Introducing AWS Glue Auto Scaling: Automatically resize serverless computing resources for lower cost with optimized Apache Spark

by Noritaka Sekiyama, Rajendra Gujja, Bo Li, and Mohit Saxena | on 21 APR 2022 | in Analytics, AWS Big Data, AWS Glue, Intermediate (200), Serverless | Permalink | Comments | Share

October 2024: This post has been updated along with Interactive Sessions support for AWS Glue Auto scaling.
June 2023: This post was reviewed and updated for accuracy.

Data created in the cloud is growing fast in recent days, so scalability is a key factor in distributed data processing. Many customers benefit from the scalability of the AWS Glue serverless Spark runtime. Today, we're pleased to announce the release of AWS Glue Auto Scaling, which helps you scale your AWS Glue Spark jobs automatically based on the requirements calculated dynamically during the job run, and accelerate job runs at lower cost without detailed capacity planning.

Before AWS Glue Auto Scaling, you had to predict workload patterns in advance. For example, in cases when you don't have expertise in Apache Spark, when it's the first time you're processing the target data, or when the volume or variety of the data is significantly changing, it's not so easy to predict the workload and plan the capacity for your AWS Glue jobs. Under-provisioning is error-prone and can lead to either missed SLA or unpredictable performance. On the other hand, over-provisioning can cause underutilization of resources and cost overruns. Therefore, it was a common best practice to experiment with your data, monitor the metrics, and adjust the number of AWS Glue workers before you deployed your Spark applications to production.

With AWS Glue Auto Scaling, you no longer need to plan AWS Glue Spark cluster capacity in advance. You can just set the maximum number of workers and run your jobs. AWS Glue monitors the Spark application execution, and allocates more worker nodes to the cluster in near-real time after Spark requests more executors based on your workload requirements. When there are idle executors that don't have intermediate shuffle data, AWS Glue Auto Scaling removes the executors to save the cost.

AWS Glue Auto Scaling is available with the optimized Spark runtime on AWS Glue version 3.0, and you can start using it today. This post describes possible use cases and how it works.

Use cases and benefits for AWS Glue Auto Scaling

Traditionally, AWS Glue launches a serverless Spark cluster of a fixed size. The computing resources are held for the whole job run until it is completed. With the new AWS Glue Auto Scaling feature, after you enable it for your AWS Glue Spark jobs, AWS Glue dynamically allocates compute resource considering the given maximum number of workers. It also supports dynamic scale-out and scale-in of the AWS Glue Spark cluster size over the course of job. As more executors are requested by Spark, more AWS Glue workers are added to the cluster. When the executor has been idle without active computation tasks for a period of time and associated shuffle dependencies, the executor and corresponding worker are removed.

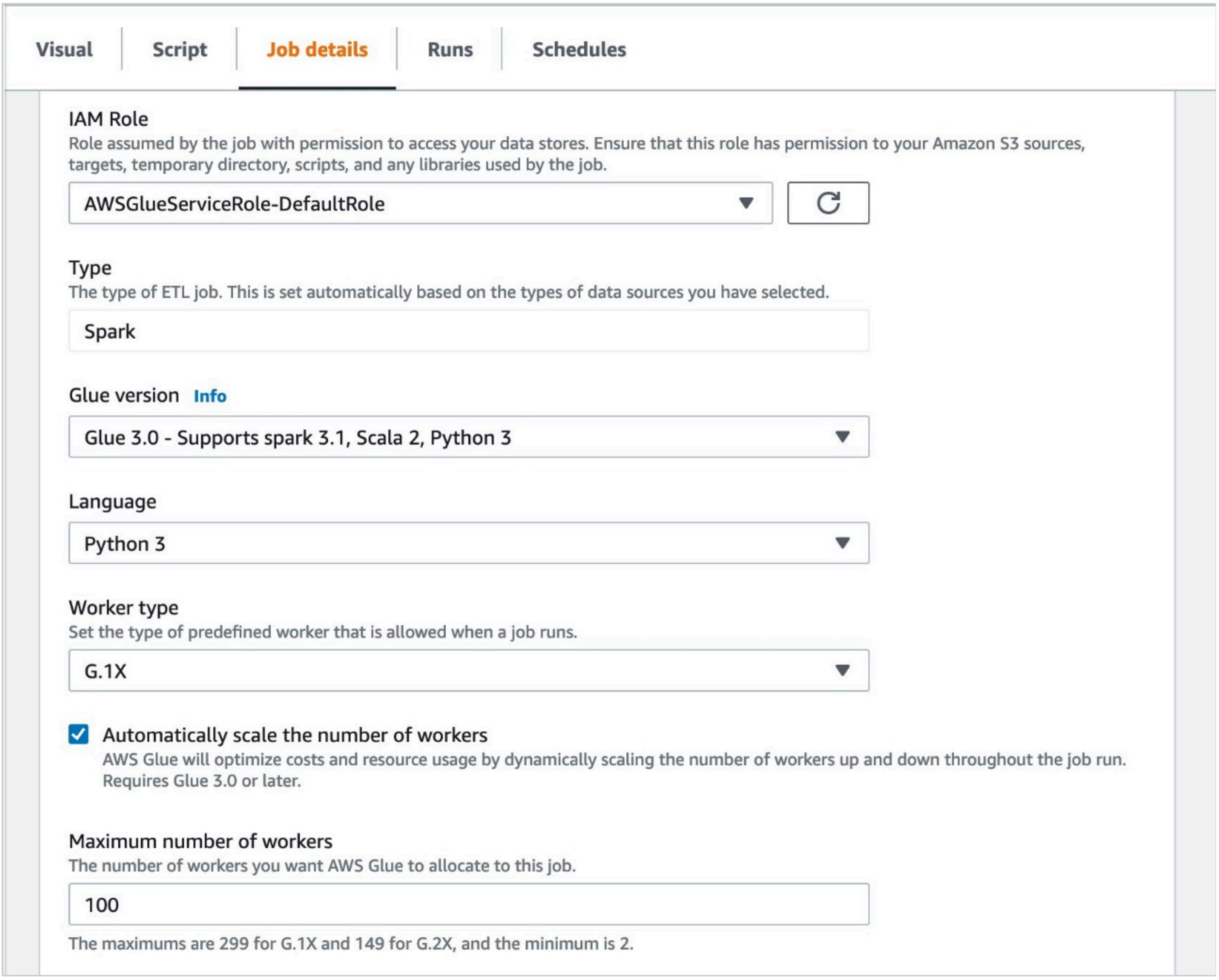
AWS Glue Auto Scaling makes it easy to run your data processing in the following typical use cases:

