Using OpenTrace to Troubleshoot DB Performance

ANATOLY MIKHAYLOV

Anatoly is a keen enthusiast in observability and performance troubleshooting. He works as a Staff SRE engineer at Zendesk in Dublin, Ireland, where he is part of a global team that builds and maintains next generation observability tools for dozens of high-traffic microservices/databases. He contributes to the Zendesk Engineering blog. He is also a runner, an avid hiker, and nature photographer. Before Zendesk, Anatoly worked as a DBA, DevOps and software engineer for over 10 years. mikhailov.anatoly@gmail.com

You cannot fix any of the problems you cannot see. I will outline how the Zendesk SRE team monitors database performance and how you can apply it to your own observability challenges. Our approach considers low-level database performance data, proxy logs, and application performance monitoring (APM) in order to expose the meaningful context behind an individual slow SQL query.

Database performance is central to users' experiences, so having excellent observability is critically important. Many of the observability tools that we build or buy are focused on ensuring optimal customer experience, or determining the extent of customer impact during outages and service degradations. These are challenges that many in the industry experience, and I hope that the work I have done at Zendesk will help you build your own observability dashboards. This approach leads to improved back-end performance and happier customers.

Ideally, what we want is a way to track just the single user request that resulted in bad performance. Imagine being able to complete an incident's root cause analysis that takes seconds rather than minutes or hours. API traffic from a large set of customers can not only be traced to relevant database internal performance metrics at a given time, but also be visualized and presented in a readable format. Why is a given SQL query fast in one case and slow in another? When does database performance degradation lead to an outage and when does it not? Is the query execution plan alone enough to understand and address performance issues?

Over the past year we substantially improved database observability, and this improved overall stability and reliability of the system. We built tools to help engineers see and understand performance issues quicker. This has also helped to prevent outages.

I will go through key elements of the observability stack we have built to create meaningful context around requests, linking SQL queries to APM distributed traces and even proxy log events. In this context the proxy event is the entry point, the time elapsed between when an individual request enters the system and once the response is ready to be sent back. SQL query analysis, proxy log event, and APM tracing are the three key elements. To support and enhance their integration we collect database internal information and link that to the rest of the system. We also collect data from information schema to have information about data-set size, which is very important for profiling SQL queries and understanding how data-set size impacts the overall performance. Each individual element provides information, and their integration helps to traverse from one to another using Open Tracing.

OpenTracing

OpenTracing is a vendor-neutral, cross-language standard for tracing distributed applications. Datadog offers OpenTracing implementations for many APM tracers, including the Ruby on Rails version we use for demo purposes. According to the official documentation (https://opentracing.io/docs/overview/spans/):

> The *span* is the primary building block of a distributed trace, representing an individual unit of work done in a distributed system. Each component of the distributed system contributes a span—a named, timed operation representing a piece of the workflow. Spans can (and generally do) contain *References* to other spans, which allows multiple Spans to be assembled into one complete Trace—a visualization of the life of a request as it moves through a distributed system.

A *trace_id* is the unique identifier we propagate from one service to another in order to keep the context. While it can be relatively easy to connect APM application requests with a proxy log event, it's much more difficult to propagate a trace_id to other services like the database process list; I will show how we use SQL comments to do so. We can reuse this approach to connect a background `cron` task job with a relevant SQL query by generating trace_id outside of the HTTP request life cycle. Any service that communicates to a database can benefit by propagating the necessary context with SQL query and tracing libraries that help to abstract complexity and use higher level objects: span and trace.

Database Observability

According to *High Performance MySQL* (https://www.highperfmysql.com):

> Performance is response time. We measure performance by task and time, not by resource. Performance optimization is the practice of reducing response time as much as possible.

MySQL Performance Schema provides a way to inspect database performance and find out why a SQL query runtime takes longer. Or saying it another way: why an SQL query is slow. This level of instrumentation is critical to address performance issues. MySQL 5.6+ supports the sys schema (https://www.percona.com/blog/2014/11/20/sys-schema-mysql-5-6-5-7/), which is a set of objects that interprets data collected by the Performance Schema in a manageable format. I will describe how we take snapshots of relevant queries from the schema with 15-second resolution and learn how to use tracing to connect SQL queries, including trace_id, with the application traces and proxy logs. This tool will not only give you a great instrument to jump from slow query to proxy logs but will also filter out HTTP requests with high database runtime, and so we will focus on these.

