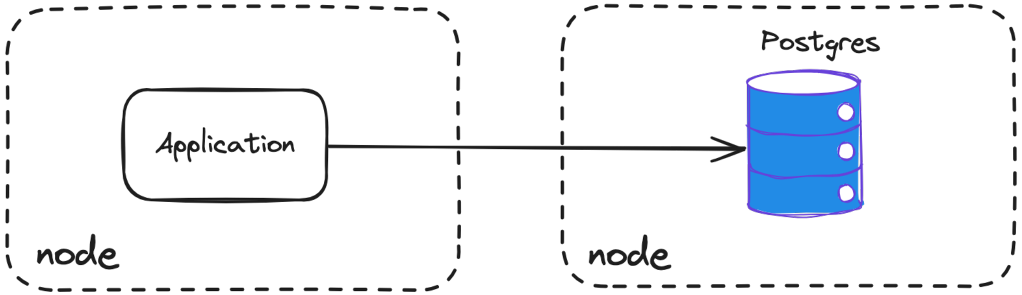
Cloud-native Postgres observability

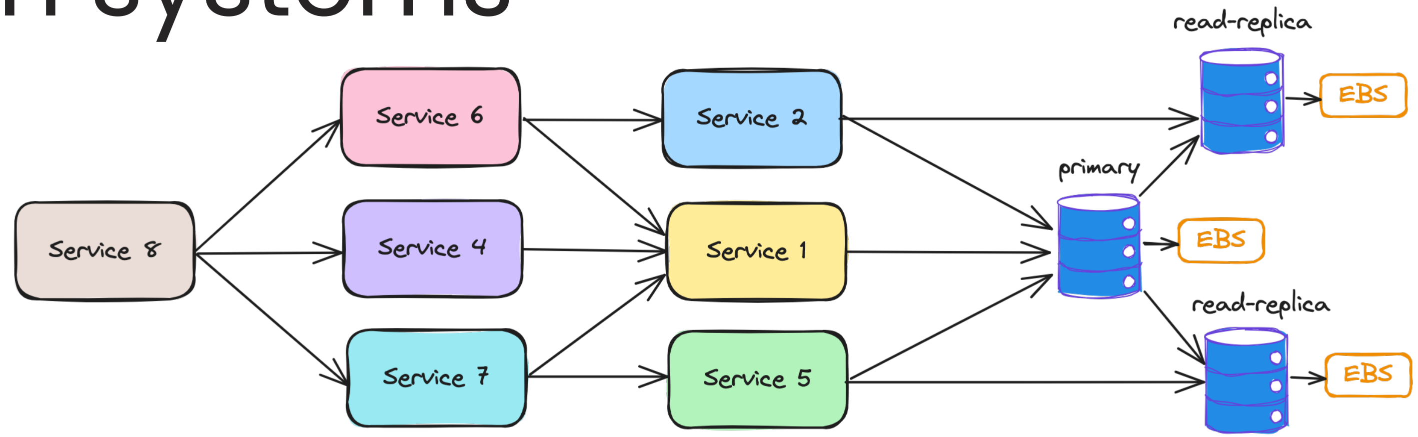
Nikolay Sivko,
Co-Founder & CEO at Coroot

A while ago, systems using Postgres



- A monolith application
- DB runs on dedicated nodes
- If something goes wrong:
  - Check the app's logs/metrics
  - Check the DB's logs/metrics
  - Check the hardware

Modern systems



- Hundreds or even thousands of services dynamically allocated to nodes
- Nodes are dynamic and can appear and disappear
- Network-attached volumes
- If something goes wrong:
  - Troubleshooting follows the system's topology
  - Analysis of extensive telemetry, from application latency to EBS performance

Observability is …

… being able answer questions about your system:

- How is the system performing right now?
- How does its performance compare to an hour ago?
- Why are some requests failing?
- Why are certain requests taking longer than expected?

- **Observability is most valuable during system failures or issues, so we should think of it by considering failure scenarios**

What can possibly go wrong here?

- The app is not available
- The DB is not available
- Network connectivity issues between the app and the DB
- Network delay between the app and the DB
- The DB responses slowly
- The DB rejects connections from the app
- …

When we only look at the DB, we don't see the big picture

- Error counters are not available in **pg_stat_***
- Per-client query statistics are not provided in **pg_stat_***
- Query latency in **pg_stat_statements** doesn’t include network latency

“The customer is always right”

- Let’s consider databases as utility services
- The Service Level metrics (availability, latency) should be measured on the client’s side
- The database-side metrics are needed to “explain” the DB’s behavior, e.g., why the DB is rejecting connections or performing slowly.

Client-side query statistics

What we want to know:

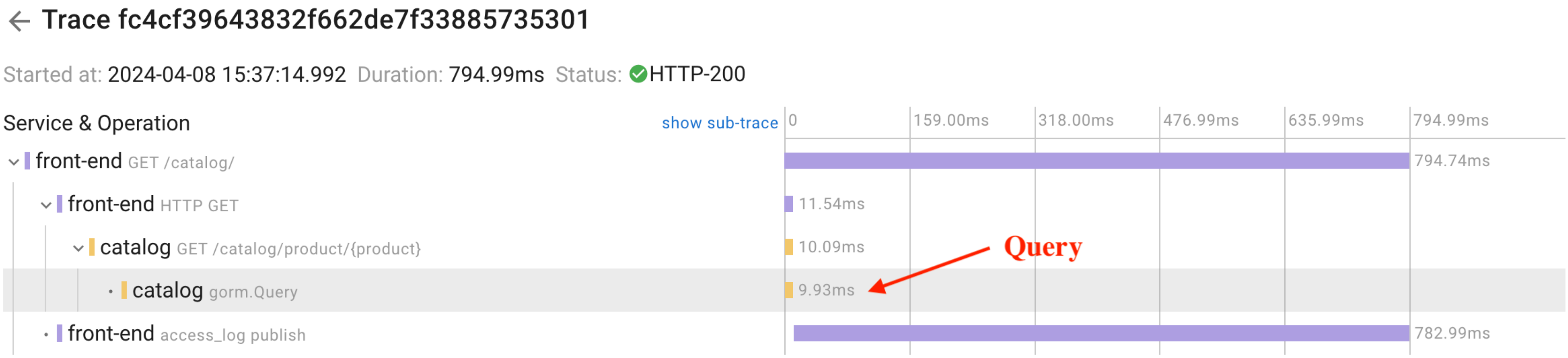
- The number of queries from a given app instance to a particular DB instance
- Errors (TCP level, L7-protocol level)
- Latency

How we can gather these statistics:

- Instrumenting apps with OpenTelemetry SDKs
- Using eBPF to capture queries made by every process/container to measure the statistics

Instrumenting apps with OpenTelemetry

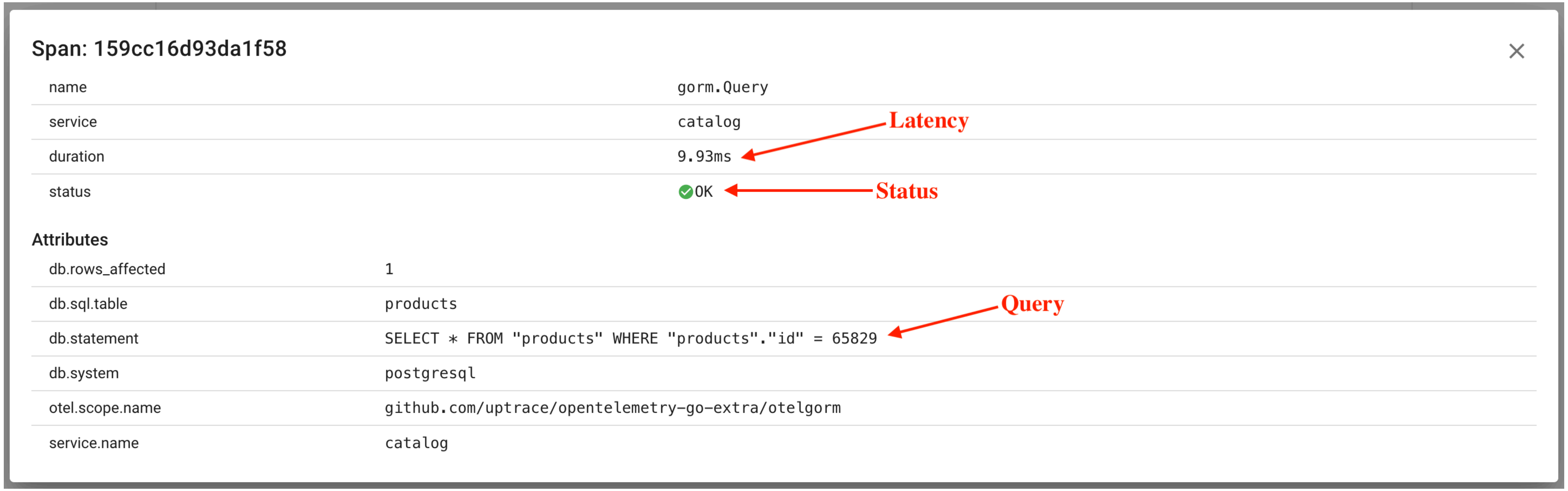
- **OpenTelemetry** is a vendor-neutral framework for instrumentation applications, including database calls
- SDKs are available for many programming languages and frameworks
- It wraps every database call to gather statistics



coroot :~#

Instrumenting apps with OpenTelemetry

With distributed tracing we can know exactly what’s happened with any given request



Span: 159cc16d93da1f58

name	gorm.Query
service	catalog
duration	9.93ms
status	OK

Attributes

db.rows_affected	1
db.sql.table	products
db.statement	SELECT * FROM "products" WHERE "products"."id" = 65829
db.system	postgresql
otel.scope.name	github.com/uptrace/opentelemetry-go-extra/otelgorm
service.name	catalog

Challenges associated with Distributed Tracing

- Huge volume of telemetry data
- Hard to achieve 100% coverage without blind spots (e.g., legacy services)
- Requires code changes and application deployments
- Potential overhead

eBPF-based instrumentation

- An agent captures network calls from each process running on the node
- It parses L7 protocols including Postgres Wire Protocol
- Doesn't require code changes, so can instrument even legacy and 3rd-party services
- Can be integrated in minutes
- Can capture stats even within SSL-enabled connections
- Query latency contains network latency since it's measured on client's side
- Doesn't affect application latency*

Open-source implementations supporting Postgres: **Coroot**, **Pixie, eCapture**

* https://coroot.com/docs/coroot-community-edition/getting-started/performance-impact

eBPF-based instrumentation

Capturing System Calls:

