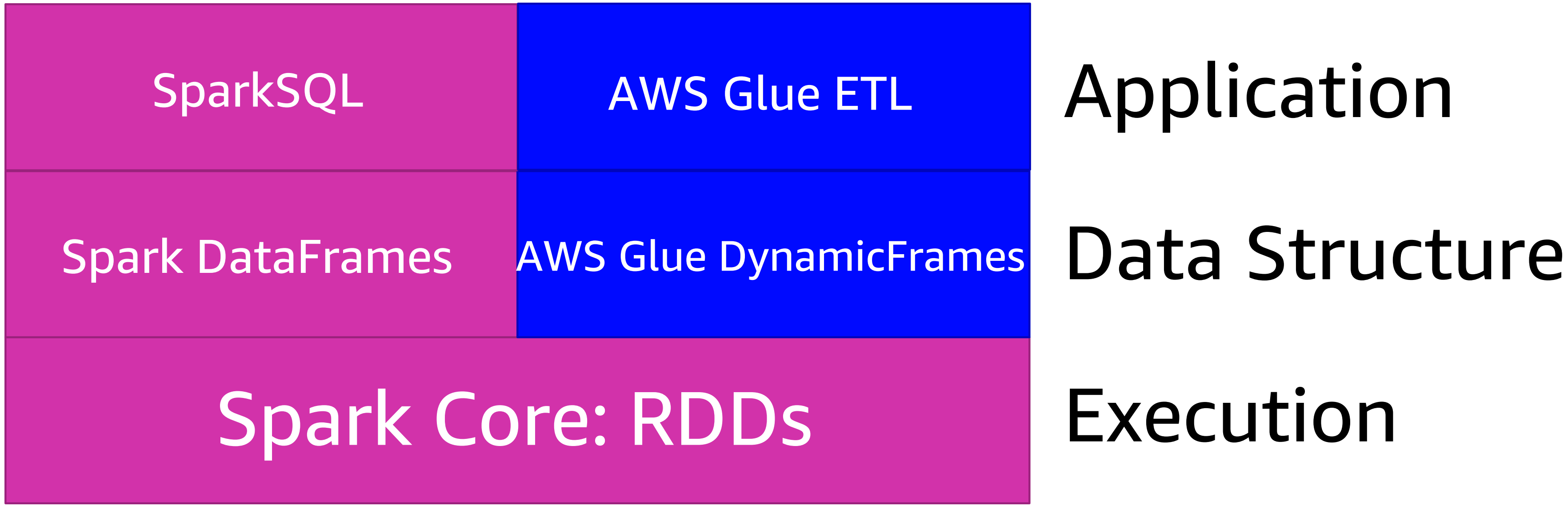
AWS re:Invent

A N T 3 3 2

METRICS-DRIVEN PERFORMANCE TUNING FOR AWS GLUE ETL JOBS

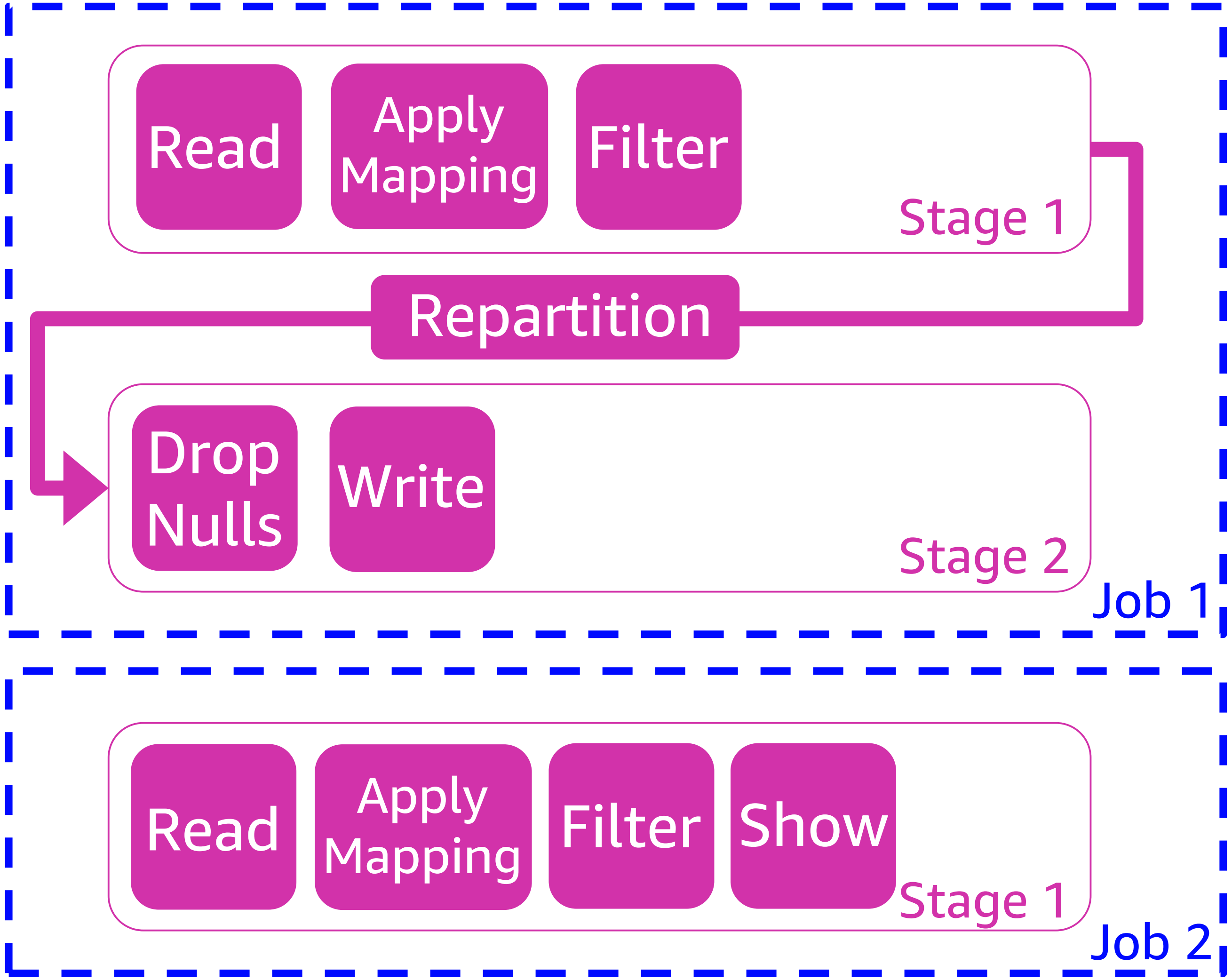
Benjamin Sowell
Principal Engineer
AWS Glue

Apache Spark and AWS Glue ETL



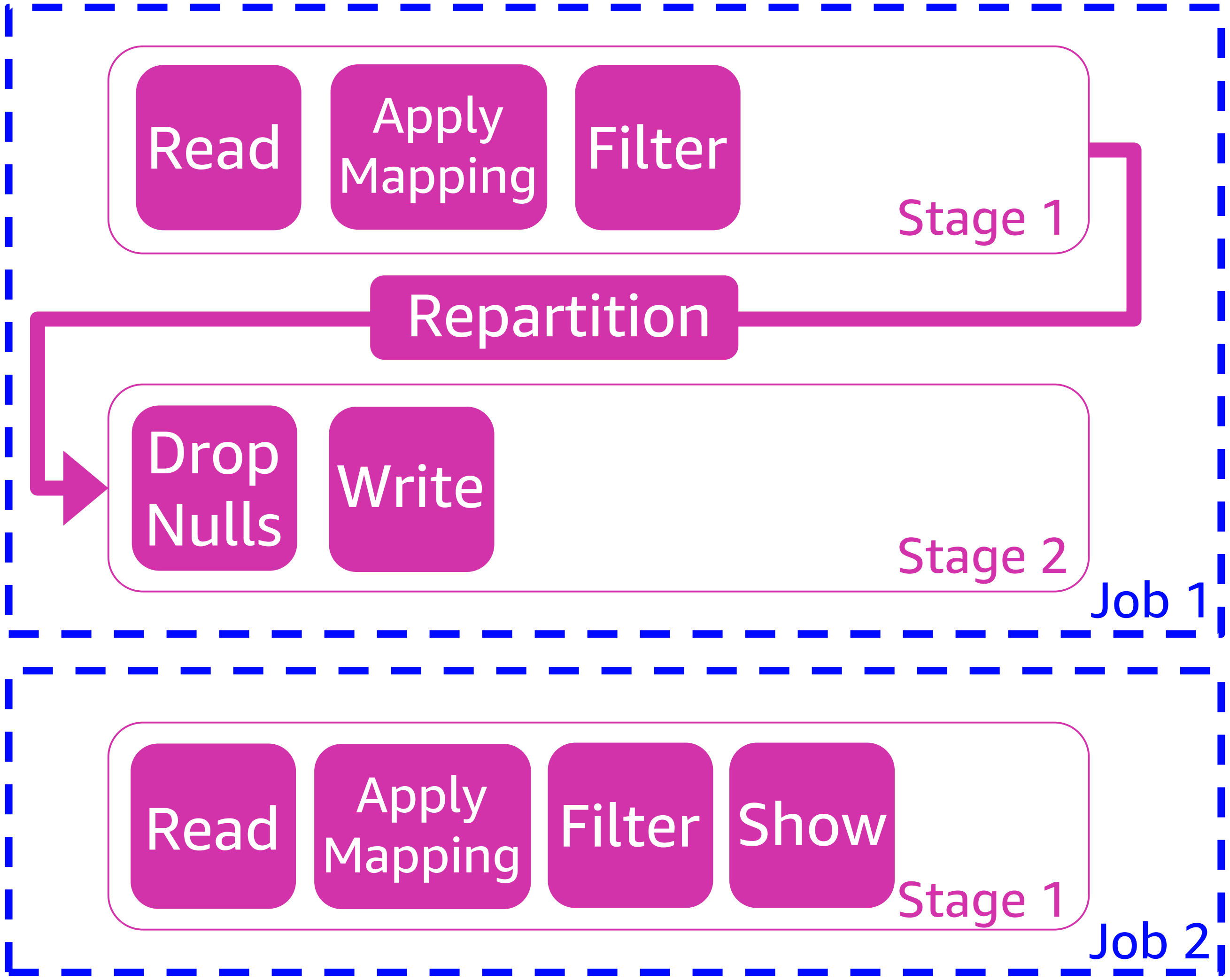
- Apache Spark is a distributed data processing engine with rich support for complex analytics.
- AWS Glue builds on the Apache Spark runtime to offer ETL specific functionality.

