
KubeCon
CloudNativeCon
Europe 2025
LONDON BUS

KubeCon | CloudNativeCon
Europe 2025

Designing an Observability Query Language

Alolita Sharma - Observability Engineering Lead at Apple
Christopher Larsen - Observability Engineer at Netflix
Pereira Braga - Observability Technical Steward at Google

Who are we? How we got together?

Alolita - CNCF Observability TAG Co-chair supporting QLS WG effort in CNCF

Chris - Author and lead of TAG OBS Query Standardization WG. A working group to research and analyze existing observability query languages with the goal of recommending a standard, unified language for following teams and projects to implement.

Pereira - Led effort in Google to run similar analysis across Google to identify an observability query language focusing on telemetry query for real-time and analytics needs aligned with other observability data and long-term focus on observability data lake and now working on the execution of extra operators and disseminate usage across the company.

Focus of this session

In this talk we are going to propose a **recommendation** to standardize a common query method for observability data.

CNCF Observability TAG is working on research around an open observability query semantic definition.

Google is working on open source SQL extensions with pipe syntax and timeseries operators to well support observability queries and share with the community.

Terminology

Metrics/Telemetry - Timeseries based measurements, we use those interchangeably in these slides.

Logs - Unstructured or semi structured text.

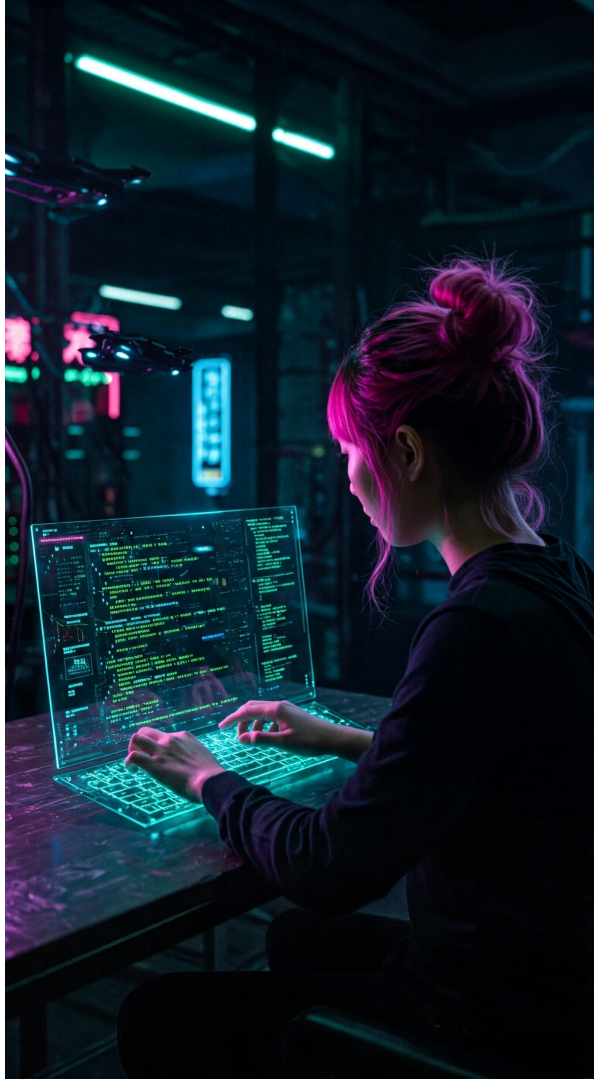
Traces - Distributed traces consisting of spans.

Profiles - Systems profiling, CPU/GPU usage, memory allocation, etc.

Wide Events - Structured logs with additional context.

What Challenges are Developers Facing?

- Pick a general purpose language for your project?
  - Go, Java, Javascript, Rust, etc.
- How do I transfer data between services?
  - JSON, gRPC, Thrift, Protobuf, ...
- How do I store this data?
  - Pick a DB with its *specific query language!*
- How do I deploy this code?
  - Infrastructure as code: Terraform, Puppet, Ansible, Salt



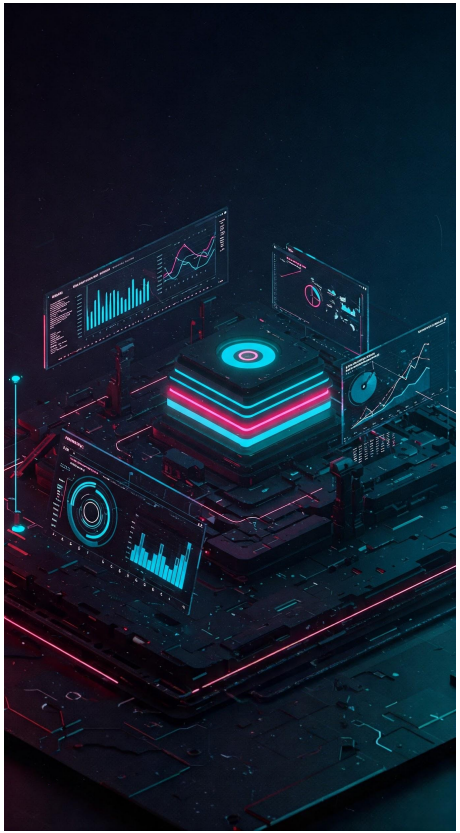
Does my service work?