- connect(): obtaining PID, FD, destination IP:PORT, and status.
- write(), writev(), sendmsg(), sendto(), SSL_write(), read(), readv(), recvmsg(), recvfrom(), SSL_read(): monitoring for Postgres protocol frames

eBPF-based instrumentation

Postgres Protocol Parsing:

- Identifying Postgres protocol frames requires parsing only ~10 bytes of payload in kernel-space
- Payloads (up to 1kB) are transferred to user-space along with (PID, FD, timestamp, payload)
- Client app container is resolved using PID
- Connection destination IP:PORT is resolved using PID + FD
- If tracing is enabled, payload parsing extracts query text
- For prepared statements, the agent maintains a mapping of statement_id to statement_text

eBPF: performance impact

The Linux kernel ensures minimal interruption to kernel code execution by validating each eBPF program before execution:

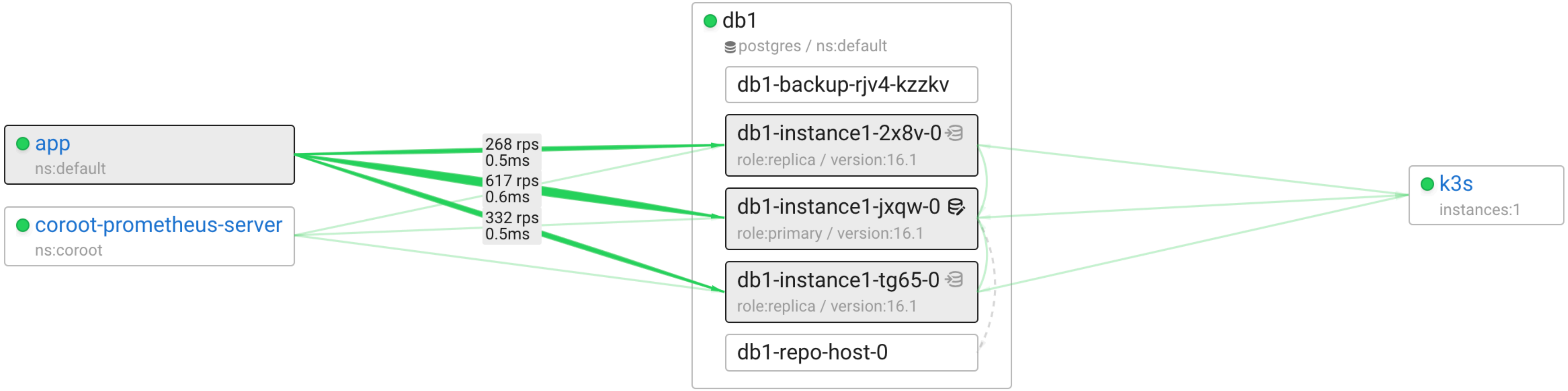
- Program **must** have a finite complexity.
- The verifier evaluates all possible execution paths within configured upper complexity limits

Communication between kernel-space and user-space programs occurs through a ring buffer:

- If the user-space program delays data reading, it may miss data due to overwriting

For observability, it's a great deal: although we might lose some telemetry data, we can be sure that there is no impact on performance

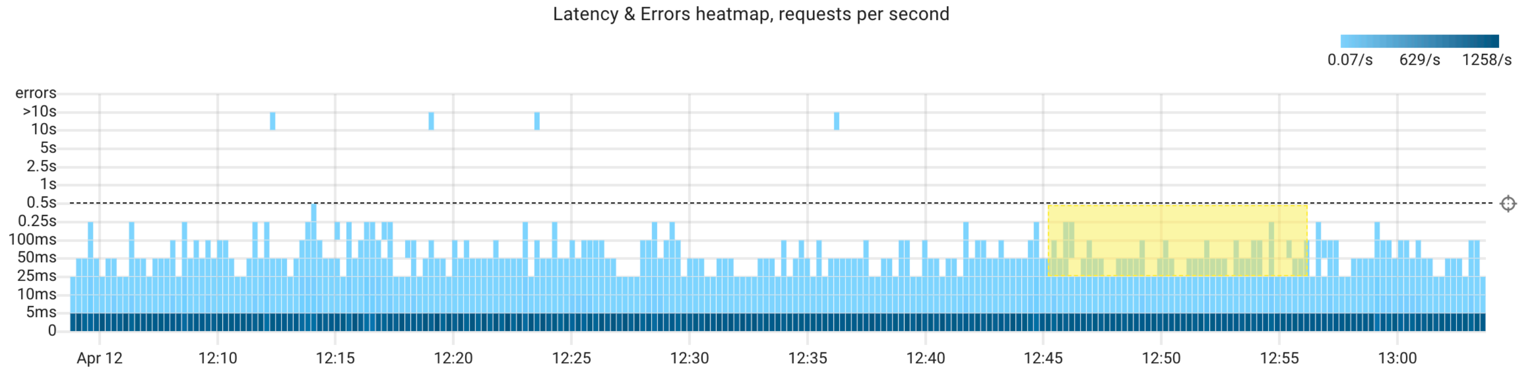
eBPF-based metrics



We know how each application instance communicates with each DB instance:

- Queries per second
- Errors
- Latency

eBPF-based traces



	Client	Status	Duration	Name	Details
	app	OK	27.04ms	query	`PREPARE AS select title, body from articles where id = any ($1)`
	app	OK	53.30ms	query	`insert into articles (created, title, body) values ($1, $2, $3)`
	app	OK	38.06ms	query	`select id from articles where created < $1 order by created desc limit $2`
	app	OK	26.22ms	query	`PREPARE AS select id from articles where created < $1 order by created desc limit $2`

