aws

Best practices for performance tuning AWS Glue for Apache Spark jobs

: Best practices for performance tuning AWS Glue for Apache Spark jobs

Table of Contents

Best practices for performance tuning AWS Glue for Apache Spark jobs

Roman Myers, Takashi Onikura, and Noritaka Sekiyama, Amazon Web Services (AWS)

December 2023 (document history)

AWS Glue provides different options for tuning performance. This guide defines key topics for tuning AWS Glue for Apache Spark. It then provides a baseline strategy for you to follow when tuning these AWS Glue for Apache Spark jobs. Use this guide to learn how to identify performance problems by interpreting metrics available in AWS Glue. Then incorporate strategies to address these problems, maximizing performance and minimizing costs.

This guide covers the following tuning practices:

- Scale cluster capacity
- Use the latest AWS Glue version
- Reduce the amount of data scan
- Parallelize tasks
- Minimize planning overhead
- Optimize shuffles
- Optimize user-defined functions

Key topics in Apache Spark

This section explains Apache Spark basic concepts and key topics for tuning AWS Glue for Apache Spark performance. It's important to understand these concepts and topics before discussing real-world tuning strategies.

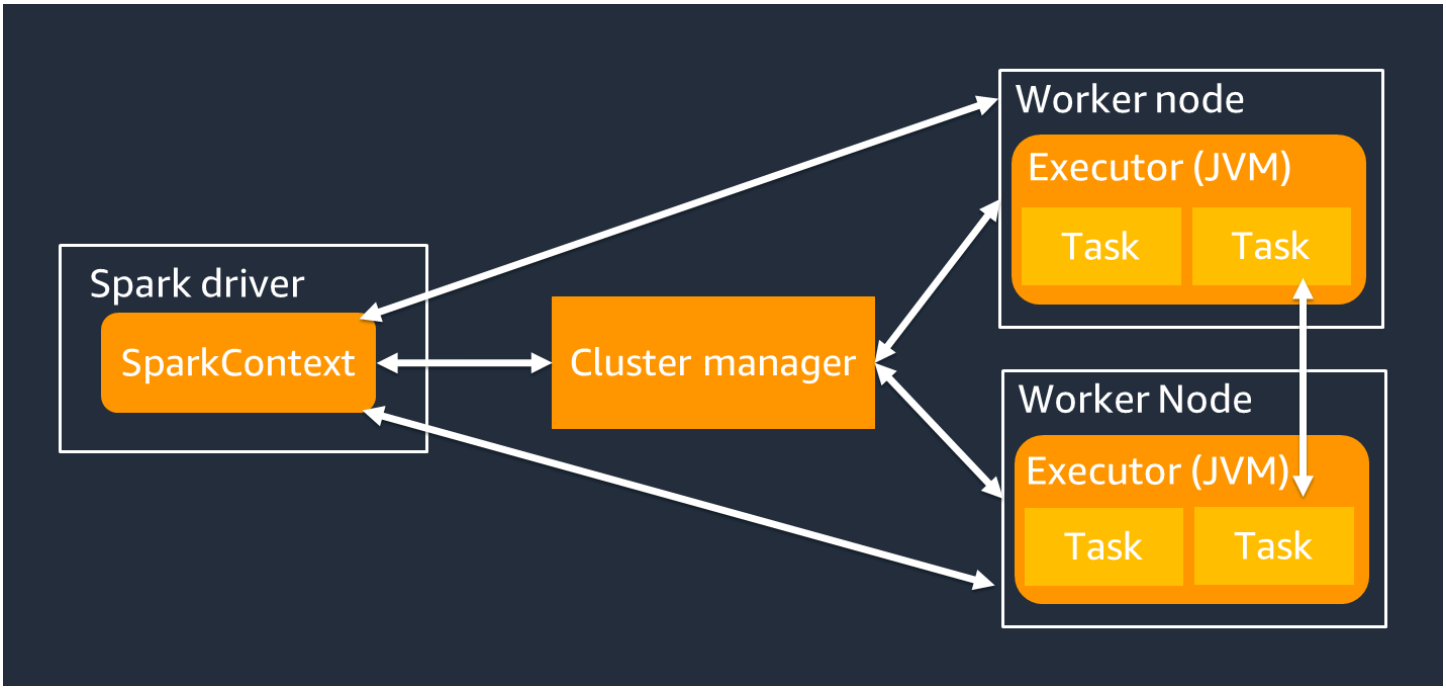
Architecture

The Spark driver is mainly responsible for splitting your Spark application up into tasks that can be accomplished on individual workers. The Spark driver has the following responsibilities:

- Running `main()` in your code
- Generating execution plans
- Provisioning Spark executors in conjunction with cluster manager, which manages resources on the cluster
- Scheduling tasks and requesting tasks for Spark executors
- Managing task progress and recovery

You use a `SparkContext` object to interact with the Spark driver for your job run.

A Spark executor is a worker for holding data and running tasks that are passed from the Spark driver. The number of Spark executors will go up and down with the size of your cluster.



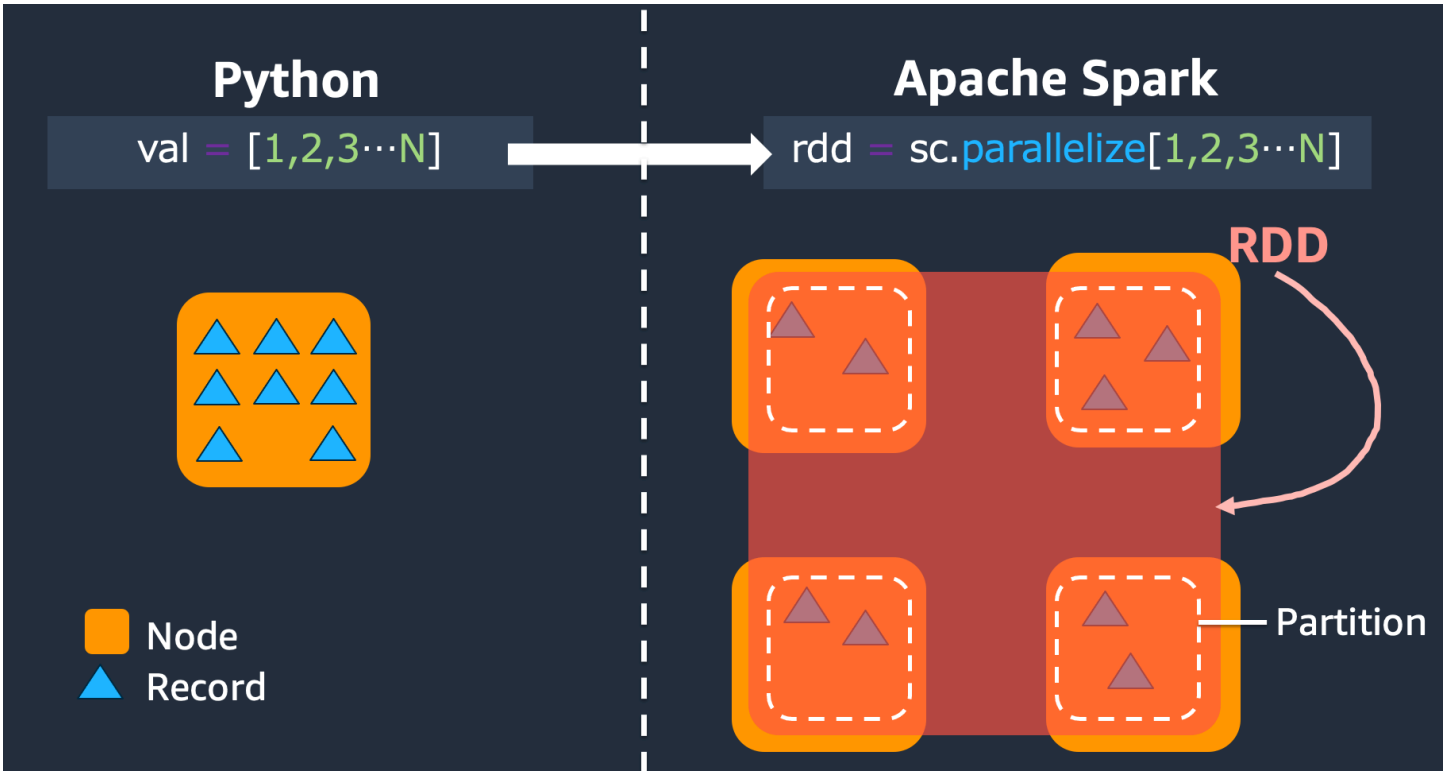
Note

A Spark executor has multiple slots so that multiple tasks to be processed in parallel. Spark supports one task for each virtual CPU (vCPU) core by default. For example, if an executor has four CPU cores, it can run four concurrent tasks.

Resilient distributed dataset

Spark does the complex job of storing and tracking large data sets across Spark executors. When you write code for Spark jobs, you don't need to think about the details of storage. Spark provides the *resilient distributed dataset (RDD)* abstraction, which is a collection of elements that can be operated on in parallel and can be partitioned across the Spark executors of the cluster.

The following figure shows the difference in how to store data in memory when a Python script is run in its typical environment and when it's run in the Spark framework (*PySpark*).



- **Python** – Writing `val = [1,2,3...N]` in a Python script keeps the data in memory on the single machine where the code is running.
- **PySpark** – Spark provides the RDD data structure to load and process data distributed across memory on multiple Spark executors. You can generate an RDD with code such as `rdd = sc.parallelize[1,2,3...N]`, and Spark can automatically distribute and hold data in memory across multiple Spark executors.

  In many AWS Glue jobs, you use RDDs through AWS Glue *DynamicFrames* and Spark *DataFrames*. These are abstractions that allow you to define the schema of data in an RDD and perform higher-level tasks with that additional information. Because they use RDDs internally, data is transparently distributed and loaded to multiple nodes in the following code:

  - DynamicFrame

    ```
    dyf= glueContext.create_dynamic_frame.from_options(
        's3', {"paths": [ "s3://<YourBucket>/<Prefix>/"]},
        format="parquet",
        transformation_ctx="dyf"
    )
    ```

  - DataFrame

```
df = spark.read.format("parquet")
    .load("s3://<YourBucket>/<Prefix>")
```

An RDD has following features:

- RDDs consist of data divided into multiple parts called *partitions*. Each Spark executor stores one or more partitions in memory, and the data is distributed across multiple executors.
- RDDs are *immutable*, meaning they can't be changed after they're created. To change a DataFrame, you can use *transformations*, which are defined in the following section.
- RDDs replicate data across available nodes, so they can automatically recover from node failures.