- Add some Telemetry!
  - Good old printf for logs! Wait, I need to search across instances.
  - Too many logs! Query slow! Add some metrics. How?
  - Metrics are too high level and I can't find out who is calling what!
  - My app is slow, can I get some profiles?
  - OpenTelemetry standardizes instrumentation and collection!

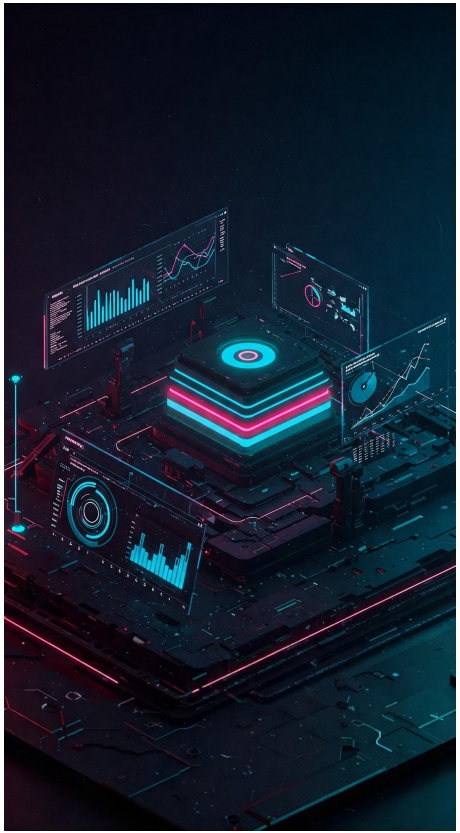
OpenTelemetry

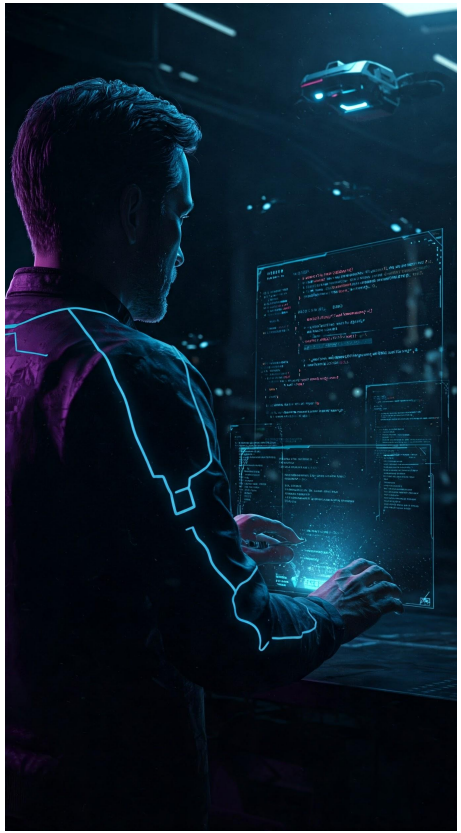
Got my Observability data! What do I do with it?

Got my Observability data! What do I do with it?

