Using Serverless Functions for Real-time Observability

SRECon, 16 March 2022

Liz Fong-Jones @lizthegrey
Jessica Kerr @jessitron

honeycomb.io

Liz Fong-Jones

Principal Developer Advocate

@lizthegrey

Jessica Kerr

Principal Developer Advocate

@jessitron

Today

How serverless is useful for on-demand compute

How serverless is painful for on-demand compute

How to experiment with serverless in your environment

What is Lambda for?

Let's talk use cases of serverless

What is ~~Lambda~~ for?

We'd like to optimize our custom datastore, Retriever

@jessitron

What is Retriever for?

It’s a distributed column store for **real-time event aggregation**

@jessitron

What is ~~Retriever~~ for?

Real-time event aggregation for interactive querying over traces

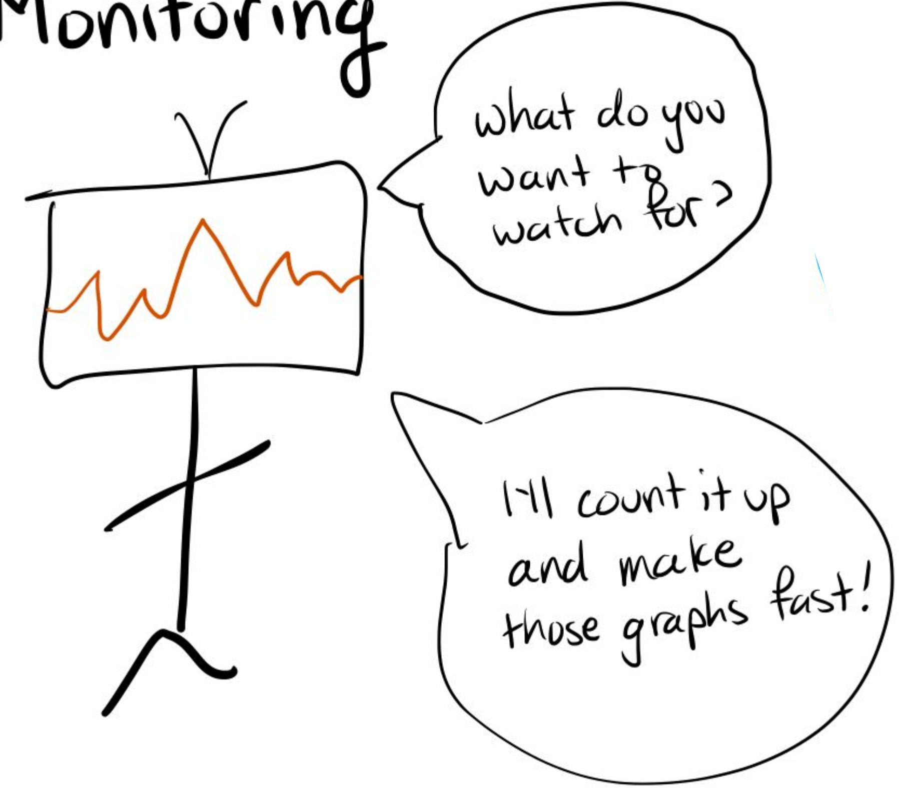
@jessitron

What is Honeycomb for?

Observability: finding out what is going on (by querying traces!)

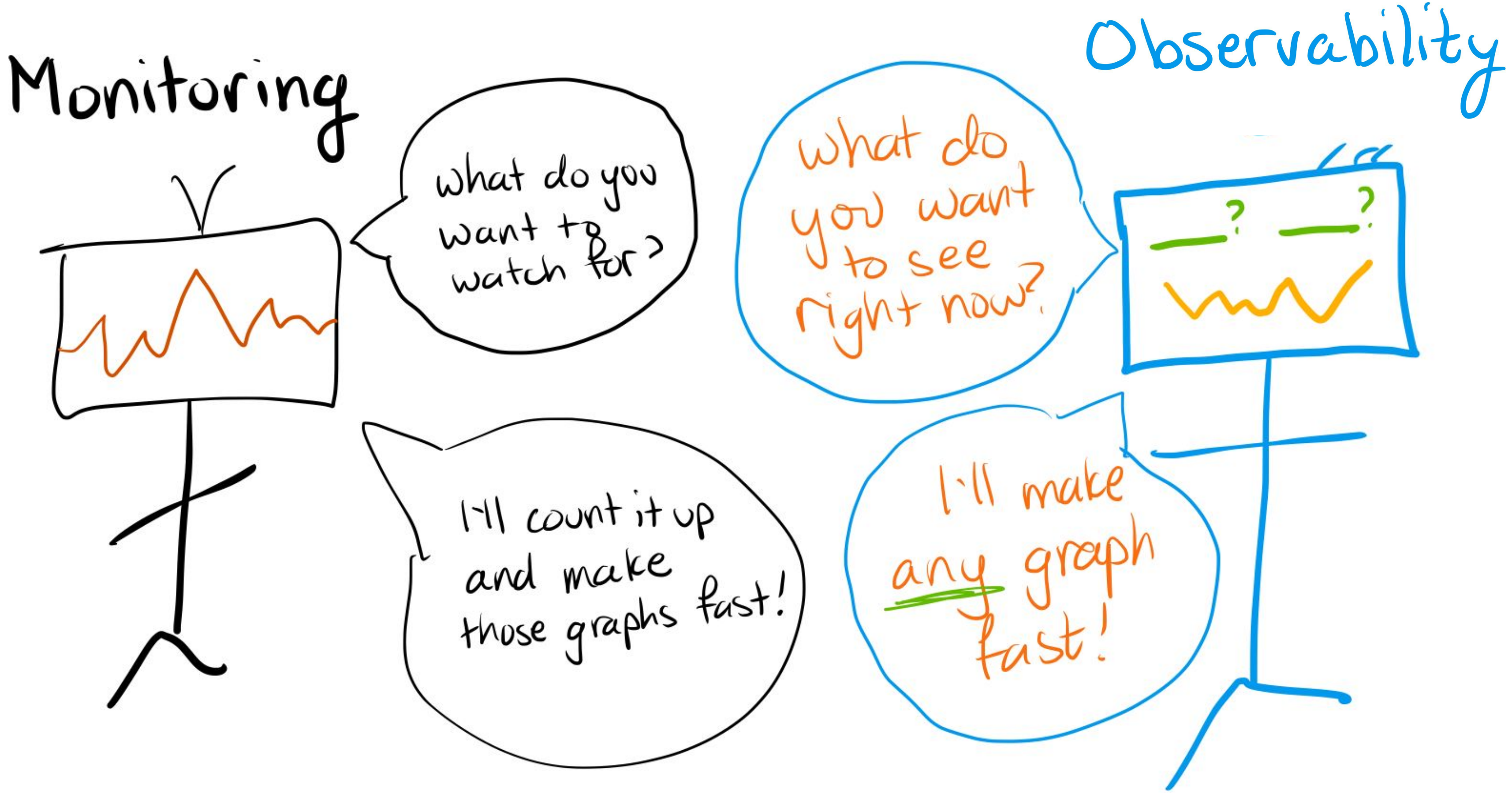
@jessitron

Monitoring


what do you want to watch for?
I'll count it up and make those graphs fast!

Monitoring

Observability


what do you want to watch for?
I'll count it up and make those graphs fast!
what do you want to see right now?
I'll make any graph fast!

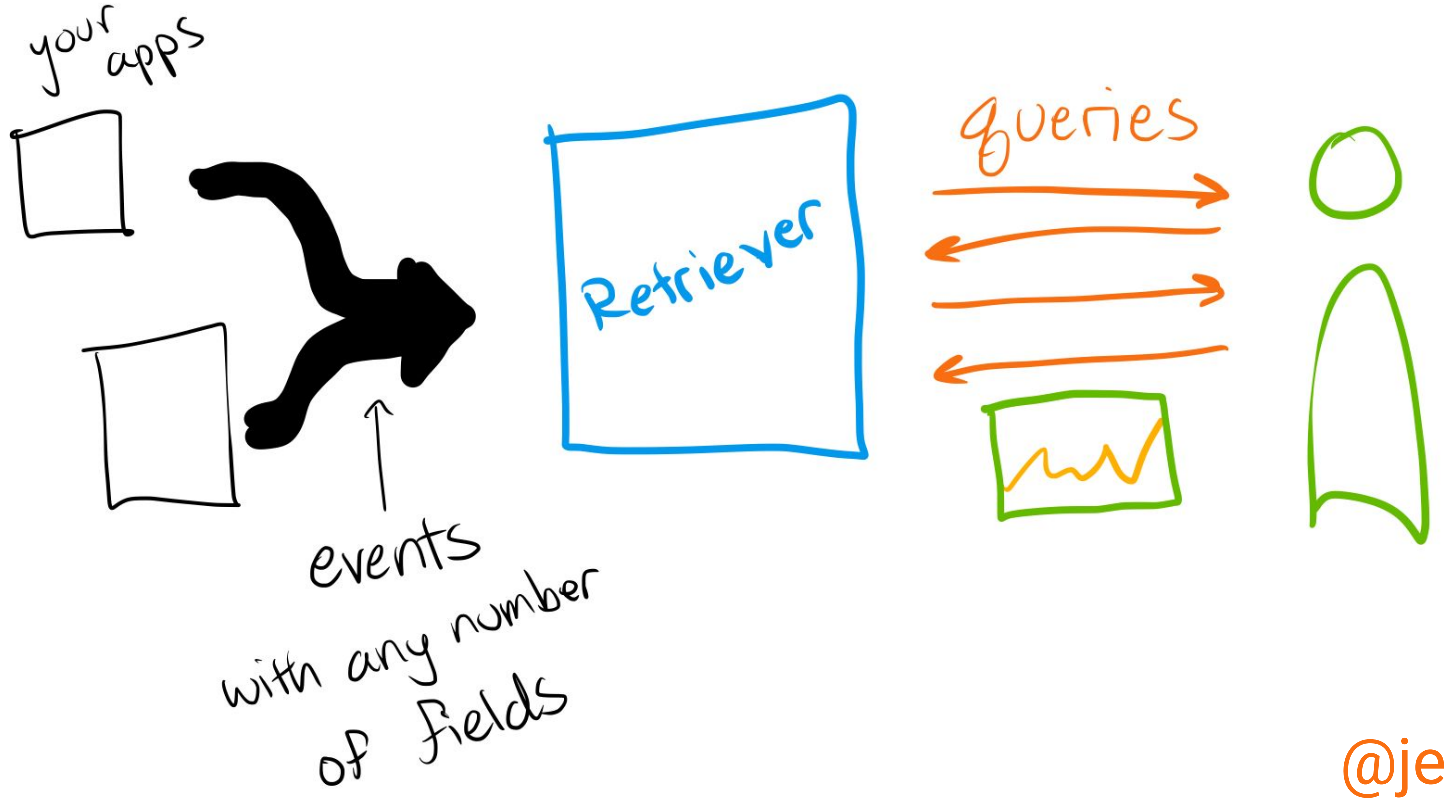
Interactive investigation of production behavior

We run fast queries across any combination of fields.

Emphasis: *interactive.*

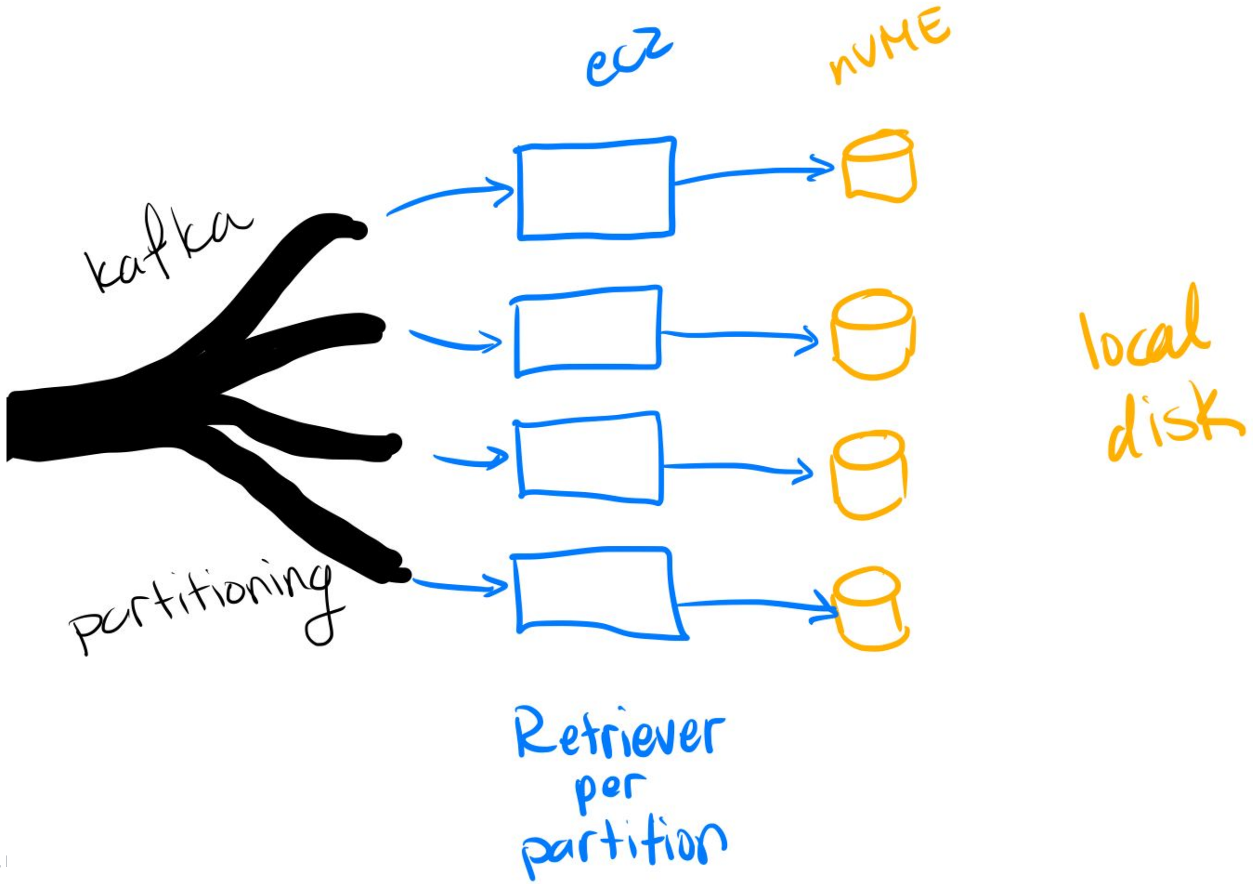
100ms is fast. 1000ms is ok. 10sec is slow. 100sec is unacceptable.