AWS Glue execution model: jobs and stages




```
df = glueContext.getSource(…)

applyMapping = df.applyMapping(…)

filter = applyMapping.filter(…)

repartition = filter.repartition(10)

dropNulls = repartition.dropNulls()

glueContext.getSink(…)\
    .writeDynamicFrame(dropNulls)


filter.show()
```

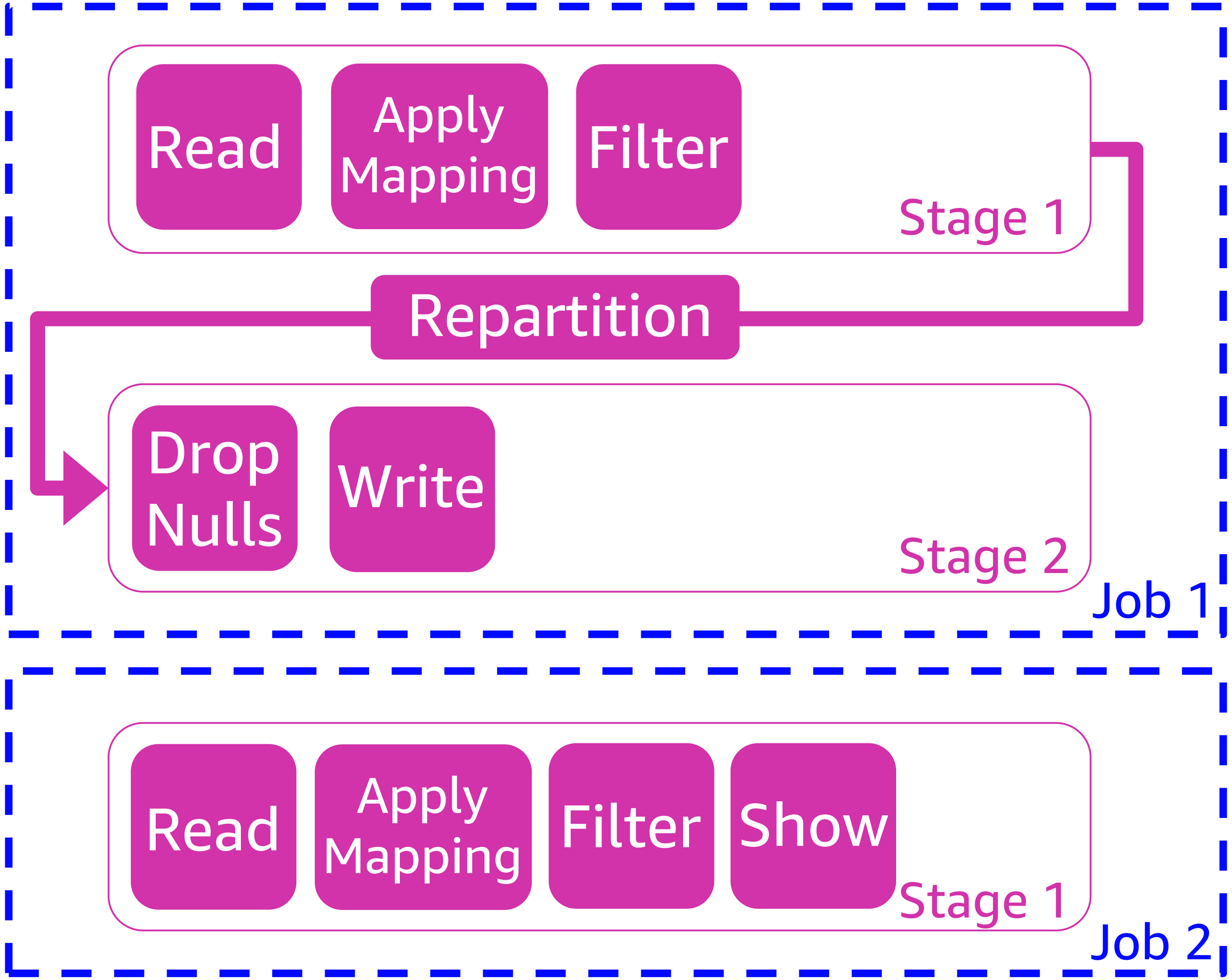
AWS Glue execution model: jobs and stages




```
df = glueContext.getSource(…)

applyMapping = df.applyMapping(…)

filter = applyMapping.filter(…)

repartition = filter.repartition(10)

dropNulls = repartition.dropNulls()

glueContext.getSink(…)\
    .writeDynamicFrame(dropNulls)

filter.show()
```

Actions

AWS Glue execution model: jobs and stages

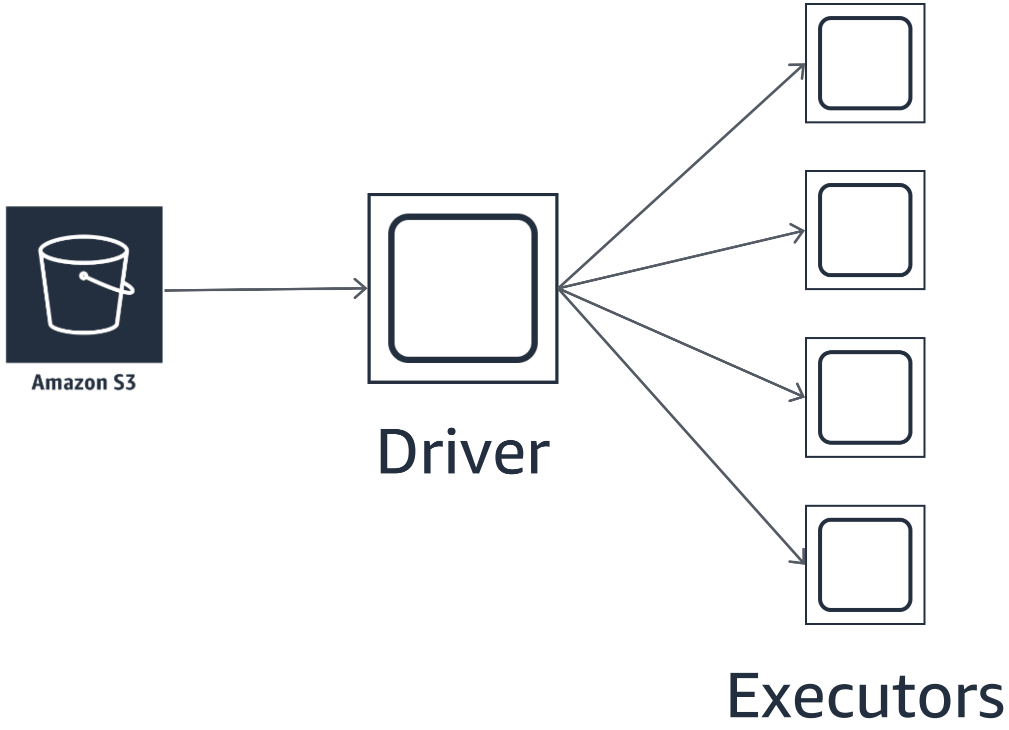



```
df = glueContext.getSource(…)

applyMapping = df.applyMapping(…)

filter = applyMapping.filter(…)

repartition = filter.repartition(10)

dropNulls = repartition.dropNulls()

glueContext.getSink(…)\
    .writeDynamicFrame(dropNulls)
```

```
filter.show()
```

Jobs

AWS Glue execution model: data partitions



- Apache Spark and AWS Glue are *data parallel*.
- Data is divided into *partitions* that are processed concurrently.
- 1 stage x 1 partition = 1 *task*

Overall throughput is limited by the number of partitions

AWS Glue performance: key questions

How is your application divided into jobs and stages?

How is your dataset partitioned?

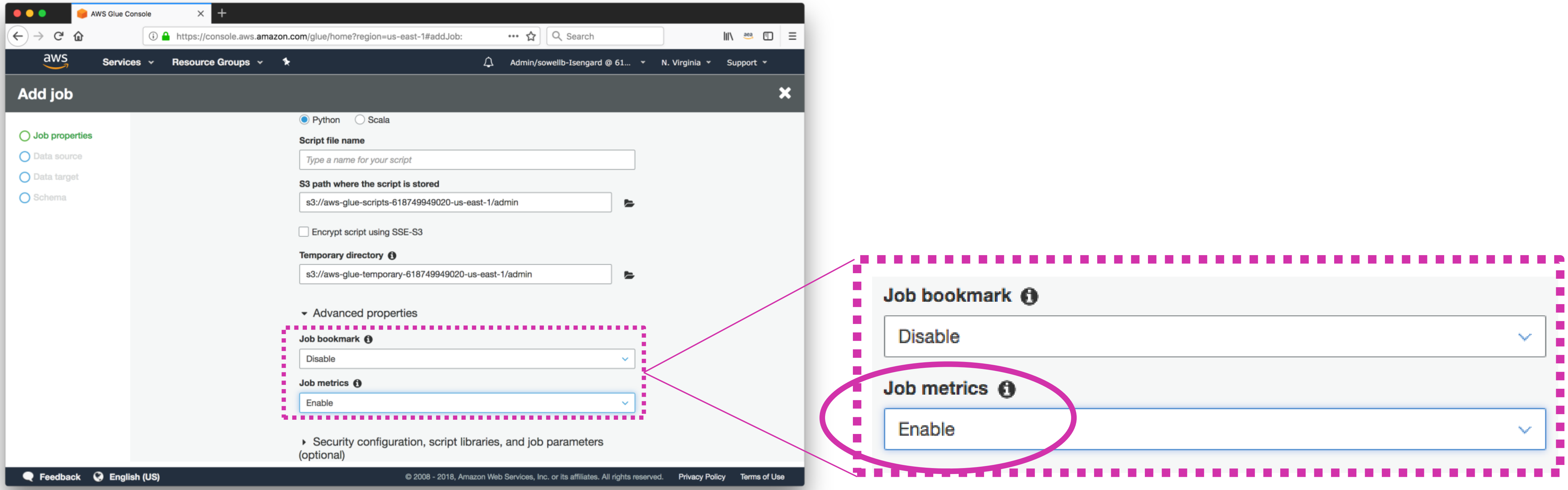
Example: Processing lots of small files

- One common problem is dealing with large numbers of small files.
  - Can lead to memory pressure and performance overhead.
- Let's look at a straightforward JSON to Parquet conversion job
  - 1.28 million JSON files in 640 partitions:

```
part1/                  part640/
     file1.json              file1.json
     …                       …
     file2000.json           file2000.json
```

- We will use AWS Glue *job metrics* to understand the performance.

Enabling job metrics

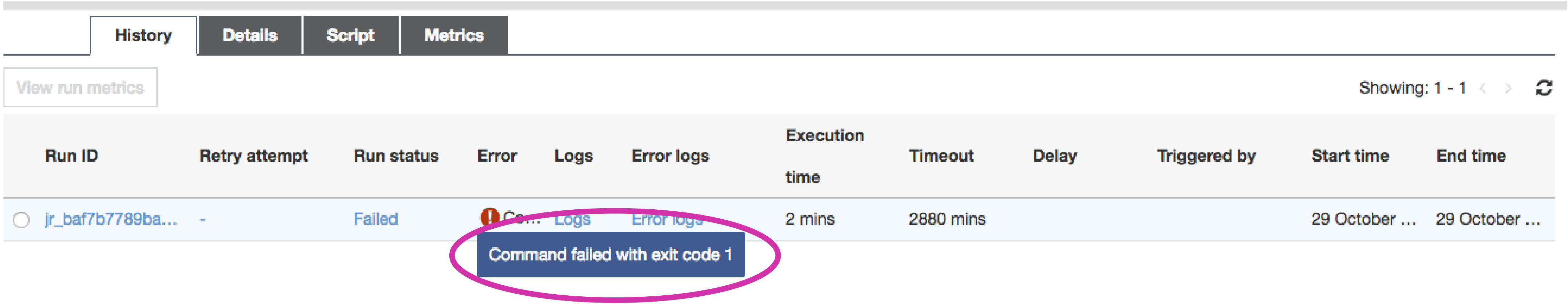


- Metrics can be enabled in the CLI/SDK by passing `--enable-metrics` as a job parameter key.