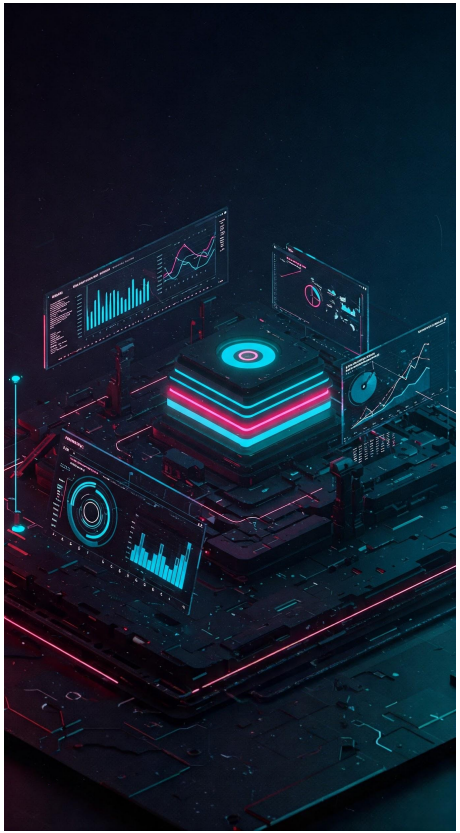
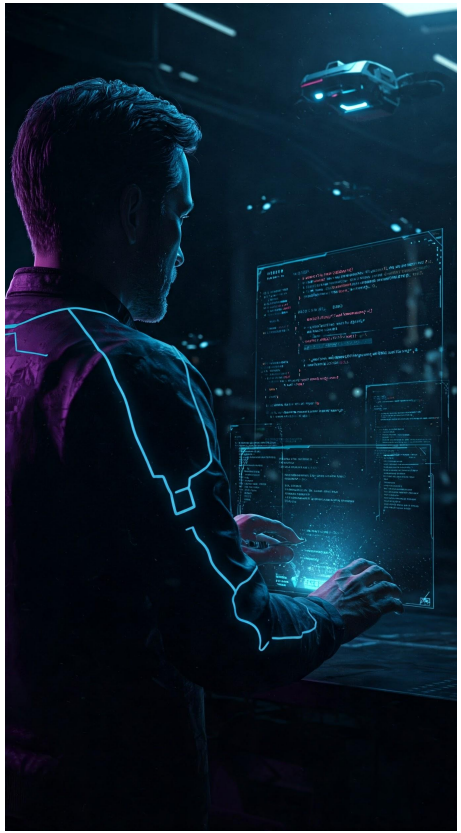
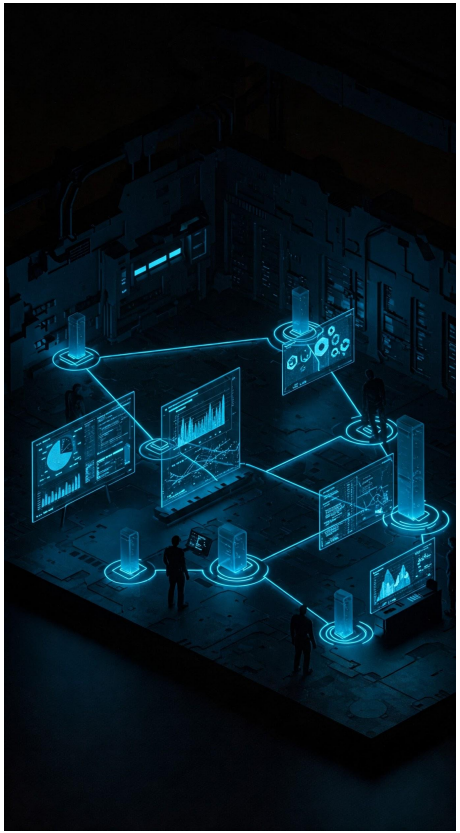


Got my Observability data! What do I do with it?





Got my Observability data! What do I do with it?

Who are Observability Users?

CLOUD NATIVE
COMPUTING FOUNDATION

- Traditionally:
  - Developers wrote something and threw code over the wall.
  - System admins or Site Reliability Engineers (SREs) would run the code and observe it.
- Moving towards:
  - Developers are on-call operators (dev-ops).
  - Deployments are gated on regression analysis (automation).
  - SREs (if present) triage issues and call in the devs.
  - Platform engineers scale based on usage.
  - Security engineers looks for odd behaviors and break ins.
  - AI engineers look to see if resources consumed are worth incremental improvements.
  - Managers, execs, marketing want availability reports, how many customers were affected by an outage, etc.
  - Support engineers help customers (hyperclouds)
  - Customers want to know why their requests failed.

In Summary

- It’s not just SREs using observability any more - whole company.
- The walls between BI data and observability data are mostly gone.
- Engineers have enough on their plates without learning more tools.
- Insights, exploration and understanding should be as easy as possible.
- Balance:
  - Understandability vs terseness
  - Deep analytics vs quick lookups
  - Interoperability vs optimization for specific telemetry

Can’t we just...

- Throw AI at it?
  - Improving, comes at cost and depends on data model.
- Pick *one* telemetry type, like wide events?
  - Expensive at scale.
- Standardize querying and analyzing the data?
  - I’m glad you asked...

What we Looked At

- Interviews with 11 observability specific DSL designers
- Cataloged language features
- Cataloged observability telemetry models
- Cataloged observability use cases

CNCF Observability TAG Github Repo

https://github.com/cncf/tag-observability/tree/main/working-groups/query-standardization

CNCF Observability TAG YouTube Channel

https://www.youtube.com/playlist?list=PLN9G8268O5igu4g7NrlsT2Kh1Ee3ooz08

What we Looked At

Multi-month internal analysis considering:

- User Experience: UXR on existing 3-4 query languages:
    - New Google Engineers (<3 months) and experienced Google Engineers (>2 years).
- Onboarding/Training Costs: Bespoke languages -> very expensive and focused on small Critical User Journeys (CUJs)
    - Added approx 2 weeks per engineer to onboarding only for telemetry needs
- Query Performance: How query language could influence/benefit the query engine alignment.
- Note: Customers of Observability UIs for querying telemetry are overall majority Developers (Not SREs).

