AWS Whitepaper

AWS Glue Best Practices: Building a Performant and Cost Optimized Data Pipeline

AWS Glue Best Practices: Building a Performant and Cost Optimized Data Pipeline : AWS Whitepaper

Table of Contents

AWS Glue Best Practices: Building a Performant and Cost Optimized Data Pipeline

Publication date: **August 26, 2022** (*Document revisions*)

Abstract

Data integration is a critical element in building a data lake and a data warehouse. Data integration enables data from different sources to be cleaned, harmonized, transformed, and finally loaded. When building a data warehouse, the bulk of the development efforts are required for building a data integration pipeline. Data integration is one of the most critical elements in data analytics ecosystems. An efficient and well-designed data integration pipeline is critical for making the data available, and being trusted amongst analytics consumers.

This whitepaper shows you some of the consideration and best practices in building high-performance, cost-optimized data pipelines with AWS Glue.

To get the most out of reading this whitepaper, it's helpful to be familiar with AWS Glue, AWS Glue DataBrew, Amazon Simple Storage Service (Amazon S3), AWS Lambda, and AWS Step Functions.

- For best practices around Operational Excellence for your data pipelines, refer to AWS Glue Best Practices: Building an Operationally Efficient Data Pipeline.
- For best practices around Security and Reliability for your data pipelines, refer to AWS Glue Best Practices: Building a Secure and Reliable Data Pipeline.

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

For more expert guidance and best practices for your cloud architecture—reference architecture deployments, diagrams, and whitepapers—refer to the AWS Architecture Center.

Introduction

Data volumes and complexities are increasing at an unprecedented rate, exploding from terabytes to petabytes or even exabytes of data. Traditional on-premises based approaches for bundling a data pipeline do not work well with a cloud-based strategy, and most of the time, do not provide the elasticity and cost effectiveness of cloud native approaches. We hear from customers all the time that they are looking to extract more value from their data but struggling to capture, store, and analyze all the data generated by today's modern and digital businesses. Data is growing exponentially, coming from new sources, it is increasingly diverse, and needs to be securely accessed and analyzed by any number of applications and people.

With changing data and business needs, the focus on building a high performing, cost effective, and low maintenance data pipeline is paramount. Introduced in 2017, AWS Glue is a fully managed, serverless data integration service which allows customers to scale based on their workload with no infrastructures to manage. In the next section, we discuss common best practices around performance efficiency and cost optimization of the data pipeline built with AWS Glue. This document is intended for advanced users, data engineers and architects.

Refer to AWS Glue Best Practices: Building an Operationally Efficient Data Pipeline to understand more about the AWS Glue product family before proceeding to the next sections.

Using AWS Well-Architected framework for building a data pipeline

Building a well architected data pipeline is critical for the success of a data engineering project. While designing a well-architected data pipeline we take the guidelines of the Amazon Web Services (AWS) Well-Architected Framework. This helps you to understand the pros and cons of decisions you make while building applications on AWS and guide the architecture considerations in operating reliable, secure, efficient, and cost-effective systems in the cloud. It provides a way for you to consistently measure your architectures against best practices and identify areas for improvement. We believe that having well-architected data pipeline using the well architected pillars greatly increases the likelihood of success. The AWS Well-Architected Framework is based on six pillars:

- **Operational Excellence** — The Operational Excellence pillar includes the ability to support development and run workloads effectively, gain insight into their operations, and to continuously improve supporting processes and procedures to deliver business value. You can find prescriptive guidance on implementation in the *Operational Excellence Pillar* whitepaper.
- **Security** — The Security pillar encompasses the ability to protect data, systems, and assets to take advantage of cloud technologies to improve your security. You can find prescriptive guidance on implementation in the *Security Pillar* whitepaper.
- **Reliability** — The Reliability pillar encompasses the ability of a workload to perform its intended function correctly and consistently when it's expected to. This includes the ability to operate and test the workload through its total lifecycle. You can find prescriptive guidance on implementation in the *Reliability Pillar* whitepaper.
- **Performance Efficiency** —The Performance Efficiency pillar includes the ability to use computing resources efficiently to meet system requirements, and to maintain that efficiency as demand changes and technologies evolve. You can find prescriptive guidance on implementation in the *Performance Efficiency Pillar* whitepaper.
- **Cost Optimization** — The Cost Optimization pillar includes the ability to run systems to deliver business value at the lowest price point. You can find prescriptive guidance on implementation in the *Cost Optimization Pillar* whitepaper.
- **Sustainability** — The Sustainability pillar focuses on environmental impacts, especially energy consumption and efficiency, since they are important levers for architects to inform direct

action to reduce resource usage. You can find prescriptive guidance on implementation in the *Sustainability Pillar* whitepaper.

This whitepaper covers best practices around Performance Efficiency and Cost Optimization of data pipelines.

- For best practices around Operational Excellence for your data pipelines, refer to AWS Glue Best Practices: Building an Operationally Efficient Data Pipeline.
- For best practices around Security and Reliability for your data pipelines, refer to AWS Glue Best Practices: Building a Secure and Reliable Data Pipeline.

Building a performance efficient data pipeline

The Performance Efficiency pillar includes the ability to use computing resources efficiently to meet workload requirements, and to maintain that efficiency as demand changes and technologies evolve. Here we consider some of the factors which help you improve the Performance Efficiency aspects of your data pipeline.

Data partitioning and bucketing

If you need to ingest data from a large dataset into your data pipeline and the data is not properly distributed for optimized usage of compute resources, the performance efficiency may not be optimal. Partitioning and bucketing can help you get the best performance from your data pipeline by distributing the data, and reducing the amount of data that needs to be read by the respective compute resources.

Partitioning data

Partitioning groups data into parts and keeps the related data together based on a specific value or values. For example, a customer who wants to store website clicks stream data might store it by grouping the data together by year, month, day, country, and so on. These grouped data sets are then stored together as files in the same partition.

Partition keys act as virtual columns. You define them at table creation time, and when data is added to Amazon S3 via services such as AWS Glue extract, transform, load (ETL), Amazon Athena, and S3, data is grouped and persisted based on the partition keys values such as `country`=US.

Data can be partitioned with one or more partition column and the partitions are hierarchical. For example:

```
s3://<bucket_name>/<table_name>/country=<country
value>/year=<year_value>/month=<month_value>/day=<day_value>/
```

Here the data is partitioned first by country and then by year, month, and day. The partition format shown is called “Hive style”, which adds a key (in this example, the country) and value (such as US) to the file path. The other partition style supported by AWS Glue is “Unnamed style”; however, hive partition style is one of the most widely used methods of partitioning data for big data processing.

Example: Data Definition Language (DDL) script of a partitioned table.

```
CREATE EXTERNAL TABLE website_clickstream_events (
`event_time` timestamp,
`ip_address` string,
`page` string,
`action` string)
PARTITIONED BY (
country string,
year bigint,
month bigint,
day bigint
)
INTO 1000 BUCKETS
STORED AS TEXTFILE
LOCATION
's3://<bucket name>/website_clickstream_events/'
```

Why partitioning?

Partitioning can help reduce the amount of data scanned, thereby improving performance and reducing overall cost of analytics using pushdown predicates. Using pushdown predicates, instead of reading the entire dataset and then filtering the data after it is loaded into memory, you can apply the filter directly on the partition metadata in the data catalog. Then you only list and read what you need.

