hadoop file system. This approach will provide the network operators with a real time understanding of the distributed system behavior.

1. INTRODUCTION

End-users of web services (e.g. Gmail, Calendar, Messaging Apps, GDrive, DropBox, Facebook, Google and Amazon or a doctor accessing patient records on a cloud drive) expect a consistent performance experience irrespective of the machine, place or network they connect from. However, understanding, analyzing, and troubleshooting performance for a web service company or network provider can be difficult due to the range of heterogeneous clients, web browsers and apps which connect to the service.

Figure 1. We can't debug distributed services it is complicated, with 1000's of nodes and services. It is very difficult to identify when cloud services for e.g. Netflix, Facebook or Twitter go down. When American Airlines cloud-based ticketing, system went down on March 26, 2019, it stranded thousands of passengers. To find performance issues in distributed systems we need to do tracing. [12]

Diagnosing problems require capturing traces within a system. These traces capture the path (e.g. services, components and functions called) and timing of each request within the system. These traces represent the flow of requests for a given workload and are called workflows [2]. The components of distributed system have to be instrumented to

generate workflow-centric traces. In this work we will present an approach to reconstruct traces in real time which will give a network operator a real time view of the performance and behavior of their system.

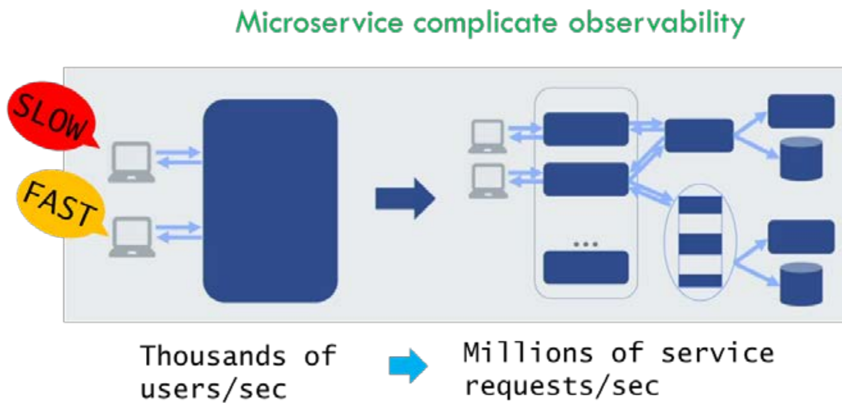


Figure 2. A typical microservice architecture, where thousands of users can access services concurrently, generate millions of services requests/sec. Depending on the latency of the services requests some users can categorize the service as slow and some can categorize the services as fast. Finding the root cause analysis of this problem is like looking for a needle in a haystack.

HDFS (Hadoop Distributed File System) is distributed file system, used with MapReduce applications in datacenters. Performance of HDFS can directly affect the performance of cloud services in datacenters. Apache Hadoop is an open source implementation of Google's MapReduce. It is composed of two main components. One component is the MapReduce software framework for the distributed processing of large data sets on compute clusters. The other is the Hadoop Distributed File System (HDFS), a distributed file system that provides high throughput access and is commonly used by the MapReduce component as its underlying file system for retrieving input and outputting results. As with any other distributed system, performance problem diagnosis is difficult in Hadoop. Assembling the traces from thousands of workflows from HDFS can be time consuming. In this research we present an approach to generate workflow-centric traces from HDFS in real time. Though we are using the HDFS for experimentation, the Model we present can be used for any micro-service architecture. This can offer a network operator a real time view of the performance and behavior of their system.

This report is structured as follows. Section 2 provides a background. Section 3 presents the Proposed Method. Section 4 presents the results. Section 5 presents the discussion and Section presents the conclusion.

2. BACKGROUND ON TRACING

Tracing captures the flow of requests. This is achieved by capturing activity records at the various instrumentation points placed in the distributed system components. Each record contains information such as instrumentation point name, a timestamp, and optionally other contextual information. In general, records are associated with individual requests and thus also propagate a request identifier. These records can then be stitched together to form a request flow graph that shows the control flow (see Figure 3). In such a graph, nodes are labeled by the instrumentation point name and edges are labeled with the latency that the request experienced between one instrumentation point and the next. The instrumented points are part of a DAG (Directed Acyclic Graphs). The events are sent from the datacenter as an event stream. The challenge is to reconstruct the DAG in real time to understand the request flow. As noted previously, end-to-end tracing can simplify the process of performance problem diagnosis with the rich information it provides. Such tracing can be implemented with low overhead as seen by the multiple independent implementations, such as Dapper [5], Magpie [2], Canopy [1], or X-Trace [10].

Tracking performance requires tracing services through a distributed system. This can be data and computation intense. For example, Facebook collects 1 Billion Traces/day and that is just 5% of all traces within Facebook. Existing tracing systems operate in mostly non-real time batch mode (e.g. Facebook ~ 1 day turnaround to find problems). Developers use stream processing to query continuous data streams and react to important events (e.g. Apache Flink, Timely Dataflow). In this work we propose an algorithm to construct traces in close to real time. This will have an immediate benefit for cloud-based services as they will be able to proactive to find problems in, they

system. Since cloud-based system are tightly integrated with our day-to-day lives and having system go down can have catastrophic consequences.