Retriever stores all your event data

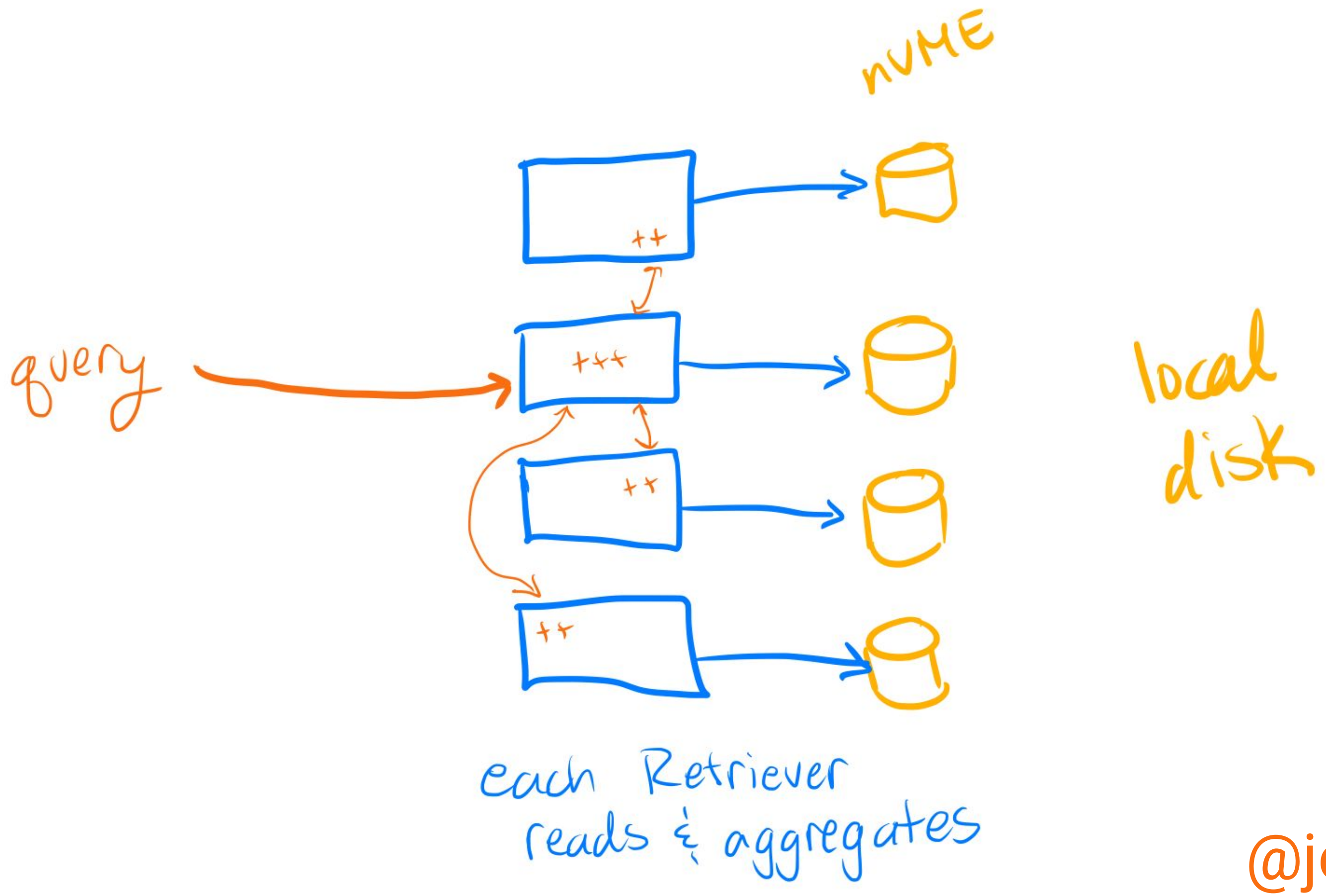


@jessitron

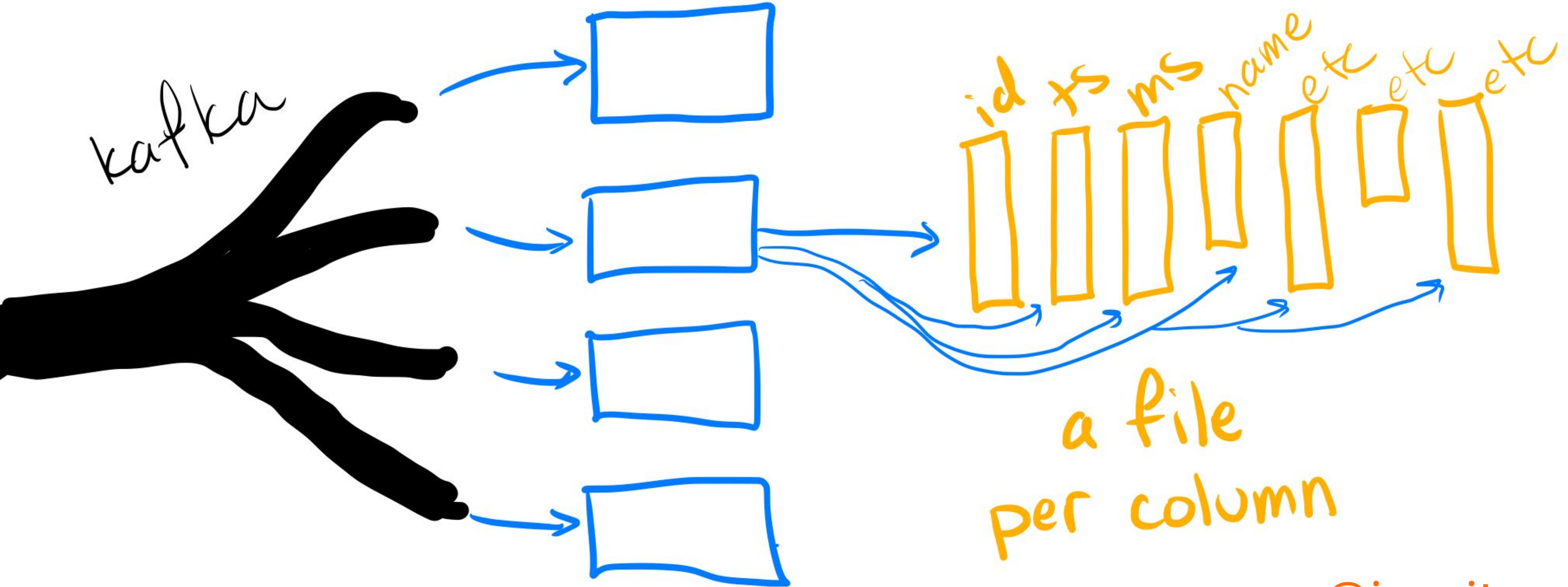
Retriever is a distributed datastore.



Retriever is a distributed datastore.

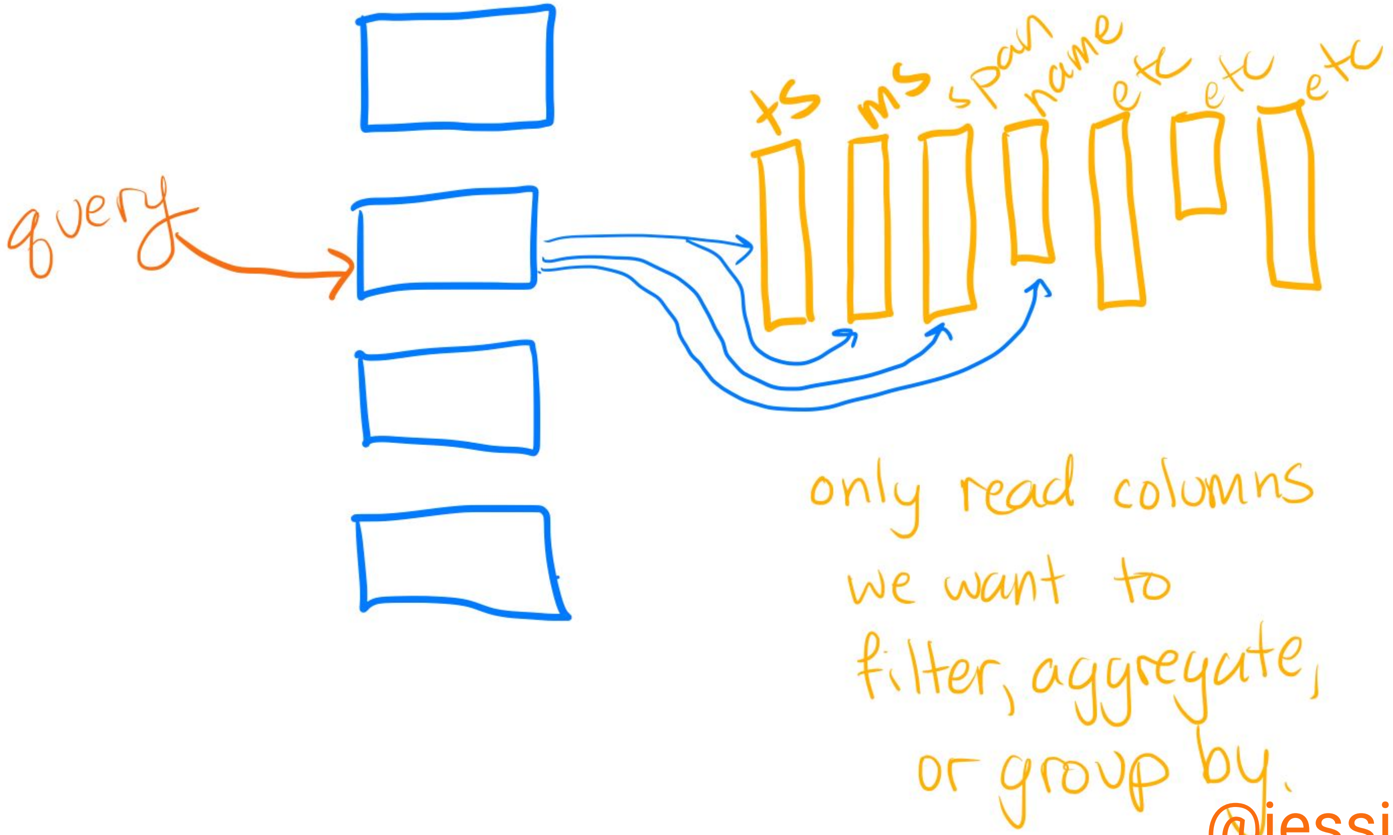


Retriever is a distributed column store.



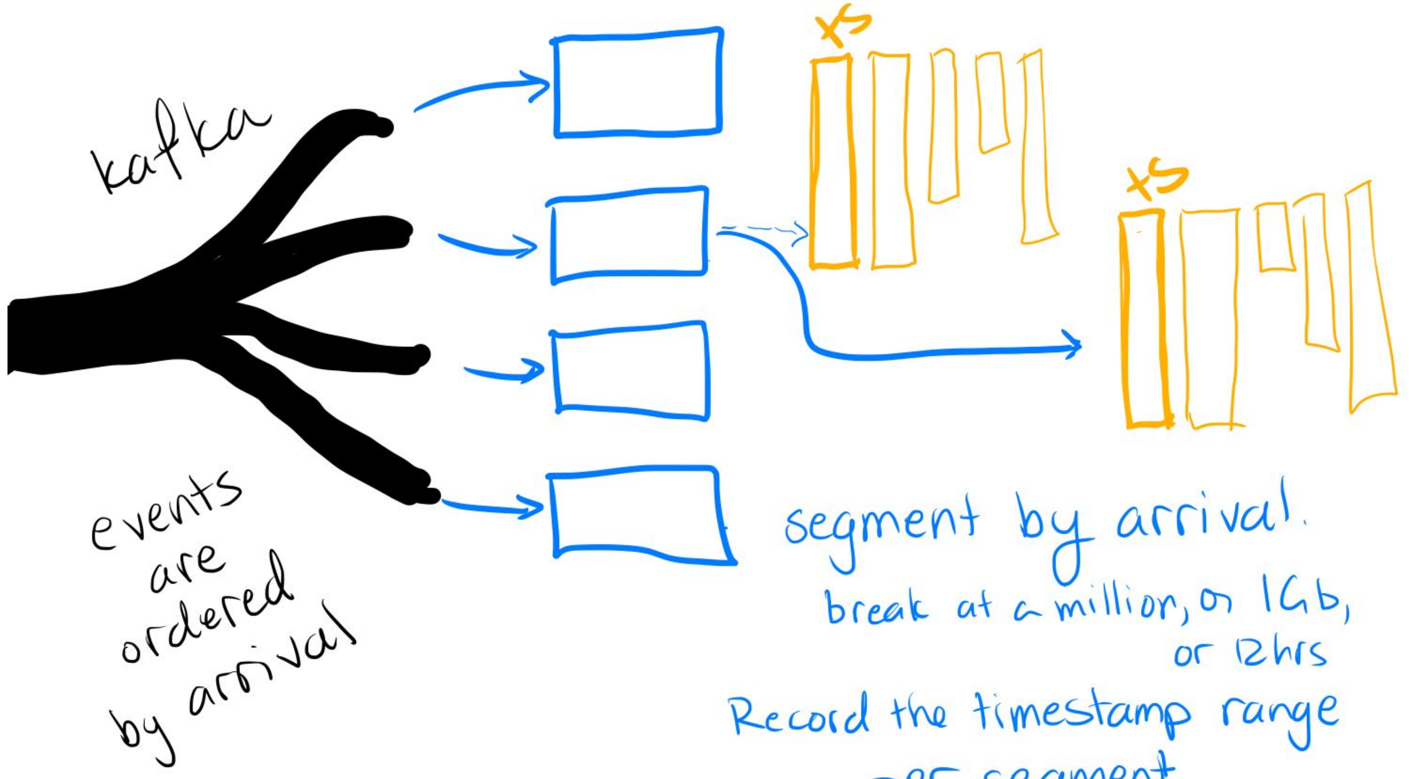
@jessitron

Retriever is a distributed column store.


query
ts
ms
span name
etc
etc
etc
only read columns we want to filter, aggregate, or group by.

@jessitron

Retriever indexes segments by timestamp.



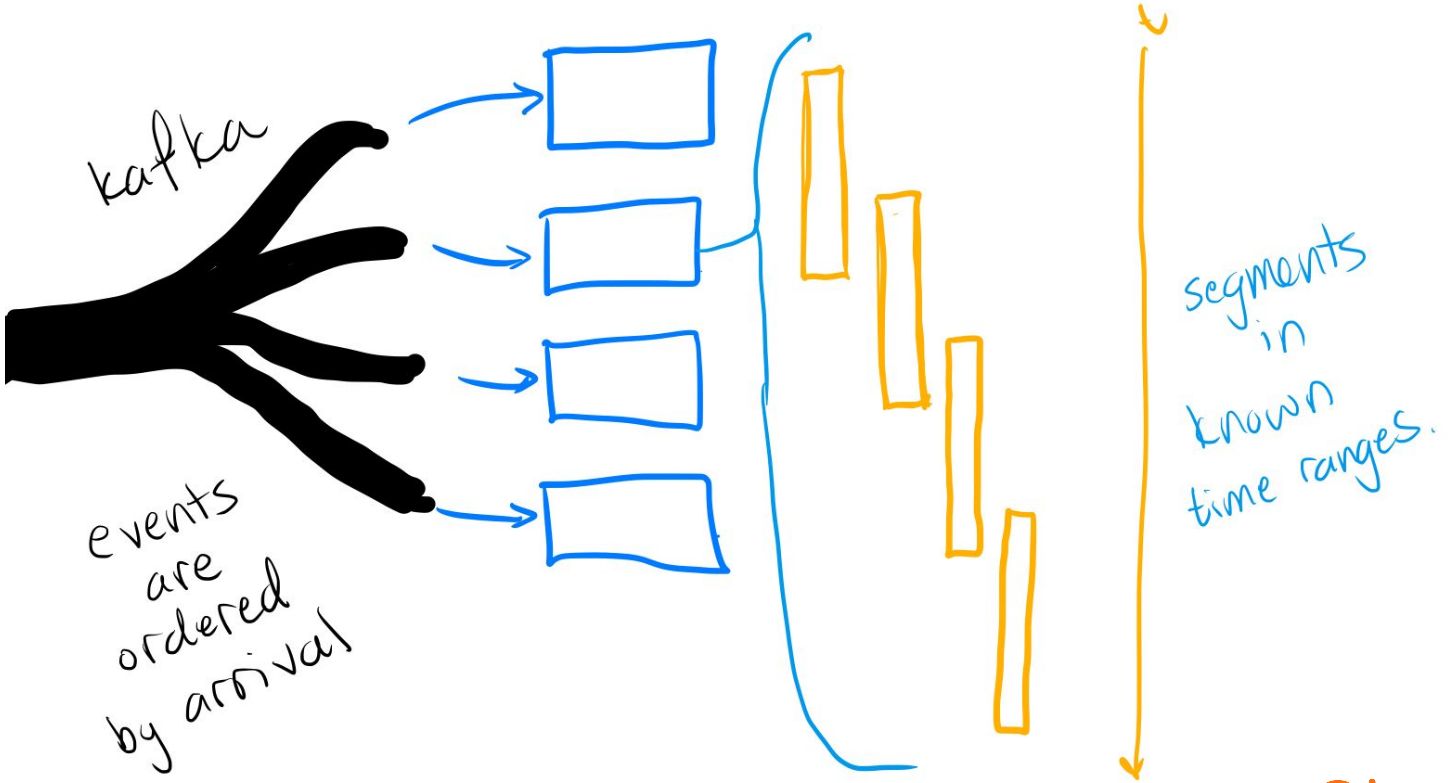
segment by arrival.

break at a million, or 1Gb, or 12hrs

Record the timestamp range per segment.