Example: Partitions

```
s3://<bucket_name>/webclickstream_events/country=US/year=2021/month=06/day=01/
s3://<bucket_name>/webclickstream_events/country=US/year=2021/month=06/day=02/
s3://<bucket_name>/webclickstream_events/country=US/year=2021/month=06/day=03/
```

Pre-filtering using pushdown predicates

In many cases, you can use a pushdown predicate to filter on partitions without having to list and read all the files in your dataset. Instead of reading the entire dataset and then filtering in a DynamicFrame, you can apply the filter directly on the partition metadata in the data catalog. Then you only list and read what you need into a DynamicFrame.

In the previous S3 partitions example, when you want to get data for `day=01`, instead of loading all files under `s3://<bucket name>/website_clickstream_events/`, AWS Glue can load data under path `s3://<bucket name>/website_clickstream_events/country=US/year=2021/month=06/day=01/` by filtering partitions based on the **day** predicate and load files under `day=01/` . This improves the data processing performance and reduces the overall cost for analytics.

Example: Query

```
datasource = glueContext.create_dynamic_frame_from_catalog(
database = "default",
table_name = "website_clickstream_events",
push_down_predicate = "year = '2021' and month = '06' and day='01' ",
transformation_ctx = "datasource")
```

How to partition?

AWS Glue supports partitioning data using Spark SQL and DataFrame APIs. You can partition the data by specifying the columns based on which you want to group the data. For example, by year, month, or country.

Each file stored inside a partition should be at least 128 MB to a maximum of one GB to get ensure that AWS Glue (Spark) can read and process the data efficiently. If the file sizes are too small (KBs to few MBs), AWS Glue will spend more time in I/O and can lead to degraded performance.

You should choose partitions column that have similar characteristics, such as records from the same country and that can have a limited number of possible values. This characteristic is known as data cardinality. For example, if you partition by the column country, and this column has a limited number of distinct values (low cardinality), partitioning by country works well and decreases query latency. But if you partition by the column transaction date, it'll have a higher number of distinct values (high cardinality) and leads to increased query latency.

Partition index

Querying tables with many partitions (10s of 1000s), create performance challenges as AWS Glue has to scan through the partitions in the AWS Glue Data Catalog and load the partitions that are relevant to the query/filter criteria. In order to improve the response time of scanning tables with large number of partitions, AWS Glue Data Catalog now provides Partition Indexes that can help improve performance.

Partition indexes are created by combining a sub list of partition keys defined in the table. A partition index can be created on any list of partition keys defined on the table. For the previous `website_clickstream_events` table, some of the possible indexes are `(country, year, month, day)`, `(country, year)`, `(country)`.

There is a soft limit for number of partitions in a table and across all AWS Glue tables in an AWS Account. A soft limit can be increased by raising support tickets.

Refer to Working with Partition Indexes to understand how to create and manage partition indexes.

File formats and data compression

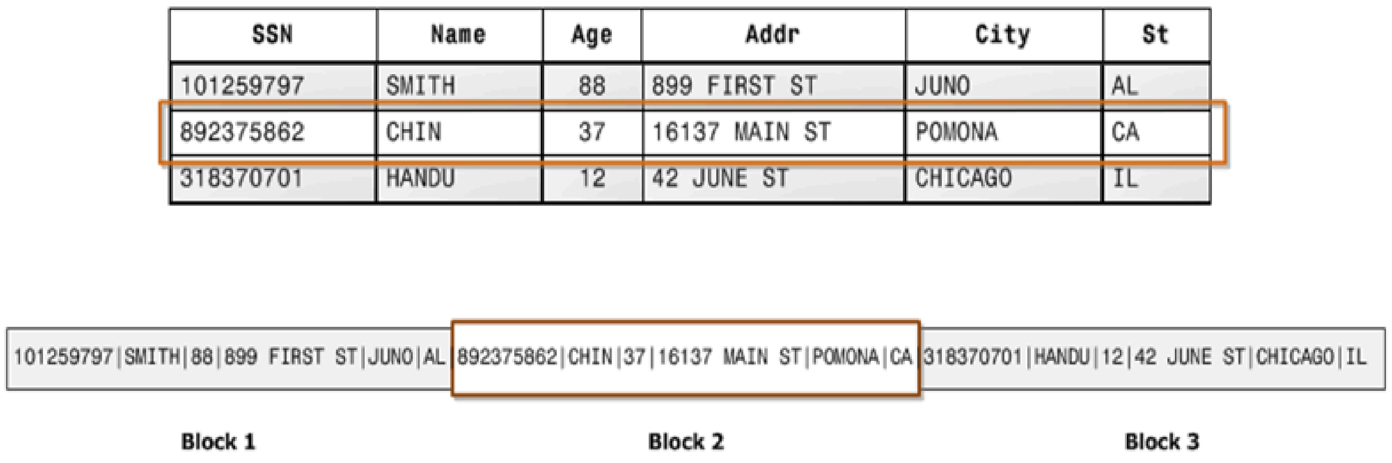
Columnar data formats are used in data lake storage for faster analytics workloads, as opposed to row formats. Columnar formats significantly reduce the amount of data that needs to be fetched by accessing columns that are relevant for the workload. Let's look at each of these formats in more detail.

Row vs. columnar storage

Columnar storage for database tables is an important factor in optimizing analytic query performance because it drastically reduces the overall disk I/O requirements and reduces the amount of data you need to load from disk.

The following series of images describe how columnar data storage implements efficiencies and how that translates into efficiencies when retrieving data into memory.

This first image shows how records from database tables are typically stored into disk blocks by row.



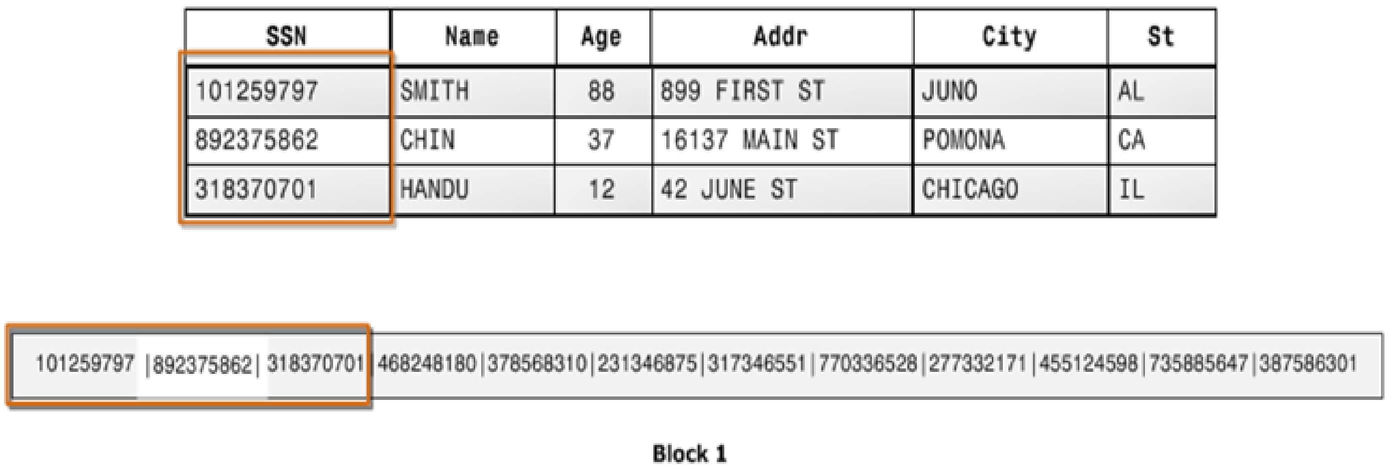
SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

101259797|SMITH|88|899 FIRST ST|JUNO|AL
892375862|CHIN|37|16137 MAIN ST|POMONA|CA
318370701|HANDU|12|42 JUNE ST|CHICAGO|IL
Block 1
Block 2
Block 3

How records from database tables are typically stored into disk blocks by row

In a typical relational database table, each row contains field values for a single record. In row-wise database storage, data blocks store values sequentially for each consecutive column making up the entire row. If block size is smaller than the size of a record, storage for an entire record may take more than one block. If block size is larger than the size of a record, storage for an entire record may take less than one block, resulting in an inefficient use of disk space. In online transaction processing (OLTP) applications, most transactions involve frequently reading and writing all the values for entire records, typically one record or a small number of records at a time. As a result, row-wise storage is optimal for OLTP databases.

