



ClickHouse

Amsterdam Meetup With The ClickHouse Team – June 8th, 2022

05.25.2022



ClickHouse

Amsterdam Meetup With The ClickHouse Team – June 8th, 2022

05.25.2022



Good Evening & Welcome 🙌



Lorenzo Mangani

- Co-Founder and CEO @ **qhip + uptrace**
- Open-Source Fanatic, Proud Father, Workaholic and totally in love with my Job
- Leading an amazing team specialized in active and passive large scale monitoring and ingestion solutions for Traces, Logs, Packets and Metrics on top of **ClickHouse** quietly serving some of the *largest and most innovative enterprises* on the planet

- ★ Proud members of the “*wild*” ClickHouse OSS community since inception
- ★ Supporting hundreds of heavy duty ClickHouse server nodes worldwide
- ★ Contributors to our community with dozens of useful projects and tools
- ★ Early alpha testers of ClickHouse Cloud (*thanks to the wonderful ClickHouse team!*)



uptrace



Tonight's Menu



A tale of two ClickHouse projects

- MEET UPTRACE & CLOKI
 - What are they for and why do we care?
 - OpenTelemetry Everything vs. Unstructured Data
 - Merging towards Data MMA (mixed monitoring arts)
 - Competition & Advantages other other solutions
- UPTRACE & CLICKHOUSE
 - High Performance, Low Maintenance, Strong Leadership
 - Super powered by ClickHouse Inc & ClickHouse Cloud
- WHAT IS NEXT
 - Uptrace Cloud Promotion
 - Open-Source & Community

What is Uptrace?

Uptrace is an OpenTelemetry tracing tool that monitors performance, errors, and structured logs. It collects, aggregates and processes data to help users quickly pinpoint *service failures* and find *performance bottlenecks*.

Opentelemetry is an open source and vendor-neutral API for distributed traces (*including logs and errors*) and metrics. It specifies how to collect and export telemetry data. With OpenTelemetry, developers can instrument their application once and then add or change monitoring vendors without changing the instrumentation.

What is cLoki?

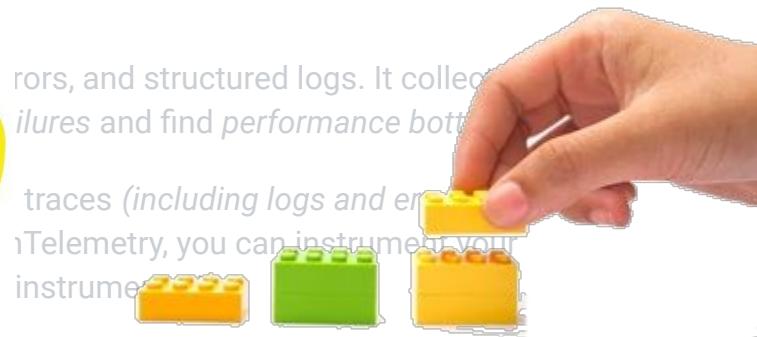
cLoki is fast and powerful Log & Metrics management tool built on top of ClickHouse. It provides ready-to-use APIs transparently compatible with *LogQL*, *PromQL*, *InfluxDB*, *Elastic and more* to ingest logs and metrics from industry standard Agents such as *Promtail*, *Grafana-Agent*, *Vector*, *Logstash*, *Telegraf* and many others.

cLoki implements a *custom, extensible LogQL/PromQL transpiler* to dynamically search, filter and extract data and metrics from structured and unstructured logs, events, shipping with plugin-less Grafana LogQL/Tempo integrations as well as a built in Explore UI and LogQL command line tools for querying and extracting data.

What is Uptrace?

Uptrace is an OpenTelemetry tracing tool that aggregates and processes data to help us

OpenTelemetry is an open source and vendor-neutral specification for collecting, aggregating, and analyzing data from distributed systems. It specifies how to collect and export metrics. It specifies how to collect and export logs. It specifies how to instrument your application once and then add or change



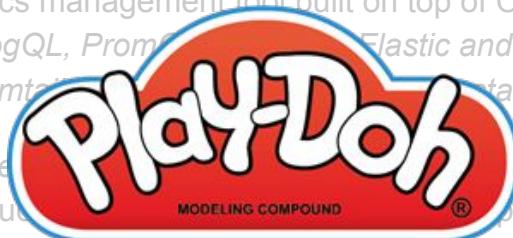
Uptrace collects logs, metrics, and structured logs. It collects traces and finds performance bottlenecks and failures.

Uptrace includes a UI for visualizing traces (including logs and errors). With OpenTelemetry, you can instrument your application once and then add or change

What is cLoki?

cLoki is fast and powerful Log & Metrics management tool built on top of ClickHouse. It provides ready-to-use APIs transparently compatible with *LogQL*, *PromQL*, *Elastic* and more to ingest logs and metrics from industry standard Agents such as *Promtail*, *Filebeat*, *Elastic*, *Logstash*, *Telegraf* and many others.

cLoki implements a *custom*, *extensible* and *fast* storage layer to store logs and metrics from structured and unstructured sources. It provides a plugin-less Grafana integration, a plugin-less ClickHouse integration, a plugin-less Elasticsearch integration, a plugin-less Logstash integration, a plugin-less Telegraf integration, and a plugin-less ClickHouse integration. It also provides a plugin-less ClickHouse integration, a plugin-less Elasticsearch integration, a plugin-less Logstash integration, a plugin-less Telegraf integration, and a plugin-less ClickHouse integration.



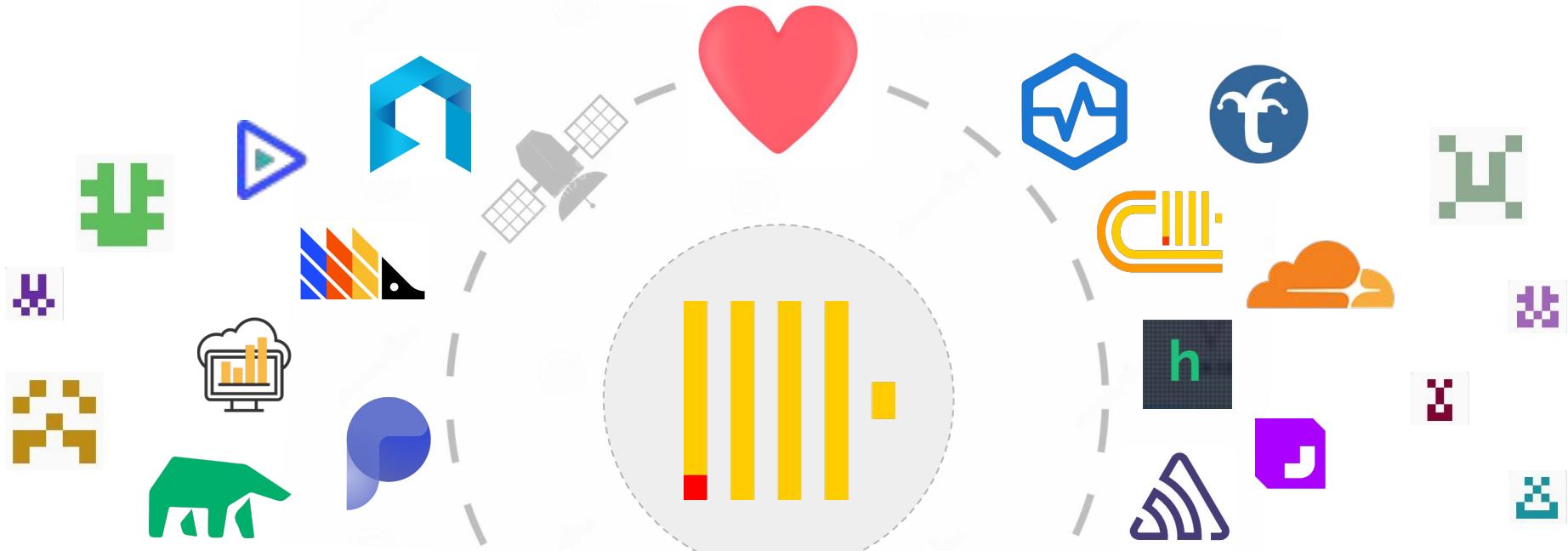
Community Powered

Like many, we've met in one of the many corners of the worldwide ClickHouse community.



Community Powered

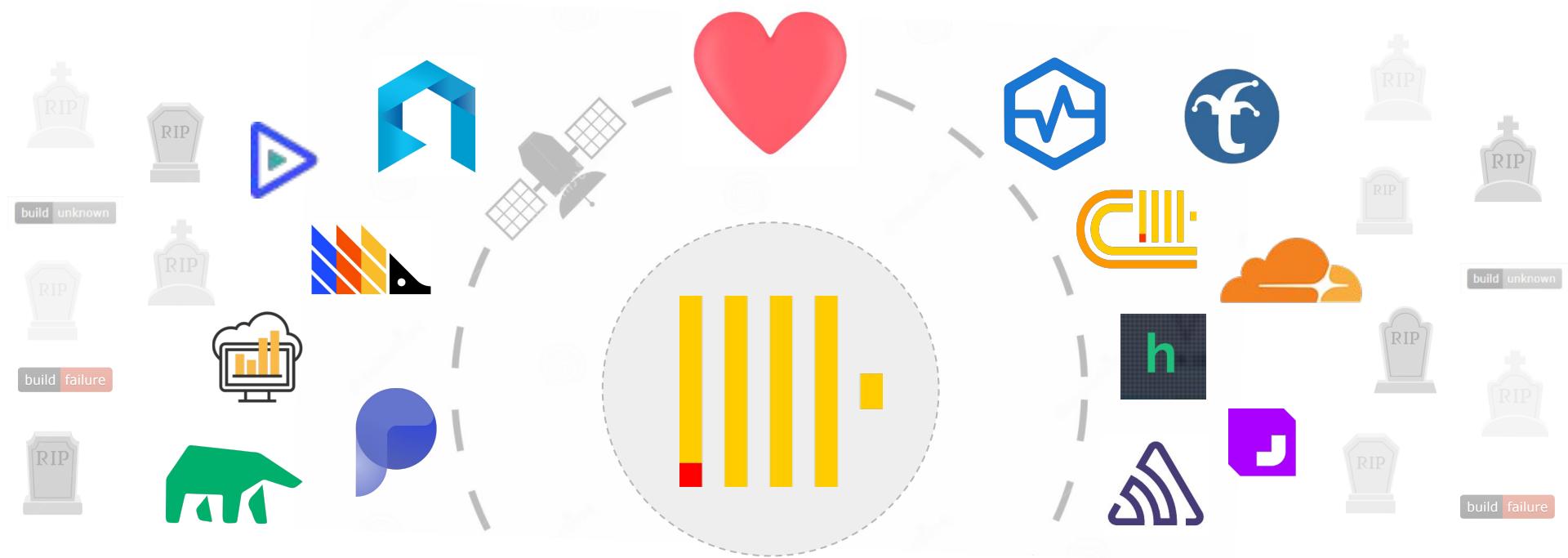
ClickHouse is like a powerful startup accelerator with many projects large and small floating around.



Community Powered... *while staying alive*

ClickHouse is like a powerful startup accelerator with many projects large and small floating around.

Not without casualties. We lost count of semi-abandonware projects claiming to redefine logging and monitoring over ClickHouse over time



Community Powered... *while staying alive*

Luckily both **Uptrace** and **cLoki** managed to remain healthy, got a following and growing user action!