- Batch jobs to process unpredictable amounts of data
- Jobs containing driver-heavy workloads (for example, processing many small files)
- Jobs containing multiple stages with uneven compute demands or due to data skews (for example, reading from a data store, repartitioning it to have more parallelism, and then processing further analytic workloads)
- Jobs to write large amounts of data into data warehouses such as Amazon Redshift or read and write from databases

Configure AWS Glue Auto Scaling

AWS Glue Auto Scaling is available with the optimized Spark runtime on Glue version 3.0. To enable Auto Scaling on the AWS Glue Studio console, complete the following steps:

1. Open AWS Glue Studio.
2. Choose **Jobs**.
3. Choose your job.
4. Choose the **Job details** tab.
5. For **Glue version**, choose **Glue 3.0 – Supports spark 3.1, Scala 2, Python**. Alternatively you can choose Glue 4.0.
6. Select **Automatically scale the number of workers**.
7. For **Maximum number of workers**, enter the maximum workers that can be vended to the job run.
8. Choose **Save**.


Visual | Script | Job details | Runs | Schedules

IAM Role
Role assumed by the job with permission to access your data stores. Ensure that this role has permission to your Amazon S3 sources, targets, temporary directory, scripts, and any libraries used by the job.
AWSGlueServiceRole-DefaultRole

Type
The type of ETL job. This is set automatically based on the types of data sources you have selected.
Spark

Glue version Info
Glue 3.0 - Supports spark 3.1, Scala 2, Python 3

Language
Python 3

Worker type
Set the type of predefined worker that is allowed when a job runs.
G.1X

☑ Automatically scale the number of workers
AWS Glue will optimize costs and resource usage by dynamically scaling the number of workers up and down throughout the job run. Requires Glue 3.0 or later.

Maximum number of workers
The number of workers you want AWS Glue to allocate to this job.
100
The maximums are 299 for G.1X and 149 for G.2X, and the minimum is 2.

To enable Auto Scaling in the AWS Glue API or AWS Command Line Interface (AWS CLI), set the following job parameters:

- **Key** – `--enable-auto-scaling`
- **Value** – `true`

To enable Auto Scaling with your Glue interactive sessions through Jupyter or AWS Glue notebook, run following Jupyter magic:

```
%%configure
{
    "--enable-auto-scaling": "true"
}
```