According to *High Performance MySQL:*

> [A] common mistake is to observe a slow query, and then look at the whole server's behavior to try to find what's wrong. If the query is slow, then it's best to measure the query, not the whole server....Because of Amdahl's law, a query that consumes only 5% of total response time can contribute only 5% to overall speedup, no matter how much faster you make it.

According to *Site Reliability Engineering* (https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/):

> Your monitoring system should address two questions: what's broken, and why? The "what's broken" indicates the symptom; the "why" indicates a (possibly intermediate) cause....When pages occur too frequently, employees second-guess, skim, or even ignore incoming alerts, sometimes even ignoring a "real" page that's masked by the noise.

"What" and "Why" have different meanings for DBA and for SRE.

- Relational storage and SRE observability worlds are somewhat disconnected. We have to close the gap between a slow HTTP request and what the DB was doing at that very moment.
- SRE teams view high-volume traffic that often has high cardinality. High-cardinality monitoring tools allow connecting with APM/logs.
- DBA teams focus on database performance and the portion of inefficient SQL queries that make it to the DB slow query log.

Improving Observability with Database Signal

Four golden signals (saturation, latency, traffic, errors) make up a well-known approach in web service monitoring, but how can we apply these signals to database performance? Is there anything unique about database performance?

According to *High Performance MySQL:*

> `Threads_running` tends to be very sensitive to problems, but pretty stable when nothing is wrong. A spike of unusual thread states in SHOW PROCESSLIST is another good indicator....If everything on the server is suffering, and then everything is okay again, then any given query that's slow isn't likely to be the problem.... Pileups typically result in a sudden drop of completions, until the culprit finishes and releases the resource that's blocking the other queries. The other queries will then complete.

We will follow the advice from this book to pick the most important performance metrics:

> The essence of this technique is to capture...[it] at high frequency...and when the problem manifests, look for *spikes* or *notches* in counters such as `Threads_running`, `Threads_connected`, `Questions` and `Queries`.

Each of the key metrics carries the signal. For this purpose we choose very low thresholds as service level indicators: seven threads connected, five threads running, DB runtime below two seconds and queries/second (QPS) not higher than 20. When the threshold is exceeded it indicates a signal (1); otherwise, there's absence of the signal (0). We will use the bitmask OR operation to calculate the resulting database signal (Table 1).

Bit	Signal	SLI
0001	Threads connected	7 sec
0010	Threads running	5 sec
0100	Database runtime	2 sec
1000	Database queries per second	20

Table 1: Bitmasks for signaling exceeded SLIs

Each individual SQL query will be instrumented with an SQL comment that contains a unique identifier trace_id—for example:

```
SELECT * from users /* 1541859401495831 */
```

For visualization purpose we use Datadog and its APM (https://docs.datadoghq.com/tracing/connect_logs_and_traces/Logs_integration):

> The correlation between Datadog APM and Datadog Log Management is improved by the injection of trace IDs, span IDs, `env`, `service`, and `version` as attributes in your logs. With these fields you can find the exact logs associated with a specific service and version, or all logs correlated to an observed trace (https://docs.datadoghq.com/tracing/visualization/#trace).

Full Circle Observability

Disclaimer: code snippets shared below are open source (MIT license) and are not used at Zendesk but are created exclusively for this article for the purpose of illustration.

A tracing library automatically generates a trace_id on the application side. When the trace_id is generated we propagate context via the HTTP header to the downstream and upstream dependencies, so when the two services communicate to each other, the HTTP header `X-Trace-ID` is the key element needed to bring the context up the stack, from the application to the proxy layer (see Figure 1). In the Ruby on Rails application, the simplified version of the middleware looks as follows:

```
class DdtraceMiddleware
  def call(env)
    result = @app.call(env)
    result[1]['X-Trace-Id']
                        Datadog.tracer.active_span.trace_
id.to_s
    result
  end
end
```