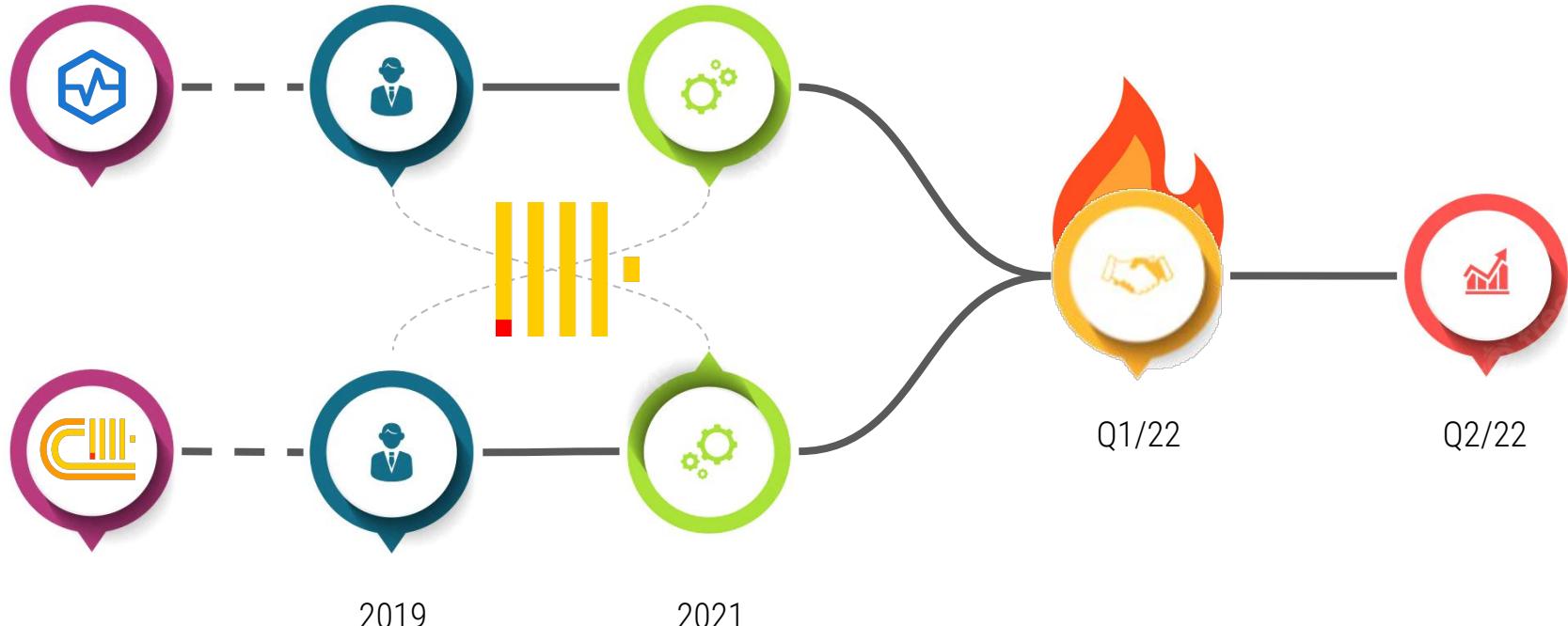
Competing Powers? 🥊

Our project coexisted fine but inevitably, users brought up the question - which does what best?
Since **Uptrace** could also do logging, and **cLoki** could also do tracing, competition was *inevitable* ...



Combined Powers 🔥

Instead, we decided to **unite** and **combine** our skills & features into one majestic single group/project. Under the **Uptrace** brand, we merged our codebases, design & ideas into a fresh groundbreaking product



Combined Dev Teams 🔥



Lorenzo Mangani



Vladimir Mihailenco



Alexandr Dubovikov



Anatolii Mihailenco



Volodymyr Akchurin



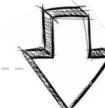
Joel Guerra



Jachen Duschletta



You?
join our team



- + many OSS code contributors and supporters on gh



Combined Projects 🔥



opentelemetry golang tracing go

uptrace clickhouse cloki nodejs logql

● JavaScript ● Go ● TypeScript ● V

uptrace <small>Public</small> OpenTelemetry distributed tracing tool that monitors performance, errors, and logs <small>Vue ⭐ 702 📂 38</small>	uptrace-go <small>Public</small> OpenTelemetry Go distribution for Uptrace <small>Go ⭐ 53 📂 9</small>
uptrace-python <small>Public</small> OpenTelemetry Python distribution for Uptrace <small>Python ⭐ 4 📂 2</small>	uptrace-ruby <small>Public</small> OpenTelemetry Ruby distribution for Uptrace <small>Ruby ⭐ 3</small>
uptrace-js <small>Public</small> OpenTelemetry JavaScript distribution for Uptrace <small>TypeScript ⭐ 6 📂 3</small>	uptrace-dotnet <small>Public</small> OpenTelemetry .NET distribution for Uptrace <small>C# ⭐ 2 📂 1</small>
cLoki <small>Public</small> Forked from imangani/cLoki Clickhouse Loki: Grafana Loki API + ClickHouse Backend in NodeJS <small>JavaScript ⭐ 4 📂 1</small>	fluxXpipe <small>Public</small> Experimental Flux API/Pipeline for ClickHouse and other embedded datasources <small>FLUX ⭐ 10 📂 2</small>
promcaso <small>Public</small> ClickHouse Custom Query / Metrics Exporter for Prometheus <small>Go ⭐ 3</small>	vLogQL <small>Public</small> Forked from imangani/vLogQL LogQL Client in V <small>V ⭐ 3</small>
cloki-view <small>Public</small> cLoki LogQL UI Client <small>JavaScript ⭐ 12 📂 2</small>	clickhouse-mate <small>Public</small> Advanced Clickhouse "Play" Client and Web User-Interface <small>TypeScript ⭐ 7 📂 1</small>



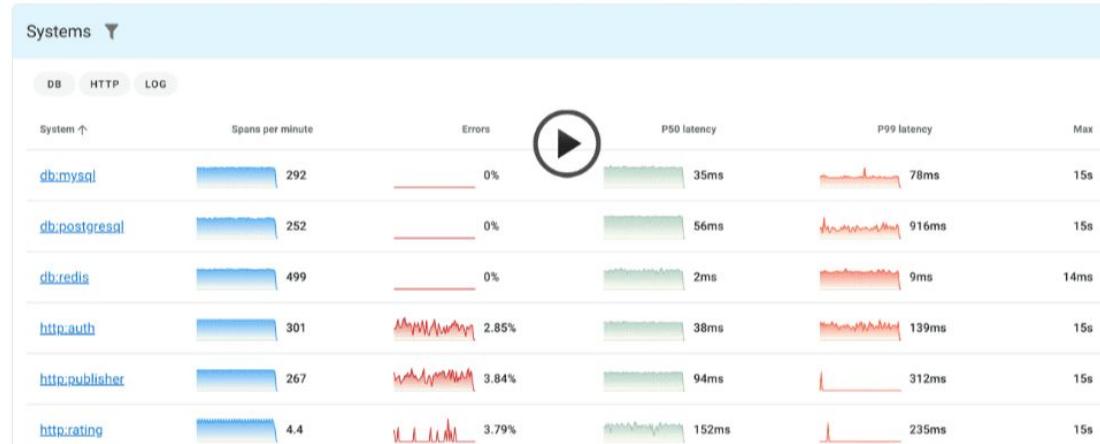
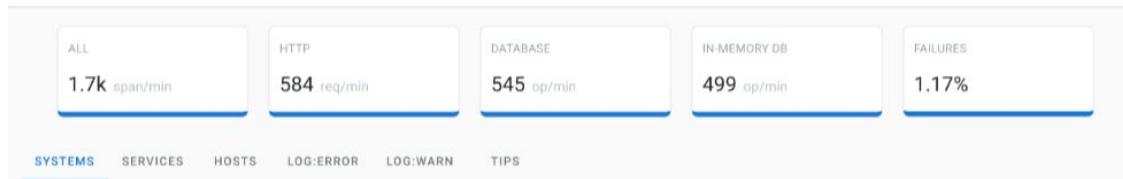
uptrace





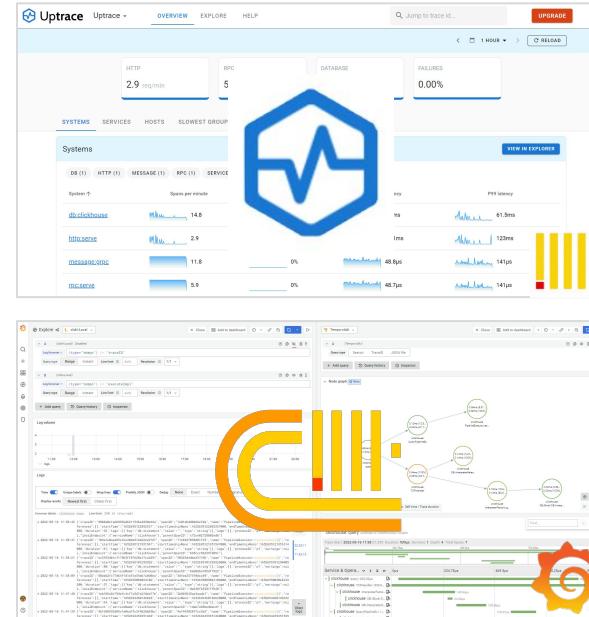


Uptrace is a cost-effective log and tracing solution that helps users monitor, understand, and optimize complex distributed systems faster than ever.



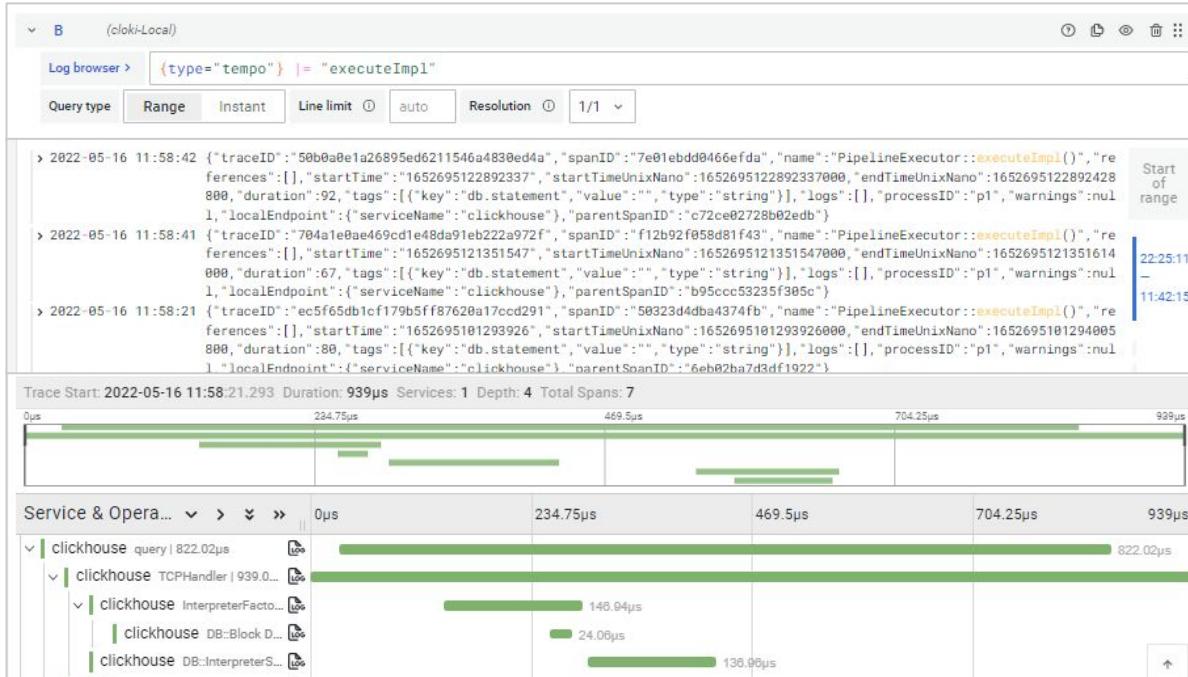


Uptrace allows analysis of logs, metrics and tracing events on a single plane, and natively supports data ingestion using *industry standard* tools and agents