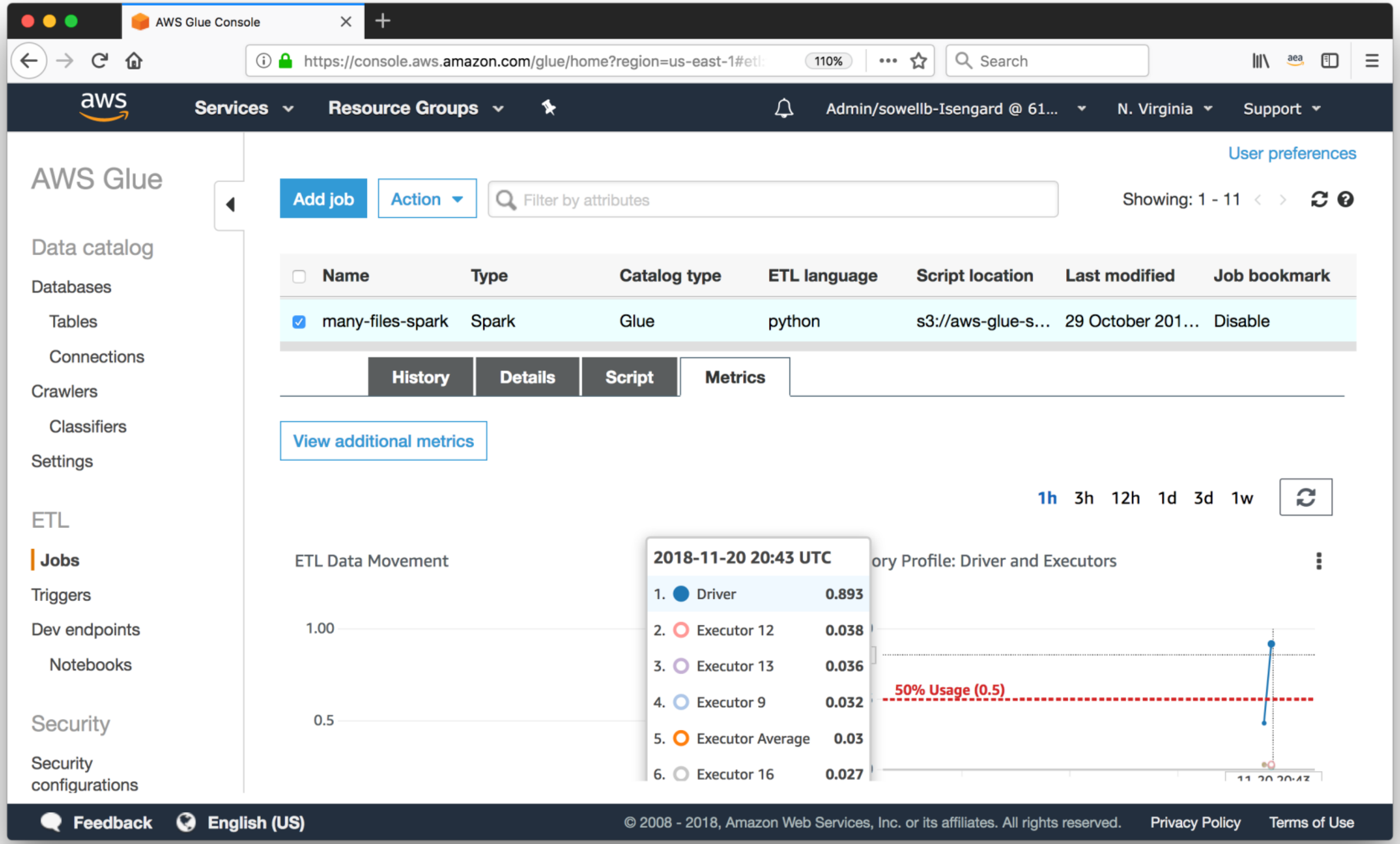
Example: Processing lots of small files

- **First try: use a standard SparkSQL job**

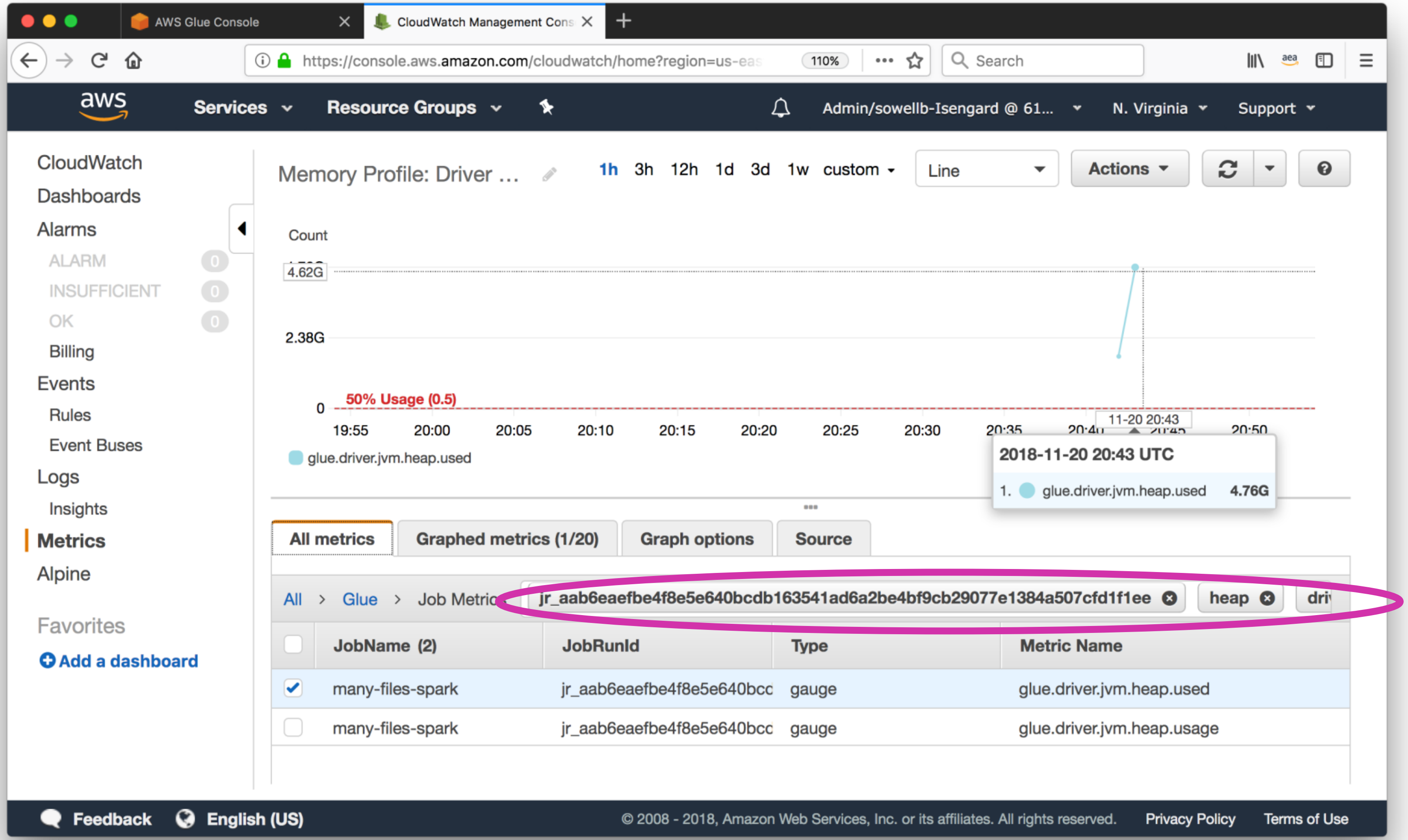
```
data = spark.read.format("json").load("<input_location>")
data.write.format("parquet").save("<output_location>")
```



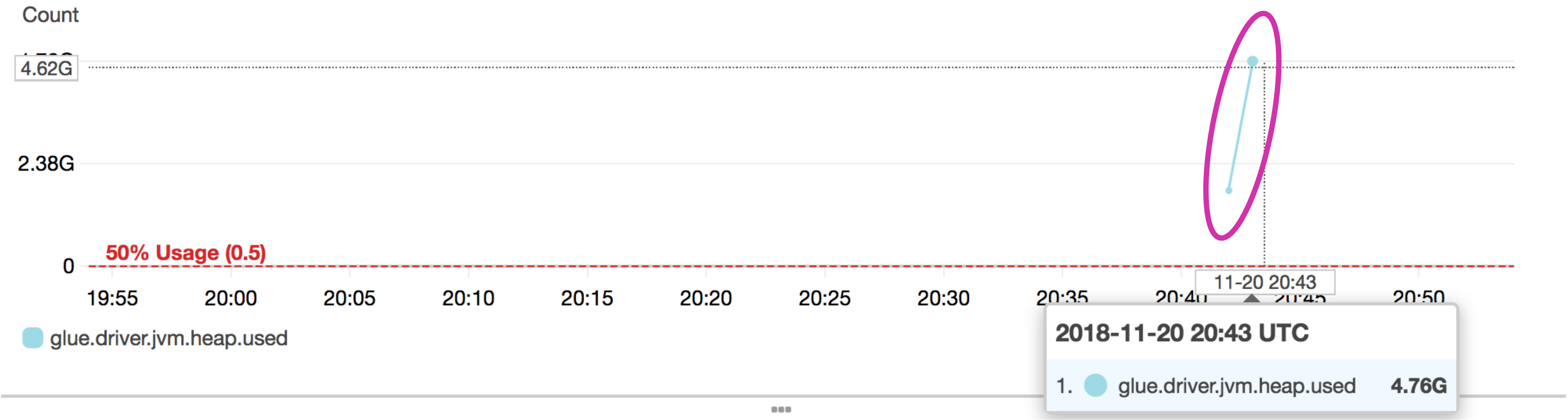
Example: Processing lots of small files



Example: Processing lots of small files



Example: Processing lots of small files



- Driver memory use is growing fast and approaching the 5g max.