Lazy evaluation

RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset.

- **Transformations** – Because RDDs are *immutable,* you can change them only by using a transformation.

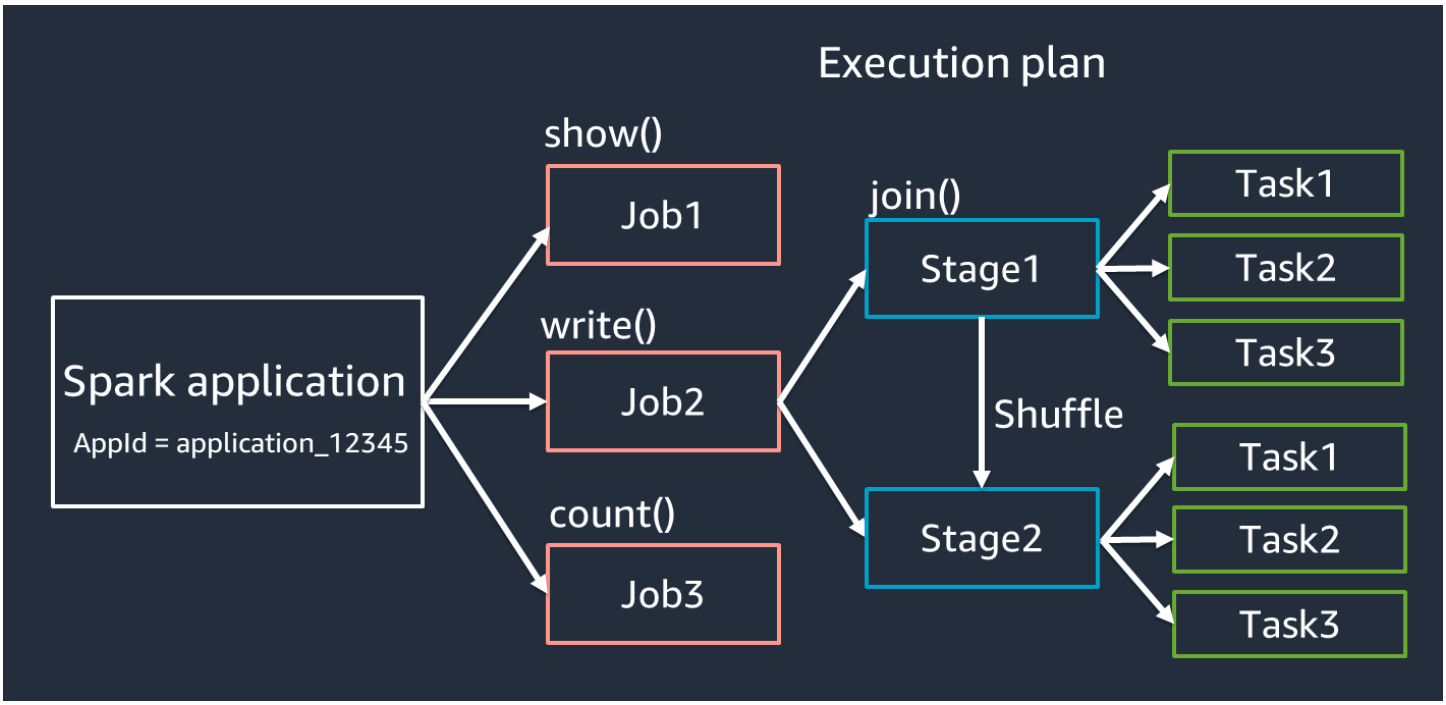
  For example, `map` is a transformation that passes each dataset element through a function and returns a new RDD representing the results. Notice that the `map` method doesn't return an output. Spark stores the abstract transformation for the future, rather than letting you interact with the result. Spark will not act on transformations until you call an action.
- **Actions** – Using transformations, you build up your logical transformation plan. To initiate the computation, you run an action such as `write`, `count`, `show`, or `collect`.

  All transformations in Spark are *lazy*, in that they don't compute their results right away. Instead, Spark remembers a series of transformations applied to some base dataset, such as Amazon Simple Storage Service (Amazon S3) objects. The transformations are computed only when an action requires a result to be returned to the driver. This design enables Spark to run more efficiently. For example, consider the situation where a dataset created through the `map` transformation is consumed only by a transformation that substantially reduces the number of rows, such as `reduce`. You can then pass the smaller dataset that has undergone both transformations to the driver, instead of passing the larger mapped dataset.

Terminology of Spark applications

This section covers Spark application terminology. The Spark driver creates an *execution plan* and controls the behavior of applications in several abstractions. The following terms are important for development, debugging, and performance tuning with the Spark UI.

- ***Application*** – Based on a Spark session (Spark context). Identified by a unique ID such as `<application_XXX>`.
- ***Jobs*** – Based on the actions created for an RDD. A job consists of one or more *stages*.
- ***Stages*** – Based on the *shuffles* created for an RDD. A stage consists of one or more tasks. The shuffle is Spark's mechanism for redistributing data so that it's grouped differently across RDD partitions. Certain transformations, such as `join()`, require a shuffle. Shuffle are discussed in more detail in the Optimize shuffles tuning practice.
- ***Tasks*** – A task is the minimum unit of processing scheduled by Spark. Tasks are created for each RDD partition, and the number of tasks is the maximum number of simultaneous executions in the stage.

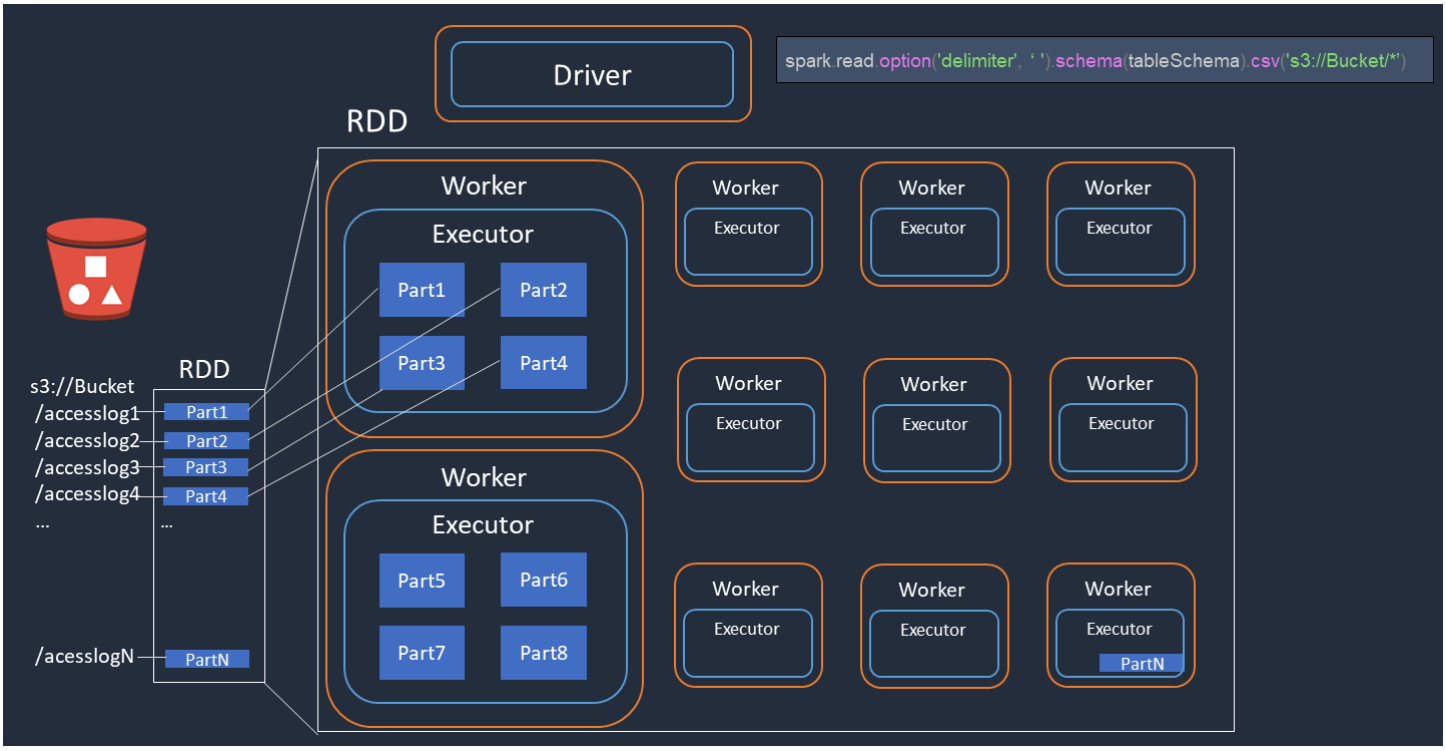


> **Note**
>
> Tasks are the most important thing to consider when optimizing parallelism. The number of tasks scales with the number of RDD

Parallelism

Spark parallelizes tasks for loading and transforming data.

Consider an example where you perform distributed processing of access log files (named `accesslog1 ... accesslogN`) on Amazon S3. The following diagram shows the distributed-processing flow.