The next image shows how with columnar storage, the values for each column are stored sequentially into disk blocks.



SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

101259797 |892375862| 318370701 |468248180|378568310|231346875|317346551|770336528|277332171|455124598|735885647|387586301

With columnar storage, values for each column are stored sequentially into disk blocks.

Using columnar storage, each data block stores values of a single column for multiple rows.

In this simplified example, using columnar storage, each data block holds column field values for as many as three times as many records as row-based storage. This means that reading the same number of column field values for the same number of records requires a third of the input/output (I/O) operations compared to row-wise storage. In practice, using tables with very large numbers of columns and very large row counts, storage efficiency is even greater.

An added advantage is that, since each block holds the same type of data, block data can use a compression scheme selected specifically for the column data type, further reducing disk space and I/O.

The savings in space for storing data on disk also carries over to retrieving and then storing that data in memory. Since many database operations only need to access or operate on one or a small number of columns at a time, you can save memory space by only retrieving blocks for columns you need for a query. Where OLTP transactions typically involve most or all the columns in a row for a small number of records, data analysis queries commonly read only a few columns for a very large number of rows. This means that reading the same number of column field values for the same number of rows requires a fraction of the I/O operations and uses a fraction of the memory that would be required for processing row-wise blocks.

In practice, using tables with very large numbers of columns and very large row counts, the efficiency gains are proportionally greater. For example, suppose a table contains 100 columns. A query that uses five columns will only need to read about five percent of the data contained in the table. This savings is repeated for possibly billions or even trillions of records for large databases. In contrast, a row-wise database would read the blocks that contain the 95 unneeded columns as well.

Apache Parquet and ORC are columnar storage formats that are optimized for fast retrieval of data and used in AWS analytical applications. Columnar storage formats have the following characteristics that make them suitable for using with data analysis:

- **Compression by column, with compression algorithm selected for the column data type** to save storage space in Amazon S3 and reduce disk space and I/O during query processing.
- **Predicate pushdown** in Parquet and ORC enables search engine to fetch only the blocks it needs, improving query performance. When a query obtains specific column values from your data, it uses statistics from data block predicates, such as max/min values, to determine whether to read or skip the block.
- **Splitting of data** in Parquet and ORC allows search engine to split the reading of data to multiple readers and increase parallelism during its query processing.

To convert your existing raw data from other storage formats to Parquet or ORC, you can run CREATE TABLE AS SELECT (CTAS) queries in Athena and specify a data storage format as Parquet or ORC, or use a AWS Glue ETL job.

Compression

Compressing data help reduce the amount of data stored in the storage layer and improves the write and read operation performance along with improved network and I/O throughput. Compared to working with uncompressed data, data compression improves overall data pipeline performance.

AWS Glue supports multiple compression formats natively. Some of the popular formats are

- **SNAPPY** — Snappy is the default compression format for files in the Parquet data storage format. It is a fast compression algorithm that provides moderate compression at a minimum speed of 250MB/s. When combined with Parquet format, you can create highly compressed, splittable files that enable better performance and throughput.
- **ZLIB** — ZLIB is the default compression format for files stored in the Optimized Row Columnar (ORC) data storage format in AWS Glue. ORC is the default storage format for Apache Hive/Tez Engine.
- **Gzip** — Gzip compression is one of most widely available compression codec. You can use this compression format when you need to exchange data across wide variety of applications and systems that may not necessarily support other formats. GZIP is CPU-intensive and it is not splittable. It cannot be processed in parallel by distributed data processing engines. Hence, it is a good fit for processing data that are not used often but require a high compression ratio, such as archival data.
- **BZIP2** — BZip2 can provide better compression ratio than GZip at the cost of speed (CPU). It is a splittable format and can be processed in parallel by distributed data processing engines. It is a good option when compression needs are critical. Because BZip2 is compute intensive, it is not recommended for data that are queried often.

Following are some factors to consider when choosing one or the other compression format

Table 1 — Factors to consider when choosing a compression format

Algorithm	**Splittable?**	**Compression ratio**	**Compress / Decompress speed**
Gzip (DEFLATE)	No	High	Medium
Bzip2	Yes	Very high	Slow
LZO	No	Low	Fast
Snappy	No	Low	Very fast

Configure compression format in AWS Glue

In AWS Glue, compression format for a file can be specified in few ways depending upon how you access the data.

Using the AWS Glue’ dynamic data frame library

```
glueContext.write_dynamic_frame.from_options(
frame = datasource1,
connection_type = "s3",
connection_options = {
"path": "s3://s3path"
},
format = "glueparquet",
format_options={"compression": "snappy"}
transformation_ctx = "datasink1")
```

Using PySpark

```
df.write
.option("compression", "snappy")
.parquet("s3://output-path")
```

Using Amazon Athena / SPARK SQL

```
CREATE EXTERNAL TABLE sampleTable (
column1 INT,
column2 INT
) STORED AS PARQUET
TBLPROPERTIES (
'classification'='parquet',
'compression'='snappy')
LOCATION '"s3://output-path"'
```

Avoid or minimize User defined functions (UDFs)

User-defined functions (UDFs) are user-programmable routines that transform values from a single row to produce a single corresponding output value per row. UDFs, gives data engineers the flexibility to create new functions in higher level languages, abstracting their lower-level language implementations. However, our recommendation when working with AWS Glue or Spark code would be to use native Spark SQL functions as much as possible and limit the usage of UDFs to scenarios where a built-in function doesn’t exist.

Spark SQL functions operate directly on a Java virtual machine (JVM) and are well integrated with both Catalyst and Tungsten. This provides the advantage that functions can be optimized in the execution plan and benefit from spark native optimizations. UDFs in general increase the memory footprint because of the need to serialize and deserialize the data to be sent across spark execution engine and the JVM (plus Python process in case of PySpark).

Building a cost-effective data pipeline

A cost-optimized data pipeline fully uses all resources, achieves an outcome at the lowest possible price point, and meets your functional requirements. Here we are providing best practices to optimized price performance of your data pipeline.

The right AWS Glue worker type

This section of the document discusses the different worker nodes available in AWS Glue, their differences, and provides guidance on selecting the appropriate worker type based on your workload.