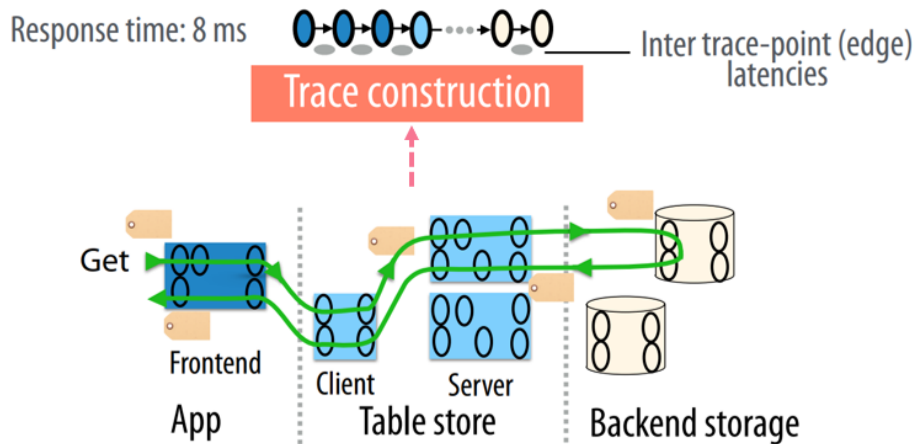


Figure 3. Workflow Tracing through a distributed system. Every request involves a workflow. The workflow shows the structure & timing of work done to process requests. It also captures the structure, order, concurrency & synchronization of the requests. [3]

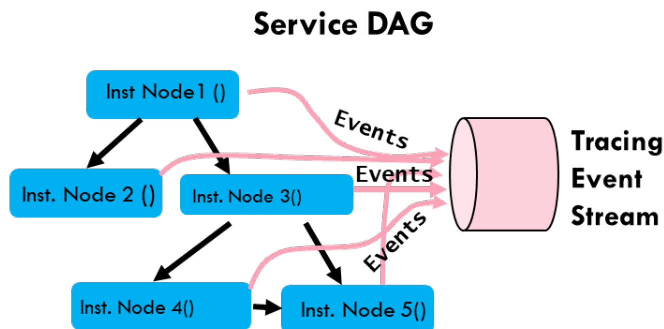


Figure 4. The instrumented points are part of a DAG (Directed Acyclic Graphs). The events are sent from the datacenter as an event stream. The challenge is to reconstruct the DAG in real time to understand the request flow.

Timely Dataflow

In this work we used Timely Dataflow (as shown in Figure 5) that is a common approach for distributed data processing because it explicitly encapsulates the boundaries between computations: the nodes of a dataflow graph represent sub-computations, and the directed edges represent the paths along which data is communicated between them. Timely Dataflow design is based on stateful dataflow, in which every node can maintain mutable state, and edges carry a potentially unbounded stream of messages. Although statefulness complicates fault tolerance, it is essential for low-latency computation. Incremental or iterative computations may hold very large indexed data structures in memory and it is essential that an application be able to rapidly query and update these data structures in response to dataflow messages, without the overhead of saving and restoring state between invocations. We chose to require state to be private to a node to simplify distributed placement and parallel execution. One consequence of adopting stateful dataflow is that loops can be implemented efficiently using cycles in the dataflow graph (with messages returning around a loop to the node that stores the state). In contrast, stateless systems implement iteration using acyclic dataflow graphs by dynamically unrolling loops and other control flow as they execute. In this work we used the Naiad Timely Dataflow implementation [4].

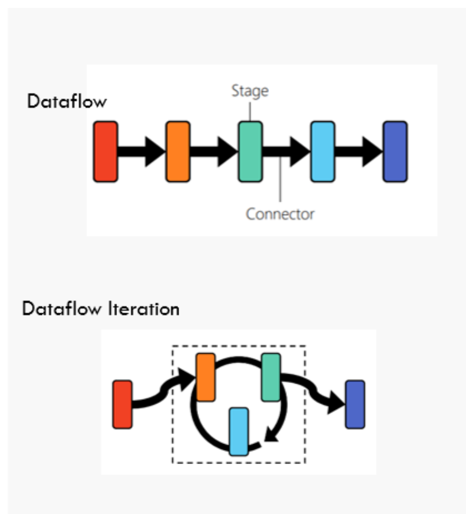


Figure 5. Timely Dataflow framework for writing dataflow programs. Dataflow programming is a programming model in which the computation can be represented as a directed graph: The data flows

along edges, while the computational logic in the vertices transforms it. The messages flowing along edges are annotated with timestamps.