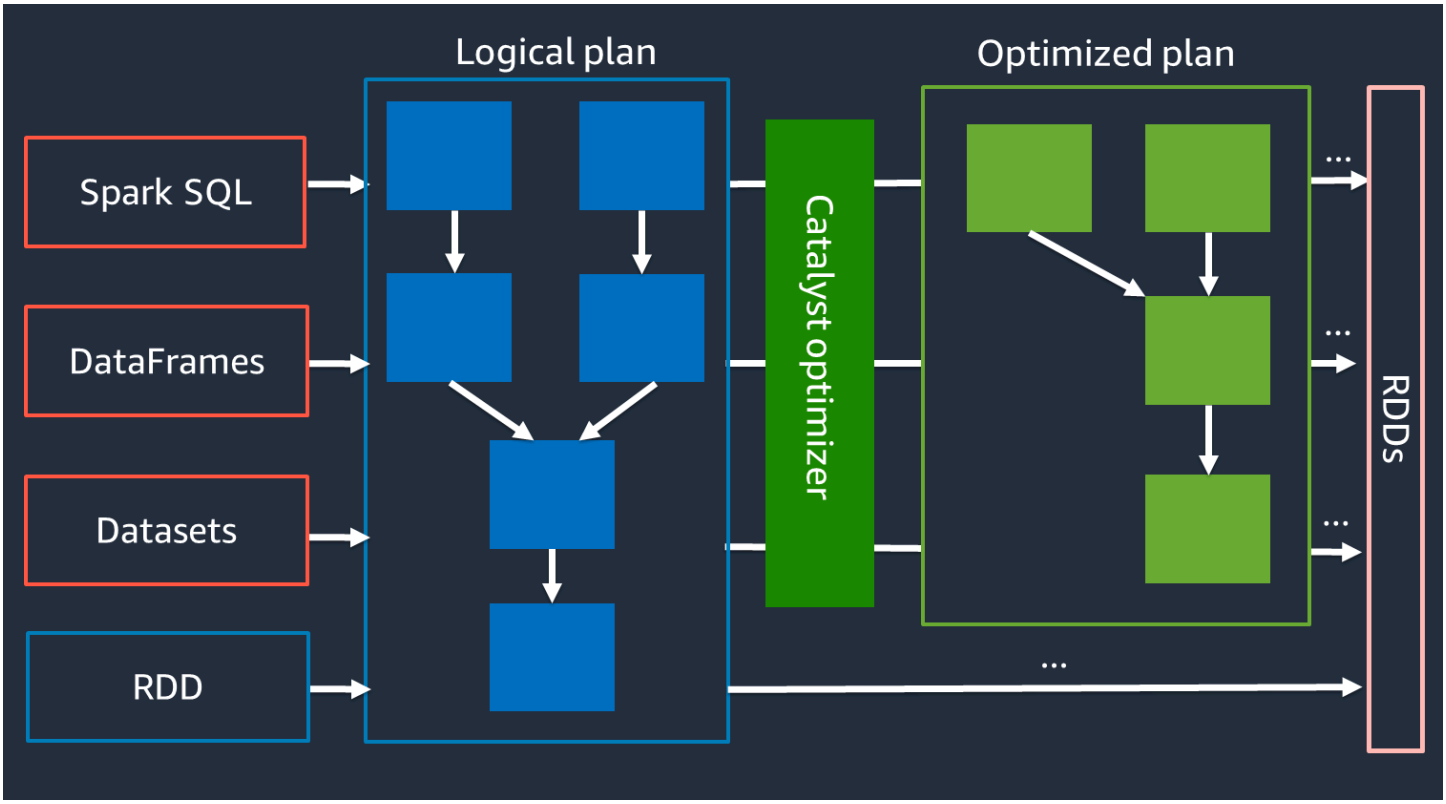


1. The Spark driver creates an execution plan for distributed processing across many Spark executors.
2. The Spark driver assigns tasks each executor based on the execution plan. By default, the Spark driver creates RDD partitions (each corresponding to a Spark task) for each S3 object (`Part1 ... N`). Then the Spark driver assigns tasks to each executor.
3. Each Spark task downloads its assigned S3 object and stores it in memory in the RDD partition. In this way, multiple Spark executors download and process their assigned task in parallel.

For more details about the initial number of partitions and optimization, see the Parallelize tasks section.

Catalyst optimizer

Internally, Spark uses an engine called Catalyst optimizer to optimize execution plans. Catalyst has a query optimizer that you can use when running high-level Spark APIs, such as Spark SQL, DataFrame, and Datasets, as described in the following diagram.



Because the Catalyst optimizer doesn't work directly with the RDD API, the high-level APIs are generally faster than the low-level RDD API. For complex joins, the Catalyst optimizer can significantly improve performance by optimizing the job run plan. You can see the optimized plan of your Spark job on the **SQL** tab of the Spark UI.

Adaptive Query Execution

The Catalyst optimizer performs runtime optimization through a process called *Adaptive Query Execution*. Adaptive Query Execution uses runtime statistics to re-optimize the run plan of the queries while your job is running. Adaptive Query Execution offers several solutions to performance

challenges, including coalescing post-shuffle partitions, converting sort-merge join to broadcast join, and skew join optimization, as described in the following sections.

Adaptive Query Execution is available in AWS Glue 3.0 and later, and it's enabled by default in AWS Glue 4.0 (Spark 3.3.0) and later. Adaptive Query Execution can be turned on and off by using `spark.conf.set("spark.sql.adaptive.enabled", "true")` in your code.

Coalescing post-shuffle partitions

This feature reduces RDD partitions (coalesce) after each shuffle based on the `map` output statistics. It simplifies the tuning of the shuffle partition number when running queries. You don't need to set a shuffle partition number to fit your dataset. Spark can pick the proper shuffle partition number at runtime after you have a large enough initial number of shuffle partitions.

Coalescing post-shuffle partitions is enabled when both `spark.sql.adaptive.enabled` and `spark.sql.adaptive.coalescePartitions.enabled` are set to true. For more information, see the Apache Spark documentation.

Converting sort-merge join to broadcast join

This feature recognizes when you are joining two datasets of substantially different size, and it adopts a more efficient join algorithm based on that information. For more details, see the Apache Spark documentation. Join strategies are discussed in the Optimize shuffles section.

Skew join optimization

Data skew is one of the most common bottlenecks for Spark jobs. It describes a situation in which data is skewed to specific RDD partitions (and consequently, specific tasks), which delays the overall processing time of the application. This can often downgrade the performance of join operations. The skew join optimization feature dynamically handles skew in sort-merge joins by splitting (and replicating if needed) skewed tasks into roughly even-sized tasks.

This feature is enabled when `spark.sql.adaptive.skewJoin.enabled` is set to true. For more details, see the Apache Spark documentation. Data skew is discussed further in the Optimize shuffles section.

Investigate performance issues by using the Spark UI

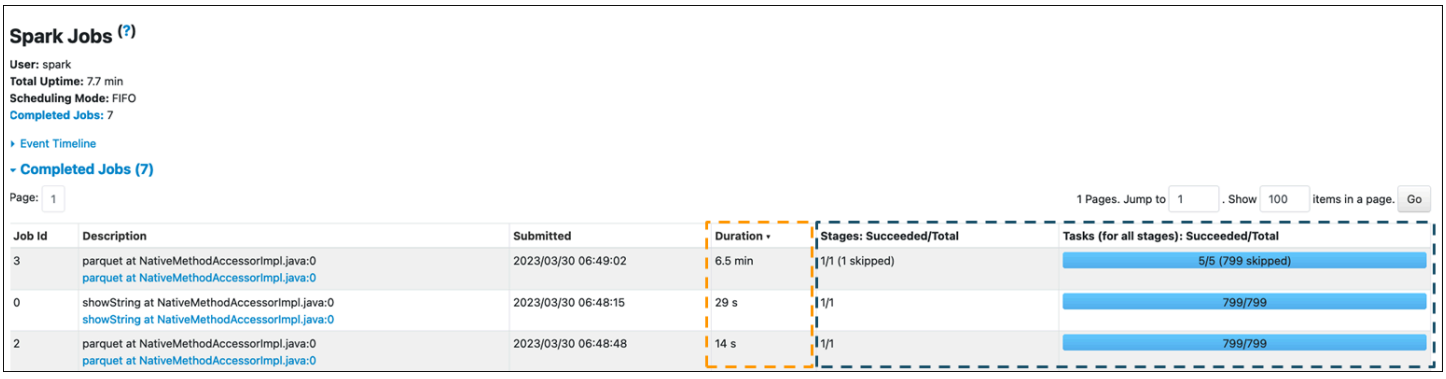
Before you apply any best practices to tune performance of your AWS Glue jobs, we highly recommended that you profile the performance and identify the bottlenecks. This will help you focus on the right things.

For quick analysis, Amazon CloudWatch metrics provide a basic view of your job metrics. The Spark UI provides a deeper view for performance tuning. To use the Spark UI with AWS Glue, you must enable Spark UI for your AWS Glue jobs. After you are familiar with the Spark UI, follow the strategies for tuning Spark job performance to identify and reduce the impact of bottlenecks based on your findings.

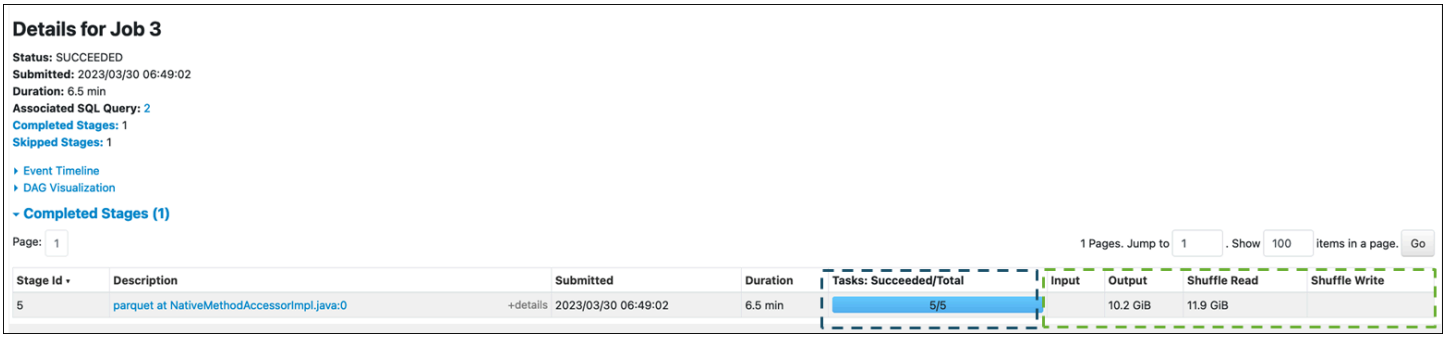
Identify bottlenecks by using the Spark UI

When you open the Spark UI, Spark applications are listed in a table. By default, an AWS Glue job's **App Name** is `nativespark-<Job Name>-<Job Run ID>`. Choose the target Spark app based on the job run ID to open the **Jobs** tab. Incomplete job runs, such as streaming job runs, are listed in **Show incomplete applications**.