Users Quantified

NETFLIX

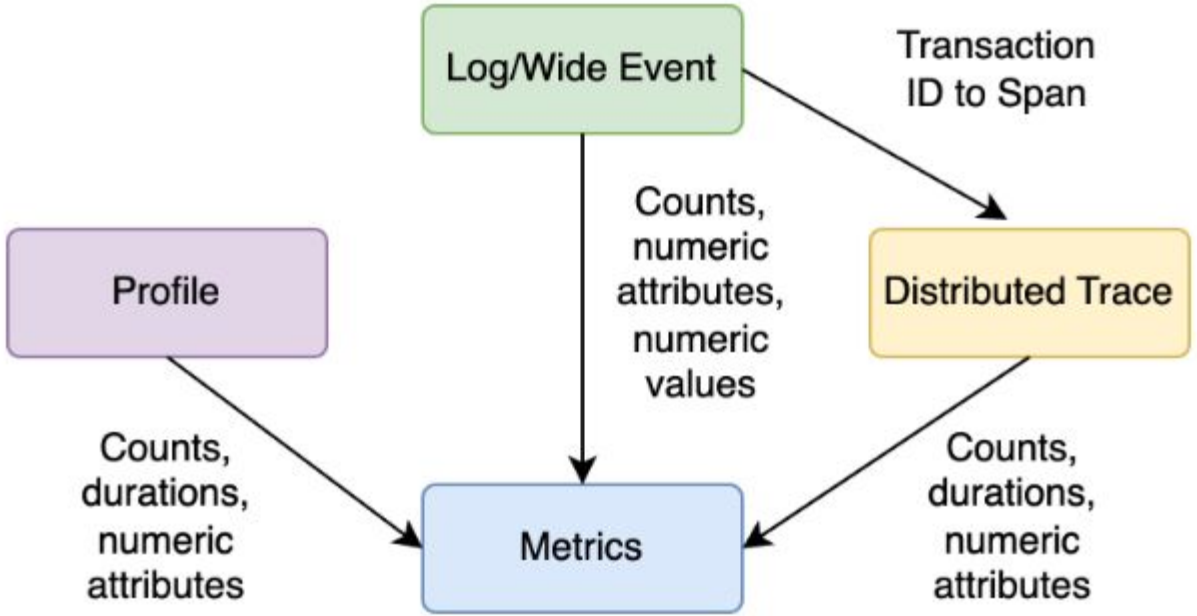
Persona	People Ratio	Support Topics	UI Tool Interactions
Developer	70%	75%	29%
Platform/Library Owners	10%	5%	31%
SREs	7%	8%	14%
Data Scientists	10%	10%	22%
Managers	3%	2%	4%

Telemetry Analysis Hierarchy

CLOUD NATIVE
COMPUTING FOUNDATION

We constantly coerce the data into metrics to understand what's going on.

We even derive traces from logs.



DSL Designer Interviews

CLOUD NATIVE
COMPUTING FOUNDATION

- Interviewed designers of PromQL, LogQL, TraceQL, DataDog QL, Google (Monarch and overall), Kusto Metrics, KX, Lightstep UQL, New Relic, etc.
- From 1 to 10 on how tightly coupled to the data store: **5.3 avg.**
- Most designers felt their language could handle additional telemetry models.
- All had similar predicates for key/value attributes.
- All had implicit or explicit joins on expressions, e.g. *metric1 / metric2*
- All supported similar aggregations, sum, min, max, avg. Most had percentiles.
- A handful support graph predicates.

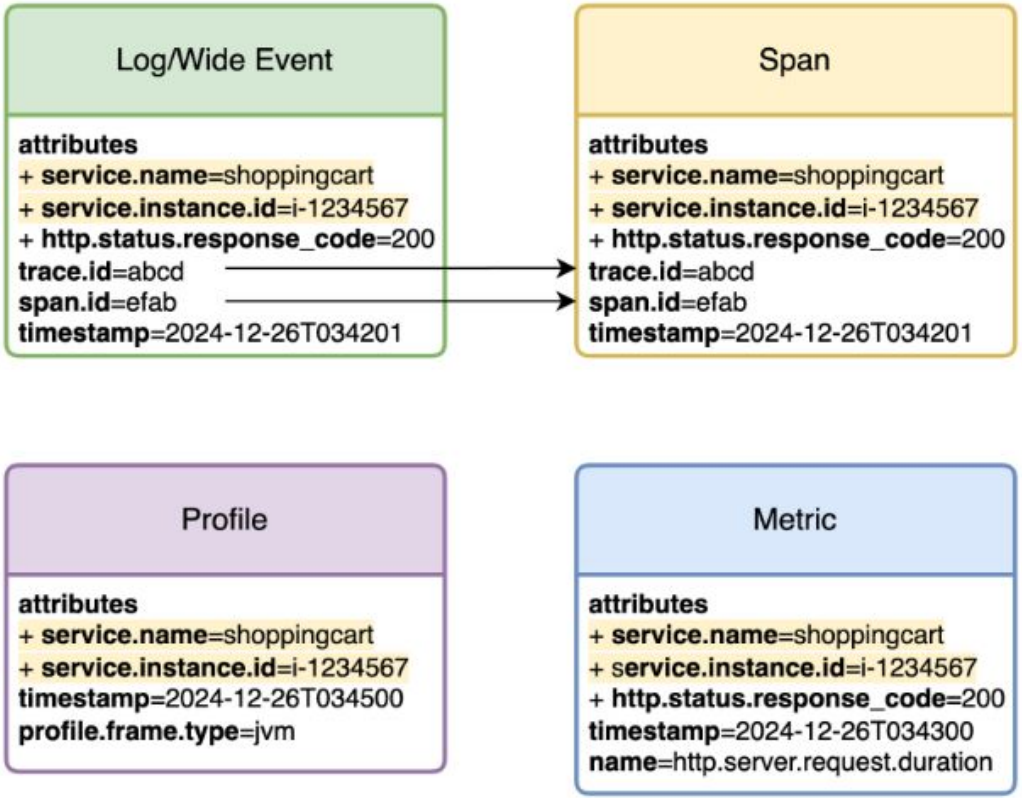
Telemetry data is Relational Data!