Tracing Frameworks

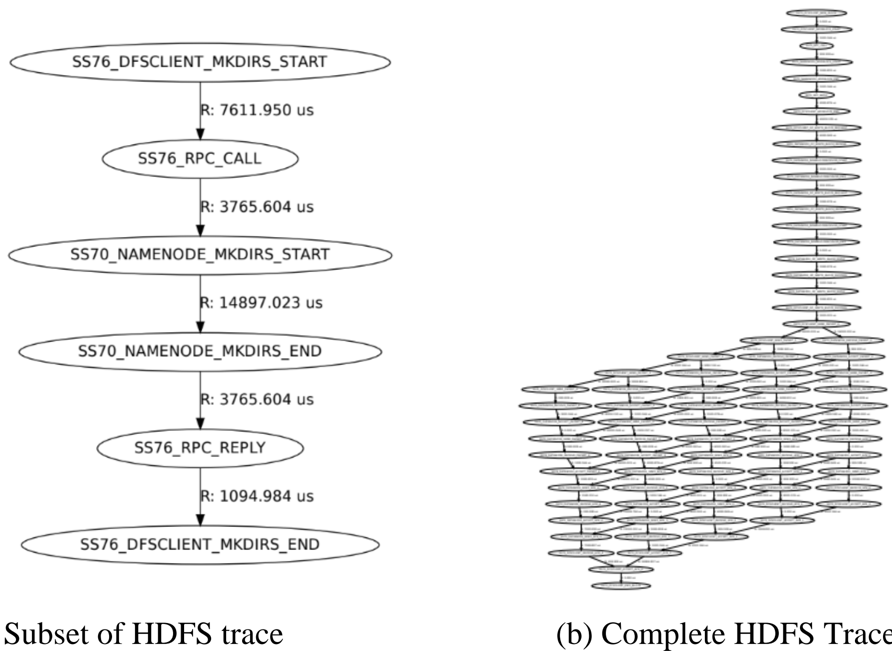
There are several tracing frameworks that have been implemented and have some different characteristics. Dapper [5] is Google's end-to-end tracing framework. Unlike other frameworks, Dapper instruments RPCs as single units, and thus provides a much higher-level view. It also does not explicitly capture sequential and parallel activity. Canopy is Facebook's tracing framework [1] does not propagate request identifiers and requires the developer to input a schema that defines the structure of the request flow graph. Stardust [2] and X-Trace [3] are two other frameworks which are very similar. Both propagate request identifiers and both explicitly capture sequential and parallel activity. All these frameworks work in batch mode. We are proposing a new Tracing Framework called Altair that would run in real time. To test out the framework we are proposing to use HDFS since it is easy to instrument HDFS to collect tracing data.

Hadoop Distributed File System

HDFS is the open source implementation of the Google File System (GFS). It is a distributed file system created with the purpose of storing large data for large scale data intensive applications. Like GFS, the architecture of HDFS consists of a single master, called the namenode, and multiple storage servers, called datanodes. Files are divided into fixed size blocks which are distributed among the datanodes and stored on the datanodes' local disks. Often, blocks are replicated across multiple datanodes for the purpose of reliability. The namenode stores all file system metadata information and file-to-block mappings and controls system-wide activities such as lease management, garbage collection, and block migrations between datanodes. As with most file systems, two common operations in HDFS are reading and writing files. To read a file, a client

requests the locations of the blocks of the file on the datanodes from the namenode. To write a file, a client requests datanodes from the namenode that it should send blocks to.

The purpose of adding tracing support to HDFS is to enable use of tools which analyze request flow graphs. In the tracing approach that we are proposing we will be comparing request flows across periods in order to find the root causes of behavioral differences between them. For performance diagnosis, it would be used to compare the request flows of a non-problem period, a period in which performance of the system is within expectations, and a problem period, a period in which performance is worse than expected. We will do this by clustering the graphs from non-problem periods into representative DAG's. Any DAG that does not correspond to the representative DAG can then be flagged for closer inspection.



(a) Subset of HDFS trace

(b) Complete HDFS Trace

Figure 6: Example request flow graph of a mkdirs RPC in HDFS. A client made a request to create a new directory and optional subdirectories in the HDFS instance. As a

result of the call, an RPC was issued and sent to the namenode, the master node in HDFS. The namenode did the required processing and returned control to the client. HDFS graphs can be very large. [6]

3. PROPOSED METHOD – ALTAIR MODEL

Our solution to the end to end tracing problem is shown in the Figure 7 below. The event stream (in our study we used HDFS streams but it can be applied to streams from any collection of services) from the datacenters are used to generate a Workflow model. The Workflow Model will be described later. The workflow model reconstructs millions of trace graphs. These graphs can then be queried to find system performance or to determine anomalous behavior.

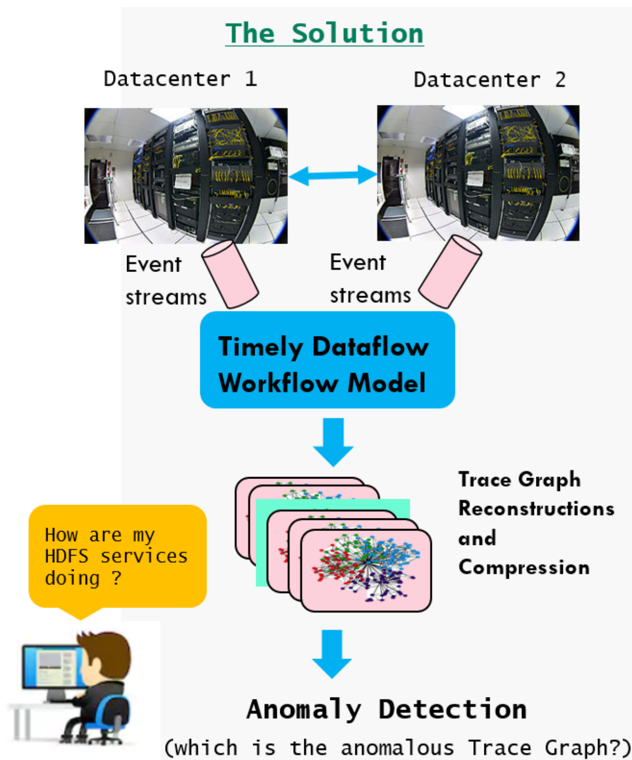


Figure 7. Proposed Method will take an input event streams and will reconstruct trace graphs that can be used for compression and analysis. The trace graphs can be compared to see if there are any mutations to identify any anomalies.