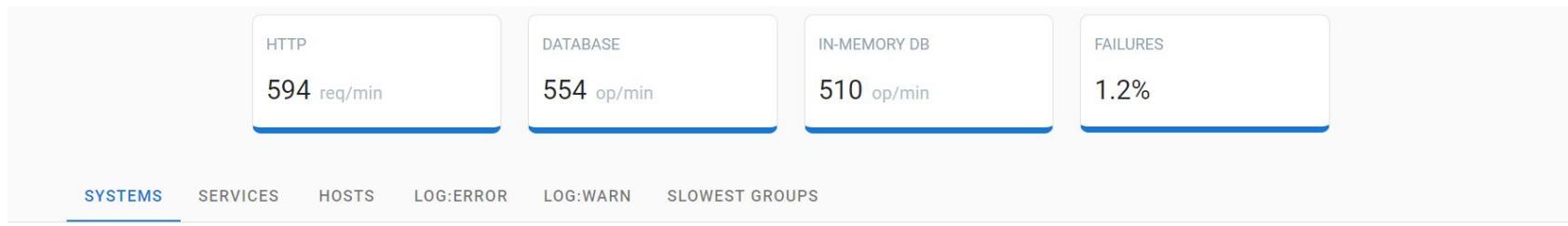


Uptrace natively integrates and supports Standard APIs (*logql*, *promql*, *tempo*) providing unprecedented flexibility interfacing with its data from any platform





Uptrace can automatically correlate **spans**, **logs** and **events** with precision while retaining relevant event tags based on front or tail sampling techniques



Systems							VIEW IN EXPLORER			
	DB (3)	HTTP (4)	LOG (2)							
System ↑	Spans per minute		Error rate	P50 latency	P99 latency	Max	Summary			
service:Handle		308		0.05%		31ms		800ms	1.9s	
service:ICE		246		0.14%		52ms		263ms	352ms	
service:Media		510		0%		2ms		9.2ms	15ms	



Uptrace does the job right and hunts for interesting and critical information.
“What are the slowest performing services right now?” - Here they are, Mike.

HTTP
594 req/min

DATABASE
554 op/min

IN-MEMORY DB
510 op/min

FAILURES
1.2%

SYSTEMS SERVICES HOSTS LOG:ERROR LOG:WARN SLOWEST GROUPS

Slowest groups

Group Name	System	count/min	err%	p50 ↓	p90	p99
ImportPaddleAlerts	service:paddle_import_subs	0.017	0%	3960ms	3960ms	3960ms

Span Name	System	Time	↓ Dur.
ImportPaddleAlerts	service:paddle_import_subs	today at 20:05	3960ms

[VIEW IN EXPLORER](#)

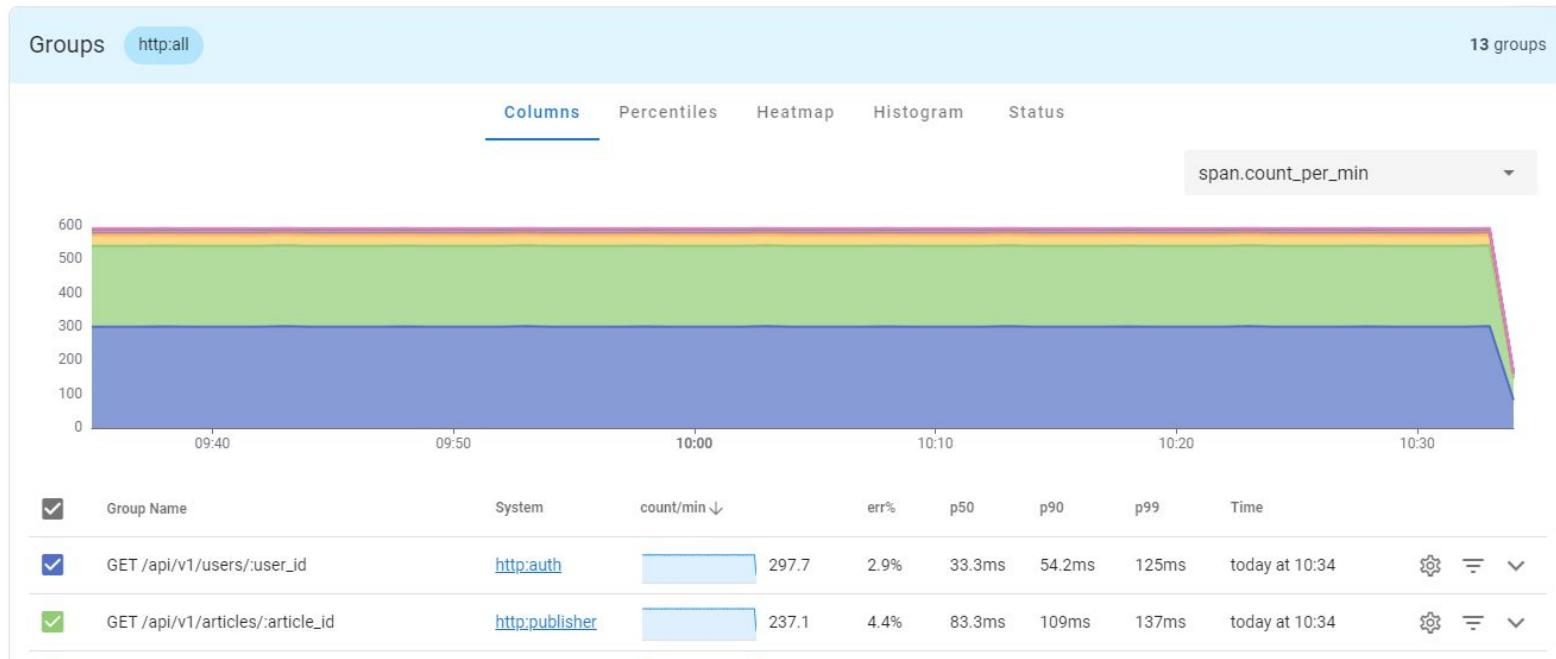
SELECT * FROM spans_data_dist WHERE id, parent_id, trace_id

[db:unknown_system](#) 2.3 0% 259ms 322ms 376ms



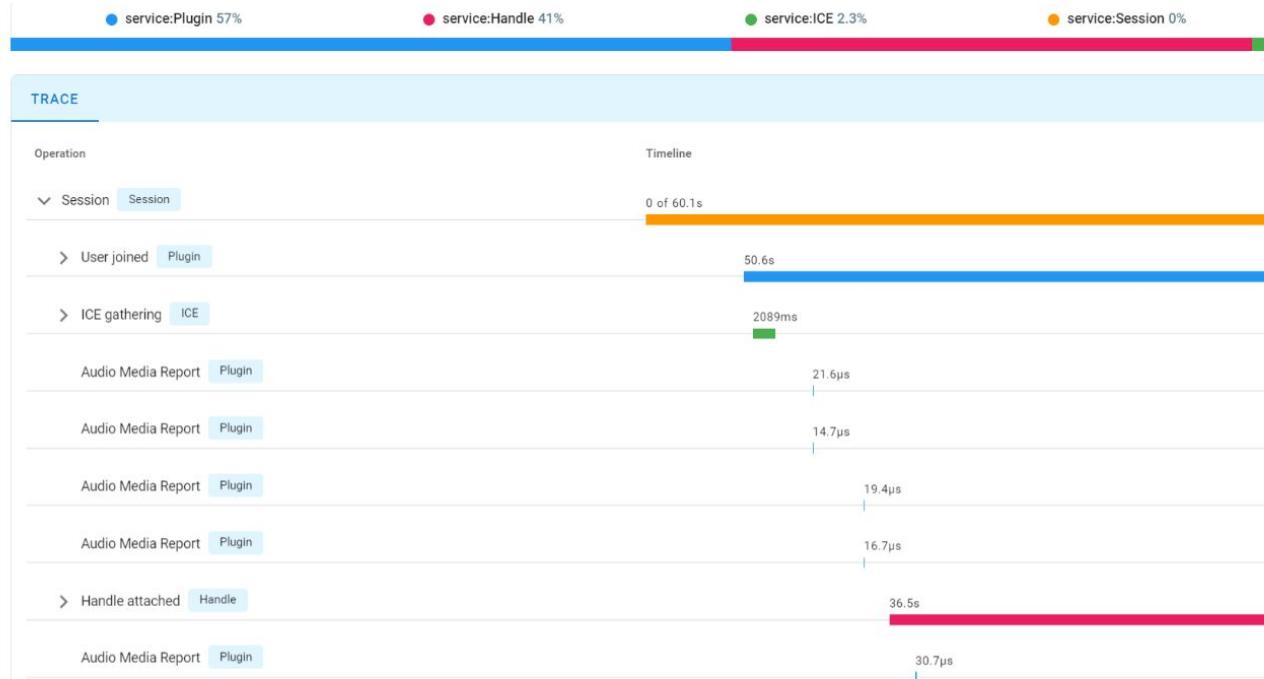


Uptrace can trace and analyze performance of monitored servers and services with great detail at any resolution, outlining performance metrics and issues in **real-time**



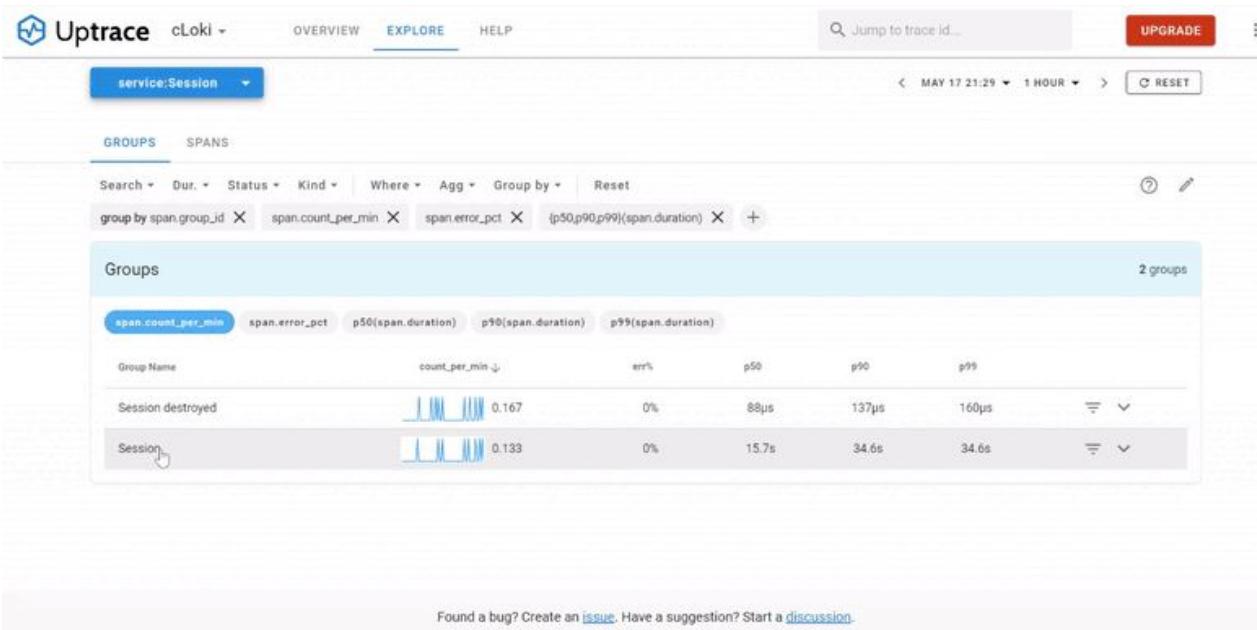


Telemetry spans are designed to reveal every detail about traced sessions with cross-referencing between processes (parent / child) and automated correlation