- Traces are extremely useful for identifying the particular queries within an anomaly
- They also provide a more granular distribution of queries by latency and status

coroot :~#

Postgres metrics

eBPF-based metrics and traces can't answer all questions:

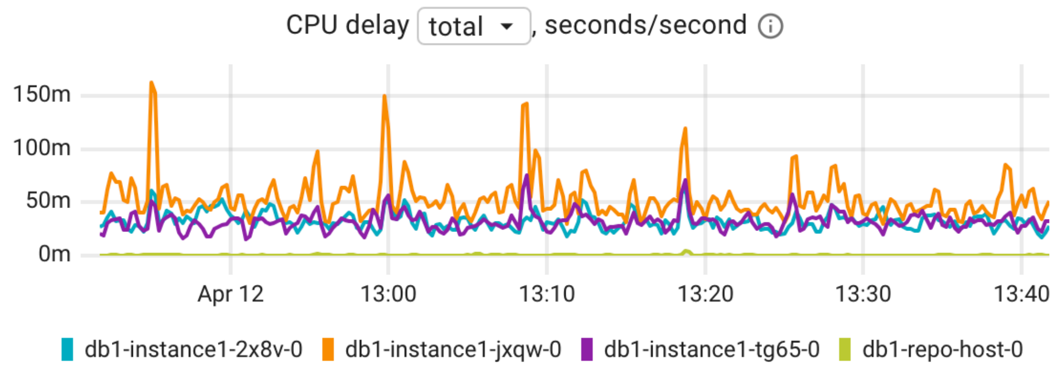
- Why is the database performing slower than before?
- Why is the database rejecting new client connections?

While eBPF-based metrics can highlight what is happening, to answer WHY it's happening, we need to collect other metrics.

Why are my queries executed slower that usual?

- A lack of CPU time
  - Node CPU capacity
  - Resource limits leading to CPU throttling
  - Resource contention caused by other applications
  - Resource contention caused by other queries
- Issues related to I/O performance
  - Volume I/O capacity (Block storage I/O limits, hardware performance)
  - High I/O latency, particularly with network-attached volumes
  - Resource contention due to other applications
  - Resource contention due to other queries
  - Using temp files due to insufficient work_mem
- Locks

CPU related metrics: CPU delay

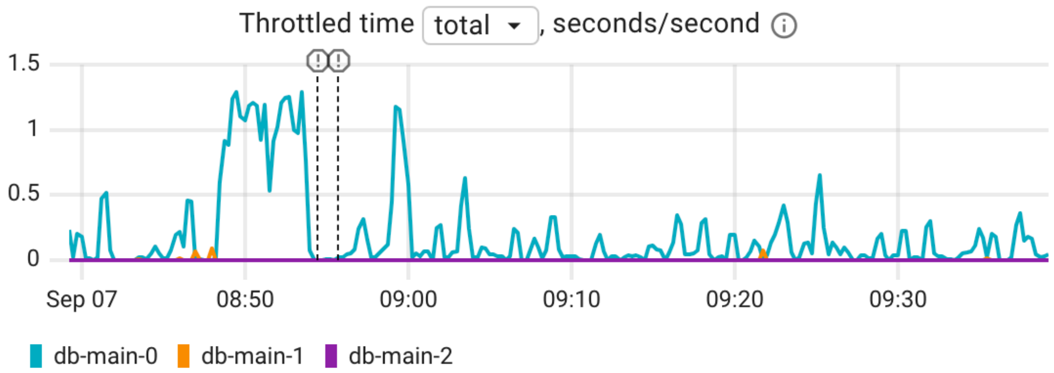


- The Linux kernel reports CPU delay, indicating how long a specific process or container has been waiting for CPU time

- For instance, if you observe a delay of 150ms per second, it signifies that you are experiencing an additional latency of 150ms, which is spread across all queries processed during that wall-clock second