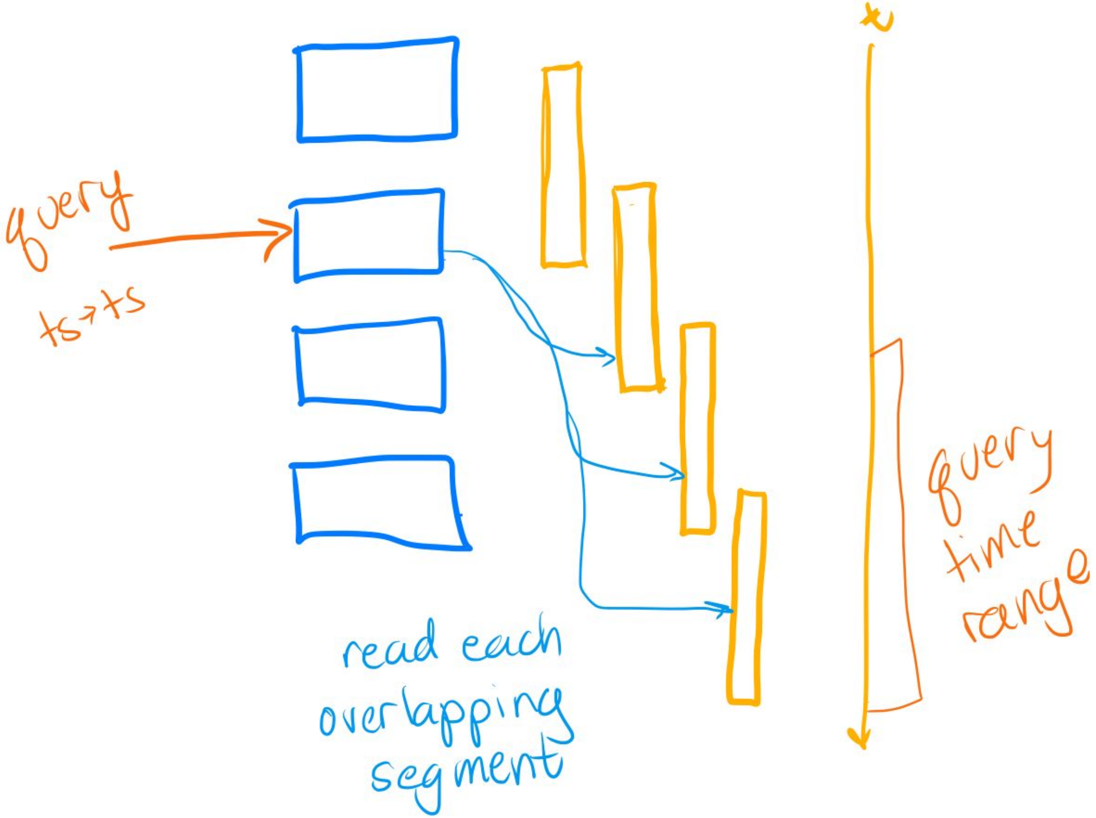
@jessitron

Retriever indexes segments by timestamp.



@jessitron

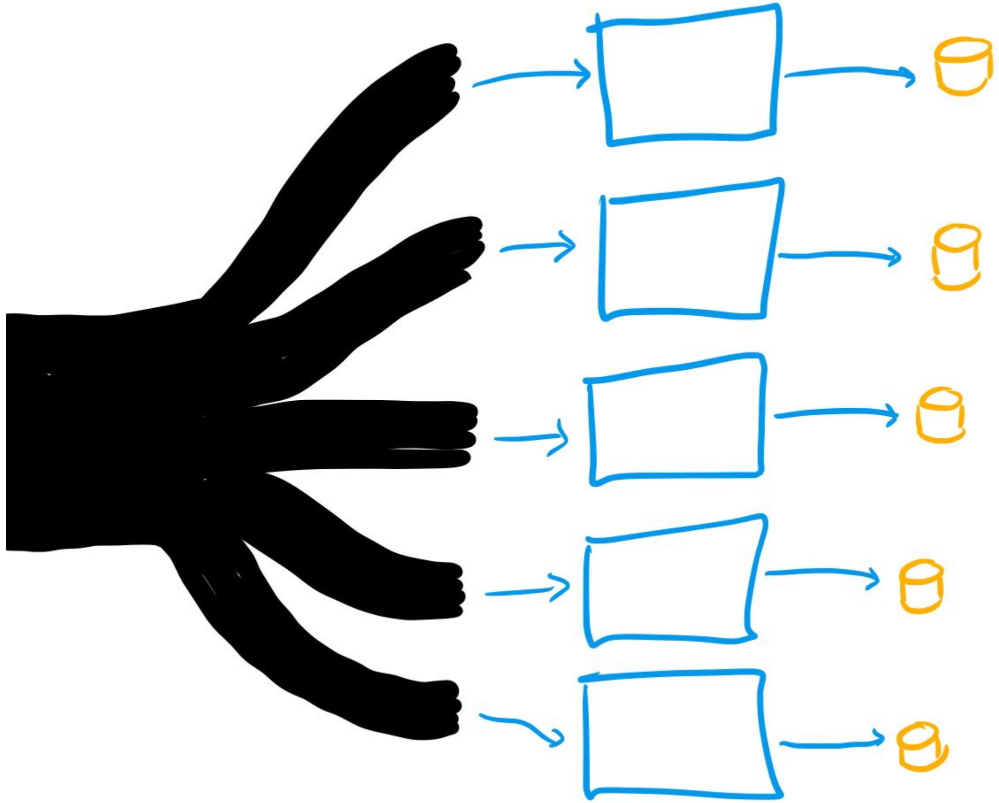
Retriever indexes segments by timestamp.



Dynamic aggregation of any fields across any time range

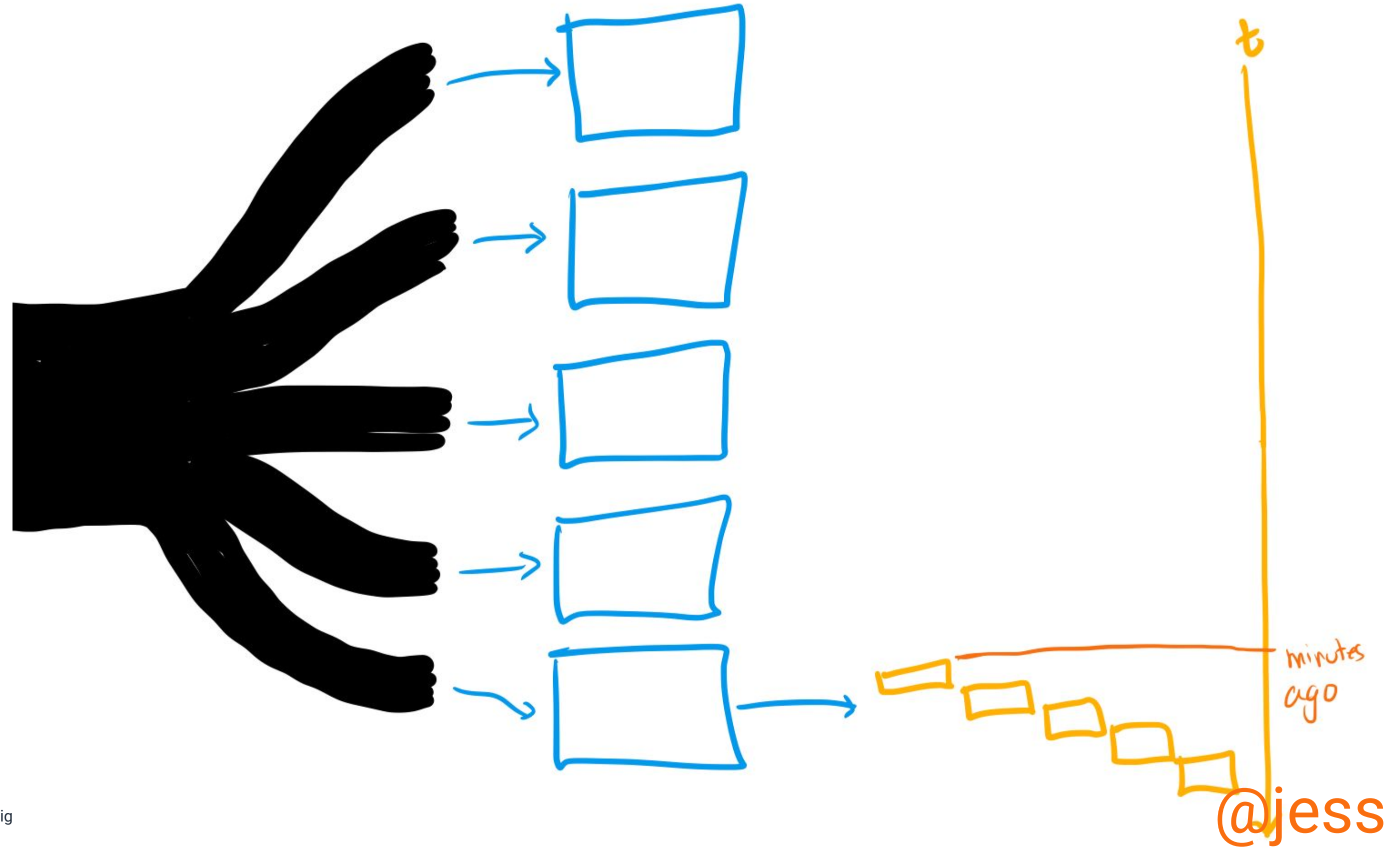
A custom datastore, carefully suited, continually optimized.

Bigger customers, more data coming in.



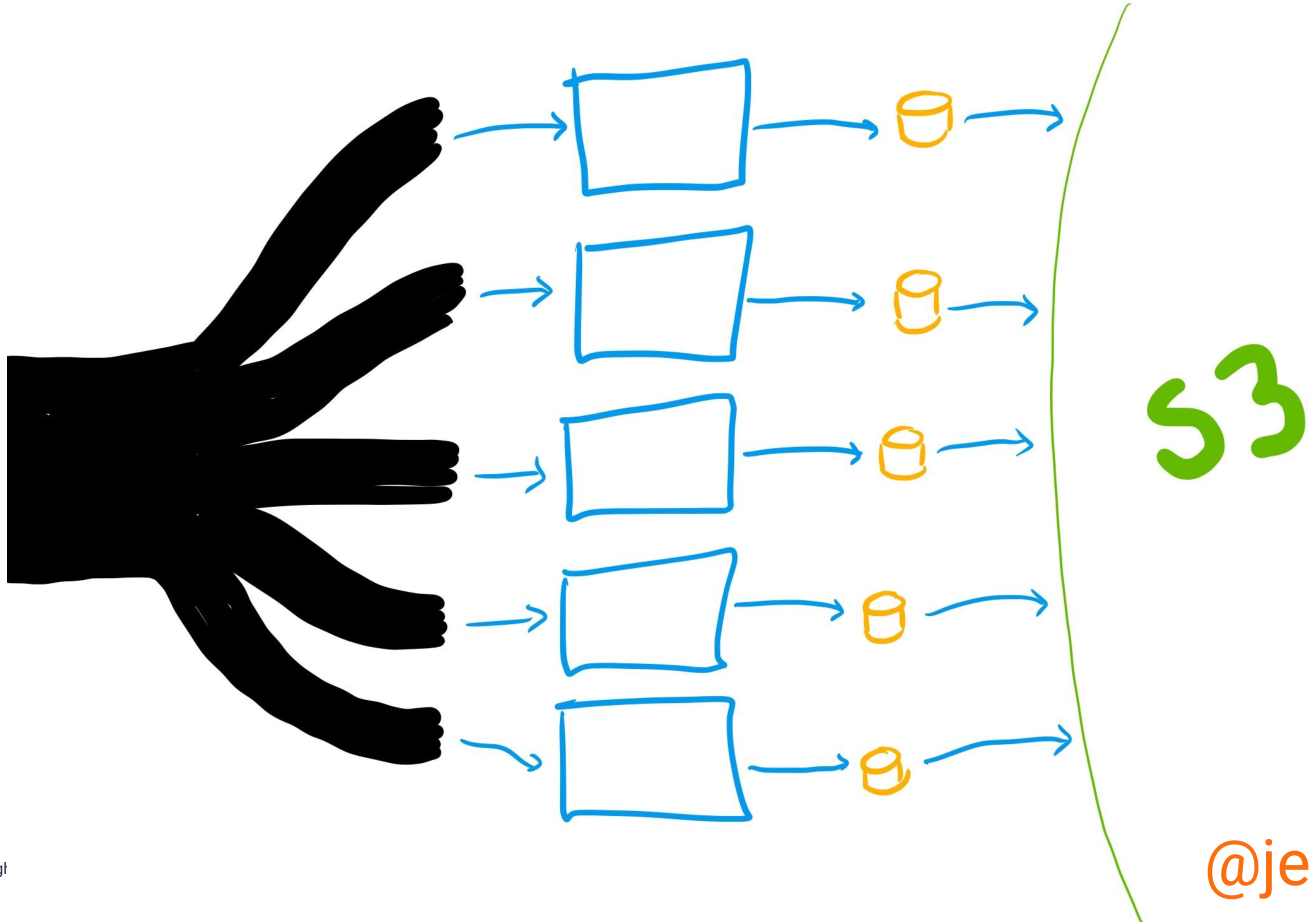
@jessitron

Segments hold a smaller time range.



