Exemplars - A Tale Of Pillars Supporting The Observability Structure

Vijay Samuel - Principal MTS, Architect
Jing Hu - Sr. Software Engineer

Pillars Of Observability

M.E.L.T

- Metrics
- Events
- Logs
- Tracing

In The Beginning…

In the beginning…

- Easy to generate
- Derive other forms of telemetry from it
- The more you spit out, the more you can learn about what's going on

```
2022-03-14 12:05:23.567 [INFO] User login successful for username "johndoe" from IP address 192.168.0.1
2022-03-14 12:10:45.123 [ERROR] Unable to connect to database. Error code: 500
2022-03-14 12:15:09.876 [DEBUG] Request received for endpoint "/api/v1/users" with query parameter "search=John"
2022-03-14 12:20:30.234 [INFO] Successfully processed 100 records in 2.5 seconds
2022-03-14 12:25:47.890 [WARN] Disk space usage is above 90%. Consider freeing up some space to avoid issues.
```

Log Volumes Grew

- More microservices ←→ More logs
- String processing burns CPU
- Easy to instrument - hard to make sense at scale

Then We Tried Adding Structure…

- Parsed using predefined patterns on “highly structured logs” or “events”
- Stored in columnar databases
- Supports “infinite” cardinality
- Allows ad-hoc OLAP style queries
- Simple to onboard - more powerful than plain old logs

Apache Access Logs

```
127.0.0.1 - - [16/Apr/2023:10:15:45 +0000] "GET /index.html HTTP/1.1" 200 876 10
127.0.0.1 - - [16/Apr/2023:10:16:02 +0000] "POST /submit_form HTTP/1.1" 302 0 5
192.168.1.1 - - [16/Apr/2023:10:16:45 +0000] "GET /images/logo.png HTTP/1.1" 304 0 3
```

```
remotehost rfc931 authuser [date] "request" status bytes responsetime_ms
```

FAQs

- Total number of GET request - 2
- Average request latency from 127.0.0.1 - 7.5ms
- Total number of requests - 3
- Total number of requests that were 2xx - 1