- Next steps: check CPU throttling, node CPU usage, other CPU consumers

CPU related metrics: CPU throttling

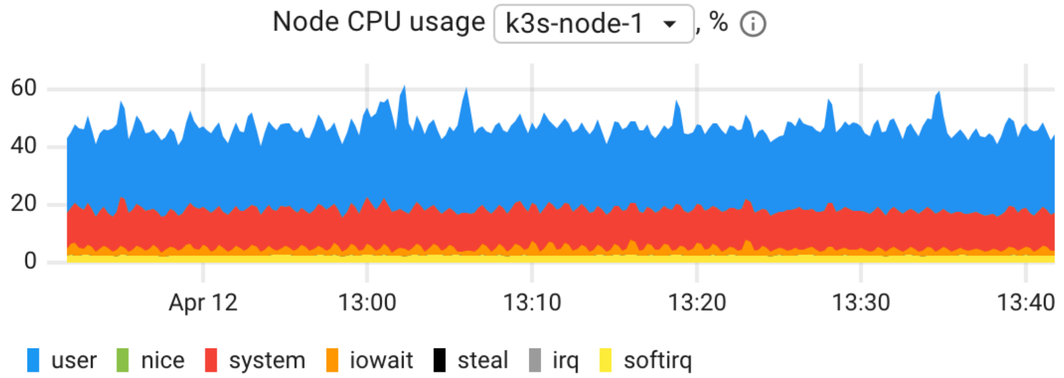


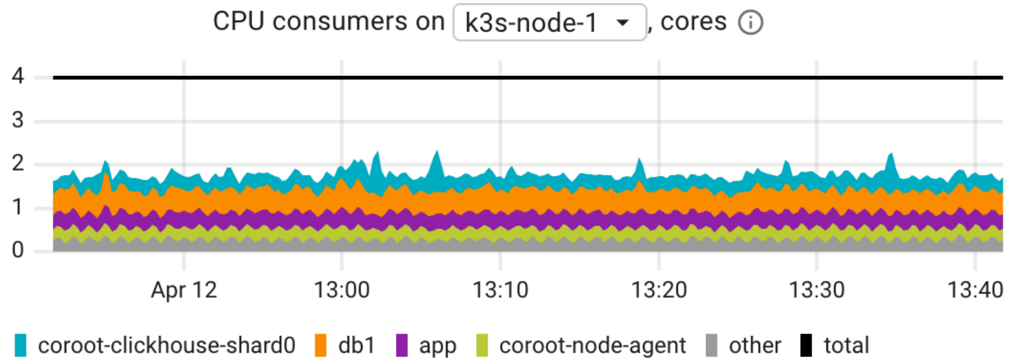
- When a container hits its CPU limit and exhausts the allowed CPU bandwidth, it gets throttled for the remainder of that period.

- This introduces additional latency spread across all queries processed during that wall-clock second.

- If a container is CPU-limited (throttled), the CPU delay metric will also increase

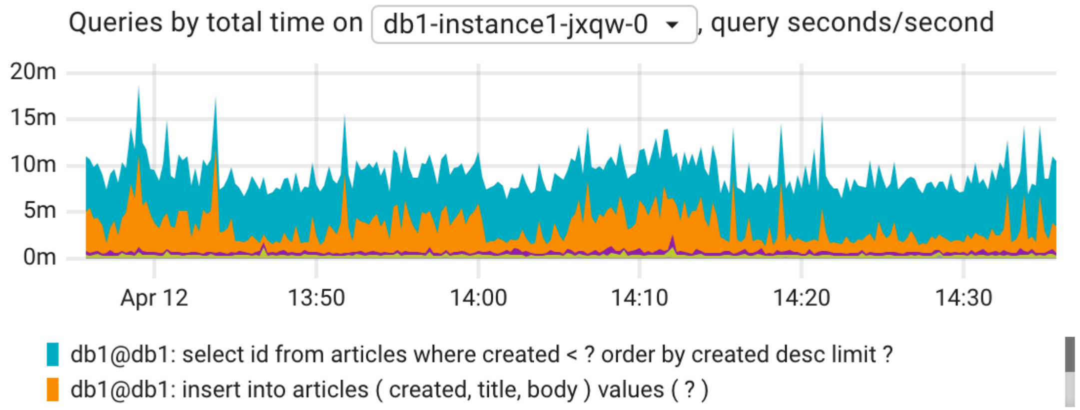
CPU related metrics: CPU usage





- Node CPU capacity always is limited
- Processes on the same node compete for CPU time
- In dynamic environments like Kubernetes, it's useful to track CPU usage per application running on a node to explain any CPU usage anomalies

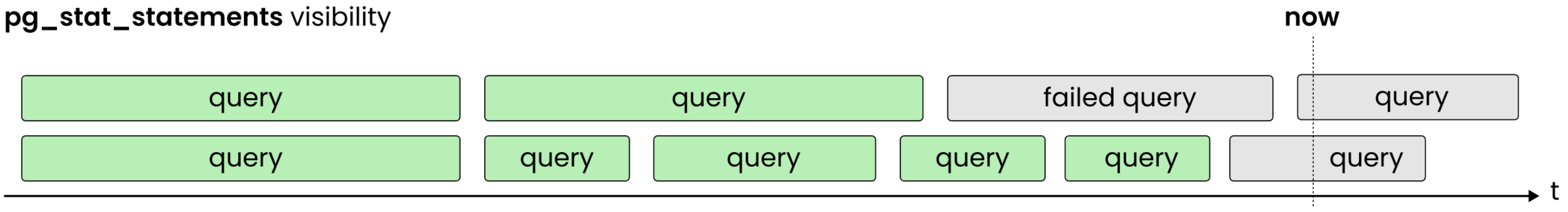
CPU related metrics: CPU usage by queries



- Postgres doesn’t count CPU usage by queries
- To **roughly** estimate that we can use total query execution time
- pg_stat_statements provides statistics only for finished queries
- To get visibility into long-running queries that are not finished yet, we need to merge statistics from pg_stat_statements and pg_stat_activity

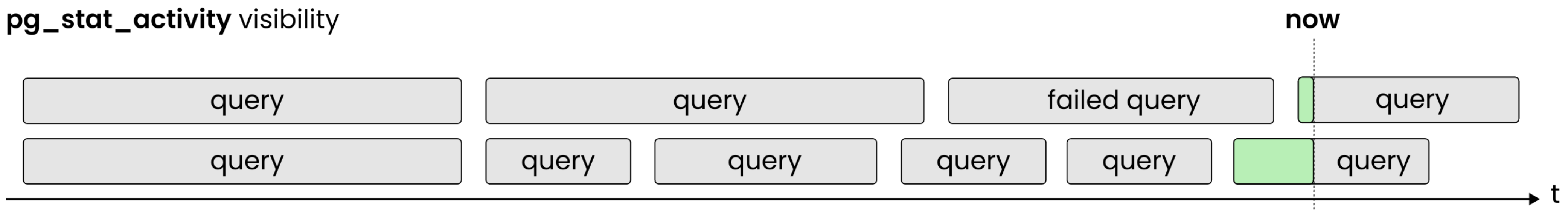
coroot :~#

pg_stat_statement visibility



- Only shows finished queries
- Queries that finish with errors/timeout are not taken into account