Example: Processing lots of small files

- Case 2: Run using AWS Glue DynamicFrames.

```
df = glueContext.create_dynamic_frame_from_catalog("<database>","<table>")

glueContext.write_from_options(df,"s3",{"path":"<output_location>"},"parquet")
```

History | Details | Script | Metrics

View run metrics

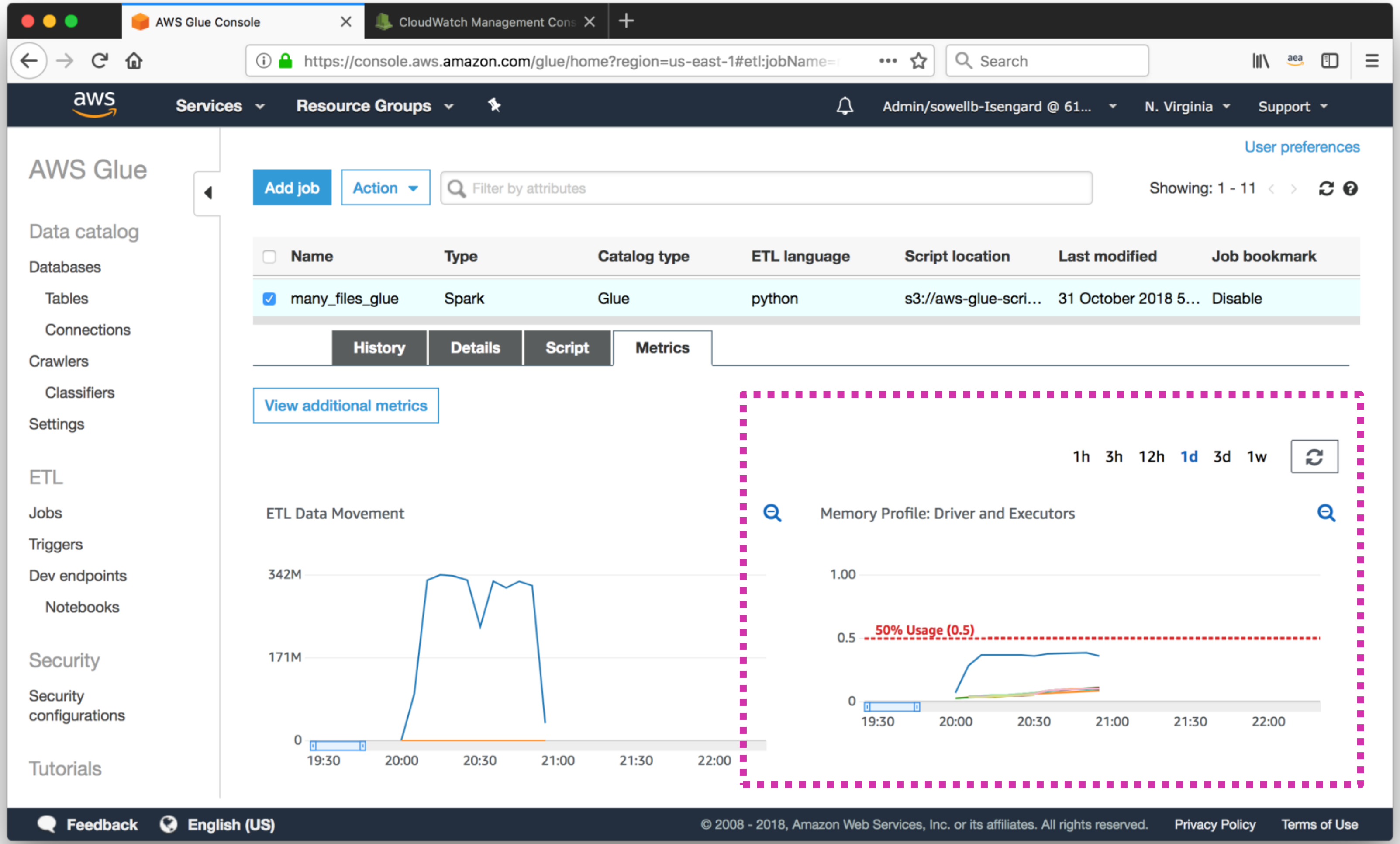
Showing: 1 - 1

	Run ID	Retry attempt	Run status	Error	Logs	Error logs	Execution time	Timeout	Delay	Triggered by	Start time	End time
	jr_671c479fa0...	-	Succeeded		Logs		59 mins	2880 mins			31 Octobe...	31 Octobe...