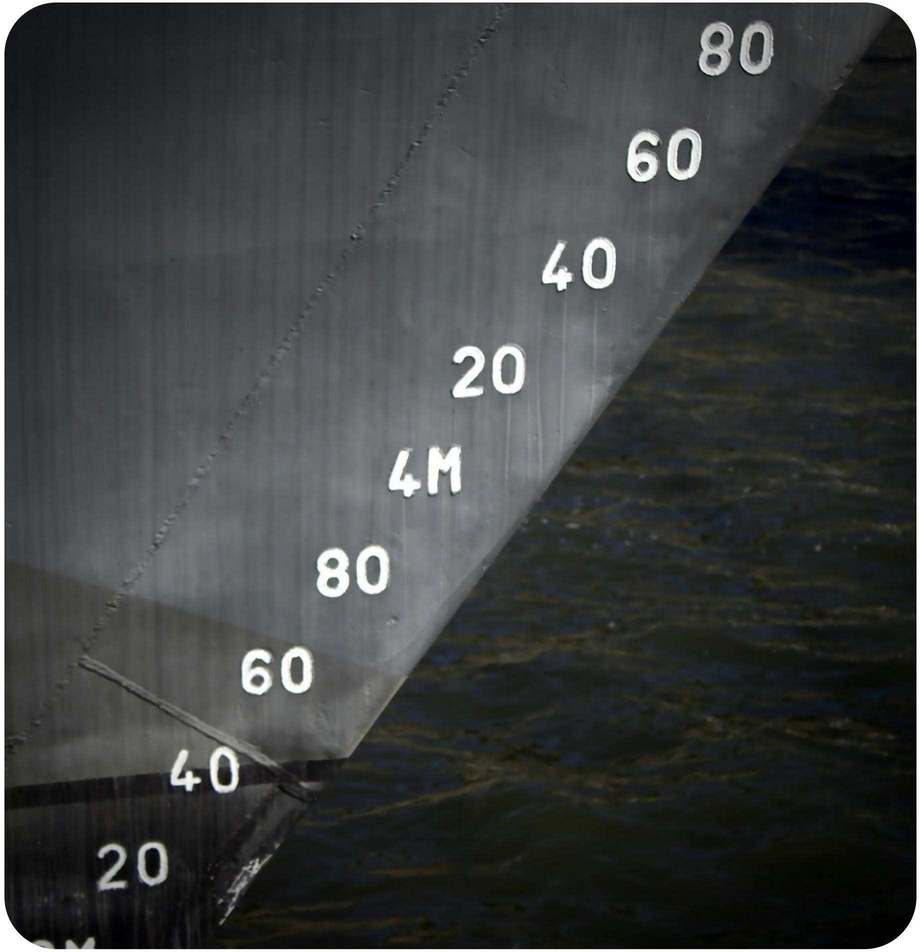
Some Problems Just Don’t Go Away

- CPU burn kills 5 engineers every year *
- Can become cost prohibitive at scale regardless how useful those debug logs are

* made that up

And Then Came Metrics…

- Numerical measurements
- Source instrumented
- Represented as **time series**
- Stored in optimized **time series databases**
- Aggregatable across time and space (cheaper)



Apache Metrics

```
# Count of HTTP requests by method, path, and status code
http_requests_total{method="GET", path="/foo/bar", status_code="200"} 1
http_requests_total{method="GET", path="/baz/qux", status_code="404"} 2
http_requests_total{method="POST", path="/foo/bar", status_code="200"} 1
```

```
# HTTP request duration histogram by method, path, and status code
http_request_duration_seconds_bucket{method="GET", path="/foo/bar",
status_code="200", le="0.005"} 0
http_request_duration_seconds_bucket{method="GET", path="/foo/bar",
status_code="200", le="0.01"} 1
http_request_duration_seconds_bucket{method="GET", path="/foo/bar",
status_code="200", le="+Inf"} 1
http_request_duration_seconds_sum{method="GET", path="/foo/bar",
status_code="200"} 0.0075
http_request_duration_seconds_count{method="GET", path="/foo/bar",
status_code="200"} 1
```

FAQs

- Total number of GET request - 2
- Average request latency from 127.0.0.1 - 7.5ms
- Total number of requests - 3
- Total number of requests that were 2xx - 1
- 99.9th percentile of request latencies - ???

He Who Shall Not Be Named…

- TSDBs suffer from cardinality restrictions
- High cardinality → slow queries
- High cardinality → higher in-memory costs