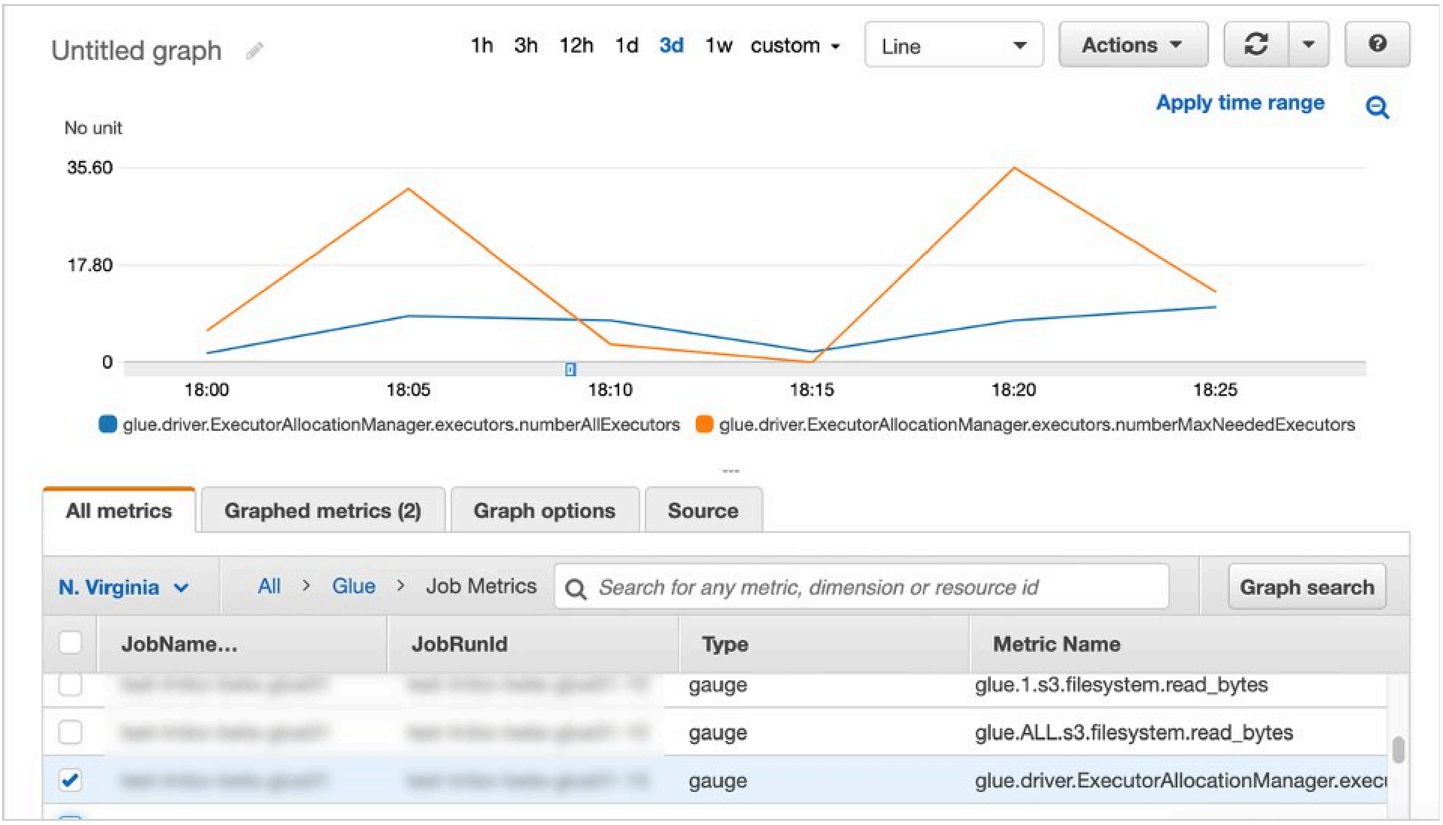
Monitor AWS Glue Auto Scaling

In this section, we discuss three ways to monitor AWS Glue Auto Scaling: via Amazon CloudWatch metrics or Spark UI.

CloudWatch metrics

After you enable AWS Glue Auto Scaling, Spark dynamic allocation is enabled and the executor metrics are visible in CloudWatch. You can review the following metrics to understand the demand and optimized usage of executors in their Spark applications enabled with Auto Scaling:

- `glue.driver.ExecutorAllocationManager.executors.numberAllExecutors`
- `glue.driver.ExecutorAllocationManager.executors.numberMaxNeededExecutors`

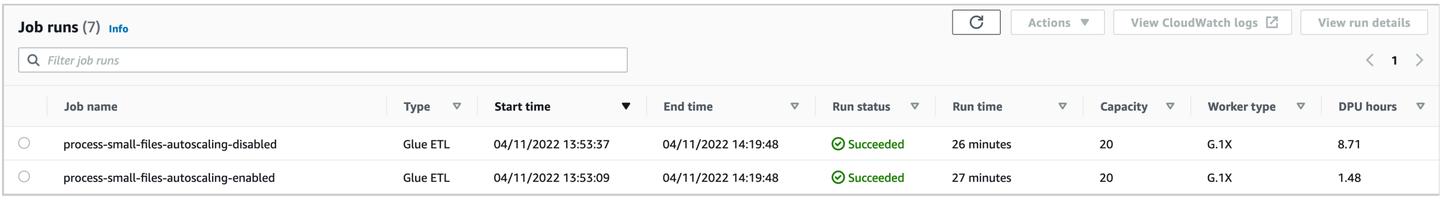


	JobName...	JobRunId	Type	Metric Name
☐	[illegible]	[illegible]	gauge	glue.1.s3.filesystem.read_bytes
☐	[illegible]	[illegible]	gauge	glue.ALL.s3.filesystem.read_bytes
✓	[illegible]	[illegible]	gauge	glue.driver.ExecutorAllocationManager.exec

Note that those CloudWatch metrics are available for jobs, but not for interactive sessions.

AWS Glue Studio Monitoring page

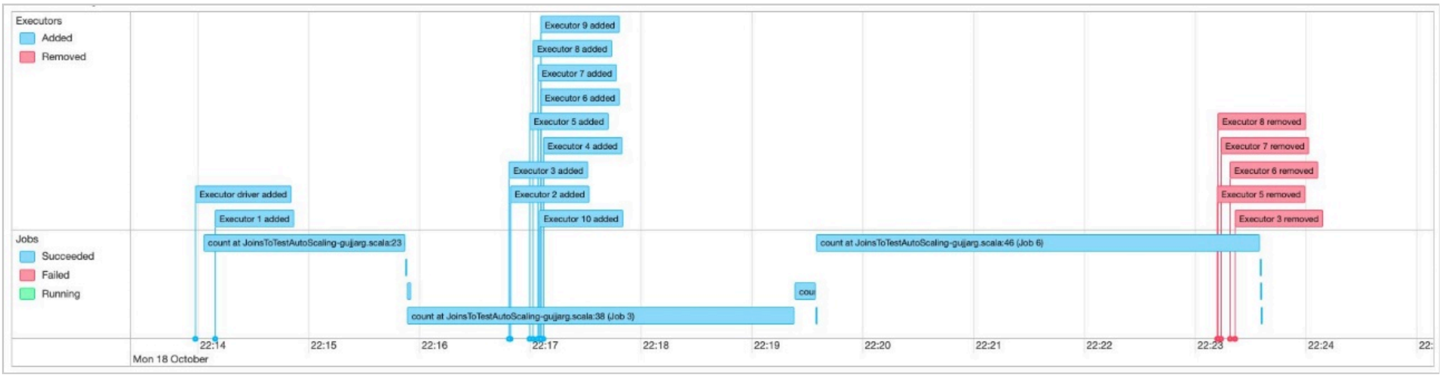
In the **Monitoring** page in AWS Glue Studio, you can monitor the DPU hours you spent for a specific job run. The following screenshot shows two job runs that processed the same dataset; one without Auto Scaling which spent 8.71 DPU hours, and another one with Auto Scaling enabled which spent only 1.48 DPU hours. The DPU hour values per job run are also available with `GetJobRun` API responses.