CLOUD NATIVE
COMPUTING FOUNDATION

But aren't the telemetry types too different? Metrics != logs != traces != profiles...

NO! They share common attributes.

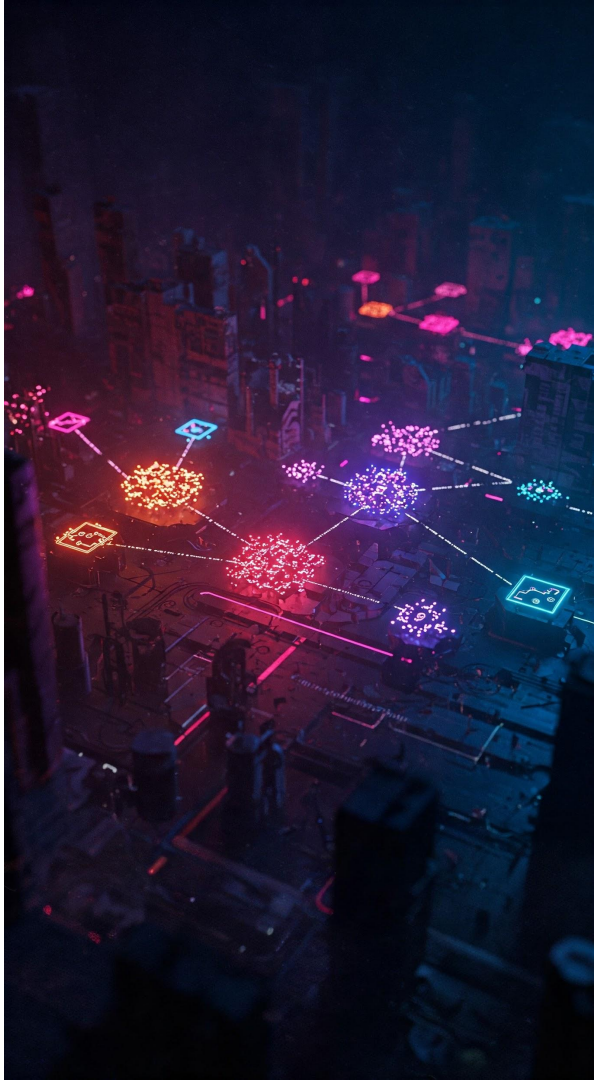
Thus they are relational!



Does any one language support

- Conditional and relational predicates?
- Graph predicates?
- Joins on relations?
- Grouping and aggregations?
- Time intervals/windows?

Well...




SQL

Because...

After a long extensive internal research at Google...

Google decided to align on SQL and extending by:

- supporting telemetry with pipe syntax and timeseries extensions;
- focusing on how to provide a query language (data model) across all Observability data sources (telemetry/metrics, logging and traces).

Because...

- Reduce cost of onboarding/training;
- Federated query joining across:
  - telemetry/metrics data,
  - logging data,
  - trace data,
  - production databases,
  - and other sources,
  - ...all with SQL;
- Unblocking observability data lake (Agentic AI);
- Improve engineering efficiency.

There and Back Again

CLOUD NATIVE COMPUTING FOUNDATION

- The data analytics industry is returning to SQL
- We learned a lot from NoSQL
- BI and operational data are no longer distinct.
- ANSI SQL has evolved!