pg_stat_activity visibility



- Doesn’t track history
- Hard to track short-lived queries

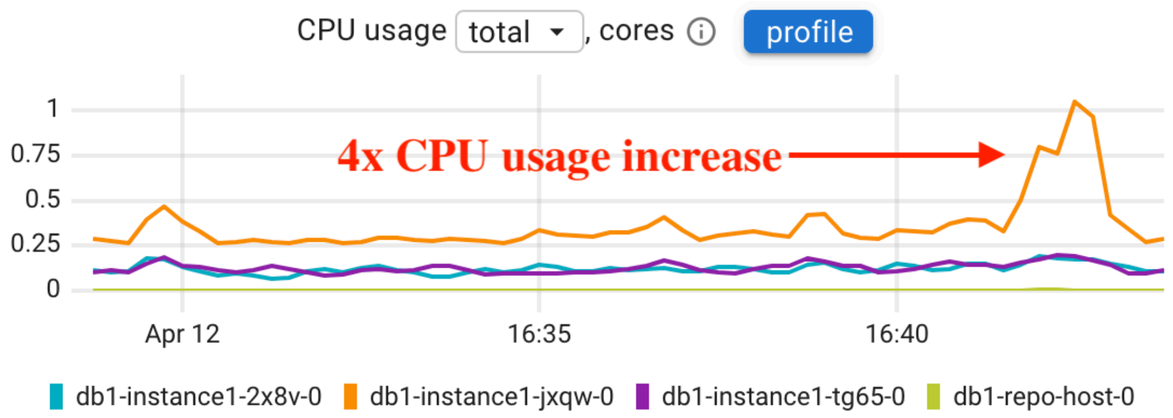
pg_stat_statements + pg_stat_activity

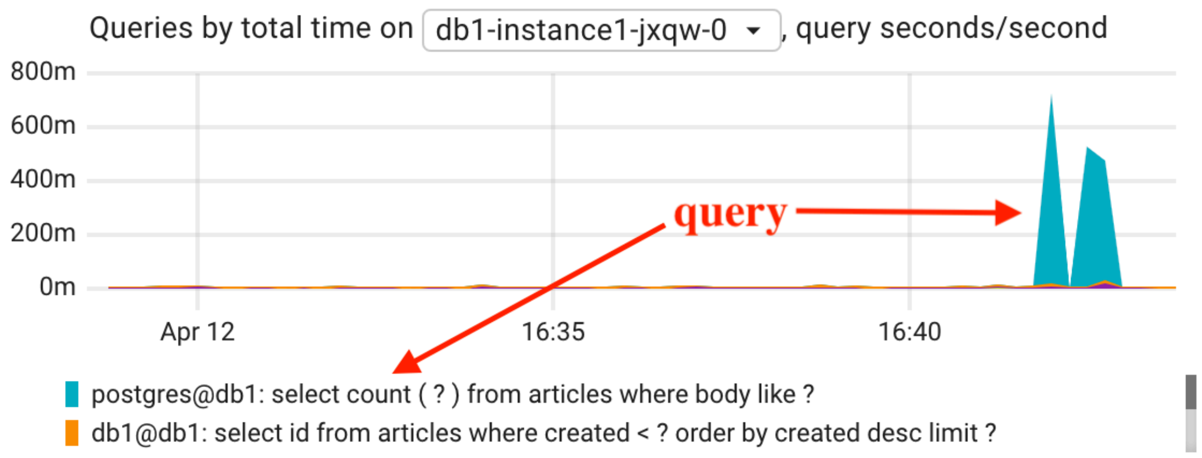
- To achieve full query visibility, we implemented an open-source (Apache 2.0) Prometheus metric exporter for Postgres
- It aggregates data from pg_stat_statements and pg_stat_activity to provide accurate metrics about queries, whether they are completed or still running
- Fully integrated with Coroot (Apache 2.0)
- Since Coroot v1.4 pg-agent is embedded into Coroot, so you don't need to deploy it manually