AWS re:Invent

aws

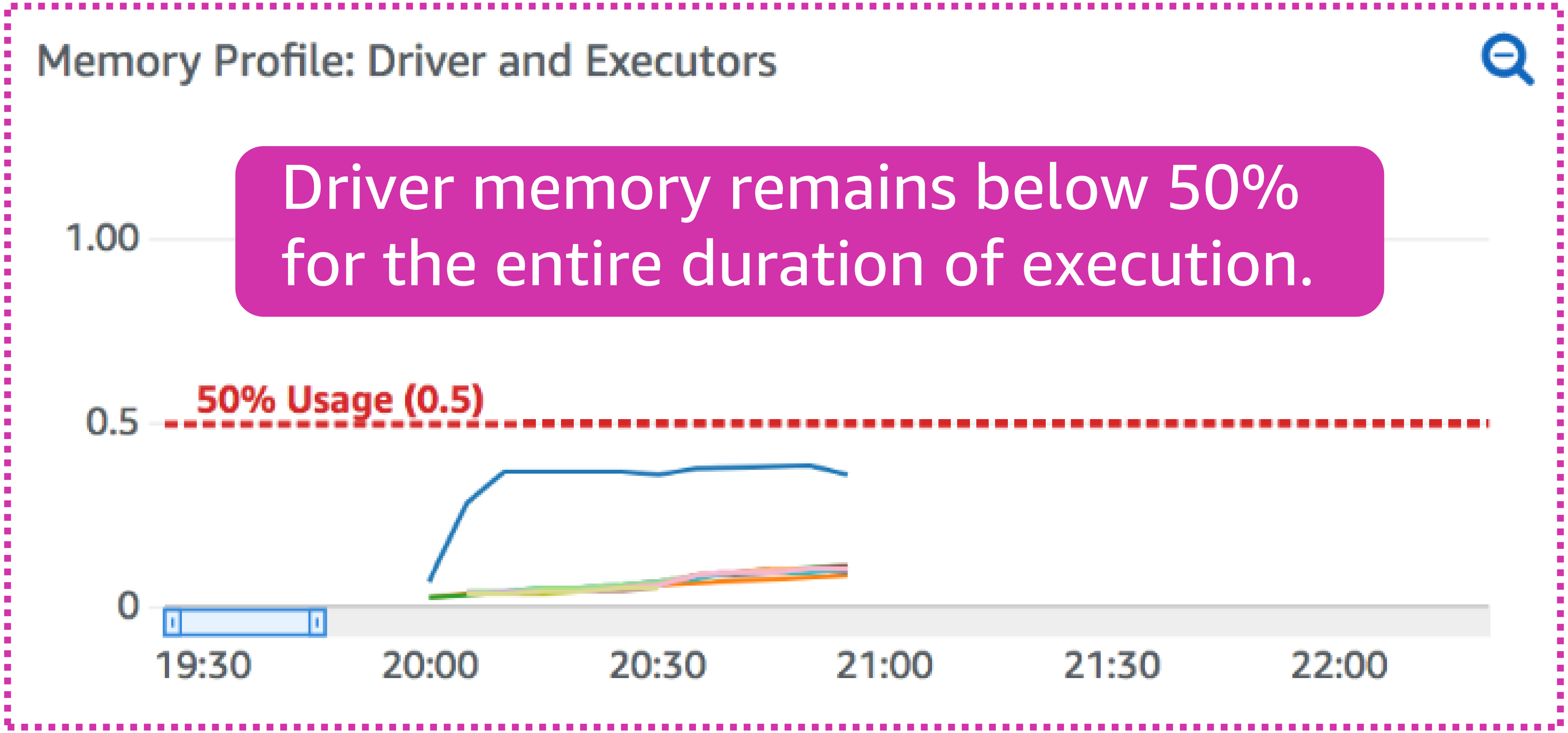
Example: Processing lots of small files



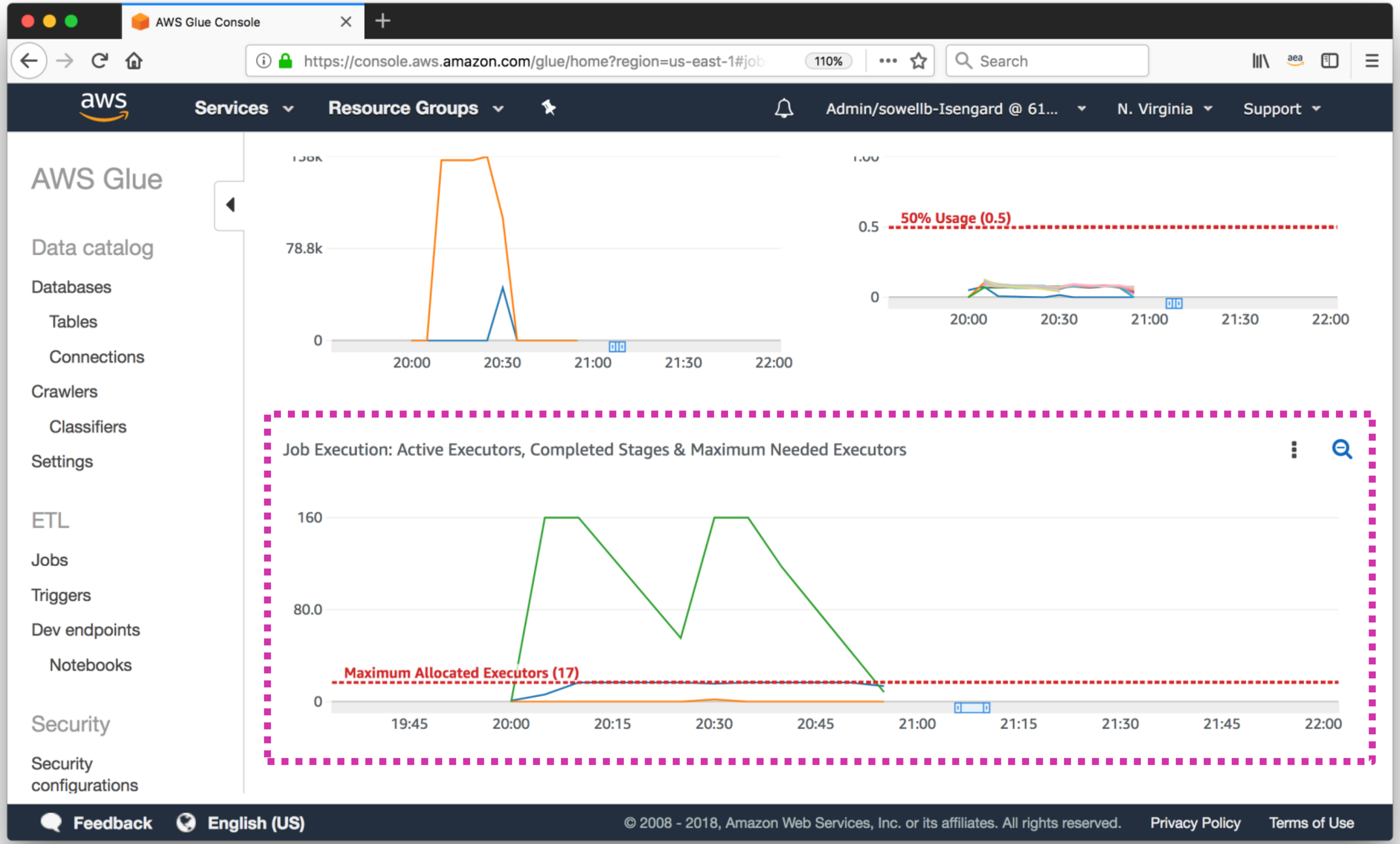
AWS re:Invent

aws

Example: Processing lots of small files



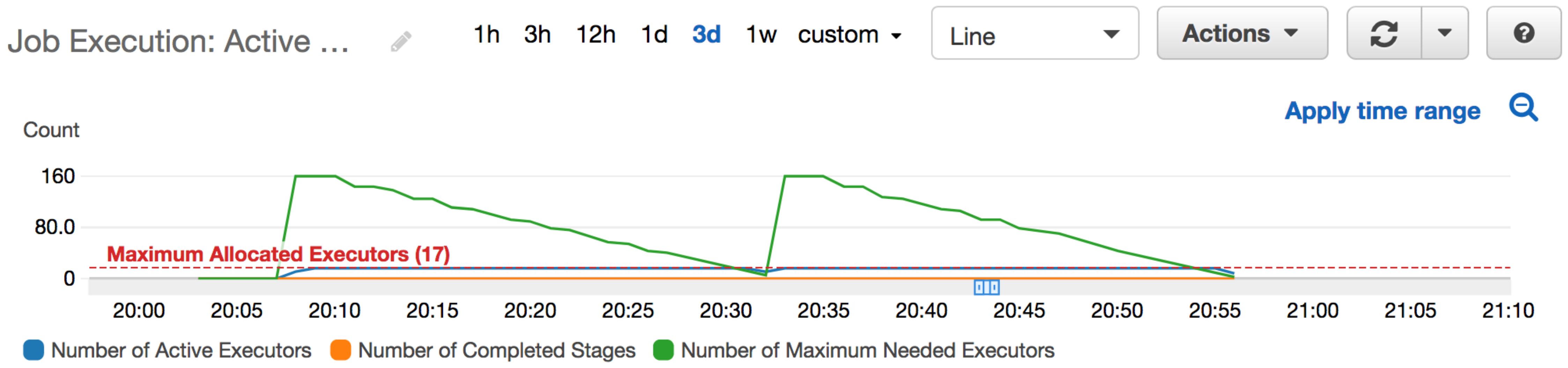
Example: Processing lots of small files



AWS re:Invent

aws

Example: Processing lots of small files



- 160 max executors: Each partition is assigned 1 task. 4 tasks can be processed on each executor, so there were 640 partitions.
  - Glue automatically grouped all of the files in each partition.

Options for grouping files

- groupFiles
  - Set to "inPartition" to enable grouping of files within a partition.
  - Set to "acrossPartition" to enable grouping of files from different partitions. The partition value will not be added to each record!
  - Grouping is automatically enabled when there are more than 50,000 files.
- groupSize
  - Target size of each group in bytes.
  - Default is based on the number of cores in the cluster.
- Let's try increasing the group size.

Example: Aggressively grouping files

- Execution is much slower, but hasn't crashed.

```
glueContext.create_dynamic_frame_from_catalog(
        database = "many_files_dataset",
        table_name="json_2k_million",
        additional_options={"groupSize": 1024 * 1024 * 1024},
                            "groupFiles": "acrossPartition")
```

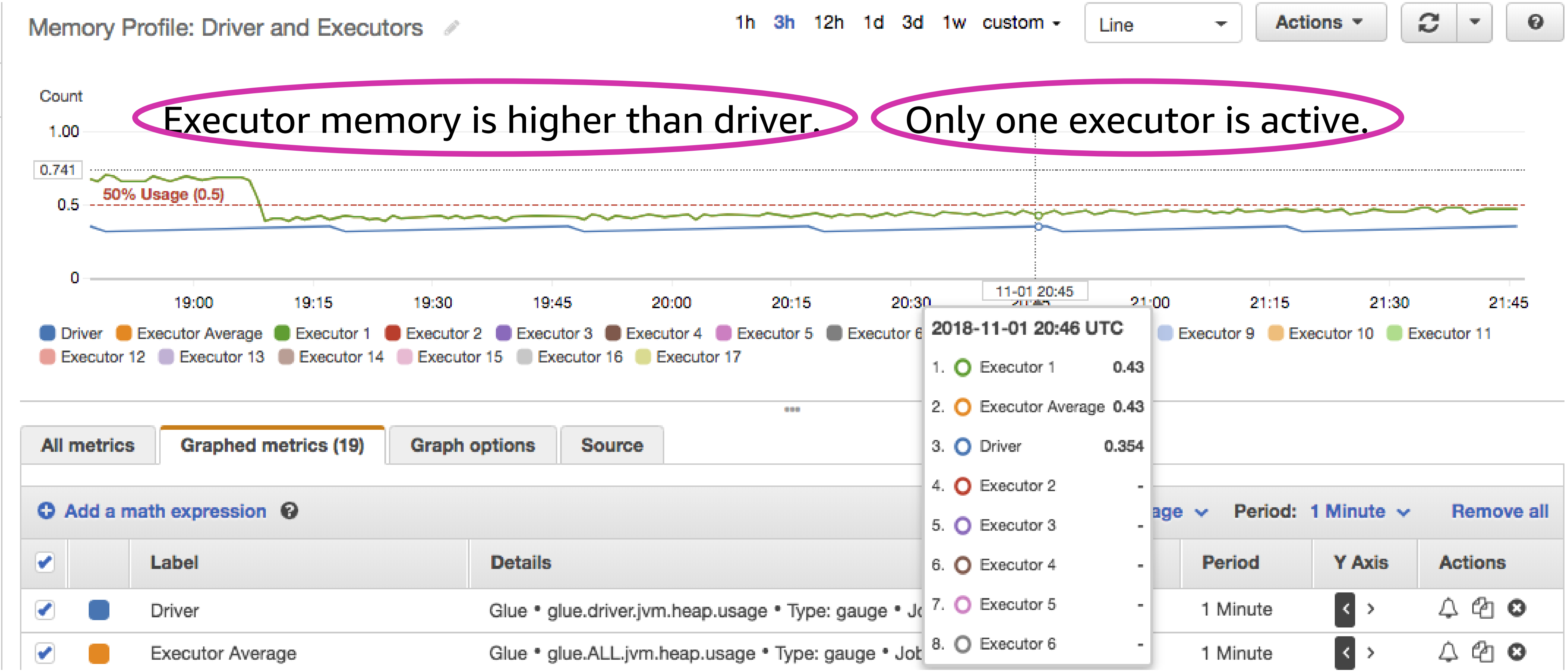
History | Details | Script | Metrics

View run metrics

Showing: 1 - 2

	Run ID	Retry attempt	Run status	Error	Logs	Error logs	Execution time	Timeout	Delay	Triggered by	Start time	End time
	jr_4b6e297a0f...	-	Running		Logs	Error logs	18 hrs, 53 mins	2880 mins			31 Octobe...	
	jr_671c479fa0...	-	Succeeded		Logs		59 mins	2880 mins			31 Octobe...	31 Octobe...

Example: Aggressively grouping files



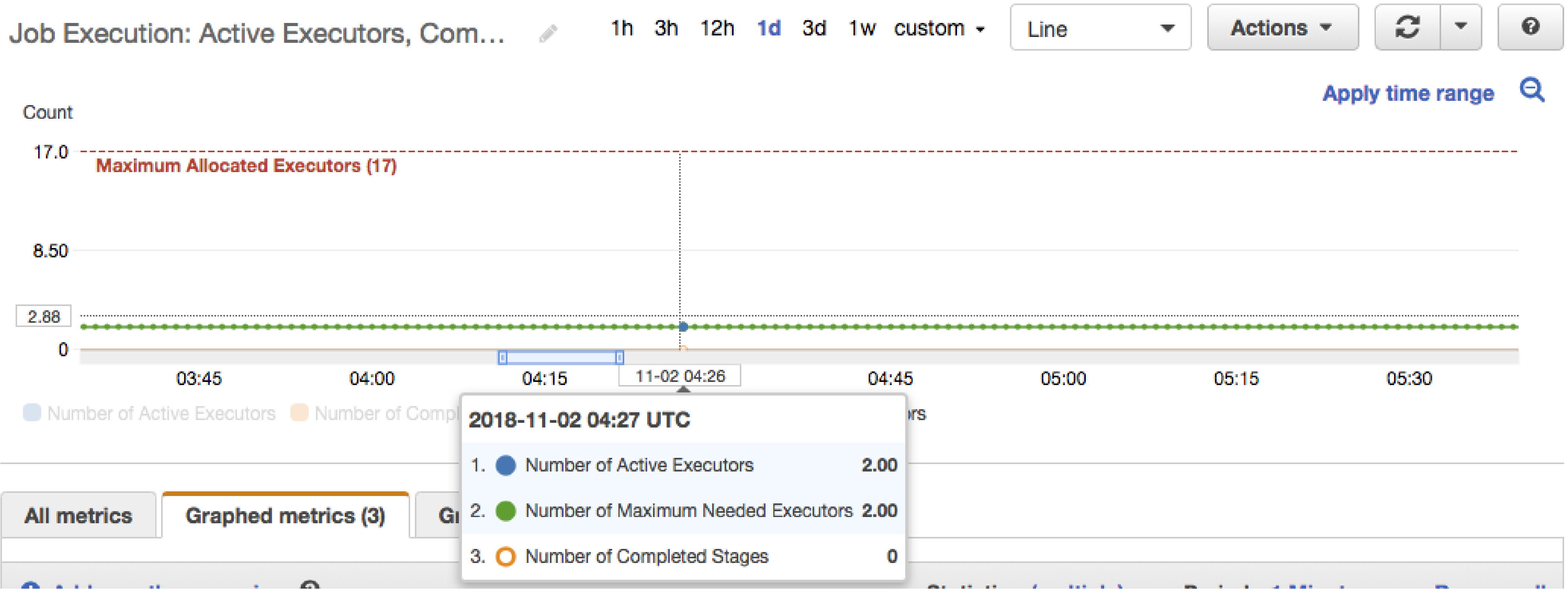
AWS re:Invent

Example: Processing a few large files

- Processing large text files can also cause problems if they cannot be split.
- How the data is compressed makes a big difference in performance.
  - Files that are *uncompressed* or *bzip2* compressed can be split and processed in parallel.
  - Files that are *gzip* compressed cannot be split.
- Let's see how this looks on a sample dataset of 5 large csv files.
- Each file is
  - 12.5 GB uncompressed
  - 1.6 GB gzip
  - 1.3 GB bzip2
- Script converts data to Parquet.