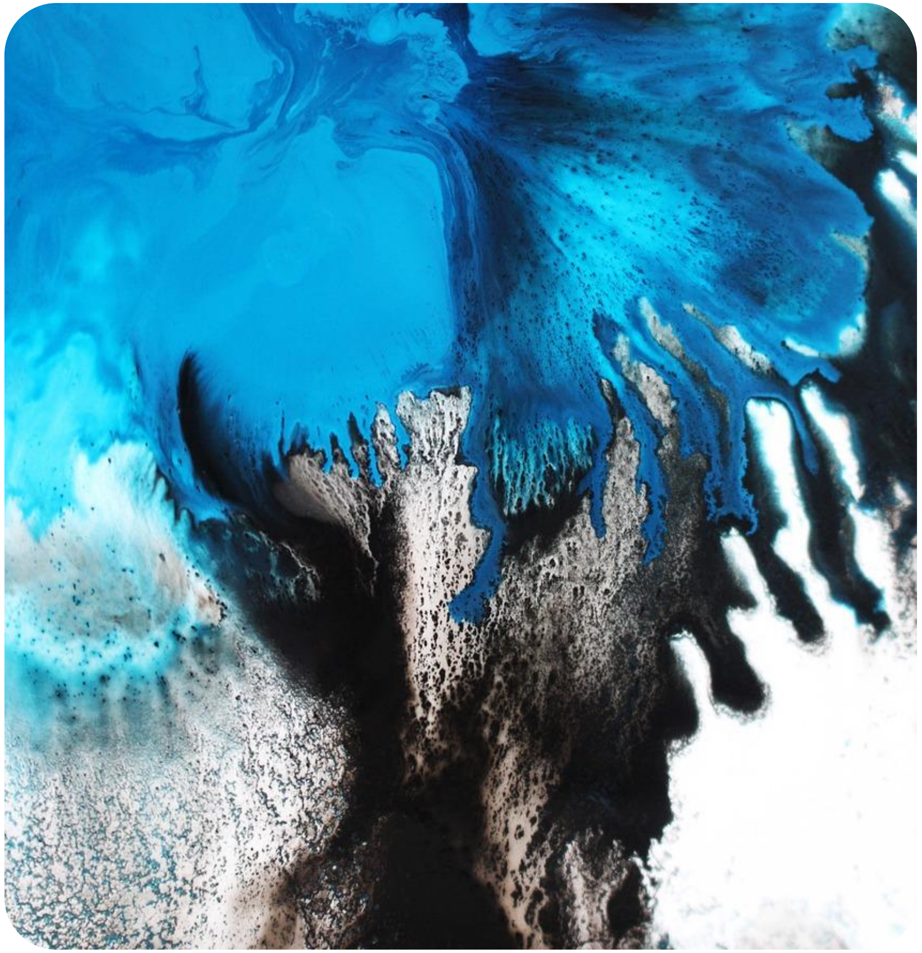
Now We Have Tracing…

- How did my the 10s of microservices behave as my request flowed through each of them?
- How do all my microservices tie together as a dependency graph?

Make M.E.L.T Meld!

Why Do That?

- Individual pillars by themselves aren’t “Observable”
- Traversal across pillars is essential
- Promotes better questioning of the data



Tracing ❤️ Logs

- Interoperability baked in by design
- Trace IDs and Span IDs emitted as part of every trace and logs
- Allows to jump between logs and traces in both directions
- Ensure logs are sampled along with traces

But What About Metrics

- Structured events can have measurements
- Span Events can also have measurements
- Can I store Trace IDs as metric labels?

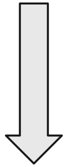
Enter Exemplars

What Are They?

Recorded value that associates OpenTelemetry context to a metric event within a Metric

- trace_id, span_id
- recorded value
- timestamp of observation

```
http_request_duration_seconds_bucket{method="GET", path="/foo/bar", status_code="200",
le="0.01"} 1
```