The following table summarizes the available AWS Glue worker types:

Table 2 — AWS Glue worker types

Worker name	vCPU	Memory (GB)	Attached storage (GB)
Standard	4	16	50
G.1X	4	16	64
G.2X	8	32	128

When creating an AWS Glue job with either of these worker types, the following rule applies:

Standard

- You specify the maximum number of Data Processing Units (DPUs) required for the job
- Each standard worker launches two executors
- Each executor launches with four Spark cores

G.1X

- You specify the maximum number of workers

- Each worker corresponds to one DPU
- Each worker launches one executor
- Each executor launches with eight Spark cores

- In AWS Glue 3.0, each job launches with four cores per executor

G.2X

- You specify the maximum number of workers
- Each worker corresponds to two DPUs
- Each worker launches one executor
- Each executor launches with 16 Spark cores

- In Glue 3.0, each job launches with eight cores per executor

We recommend using G.1X or G.2X workers for jobs authored in AWS Glue 2.0 and above. Based on whether your job requires more data parallelism, (for example, they benefit from horizontal scaling) adding more G.1X workers is recommended. For jobs that have intense memory requirements - or ones that benefit from vertical scaling - adding more G.2X workers is recommended. Additionally, the G.2X jobs benefit from having additional disk space.

Estimate AWS Glue DPU

AWS Glue has autoscaling feature which helps to avoid the complexities involved in calculating the right number of DPUs for a job. AWS Glue 3.0 jobs can be configured to auto-scale, meaning the jobs can now dynamically scale resources up and down based on the workload, for both batch and streaming jobs. With autoscaling, there is no longer a need to worry about over-provisioning resources for jobs, spend time optimizing the number of workers, or pay for idle workers.

Common scenarios where automatic scaling helps with cost and utilization for your Spark applications include a Spark driver listing a large number of files in Amazon S3 or performing a load while executors are inactive, Spark stages running with only a few executors due to overprovisioning, and data skews or uneven computation demand across Spark stages.

To enable autoscaling, set the **--enable-auto-scaling** flag to true, or enable it manually from AWS Glue Studio while authoring the job. Additionally, choose the type and maximum number of workers and AWS Glue will choose the right size resources for the workload.

automatic scaling is available for AWS Glue jobs with both G1.X and G2.X worker types. Standard DPUs are not supported.

When not using autoscaling, we use a rough calculator to try estimate the AWS Glue job's DPU requirement, the following section provides more details on the approach.

For estimating the DPU requirements at job level, let's break down the jobs into different complexity grades - Low, Medium, and High. The sizing of the jobs is purely based on the number of transformations.

A job that does only source A to target B data movement, with no transformation or with minor data filtering, can be considered Low on the complexity scale. Similarly, a job that involves multiple joins, UDFs, window functions, and so on can be considered a High complexity job.

The maximum number of workers you can define is 299 for G.1X, and 149 for G.2X. These are not hard limits and can be increased.

Let's attach the following weights to each complexity scale:

Table 3 — Complexity weight by complexity level

Complexity level	Weight
Low	2
Medium	6
High	10

Next, we apply the following formula to calculate the DPU requirements for a job based on G.1X worker.

```
DPU Estimate = MIN((CEIL((((data_volume_in_GB *
weight)/16),1)+1,299)
```

Let's consider the following scenario:

Table 4 — Sample low complexity job

Job name	Job 1
Profile	Low
Data volume	160 GB

Based on the previous scenario, the following calculation applies:

```
DPU Estimate = MIN(CEIL((160*2)/16,1)+1,299) = MIN (21,299) = 21
```

For the same data input, the following table lists the DPU estimates for each complexity level:

Table 5 — DPU estimate by complexity level

Complexity level	DPU estimate
Low	21
Medium	61
High	101

Be advised that the data needs to be partitioned and should have at least as many partitions as the number of spark cores in order to efficiently process the data. The calculations above are designed to assist you with getting started with a worker configuration. Once you set up and run your AWS Glue jobs, you will be able to monitor for the actual usage which may be just right with the demand, slightly over or below. Based on the outcome you can adjust and further optimize your worker counts to meet your processing requirements.

Additional considerations

PySpark vs. Python Shell vs. Scala

AWS Glue ETL scripts can be coded in Python or Scala. Python scripts use a language that is an extension of the PySpark Python dialect for ETL jobs. The script contains extended constructs to deal with ETL transformations. When you automatically generate the source code logic for your job, a script is created. You can edit this script, or you can provide your own script to process your ETL work.

Python shell

AWS Glue ETL supports running plain non-distributed Python scripts as a shell script to run small to medium-sized generic tasks that are often part of an ETL workflow. For example, to submit SQL queries to services such as Amazon Redshift, Amazon Athena, or Amazon EMR, or run machine learning (ML) and scientific analyses.

Python shell jobs in AWS Glue come pre-loaded with libraries such as Boto3, NumPy, SciPy, pandas, and others.

You can run Python shell jobs using one Data Processing Unit (DPU) or 0.0625 DPU (which is 1/16 DPU), allowing you to run cost effective small to medium jobs that does not require Spark runtime.

Compared to AWS Lambda, which has a strict 15-minute maximum timeout, AWS Glue Python Shell can be configured with a much longer timeout and higher memory, often required for data engineering jobs.

PySpark jobs

AWS Glue version 2.0 and later (PySpark and Scala) provides an upgraded infrastructure for running Apache Spark ETL jobs in AWS Glue with reduced startup times. With the reduced wait times, data engineers can be more productive and increase their interactivity with AWS Glue. The reduced variance in job start times can help you with your SLAs of making data available for analytics.

AWS Glue PySpark extensions of Apache Spark provides additional capabilities and convenience functions to manipulate data. For example, PySpark extensions such as Dynamic Dataframe, Relationalize, FindMatches, FillMissingValues, and so on can be used to easily enrich transform and normalize data with few lines of code. For more information, refer to the AWS Glue PySpark Transforms Reference.

Scala jobs

AWS Glue provides high-level APIs in Scala and Python for scripting ETL Spark jobs. Customers who use Scala as their primary language to develop Spark jobs can now run those jobs on AWS Glue with little or no changes to their code. AWS Glue provides all PySpark equivalent extension libraries in Scala as well, such as Dynamic DataFrame, Relationalize, and so on. You can take full benefit of these extensions in both Scala and PySpark based ETL jobs.

Comparison chart

Table 6 — Comparing available AWS Glue ETL programing languages

Topic	Glue PySpark	Glue Scala	Glue Python Shell
Batch job DPUs	Minimum two, default ten	Minimum two, default 10	Minimum 0.0625, maximum one, default 0.0625
Batch job billing duration	Per second billing, minimum of one minute	Per second billing, minimum of one minute	Per second billing, minimum of one minute
Streaming job DPUs	Minimum two, default five	Minimum two, default five	N/A
Glue worker type	Standard (about to be deprecate d in favor of AWS Glue 1.x), AWS Glue 1.x, AWS Glue 2.X (memory intensive jobs)	Standard (about to be deprecate d in favor of AWS Glue 1.x), AWS Glue 1.x, AWS Glue 2.X (memory intensive jobs)	N/A
Streaming job billing duration	Per second billing, minimum of ten minutes	Per second billing, minimum of ten minutes	N/A

Topic	Glue PySpark	Glue Scala	Glue Python Shell
Language	Python	Scala	Python
Visual authoring	Yes (AWS Glue Studio)	No	No
Additional libraries	S3, Pip	S3	S3
Typical use case	Big data ETL, ML transforms	Big data ETL, ML transforms	Data integration jobs that typically do not need to run in a distributed environment (such as REST API calls, Amazon Redshift SQL queries, and so on)
Spark runtime	2.2, 2.4, and 3.1	2.2, 2.4, and 3.1	N/A
AWS Glue Studio support (visual authoring)	Yes	No	No
Notebook developme nt support	Yes	Yes	Yes

Custom classifiers

Classifiers in AWS Glue are mechanisms that help the crawlers determine the schema of our data. In most cases the default classifiers work well and suits the requirements. However, there are scenarios where we have to author our own customer classifiers. For example, log files that may not fall into regular CSV/JSON or XML messages, but would need a GROK expression to parse them, or CSV files with a non-standard delimiter, quote, characters, and so on.