The Workflow Model is described next. The Workflow model is a new approach that we use it to perform tracing that can reconstruct traces in close to real time. We call our Workflow Model - **ALTAIR**. The Altair system is built on top of the Naiad Timely Dataflow open-source code and uses a cloud infrastructure to scale its computations to meet the real time requirements. The Altair System can represent traces both as trees and DAG.

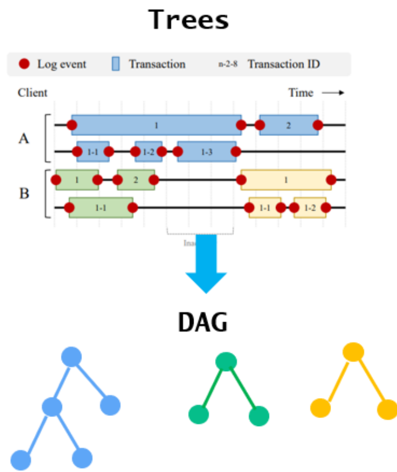


Figure 8. Representing traces as DAG instead of trees would allow more request flow being represented (e.g when two computation thread are messaging with each other and merge together, this flow cannot be captured with a tree representation). [12]

Figure 9. Workflow model that we developed for processing traces called Altair

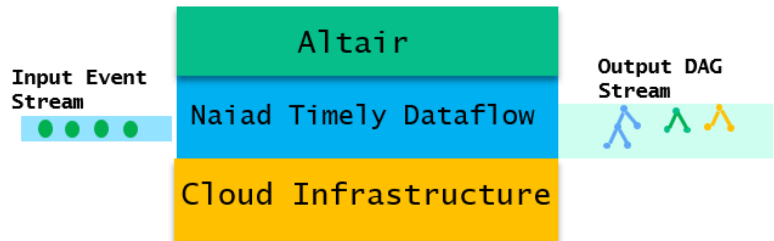
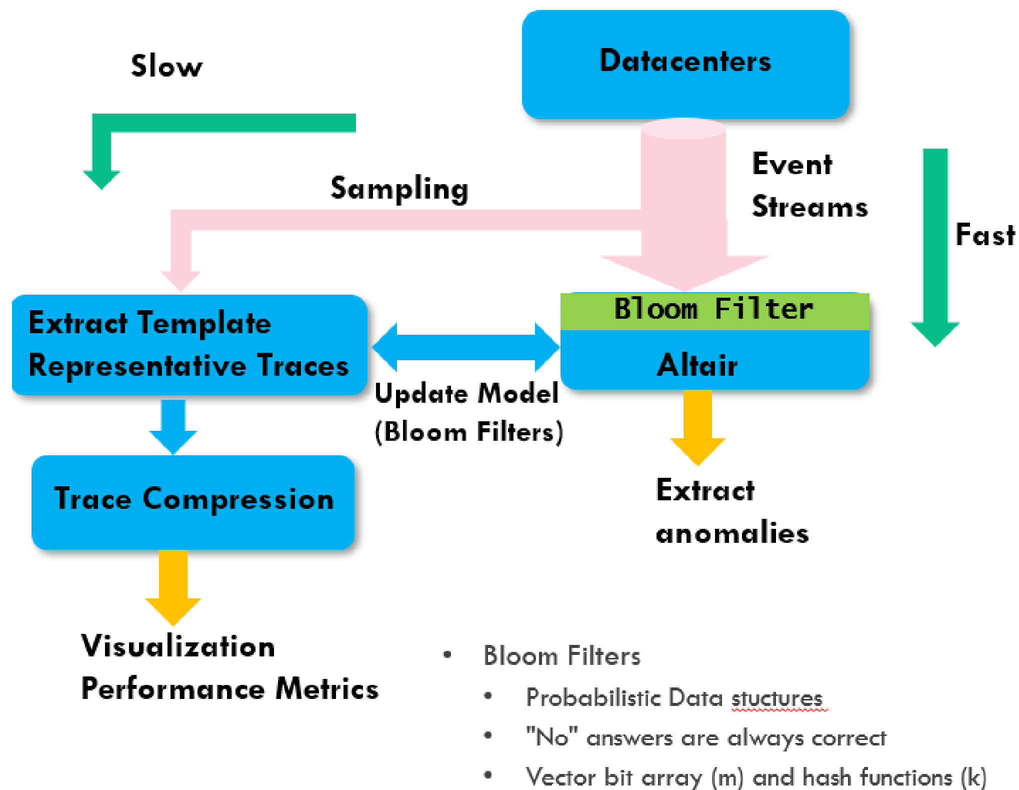


Figure 9 shows the Altair Model. The input event stream captures the logs of the instrumented data that is input to the system. The timely Dataflow allows for rapid graph reconstruction and is extremely scalable. The dataflow graph starts by taking the input event and checking the session id. It is then mapped and passed to the worker who is responsible for the graph (via a hash function), and collected. The graph edge is then reconstructed into the aggregate graph and saved until all edges have been received. There is a final trace packet which terminates the collection for the specific trace id and transfers and notifies the program interpreting the graphs. The dataflow graph also features expiry time safeguards which drops fractured traces due to trace points not reaching the input stream for reconstruction.

The Altair Model can be used in various application for e.g. Anomaly Detection, Performance and root cause analysis. In this work we will apply the Altair Model for Anomaly detection and will be described next.