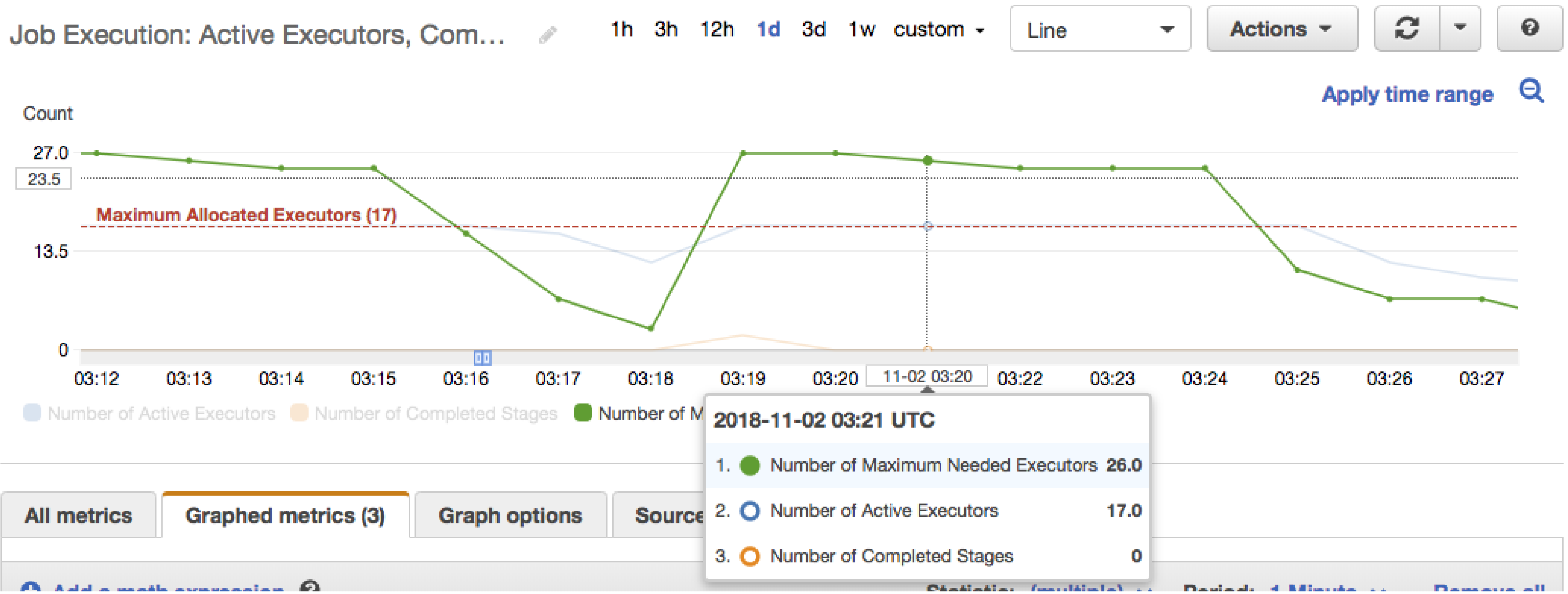
Example: Processing a few large gzip files

- We only have 5 partitions – one for each file.
- Job fails after 2 hours.



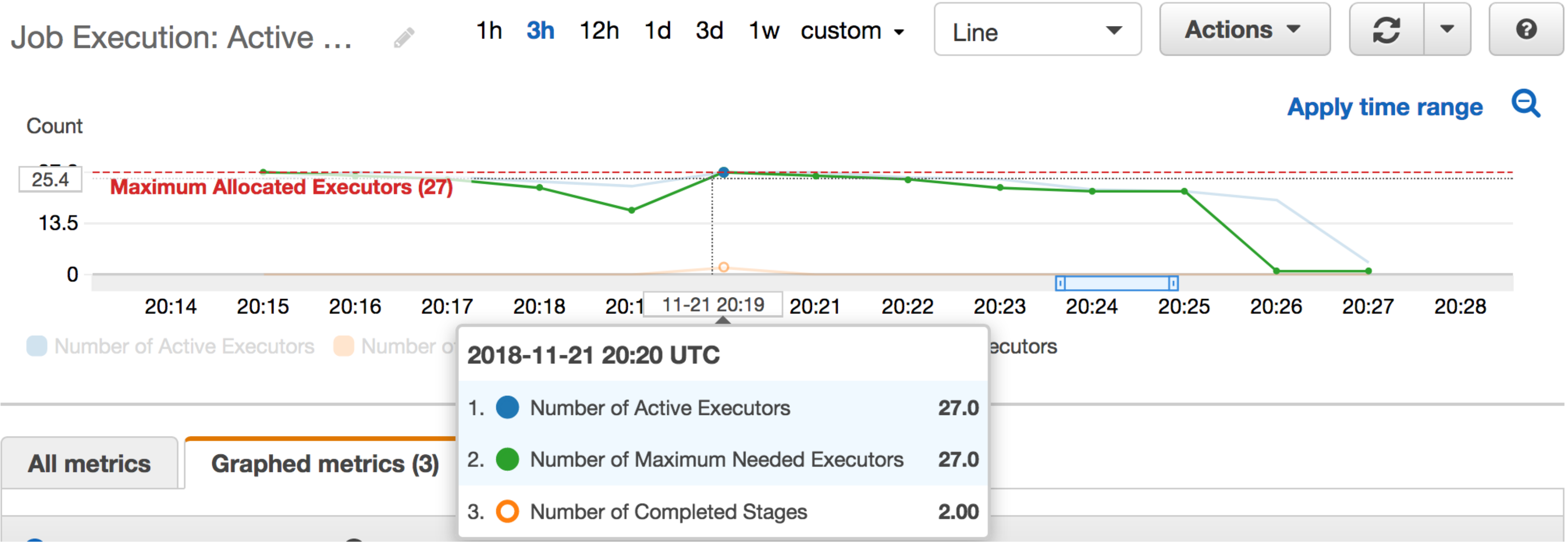
Example: Processing a few large bzip2 files

- Bzip2 files can be split into blocks, so we see up to 104 tasks.
- Job completes in 18 minutes.



Example: Processing a few large bzip2 files

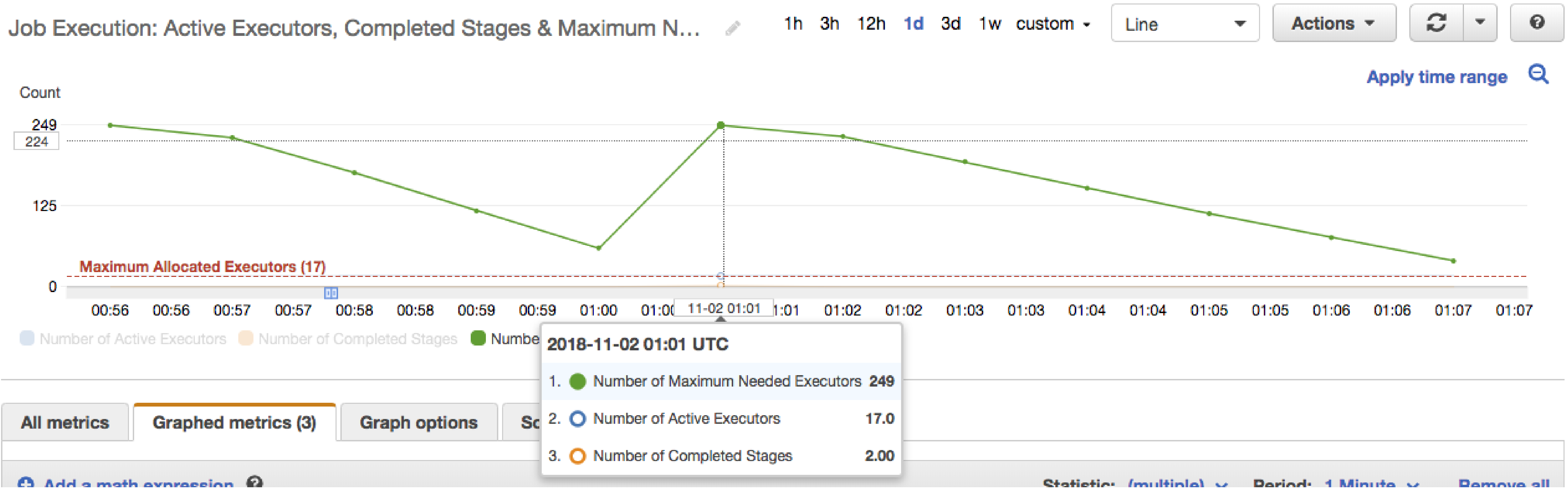
- With 15 DPU, the number of active executors closely tracks the maximum needed number of executors.



AWS re:Invent

Example: Processing a few large uncompressed files

- Uncompressed files can be split into lines, so we construct 64MB partitions.
- Job completes in 12 minutes.



AWS re:Invent

aws

Example: Processing a few large files

- If you have a choice of compression type, prefer bzip2.
- If you are using gzip, make sure you have enough files to fully utilize your resources.
- Bandwidth is rarely the bottleneck for AWS Glue jobs, so consider leaving files uncompressed.

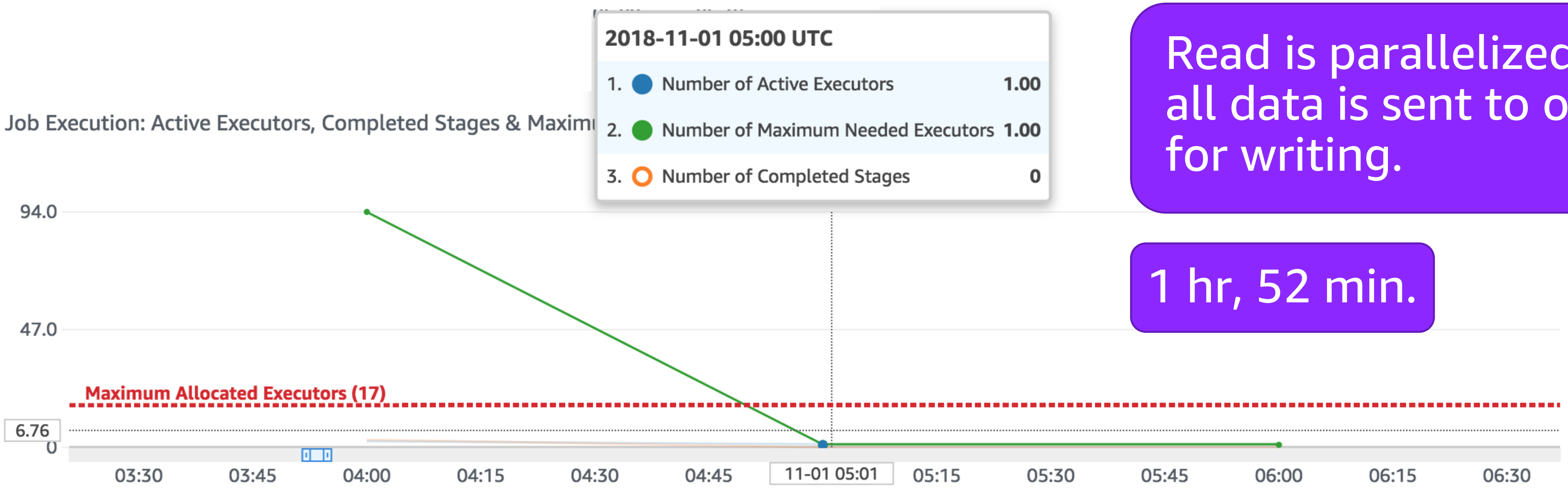
AWS re:Invent

aws

Example: Coalescing a dataset

- Common use case: reduce the number of files (32 GB compressed)

```
df = spark.read.format("json").load(...)
df2 = df.coalesce(1)
df2.write.format("json").save(...)
```



Read is parallelized and then all data is sent to one executor for writing.