https://github.com/coroot/coroot-pg-agent

https://github.com/coroot/coroot

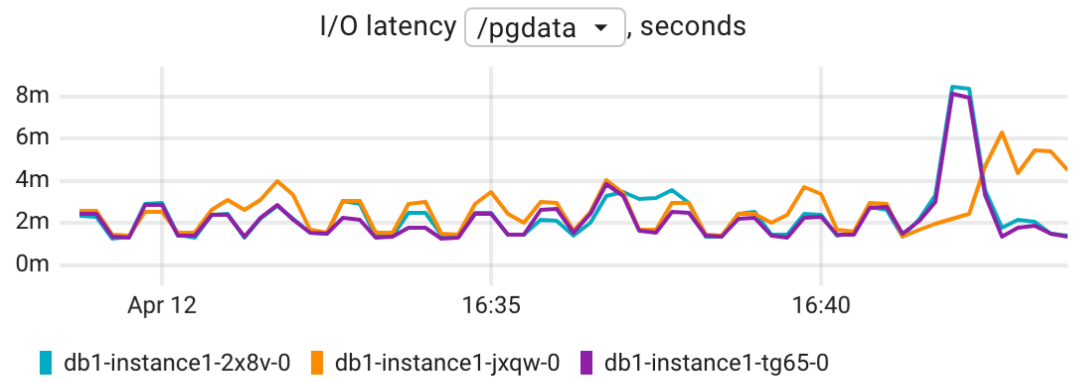
Explaining a CPU anomaly





coroot :~#

I/O related metrics: I/O latency

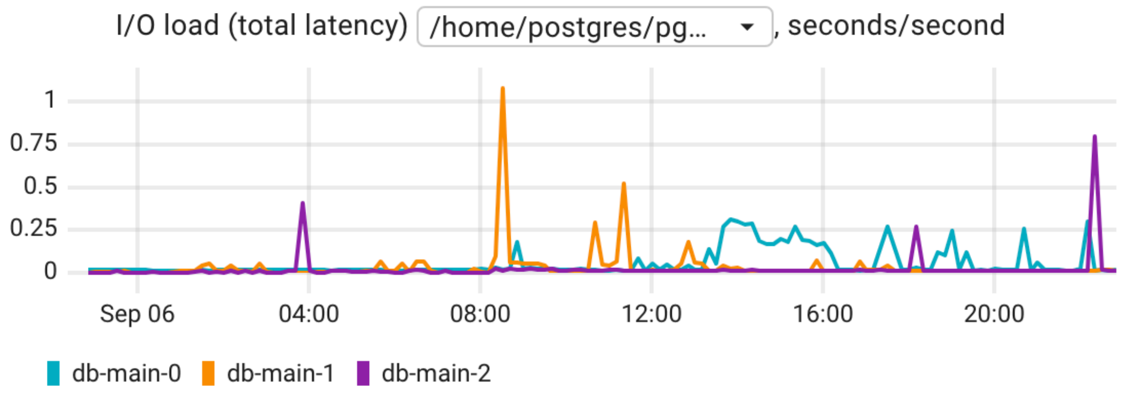


- An average time spent doing read and write operations

Type	Avg latency
Amazon EBS gp2/gp3/io1/io2	"single-digit millisecond"
Amazon EBS io2 Block Express	"sub-millisecond"
HDD	2-4ms
NVMe SSD	0.1-0.3ms

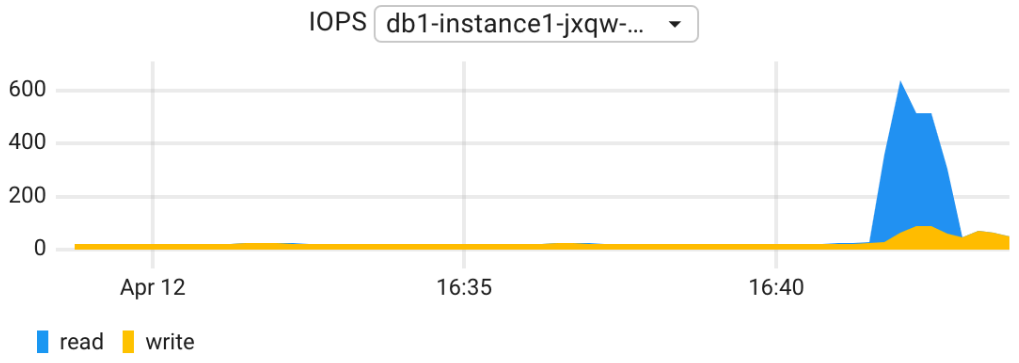
coroot :~#

I/O related metrics: I/O load



- Total number of seconds the disk spent doing I/O
- Modern SSD/NVMe disks can handle multiple I/O requests simultaneously, so I/O load can be >1 second per second
- Coroot uses a default threshold of 5 seconds/second to highlight high I/O load

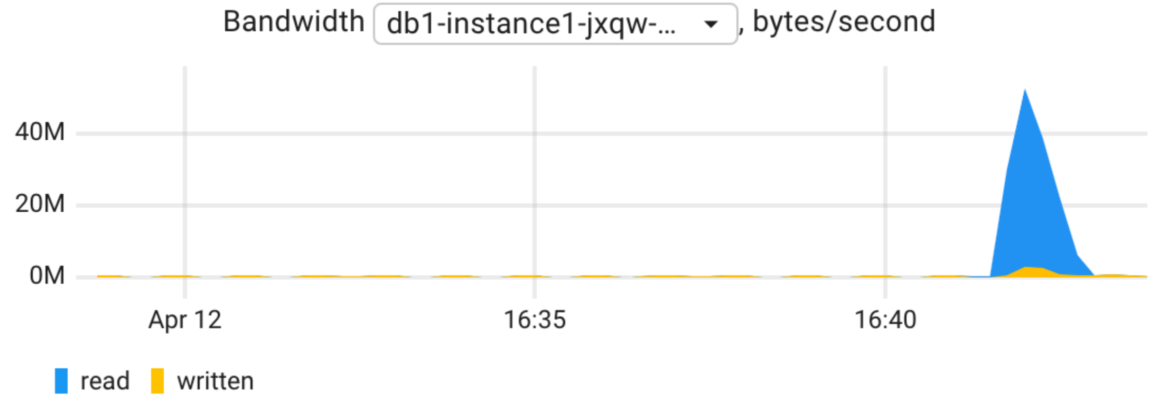
I/O related metrics: IOPS



- Total number of reads or writes completed successfully.