The Recommendation

- Use a *subset* of SQL *semantics* as a base standard
- Define standard models by type (metric, log/wide event, trace, profile, etc.)
  - Supporting OpenTelemetry models!
- Focus on relational execution engines divorced from *syntax*
- Federate queries with query plan intermediate representation (IR).
- Specify standard functions (syntactic sugar in SQL) to support existing systems and reduce verbosity.
- Create observability gateways that translate between dialects and backends.

Draft Specification and Models (QR Code in future slides)

What is coming now: Pipe syntax

- Aligned with Unix philosophy, modern query languages and APIs.
- Pipes assist with unknown data exploration.
  - Show a sample of data
  - Filter on specifics
  - Group and aggregate data
  - Repeat
- Google has released the pipe syntax that *extends* SQL without replacing it entirely.
- Pipe syntax are already available on GoogleSQL Bigquery/F1/Spanner, Databricks/Spark, Firebolt and OSS: ZetaSQL.

DuckDB Prometheus Rate SQL Example

```
WITH buckets AS (
  SELECT
    metric, labels, TIME_BUCKET(INTERVAL '30 seconds', epoch_ms(timestamp * 1000)) AS bucket, value,
    FIRST(timestamp) OVER (PARTITION BY TIME_BUCKET(INTERVAL '30 seconds', epoch_ms(timestamp * 1000)) ORDER BY epoch_ms(timestamp *
1000)) AS first_ts,
    LAST(timestamp) OVER (PARTITION BY TIME_BUCKET(INTERVAL '30 seconds', epoch_ms(timestamp * 1000)) ORDER BY epoch_ms(timestamp *
1000)) AS last_ts,
    FIRST(value) OVER (PARTITION BY TIME_BUCKET(INTERVAL '30 seconds', epoch_ms(timestamp * 1000)) ORDER BY epoch_ms(timestamp *
1000)) AS first_val,
    LAST(value) OVER (PARTITION BY TIME_BUCKET(INTERVAL '30 seconds', epoch_ms(timestamp * 1000)) ORDER BY epoch_ms(timestamp *
1000)) AS last_val,
    ROW_NUMBER() OVER (PARTITION BY TIME_BUCKET(INTERVAL '30 seconds', epoch_ms(timestamp * 1000)) ORDER BY epoch_ms(timestamp *
1000)) AS rownum,
    COUNT(value) OVER (PARTITION BY TIME_BUCKET(INTERVAL '30 seconds', epoch_ms(timestamp * 1000)) ORDER BY epoch_ms(timestamp *
1000)) AS valCount,
  FROM 'cumulative_counters.json'
  WHERE labels.job = 'sample' AND timestamp BETWEEN
),
extrapolate_vals AS (
SELECT
    metric, labels, bucket, value, first_ts, last_ts, first_val, last_val, rownum,
    epoch_ms(first_ts * 1000) - bucket AS durationToStart,
    (bucket + INTERVAL '30 seconds') - epoch_ms(last_ts * 1000) AS durationToEnd,
    last_ts - first_ts AS sampledInterval,
    ((last_ts - first_ts) * 1.0) / (valCount - 1) AS averageDurationBetweenSamples,
    ((last_ts - first_ts) * 1.0) / (valCount - 1) * 1.1 AS extrapolationThreshold
  FROM buckets
),
extrapolate_durations AS (
…
```

DuckDB Prometheus Rate SQL Example

```
…
SELECT
    metric, labels, bucket, value,
    CASE WHEN EXTRACT('SECONDS' FROM durationToStart) >= extrapolationThreshold THEN averageDurationBetweenSamples / 2.0 ELSE
EXTRACT('SECONDS' FROM durationToStart) END AS startts,
    CASE WHEN EXTRACT('SECONDS' FROM durationToEnd) >= extrapolationThreshold THEN averageDurationBetweenSamples / 2.0 ELSE
EXTRACT('SECONDS' FROM durationToEnd) END AS endts,
    first_val, last_val, rownum, sampledInterval, averageDurationBetweenSamples
  FROM extrapolate_vals
),
extrapolate AS (
SELECT
  metric, labels, bucket, value,
  ((sampledInterval + startts + endts) / sampledInterval) / EXTRACT('SECONDS' FROM INTERVAL '30 seconds') AS extrapolateToInterval,
  first_val, last_val,  rownum, averageDurationBetweenSamples, sampledInterval, startts, endts
FROM extrapolate_durations
),
rate_calculation AS (
    SELECT
        metric, bucket, value,
        (last_val - first_val) * extrapolateToInterval AS rate,
        extrapolateToInterval, averageDurationBetweenSamples,  sampledInterval, startts, endts
    FROM
        extrapolate
    WHERE
        first_val IS NOT NULL
        AND rownum = 2
)
SELECT * FROM  rate_calculation
ORDER BY bucket;
```