The Altair model that are proposing will be run continuously in two steps. The first step involves designing the Bloom filter. The design of the Bloom filter requires a representative set of graphs. This step is not real-time. We are proposing graph clustering approach to extract template representative graphs that would be programmed into Bloom Filter. Second Step with Timely Dataflow will run in real time. Any anomaly traces will be flagged by the Bloom Filter with little overhead. The Altair



Model for Anomaly Detection is shown in Figure 10. The Model shows the input stream is split between a fast and slow path. The slow path does a complete reconstruction of the graphs. The fast path is programmed using representative traces using Bloom Filters that are extremely fast. Traces that have been encountered before can be rapidly recognized using the Bloom filter. Any trace that has not been encountered before will be flagged as an anomaly. This will allow for anomalies to be flagged in close to real time.

Figure 10. Anomaly detection using the Altair Workflow Model.

Bloom Filters are probabilistic data structures that allow (1) Insert an element into a set (2) Allow query whether an element is in a set. The gotcha with Bloom Filters is that when the answer to the query is "yes", it can be wrong. "No" answers are always correct

Bloom Filters can be described using a Vector bit array (m) and hash functions (k). They have a complexity O(k), and are very efficient. They are applied in cache and browsing

In our model we will train Bloom Filter using Template/Representative Traces. Any trace not recognized by Bloom Filter is then an anomaly trace (false positives). They can be coupled with Timely Dataflow.

4. Results

We ran the evaluation of the algorithm on Mass Open Cloud which runs OpenStack.

We had access to 10 compute instances, Altair runs on 8 instances, 2 instances run Trace compression. We instrumented HDFS and used X-Trace server [10] to collect the input event stream data. We collected 3000 Traces, with about ~350 graph nodes/traces, this translated to around 0.525 Million event/Epoch.

We used a streaming simulator to generate event streams that can replay the event streams to test the scalability of the system.

```
X-Trace Report ver 1.0
X-Trace: 194C79295549A3866ECCC315DD47301EA7
Host: ww2-laptop
Agent: Namenode
Label: WW2-LAPTOP_NAMENODE_MKDIRS_START
Edge: E7EB8787515DA35E
Timestamp: 181854865781004
```

Figure 11. This is a sample HDFS event in X-trace format that is captured by the system. This event is part of a large graph that would need to reconstructed. The implementation blocks are shown in the Figure. The trace simulation takes the traces and

replays them at line speed. In our testing we found that we could replay the data at around 0.25 Million Events/second. Some of the bottleneck associated with the system was unrelated to the Altair and was a bottleneck in the Redis logging server. [10]

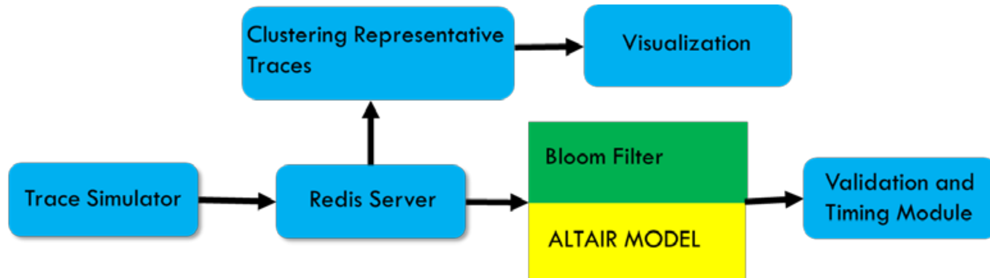


Figure 12. Implementation Block diagram of the Altair Model.

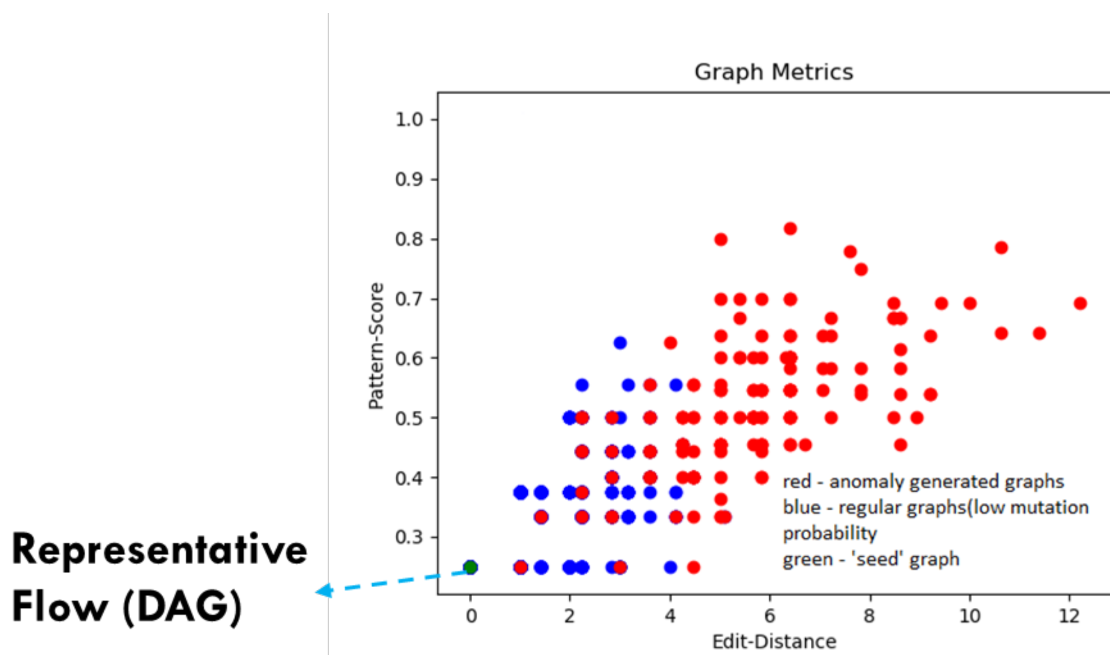
Figure 12 shows the implementation blocks of the Altair System. The Trace Simulator output synthetic traces in X-trace format and is implemented in Python. The event output speed of the simulator is 0.25M events/sec. The clustering Module performs feature metric extraction. Compression Metric is implemented in Python and use the GraphZip Libraries. Timely Dataflow framework is based on a public domain implementation in Rust. Bloom Filter is implemented in Rust. Validation and Timing module implemented in Python. Visualization is implemented on python using the matplot Library.