Uptrace workflows allows browsing data and spans at all levels of knowledge and expertise via the built-in user interface with zero training or learning curve



The screenshot shows the Uptrace Explore interface. At the top, there's a navigation bar with tabs for "OVERVIEW", "EXPLORE" (which is active), and "HELP". A search bar says "Jump to trace id..." and a red "UPGRADE" button. Below the navigation is a dropdown menu set to "service:Session". The main area has two tabs: "GROUPS" (selected) and "SPANS". Under "GROUPS", there are search filters: "Search", "Dur.", "Status", "Kind", "Where", "Agg", "Group by", and "Reset". A query builder shows filters: "group by span.group_id" (selected), "span.count_per_min" (selected), "span.error_pct" (selected), and "(p50,p90,p99)(span.duration)" (selected). Below this is a table titled "Groups" showing two entries:

Group Name	count_per_min ↓	err%	p50	p90	p99
Session destroyed	0.167	0%	88µs	137µs	160µs
Session	0.133	0%	15.7s	34.6s	34.6s

At the bottom, a footer bar says "Found a bug? Create an [issue](#). Have a suggestion? Start a [discussion](#)".



Uptrace

Uptrace now integrates 100% the cLoki LogQL functionality natively and can be used to ingest massive amounts of inserts and *query, filter and extract* meta out of any **log** format.

emitter ▾ mediatype ▾ type ▾ metric ▾ subtype ▾ test_id ▾ freq ▾ forwarder ▾ event ▾ session_id ▾

Limit: 1000

```
{forwarder="vector", type="syslog"} |~ "boat" | json
```

X

LogQL		413 logs
>	today at 17:32	{"appname":"ahmadajmi","facility":"alert","hostname":"names.us","message":"We're gonna need a bigger boat","msgid":"ID662","procid":5100,"severity":"info","version":1}
>	today at 17:25	{"appname":"meln1ks","facility":"alert","hostname":"for.org","message":"We're gonna need a bigger boat","msgid":"ID942","procid":7942,"severity":"emerg","version":1}
>	today at 17:23	{"appname":"benefritz","facility":"alert","hostname":"some.de","message":"We're gonna need a bigger boat","msgid":"ID14","procid":1734,"severity":"crit","version":2}
>	today at 17:23	{"appname":"meln1ks","facility":"alert","hostname":"names.net","message":"We're gonna need a bigger boat","msgid":"ID388","procid":2062,"severity":"warning","version":1}
>	today at 17:21	{"appname":"ahmadajmi","facility":"alert","hostname":"names.org","message":"We're gonna need a bigger boat","msgid":"ID993","procid":271,"severity":"info","version":2}
>	today at 17:19	{"appname":"ahmadajmi","facility":"alert","hostname":"random.org","message":"We're gonna need a bigger boat","msgid":"ID739","procid":2933,"severity":"warning","version":1}
>	today at 17:17	{"appname":"benefritz","facility":"alert","hostname":"random.net","message":"We're gonna need a bigger boat","msgid":"ID840","procid":4723,"severity":"alert","version":2}



On top of the existing functionality **Uptrace** can now also automatically correlate analyzed **Logs** to **Telemetry Spans** and/or scraped **Metrics** in a *single click*

emitter ▾ mediatype ▾ type ▾ metric ▾ subtype ▾ test_id ▾ freq ▾ forwarder ▾ event ▾ session_id ▾

Limit - 1000

{forwarder="vector", type="syslog"} |~ "boat" | json X

LogQL 3 logs

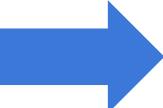
> today at 17:36 {"log":"The XML monitor is down, hack the online circuit so we can copy the THX system!","severity":"err","traceID":"bb8951f4137c6251b176be338b48d252","spanID":"c1effc1c8003ed85","service":"horse-http"}

▼ today at 17:36 {"log":"If we bypass the microchip, we can get to the PCI matrix through the primary EXE microchip!","severity":"warning","traceID":"bb8951f4137c6251b176be338b48d252","spanID":"c1effc1c8003ed85","service":"horse-http"}

FIND TRACE

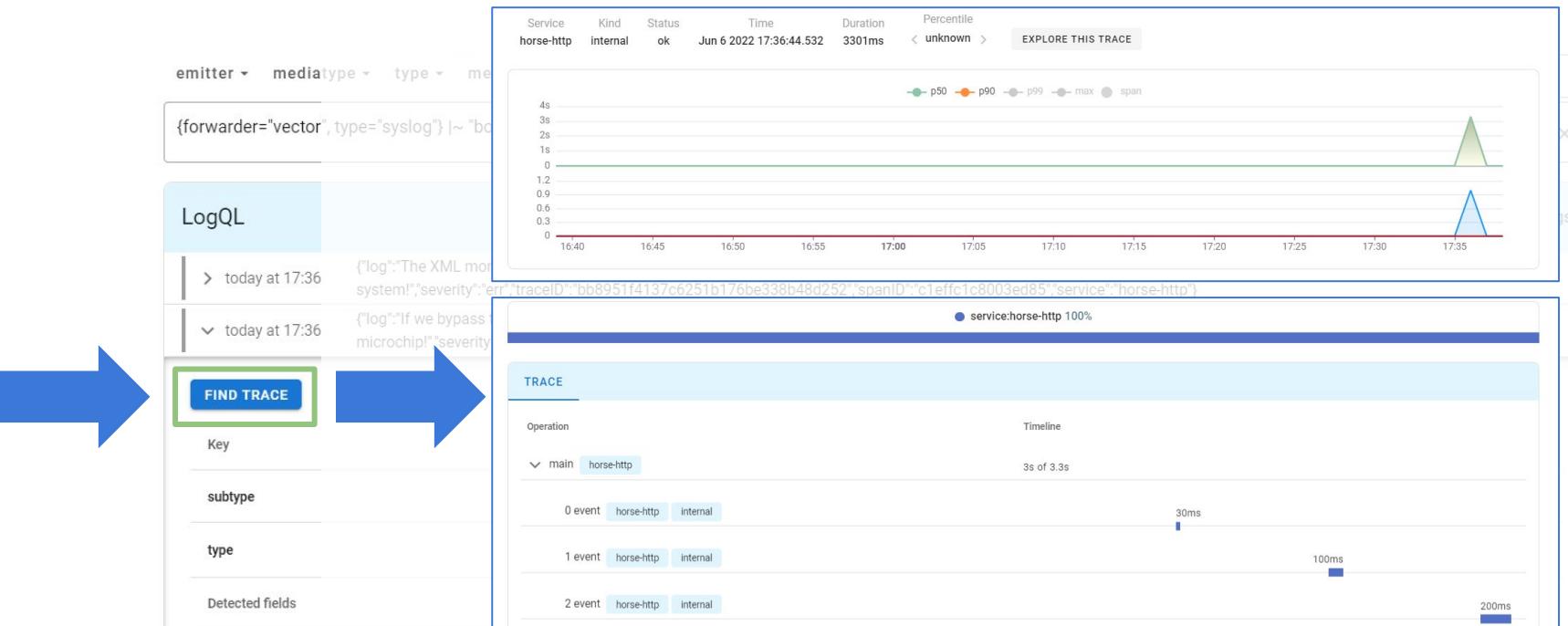
Key	Value
subtype	<input type="text"/> <input type="button"/> log
type	<input type="text"/> <input type="button"/> fake

Detected fields





On top of the existing functionality **Uptrace** can now also automatically correlate filtered Logs to Telemetry Spans and/or Collected Metrics in a *single click*



The screenshot illustrates the Uptrace interface for correlating logs and metrics. On the left, a sidebar shows a search bar with a LogQL query: `{forwarder="vector", type="syslog"} |~ "bo`. Below it are sections for **LogQL**, **Key**, **subtype**, **type**, and **Detected fields**. Two large blue arrows point from the **FIND TRACE** button in the sidebar to the main trace visualization on the right. The main area displays a timeline from 16:40 to 17:35. At the top, service information is shown: **horse-http**, **internal**, **ok**, **Time Jun 6 2022 17:36:44.532**, **Duration 3301ms**, and percentile markers (**p50**, **p90**, **p99**, **max**, **span**). A green shaded area represents the metric distribution. Below the timeline, a log entry is shown: `("log": "The XML model system!", "severity": "err", "traceID": "bb8951f4137c6251b176be338b48d252", "spanID": "c1effc1c8003ed85", "service": "horse-http")`. The bottom section, titled **TRACE**, shows a timeline with three events: **0 event horse-http internal** (30ms), **1 event horse-http internal** (100ms), and **2 event horse-http internal** (200ms). The entire interface is framed by a blue border.



On top of the existing functionality **Uptrace** can now also automatically correlate filtered Logs to Telemetry Spans and/or Collected Metrics in a *single click*

A screenshot of the Uptrace web interface. At the top, there's a navigation bar with tabs for Overview, Explore, Logs (selected), Metrics, Compare, and Alerts (with 64 notifications). A search bar says "Search or jump to trace id...". Below the navigation is a toolbar with a dropdown set to "log-all", a timestamp selector from "1 HOUR", and a "RELOAD" button. The main area has sections for "GROUPS", "LOGS" (selected), and "LOGQL". The LOGQL section contains a search bar with the query "(subtype='log') i json" and a "Limit" dropdown set to 1000. Below this is a "LogQL" section with a "3 logs" summary. Three log entries are listed, each with a timestamp and a detailed log message. The first log is orange and says "You can't index the protocol without transmitting the solid state RAM card!". The second and third logs are green and say "Try to compress the XML bus, maybe it will hack the online monitor!" and "You can't override the alarm without indexing the solid state SAS microchip!", respectively. All logs mention a traceID and spanID.



There's more! How about searching uptrace through Grafana using LogQL + Tempo APIs?

(Tempo-Uptrace)

Query type: Search - Beta, TraceID, JSON file, Loki Search

Beta: Tempo search is currently in beta.