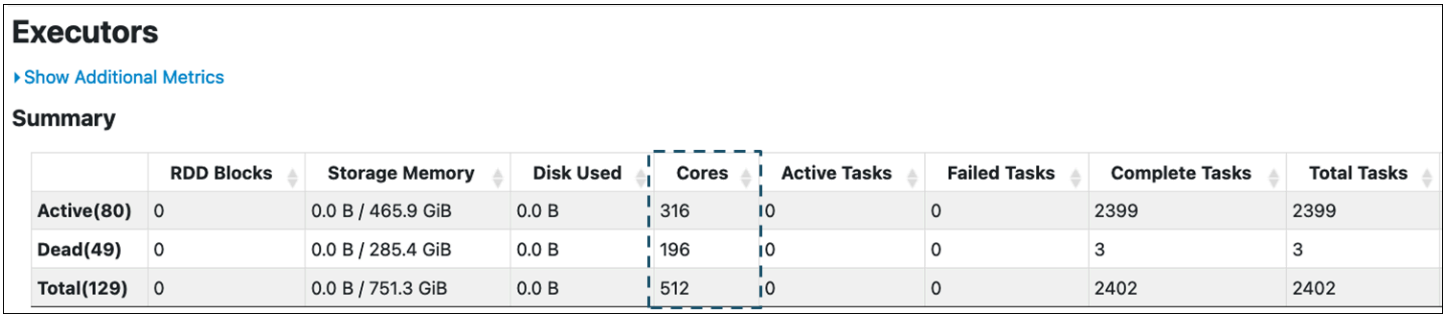
The **Jobs** tab shows a summary of all jobs in the Spark application. To determine any stage or task failures, check the total number of tasks. To find the bottlenecks, sort by choosing **Duration**. Drill down to the details of long-running jobs by choosing the link shown in the **Description** column.



The **Details for Job** page lists the stages. On this page, you can see overall insights such as duration, the number of succeeded and total tasks, the number of inputs and outputs, and the amount of shuffle read and shuffle write.



The **Executor** tab shows the Spark cluster capacity in detail. You can check the total number of cores. The cluster shown in the following screenshot contains 316 active cores and 512 cores in total. By default, each core can process one Spark task at the same time.



	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks
Active(80)	0	0.0 B / 465.9 GiB	0.0 B	316	0	0	2399	2399
Dead(49)	0	0.0 B / 285.4 GiB	0.0 B	196	0	0	3	3
Total(129)	0	0.0 B / 751.3 GiB	0.0 B	512	0	0	2402	2402

Based on the value 5/5 shown on the **Details for Job** page, stage 5 is the longest stage, but it uses only 5 cores out of 512. Because the parallelism for this stage is so low, but it takes a significant amount of time, you can identify it as a bottleneck. To improve performance, you want to understand why. To learn more about how to recognize and reduce the impact of common performance bottlenecks, see Strategies for tuning Spark job performance.

Strategies for tuning Spark job performance

When preparing to tune parameters, use the following best practices:

- Determine your performance goals before beginning to identify problems.
- Use metrics to identify problems before attempting to change tuning parameters.

For the most consistent results when tuning a job, develop a baseline strategy for your tuning work.

Baseline strategy for performance tuning

Generally, performance tuning is performed in the following workflow:

1. Determine performance goals.
2. Measure metrics.
3. Identify bottlenecks.
4. Reduce the impact of the bottlenecks.
5. Repeat steps 2-4 until you achieve the intended target.

First, determine your performance goals. For example, one of your goals might be to complete the run of an AWS Glue job within 3 hours. After you define your goals, measure job performance metrics. Identify trends in metrics and bottlenecks to meet the goals. In particular, identifying bottlenecks is most important for troubleshooting, debugging, and performance tuning. During the run of a Spark application, Spark records the status and statistics of each task in the Spark event log.

In AWS Glue, you can view Spark metrics through the Spark Web UI that's provided by the Spark history server. AWS Glue for Spark jobs can send Spark event logs to a location that you specify in Amazon S3. AWS Glue also provides an example AWS CloudFormation template and Dockerfile to start the Spark history server on an Amazon EC2 instance or your local computer, so you can use the Spark UI with event logs.

After you determine your performance goals and identify metrics to assess those goals, you can begin to identify and remediate bottlenecks by using the strategies in following sections.

Tuning practices for Spark job performance

You can use the following strategies for performance tuning AWS Glue for Spark jobs:

- AWS Glue resources:
  - Scale cluster capacity
  - Use the latest AWS Glue version
- Spark applications:
  - Reduce the amount of data scan
  - Parallelize tasks
  - Optimize shuffles
  - Minimize planning overhead
  - Optimize user-defined functions

Before you use these strategies, you must have access to metrics and configuration for your Spark job. You can find this information in the AWS Glue documentation.

From the AWS Glue resource perspective, you can achieve performance improvements by adding AWS Glue workers and using the latest AWS Glue version.

From an Apache Spark application perspective, you have access to several strategies that can improve performance. If unnecessary data is loaded into the Spark cluster, you can remove it to reduce the amount of loaded data. If you have underused Spark cluster resources and you have low data I/O, you can identify tasks to parallelize. You might also want to optimize heavy data transfer operations such as joins if they are taking substantial time. You can also optimize your job query plan or reduce the computational complexity of individual Spark tasks.

To efficiently apply these strategies, you must identify when they are applicable by consulting your metrics. For more details, see each of the following sections. These techniques work not only for performance tuning but also for solving typical problems such as out-of-memory (OOM) errors.

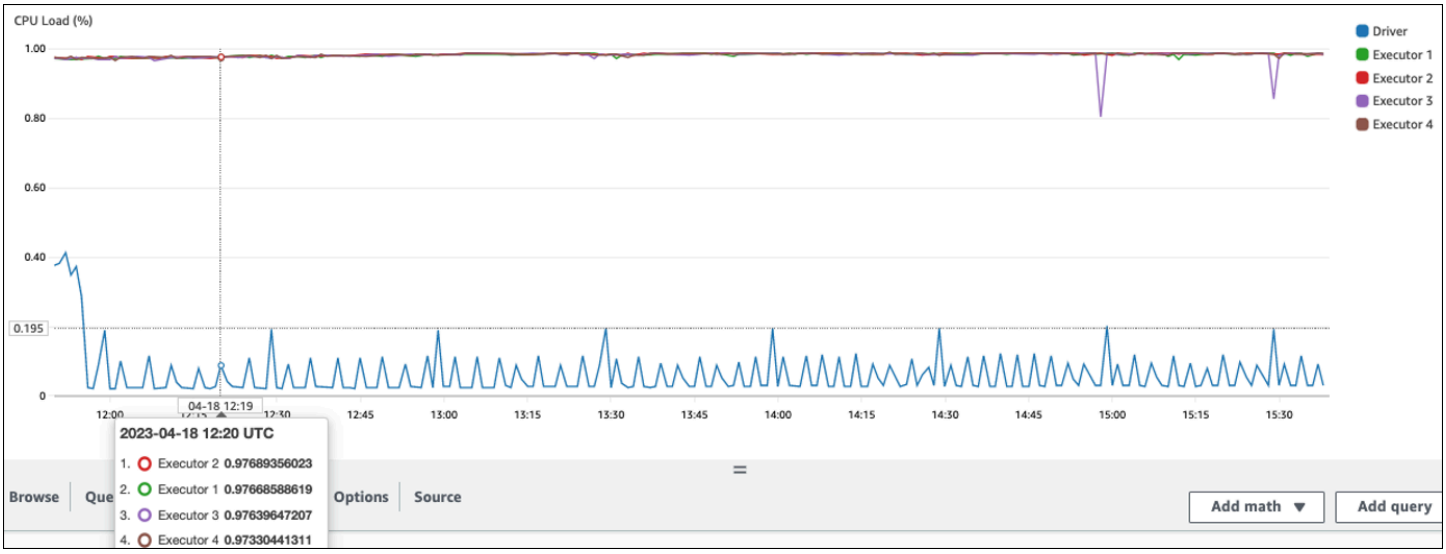
Scale cluster capacity

If your job is taking too much time, but executors are consuming sufficient resources and Spark is creating a large volume of tasks relative to available cores, consider scaling cluster capacity. To assess if this is appropriate, use the following metrics.

CloudWatch metrics

- Check **CPU Load** and **Memory Utilization** to determine whether executors are consuming sufficient resources.
- Check how long the job has run to assess whether the processing time is too long to meet your performance goals.

In the following example, four executors are running at more than 97 percent CPU load, but processing has not been completed after about three hours.

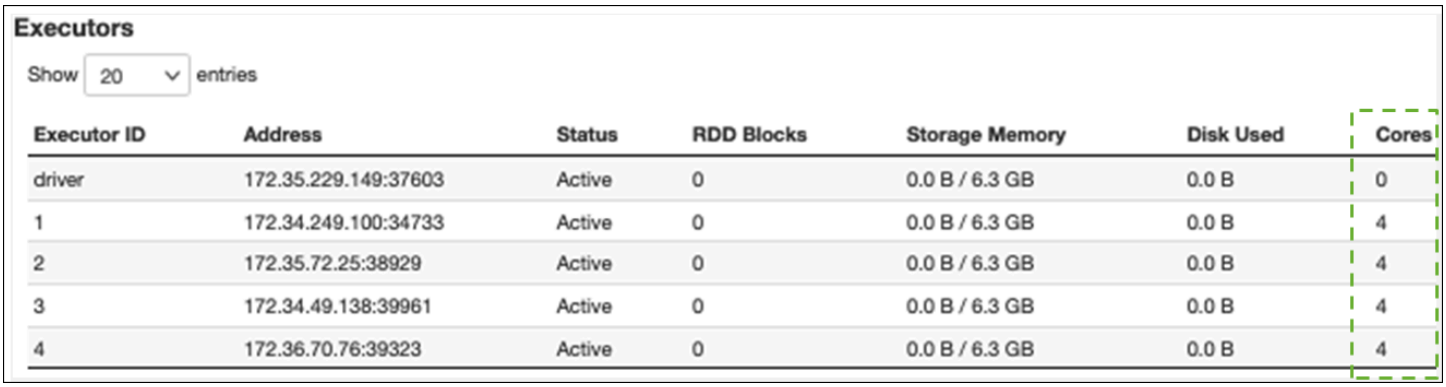


> **Note**
>
> If CPU load is low, you probably will not benefit from scaling cluster capacity.

Spark UI

On the **Job** tab or the **Stage** tab, you can see the number of tasks for each job or stage. In the following example, Spark has created `58100` tasks.

On the **Executor** tab, you can see the total number of executors and tasks. In the following screenshot, each Spark executor has four cores and can perform four tasks concurrently.



Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores
driver	172.35.229.149:37603	Active	0	0.0 B / 6.3 GB	0.0 B	0
1	172.34.249.100:34733	Active	0	0.0 B / 6.3 GB	0.0 B	4
2	172.35.72.25:38929	Active	0	0.0 B / 6.3 GB	0.0 B	4
3	172.34.49.138:39961	Active	0	0.0 B / 6.3 GB	0.0 B	4
4	172.36.70.76:39323	Active	0	0.0 B / 6.3 GB	0.0 B	4

In this example, the number of Spark tasks (`58100`) is much larger than the 16 tasks that the executors can process concurrently (4 executors × 4 cores).

If you observe these symptoms, consider scaling the cluster. You can scale cluster capacity by using the following options:

- **Enable AWS Glue Auto Scaling** – Auto Scaling is available for your AWS Glue extract, transform, and load (ETL) and streaming jobs in AWS Glue version 3.0 or later. AWS Glue automatically adds and removes workers from the cluster depending on the number of partitions at each stage or the rate at which microbatches are generated on the job run.

  If you observe a situation where the number of workers does not increase even though Auto Scaling is enabled, consider adding workers manually. However, note that scaling manually for one stage might cause many workers to be idle during later stages, costing more for zero performance gain.

  After you enable Auto Scaling, you can see the number of executors in the CloudWatch executor metrics. Use the following metrics to monitor the demand for executors in Spark applications:

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

  For more information about metrics, see Monitoring AWS Glue using Amazon CloudWatch metrics.
- **Scale out: Increase the number of AWS Glue workers** – You can manually increase the number of AWS Glue workers. Add workers only until you observe idle workers. At that point, adding

more workers will increase costs without improving results. For more information, see Parallelize tasks.

- **Scale up: Use a larger worker type** – You can manually change the instance type of your AWS Glue workers to use workers with more cores, memory, and storage. Larger worker types make it possible for you to vertically scale and run intensive data integration jobs, such as memory-intensive data transforms, skewed aggregations, and entity-detection checks involving petabytes of data.

  Scaling up also assists in cases where the Spark driver needs larger capacity—for instance, because the job query plan is quite large. For more information about worker types and performance, see the AWS Big Data Blog post Scale your AWS Glue for Apache Spark jobs with new larger worker types G.4X and G.8X.

  Using larger workers can also reduce the total number of workers needed, which increases performance by reducing shuffle in intensive operations such as join.

Use the latest AWS Glue version

We recommend using the latest AWS Glue version. There are several optimizations and upgrades built into each version that might automatically improve job performance. For example, AWS Glue 4.0 provides following new features:

- **New optimized Apache Spark 3.3.0 runtime** – AWS Glue 4.0 builds upon the Apache Spark 3.3.0 runtime, bringing comparable performance improvements to open source Spark. The Spark 3.3.0 runtime builds upon many of the innovations from Spark 2.x.
- **Enhanced Amazon Redshift connector** – AWS Glue 4.0 and later versions provide Amazon Redshift integration for Apache Spark. The integration builds on an existing open source connector and enhances it for performance and security. The integration helps applications perform up to 10 times faster. For more information, see the blog post about Amazon Redshift integration with Apache Spark.
- **SIMD-based execution for vectorized reads with CSV and JSON data** – AWS Glue version 3.0 and later versions add optimized readers that can significantly speed up overall job performance compared with row-based readers. For more information about CSV data, see Optimize read performance with vectorized SIMD CSV reader. For more information about JSON data, see Using vectorized SIMD JSON reader with Apache Arrow columnar format.

Each AWS Glue version will include upgrades of this sort, among many, including connectors, driver and library updates. For more information, see AWS Glue versions and Migrating AWS Glue jobs to AWS Glue version 4.0.

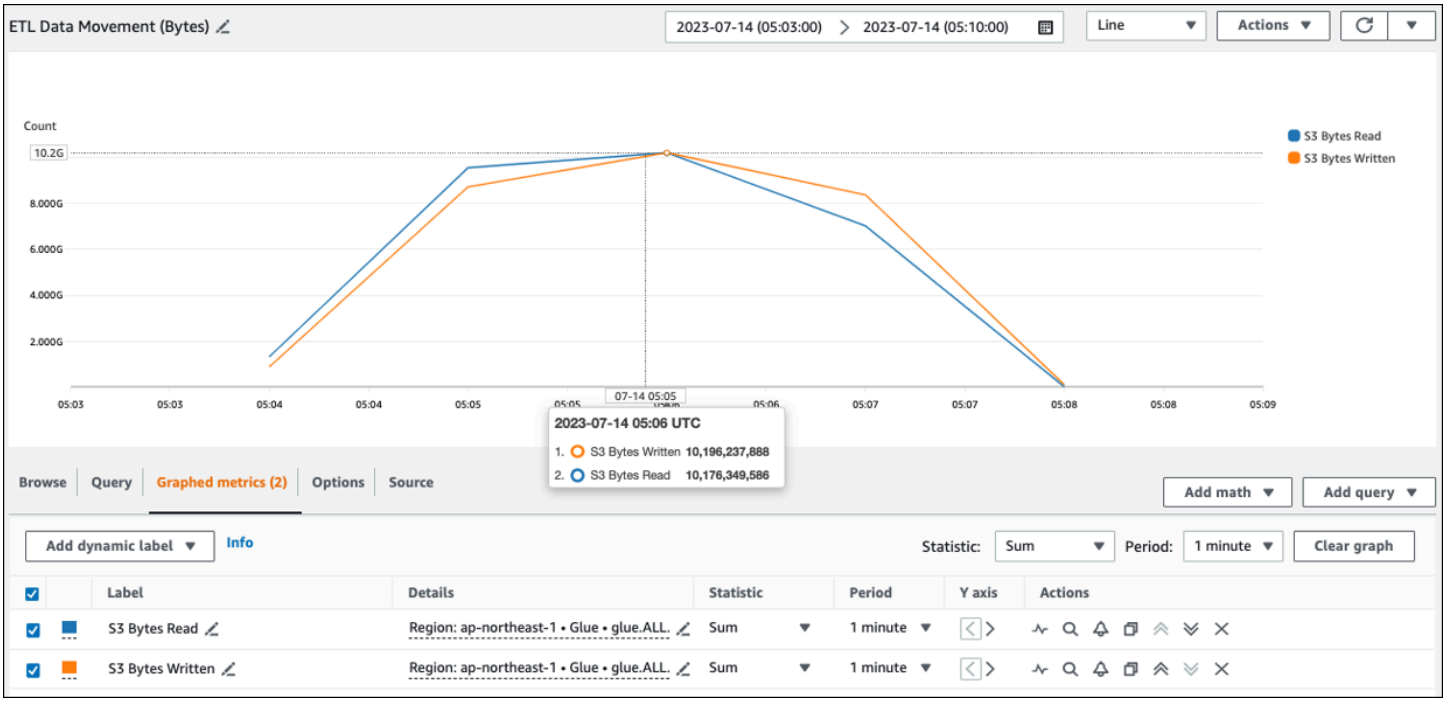
Reduce the amount of data scan

To begin, consider loading only the data that you need. You can improve performance just by reducing the amount of data loaded into your Spark cluster for each data source. To assess whether this approach is appropriate, use the following metrics.

You can check read bytes from Amazon S3 in CloudWatch metrics and more details in the Spark UI as described in the Spark UI section.

CloudWatch metrics

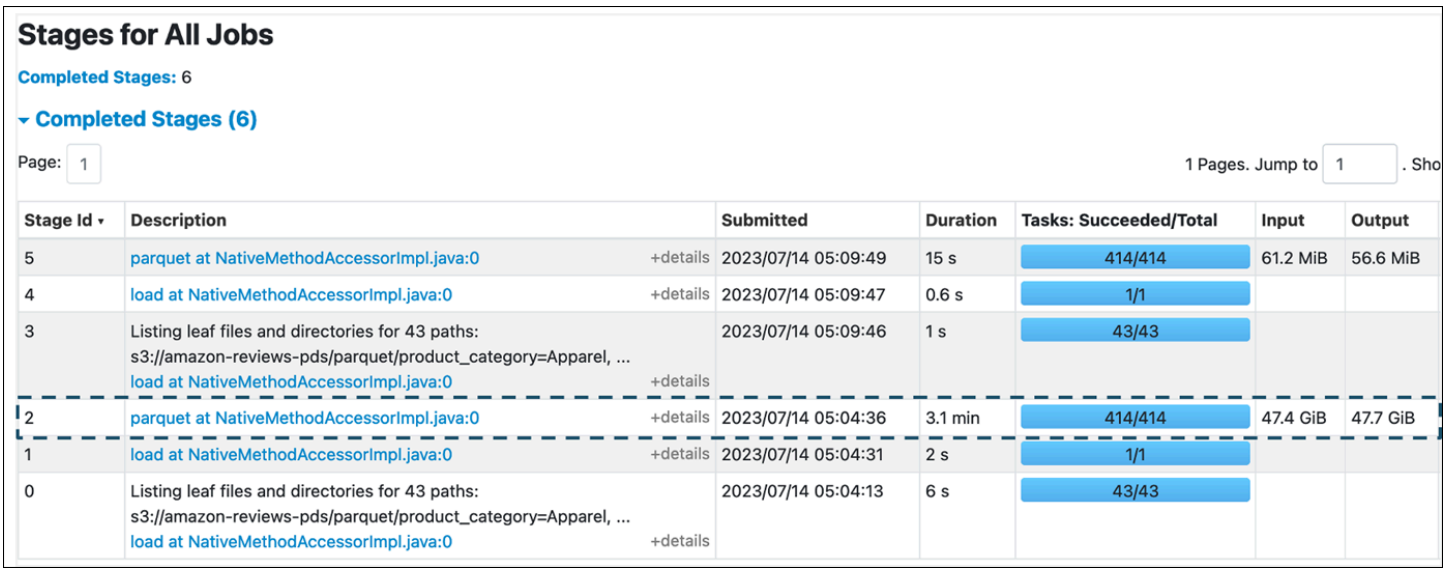
You can see the approximate read size from Amazon S3 in ETL Data Movement (Bytes). This metric shows the number of bytes read from Amazon S3 by all executors since the previous report. You can use it to monitor ETL data movement from Amazon S3, and you can compare reads to ingestion rates from external data sources.