Then the trace_id can be part of Nginx proxy logs, application log, all dependent microservices and external services that were called to serve the original requests. For example, the Nginx access log may appear as follows:

```
log_format    json '{"dd":{"trace_id":"$upstream_http_x_
trace_id"}}'
```

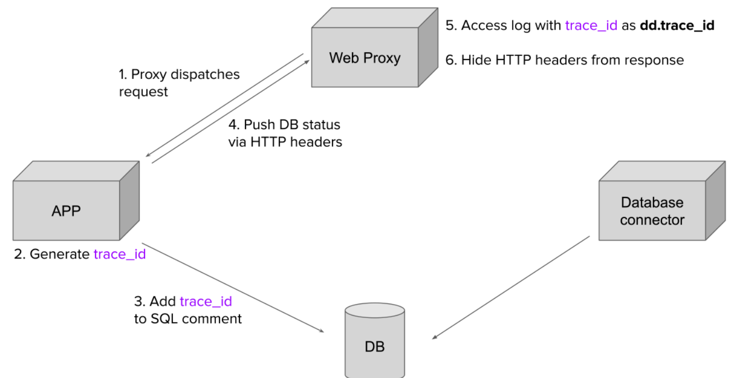

Figure 1: The trace_id gets added by the application and pushed back upstream to the web proxy and downstream to the database.

We can bring more information from the application up to the proxy layer, store it in `access_log` for observability purposes, and then remove the service information from the HTTP response. For example, if we collect the information about DB runtime, connected and running threads, as well as QPS and calculated Database signal, then the proxy configuration will look as follows:

```
log_format    json '{'
  '"dd":{'
    '"trace_id":"$upstream_http_x_trace_id"'
  '},'
 '"http":{'
    '"performance":{'
      '"queueing_delay":$upstream_http_x_queueing_delay_
digits,'
      '"total_runtime":$upstream_http_x_total_runtime_
digits,'
      '"db_runtime":$upstream_http_x_db_runtime_digits,'
      '"db_signal":$upstream_http_x_db_signal_digits,'
      '"db_threads_running":
               $upstream_http_x_db_threads_running_digits,'
      '"db_threads_connected":
               $upstream_http_x_db_threads_connected_
digits,'
      '"db_qps":$upstream_http_x_db_qps_digits'
    '}'
  '}'
'}';
```

Database signal calculation can be another middleware layer with the following code:

```
def get_db_signal
  db_threads_connected_slo \
             = Thread.current[:db_threads_connected] > 7
  db_threads_running_slo   \
             = Thread.current[:db_threads_running] > 4
  db_runtime_slo          \
             = Thread.current[:db_runtime] > 2
  db_qps_slo              \
             = Thread.current[:db_qps] > 20

  db_threads_connected_bit = db_threads_connected_slo ? 1 :
0
```

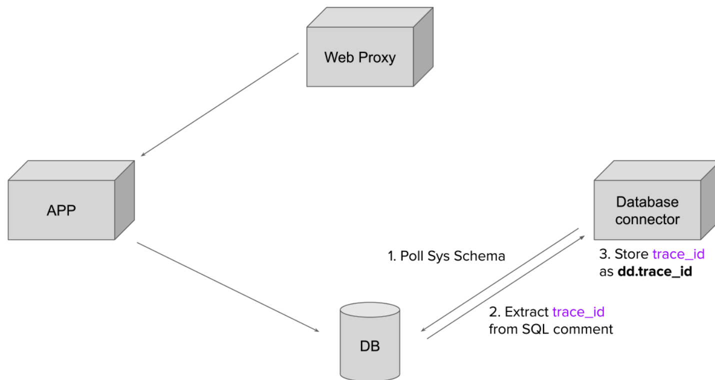

Figure 2: A second way of using the trace_id is for the database connector to query sys schema.