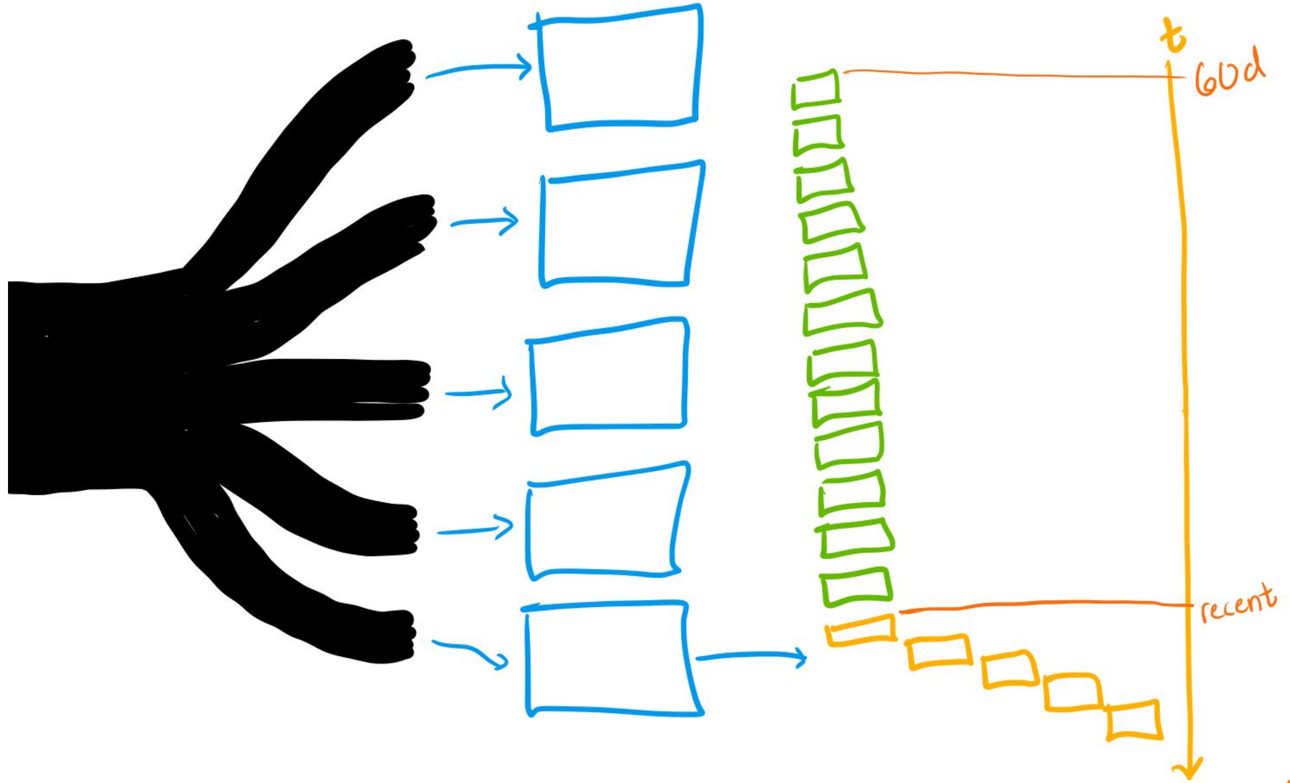
@jessitron

Solution: MOAR storage



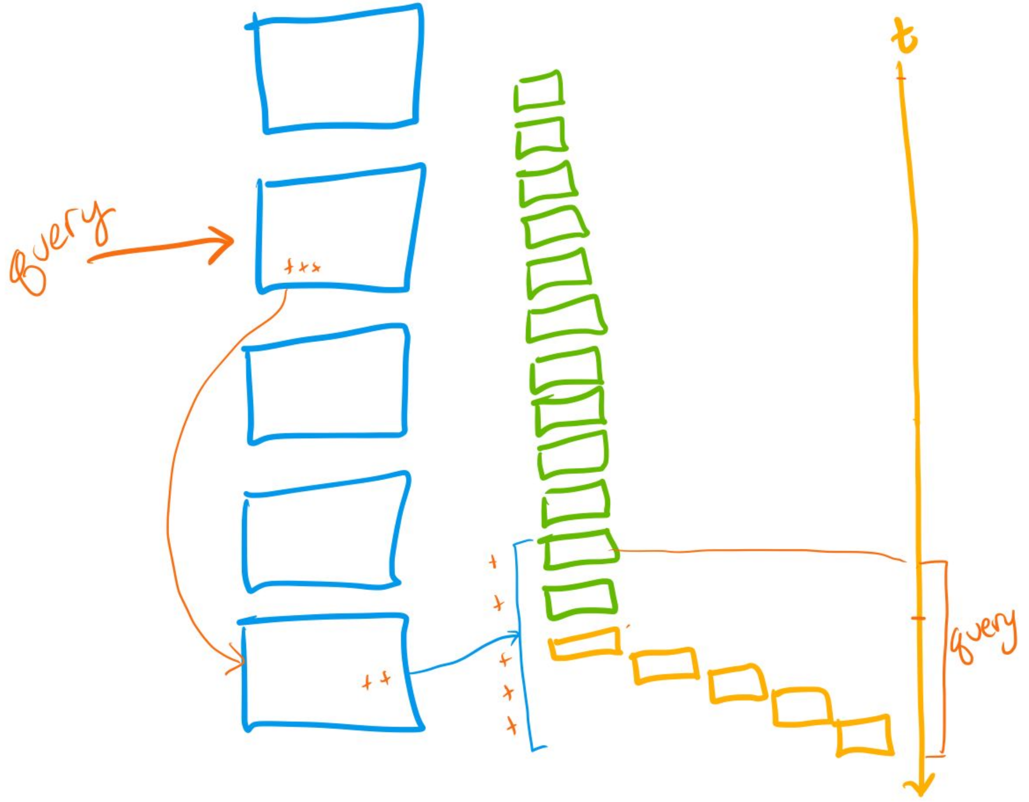
@jessitron

Now we can keep data for a fixed time range!



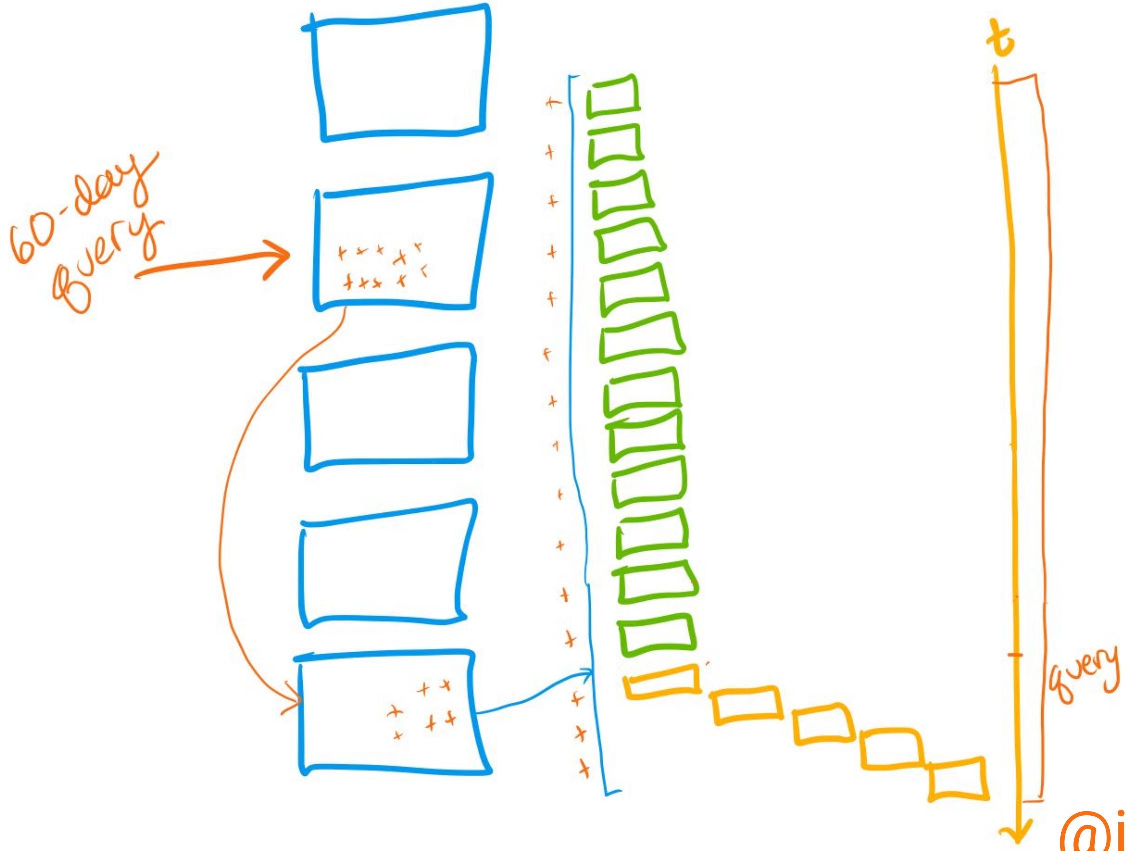
@jessitron

Retrievers grab data back from S3 at need.



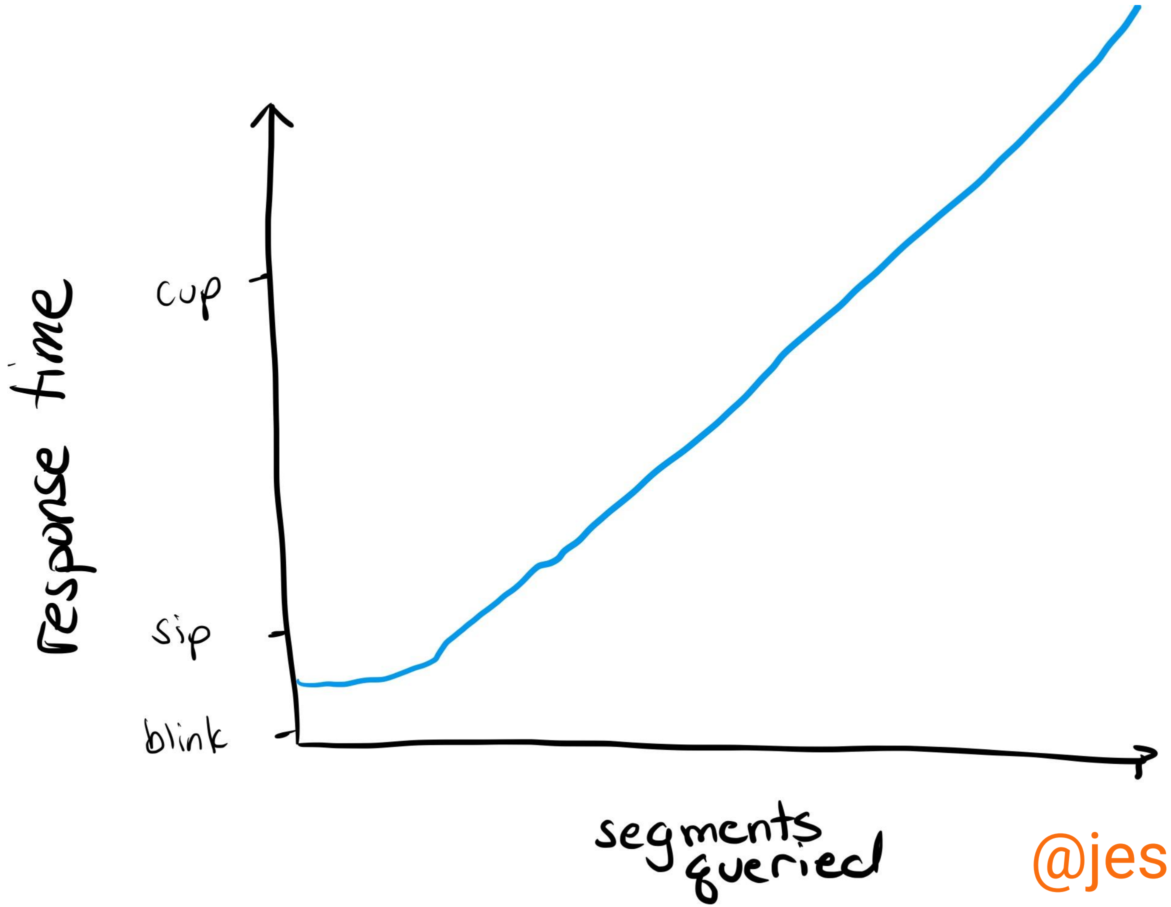
@jessitron

Now people can run queries over 60 days 😯



@jessitron

Now people can run queries over 60 days 😯

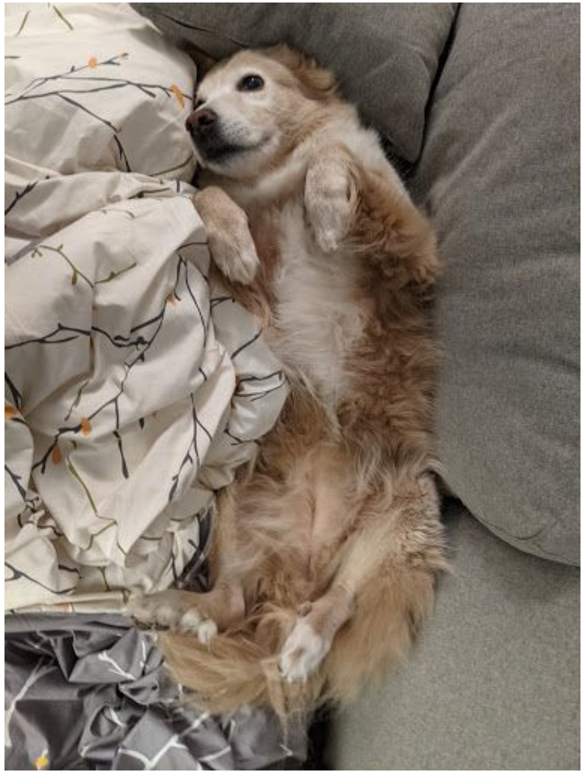


@jessitron

“

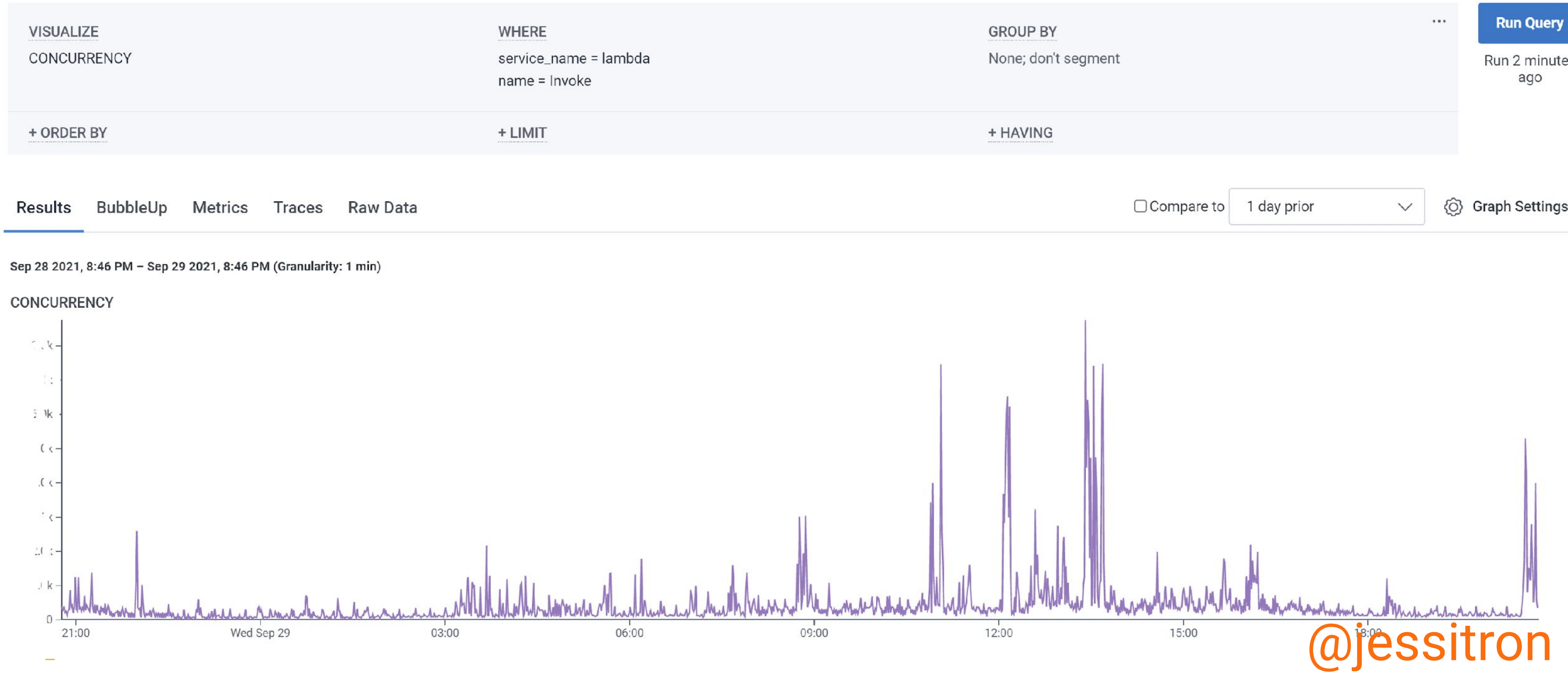
Lots more compute to play with, pretty please!
but only if I want to play!

Retrievers



@jessitron

MOAR compute, on demand.