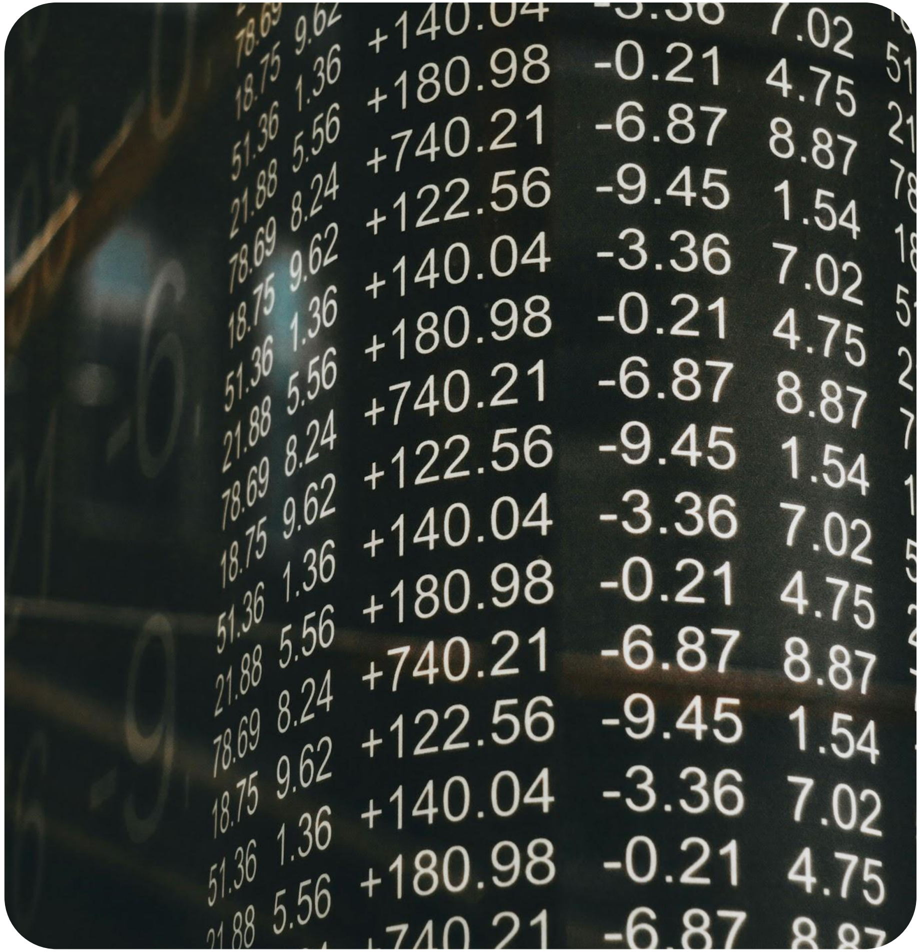
```
http_request_duration_seconds_bucket{method="GET", path="/foo/bar", status_code="200",
le="0.01"} 1 # {trace_id="120387341874", span_id="31498365"} 0.008 15209746.56
```

Why Are They Useful?

- Promotes full interoperability within all 4 pillars
  - metric ←→ trace ←→ log/event
- Ease of transition from aggregation to individual measurement
  - 90th percentile request latency → trace of a request with value higher than 90th

How Can I Use Them?

- Instrumentation - OTEL SDK, Prometheus Client
  - OTLP
  - OpenMetrics exposition
- Storage - Prometheus
  - In memory ring buffer based
  - Capped by maximum number of exemplars that can be stored