	Job name	Type	Start time	End time	Run status	Run time	Capacity	Worker type	DPU hours
○	process-small-files-autoscaling-disabled	Glue ETL	04/11/2022 13:53:37	04/11/2022 14:19:48	Succeeded	26 minutes	20	G.1X	8.71
○	process-small-files-autoscaling-enabled	Glue ETL	04/11/2022 13:53:09	04/11/2022 14:19:48	Succeeded	27 minutes	20	G.1X	1.48

Spark UI

With the Spark UI, you can monitor that the AWS Glue Spark cluster dynamically scales out and scales in with AWS Glue Auto Scaling. The event timeline shows when each executor is added and removed gradually over the Spark application run.



When you use interactive sessions from Jupyter notebook, you can run following magic to enable auto scaling along with Spark UI:

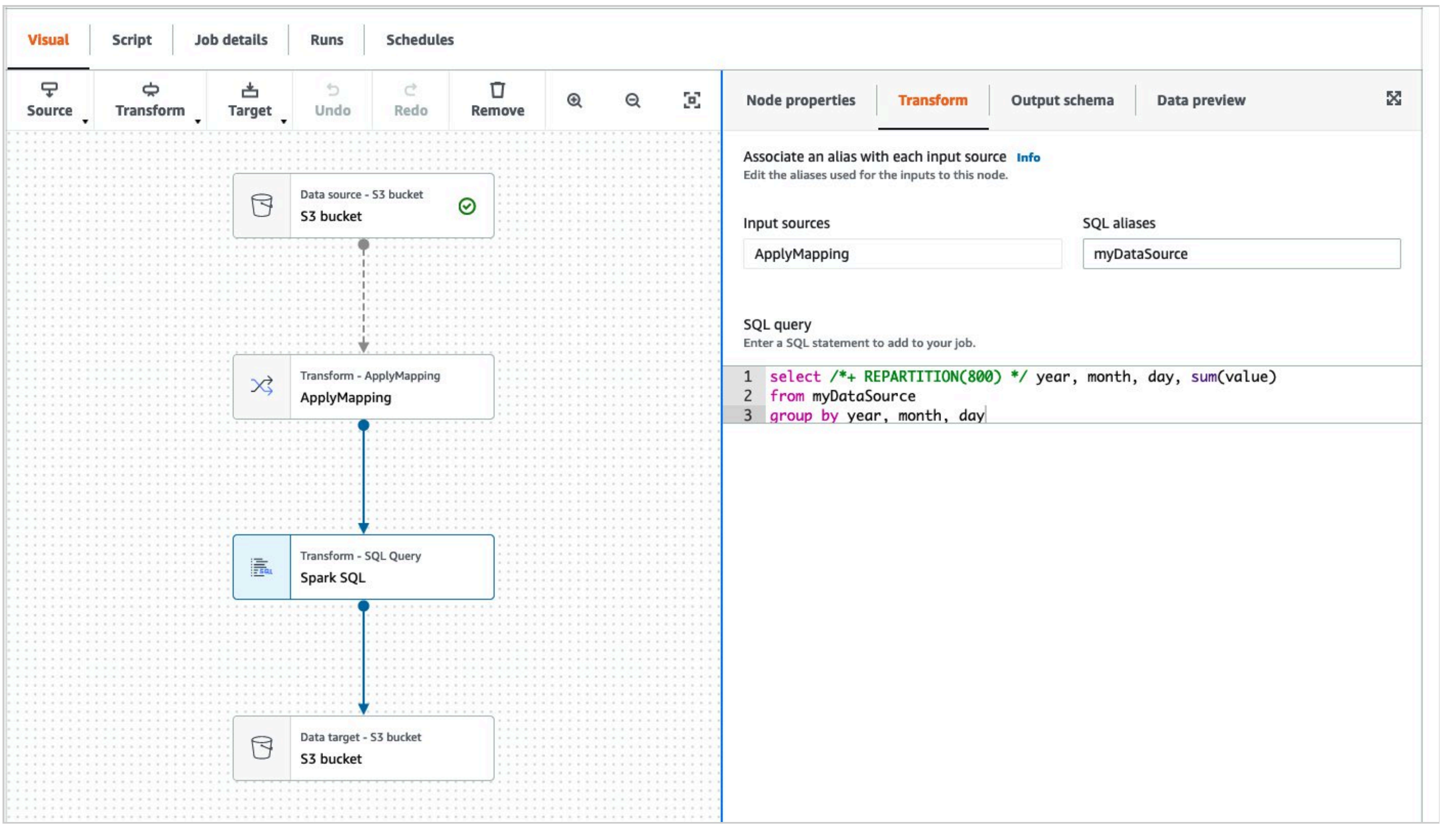
```
%%configure
{
    "--enable-auto-scaling": "true",
    "--enable-continuous-cloudwatch-log": "true"
}
```

In the following sections, we demonstrate AWS Glue Auto Scaling with two use cases: jobs with driver-heavy workloads, and jobs with multiple stages.

