QueryOT: Query-Driven OpenTelemetry Extension

1 Introduction

Our idea is inspired by Snicket [4], a query-driven distributed tracing tool integrated with microservices frameworks. In Snicket, the user is able to issue an SQL-like query that defines what attributes should be recorded in the trace when certain happens-before relations between microservices is matched successfully. Snicket achieves its functionalities by compiling the query down to a WebAssembly code that can be run in proxies, like Envoy [1]. Yet such implementation imposes relatively high latency overheads since, as the author states, it is costly to get into and out of the WASM runtime. So we design and implement a light-weight filtering mechanism for traces with similar functionalities in OpenTelemetry [3], an open-source observability framework, to achieve more expressiveness and see possible performance improvements. Our solution evaluated to be imposing moderate overhead, and can even save time when filtering out most traces, compared to full tracing exportation without filtering.

2 Background

Query-driven tracing like Pivot Tracing [5] and Snicket [4] lead a new paradigm in tracing: only collecting the traces satisfying the condition, i.e., what the programmers need, while generating full traces. This is promising since works dated back to Dapper [6] have been claiming the generation cost is low, while the storage and transmission cost is high. However, uniform trace sampler faces the problem of tail cases. To tackle this, one approach is retroactive tracing like what HindSight [7] does, holding the trace data until making sure no fault has happened. But this approach also faces the memory resource issue, and it needs to limit the CPU occupancy of the independent eager-polling agent. Another approach is query-driven tracing like Snicket, but it incurs a 30% overhead which is relatively high for today's performance requirement. Nevertheless, we do think query-driven tracing is an interesting way to achieve efficient and precise tracing, so we decide to design and implement the feature into OpenTelemetry, to make it light-weight.

3 QueryOT Design

3.1 Challenges and Solutions

To achieve the query-driven tracing in a instrumentation library, there are some key challenges to overcome:

1. *Where to filter out traces? Exporter or Sampler?* Exporter in OpenTelemetry is a combination of marshalling and client of collecting service. And Sampler acts in a head-sampling manner, providing ratio-based and parent-based sampling by default. Due to the nature of query, we need to make decisions until all necessary information has been gathered. This can only be guaranteed at the position of Exporter. Therefore, we need to hijack the exporting logic instead of implementing another independent Sampler.

2. *How to achieve live updating in a library?* Engineers may update their query as the problems evolve or change. Under the microservices context, an immediate solution could be launching a new instance with fresh Environment Variables, which is easy to configure by updating the Kubernetes YAML file for the application, and is transparent to the app. However, frequent launching and shutting down requires correct and efficient migration mechanism, and incurs relatively high expense. Unlike Snicket in Sidecar, implementing filters in a library means we don't have a routable address, nor should we create an extra process and occupy too much computation resource. Thus, we give up some very moderated application transparency: requiring the application to setup an RPC endpoint that calls our pre-defined handler, which will instantly take over the query update logic.

3. *Which component is responsible for maintaining the set of rules?* It might be natural to hold the rules in the implementation of Exporter, since it is responsible for