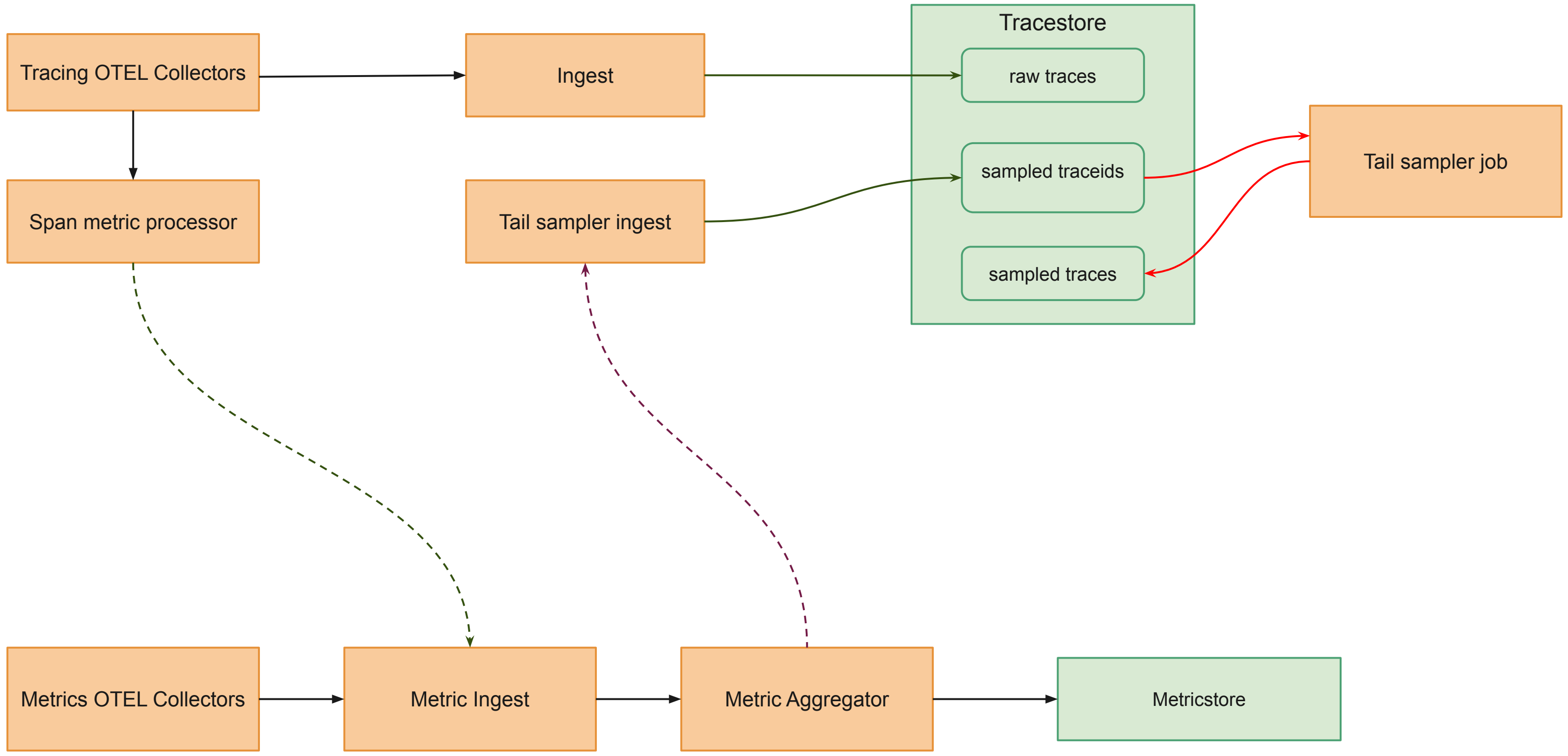


Use Exemplars Wisely!

Sample Efficiently

- Tail sampling is hard on OTEL Collector
  - Traffic from all sources need to be funneled into single instance
  - In memory sampling is expensive
- Tail sampling is easier on storage
  - Simpler to execute SELECT…INSERT queries
- Clients pick Exemplars as a representation of the source
  - Could it determine a true representative sample space?

Exemplar Based Tail Sampling



Keep Them Around Longer

- Exemplars on Prometheus have a shorter shelf life
- Showing Exemplars (sampled traces) on graphs for the life of the trace is **useful!**
- Tracestore already retains harvested Exemplars
  - Maybe serve them via query_exemplars API?

ClickHouse Based Exemplar Storage

Easier Said Than Done!