Extracting Representative Flows

There are numerous techniques available to determine graph similarity. Graph similarity methods can be generally divided in three categories. (1) Graph Edit Distance (Graph Isomorphism). It measures the distance (dissimilarity) of given graphs g1and g2 and is based in the idea of editing g1 into g2. Common edit operations are deletion, insertion and substitution of nodes and edges. It is computationally expensive, but approximate solutions with complexity O(n3)

(2) Graph Feature Extraction. The key idea behind these methods is that similar graphs probably share certain properties, such as degree distribution, diameter, eigenvalues, information metrics like Hoffman Coding and Minimum Descriptor Length (MDL) Score [9]. After extracting these features, a similarity measure is applied in order to assess the similarity between the aggregated statistics and, equivalently, the similarity between the graphs.

(3) Iterative methods for Graph Similarity. The key idea behind the iterative methods is that "two nodes are similar if their neighborhoods are also similar". By performing random walks in the neighborhood, we can identify graphs, can be couple with Machine Learning. Clustering is another iterative method that can be used with graph features to



group similar graphs.

Figure 13. The representative flows can be identified using clustering of DAG features.

In Altair we use three graph metrics to iteratively cluster the graphs on the slow path. The three metrics we use are Huffman Score, String Edit Distance and MDL Score [8]. Figure shows the results from the clustering approach. The Green Dot in the representative DAG that was identified in the clustering. Since most of the graphs are not

anomalous the green dot signifies ~ 3000 graphs in these figures. The representative graphs are programmed as Bloom Filter and updated on the Altair Model for Anomaly detection. The DAG clustering is only performed on the slow path and on a very small subset of the complete stream (<5%) and hence the cost of extracting the representative DAG Flow will not impact the overall

All the representative graphs are similar to each other and hence the green dot is the representative graph. All the representative graphs are cluster and overlay on each other.

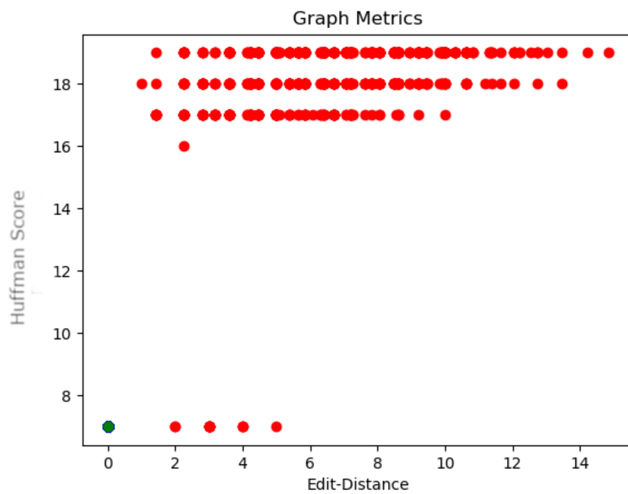
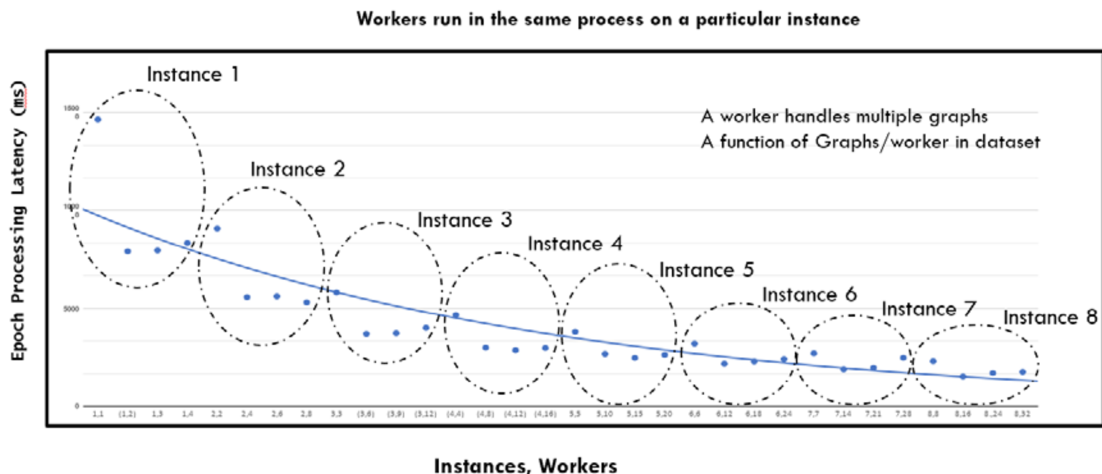


Figure 14. The representative flows can be identified using clustering of DAG features.