Service Name: serve

Span Name: flush-spans

Tags: http.status_code=200 error=true

Min Duration: e.g. 1.2s, 100ms

Max Duration: e.g. 1.2s, 100ms

Limit: ①

+ Add query, ⏪ Query history, ⏴ Inspector

Table

Trace ID	Trace name	Start time	Duration
0bf498e44bae234d3b49f2c2830e8...	serve flush-spans	less than 20 seconds ago	345 ms
76ca6f0c779c2f9006ec9f73e18a...	serve flush-spans	half a minute ago	403 ms
f176e7179264e597cd861538f4da5ff...	serve flush-spans	half a minute ago	451 ms
8286489c9ea895d9bca1adef8acc...	serve flush-spans	less than a minute ago	497 ms
6519b0a280b336e20906326374b4...	serve flush-spans	about 1 hour ago	343 ms
6133adeb9c800c74722253549f7d...	serve flush-spans	about 1 hour ago	543 ms
dabe9d1828bf6567fddfaa11c1027...	serve flush-spans	about 1 hour ago	340 ms
212770f151ccc4cb11bd0472c516...	serve flush-spans	about 1 hour ago	344 ms



(Tempo-Uptrace)

Query type: Search - Beta, TraceID, JSON file, Loki Search

Trace ID: f176e7179264e597cd861538f4da5ffb

+ Add query, ⏪ Query history, ⏴ Inspector

> Node graph (Beta)

Trace View

serve: flush-spans f176e7179264e597cd861538f4da5ffb

Trace Start: 2022-06-06 17:33:07.776 Duration: 450.85ms Services: 1 Depth: 2 Total Spans: 3

0μs 112.71ms 225.42ms 338.14ms 450.85ms

Service & O...	0μs	112.71ms	225.42ms	338.14ms	450.85ms
serve	450.85ms	112.71ms	225.42ms	338.14ms	450.85ms
serve	INSERT INTO "spans_dat...				2.74ms

INSERT INTO "spans_data_buffer" ("trace_id", "id", "parent_id", "time", "data") VALUES

Service: serve, Duration: 2.74ms, Start Time: 443.53ms

> Tags: code.filepath = github.com/uptrace/uptrace/pkg/tracing/otlp_trace_grpc.go | code.function = tracing...

> Process: service.name = serve | host.name = abb5f940ea08

SpanID: 4218fd92a2c4626b, 4.52ms



You can access your Telemetry, Logs and Metrics without having to install any **Grafana** plugin

A (cloki-uptrace-demo-8)

Log browser > {`type="syslog"`}

Query type Range Instant Line limit ① auto Resolution ② 1/1 ③

+ Add query ⌂ Query history ⌂ Inspector

Logs

This datasource does not support full-range histograms. The graph is based on the logs seen in the response.

Time Unique labels Wrap lines Prettify JSON Dedup None Exact Numbers Signature

Flip results order

Common labels: `vector syslog` Line limit: 1000 reached, received logs cover 27.78% (16min 40sec) of your selected time range (1h)

```
|> 2022-06-06 18:18:06 {"appname":"devankoshal","facility":"ftp","hostname":"some.org","message":"You're not gonna believe what just happened","msgid":ID357,"procid":5609,"severity":"info","version":2}
|> 2022-06-06 18:18:05 {"appname":"shaneIx0","facility":"local0","hostname":"random.com","message":"There's a breach in the warp core, captain","msgid":ID471,"procid":1687,"severity":"notice","version":2}
|> 2022-06-06 18:18:04 {"appname":"jessedddy","facility":"news","hostname":"random.net","message":"Take a breath, let it go, walk away","msgid":ID794,"procid":3411,"severity":"notice","version":1}
|> 2022-06-06 18:18:03 {"appname":"ahmadajmi","facility":"cron","hostname":"for.de","message":"Pretty pretty pretty good","msgid":ID882,"procid":1479,"severity":"notice","version":1}
|> 2022-06-06 18:18:02 {"appname":"jessedddy","facility":"uucp","hostname":"up.org","message":"#hugops to everyone who has to deal with this","msgid":ID75,"procid":8235,"severity":"debug","version":2}
|> 2022-06-06 18:18:01 {"appname":"ahmadajmi","facility":"local13","hostname":"random.de","message":"You're not gonna believe what just happened","msgid":ID802,"procid":7429,"severity":"crit","version":1}
|> 2022-06-06 18:18:00 {"appname":"Karimmove","facility":"auth","hostname":"names.de","message":"A bug was encountered but not in Vector, which doesn't have bugs","msgid":ID911,"procid":9830,"severity":"alert","version":1}
```

Start of range 18:18:07 - 18:01:16



Uptrace

... including our favourite features allowing us to extract metrics from just about *anywhere*

Log browser > {type="clickhouse"} |~"Reserving" | regexp "Reserving (?<token>\d+\.\d+)"

Query type Range Instant Line limit auto

Common labels: clickhouse Line limit: 1000 reached, received logs cover 15.19% (4min 33sec) of your selected time range (30min)

> 2021-10-26 14:05:49 2021.10.26 14:05:48.595952 [31198] {282af02f-9e7d-42d4-bd3e-a7a62de68f1d} <Debug> DiskLocal: Reserving 1.00 MiB on disk `default`, having unreserved 1.45 Ti B.
> 2021-10-26 14:05:49 2021.10.26 14:05:48.605604 [31198] {1fb0d78b-3315-4ec2-b116-85d13976a8e4} <Debug> DiskLocal: Reserving 1.00 MiB on disk `default`, having unreserved 1.45 Ti B.
> 2021-10-26 14:05:48 2021.10.26 14:05:48.004592 [31198] {c2e8ae8d-8217-4f62-bdd5-87eb9b544a58} <Debug> DiskLocal: Reserving 1.00 MiB on disk `default`, having unreserved 1.45 Ti B.
▼ 2021-10-26 14:05:48 2021.10.26 14:05:47.308224 [4800] {} <Debug> DiskLocal: Reserving 1.32 MiB on disk `default`, having unreserved 1.45 TiB.

Start of range 14:05:51
14:01:15

Log labels

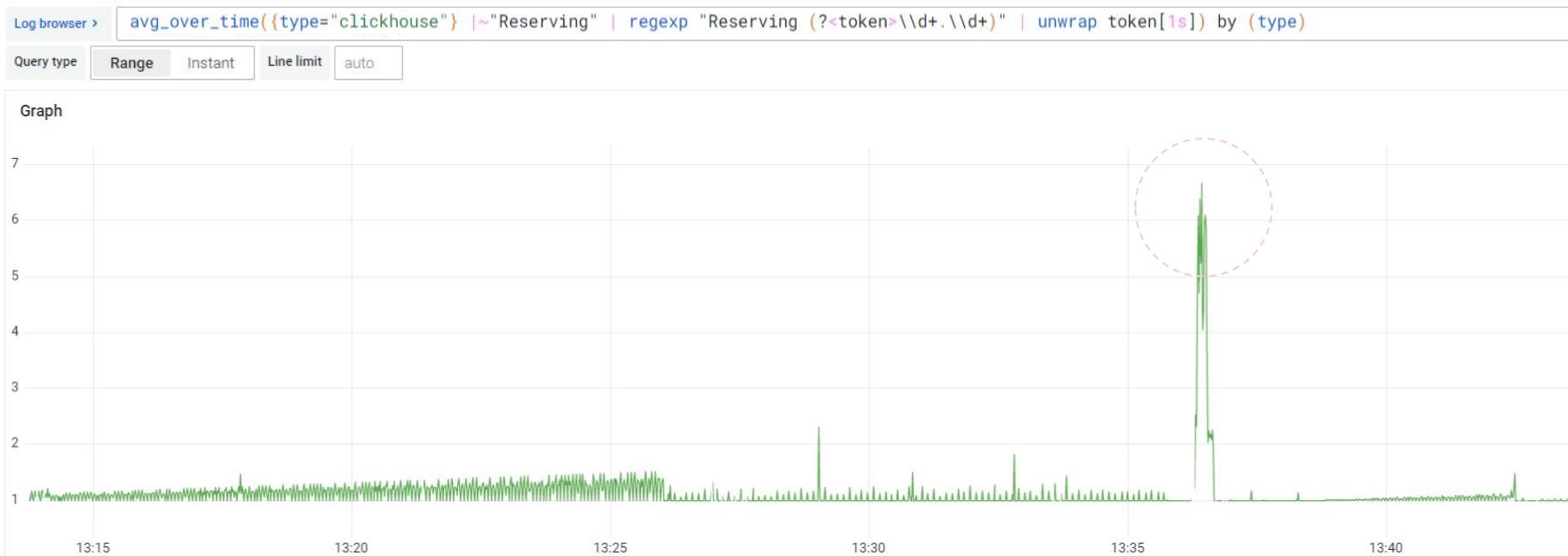
all	⊕	⊖	level	Debug
all	⊕	⊖	pid	4800
all	⊕	⊖	token	1.32
all	⊕	⊖	type	clickhouse

Detected fields ⓘ

all	⌚	ts	2021-10-26T12:05:48.230Z
all	⌚	tsNs	1635249948230000000



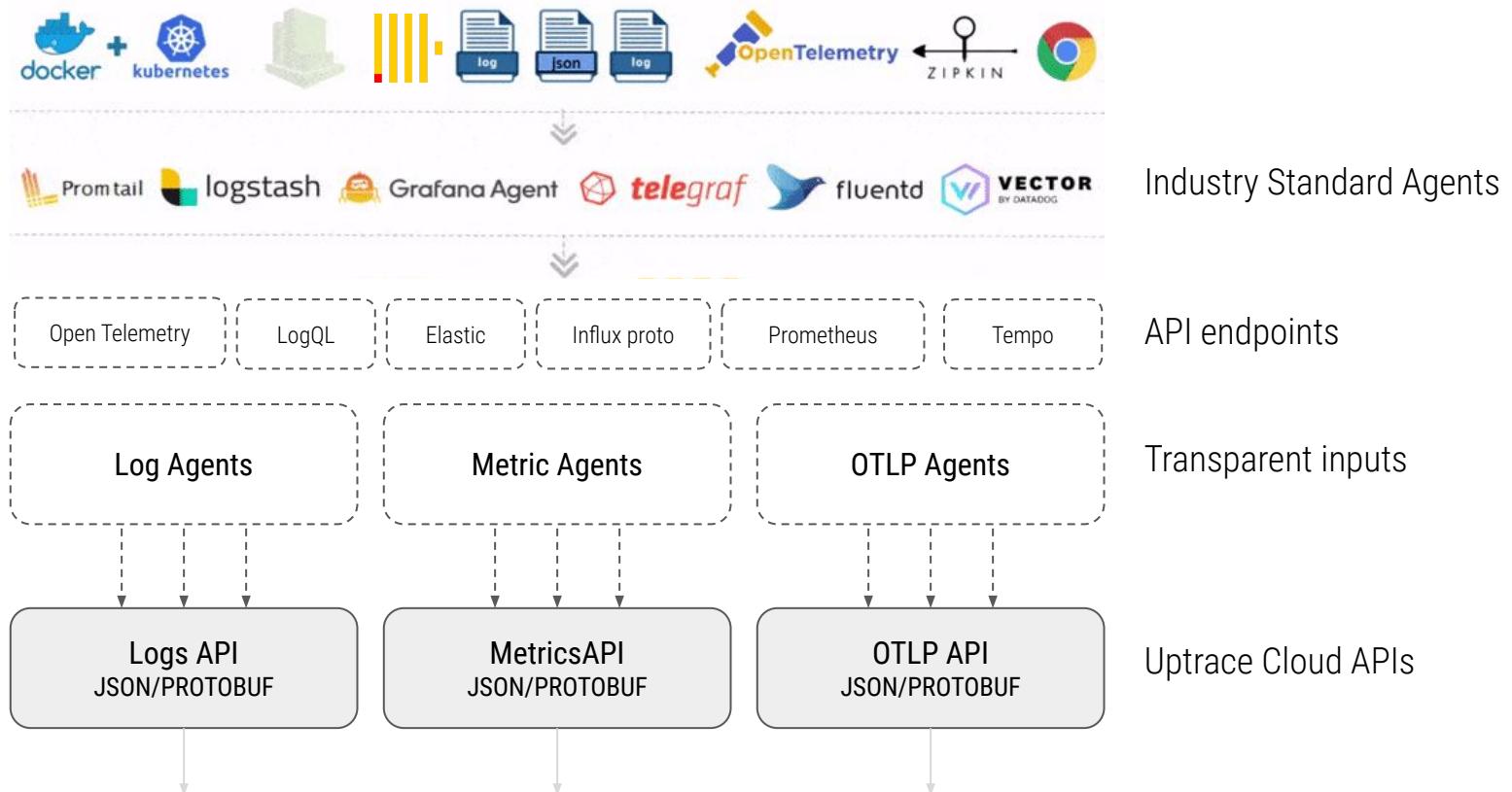
"How much disk memory was ClickHouse reserving between 13:13 and 13:40?" - boom, done!



