Using Serverless Functions for Real-time Observability

SRECon, 16 March 2022

Liz Fong-Jones @lizthegrey
Jessica Kerr @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
What is Retriever for?

It’s a distributed column store for **real-time event aggregation**
What is Retriever for?

Real-time event aggregation for interactive querying over traces
What is Honeycomb for?

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

what do you want to watch for?

I'll count it up and make those graphs fast!
Monitoring

what do you want to watch for?

Observability

what do you want to see right now?

I'll count it up and make those graphs fast!

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

Your apps

Events

with any number of fields

Retriever

Queries
Retriever is a distributed datastore.
Retriever is a distributed datastore.

Each Retriever reads & aggregates

query
Retriever is a distributed column store.
Retriever is a distributed column store.
Retriever indexes segments by timestamp.

Events are ordered by arrival.

Segment by arrival. Break at a million, or 1Gb, or 12hrs.

Record the timestamp range per segment.
Retriever indexes segments by timestamp.

Events are ordered by arrival.
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.
Segments hold a smaller time range.
Solution: MOAR storage
Now we can keep data for a fixed time range!
Retrievers grab data back from S3 at need.
Now people can run queries over 60 days 😯
Now people can run queries over 60 days 😯
"Lots more compute to play with, pretty please! but only if I want to play!"
Retrievers
MOAR compute, on demand.

<table>
<thead>
<tr>
<th>VISUALIZE</th>
<th>WHERE</th>
<th>GROUP BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCURRENCY</td>
<td>service_name = lambda name = invoke</td>
<td>None; don't segment</td>
</tr>
</tbody>
</table>

+ ORDER BY + LIMIT + HAVING

Results | BubbleUp | Metrics | Traces | Raw Data
---|---|---|---|---

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

CONCURRENCY

@jessitron
Problem: too much data for one retriever...
Solution: more compute, on demand.
Increase in query time is sublinear

query duration (ms)

Segments in S3

P50
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

@jessitron
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

- Absolute limit
- Increment
- Burst limit

AWS concurrency limit

Lambdas you try to run
Lambda scales... within limits

- **Absolute limit**
- **Increment**
- **Burst limit**

AWS concurrency limit

Requires 1 min sustained load

Lambdas we try to run
Observability helps: concurrency

```
+ ORDER BY
+ LIMIT
+ HAVING
```
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
Lambda scales up our compute

- 50ms median startup time
- 90% of ours return within 1.5s
- 3–4x as expensive as EC2

@lizthegrey
Functions start up... when they do

<table>
<thead>
<tr>
<th>name</th>
<th>service_name</th>
<th>0s</th>
<th>200s</th>
<th>400s</th>
<th>600s</th>
<th>800s</th>
<th>1,000s</th>
<th>1,180s</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>lambda</td>
<td>13.24ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>1.945s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>393.7ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>33.204s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>614.8ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>1.225s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>15.48ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>974.4ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>9.063s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>1.332s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>8.284s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>1.297s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>12.119s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>1.356s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>5.723s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>3.600s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>22.405s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>1.738s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sleep</td>
<td>1.153s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lambda invocations running

Lambdas running/sleeping
Lambda scales up our compute

50ms median startup time

90% of ours return within 1.5s

3–4x as expensive as EC2
Functions return... usually
Functions accept... JSON

Put the data in S3 and send a link.
Functions return... up to 6Mb

Put the data in S3 and send a link.
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  WHERE  AND  GROUP BY

- SUM(lambda_cost)
- dataset_id exists
- dataset_id

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

Below, we show the \( \text{SUM}(\text{lambda_cost}) \) 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 \( \text{SUM}(\text{lambda_cost}) \) for any dataset_id is \( \geq \) 300
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
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!

<table>
<thead>
<tr>
<th>arch</th>
<th>COUNT</th>
<th>HEATMAP(Log_Duration)</th>
<th>P99(duration_ms)</th>
<th>P50(duration_ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>amd64</td>
<td>262,988</td>
<td></td>
<td>1,168.09377</td>
<td>139.24663</td>
</tr>
<tr>
<td>arm64</td>
<td>161,394</td>
<td></td>
<td>2,677.62006</td>
<td>175.58275</td>
</tr>
</tbody>
</table>
LaunchDarkly 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.
Why so slow?

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

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

- Turned the flag off
Yes*, do this at home!

@lizthegrey
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
Observability Engineering

Explore preview chapters from our new book

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

AWS Lambda Instrumentation | Honeycomb