Once attached to a crawler, a custom classifier is executed before the built-in classifiers. If the data is matched, the classification and schema is returned to the crawler, which is used to create the target tables.

Glue allows you to create custom classifiers for CSV, XML, JSON, and GROK-based datasets. In this document, we will explore how to create a classifier for a given dataset.

Assume you have a log file with the following structure:

```
2017-03-30npelling04C-50-CC-BB-F9-57/erat/nulla/tempus/vivamus.jpg
```

In this scenario, the data is unstructured, but you can apply a GROK expressions, such as a named regular expression (regex), to parse it to the form you want.

The target data structure is:

Table 7 — Expected data structure for log data

Column name	Sample value
`log_year`	2017
`log_month`	03
`log_day`	30
`username`	npelling
`mac_address`	04C-50-CC-BB-F9-57
`referer_url`	/erat/nulla/tempus/vivamus.jpg

The corresponding GROK expression is as follows:

```
%{YEAR:log_year}-%{MONTHNUM:log_month}-
%{MONTHDAY:log_day}%{USERNAME:username}
```

```
%{WINDOWSMAC:mac_address}%{URIPATH:referer_url}
```

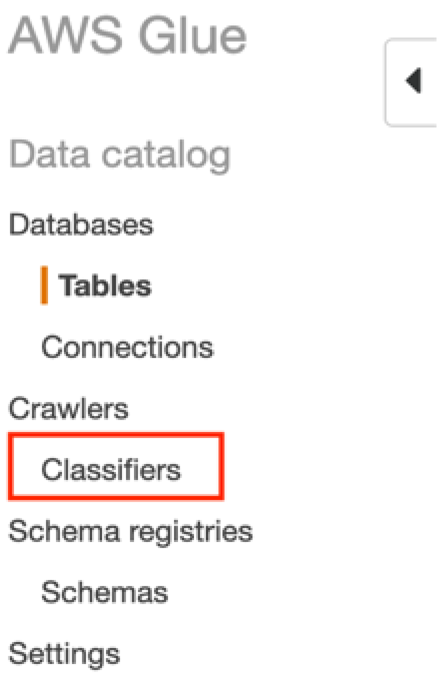
When working with a GROK pattern, you can use many built-in patterns that AWS Glue provides, or you can define your own.

Creating a custom classifier

Let's look at how to create a custom classifier from the previous GROK expression. Keep in mind that you can also create JSON, CSV, or XML-based custom classifiers, but we are limiting the scope of this document to a GROK-based example.

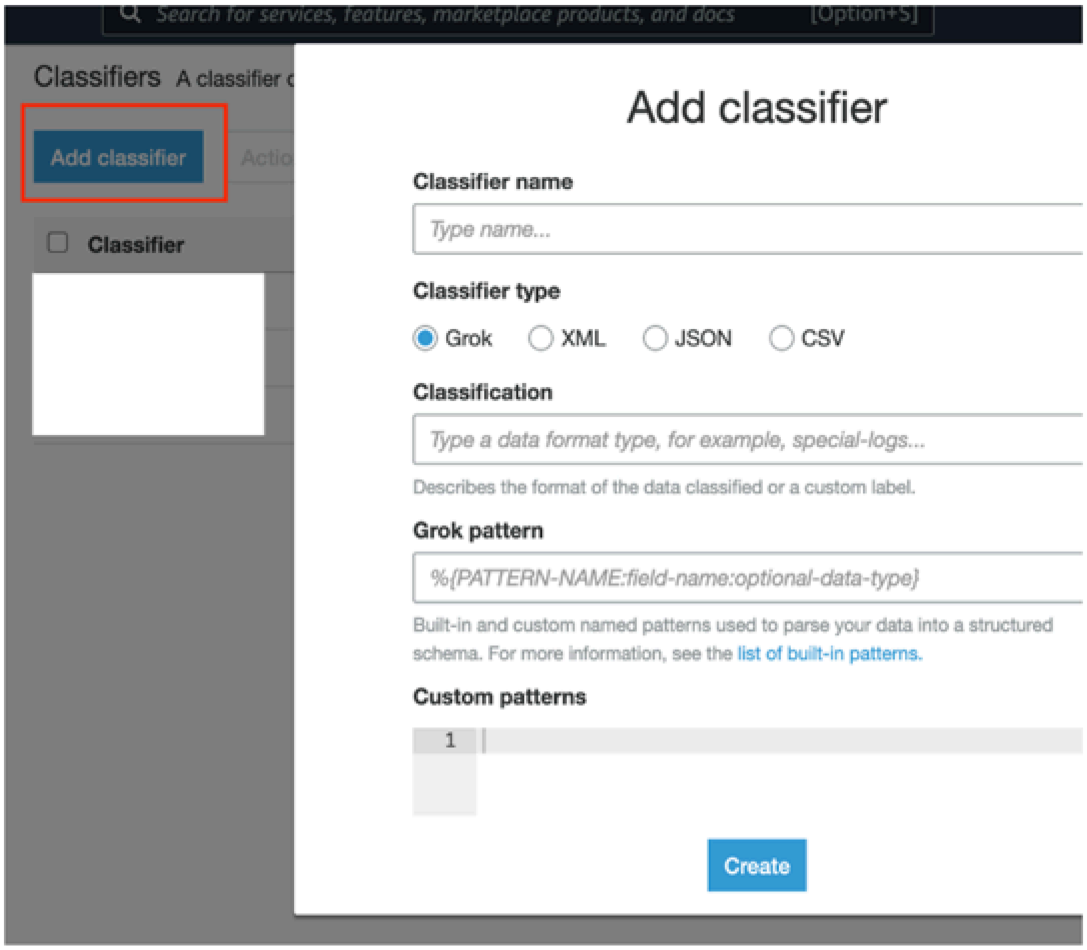
To create a custom classifier:

1. From the AWS Glue console, choose **Classifiers**.



*From the AWS Glue console, choose **Classifiers***

2. Choose **Add Classifier** and use the form to add the details.


Search for services, features, marketplace products, and docs [Option+S]
Classifiers A classifier
Add classifier
Actio
Classifier

Add classifier

Classifier name

Type name...

Classifier type

[✓ Grok] XML JSON CSV

Classification

Type a data format type, for example, special-logs...

Describes the format of the data classified or a custom label.