Simplified SQL with Pipe Syntax Rate Example

```
# Fetch and window(rate) with non overlapping windows.

FROM telemetry_table
|> WHERE Timestamp
   BETWEEN TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 12 HOUR) AND CURRENT_TIMESTAMP()
# We truncate the timestamps to a minute, aggregate the values per timestamp
# and divide the value by 60 to get the per second rate.
|> AGGREGATE SUM(int64_delta) / 60 AS per_second_rate_over_1_minute
   GROUP BY TIMESTAMP_TRUNC(timestamp, MINUTE) AS timestamp, cloudprober_metro AS metro
|> ORDER BY timestamp DESC;
```

Example of Observability Query with Pipe Syntax

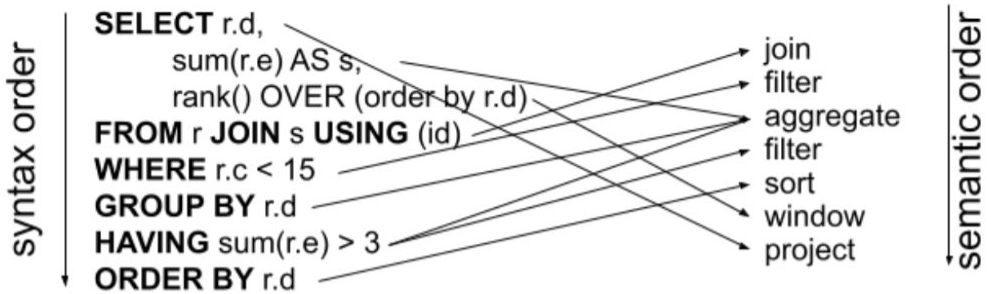
Standard SQL

```
SELECT *
FROM telemetry
WHERE timestamp >=
   TIMESTAMP_SUB(
     CURRENT_TIMESTAMP(), INTERVAL 1 hour)
AGGREGATE SUM(int64_delta)
GROUP BY
     timestamp,
     probe_name,
     origin_location,
     destination_location,
     status;
```

Pipe syntax

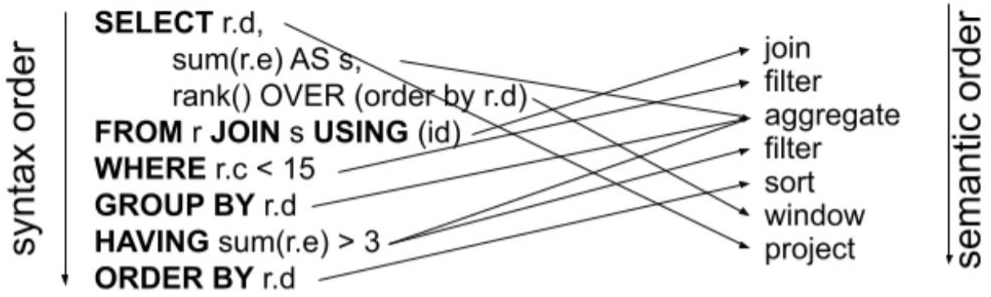
```
FROM telemetry
|> WHERE timestamp >=
   TIMESTAMP_SUB(
     CURRENT_TIMESTAMP(), INTERVAL 1 hour)
|> AGGREGATE SUM(int64_delta)
   GROUP BY
     timestamp,
     probe_name,
     origin_location,
     destination_location,
     status;
```

The insanity of standard SQL...

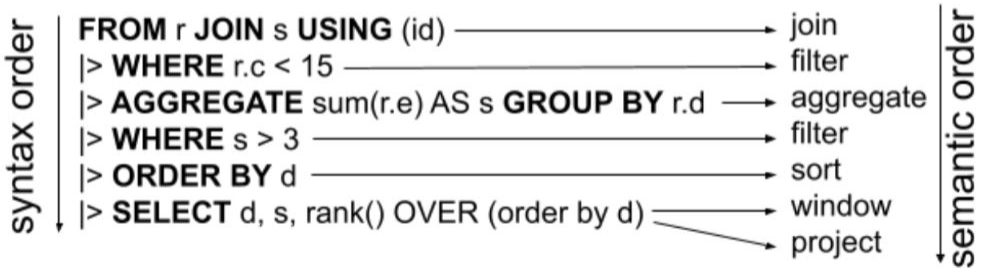


From: A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language (Thomas Neumann and Viktor Leis, CIDR 2024)

The insanity of standard SQL...