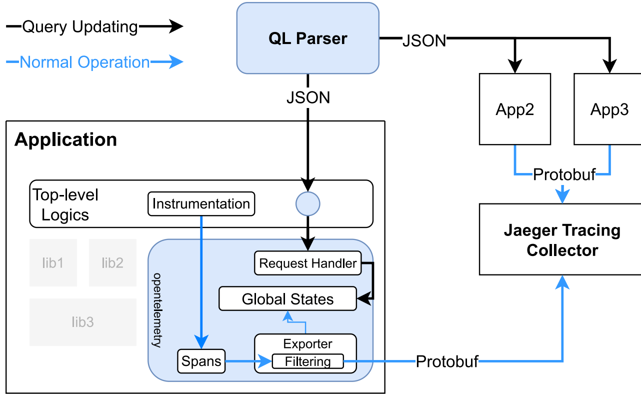

Figure 1: Architecture of QueryOT

enforcing them. However, the project structure prevents us to do so directly. First, the Exporter hides the internal states and functions, so we cannot access it from application code. This lead us to set up a Global State Manager, which delegates the states pool. It is already in use, as OpenTelemetry leverages this structure to maintain Baggage Propagator and TraceSDKProvider. In this way, both the application and Exporter can touch the information. However, the states pool still can't be implemented as an object of Exporter Module, since the global manager is also not able to access internal states. Therefore, we choose a place having most to do with Attributes, Keys and Values pairs, because our filter rules are of similar form: either a key, a key-value pair, or a key and two boundary values. The *attribute* module of OpenTelemetry, used as a essential building block by many other modules, so implementing the rule holder here will incur least dependency issue. The final solution is to implement the actual rule holder in a basic *attribute* module of OpenTelemetry, but instantiate it in a wrapper, namely Global State Manager.

4. *How to ensure thread safety?* Introducing another global states inevitably brings thread safety concern. Consider the nature of trace exporting. Accessing the set of rules is much more frequent than updating the rules. We therefore add a Read-Write lock (the RWMutex in Go), to protect thread safety while accommodating multi-threaded user serving.

3.2 Architecture Overview

Figure 1 shows the architecture of QueryOT. Blue arrows indicate the data flow in normal operations. In handling a request or doing normal computation, the instrumentation code will be called, and create spans as internal states of OpenTelemetry library. OpenTelemetry may export spans in a micro-batch manner, namely a *Scope* of spans to centralized collectors, such as Jaeger [2]. Our solution hijack this procedure, and enforce the library to match attributes and ranges before marshalling the spans. If an attribute doesn't satisfy the name requirement or value requirement, it will be discarded and not marshalled. Under the full trace filtering mode, if not all the filtering rules are satisfied, the whole trace is dropped.

Black arrows draws the data flow in live updating of filtering rules. After the engineer issue a query, the frontend QL Parser will parse it into JSON files and distribute them to corresponding services. From the application perspective, upon receiving a update request, it will immediately pass the request to the handler provided by the library (which is implemented by us), with no worry of dealing with details. As the library takes over the request, it will refresh the global states according to the contents, preparing it for the next access from Exporter.

3.3 Frontend

We adopt the sqlparser package from the Vitess system, which supports a full-fledged MySQL grammar. We choose SQL, with a rather limited set of grammar, since it , though might not be the most intuitive language, is one of the most widely used one. The following inputs are supported:

```
01 |  SELECT <*>|<app1.attr1[,app2.attr2]>
02 |  FROM <app1>
03 |  [JOIN <app2> ON app1->app2]
04 |  [WHERE <attr1><"="|">="|"<="|"<="|">"><
          value> [AND attr2>=lb AND attr2<=rb]]
```

Listing 1: QueryOT supported query interface

For instance, the user can input such a query:

```
01 |  SELECT Ads.brand, UserInfo.id
02 |  FROM Ads, UserInfo
03 |  WHERE UserInfo.id < 2000 AND UserInfo.id >
          27 OR Ads.length = 10.5
```

Listing 2: A sample query involving two services

In this query, only the attributes from the two services, Ads and UserInfo, will be collected. The WHERE clause defines that only the traces with a UserInfo.id between 27 and 2000 or with ads whose length is equal to 10.5 minutes will be exported. As a result, three attributes will appear in the traces: Ads.brand, UserInfo.id, and Ads.length (for the convenience of jsonfying, all of the attributes that are used in the WHERE clause are included automatically). The parser will extract all the information and summarize it into two JSONs in such form:

```
01 |  {
02 |      "filters": [
03 |          { "key":"length", "type":"float64",
            "values":[10.5]},
```

```
04 |           // Here the type is invalid for the
         parser cannot infer its type
05 |           { "key":"brand", "type":"invalid",
        "values":[]}
06 |        ]
07 |    }
```