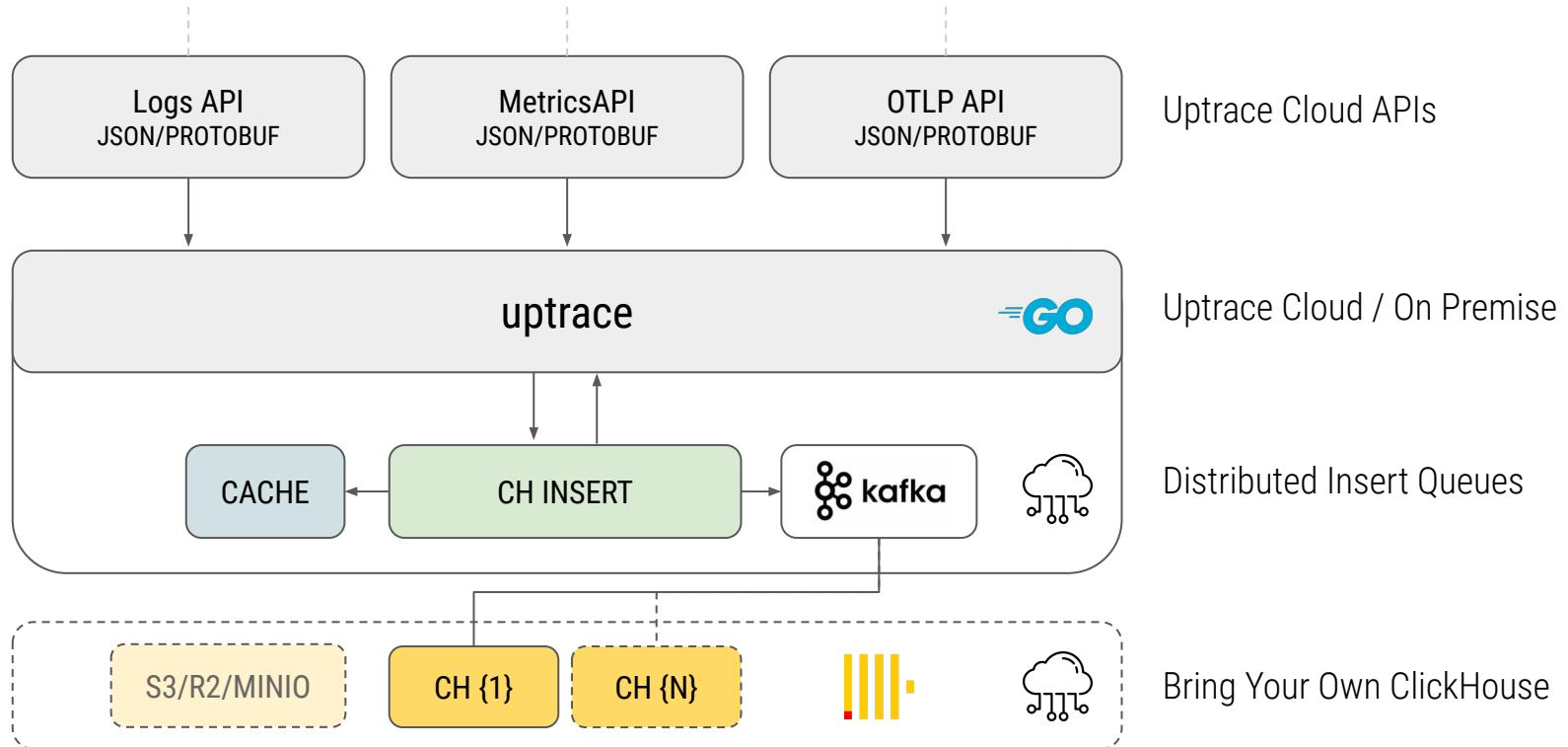
High Level Design



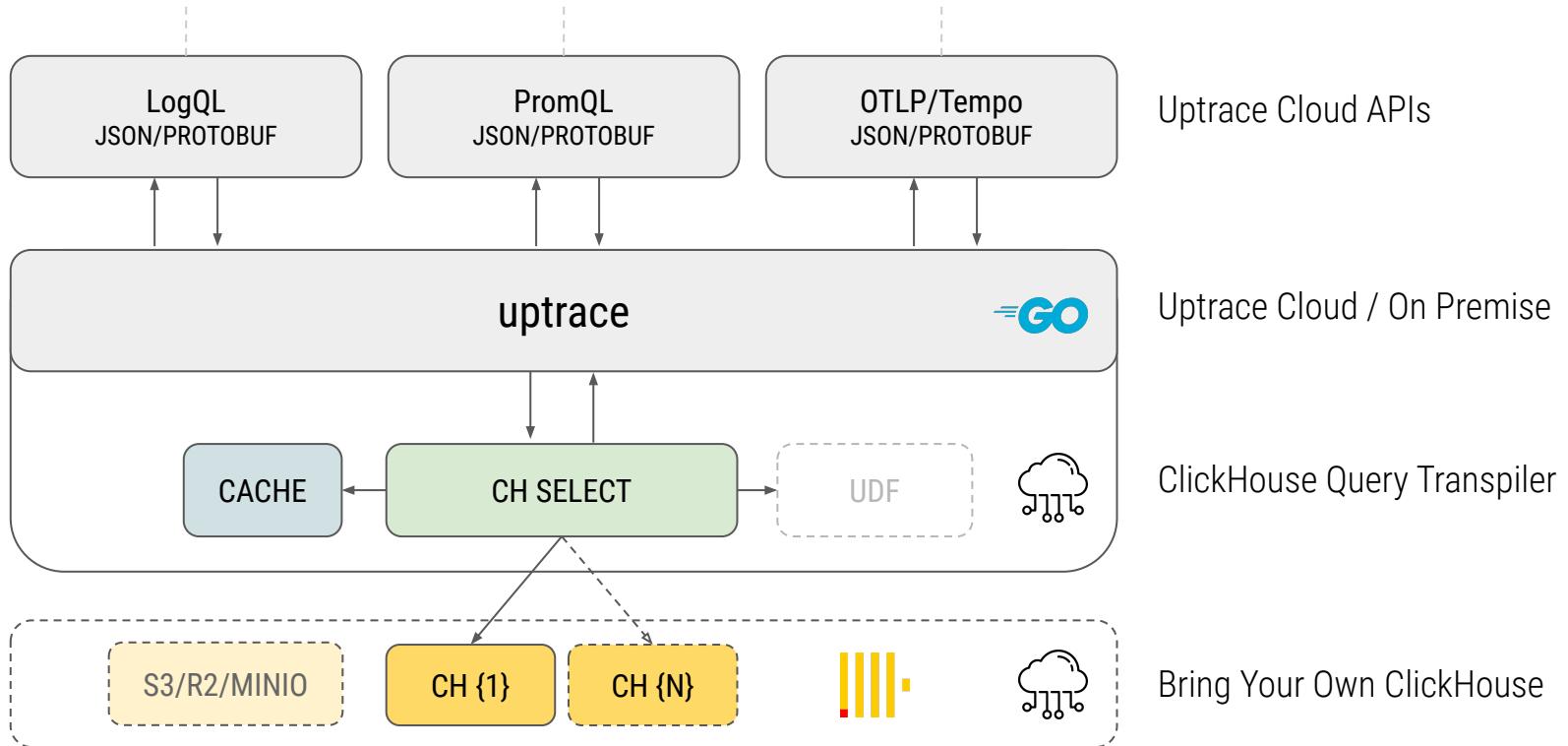
Uptrace: Agents & APIs



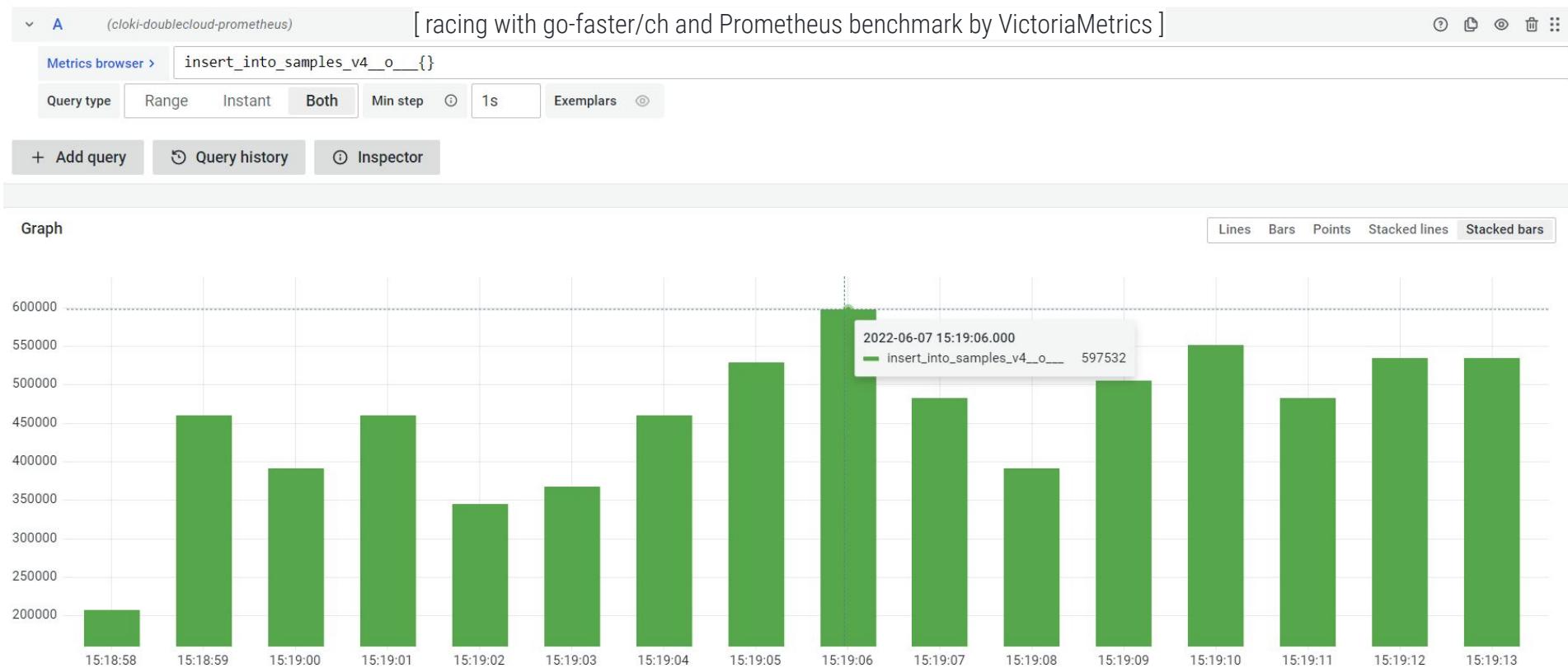
Uptrace: Insert Pipeline



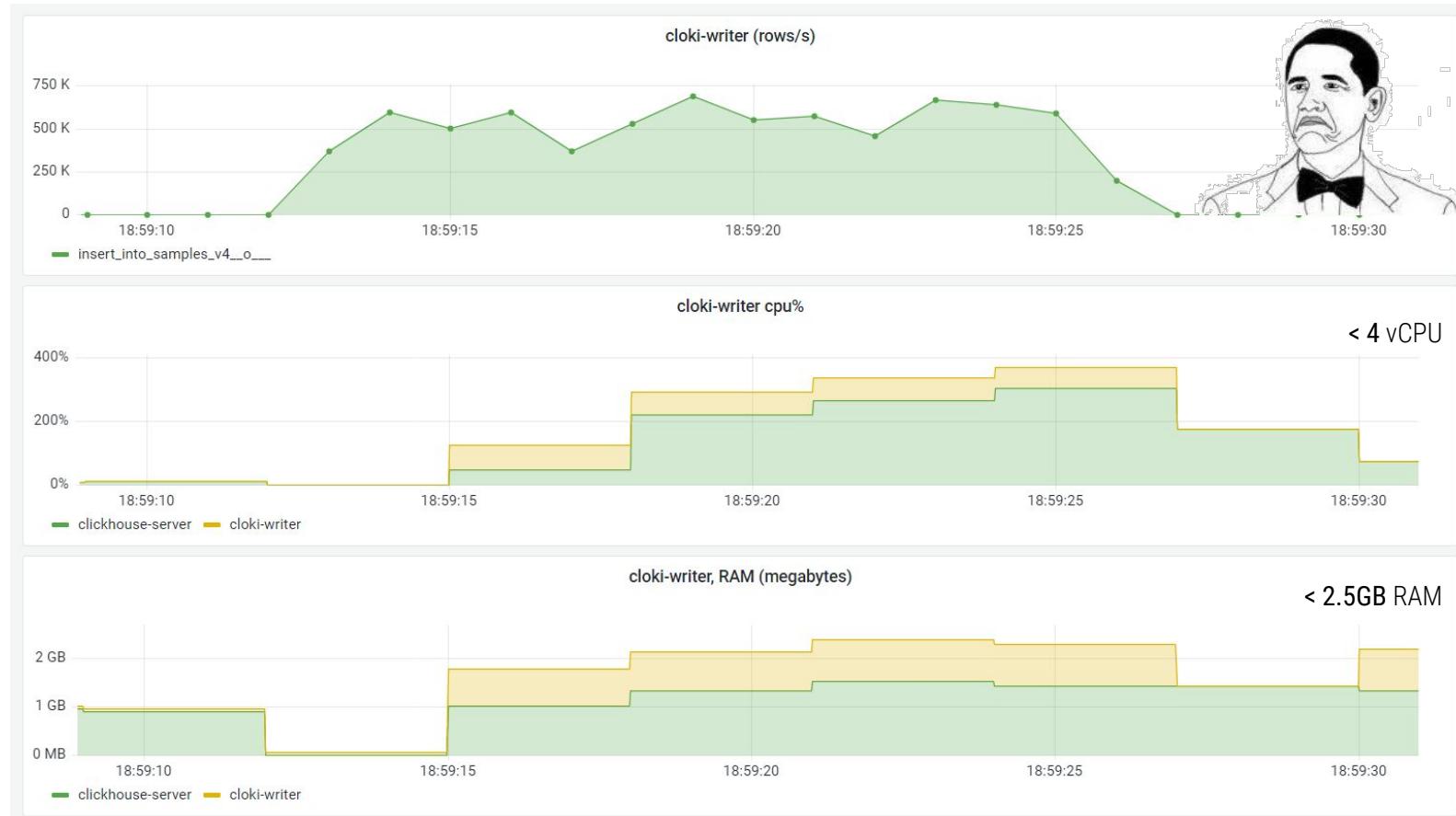
Uptrace: Query Pipeline



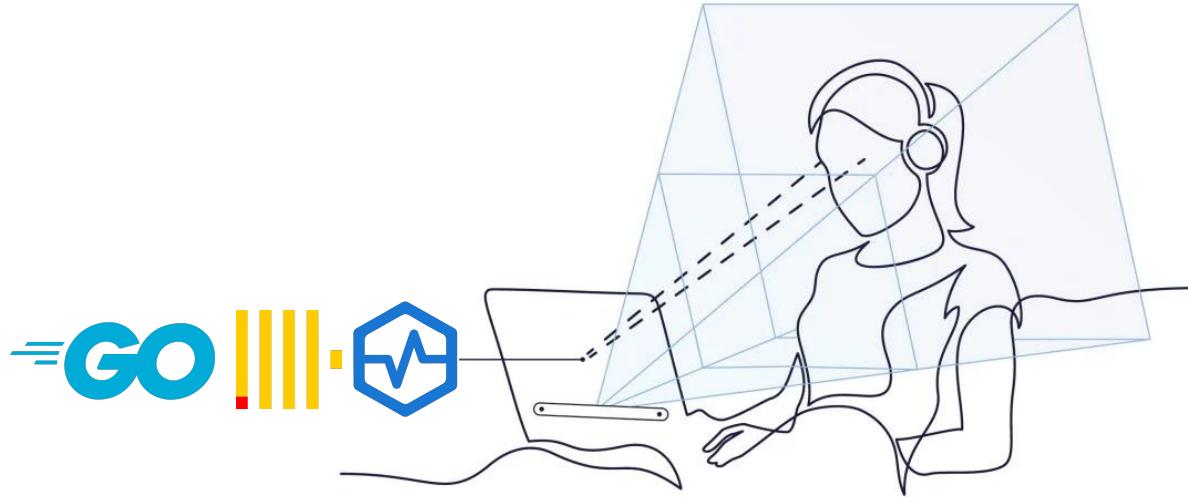
ClickHouse INSERT performance up to ~500k samples/thread/second

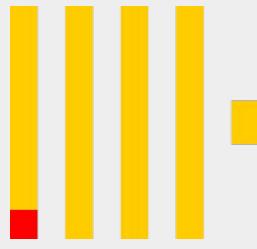


ClickHouse INSERT performance up to ~500k samples/thread/second



Under the Hood





ClickHouse

Always has been

Wait, is it all based on **ClickHouse**?!



Favourite ClickHouse Features @uptrace

Our team is very passionate about ClickHouse and we spend *lots* of time discussing techniques and performance, living and breathing all the material published by its vibrant community and by awesome companies such as Altinity

Here's a brief list of the features we love to use everyday:

- Data Sampling
- Tiered storage and S3
- Codecs & Compression
- Distributed and Merge tables
- Distributed Tracing (MV + URL Engine)
- ClickHouse-Mate

... and last but not least, a brief look at our **ROADMAP** for 2022-23!

ClickHouse Sampling

Get approximated results **fast** by processing a fraction of data

```
CREATE TABLE sampling_test ( id UInt64, group_id UInt64, sum UInt64 )
ENGINE = MergeTree()
ORDER BY (group_id, intHash32(id))
SAMPLE BY intHash32(id)
```

Without Sampling

```
SELECT sum(sum) FROM sampling_test WHERE group_id != 0
└── sum(sum) ──
| 4950000000 |
└─────────────
Elapsed: 1.201 sec. Processed 1.09 billion rows, 17.42 GB
```

With Sampling

```
SELECT sum(sum * _sample_factor) FROM sampling_test SAMPLE 1/10 WHERE group_id != 0
└── sum(multiply(sum, _sample_factor)) ──
| 4951544350 |
└─────────────
Elapsed: 0.697 sec. Processed 115.38 million rows, 2.77 GB
```

Sample key must

- Be a part of the primary key
- Be uniformly distributed (aka random)
- Come after low cardinality columns in the PK or performance will suffer

Syntax

- SAMPLE 1/10 - sample 10% of data
- SAMPLE 1000000 - sample not less than 1M rows
- SAMPLE 1/10 OFFSET 1/10 - sample the 2nd 10% of data

Tiered Storage and S3

Configure S3 Disk

```
<storage_configuration>
  <disks>
    <s3>
      <type>s3</type>
      <endpoint>http://[BUCKET_NAME].s3.amazonaws.com/[NODE_NAME]</endpoint>
      <access_key_id>[FIXME]</access_key_id>
      <secret_access_key>[FIXME]</secret_access_key>
    </s3>
  </disks>