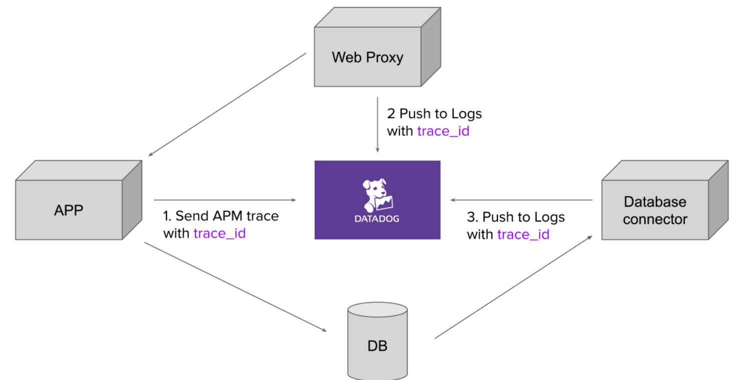

Figure 3: Full observability circle. The unique identifier trace_id gets propagated from the application to database and proxy logs.

```
    db_threads_running_bit    = db_threads_running_slo ? 2 : 0
    db_runtime_bit            = db_runtime_slo ? 4 : 0
    db_qps_bit                = db_qps_slo ? 8 : 0

    (db_threads_connected_bit | db_threads_running_bit \
        | db_runtime_bit | db_qps_bit).to_s
  end
```

Both the middleware layers and enhanced proxy log configuration help to traverse and debug slow SQL queries in either direction: from proxy to the database process list data and also from process list up to the proxy log. Figure 2 shows a communication between an asynchronous process and the database to collect performance information. Step 1 polls the sys schema process list, extracts the individual SQL query, parses the SQL comment with trace_id, and constructs the JSON event with the `dd.trace_id` identifier. This is a very important step to connect asynchronous data collection with request/response events later on and in being able to create context around slow SQL queries.

Process list aggregation can be done via a bash script:

```
    function process_list_json() {
      trace_id=$(echo "$1" |grep -Eo '/\* [0-9]{16,20} \*/' \
         | awk '{print $2}')

      if [ -z "$trace_id" ]
      then
        echo "{\"mysql\": \"process_list\", \"process_list\":\
  $1}"
      else
        echo "{\"mysql\": \"process_list\", \"process_list\":\
  $1, \
         \"dd\": {\"trace_id\": ${trace_id}}}"
      fi
    }

    function process_list() {
      /usr/bin/mysql -h127.0.0.1 -uroot -s -r -e "SELECT
      JSON_OBJECT(
        'thd_id', thd_id,
        'conn_id', conn_id,
        'command', command,
        'state', state,
          'current_statement', current_statement,
          'statement_latency', statement_latency / 1000,
          'progress', progress,
          'lock_latency', lock_latency / 1000,
          'rows_examined', rows_examined,
          'rows_sent', rows_sent,
          'rows_affected', rows_affected,
          'tmp_tables', tmp_tables,
          'tmp_disk_tables', tmp_disk_tables,
          'full_scan', full_scan,
          'last_statement', last_statement,
          'last_statement_latency', last_statement_latency /
  1000
        )
        FROM sys.x\$processlist
        WHERE pid IS NOT NULL
        AND db = 'db'
        LIMIT 25;"
    }

  while true; do
      process_list | while read -r item; do
        process_list_json "$item" "$threads" > /proc/1/fd/1
      done
      sleep 1;
    done
```

This script contains two key functions: `process_list()` to collect SQL queries, and `process_list_json()` to extract trace_id from the SQL comment; it also contains a loop to keep these two functions running once per minute. This script is running in a docker container; output gets redirected to `STDOUT` and is collected by the OpenTracing agent: in this case, the datadog-agent.

An OpenTracing agent receives an APM event from the application and a log event from the proxy and JSON log events. Log events get sent to the log intake endpoint separately. Note: Datadog is used for illustration purposes, but database performance monitoring can be done by any alternative OpenTracing provider.

Conclusion

With comprehensive instrumentation and distributed tracing, we created an observability basis to detect database performance degradation and have the necessary context for further investigation. A database signal can help to address the following questions:

- How much time a database spent processing an SQL query for a given HTTP request
- How saturated the database resources have been during the time of request
- Where the database spent most of the time processing database requests
- How many customers are impacted and what their user experience was

You can find more information with related visualizations at https://medium.com/@unknown_runner.

usenix
SRECON® AMERICAS
DECEMBER 7–9, 2020 • VIRTUAL EVENT
PROGRAM CO-CHAIRS

Nora Jones
Jeli.io

Mike Rembetsy
Bloomberg
The full program and registration will be available soon.
www.usenix.org/srecon20americas