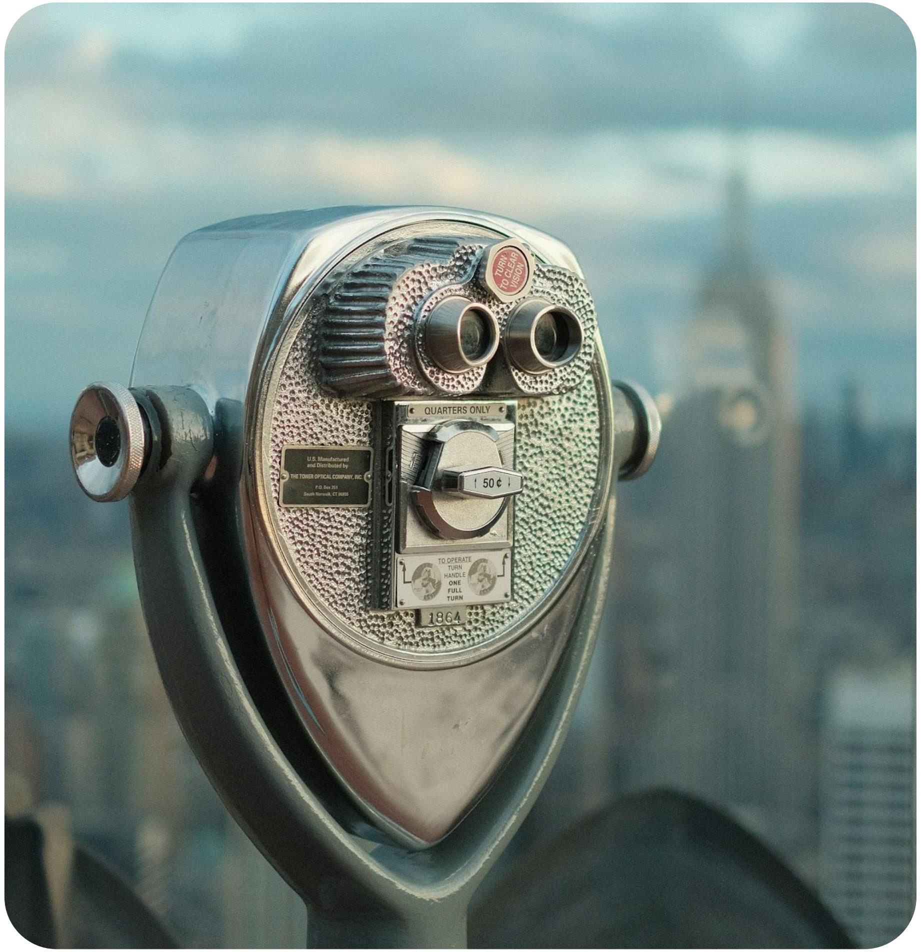
- Prometheus is blazing fast!
  - Can not slow down dashboard rendering
- **SELECT *** and columnar databases don't agree with each other
- Different table schemas have different performance characteristics

Attempt 1 - Flat Table

```sql
CREATE TABLE exemplars
(
    "traceid" String CODEC(ZSTD(1)),
    "spanid" String CODEC(ZSTD(1)),
    "timestamp" DateTime CODEC(Delta(4), LZ4),
    "value" Float64 CODEC(Gorilla, ZSTD(1)),
    "label_keys" Array(LowCardinality(String)) CODEC(ZSTD(1)),
    "label_values" Array(String) CODEC(ZSTD(9)),
    "Exemplar.label_keys" Array(LowCardinality(String))
CODEC(ZSTD(1)),
    "Exemplar.label_values" Array(String) CODEC(ZSTD(9))
)
ENGINE = ReplacingMergeTree
PARTITION BY toYYYYMMDD(toStartOfHour(timestamp))
ORDER BY (timestamp, traceid)
TTL timestamp + toIntervalDay(14)
SETTINGS index_granularity = 8192, ttl_only_drop_parts = 1
```

Some Observations

- Lot of time spent on filtering based on label key and label values
- Regexes are slow
- **LIMIT** the number of responses speeds things up. But the sample space is skewed towards the most recent Exemplars



Attempt 2 - Two Tables

```
CREATE TABLE default.exemplars
(
   "hashid" UInt64 CODEC(T64, ZSTD(1)),
   "timestamp" DateTime CODEC(Delta(4), LZ4),
   "__name__" String CODEC(ZSTD(1)),
   "_namespace_" String CODEC(ZSTD(1)),
   "value" Float64 CODEC(Gorilla, ZSTD(1)),
   "label_keys" Array(LowCardinality(String))
CODEC(ZSTD(1)),
   "label_values" Array(String) CODEC(ZSTD(9)),
   "Exemplar.labels" Map(LowCardinality(String),
String) CODEC(ZSTD(1)),
)
ENGINE = MergeTree()
PARTITION BY toYYYYMMDD(timestamp)
ORDER BY (timestamp, __name__, hashid)
```

```
CREATE TABLE default.exemplar_metadata
(
   "key" LowCardinality(String)
CODEC(ZSTD(1)),
   "value" String CODEC(ZSTD(1)),
   "hashid" UInt64 CODEC(T64, ZSTD(1)),
   "timestamp" DateTime DEFAULT now()
CODEC(Delta(4), LZ4)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMMDD(timestamp)
ORDER BY (key, value, hashid)
```