Type	Max IOPS
Amazon EBS sc1	250
Amazon EBS st1	500
Amazon EBS gp2/gp3	16,000
Amazon EBS io1/io2	64,000
Amazon EBS io2 Block Express	256,000
HDD	200
SATA SSD	100,000
NVMe SSD	10,000,000

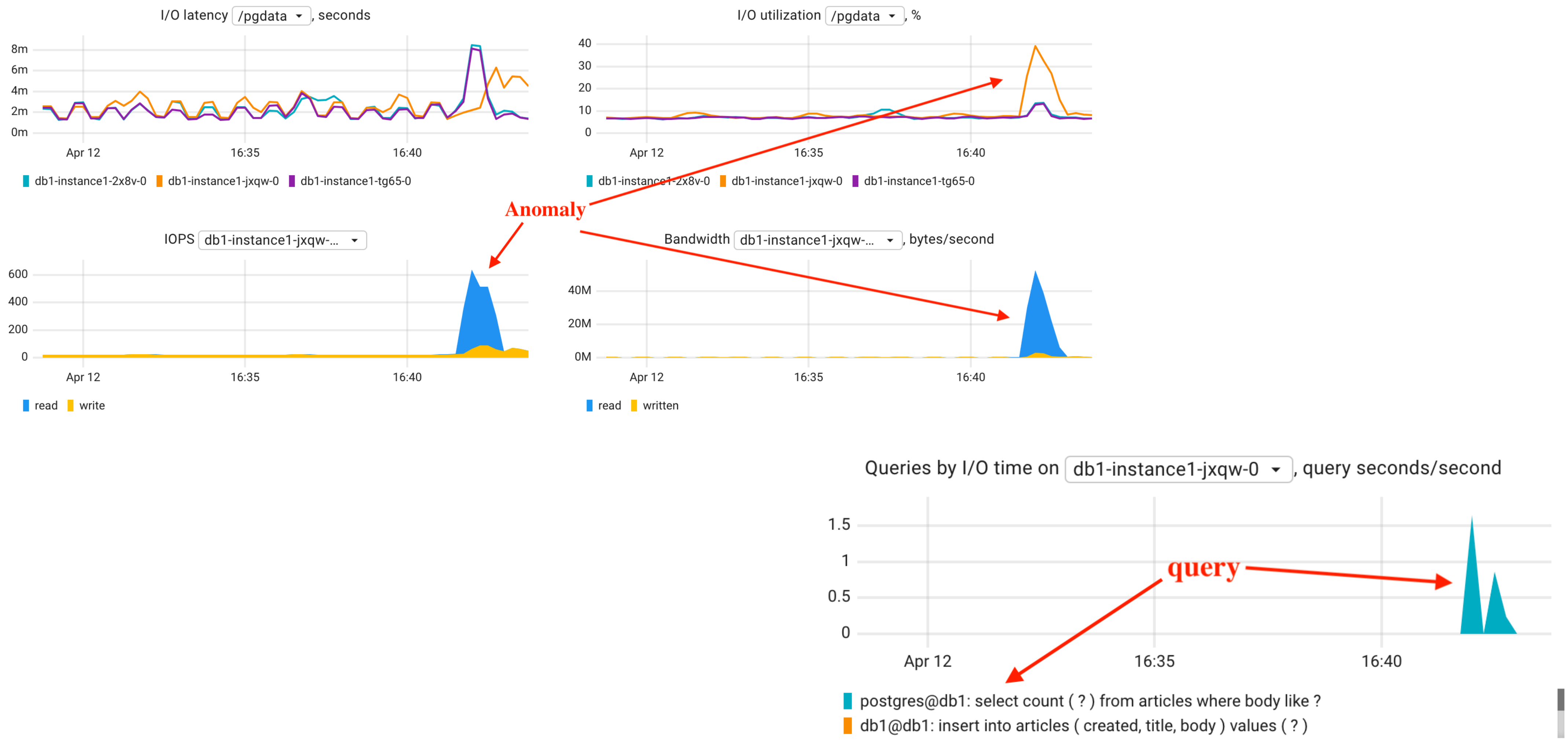
I/O related metrics: I/O bandwidth



- Total number of bytes read from the disk or written to the disk

Type	Max throughput
Amazon EBS sc1	250 MB/s
Amazon EBS st1	500 MB/s
Amazon EBS gp2	250 MB/s
Amazon EBS gp3	1,000 MB/s
Amazon EBS io1/io2	1,000 MB/s
Amazon EBS io2 Block Express	4,000 MB/s
SATA	600 MB/s
SAS	1,200 MB/s
NVMe	4,000 MB/s

Explaining an I/O anomaly



Try Coroot

- **Coroot is open source**: https://github.com/coroot/coroot
- **Quick integration:** thanks to eBPF, it can be integrated in minutes
- **Run everywhere**: Kubernetes, containers, VM, bare-metal, AWS RDS
- **Advanced database monitoring:** supports popular open-source databases like Postgres, MySQL, MongoDB, Redis, and Memcached
- **All-in-one observability platform**: metrics, logs, traces, profiles
- **Community Edition:** licensed under Apache 2.0
- **Enterprise Edition:** SSO, RBAC, and AI-driven root cause analysis

Thank you, Let’s connect!

https://www.linkedin.com/in/nikolay-sivko

https://twitter.com/NikolaySivko

https://github.com/coroot/coroot

coroot :~#