Example 1: Jobs containing driver-heavy workloads

A typical workload for AWS Glue Spark jobs is to process many small files to prepare the data for further analysis. For such workloads, AWS Glue has built-in optimizations, including file grouping, a Glue S3 Lister, partition pushdown predicates, partition indexes, and more. For more information, see Optimize memory management in AWS Glue. All those optimizations execute on the Spark driver and speed up the planning phase on Spark driver to compute and distribute the work for parallel processing with Spark executors. However, without AWS Glue Auto Scaling, Spark executors are idle during the planning phase. With Auto Scaling, Glue jobs only allocate executors when the driver work is complete, thereby saving executor cost.

Here's the example DAG shown in AWS Glue Studio. This AWS Glue job reads from an Amazon Simple Storage Service (Amazon S3) bucket, performs the `ApplyMapping` transformation, runs a simple SELECT query repartitioning data to have 800 partitions, and writes back to another location in Amazon S3.



Associate an alias with each input source Info
Edit the aliases used for the inputs to this node.

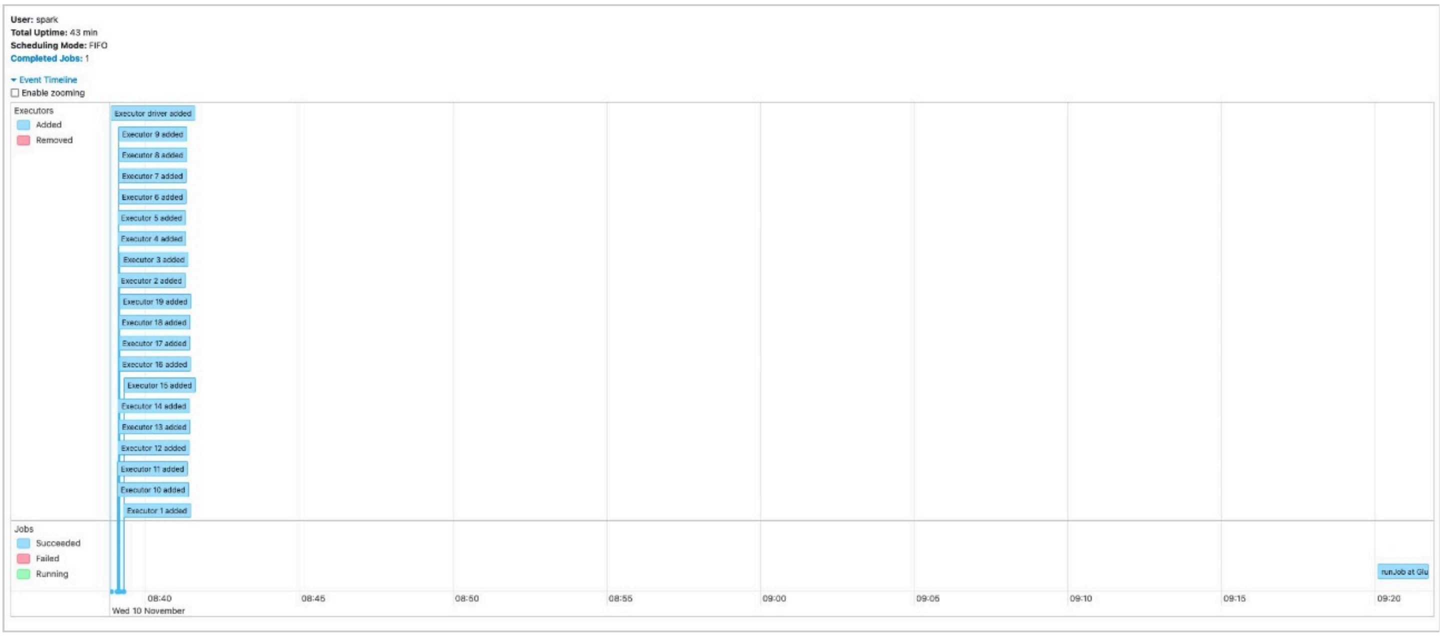
Input sources: ApplyMapping
SQL aliases: myDataSource

SQL query
Enter a SQL statement to add to your job.

```
select /*+ REPARTITION(800) */ year, month, day, sum(value)
from myDataSource
group by year, month, day
```