Grok pattern

%{PATTERN-NAME:field-name:optional-data-type}

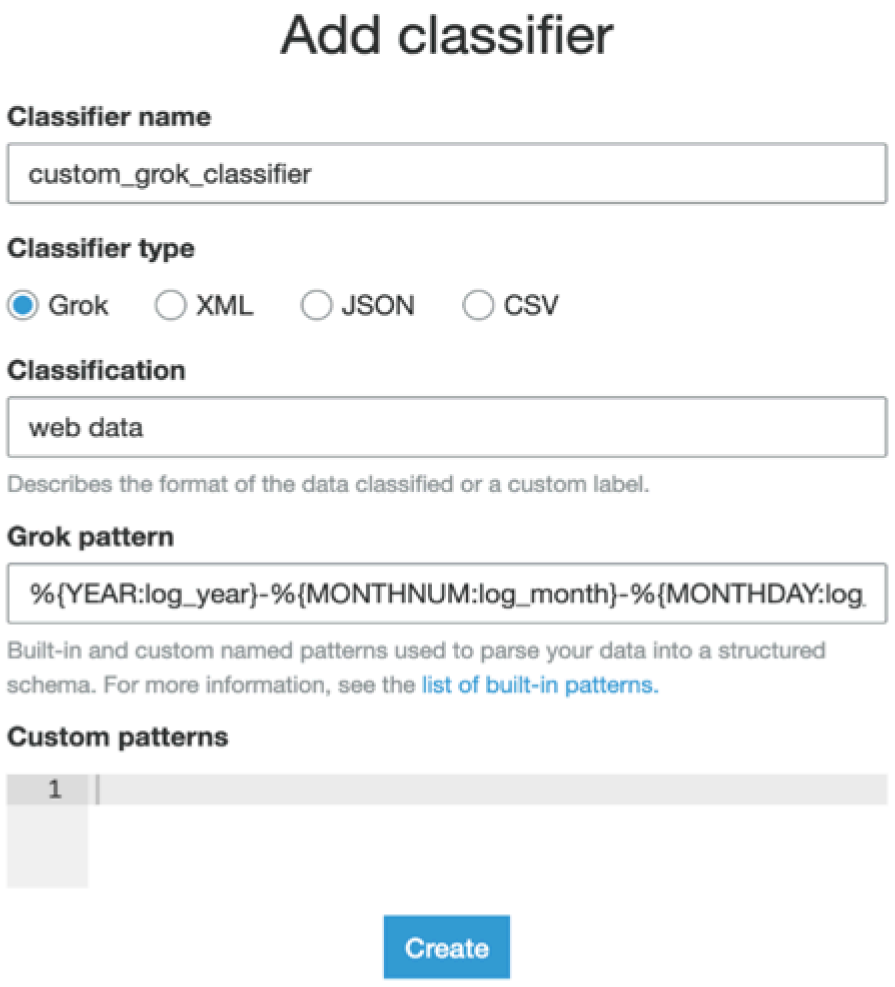
Built-in and custom named patterns used to parse your data into a structured schema. For more information, see the list of built-in patterns.

Custom patterns

Create

*From the AWS Glue console, choose **Add classifier***

3. Options in the form vary based on our choice of the classifier type. In this case, use the Grok Classifier. Following is an instance of the form updated to meet our parsing requirements. Choose **Create** to create the classifier.



Add classifier

Classifier name: custom_grok_classifier

Classifier type: [✓ Grok] XML JSON CSV

Classification: web data

Describes the format of the data classified or a custom label.

Grok pattern: %{YEAR:log_year}-%{MONTHNUM:log_month}-%{MONTHDAY:log

Built-in and custom named patterns used to parse your data into a structured schema. For more information, see the list of built-in patterns.

Custom patterns:

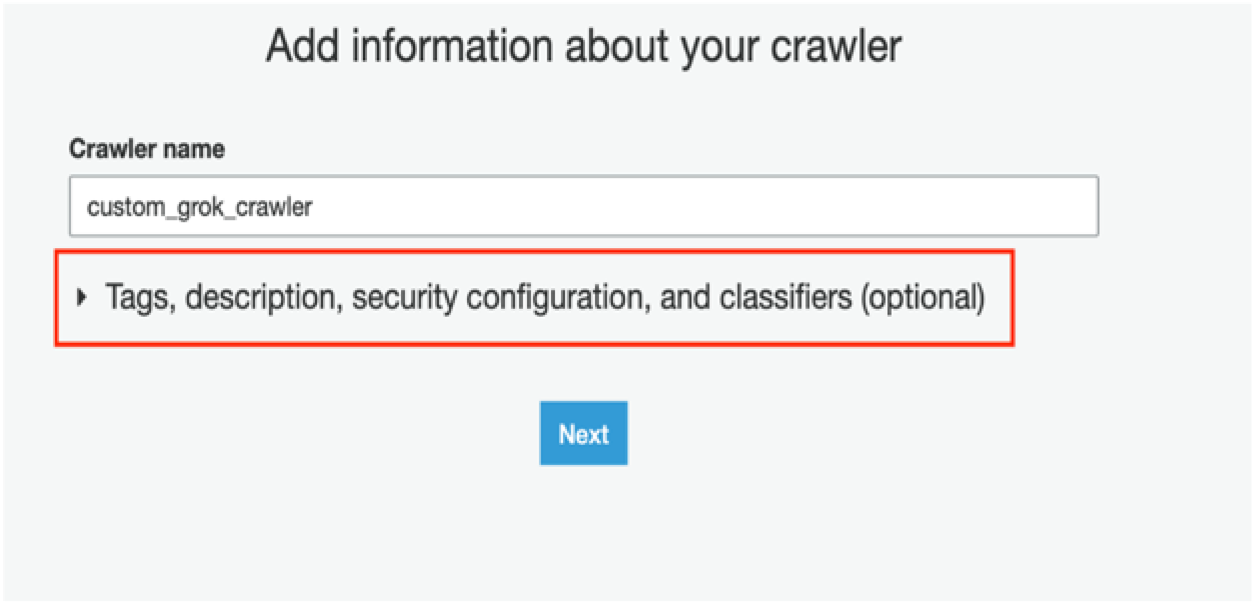
Fill in the forms to create the classifier

Adding the classifier to a crawler

Now that we have created the classifier, the next step is to attach this to a crawler.

To add the classifier to a crawler:

1. On the **Create crawler** window in the AWS Glue console, choose and expand the **Tags, description, security configuration, and classifiers (optional)** section.

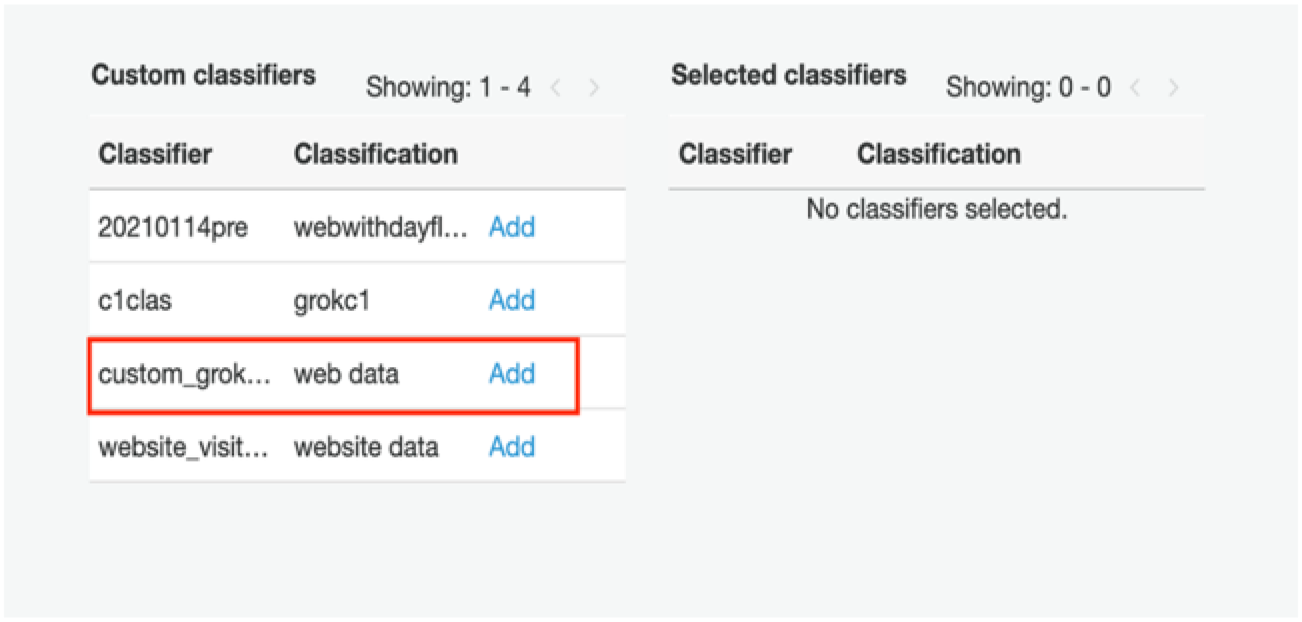

Add information about your crawler

Crawler name: custom_grok_crawler

▸ Tags, description, security configuration, and classifiers (optional)
Next

*Choose and expand the **Tags, description, security configuration, and classifiers (optional)** section.*