Anomaly Detection Results

We will now present the results for the Altair Model for Anomaly Detection shown in Figure 10. The Representative Flows are programmed to design the Bloom filters. In Figure 15 we present the processing latency for anomaly detections as we increase the



number of cloud compute instance. The latency is functions of instances and workers.

Figure 15. Epoch Processing Latency of the Altair Model as we increase the instances and Workers.

The number of workers takes advantage of the threading efficiencies in the compute node. Adding additional workers beyond 1 does not decrease the processing latency.

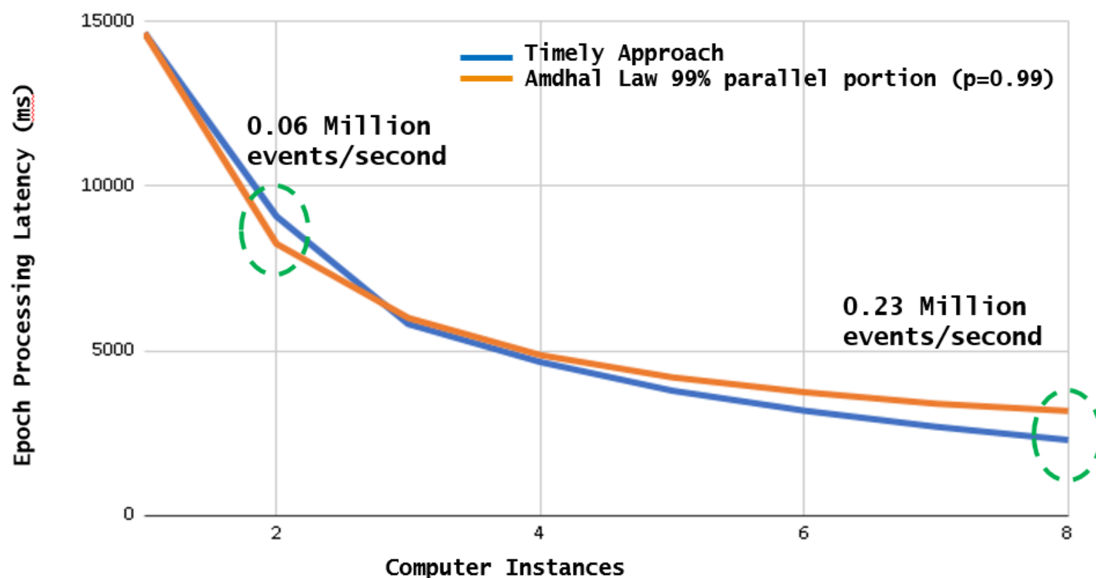


Figure 14. Processing Latency as a function of computer instances for the Altair Model.

We can fit the epoch processing latency to the Amdhal Law (Figure 14). Amdahl's law is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. The Amdhal law overlay with the Altair Timely Approach shows that the Altair performs

better than a system that is more that 99% parallel. This shows that the Altair model is highly scalable and as the event stream gets bigger, more computer instances can be added to meet the computations requirements. With 8 instances Altair was able to attain a throughput of 0.23 Million events/second.

Figure 15 shows the throughput of the Altair Model. It also shows the Amdhal Law with p=0.99 that is overlaid on the chart. To put this result is perspective, Facebook collects 11 Million events /sec. With enough computer instances the Altair model can be scaled to be used in a production environment. This model is a significant improvement to the offline batch processing approach to reconstructing graphs. The efficiency is gained in this system by the use of Bloom Filter to speed up the graph reconstruction step if it belongs to the representative flow set.

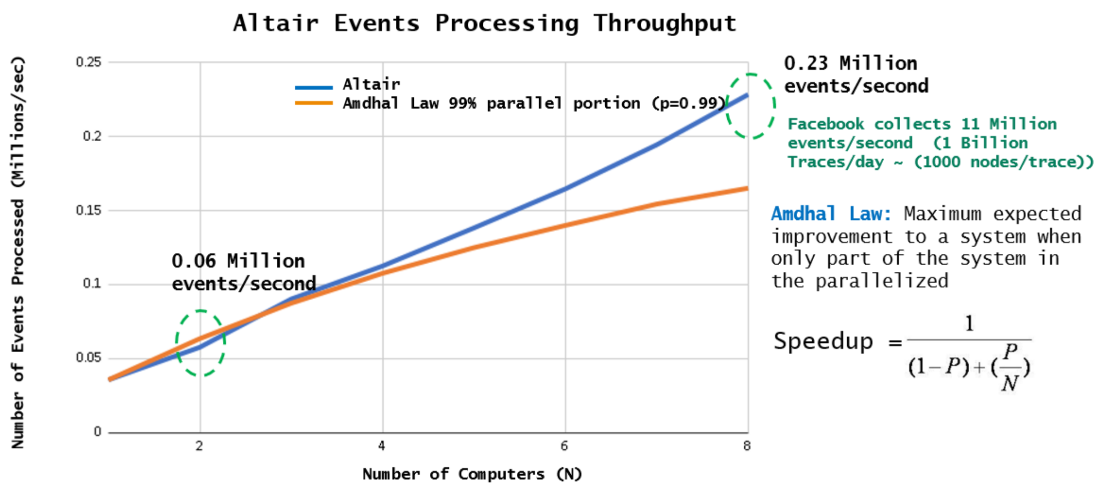


Figure 15. Throughput processing for the Altair Model.