If you observe a larger **S3 Bytes Read** data point than you expected, consider the following solutions.

Spark UI

On the **Stage** tab in the AWS Glue for Spark UI, you can see the **Input** and **Output** size. In the following example, stage 2 reads 47.4 GiB input and 47.7 GiB output, while stage 5 reads 61.2 MiB input and 56.6 MiB output.



Stages for All Jobs

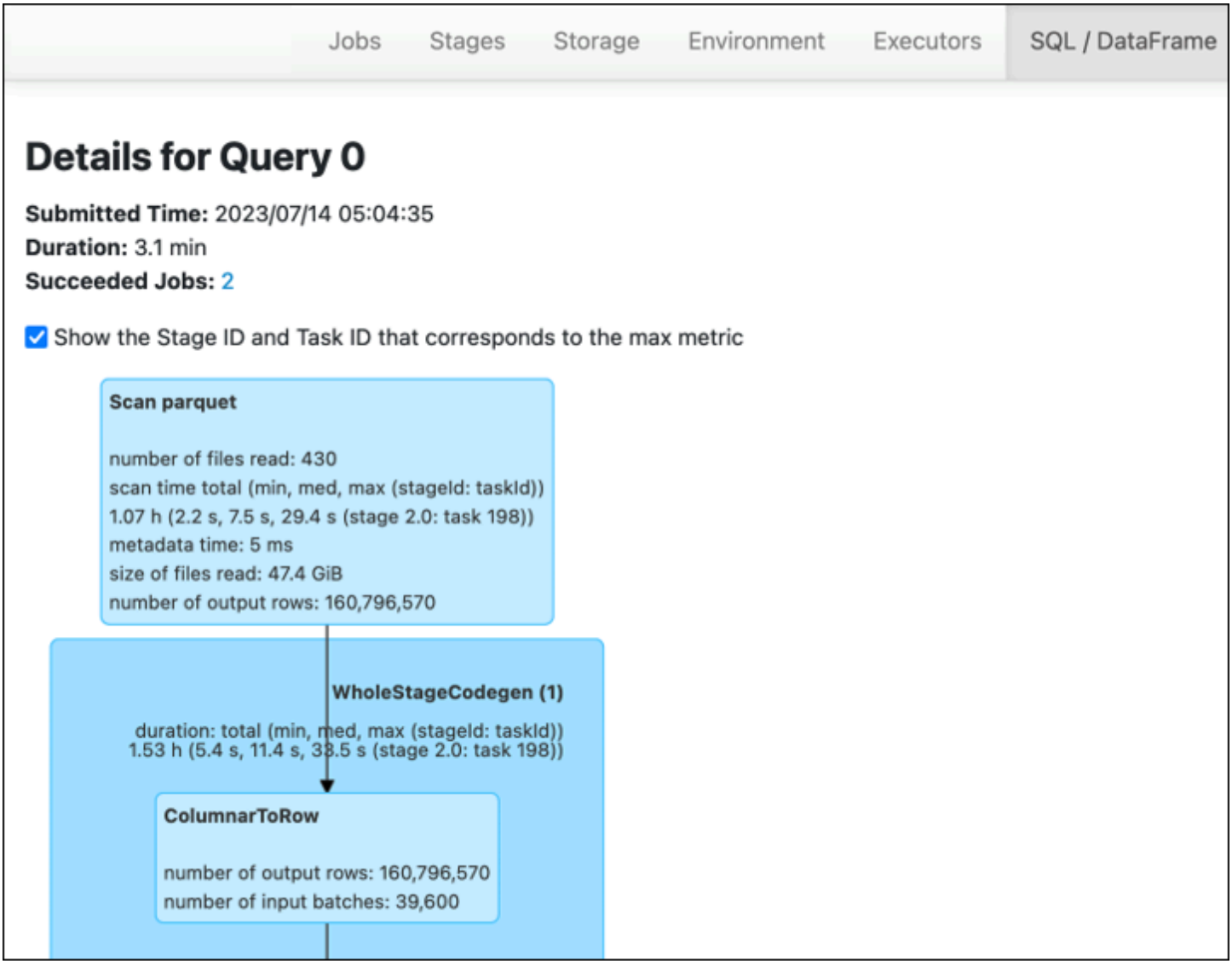
Completed Stages: 6

Completed Stages (6)

Page: 1 — 1 Pages. Jump to 1 . Sho

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
5	parquet at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:09:49	15 s	414/414	61.2 MiB	56.6 MiB
4	load at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:09:47	0.6 s	1/1		
3	Listing leaf files and directories for 43 paths: s3://amazon-reviews-pds/parquet/product_category=Apparel, ... load at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:09:46	1 s	43/43		
2	parquet at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:04:36	3.1 min	414/414	47.4 GiB	47.7 GiB
1	load at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:04:31	2 s	1/1		
0	Listing leaf files and directories for 43 paths: s3://amazon-reviews-pds/parquet/product_category=Apparel, ... load at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:04:13	6 s	43/43		

When you use the Spark SQL or DataFrame approaches in your AWS Glue job, the **SQL /DataFrame** tab shows more statistics about these stages. In this case, stage 2 shows **number of files read: 430**, **size of files read: 47.4 GiB**, and **number of output rows: 160,796,570**.



If you observe that there is a substantial difference in size between the data you are reading in and the data you are using, try the following solutions.

Amazon S3

To reduce the amount of data loaded into your job when reading from Amazon S3, consider file size, compression, file format, and file layout (partitions) for your dataset. AWS Glue for Spark jobs are often used for ETL of raw data, but for efficient distributed processing, you need to inspect the features of your data source format.

- **File size** – We recommend keeping the file size of inputs and outputs within a moderate range (for example, 128 MB). Files that are too small and files that are too large can cause issues.

A large number of small files cause following issues:

- Heavy network I/O load on Amazon S3 because of the overhead required to make requests (such as `List`, `Get`, or `Head`) for many objects (compared with a few objects that store the same quantity of data).
- Heavy I/O and processing load on the Spark driver, which will generate many partitions and tasks and lead to excessive parallelism.

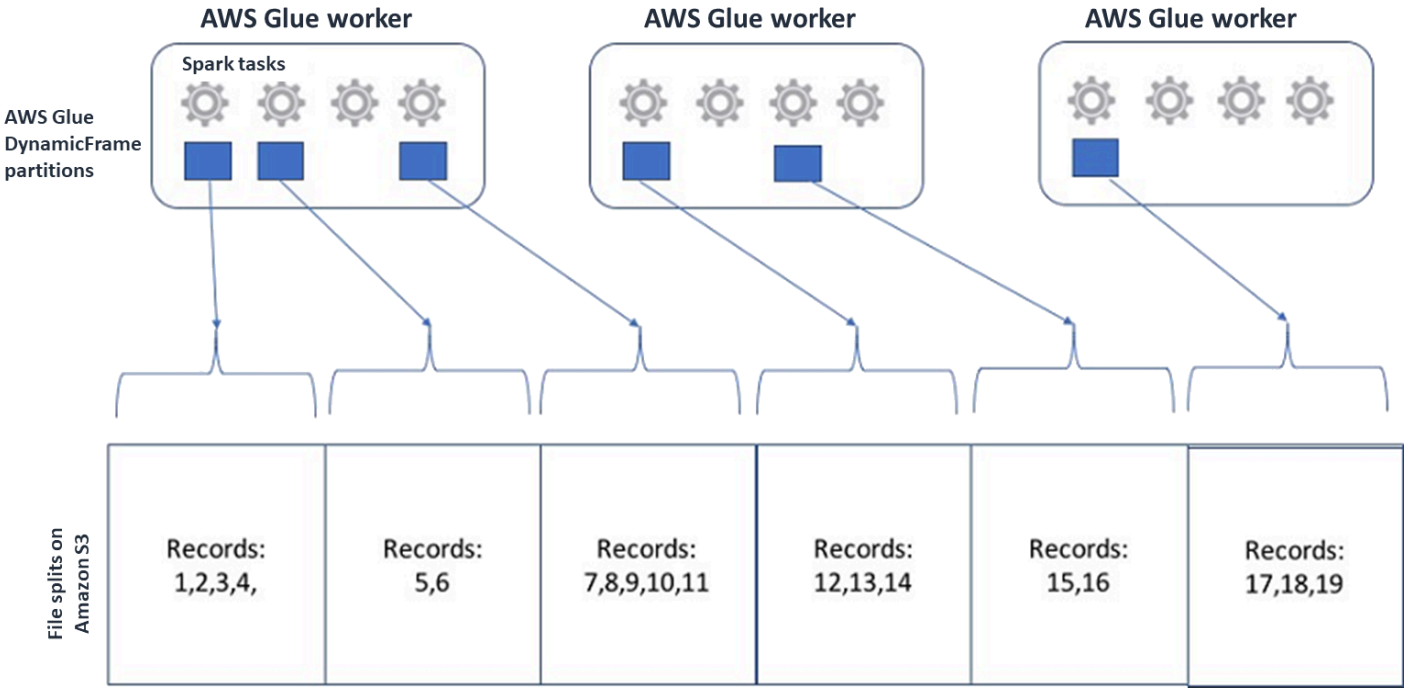
On the other hand, if your file type is not splittable (such as gzip) and the files are too large, the Spark application must wait until a single task has completed reading the entire file.

To reduce excessive parallelism incurred when an Apache Spark task is created for each small file, use file grouping for DynamicFrames. This approach reduces the chances of an OOM exception from the Spark driver. To configure file grouping, set the `groupFiles` and `groupSize` parameters. The following code example uses the AWS Glue DynamicFrame API in an ETL script with these parameters.

```
dyf = glueContext.create_dynamic_frame_from_options("s3",
    {'paths': ["s3://input-s3-path/"],
    'recurse':True,
    'groupFiles': 'inPartition',
    'groupSize': '1048576'},
    format="json")
```

- **Compression** – If your S3 objects are in the hundreds of megabytes, consider compressing them. There are various compression formats, which can be broadly classified into two types:
  - *Unsplittable* compression formats such as gzip require the entire file to be decompressed by one worker.
  - *Splittable* compression formats, such as bzip2 or LZO (indexed), allow partial decompression of a file, which can be parallelized.

  For Spark (and other common distributed-processing engines), you will split up your source data file into chunks your engine can process in parallel. These units are often referred to as *splits*. After your data is in a splittable format, the optimized AWS Glue readers can retrieve splits from an S3 object by providing the `Range` option to the `GetObject` API to retrieve only specific blocks. Consider the following diagram to see how this would work in practice.