2. Scroll down to the **Classifiers** section, and choose **Add** (close to the classifier we just created).


Custom classifiers Showing: 1 - 4

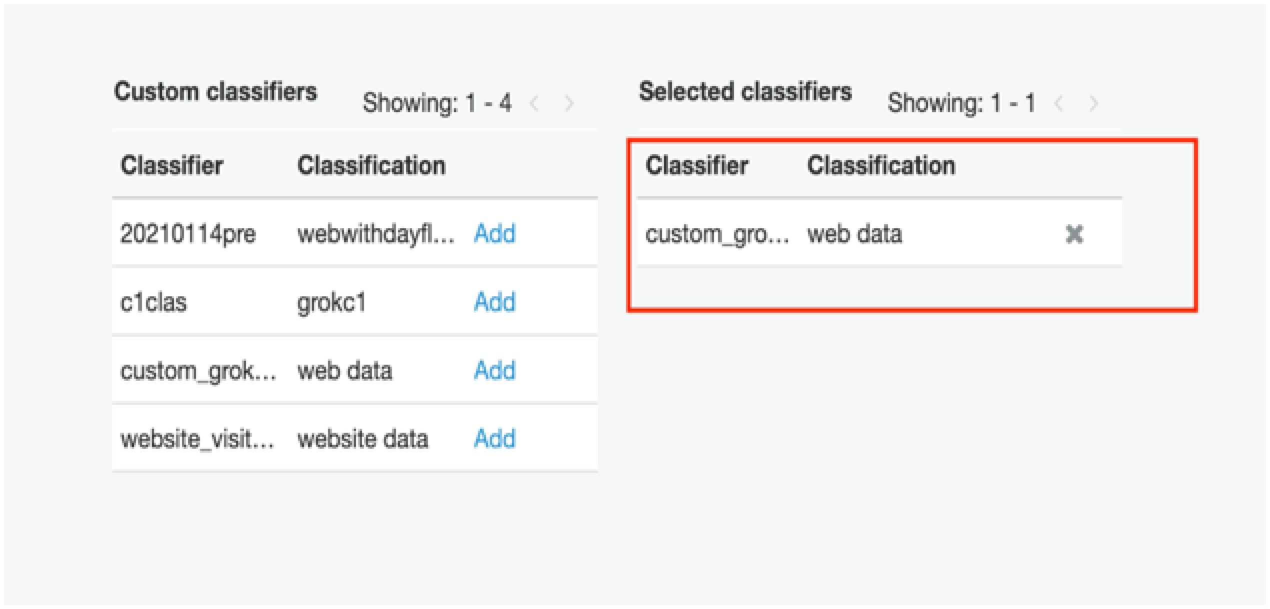
Classifier	Classification	
20210114pre	webwithdayfl...	Add
c1clas	grokc1	Add
custom_grok...	web data	Add
website_visit...	website data	Add

Selected classifiers Showing: 0 - 0

Classifier	Classification
No classifiers selected.	

*Scroll down to the **Classifiers** section and choose **Add***

The classifier should appear on the right side of the screen. You can complete the remaining crawler configurations, run it, and observe the target table it created:



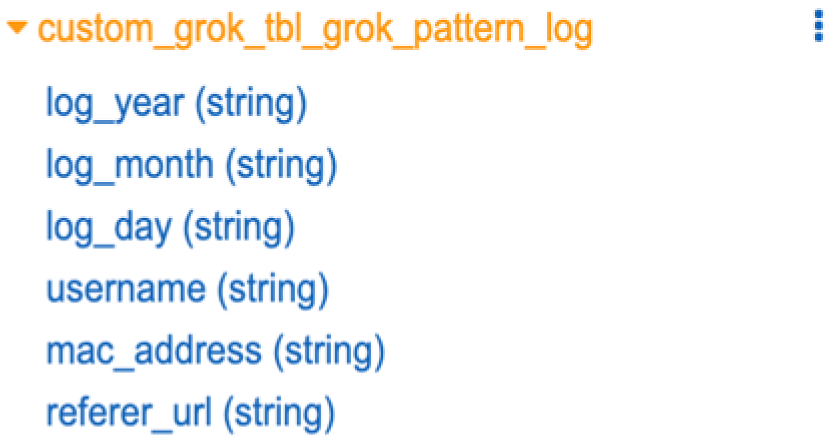
Custom classifiers Showing: 1 - 4

Classifier	Classification	
20210114pre	webwithdayfl...	Add
c1clas	grokc1	Add
custom_grok...	web data	Add
website_visit...	website data	Add

Selected classifiers Showing: 1 - 1

Classifier	Classification	
custom_gro...	web data	✖

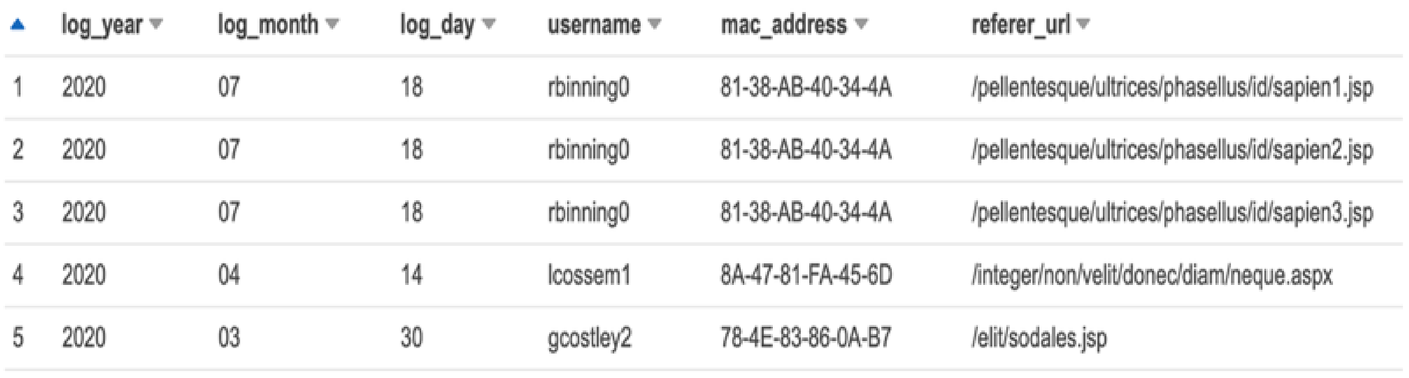
The classifier appears



custom_grok_tbl_grok_pattern_log

- log_year (string)
- log_month (string)
- log_day (string)
- username (string)
- mac_address (string)
- referer_url (string)

Schema of log data identified by classifier



	log_year	log_month	log_day	username	mac_address	referer_url
1	2020	07	18	rbinning0	81-38-AB-40-34-4A	/pellentesque/ultrices/phasellus/id/sapien1.jsp
2	2020	07	18	rbinning0	81-38-AB-40-34-4A	/pellentesque/ultrices/phasellus/id/sapien2.jsp
3	2020	07	18	rbinning0	81-38-AB-40-34-4A	/pellentesque/ultrices/phasellus/id/sapien3.jsp
4	2020	04	14	lcossem1	8A-47-81-FA-45-6D	/integer/non/velit/donec/diam/neque.aspx
5	2020	03	30	gcostley2	78-4E-83-86-0A-B7	/elit/sodales.jsp

Parsed log data

Incremental data pipeline

In the modern world of data engineering, one of the most common requirements is to store the data in its raw format and enabling a variety of consumption patterns (analytics, reporting, search, ML, and so on) on it. The data being ingested is typically of two types:

- Immutable data such as social network feeds, Internet of Things (IoT) sensor data, log files, and so on.
- Mutable data that is updated or deleted in transactional systems such as enterprise resource planning (ERP) or online transaction processing (OLTP) databases.