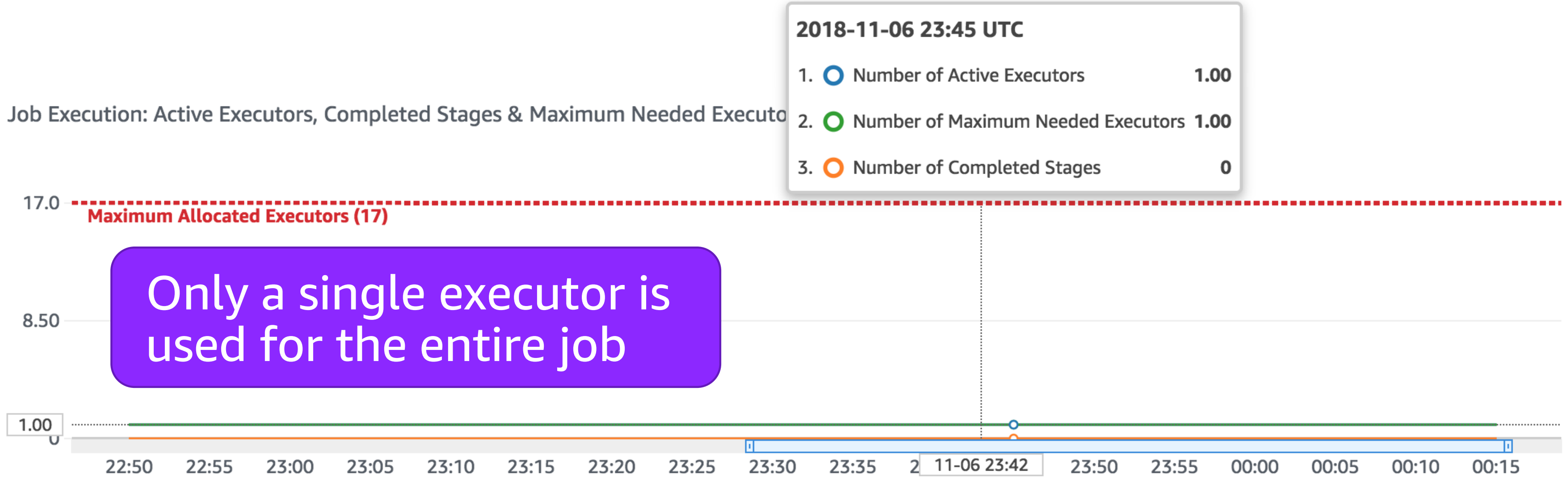
1 hr, 52 min.

AWS re:Invent

aws

Example: Coalescing with grouping

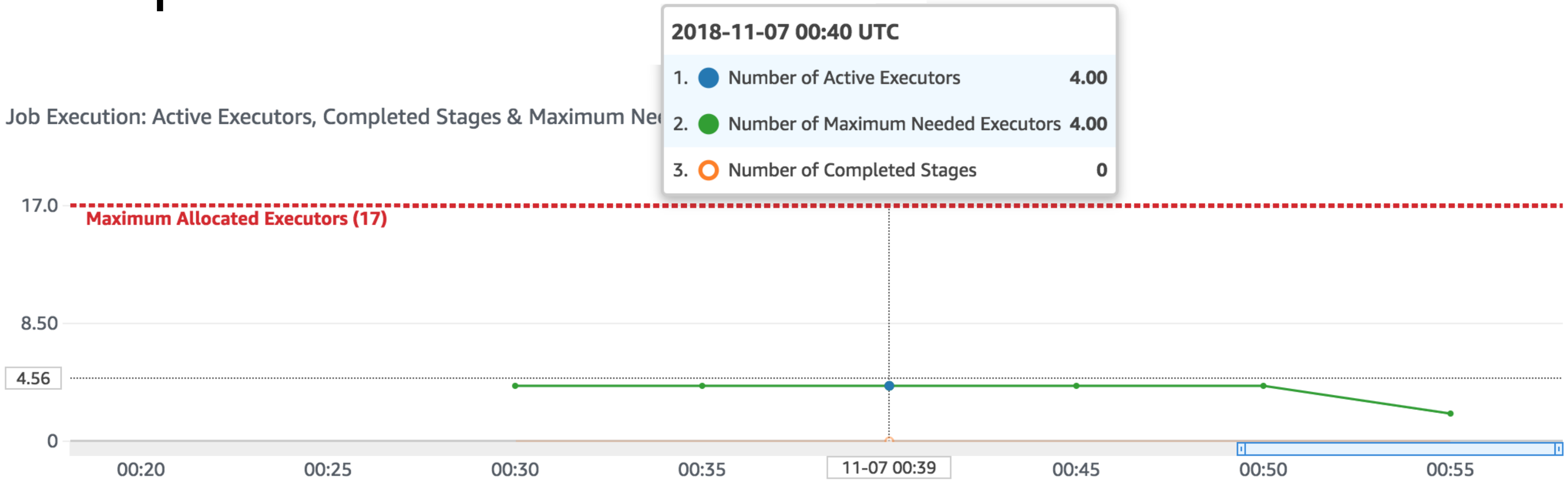
- You can control this further with grouping
- Group size 500 GB.



1 hr, 27 min.

Example: Coalescing with grouping

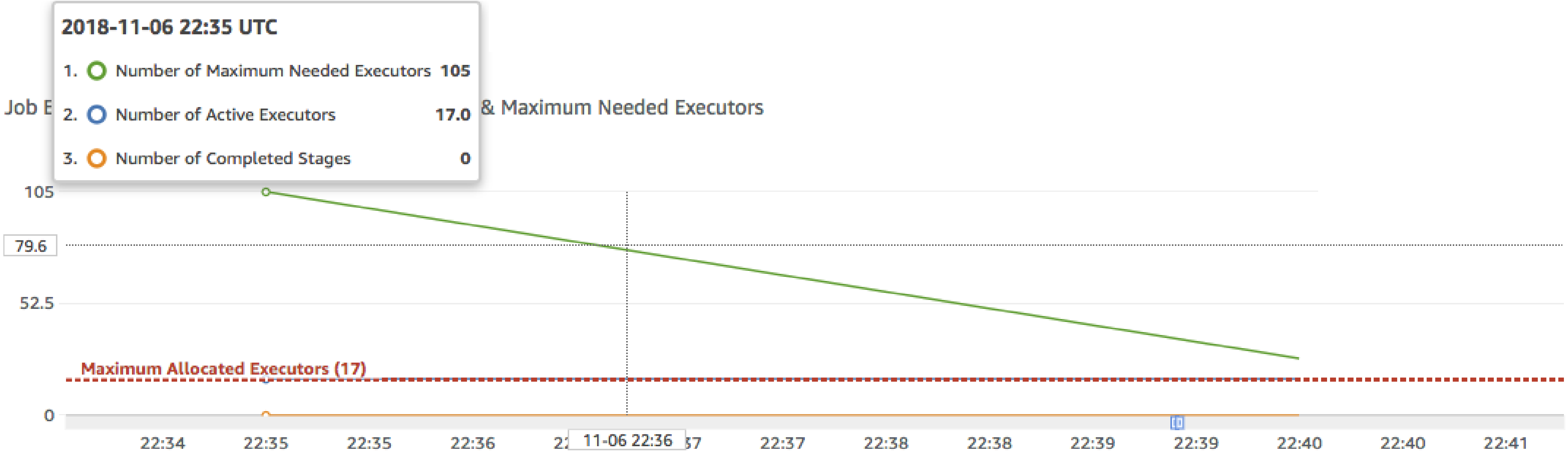
- You can control this further with grouping
- Group size 2 GB.



Four executors were used to process 16 partitions of ~2 GB each.

26 min.

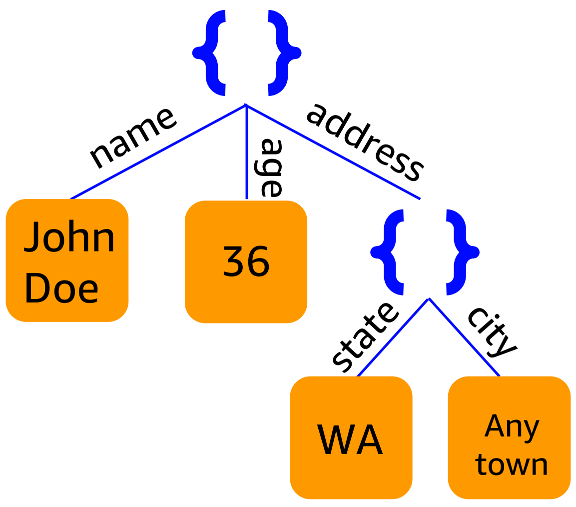
Example: Baseline performance without coalescing



- Job is fully parallelized and completes in 6 minutes.
- Output data files are around 500 MB.

DynamicFrames and schema computation

- DynamicFrames are Spark RDDs of *self-describing* records.
- An overall schema is not required for basic ETL operations.
  - I can drop the field "age" without looking at other records.
- Some operations do require a complete schema.
  - This can force an extra job in your application.



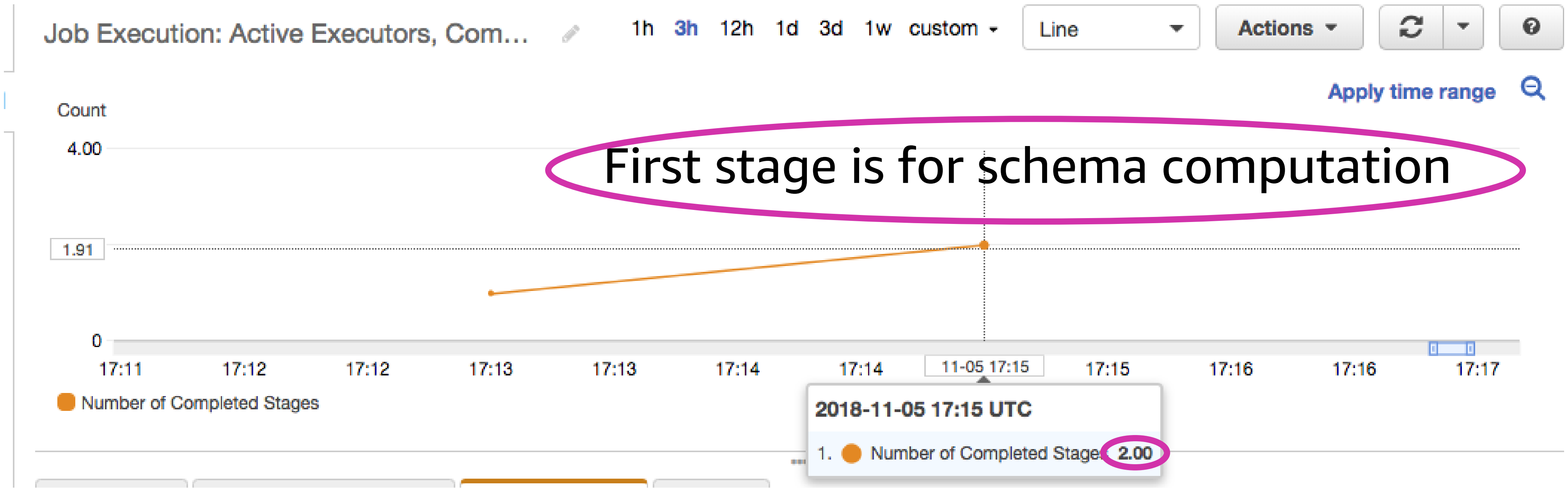
Transforms that require a schema:

- DropNullFields
- Relationalize
- ResolveChoice without specifying columns

…

DynamicFrame Schema Example: DropNullFields

```
df = glueContext.create_dynamic_frame_from_catalog(…)
df2 = DropNullFields.apply(df)
glueContext.write_dynamic_frame_from_options(df2, …)
```