Compressed data can speed up your application significantly, as long as the files are either of an optimal size or the files are splittable. The smaller data sizes reduce the data scanned from Amazon S3 and the network traffic from Amazon S3 to your Spark cluster. On the other hand, more CPU is required to compress and decompress data. The amount of compute required scales with the compression ratio of your compression algorithm. Consider this trade-off when choosing your splittable compression format.

> **Note**
>
> While gzip files are not generally splittable, you can compress individual parquet blocks with gzip, and those blocks can be parallelized.

- **File format** – Use a columnar format. Apache Parquet and Apache ORC are popular columnar data formats. Parquet and ORC store data efficiently by employing column-based compression, encoding and compressing each column based on its data type. For more information about Parquet encodings, see Parquet encoding definitions. Parquet files are also splittable.

  Columnar formats group values by column and store them together in blocks. When using columnar formats, you can skip blocks of data that correspond to columns you don't plan to use. Spark applications can retrieve only the columns you need. Generally, better compression

ratios or skipping blocks of data means reading fewer bytes from Amazon S3, leading to better performance. Both formats also support the following pushdown approaches to reduce I/O:

- **Projection pushdown** – Projection pushdown is a technique for retrieving only the columns specified in your application. You specify columns in your Spark application, as shown in the following examples:
  - DataFrame example: `df.select("star_rating")`
  - Spark SQL example: `spark.sql("select start_rating from <table>")`
- **Predicate pushdown** – Predicate pushdown is a technique for efficiently processing `WHERE` and `GROUP BY` clauses. Both formats have blocks of data that represent column values. Each block holds statistics for the block, such as maximum and minimum values. Spark can use these statistics to determine whether the block should be read or skipped depending on the filter value used in the application. To use this feature, add more filters in the conditions, as shown in the following examples as follows:
  - DataFrame example: `df.select("star_rating").filter("star_rating < 2")`
  - Spark SQL example: `spark.sql("select * from <table> where star_rating < 2")`
- **File layout** – By storing your S3 data to objects in different paths based on how the data will be used, you can efficiently retrieve relevant data. For more information, see Organizing objects using prefixes in the Amazon S3 documentation. AWS Glue supports storing keys and values to Amazon S3 prefixes in the format `key=value`, partitioning your data by the Amazon S3 path. By partitioning your data, you can restrict the amount of data scanned by each downstream analytics application, improving performance and reducing cost. For more information, see Managing partitions for ETL output in AWS Glue.

  Partitioning divides your table into different parts and it keeps the related data in grouped files based on column values such as *year*, *month*, and *day*, as shown in the following example.

```
# Partitioning by /YYYY/MM/DD
s3://<YourBucket>/year=2023/month=03/day=31/0000.gz
s3://<YourBucket>/year=2023/month=03/day=01/0000.gz
s3://<YourBucket>/year=2023/month=03/day=02/0000.gz
s3://<YourBucket>/year=2023/month=03/day=03/0000.gz
...
```

  You can define partitions for your dataset by modeling it with a table in the AWS Glue Data Catalog. You can then restrict the amount of data scan by using *partition pruning* as follows:

- For AWS Glue DynamicFrame, set `push_down_predicate` (or `catalogPartitionPredicate`).

```
dyf = Glue_context.create_dynamic_frame.from_catalog(
    database=src_database_name,
    table_name=src_table_name,
    push_down_predicate = "year='2023' and month ='03'",
)
```

- For Spark DataFrame, set a fixed path to prune partitions.

```
df = spark.read.format("json").load("s3://<YourBucket>/year=2023/month=03/*/*.gz")
```

- For Spark SQL, you can set the where clause to prune partitions from the Data Catalog.

```
df = spark.sql("SELECT * FROM <Table> WHERE year= '2023' and month = '03'")
```

- To partition by date when writing your data with AWS Glue, you set partitionKeys in DynamicFrame or partitionBy() in DataFrame with the date information in your columns as follows.
  - DynamicFrame

```
glue_context.write_dynamic_frame_from_options(
    frame= dyf, connection_type='s3',format='parquet'
    connection_options= {
        'partitionKeys': ["year", "month", "day"],
        'path': 's3://<YourBucket>/<Prefix>/'
    }
)
```

  - DataFrame

```
df.write.mode('append')\
    .partitionBy('year','month','day')\
    .parquet('s3://<YourBucket>/<Prefix>/')
```

  This can improve the performance of the consumers of your output data.

  If you don't have access to alter the pipeline that creates your input dataset, partitioning is not an option. Instead, you can exclude unneeded S3 paths by using glob patterns. Set exclusions

when reading in DynamicFrame. For example, the following code excludes days in months 01 to 09, in year 2023.

```
dyf = glueContext.create_dynamic_frame.from_catalog(
    database=db,
    table_name=table,
    additional_options = { "exclusions":"[\"**year=2023/month=0[1-9]/**\"]" },
    transformation_ctx='dyf'
)
```

You can also set exclusions in the table properties in the Data Catalog:

- Key: `exclusions`
- Value: `["**year=2023/month=0[1-9]/**"]`

- **Too many Amazon S3 partitions** – Avoid partitioning your Amazon S3 data on columns that contain a wide range of values, such as an ID column with thousands of values. This can substantially increase the number of partitions in your bucket, because the number of possible partitions is the product of all of the fields you have partitioned by. Too many partitions might cause the following:
  - Increased latency for retrieving partition metadata from the Data Catalog
  - Increased number of small files, which requires more Amazon S3 API requests (`List`, `Get`, and `Head`)

  For example, when you set a date type in `partitionBy` or `partitionKeys`, date-level partitioning such as `yyyy/mm/dd` is good for many use cases. However, `yyyy/mm/dd/<ID>` might generate so many partitions that it would negatively impact performance as a whole.

  On the other hand, some use cases, such as real-time processing applications, require many partitions such as `yyyy/mm/dd/hh`. If your use case requires substantial partitions, consider using AWS Glue partition indexes to reduce latency for retrieving partition metadata from the Data Catalog.

Databases and JDBC

To reduce data scan when retrieving information from a database, you can specify a `where` predicate (or clause) in a SQL query. Databases that do not provide a SQL interface will provide their own mechanism for querying or filtering.

When using Java Database Connectivity (JDBC) connections, provide a select query with the `where` clause for the following parameters:

- For DynamicFrame, use the sampleQuery option. When using `create_dynamic_frame.from_catalog`, configure the `additional_options` argument as follows.

```
query = "SELECT * FROM <TableName> where id = 'XX' AND"
datasource0 = glueContext.create_dynamic_frame.from_catalog(
  database = db,
  table_name = table,
  additional_options={
      "sampleQuery": query,
      "hashexpression": key,
      "hashpartitions": 10,
      "enablePartitioningForSampleQuery": True
    },
  transformation_ctx = "datasource0"
)
```

  When using `create_dynamic_frame.from_options`, configure the `connection_options` argument as follows.

```
query = "SELECT * FROM <TableName> where id = 'XX' AND"
datasource0 = glueContext.create_dynamic_frame.from_options(
    connection_type = connection,
    connection_options={
        "url": url,
        "user": user,
        "password": password,
        "dbtable": table,
        "sampleQuery": query,
        "hashexpression": key,
        "hashpartitions": 10,
        "enablePartitioningForSampleQuery": True
    }
)
```

- For DataFrame, use the query option.

```
query = "SELECT * FROM <TableName> where id = 'XX'"
```

```
jdbcDF = spark.read \
.format('jdbc') \
.option('url', url) \
.option('user', user) \
.option('password', pwd) \
.option('query', query) \
.load()
```

- For Amazon Redshift, use AWS Glue 4.0 or later to take advantage of pushdown support in the Amazon Redshift Spark connector.

```
dyf = glueContext.create_dynamic_frame.from_catalog(
    database = "redshift-dc-database-name",
    table_name = "redshift-table-name",
    redshift_tmp_dir = args["temp-s3-dir"],
    additional_options = {"aws_iam_role": "arn:aws:iam::role-account-id:role/rs-role-name"}
)
```

- For other databases, consult the documentation for that database.

AWS Glue options

- To avoid a full scan for all continuous job runs, and process only data that wasn't present during the last job run, enable job bookmarks.
- To limit the quantity of input data to be processed, enable bounded execution with job bookmarks. This helps to reduce the amount of scanned data for each job run.