</storage_configuration>
```

Enable data cache on SSD to optimize r/w performance

```
<storage_configuration>
  <disks>
    <s3>
      <type>s3</type>
      <endpoint>http://[BUCKET_NAME].s3.amazonaws.com/[NODE_NAME]</endpoint>
      <access_key_id>[FIXME]</access_key_id>
      <secret_access_key>[FIXME]</secret_access_key>
      <data_cache_enabled>1</data_cache_enabled>
      <data_cache_max_size>8589934592</data_cache_max_size>
    </s3>
  </disks>
</storage_configuration>
```

Relocate data from SSD to S3 when free space < 5%

```
<policies>
  <tiered>
    <move_factor>0.05</move_factor>
    <volumes>
      <hot>
        <disk>ssd</disk>
      </hot>
      <s3>
        <disk>s3</disk>
        <prefer_not_to_merge>true</prefer_not_to_merge>
      </s3>
    </volumes>
  </tiered>
</policies>
```

Manually relocate data from/to S3 or SSD storage

```
-- upload to s3
ALTER TABLE table1 MOVE PARTITION '20220102' TO VOLUME 's3';

-- download from s3 if you changed your mind
ALTER TABLE table1 MOVE PARTITION '20220102' TO VOLUME 'ssd';
```

ClickHouse Codecs & Compression

ClickHouse **codecs** are a *must-have* to efficiently encode numeric values at scale:

- **DoubleDelta** for monotonic numbers
- **Delta** for mostly increasing/decreasing numbers
- **Gorilla** for floats
- **T64** for random-like numbers
- **None** don't use a codec if numbers are truly randomized

ClickHouse should use **LZ4 compression** for small partitions and **ZSTD** for larger ones:

```
<compression incl="clickhouse_compression">
    <!-- Use ZSTD(3) for parts >= 64MB -->
    <case>
        <method>zstd</method>
        <level>3</level>
        <min_part_size>67108864</min_part_size>
        <min_part_size_ratio>0.001</min_part_size_ratio>
    </case>

    <!-- Add more cases for ZSTD(5) etc -->
</compression>
```

- The average uncompressed span size is ~1KB
- ClickHouse can compress that data down to 40 bytes (25x)
- To get best results, use codecs and ZSTD compression levels

Use **Default codec** to specify the default compression method, for example: `Codec(Delta, Default).`

ClickHouse Table Engines

Distributed Table Engine

You can have chunks of data on different servers and "join" them together using a distributed table engine:

```
CREATE TABLE table1_dist AS table1
ENGINE = Distributed(cluster1, currentDatabase(), table1)
```

Selecting from **table1_dist** will query all servers defined in the **cluster1**