Without AWS Glue Auto Scaling

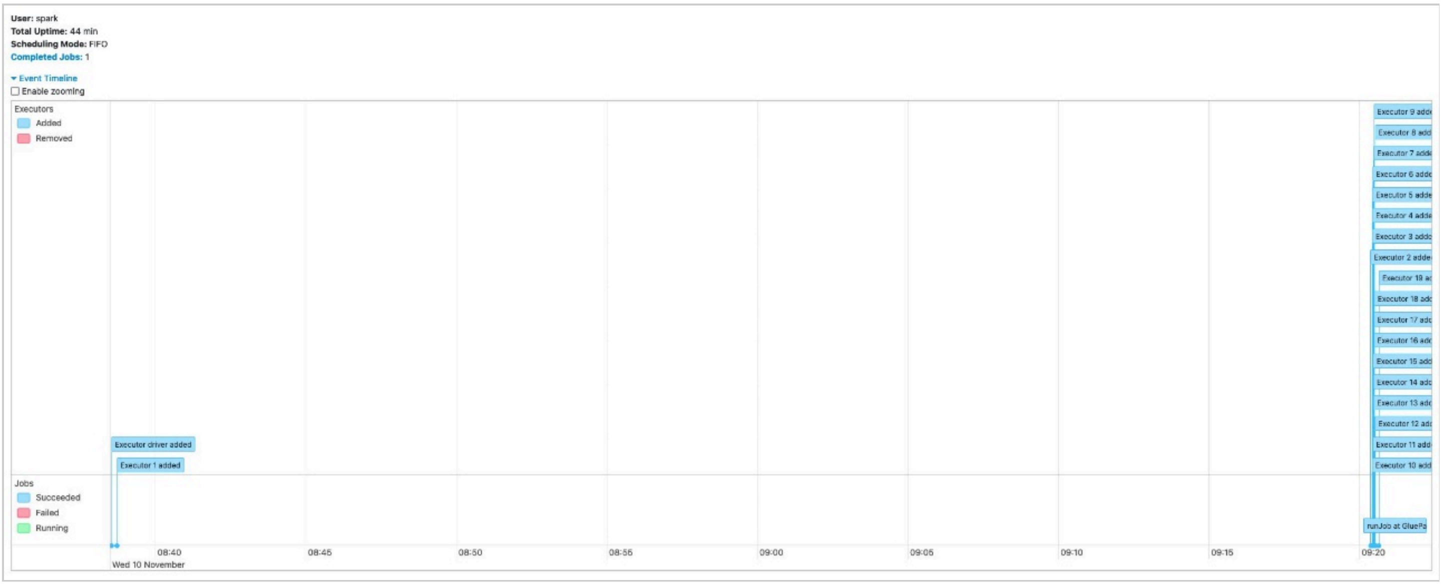
The following screenshot shows the executor timeline in Spark UI when the AWS Glue job ran with 20 workers without Auto Scaling. You can confirm that all 20 workers started at the beginning of the job run.



User: spark
Total Uptime: 43 min
Scheduling Mode: FIFO
Completed Jobs: 1

With AWS Glue Auto Scaling

In contrast, the following screenshot shows the executor timeline of the same job with Auto Scaling enabled and the maximum workers set to 20. The driver and one executor started at the beginning, and other executors started only after the driver finished its computation for listing 367,920 partitions on the S3 bucket. These 19 workers were not charged during the long-running driver task.