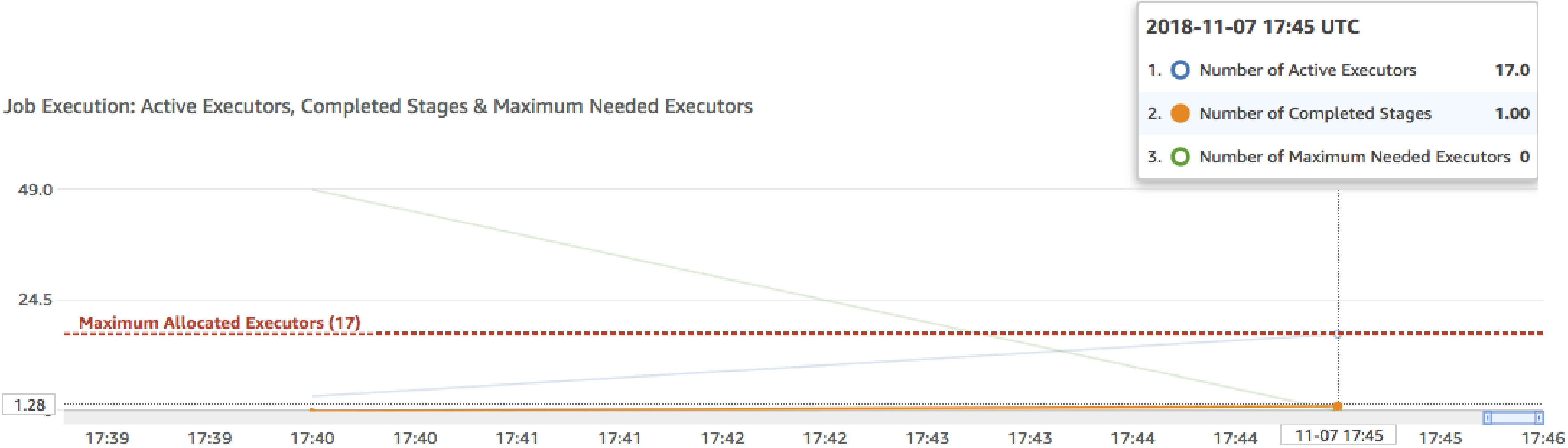
DynamicFrame Schema Example: DropNullFields

```
df = glueContext.create_dynamic_frame_from_catalog(…)
df2 = df.applyMapping([
    ('col1', 'col1', 'string'),
    ('col2', 'col2', 'string'),
    ('col3', 'col3', 'string'),
    ('col4', 'col4', 'string'),
    ('col5', 'col5', 'boolean')
])
df3 = DropNullFields.apply(df2)
glueContext.write_dynamic_frame_from_options(df3, …)
```

ApplyMapping sets the schema without an additional pass.

DynamicFrame Schema Example: DropNullFields

- There is only one stage in the application.

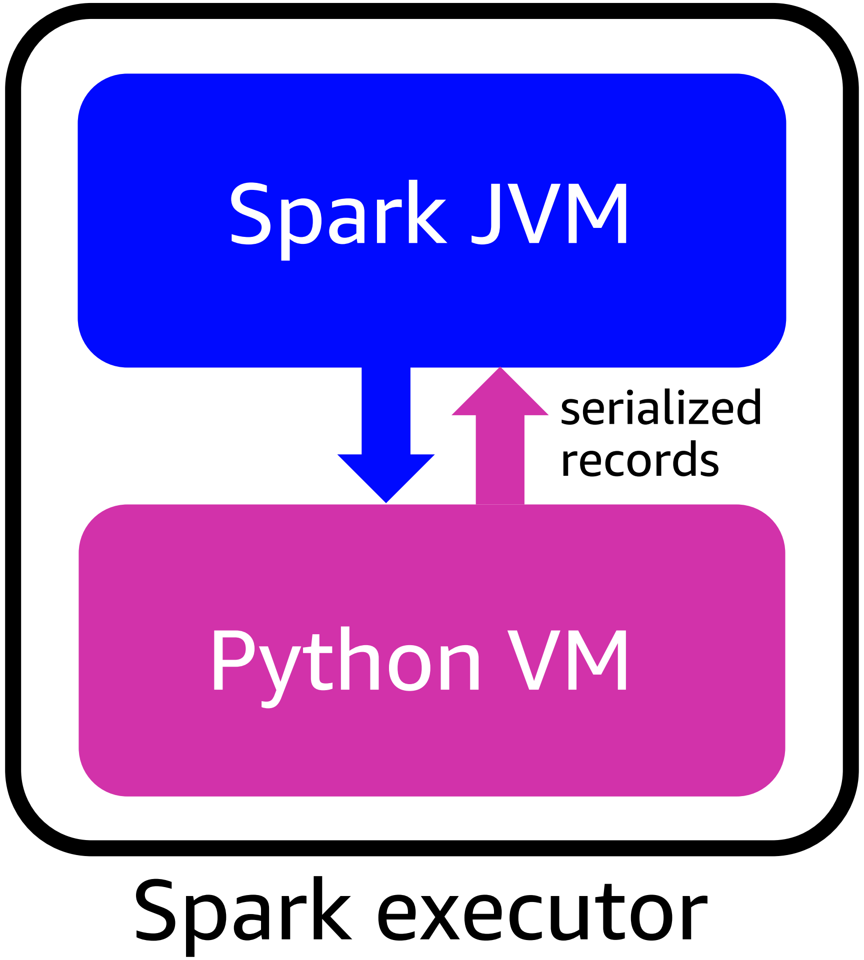


re:Invent

Optimizing DynamicFrames

- Use ApplyMapping where appropriate to set a schema.
- Be thoughtful about where to add transformations like DropNullFields.
  - The default generated scripts are designed to be safe, but you may be able to optimize if you know your data.
- Try your old scripts!
  - We've increased data conversion speed by 4x in the past year.

Python performance in AWS Glue



- Using map and filter in Python is expensive for large data sets.
  - All data is serialized and sent between the JVM and Python.
- Alternatives
  - Use AWS Glue Scala SDK.
  - Convert to DataFrame and use Spark SQL expressions.

AWS re:Invent

aws

AWS Lake Formation

Build a secure data lake in days

Register existing data or load new data using blueprints. Data stored in Amazon S3.

Secure data access across multiple services using single set of permissions.

No additional charge. Only pay for the underlying services used.

Quickly build data lakes

Move, store, catalog, and clean your data faster. Use ML transforms to de-duplicate data and find matching records.

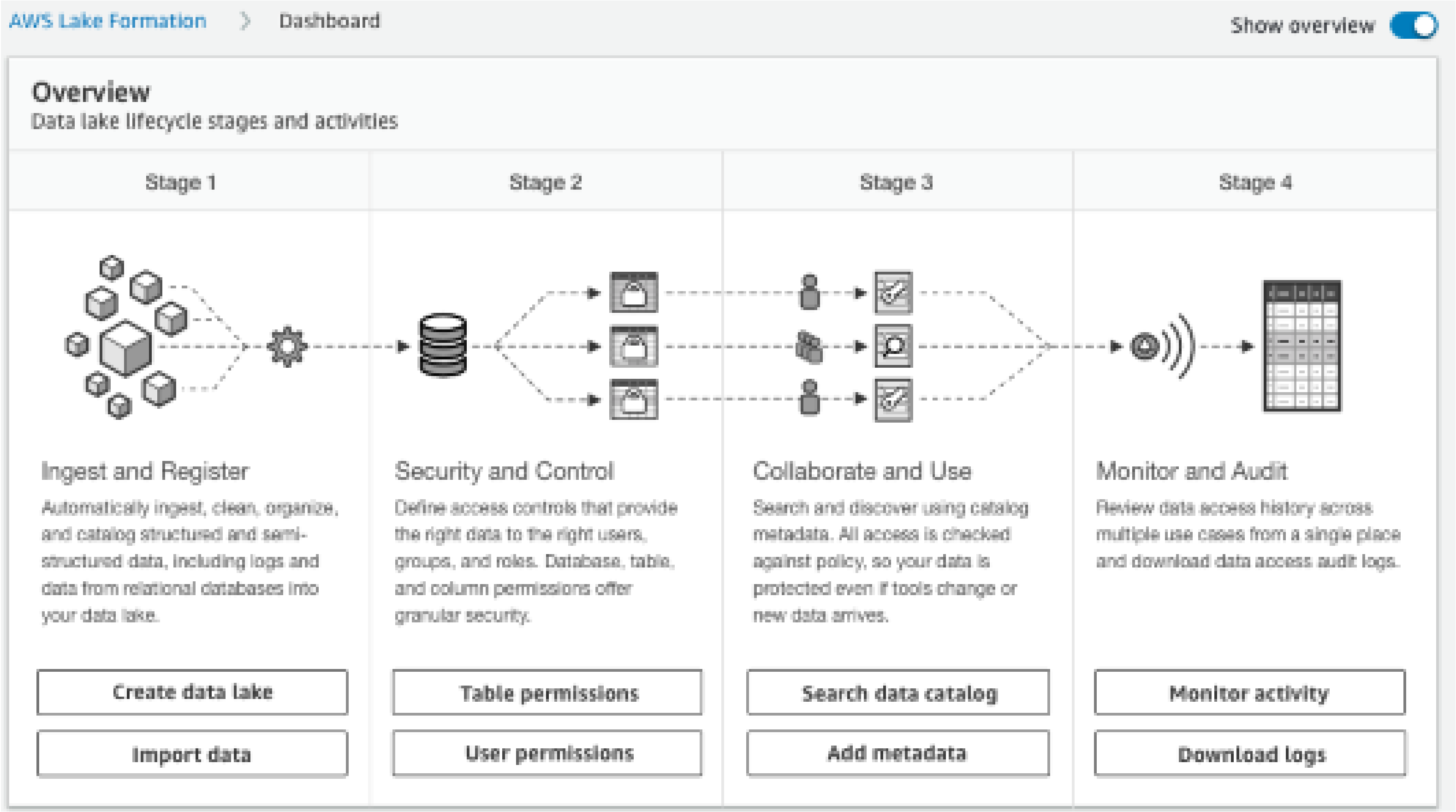
Easily secure access

Centrally define table and column-level data access and enforce it across Amazon EMR, Amazon Athena, Amazon Redshift Spectrum, Amazon SageMaker, and Amazon QuickSight

Share and collaborate

Use data catalog in Lake Formation to search and find relevant data sets and share them across multiple users and accounts

How it works



AWS Lake Formation
Dashboard
Show overview
Overview
Data lake lifecycle stages and activities
Stage 1
Stage 2
Stage 3
Stage 4
Ingest and Register
Automatically ingest, clean, organize, and catalog structured and semi-structured data, including logs and data from relational databases into your data lake.
Security and Control
Define access controls that provide the right data to the right users, groups, and roles. Database, table, and column permissions offer granular security.
Collaborate and Use
Search and discover using catalog metadata. All access is checked against policy, so your data is protected even if tools change or new data arrives.
Monitor and Audit
Review data access history across multiple use cases from a single place and download data access audit logs.
Create data lake
Import data
Table permissions
User permissions
Search data catalog
Add metadata
Monitor activity
Download logs

Thank you!

Benjamin Sowell

AWS re:Invent

Please complete the session survey in the mobile app.