Merge Table Engine

Merge tables allows reading from multiple local tables simultaneously.

```
CREATE TABLE table_merge AS table1
ENGINE = Merge(currentDatabase(), '^table1|table2|table3$');
```

Super useful for updating your schema without copying old data around

ClickHouse GIN Indexing

Fast filtering JSON data by key-value pairs during fingerprint selection

```
CREATE TABLE IF NOT EXISTS time_series_v2 (
    date Date,
    fingerprint UInt64, //the same ID
    labels String, //we know about JSONB, it's under our research for now :)
) ENGINE = ReplacingMergeTree(date) PARTITION BY (date) ORDER BY (fingerprint);

CREATE MATERIALIZED VIEW IF NOT EXISTS time_series_gin_v2
ENGINE = ReplacingMergeTree()
AS SELECT date, fingerprint, pairs.1 as key, pairs.2 as val,
FROM time_series_v2
ARRAY JOIN JSONExtractKeysAndValues(time_series_v2.labels, 'String') as pair
ORDER BY key, val, fingerprint
```

NO IDX (visitParamExtractString)

```
SELECT * FROM time_series_v2
WHERE (visitParamExtractString(labels, 'key_1_1') = 'val1_1')
    AND (visitParamExtractString(labels, 'key_2_1') = 'val2_1')

1 rows in set. Elapsed: 0.072 sec. Processed 3.00 million rows, 214.57 MB (41.91 million rows/s., 3.00 GB/s.)
```

With IDX

```
SELECT * FROM time_series_v2
WHERE fingerprint IN (
    SELECT fingerprint FROM time_series_gin_v2 ANY INNER JOIN (
        SELECT fingerprint FROM time_series_gin_v2 WHERE (key = 'key_2_1') AND (val = 'val2_1')
    ) AS idx_2
    ON time_series_gin_v2.fingerprint = idx_2.fingerprint
    PREWHERE (key = 'key_1_1') AND (val = 'val1_1')
)
1 rows in set. Elapsed: 0.024 sec. Processed 41.26 thousand rows, 1.18 MB (1.74 million rows/s., 49.56 MB/s.)
```

ClickHouse Distributed Tracing

Send ClickHouse spans from the `opentelemetry_span_log` table to Uptrace API using Materialize View and URL Engine

```
CREATE MATERIALIZED VIEW default.zipkin_spans
ENGINE = URL('https://api.uptrace.dev/api/v2/spans', 'JSONEachRow')
SETTINGS output_format_json_named_tuples_as_objects = 1,
        output_format_json_array_of_rows = 1 AS
SELECT
    lower(hex(trace_id)) AS traceId,
    case when parent_span_id = 0 then '' else lower(hex(parent_span_id)) end AS parentId,
    lower(hex(span_id)) AS id,
    operation_name AS name,
    start_time_us AS timestamp,
    finish_time_us - start_time_us AS duration,
    cast(tuple('clickhouse'), 'Tuple(serviceName text)') AS localEndpoint,
    attribute AS tags
FROM system.opentelemetry_span_log
```



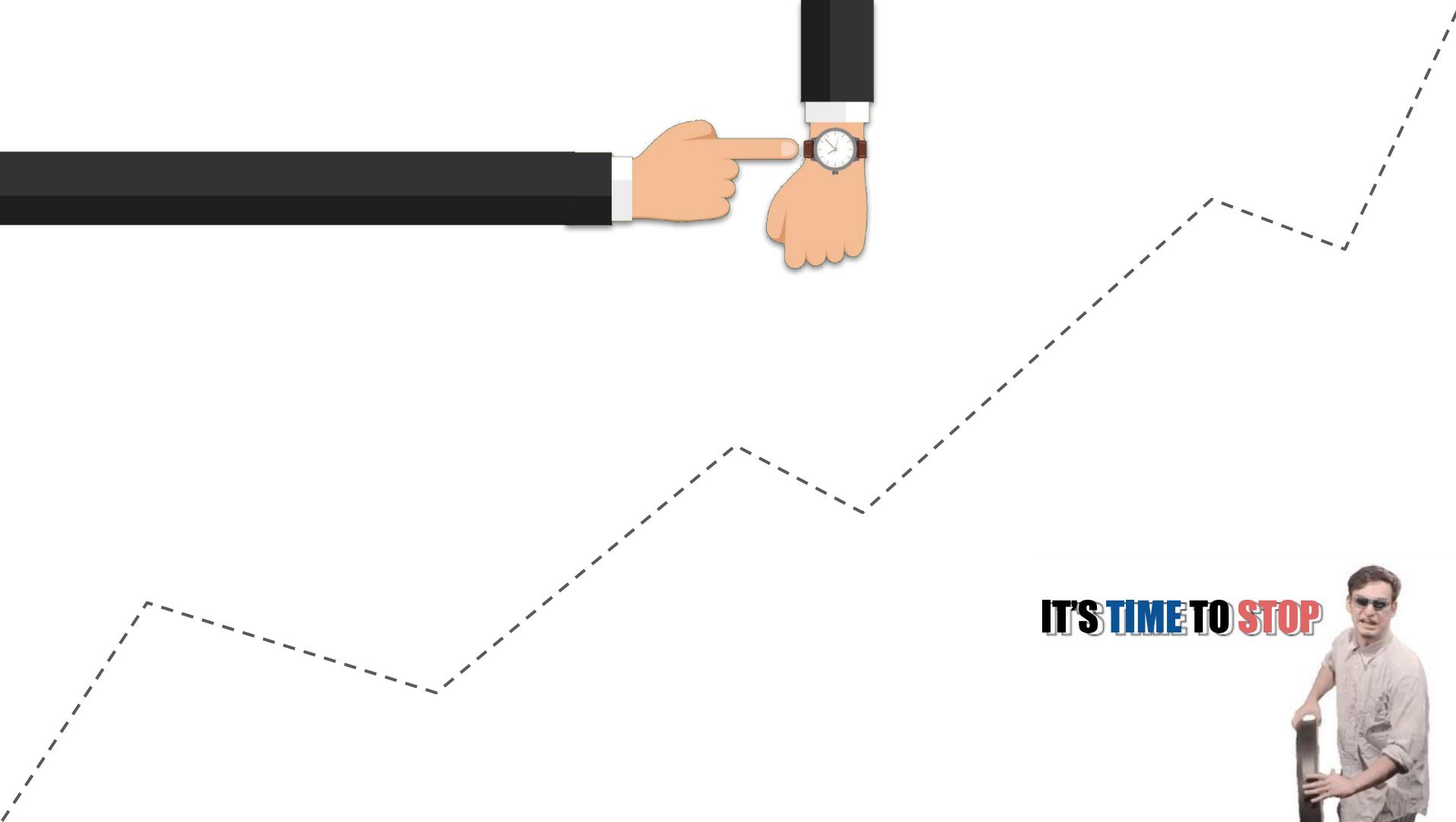
ClickHouse Mate

<https://metrico.github.io/clickhouse-mate>

Running queries all day? Try out *ClickHouse-Mate* the first "play" webclient written under bomb attacks by our brave **Ukrainian** team

Roadmap

- Multi-Tenancy (done)
- PromQL API support for OpenTelemetry Metrics (in progress)
- JSON datatype option for semi structured data (in progress)
- Additional Input formats (what would you like to see?)
- Extend AlertManager API emulator
- External CH pull agent w/ custom Exporting and pseudo URL Engine
- Extended Provisioning APIs for Uptrace Cloud



IT'S TIME TO STOP

Join the Uptrace Party! 🎉

[SIGN IN](#)[TRY IT FREE](#)

<https://uptrace.dev>

Unlimited users, services, and hosts

Free

\$0

50 gigabytes

500 timeseries

[SUBSCRIBE](#)

Personal

\$30

200 gigabytes

2000 timeseries

\$0.13 per gigabyte

[SUBSCRIBE](#)

Team

\$100

800 gigabytes

8,000 timeseries

\$0.12 per gigabyte

[SUBSCRIBE](#)



On Premise

custom

unlimited gigabytes

unlimited timeseries

unlimited retention

[CONTACT US](#)

That's all Folks!

JOIN US

<https://uptrace.dev> ★ <https://docs.uptrace.dev> ★ <https://uptrace.cloud/matrix>

ClickHouse



uptrace



Demo Links

[Uptrace Demo](#)

[Performance Test 200k/s](#)

[LogQL Grafana Syslog \(demo\)](#)

[Tempo Grafana Uptrace \(demo\)](#)

[Zipkin Test Sender \(demo\)](#)

[ClickHouse Mate \(demo\)](#)

Clickhouse VS GIN index

Task:

- Lots of plain JSON objects
- We want to filter by key:value pairs fast

In PostgreSQL it would have been done like that:

```
CREATE TABLE jsons (
    fingerprint int8, //like ID
    obj JSONB
);
CREATE INDEX jsons_gin_idx ON jsons USINGgin (obj);
SELECT * FROM jsons WHERE obj @> '{"key1":"val1", "key2":"val2"}';
```

Clickhouse VS GIN index (2)

In Clickhouse it transformed into the next:

```
CREATE TABLE IF NOT EXISTS time_series_v2 (
    date Date,
    fingerprint UInt64, //the same ID
    labels String, //we know about JSONB, it's under our research for now :)
) ENGINE = ReplacingMergeTree(date) PARTITION BY (date) ORDER BY (fingerprint);

CREATE MATERIALIZED VIEW IF NOT EXISTS time_series_gin_v2
ENGINE = ReplacingMergeTree()
AS SELECT date, fingerprint, pairs.1 as key, pairs.2 as val,
FROM time_series_v2
    ARRAY JOIN JSONExtractKeysAndValues(time_series_v2.labels, 'String') as pair
ORDER BY key, val, fingerprint
```

After some research the fastest lookup request is:

```
SELECT * FROM time_series_v2 WHERE fingerprint IN (
    SELECT fingerprint FROM time_series_gin_v2 INNER ANY JOIN (
        SELECT fingerprint FROM time_series_gin_v2 WHERE key='key2' AND val='val2',
    ) as idx_2 ON time_series_gin_v2.fingerprint = idx_2.fingerprint
    PREWHERE key='key1' AND val='val1'
)
```

Clickhouse VS GIN index: comparison

Insert sample data:

```
INSERT INTO time_series_v2 (date, fingerprint, labels) SELECT
    toDate(NOW()) AS date,
    number AS fingerprint,
    format('{{"key_1_{0}":"val1_{0}", "key_2_{0}":"val2_{0}"}}', toString(number)) AS labels
FROM numbers(3000000)
```

NO IDX (visitParamExtractString):

```
SELECT * FROM time_series_v2
WHERE (visitParamExtractString(labels, 'key_1_1') = 'val1_1') AND (visitParamExtractString(labels, 'key_2_1')
= 'val2_1')
1 rows in set. Elapsed: 0.072 sec. Processed 3.00 million rows, 214.57 MB (41.91 million rows/s., 3.00 GB/s.)
```

With IDX:

```
SELECT * FROM time_series_v2
WHERE fingerprint IN (
    SELECT fingerprint FROM time_series_gin_v2 ANY INNER JOIN
        (SELECT fingerprint FROM time_series_gin_v2 WHERE (key = 'key_2_1') AND (val = 'val2_1')) AS idx_2
    ON time_series_gin_v2.fingerprint = idx_2.fingerprint
    PREWHERE (key = 'key_1_1') AND (val = 'val1_1')
)
1 rows in set. Elapsed: 0.024 sec. Processed 41.26 thousand rows, 1.18 MB (1.74 million rows/s., 49.56 MB/s.)
```