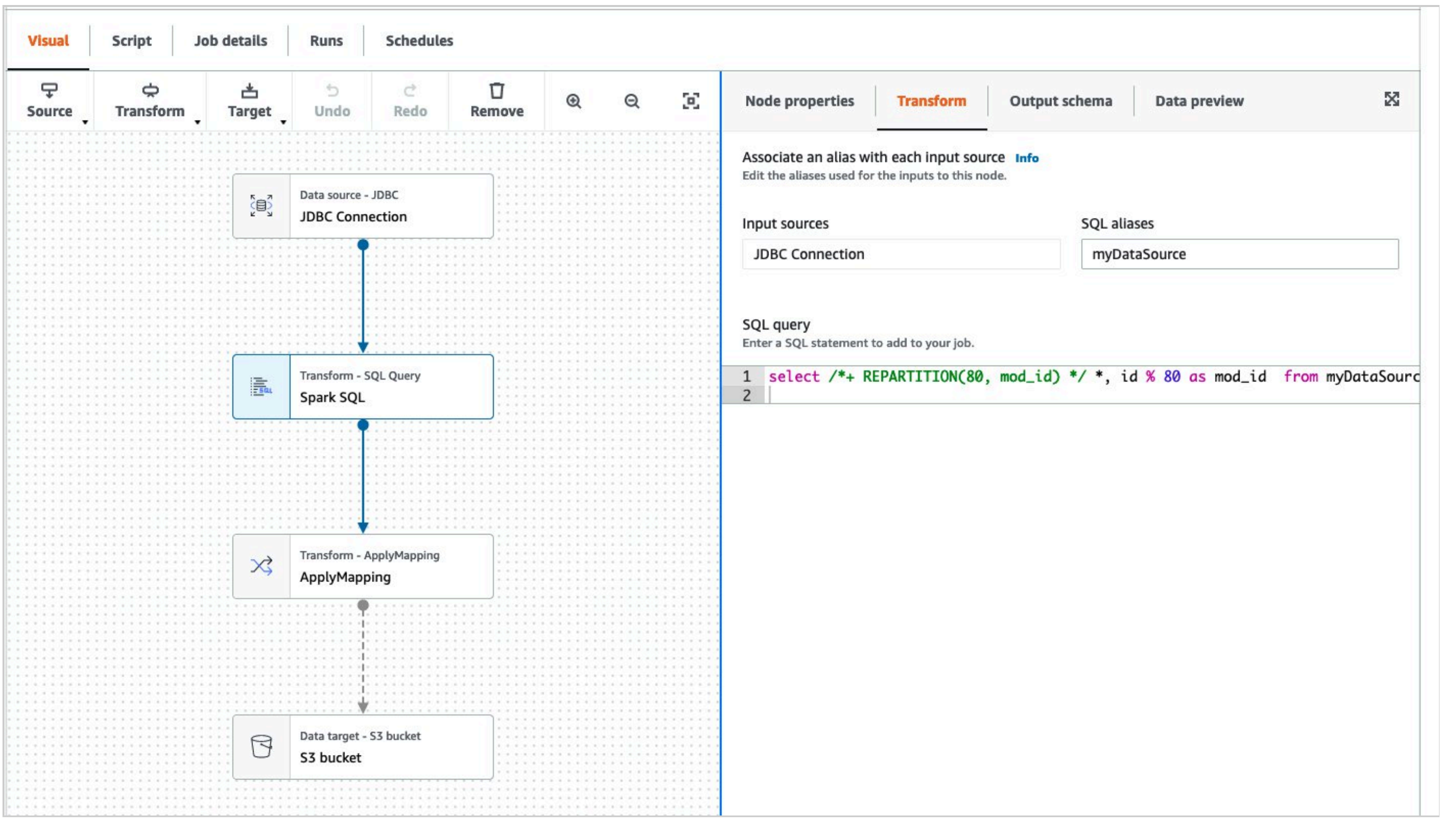

User: spark
Total Uptime: 44 min
Scheduling Mode: FIFO
Completed Jobs: 1

Both jobs completed in 44 minutes. With AWS Glue Auto Scaling, the job completed in the same amount of time with lower cost.

Example 2: Jobs containing multiple stages

Another typical workload in AWS Glue is to read from the data store or large compressed files, repartition it to have more parallelism for downstream processing, and process further analytic queries. For example, when you want to read from a JDBC data store, you may not want to have many concurrent connections, so you can avoid impacting source database performance. For such workloads, you can have a small number of connections to read data from the JDBC data store, then repartition the data with higher parallelism for further analysis.

Here’s the example DAG shown in AWS Glue Studio. This AWS Glue job reads from the JDBC data source, runs a simple SELECT query adding one more column (`mod_id`) calculated from the column ID, performs the `ApplyMapping` node, then writes to an S3 bucket with partitioning by this new column `mod_id` . Note that the JDBC data source was already registered in the AWS Glue Data Catalog, and the table has two parameters, `hashfield=id` and `hashpartitions=5` , to read from JDBC through five concurrent connections.



Associate an alias with each input source Info
Edit the aliases used for the inputs to this node.

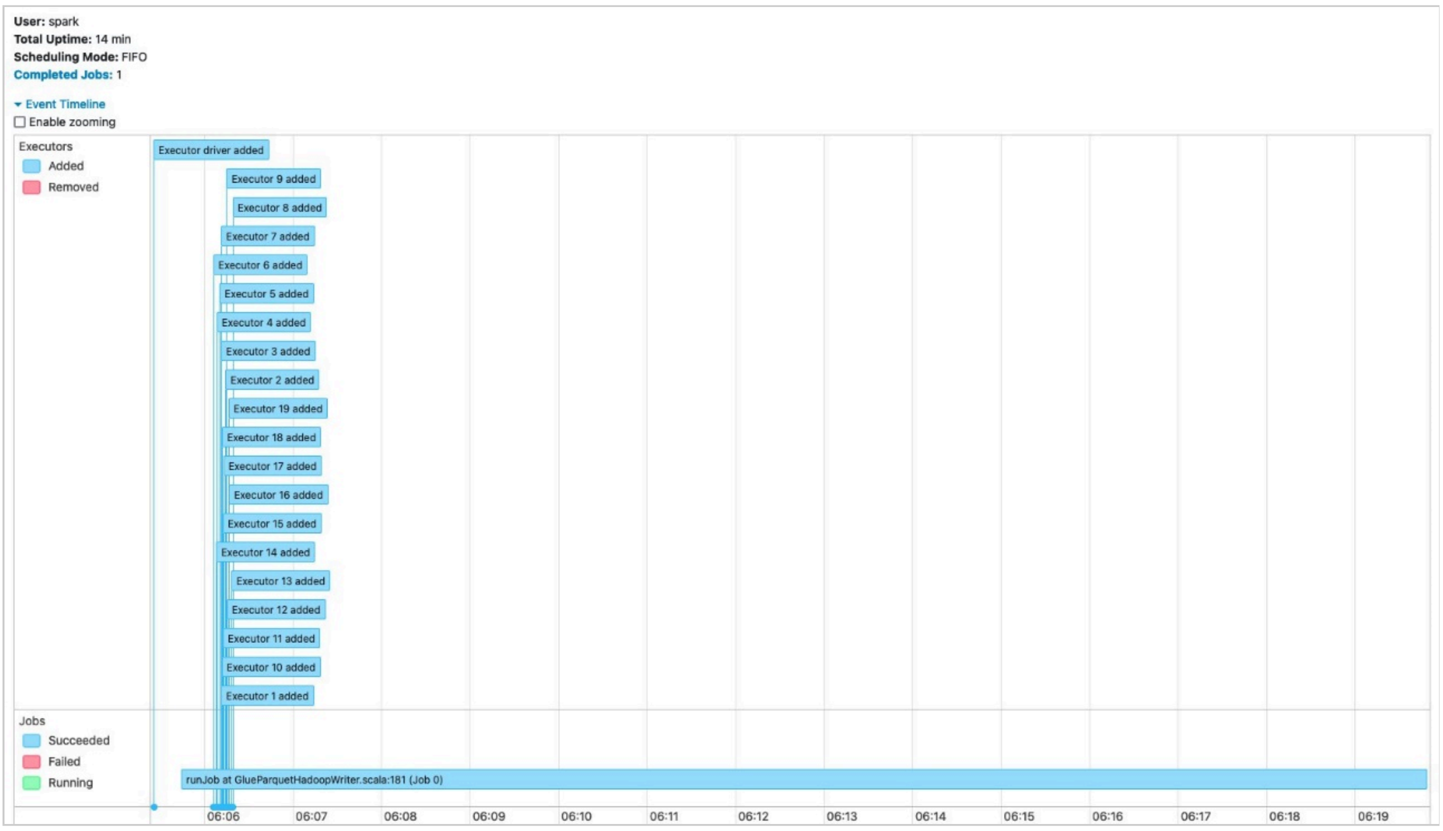
Input sources: JDBC Connection
SQL aliases: myDataSource