The need for data in its raw format leads to a huge volume of data being processed and engineered in an integration solution. Loading data incrementally (or delta) in the form of batches after an initial full data load is a widely accepted approach for such scenarios. The idea is to identify and extract only the newly added or updated records in tables in a source system instead of dealing with the entire table data. It reduces the volume of data being moved/processed during each load and results in efficient processing of data pipelines. Following are some of the ways of loading data incrementally.

- **Change tracking/CDC** — Depending on the type of source database, one of the most efficient way of extracting delta records from source system is by enabling change data capture (CDC), or change tracking. It records the changes in a table at the most granular level (insert/update/delete) and allows you to store the entire history of changes and transactions in a data lake or data warehouse. While AWS Glue doesn't support extracting data using CDC, AWS Data Migration Service (AWS DMS) is the recommended service for this purpose. Once the delta records are exported to the data lake or stage tables by AWS DMS, AWS Glue can then load them into a data warehouse efficiently (refer to the next section, AWS Glue job bookmarks).
- **AWS Glue job bookmarks** — If your source is an Amazon S3 data lake or a database that supports JDBC connection, AWS Glue job bookmarks are a great way to process delta files and records. They're an AWS Glue feature that removes all the overhead of implementing any algorithm to identify delta records. AWS Glue keeps track of bookmarks for each job. If you delete a job, you also delete the job bookmark. If for some reason, you need to reprocess all or part of the data from previous job runs, you can pick a bookmark for Glue to start processing the data from that bookmark onward. If you need to re-process all data, you can disable job bookmarks.

Popular S3-based storage formats, including JSON, CSV, Apache Avro, XML, and JDBC sources, support job bookmarks. Starting with AWS Glue version 1.0, columnar storage formats such as Apache Parquet and ORC are also supported.

For S3 input sources, AWS Glue job bookmarks check the last modified time of the objects to verify which objects to reprocess. If there are new files that have arrived, or existing files changed, since your last job run, the files are reprocessed when the job is run again using a periodic AWS Glue job trigger or an S3 trigger notification.

For JDBC sources, job bookmarks require source tables to either have a primary key column(s) or a column(s) with incrementing values, which need to be specified in the source options. The AWS Glue bookmark checks for newly added records based on the columns provided and processes the delta records.

- **Limitation** — For JDBC sources, job bookmarks can capture only newly added rows and it needs to be processed in batches. This behavior does not apply to source tables stored on S3.

  For examples of implementing job bookmark, refer to the blog post Load data incrementally and optimized Parquet writer with AWS Glue.
- **High watermark** — If the source database system doesn't have CDC feature at all, then high watermark is a classic way of extracting delta records. It is the process of storing data load status and its timestamp into metadata tables. During the ETL load, it calculates the maximum value of load timestamp (high watermark) from metadata tables and filters the data being extracted. It does require a create timestamp (new records) and update timestamp (updated records) field in each of the table in source system to allow filtering on them based on high watermark timestamp. While this process requires creation and maintenance of metadata tables, it provides great flexibility of rewinding or reprocessing data from a time in past with a simple update of the high watermark value. These high watermark filters can easily be embedded into the SQL scripts in AWS Glue ETL jobs to extracting delta records.
- **Use cases** — Source system is a database that doesn't have CDC/change tracking available, and updated records must be processed.
- **Event driven** — In the modern era, event driven data pipelines have become really popular especially for streaming and micro batch (< 15 min) data load patterns where the data pipeline is decoupled. The first part is to extract data from source system and load via streaming to S3 data lake within seconds. The second part is to load the data from the data lake to the data warehouse via event-driven triggers. This eliminates the need to identify delta records based on a column or timestamp, and instead relies on object/bucket level events such as put, copy, and

delete to process the data, resulting in a seamless process with very less overhead. Both S3 and Amazon EventBridge support this feature, where an AWS Glue workflow or job loads the delta records to a target system as an incremental load.

Following are few use cases where the event-driven approach may be more suitable:

- Decoupled data pipelines that have an extract process (CDC/streaming) from source systems to the S3 data lake then use events to load data to the data warehouse.
- It's difficult to predict the frequency at which upstream systems generate data. Once generated, it needs to load to the target system as soon as possible.

The following table provides considerations using different mechanism for incremental data loads.

Table 8 — Incremental data

Source	**CDC/change tracking**	**High watermark**	**Job bookmark for S3 source**	**Job bookmark for JDBC source**	**Event driven**
Source system is a database	Yes (CDC must be supported and enabled)	Yes	No	Yes (must support JDBC connection)	No
Source is S3	No	No	Yes	NA	Yes
Inserting new records	Yes	Yes	Yes	Yes	Yes
Updating records	Yes	Yes (source table should have update timestamp column)	Yes	No	Yes

Source	CDC/change tracking	High watermark	Job bookmark for S3 source	Job bookmark for JDBC source	Event driven
Streaming datasets	Yes	No	No	No	Yes
Micro batches (< 15 min)	Yes	Yes	No	No	Yes
Batches (> 15 min)	Yes	Yes	Yes	Yes	Yes
Proprietary feature	Yes	No	Yes	Yes	Yes

Conclusion

In this whitepaper, we explained some of the best practices for building a performance efficient and cost optimized data pipeline using AWS Glue, considering the guidance from AWS Well-Architecture framework. We also looked at different programming languages that are available for building your data pipelines, custom classifiers, and how AWS Glue helps you in building incremental data pipelines.

Contributors

Contributors to this document include:

- Durga Mishra, Sr. Solutions Architect, Amazon Web Services
- Arun A K, Solutions Architect, Amazon Web Services
- Narendra Gupta, Sr. Solutions Architect, Amazon Web Services
- Jay Palaniappan, Sr. Solutions Architect, Amazon Web Services
- Rajesh Agarwalla, Data Architect, Amazon Web Services

Further reading

For additional information, refer to:

- Snappy compressed format description
- Parquet compression definitions
- LanguageManual ORC
- Bucketing vs partitioning in the Amazon Athena user guide
- Top 10 Performance Tuning Tips for Amazon Athena (blog post)
- AWS Glue pricing
- Load data incrementally and optimized Parquet writer with AWS Glue (blog post)
- *AWS Glue Best Practices: Building an Operationally Efficient Data Pipeline* (AWS whitepaper)
- AWS Glue Best Practices: Building a Secure and Reliable Data Pipeline (AWS whitepaper)

Document revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Initial publication	Whitepaper published.	August 26, 2022

Notices

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.