Listing 3: JSON for service *Ads*

```
01 |    {
02 |        "filters": [
03 |           { "key":"id", "type":"int64", "
        values":[26,1999]}
04 |        ]
05 |    }
```

Listing 4: JSON for service *UserInfo*

3.4 Backend

The JSONs above will then be sent to every service. One for microservice Ads, and the other for microservice UserInfo. Handled by the *HandleRequest* callback function in each service, the filtering rules are defined with the JSON: such operation of receiving a set of new rules is *Update*. *Update* operation will create new filter or overwrite the existing filter. There are two other operations: *Remove* and *Clear*, which are as well requested via similar service interaction. *Remove* operation is accompanied by a list of strings determining which filters to be removed. And *Clear* remove all filters in a batch, which means stop collecting from this application.

The rules are saved in each service as "Global states". Once there is a trace goes through the service, the exporter is responsible for the actual filtering during which the states maintained will be matched with the attributes in the exporter. If all the applicable predicates are said to be true for the corresponding set of attributes in the application, then the trace can be successfully exported, otherwise discarded. It is worth noting that while the trace is not filtered out, OT would only export those selected attributes and those used in predicates.

To support flexible usage, we add some mode flags for user to choose from. OpenTelemetry can then be initialized with either, both or none of Full Trace Filtering and Attribute Filtering. The scenario above describes what will happen when both modes are enabled. Full Trace Filtering means that if not all filter rules are matched, the whole trace will be discarded. And Attribute Filtering Mode means that if a trace is to be exported (regardless of whether Trace Filtering is enabled), all attributes that don't match the filtering rule will be discarded, but the trace itself is kept.

We also design an Event Filter, leveraging our filter pools. *Event* in OpenTelemetry can be viewed as a string (Event Name) with a timestamp, so it easily fits into our previous model.

```
Single Process Single Thread:
No Filtering:       22.62s user 18.18s system  96% cpu 42.382 total
Filtered Out 93%:   21.17s user 17.08s system  96% cpu 39.716 total
Filtered Out 0%:    22.88s user 18.00s system  95% cpu 42.696 total

Multiprocessing - Parallel 3:
No Filtering:       11.65s user 7.57s system  200% cpu  9.578 total
Filtered Out 93%:    5.78s user 3.67s system  100% cpu  9.420 total
Filtered Out 0%:     5.84s user 3.86s system  100% cpu  9.676 total
```

Figure 2: Preliminary performance evaluation results

4 Implementation

We implement QueryOT Backend based on opentelemetry-go, and a demo application fully utilizing our features. And we implement Frontend also using Go, with *sqlparser*. We implement around 1000 lines of code in total, and open-source them at GitHub Repo 1 and Github Repo 2.

5 Evaluation

Here are some preliminary evaluation results, tested on a Linux Laptop equipped with AMD Ryzen R7-5800H CPU and 16GB RAM. Both Single Process and Multiple Process request mocking script can be viewed at GitHub. Figure 2 shows the time consumption in handling 10000 requests for calculating Fibonacci's numbers. And we observe the following findings:

1. QueryOT saves 6.3% end-to-end running time when filtering enabled, compared with full exportation. (Time saved = Exportation Time Saved - Filtering Overhead)
2. QueryOT imposes 0.74% end-to-end overhead when filtering is enabled but no spans were dropped. (Pure Filtering Overhead)
3. With multi-threaded enabled, the saving and overhead is 1.65%, 1.04% respectively.

References

[1] Envoy proxy. https://www.envoyproxy.io/.

[2] Jaeger. https://www.jaegertracing.io/.

[3] Opentelemetry. https://opentelemetry.io/.

[4] Jessica Berg, Fabian Ruffy, Khanh Nguyen, Nicholas Yang, Taegyun Kim, Anirudh Sivaraman, Ravi Netravali, and Srinivas Narayana. Snicket: Query-driven distributed tracing. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *Commun. ACM*, 63(3):94–102, feb 2020.

[6] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[7] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing edge-cases in distributed systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 321–339, 2023.