SQL query
Enter a SQL statement to add to your job.

```
select /*+ REPARTITION(80, mod_id) */ *, id % 80 as mod_id  from myDataSourc
```

Without AWS Glue Auto Scaling

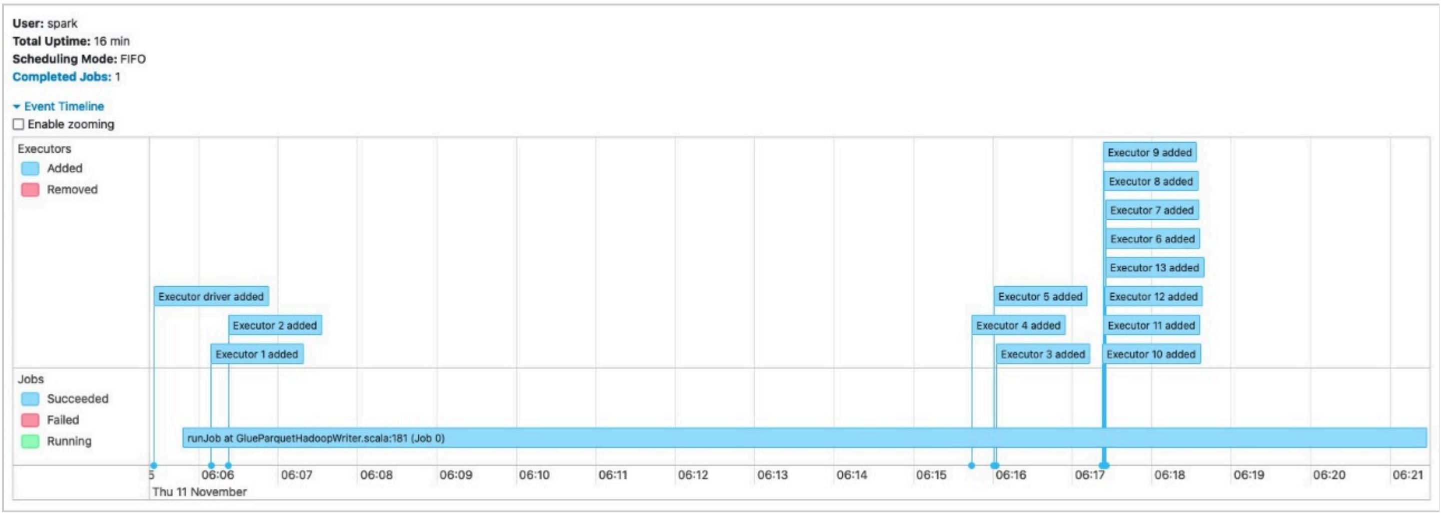
The following screenshot shows the executor timeline in the Spark UI when the AWS Glue job ran with 20 workers without Auto Scaling. You can confirm that all 20 workers started at the beginning of the job run.



User: spark
Total Uptime: 14 min
Scheduling Mode: FIFO
Completed Jobs: 1

With AWS Glue Auto Scaling

The following screenshot shows the same executor timeline in the Spark UI with Auto Scaling enabled with 20 maximum workers. The driver and two executors started at the beginning, and other executors started later. The first two executors read data from the JDBC source with fewer number of concurrent connections. Later, the job increased parallelism and more executors were started. You can also observe that there were 16 executors, not 20, which further reduced cost.



User: spark
Total Uptime: 16 min
Scheduling Mode: FIFO
Completed Jobs: 1

Conclusion

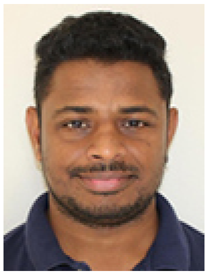
This post discussed AWS Glue Auto Scaling, which automatically resizes the computing resources of your AWS Glue Spark job capacity and reduce cost. You can start using AWS Glue Auto Scaling for both your existing workloads and future new workloads, and take advantage of it today! For more information about AWS Glue Auto Scaling, see Using Auto Scaling for AWS Glue. Migrate your jobs to Glue version 3.0 and get the benefits of Auto Scaling.

Special thanks to everyone who contributed to the launch: Raghavendhar Thiruvoipadi Vidyasagar, Ping-Yao Chang, Shashank Bhardwaj, Sampath Shreekantha, Vaibhav Porwal, and Akash Gupta.

About the Authors

Noritaka Sekiyama is a Principal Big Data Architect on the AWS Glue team. He is passionate about architecting fast-growing data platforms, diving deep into distributed big data software like Apache Spark, building reusable software artifacts for data lakes, and sharing the knowledge in AWS Big Data blog posts. In his spare time, he enjoys taking care of killifish, hermit crabs, and grubs with his children.

Bo Li is a Software Development Engineer on the AWS Glue team. He is devoted to designing and building end-to-end solutions to address customers' data analytic and processing needs with cloud-based, data-intensive technologies.



Rajendra Gujja is a Software Development Engineer on the AWS Glue team. He is passionate about distributed computing and everything and anything about data.

Mohit Saxena is a Senior Software Development Manager on the AWS Glue team. His team works on distributed systems for efficiently managing data lakes on AWS and optimizes Apache Spark for performance and reliability.

Like Share

Comments

Log in to comment