VISUALIZE
CONCURRENCY

WHERE
service_name = lambda
name = Invoke

GROUP BY
None; don't segment

+ ORDER BY
+ LIMIT
+ HAVING

Run Query
Run 2 minute ago

Results BubbleUp Metrics Traces Raw Data

Compare to 1 day prior

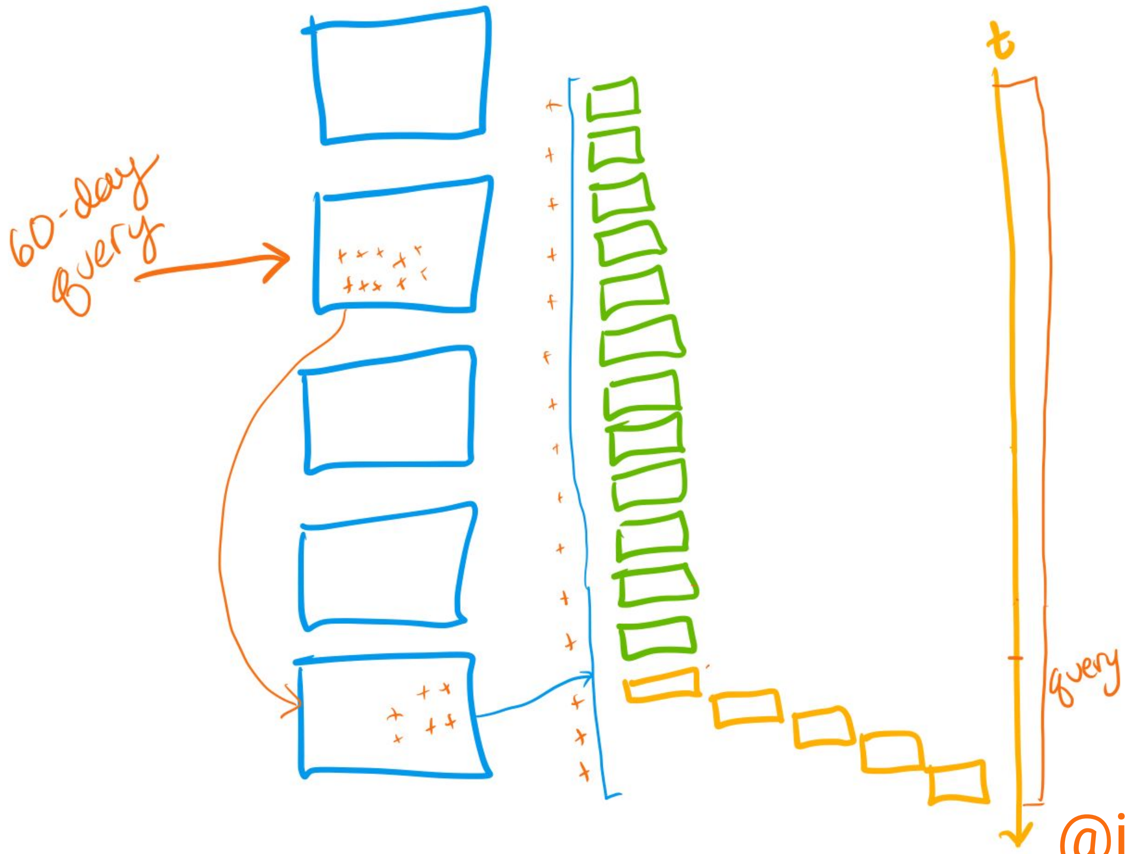
Graph Settings

Sep 28 2021, 8:46 PM – Sep 29 2021, 8:46 PM (Granularity: 1 min)

CONCURRENCY

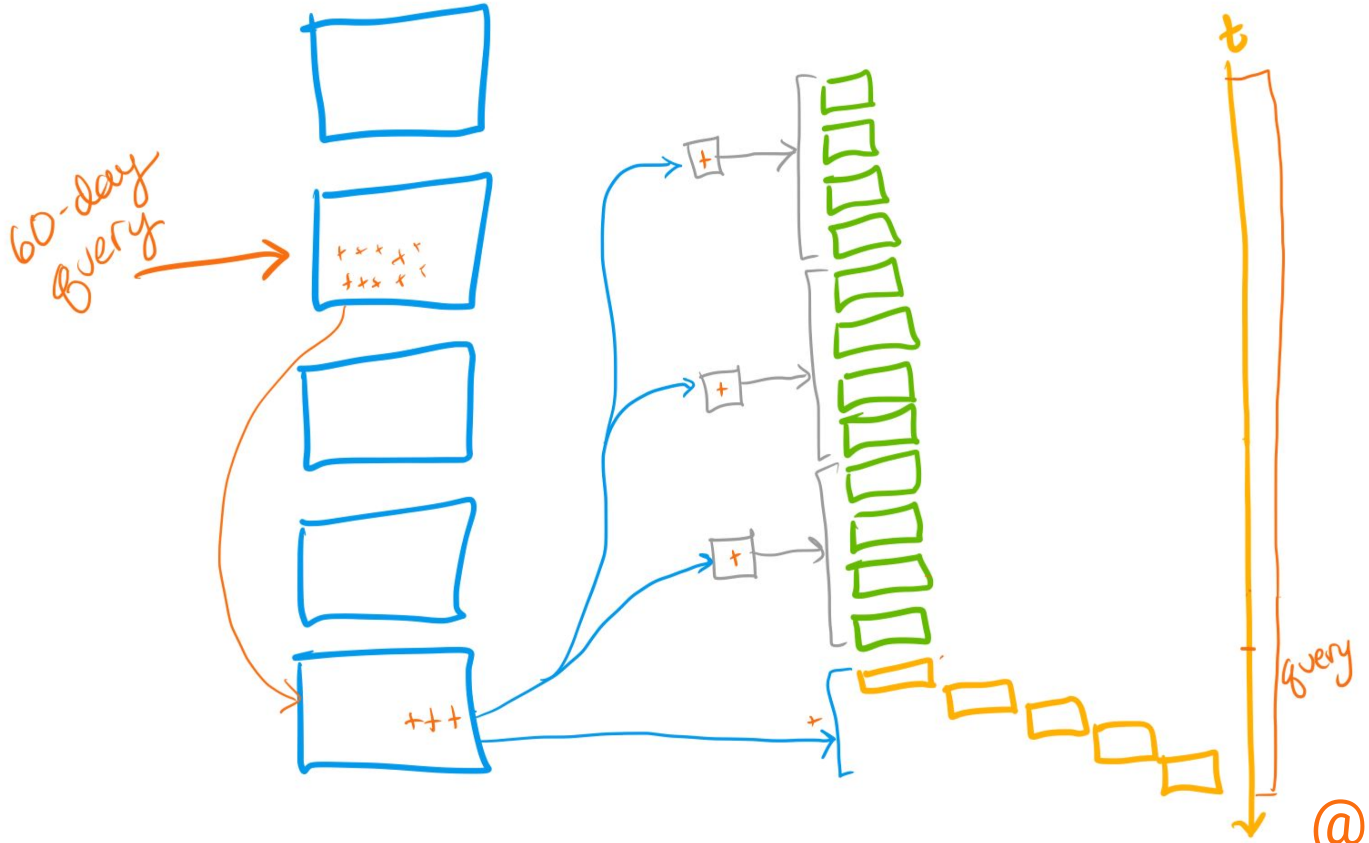
@jessitron

Problem: too much data for one retriever…

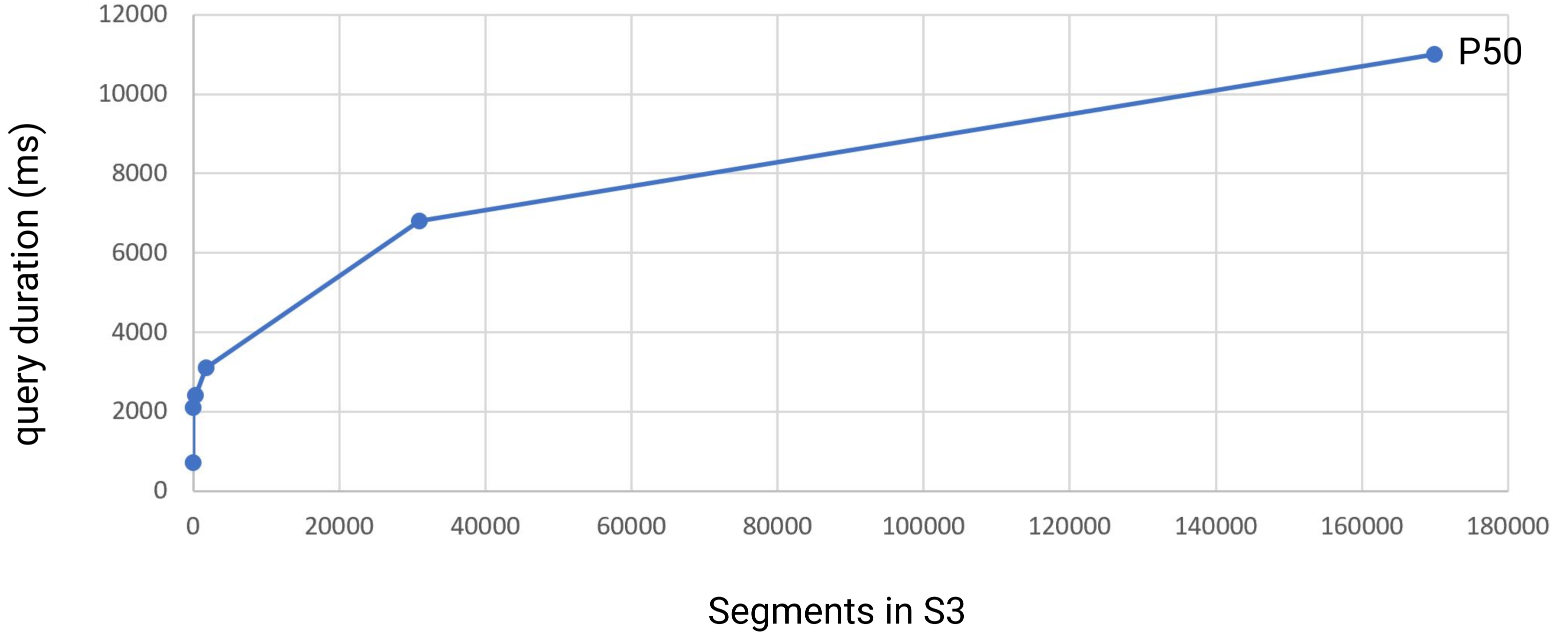


@jessitron

Solution: more compute, on demand.



Increase in query time is sublinear



Buy compute in ~~100ms~~1ms units

Compute scales with time range, so response time doesn't have to.

Lambda scales* up our compute

50ms

median* startup time

90%

of ours return* within 1.5s

3-4x

as expensive* as EC2

Considerations

Lambda *is* on-demand compute, but they didn’t build it for this.