We will next present some discussion of the results.

5. Discussion

This section discusses our results and some of the challenges we had to overcome to develop the Altair end-to-end tracing framework.

Low overhead

Ideally, tracing framework should always be running, even in a production system. This is so that, if a performance problem were to be detected, one can immediately look at the resulting traces as they have already been collected, which then speeds up the diagnosis process. Otherwise, the process will be much slower as it would require tracing to be enabled, traces to be collected, and then tracing to be disabled. In doing so, one may have to wait until the problem appears again, which could happen again in some arbitrary time in the future. However, always-on tracing is only achievable if the framework incurs a low performance overhead since few are willing sacrifice performance in their systems. The Altair Workflow Model has extremely low overhead. It is able to achieve the low overhead in three areas (1) It uses Timely Dataflow Stream processing with is very efficient in reconstructing DAGs. (2) It is programmed in Rust which has very low overhead processing for a programming language. (3) It breaks up the event stream into 2 parts, the slow path and the fast path. The slow path extracts representative flows from the stream. Since most of the graphs are not anomalous, DAG clustering is able to identify representative flows with a very small subset. (4) The fast path of the event stream using Bloom Filter hashing. The bloom filter are very efficient data structures for k Bloom Filter, they can be run in O(k). In addition, they are space efficient. This approach is absolutely new and is more efficient than the batch processing methods that are being used.

Detailed Low-level Instrumentation

In this project we have instrumented the services at a fine granularity. For example, we have instrumented at the HDFS packet level for the writes. The reason for this is to expose as much detail as possible to the developer. Also, in this way, tools can be more

precise about the locations of the mutations in a request category, possibly helping to pinpoint the root cause faster and more precisely. Another benefit of low-level instrumentation is its ability for its traces to be aggregated to create a higher-level view if necessary. Essentially, by instrumenting at the lowest level, we can expose information at any level above it. Thus, such a framework can be used to capture problems both at the micro and macro level. By performing clustering and sub graph mining using MDL Score on the representative workflow we can visualize the graphs at multiple granularity levels.

Large Graphs and Large Number of Graphs

As a result of the detailed instrumentation, some graphs are very large. This occurs in HDFS when capturing the behavior during the execution of a HDFS block write request. This graph is the result of capturing every packet send iteration, each of which asynchronously waits for an acknowledgement. Depending on how many iterations the operation must perform to complete the task, the graph can grow very large very quickly. This is a problem for diagnosis tools as they are generally not written to efficiently handle graphs of that size. Some tools will run out of memory attempting to interpret the large number of graphs. By extracting representative workflows, we only need to generate small numbers of large graphs and we can very efficiently visualize any problems that we see in the event stream without have to go through all the trace graphs (needle in the haystack problem).

6. Conclusion

Diagnosing problems in large scale systems using cloud based distributed services is a challenging problem. The micro-service architecture can cause latency and anomalies in performance. With so much of our everyday lives and many critical applications closely linked to the cloud and not having access to the data or service can in some case be catastrophic (e.g. hospital not able to access patient records that are stored in the cloud in a timely manner). We presented a new model called Altair to capture workflow-centric tracing dependency graph of causally-related events among the components of a distributed system. Altair model is able to contract the traces in real time and is a big improvement to approaches that have historically been performed offline in batch fashion. The real time approach will have the trace data immediately available to engineers for their diagnosis efforts and improve the performance of critical applications. The Altair Model is highly scalable and can be scaled to the production environment of even the big service providers. In this project we tested the system for the Hadoop File System which is a component in all distributed systems, the trace reconstructed is agnostic to the underlying service and can be applied to any collection of services. The Altair approach is based on graph abstraction and streaming framework and will provide the network operation with a real time understanding of the distributed system behavior.

References

1. Canopy: An End-to-End Performance Tracing And Analysis System, Kaldor et al, SOSP '17, October 28, 2017, Shanghai, China
2. Principled workflow-centric tracing of distributed systems, Raja R. Sambasivan, et. al, In Proceedings of SoCC 2016
3. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. Raja R. Sambasivan, et. al, IEEE Transactions on Visualization and Computer Graphics (Proc. Information Visualization 2013), Vol. 19, no. 12, Dec. 2013
4. Naiad: A Timely Dataflow System, Derek G. Murray, et. al., SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania
5. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Benjamin H. Sigelman, et. Al, Google Technical Report, 2010 https://research.google.com/archive/papers/dapper-2010-1.pdf
6. Diagnosing performance changes by comparing request flows, Raja R. Sambasivan, et. Al., Proceedings of NSDI 2011
7. GraphZIP: a clique-based sparse graph compression method, Ryan A. Rossi, et al., Journal of Big Data, December 2018
8. Mining of Massive Datasets, Jure Leskovec, Book, 2010
9. I. Jonyer, L. Holder and D. Cook, "MDL-Based Context-Free Graph Grammar Induction and Applications," International Journal on Artificial Intelligence Tools, 13(1):65-79, March 2004.
10. R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. XTrace: a pervasive network tracing framework. NSDI '07.
11. Medium Article, How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play, https://medium.com/

12. Chothia, et. all, Online Reconstruction of Structural Information from Datacenter Logs, EuroSys ’17