Still Not Fast Enough…

- Time to filter went down but time to fail on non-existent metrics/namespaces was substantially high
    - Not all namespaces had Exemplars at the time
- Larger the number of Exemplars returned, slower the response times

Attempt 3 - Add Dictionaries

```
CREATE DICTIONARY
default.exemplar_valid_metrics
(
   "__name__" String,
   "timestamp" DateTime
)
PRIMARY KEY __name__
SOURCE(CLICKHOUSE(QUERY 'select distinct
value as __name__, now() as timestamp from
default.exemplar_metadata where
key=\'__name__\''))
LIFETIME(MIN 0 MAX 300)
LAYOUT(COMPLEX_KEY_HASHED())
```

```
CREATE DICTIONARY
default.exemplar_valid_namespaces
(
   "_namespace_" String,
   "timestamp" DateTime
)
PRIMARY KEY _namespace_
SOURCE(CLICKHOUSE(QUERY 'select distinct
value as namespace, now() as timestamp from
default.exemplar_metadata where
key=\'_namespace_\''))
LIFETIME(MIN 0 MAX 300)
LAYOUT(COMPLEX_KEY_HASHED())
```

It's All In The Querying…

- **dictGet()** queries circuit break pretty fast on name and namespace
- Use inverted indexes on the exemplar table
    - Apply regexes against exemplar table and not metadata table
- Sampling before applying LIMIT
    - adjust sample % by time window
    - by hashid

Putting It All Together…

```sql
SELECT * FROM sherlockio.exemplars
    WHERE timestamp >= 1738751880 AND timestamp <= 1738795080
    AND hashid IN (
        SELECT * FROM (
        SELECT hashid FROM(
            SELECT hashid FROM sherlockio.exemplar_metadata WHERE (timestamp >= 1738751880) AND (key = '_namespace_')
and (value = 'tracing') INTERSECT DISTINCT SELECT hashid FROM sherlockio.exemplar_metadata WHERE (timestamp >=
1738751880) AND (key = 'service_name') and (value = 'foobar') INTERSECT DISTINCT SELECT hashid FROM
sherlockio.exemplar_metadata WHERE (timestamp >= 1738751880) AND (key = 'service_name') and (value = 'foobar')
INTERSECT DISTINCT SELECT hashid FROM sherlockio.exemplar_metadata WHERE (timestamp >= 1738751880) AND (key =
'service_name') and (value = 'foobar') INTERSECT DISTINCT SELECT hashid FROM sherlockio.exemplar_metadata WHERE
(timestamp >= 1738751880) AND (key = 'span_kind') and (value = 'SPAN_KIND_SERVER') INTERSECT DISTINCT SELECT hashid
FROM sherlockio.exemplar_metadata WHERE (timestamp >= 1738751880) AND (key = '__name__') and (value =
'duration_milliseconds')
        )
        )
     WHERE dictHas(sherlockio.exemplar_valid_namespaces, 'tracing') AND dictHas(sherlockio.exemplar_valid_metrics,
'duration_milliseconds')
    )
     AND has(label_keys, 'span_name') AND not match(arrayElement(label_values, indexOf(label_keys, 'span_name')),
'BadCommand) AND has(label_keys, 'span_name')) AND cityHash64(toStartOfFifteenMinutes(timestamp), hashid) % 10 == 0
    LIMIT 1000
    SETTINGS max_execution_time=10.000000
```

What Did We Gain?

We Can Do Things Better

- Embed Exemplars into Alerts and show sample traces
  - Faster time to triage
  - Automation can look at violating requests directly
- Longer retention of Exemplars allows better troubleshooting

We Can Do More In The Future

- Automated Triage
  - AI agents that look at alerts and the Exemplars in them to perform comprehensive troubleshooting
- Enhanced User Experiences
  - Exemplar based latency heat maps
  - Visually simpler to find problematic requests to triage

Questions?

Thank You!