Lambda scales up our compute

50ms

median startup time

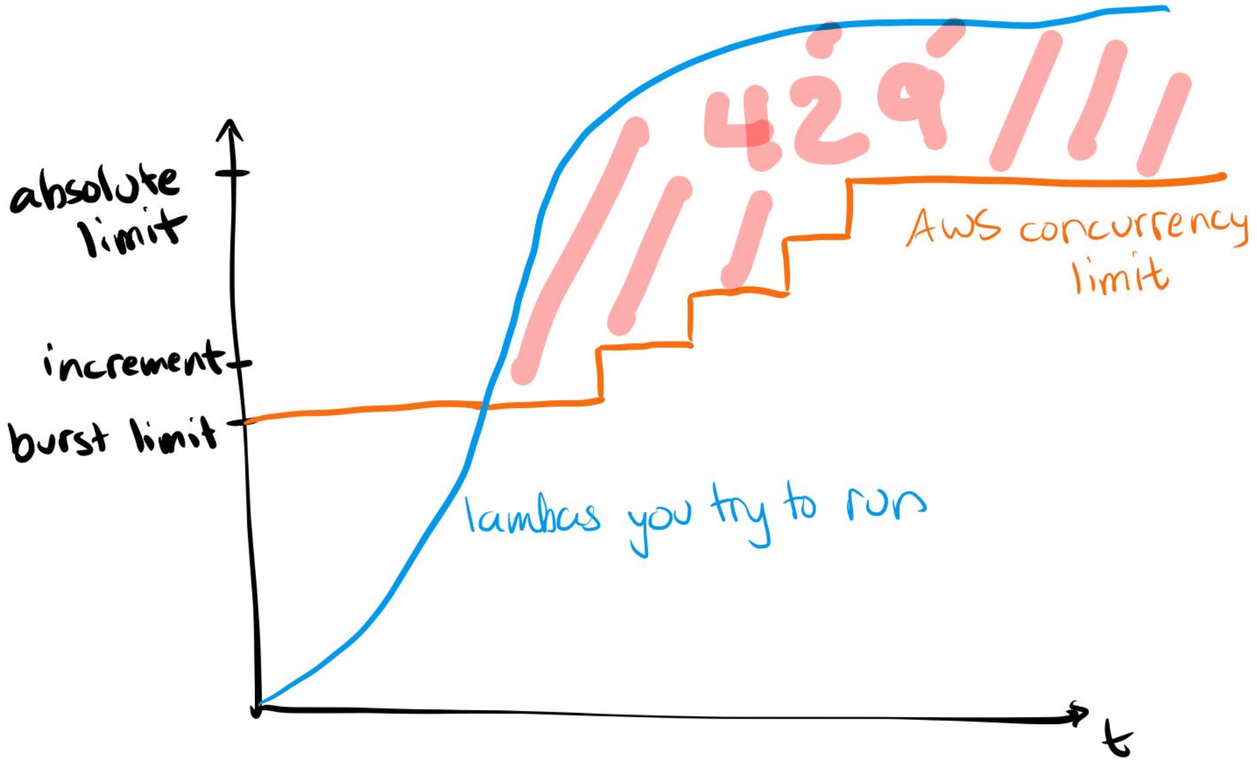
90%

of ours return within 1.5s

3-4x

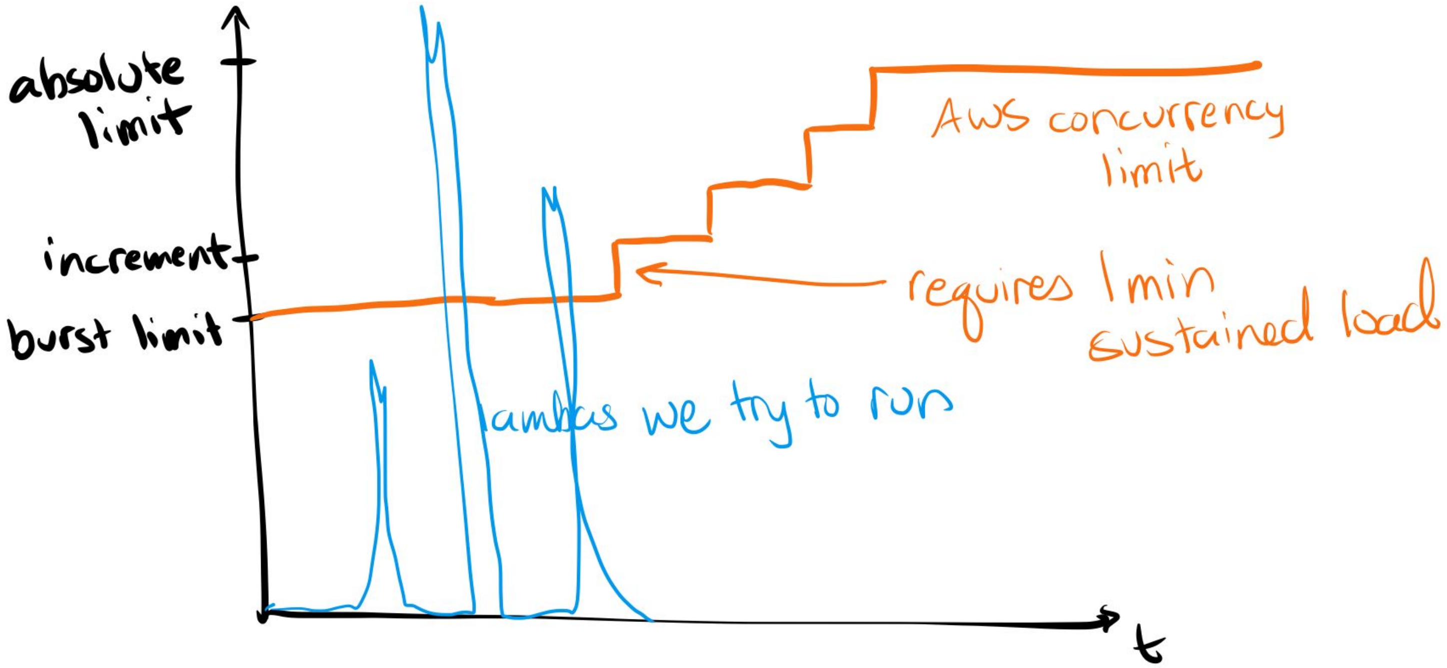
as expensive as EC2

Lambda scales... within limits



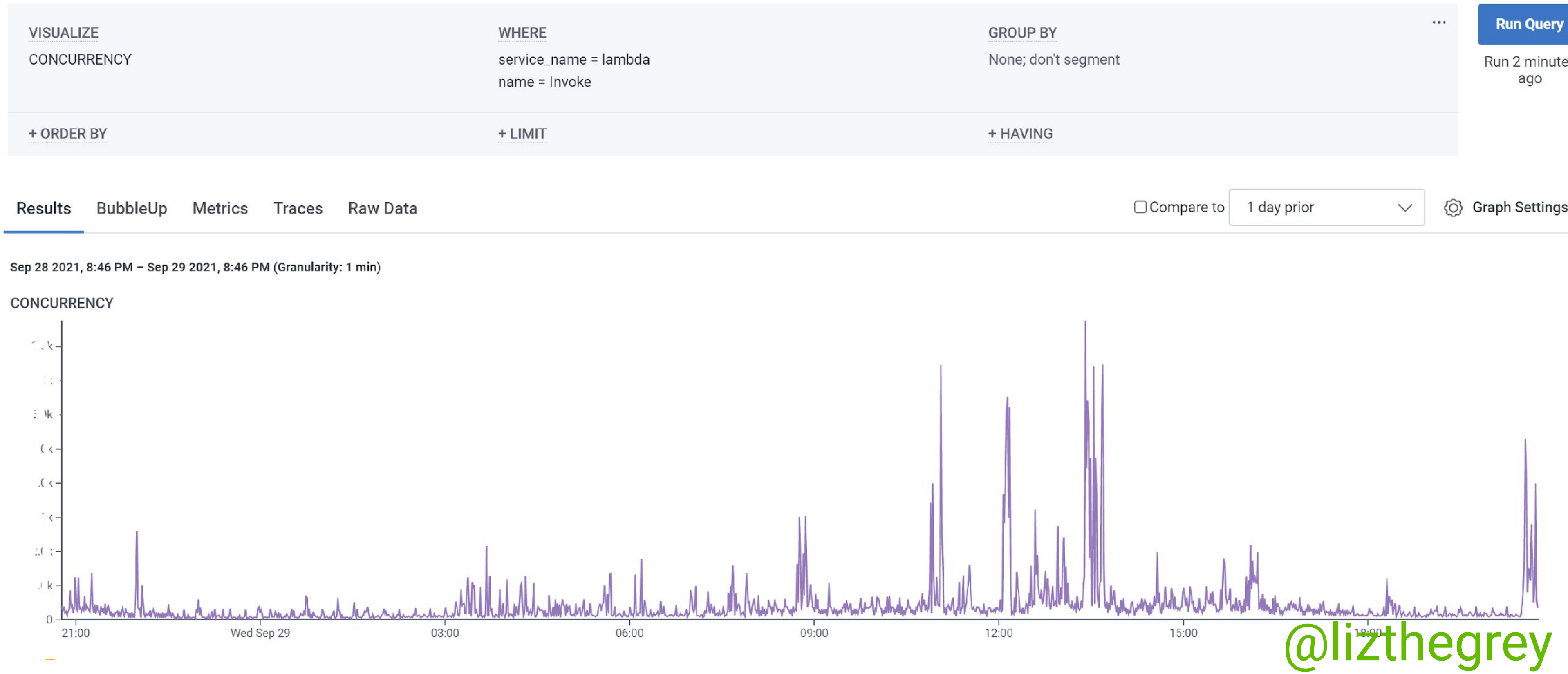
@lizthegrey

Lambda scales... within limits



@lizthegrey

Observability helps: concurrency



VISUALIZE
CONCURRENCY

WHERE
service_name = lambda
name = Invoke

GROUP BY
None; don't segment

+ ORDER BY

+ LIMIT

+ HAVING

Run Query

Run 2 minute ago

Results | BubbleUp | Metrics | Traces | Raw Data

☐ Compare to: 1 day prior

Graph Settings

Sep 28 2021, 8:46 PM – Sep 29 2021, 8:46 PM (Granularity: 1 min)

CONCURRENCY

Lambda scales... within limits

Study your limits:

https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

Change the SDK retry parameters

Observability helps 😉

Talk to your account reps

@lizthegrey

Lambda scales up our compute

50ms

median startup time

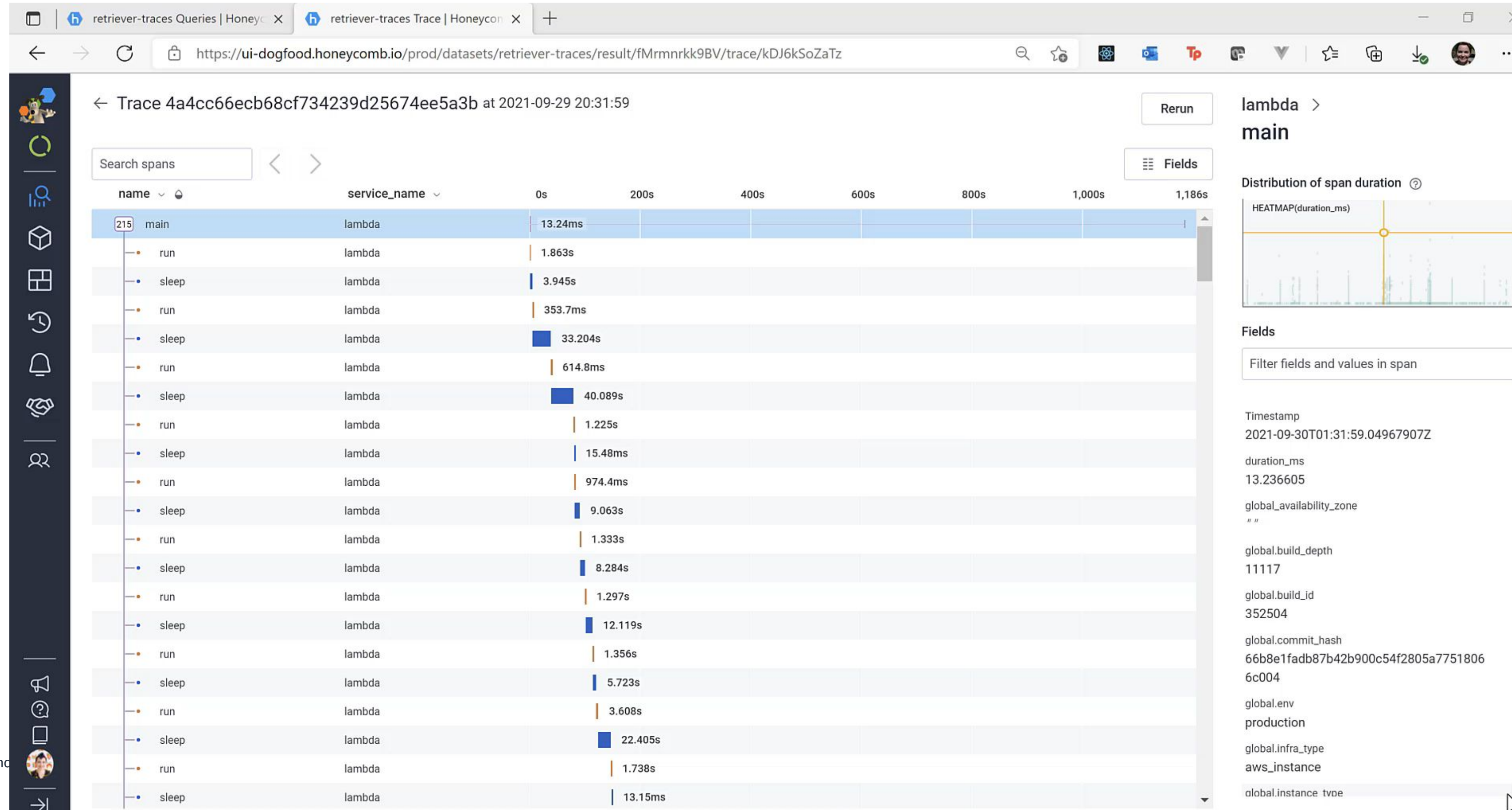
90%

of ours return within 1.5s

3-4x

as expensive as EC2

Functions start up… when they do



retriever-traces Queries | Honeyc
retriever-traces Trace | Honeycon

https://ui-dogfood.honeycomb.io/prod/datasets/retriever-traces/result/fMrmnrkk9BV/trace/kDJ6kSoZaTz

Trace 4a4cc66ecb68cf734239d25674ee5a3b at 2021-09-29 20:31:59

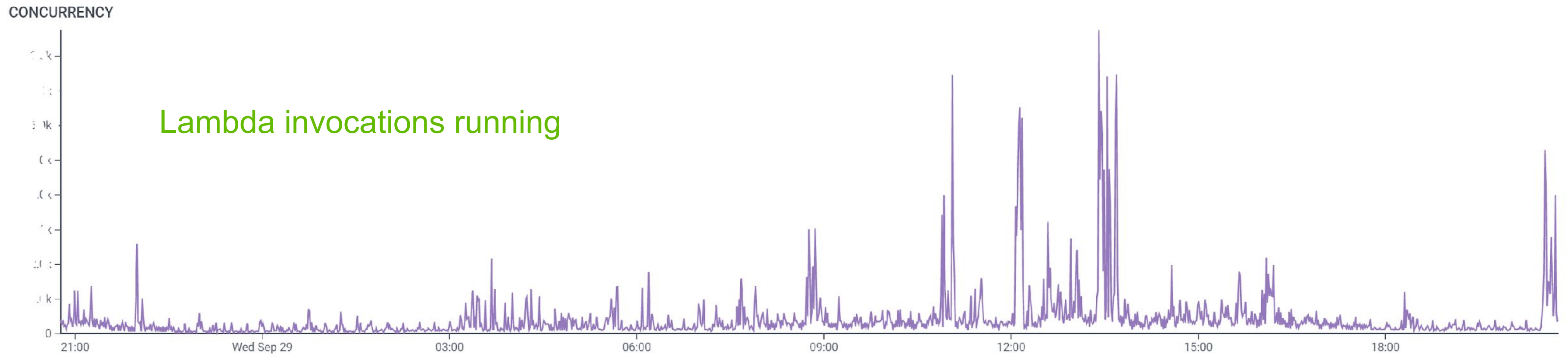
name	service_name	duration
215 main	lambda	13.24ms
run	lambda	1.863s
sleep	lambda	3.945s
run	lambda	353.7ms
sleep	lambda	33.204s
run	lambda	614.8ms
sleep	lambda	40.089s
run	lambda	1.225s
sleep	lambda	15.48ms
run	lambda	974.4ms
sleep	lambda	9.063s
run	lambda	1.333s
sleep	lambda	8.284s
run	lambda	1.297s
sleep	lambda	12.119s
run	lambda	1.356s
sleep	lambda	5.723s
run	lambda	3.608s
sleep	lambda	22.405s
run	lambda	1.738s
sleep	lambda	13.15ms

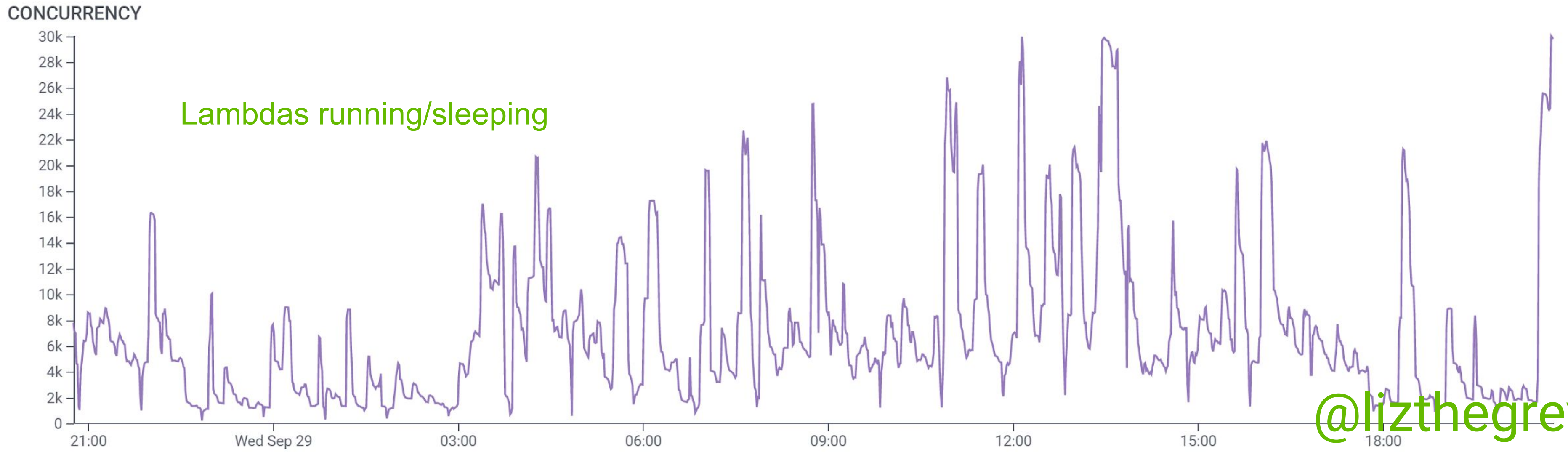
lambda
main

Distribution of span duration

Fields

Timestamp	2021-09-30T01:31:59.04967907Z
duration_ms	13.236605
global_availability_zone	" "
global.build_depth	11117
global.build_id	352504
global.commit_hash	66b8e1fadb87b42b900c54f2805a77518066c004
global.env	production
global.infra_type	aws_instance
global.instance_type	


CONCURRENCY
Lambda invocations running
0
21:00
Wed Sep 29
03:00
06:00
09:00
12:00
15:00
18:00


CONCURRENCY
Lambdas running/sleeping
30k
28k
26k
24k
22k
20k
18k
16k
14k
12k
10k
8k
6k
4k
2k
0
21:00
Wed Sep 29
03:00
06:00
09:00
12:00
15:00
18:00

@lizthegrey

Lambda scales up our compute

50ms

median startup time

90%

of ours return within 1.5s

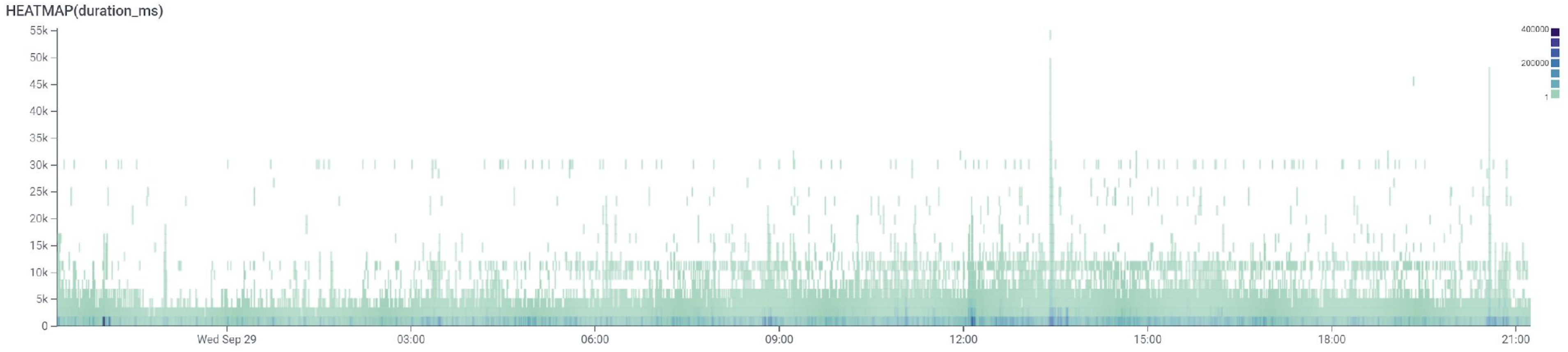
3-4x

as expensive as EC2

@lizthegrey

Functions return... usually

Sep 28 2021, 9:14 PM – Sep 29 2021, 9:14 PM (Granularity: 1 min)



@lizthegrey

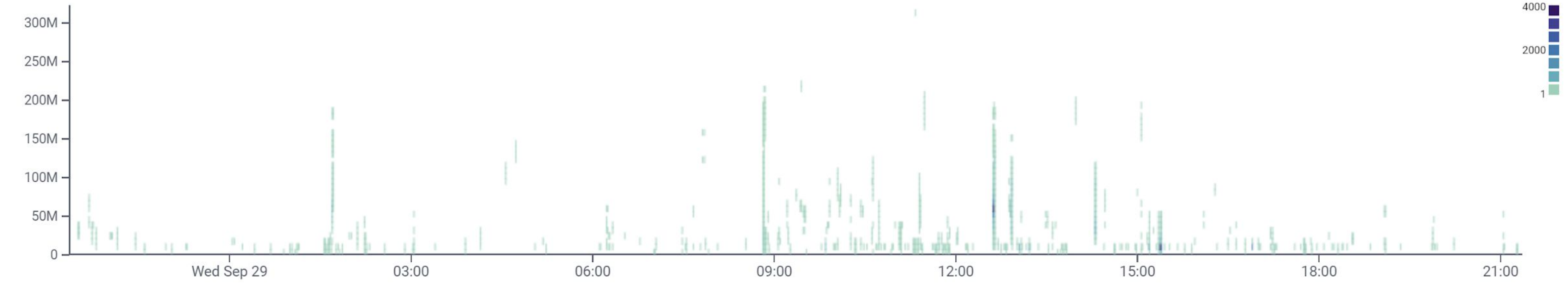
Functions accept... JSON

Put the data in S3 and send a link.