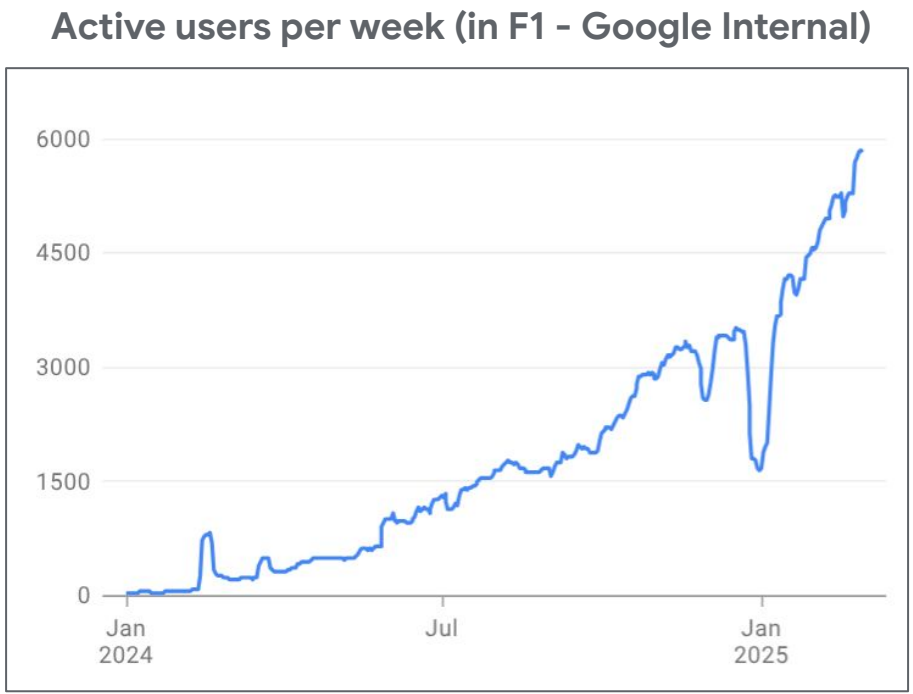


... versus the sanity of pipe syntax



Usage at Google - First year

- Users see it, learn it quickly, want to use it.
- It's sticky, and spreads virally



Pipe syntax: status and next

- **Try it in BigQuery** (docs)
  - Open to all as of February!
- Try it in DataBricks / Spark (docs) and in Firebolt (docs).
  - Release of Firebolt was this week!

- Read the paper: SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL (VLDB 2024)
- See OSS Zetasql
  - Query parser, analyzer, runnable reference implementation, etc.
- **For the community:** Support SQL pipe syntax in more systems?
  - Hint: Jacek Migdal's talk: https://kccnceu2025.sched.com/speaker/jacek21 (16:45 today - Room G)

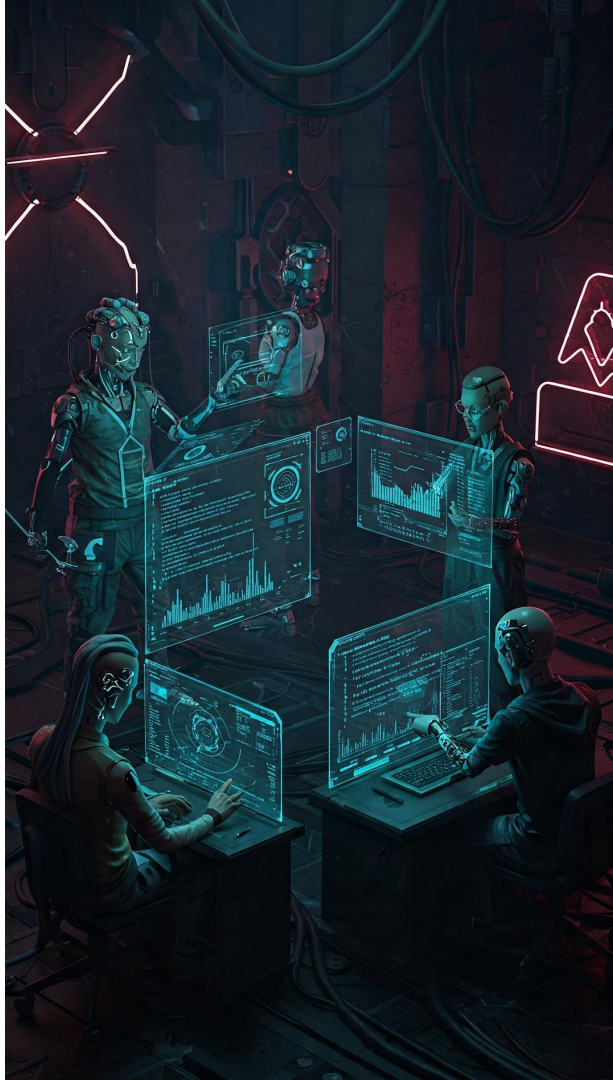
Collaboration: Google and the CNCF

Conclusion:

- From different perspectives, we reached same conclusion:
  - Aligned on SQL semantics for querying observability

Next steps:

- Joint effort to share ongoing research on operators for observability:
  - Google sharing specs (target: second half 2025 in OSS):
    - Align/Window
    - Histogram
    - Some syntactic sugar
- Work together on alignment for a unified syntax for observability querying.



QR Codes

CNCF DRAFT
Semantic Specification

ZetaSQL

Pipe Syntax in SQL
Paper

Questions?

Rate/Feedback