QueryOT: Query-Driven OpenTelemetry Extension

Qiutong Men, Xiyu Hao

Introduction

Snicket[1] is a query-driven distributed tracing tool integrated with microservices frameworks. However, Snicket imposes relatively higher latency overheads due to many reasons, and currently cannot deal with in-app attributes. We tend to implement similar functionality in OT for more expressiveness and see possible performance improvements.

Challenges & Design Choices

Filtering at Exporter v. Sampler:

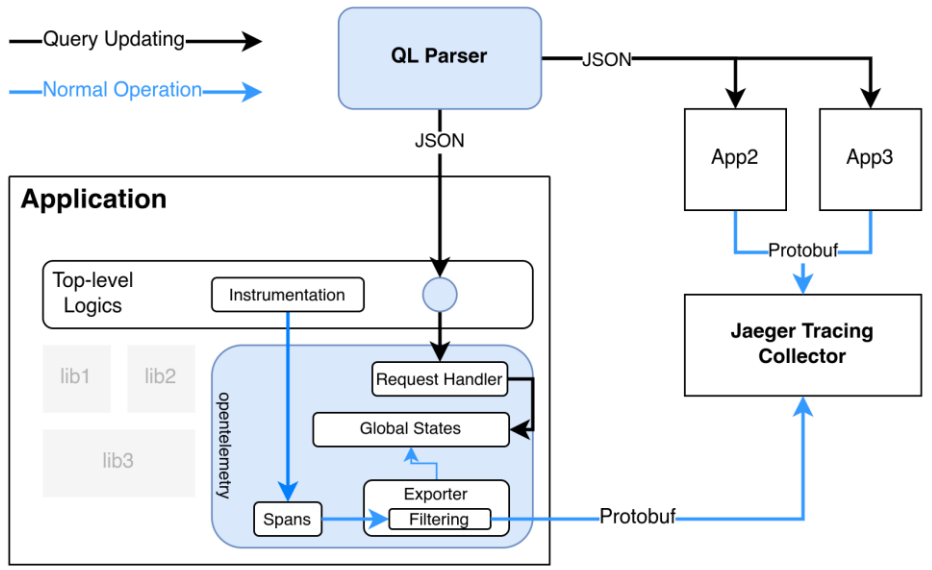
- Exporter: A client of trace collector(e.g., Jaeger)
- Where it is guaranteed all information in one trace has been generated and gathered.
- Sampler: Act in head-sampling manner. Doesn't touch trace contents and only operates with ctx.

Thus, we implement filtering at Exporter.

Live Updating Queries:

- A frontend parser is necessary, translating QL into configurations to distribute to instances.
- An immediate solution: let the orchestrator launch a fresh instance with the new query encoded as ENV variables, and gradually take over the REQs.
- Refreshing the query by launching and shutting down services every time is expensive.
- However, as a library embedded into application, achieving live updating isn't as free as in Sidecar: we don't have an independent process aside.

Thus, we give up some very moderated application transparency: requiring application to setup an RPC endpoint that calls our pre-defined handler, which will be accessed by the parser/distributor.



Enable OpenTelemetry with Query-Driven Capability & Live Updating.

Reference

[1] Jessica Berg, Fabian Ruffy, Khanh Nguyen, Nicholas Yang, Taegyun Kim, Anirudh Sivaraman, Ravi Netravali, and Srinivas Narayana. 2021. Snicket: Query-Driven Distributed Tracing. In Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21). Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/3484266.3487393

Implementation

Around 900 lines in Go.

QL Parser: We build a parser with a reduced API:

```
SELECT <*>|<app1.attr1[,app2.attr2]>
FROM <app1>
[JOIN <app2> ON app1->app2]
[WHERE <attr1><"="|">="|"<=""><v1> [AND
attr2>=lb AND attr2<=rb]]
```

Structural Filtering is left as future work.

Exporter Augmentation: We implement the filtering logic in the OTLP exporter, an exporter with widely accepted protocol, of `opentelemetry-go`. We hijack the span marshalling procedure and discard the unneeded spans/attrs/events.

Global States: Where do the filter rules locate? Maintaining the rules as exporter's internal states would make it hard to update (from software engineering view), and cause synchronization issues if multi-threaded. We hold the rules in a global `AttributeFilterRuleHolder`, equipped with RW Lock, given that query updating happens much less frequently than trace exporting.

Evaluation

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

```
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
```