Functions return… up to 6Mb

Sep 28 2021, 9:22 PM – Sep 29 2021, 9:22 PM (Granularity: 1 min)

HEATMAP(app.response_size)



Put the data in S3 and send a link.

@lizthegrey

Lambda scales up our compute

50ms

median startup time

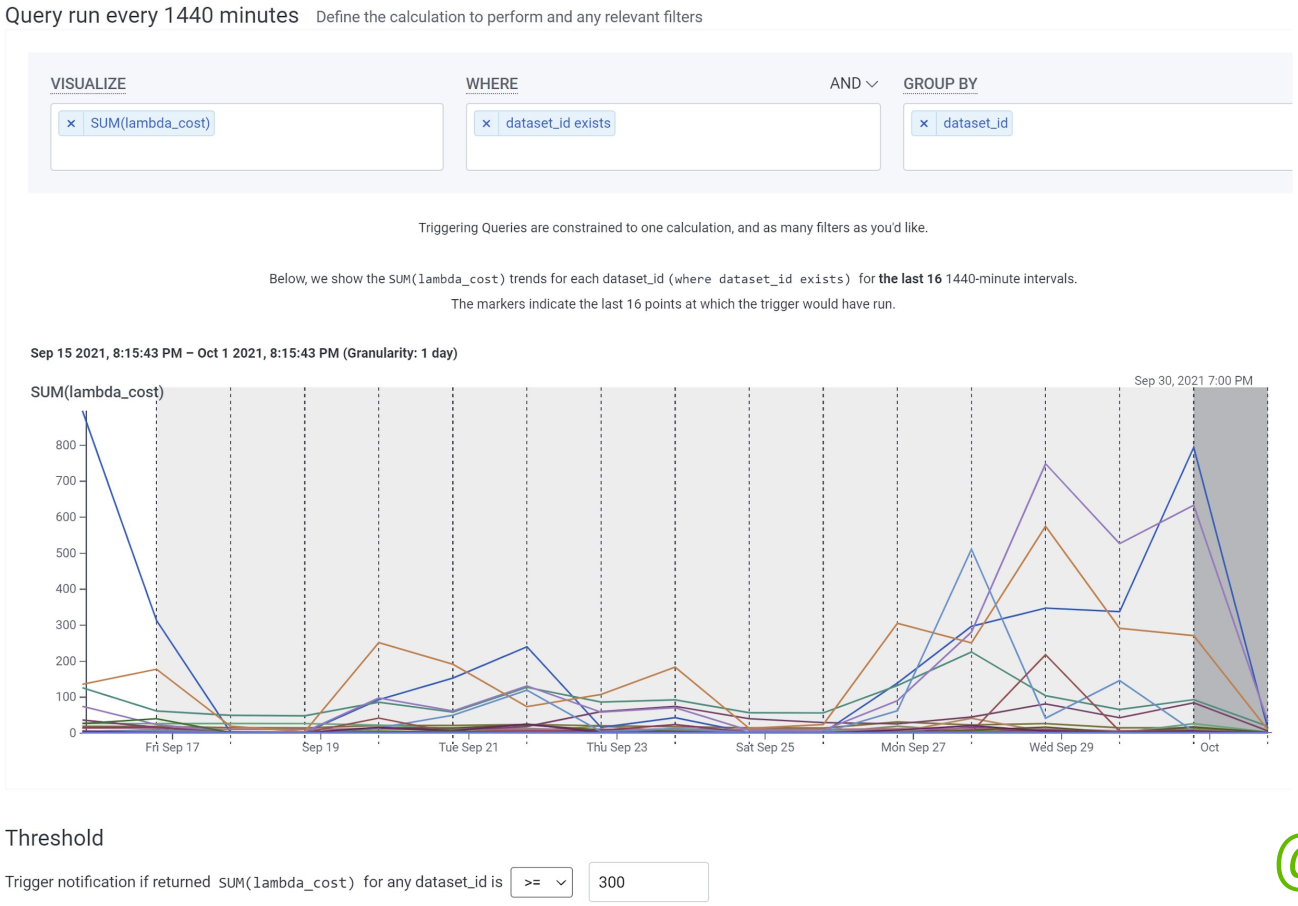
90%

of ours return within 1.5s

3-4x

as expensive as EC2

Functions cost… something



Query run every 1440 minutes Define the calculation to perform and any relevant filters

VISUALIZE: × SUM(lambda_cost)

WHERE AND: × dataset_id exists

GROUP BY: × dataset_id

Triggering Queries are constrained to one calculation, and as many filters as you'd like.

Below, we show the `SUM(lambda_cost)` trends for each dataset_id (`where dataset_id exists`) for **the last 16** 1440-minute intervals.
The markers indicate the last 16 points at which the trigger would have run.

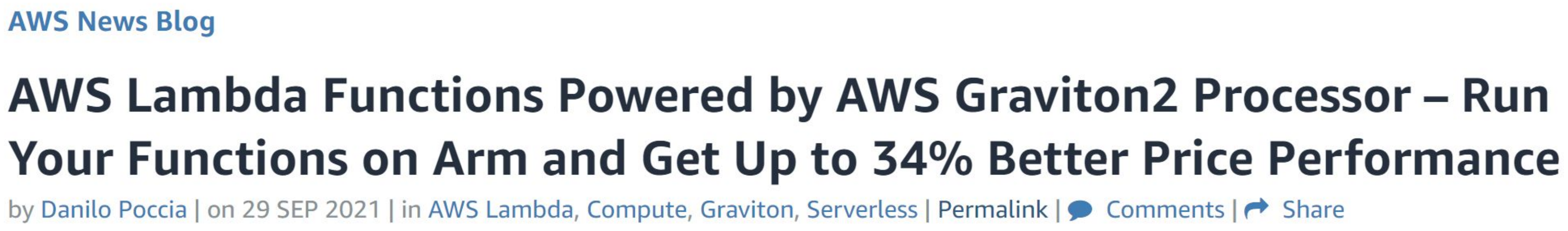
Sep 15 2021, 8:15:43 PM – Oct 1 2021, 8:15:43 PM (Granularity: 1 day)

Threshold

Trigger notification if returned `SUM(lambda_cost)` for any dataset_id is >= 300

@lizthegrey

Functions cost… let's make it less?



AWS News Blog

AWS Lambda Functions Powered by AWS Graviton2 Processor – Run Your Functions on Arm and Get Up to 34% Better Price Performance

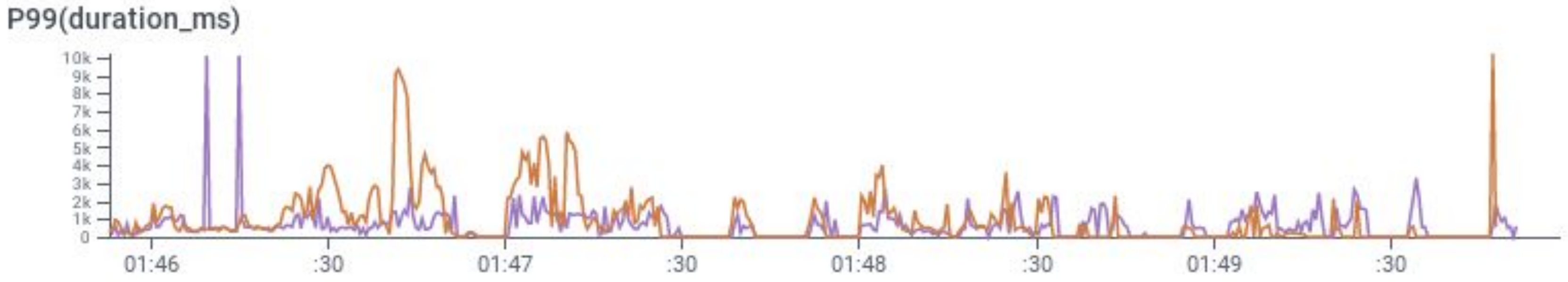
by Danilo Poccia | on 29 SEP 2021 | in AWS Lambda, Compute, Graviton, Serverless | Permalink | Comments | Share

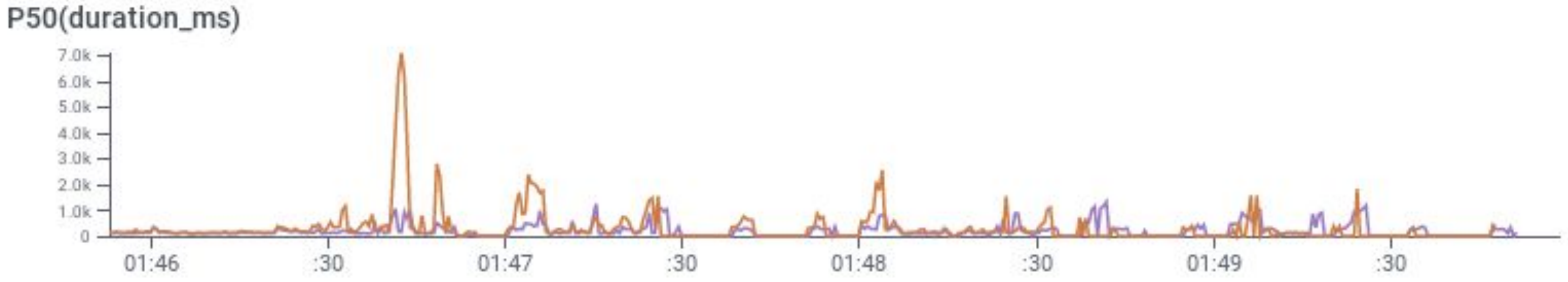
@lizthegrey

“ **M6g instances are superior to C5 in every aspect—they cost less, have more RAM, exhibit lower median and significantly narrower tail latency, and run cooler with the same proportional workload per host. Converting our entire ingest worker fleet has allowed us to run 30% fewer instances, and each instance costs 10% less.**

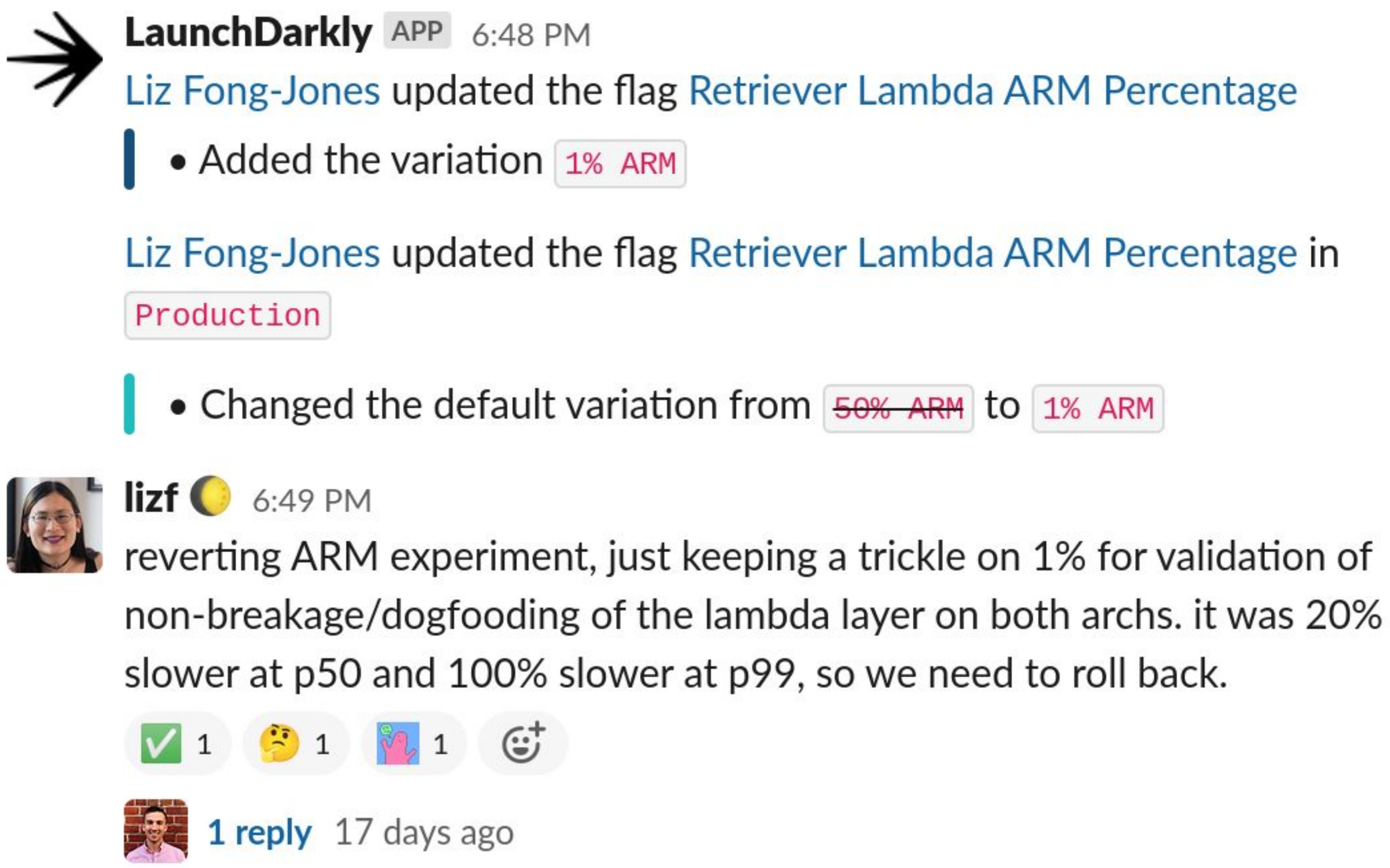
Yours Truly

Observability helps!





	arch	COUNT	HEATMAP(Log_Duration)	P99(duration_ms)	P50(duration_ms)
🟪	amd64	262,988		1,168.09377	139.24663
🟫	arm64	161,394		2,677.62006	175.50275



LaunchDarkly APP 6:48 PM

Liz Fong-Jones updated the flag Retriever Lambda ARM Percentage

- Added the variation `1% ARM`

Liz Fong-Jones updated the flag Retriever Lambda ARM Percentage in `Production`

- Changed the default variation from ~~`50% ARM`~~ to `1% ARM`

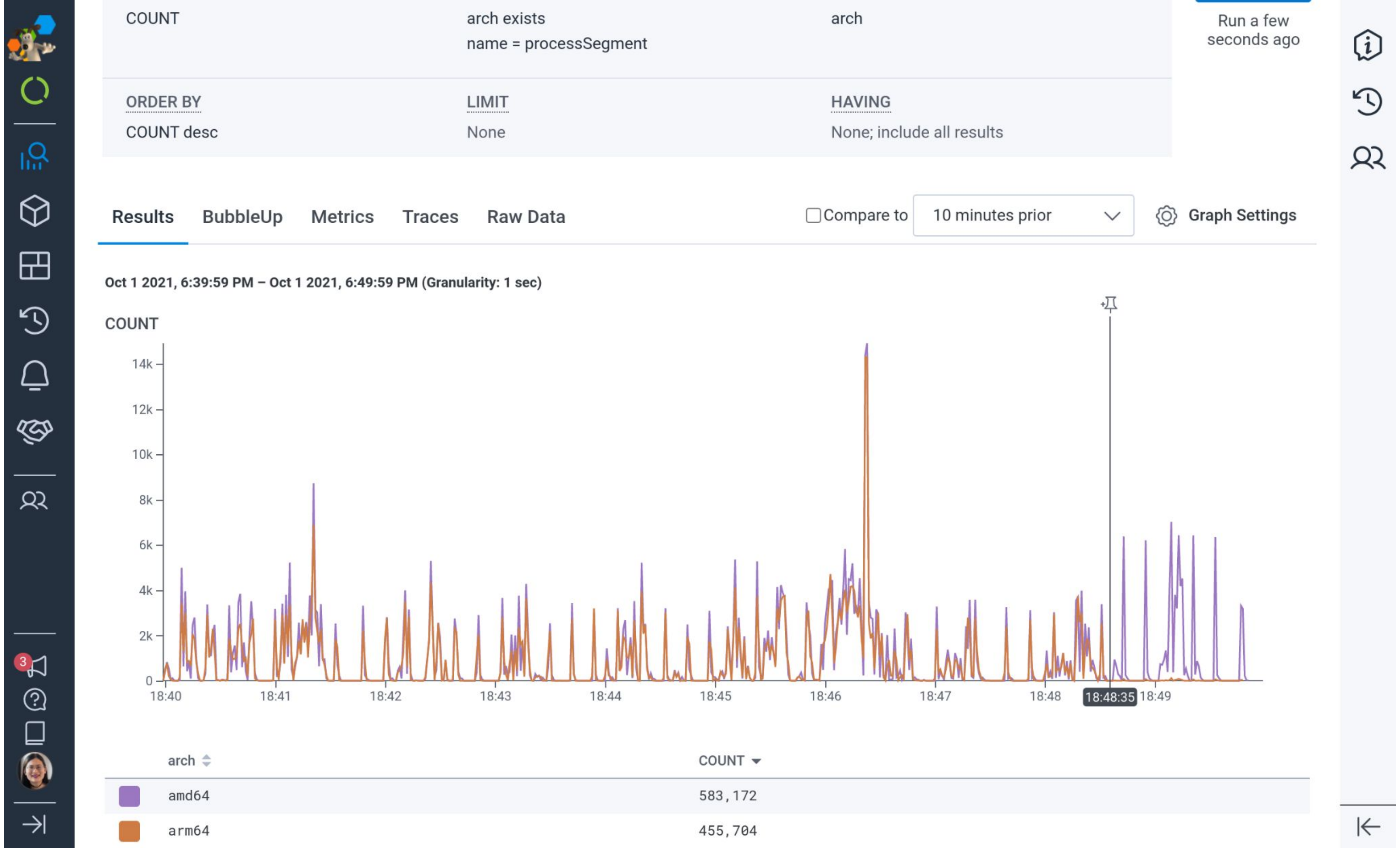
lizf 6:49 PM

reverting ARM experiment, just keeping a trickle on 1% for validation of non-breakage/dogfooding of the lambda layer on both archs. it was 20% slower at p50 and 100% slower at p99, so we need to roll back.

✅ 1 🤔 1 1

1 reply 17 days ago

@lizthegrey



COUNT	arch exists name = processSegment	arch
ORDER BY COUNT desc	LIMIT None	HAVING None; include all results

Run a few seconds ago

Oct 1 2021, 6:39:59 PM – Oct 1 2021, 6:49:59 PM (Granularity: 1 sec)

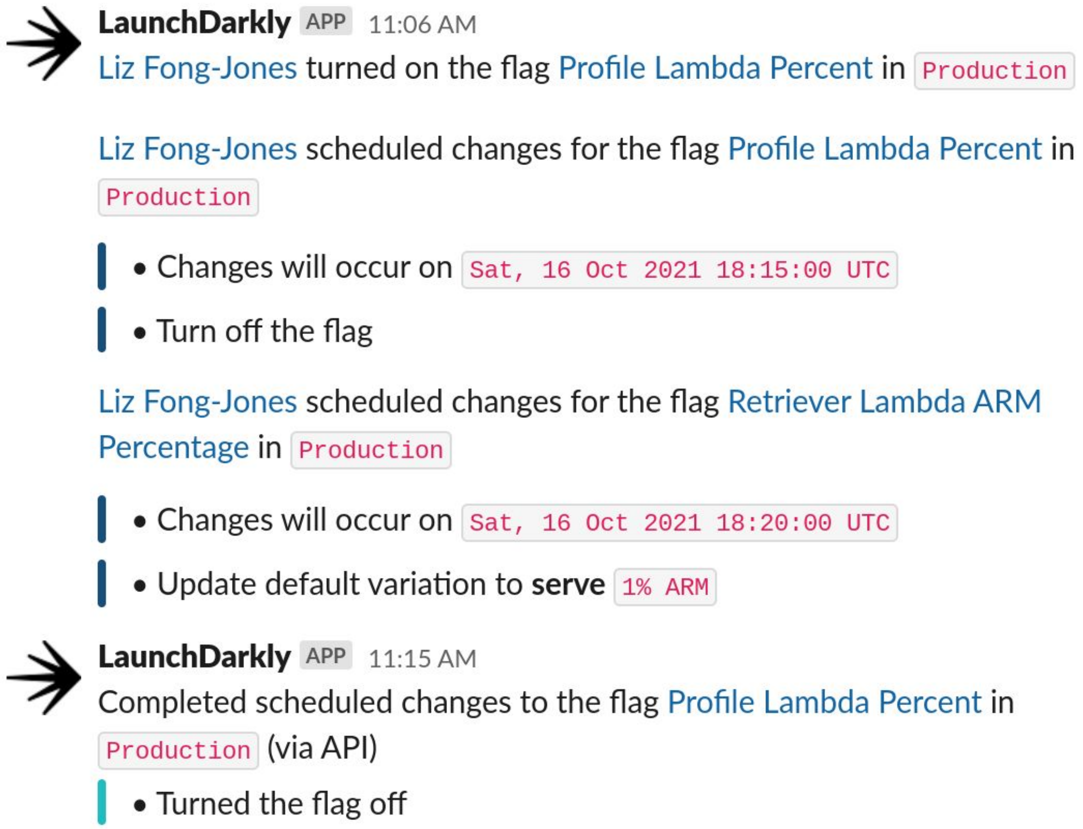
	arch	COUNT
🟪	amd64	583,172
🟧	arm64	455,704

@lizthegrey

Why so slow?

- AWS capacity constraints
- Go register calling convention
- lz4 library asm optimization

Making progress carefully



LaunchDarkly APP 11:06 AM
Liz Fong-Jones turned on the flag Profile Lambda Percent in `Production`

Liz Fong-Jones scheduled changes for the flag Profile Lambda Percent in `Production`

- Changes will occur on `Sat, 16 Oct 2021 18:15:00 UTC`
- Turn off the flag

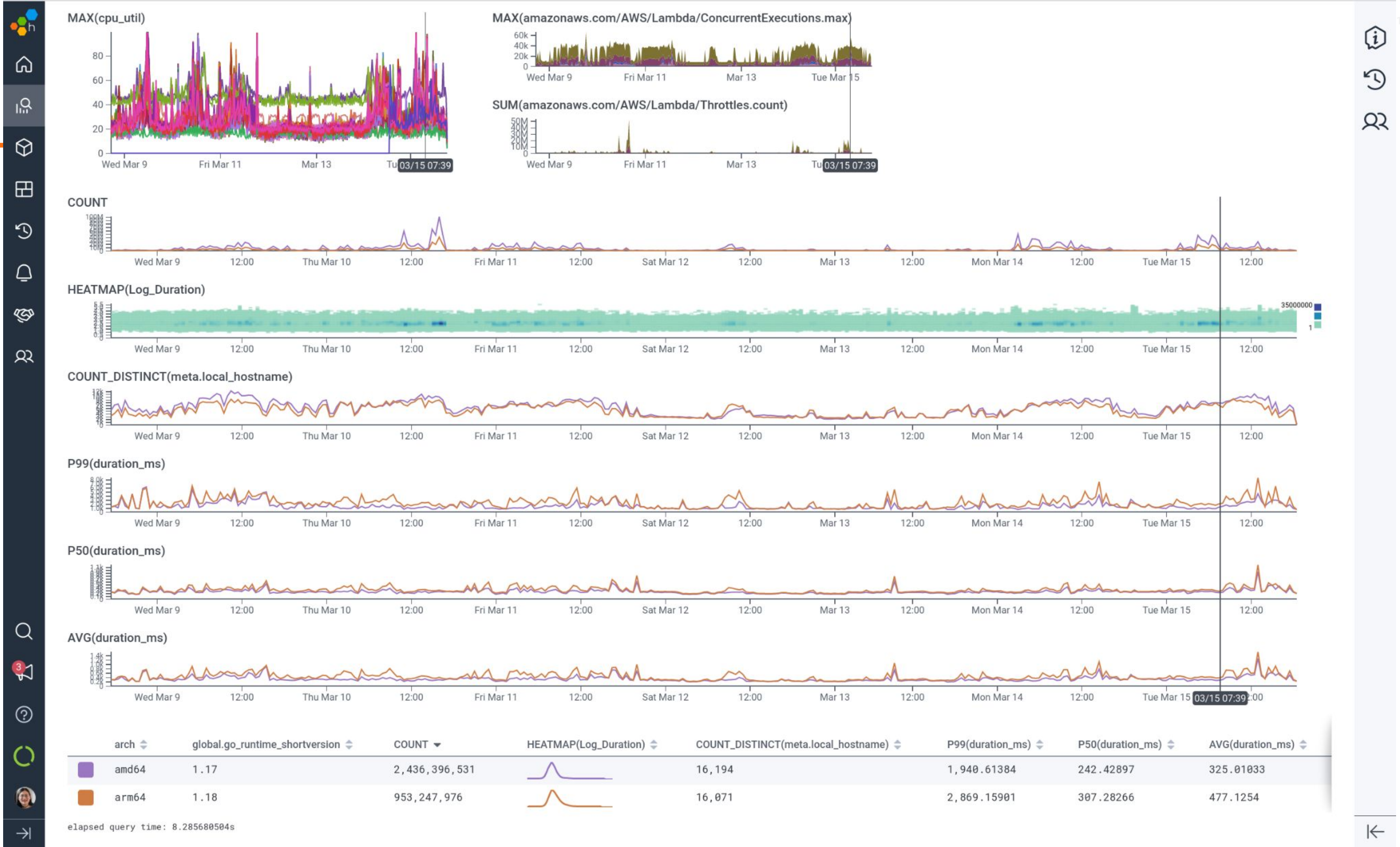
Liz Fong-Jones scheduled changes for the flag Retriever Lambda ARM Percentage in `Production`

- Changes will occur on `Sat, 16 Oct 2021 18:20:00 UTC`
- Update default variation to **serve** `1% ARM`

LaunchDarkly APP 11:15 AM
Completed scheduled changes to the flag Profile Lambda Percent in `Production` (via API)

- Turned the flag off

@lizthegrey



	arch	global.go_runtime_shortversion	COUNT	HEATMAP(Log_Duration)	COUNT_DISTINCT(meta.local_hostname)	P99(duration_ms)	P50(duration_ms)	AVG(duration_ms)
🟪	amd64	1.17	2,436,396,531		16,194	1,940.61384	242.42897	325.01033
🟧	arm64	1.18	953,247,976		16,071	2,869.15901	307.28266	477.1254

elapsed query time: 8.285680504s

Yes*, do this at home!

Most realtime bulk workloads benefit

- **Move** state from local machines onto object storage
- **Shard** list of objects into work units
- **Parallelize** object processing
- **Reduce** results outside Lambda afterwards

Just beware the dragons

- **Avoid latency-insensitive** batch workloads (cost)
- **Avoid tiny** workloads (set-up latency)
- Check **cloud provider limits**, state your intentions (capacity planning)

Do this before scaling out

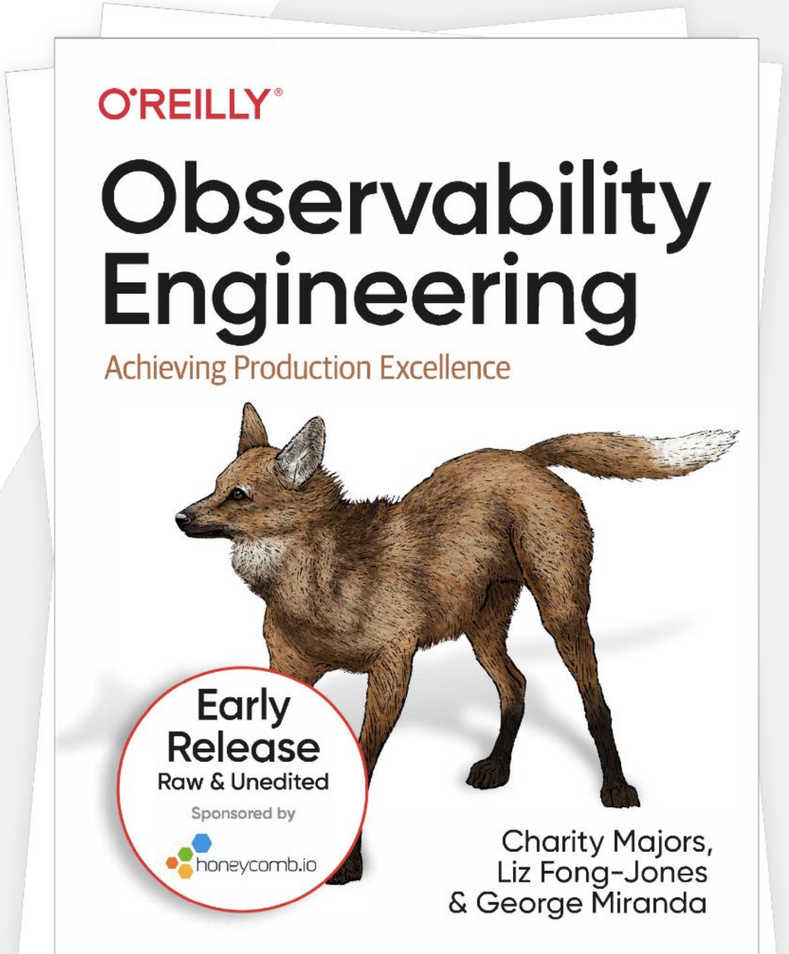
- Ensure it's **tuned properly** (items/invoke, CPU/RAM ratio)
- Ensure your code is **optimized properly** (esp if multi-arch)
- Ensure you use **observability layers** (e.g. OTel layer)
- Measure **metrics** carefully (esp cost)

> “ Remember: nothing matters unless users (developers) are happy

@jessitron

Observability Engineering

Explore preview chapters from our new book



@lizthegrey

@honeycombio @jessitron @lizthegrey

honeycomb.io

www.honeycomb.io

https://www.honeycomb.io/blog/speeding-things-up-so-your-queries-can-bee-faster/

AWS Lambda Instrumentation | Honeycomb