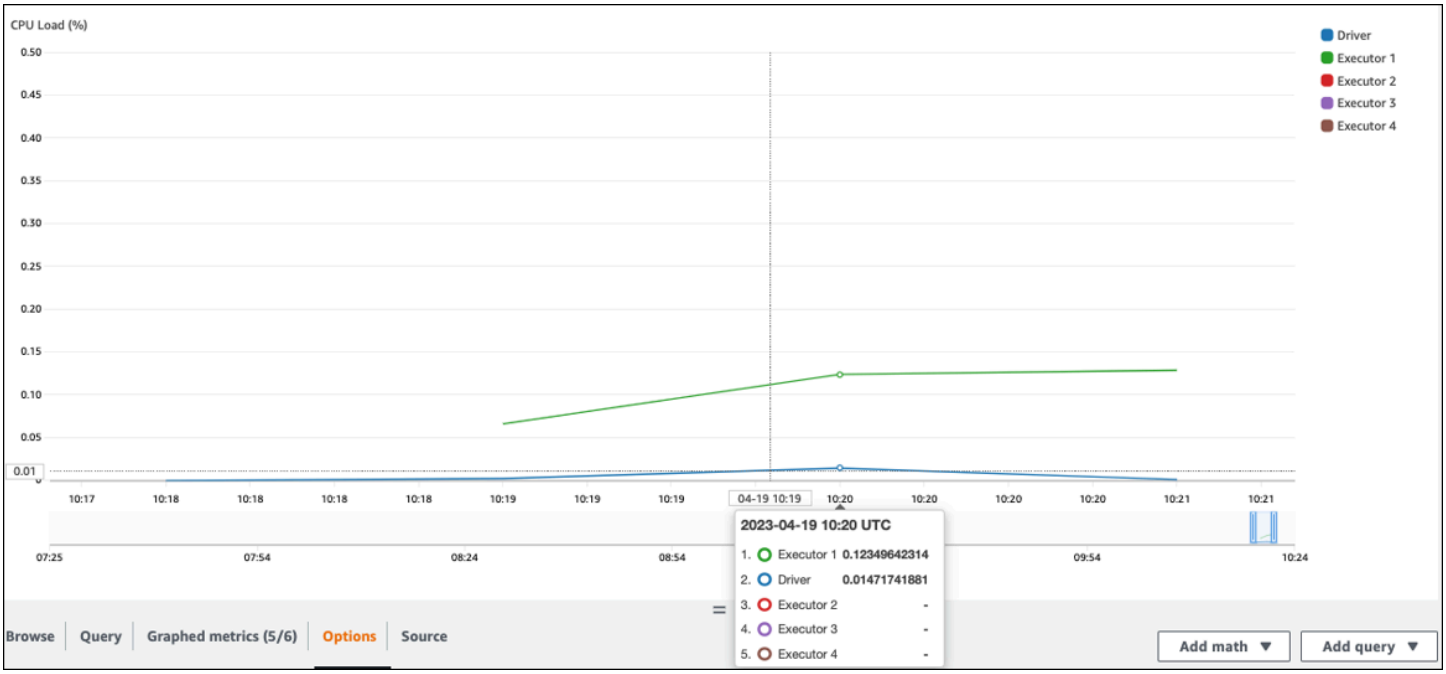
Parallelize tasks

To optimize performance, it's important to parallelize tasks for data loads and transformations. As we discussed in Key topics in Apache Spark, the number of resilient distributed dataset (RDD) partitions is important, because it determines the degree of parallelism. Each task that Spark creates corresponds to an RDD partition on a 1:1 basis. To achieve the best performance, you need to understand how the number of RDD partitions is determined and how that number is optimized.

If you do not have enough parallelism, the following symptoms will be recorded in CloudWatch metrics and the Spark UI.

CloudWatch metrics

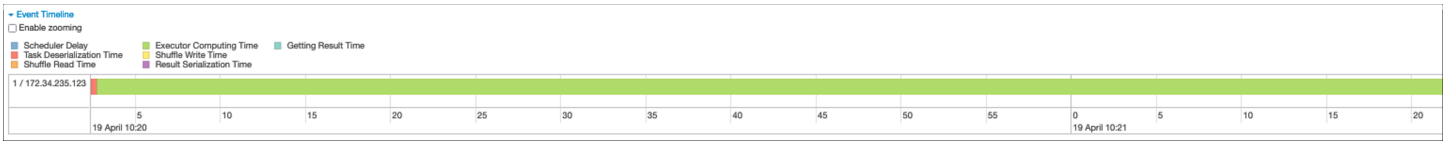
Check the **CPU Load** and **Memory Utilization**. If some executors are not processing during a phase of your job, it's appropriate to improve parallelism. In this case, during the visualized timeframe, **Executor 1** was performing a task, but the remaining executors (2, 3, and 4) were not. You can infer that those executors were not assigned tasks by the Spark driver.



Spark UI

On the **Stage** tab in the Spark UI, you can see the *number of tasks* in a stage. In this case, Spark has performed only one task.

Additionally, the event timeline shows **Executor 1** processing one task. This means that the work in this stage was performed entirely on one executor, while the others were idle.



If you observe these symptoms, try the following solutions for each data source.

Parallelize data load from Amazon S3

To parallelize data loads from Amazon S3, first check the default number of partitions. You can then manually determine a target number of partitions, but be sure to avoid having too many partitions.

Determine the default number of partitions

For Amazon S3, the initial number of Spark RDD partitions (each of which corresponds to a Spark task) is determined by features of your Amazon S3 dataset (for example, format, compression, and size). When you create an AWS Glue DynamicFrame or a Spark DataFrame from CSV objects stored in Amazon S3, the initial number of RDD partitions (`NumPartitions`) can be approximately calculated as follows:

- Object size <= 64 MB: `NumPartitions = Number of Objects`
- Object size > 64 MB: `NumPartitions = Total Object Size / 64 MB`
- Unsplittable (gzip): `NumPartitions = Number of Objects`

As discussed in the Reduce the amount of data scan section, Spark divides large S3 objects into splits that can be processed in parallel. When the object is larger than the split size, Spark splits the object and creates an RDD partition (and task) for each split. Spark's split size is based on your data format and runtime environment, but this is a reasonable starting approximation. Some objects are compressed using unsplittable compression formats such as gzip, so Spark cannot split them.

The `NumPartitions` value might vary depending on your data format, compression, AWS Glue version, number of AWS Glue workers, and Spark configuration.

For example, when you load a single 10 GB `csv.gz` object using a Spark DataFrame, the Spark driver will create only one RDD Partition (`NumPartitions=1`) because gzip is unsplittable. This results in a heavy load on one particular Spark executor and no tasks are assigned to the remaining executors, as described in following figure.

Check the actual number of tasks (`NumPartitions`) for the stage on the Spark Web UI **Stage** tab, or run `df.rdd.getNumPartitions()` in your code to check the parallelism.

When encountering a 10 GB gzip file, examine whether the system generating that file can generate it in a splittable format. If this isn't an option, you might need to scale cluster capacity to process the file. To run transforms efficiently on the data that you loaded, you will need to rebalance your RDD across the workers in your cluster by using repartition.

Manually determine a target number of partitions

Depending on the properties of your data and Spark's implementation of certain functionalities, you might end up with a low `NumPartitions` value even though the underlying work can still be parallelized. If `NumPartitions` is too small, run `df.repartition(N)` to increase the number of partitions so that the processing can be distributed across multiple Spark executors.

In this case, running `df.repartition(100)` will increase `NumPartitions` from 1 to 100, creating 100 partitions of your data, each with a task that can be assigned to the other executors.

The operation `repartition(N)` divides the entire data equally (10 GB / 100 partitions = 100 MB/partition), avoiding data skew to certain partitions.

> **Note**
>
> When a shuffle operation such as `join` is run, the number of partitions is dynamically increased or decreased depending on the value of `spark.sql.shuffle.partitions` or `spark.default.parallelism`. This facilitates a more efficient exchange of data between Spark executors. For more information, see the Spark documentation.

Your goal when determining the target number of partitions is to maximize the use of the provisioned AWS Glue workers. The number of AWS Glue workers and the number of Spark tasks are related through the number of vCPUs. Spark supports one task for each vCPU core. In AWS Glue version 3.0 or later, you can calculate a target number of partitions by using the following formula.

```
# Calculate NumPartitions by WorkerType
numExecutors = (NumberOfWorkers - 1)
numSlotsPerExecutor =
  4 if WorkerType is G.1X
  8 if WorkerType is G.2X
  16 if WorkerType is G.4X
  32 if WorkerType is G.8X
NumPartitions = numSlotsPerExecutor * numExecutors

# Example: Glue 4.0 / G.1X / 10 Workers
numExecutors = ( 10 - 1 ) = 9  # 1 Worker reserved on Spark Driver
numSlotsPerExecutor       = 4  # G.1X has 4 vCpu core ( Glue 3.0 or later )
NumPartitions = 9  * 4    = 36
```

In this example, each G.1X worker provides four vCPU cores to a Spark executor (`spark.executor.cores = 4`). Spark supports one task for each vCPU Core, so G.1X Spark executors can run four tasks simultaneously (`numSlotPerExecutor`). This number of partitions makes full use of the cluster if tasks take an equal amount of time. However, some tasks will take longer than others, creating idle cores. If this happens, consider multiplying `numPartitions` by 2 or 3 to break up and efficiently schedule the bottleneck tasks.

Too many partitions

An excessive number of partitions creates an excessive number of tasks. This causes a heavy load on the Spark driver because of overhead related to distributed processing, such as management tasks and data exchange between Spark executors.

If the number of partitions in your job is substantially larger than your target number of partitions, consider reducing the number of partitions. You can reduce partitions by using the following options:

- If your file sizes are very small, use AWS Glue groupFiles. You can reduce the excessive parallelism resulting from the launch of an Apache Spark task to process each file.
- Use `coalesce(N)` to merge partitions together. This is a low-cost process. When reducing the number of partitions, `coalesce(N)` is preferred over `repartition(N)`, because `repartition(N)` performs shuffle to distribute the amount of records in each partition equally. That increases costs and management overhead.
- Use Spark 3.x Adaptive Query Execution. As discussed in the Key topics in Apache Spark section, Adaptive Query Execution provides a function to automatically coalesce the number of partitions. You can use this approach when you can't know the number of partitions until you perform the execution.

Parallelize data load from JDBC

The number of Spark RDD partitions is determined by configuration. Note that by default only a single task is run to scan an entire source dataset through a `SELECT` query.

Both AWS Glue DynamicFrames and Spark DataFrames support parallelized JDBC data load across multiple tasks. This is done by using `where` predicates to split one `SELECT` query into multiple queries. To parallelize reads from JDBC, configure the following options:

- For AWS Glue DynamicFrame, set `hashfield` (or `hashexpression`) and `hashpartition`. To learn more, see Reading from JDBC tables in parallel.

```
connection_mysql8_options = {
    "url": "jdbc:mysql://XXXXXXXXXX.XXXXXXX.us-east-1.rds.amazonaws.com:3306/test",
    "dbtable": "medicare_tb",
    "user": "test",
    "password": "XXXXXXXXX",
    "hashexpression":"id",
    "hashpartitions":"10"
}
datasource0 = glueContext.create_dynamic_frame.from_options(
    'mysql',
    connection_options=connection_mysql8_options,
    transformation_ctx= "datasource0"
)
```

- For Spark DataFrame, set `numPartitions`, `partitionColumn`, `lowerBound`, and `upperBound`. To learn more, see JDBC To Other Databases.

```
df = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:mysql://XXXXXXXXXX.XXXXXXX.us-east-1.rds.amazonaws.com:3306/
test") \
    .option("dbtable", "medicare_tb") \
    .option("user", "test") \
    .option("password", "XXXXXXXXXX") \
    .option("partitionColumn", "id") \
    .option("numPartitions", "10") \
    .option("lowerBound", "0") \
    .option("upperBound", "1141455") \
    .load()

df.write.format("json").save("s3://bucket_name/Tests/sparkjdbc/with_parallel/")
```

Parallelize data load from DynamoDB when using the ETL connector

The number of Spark RDD partitions is determined by the `dynamodb.splits` parameter. To parallelize reads from Amazon DynamoDB, configure the following options:

- Increase the value of `dynamodb.splits`.

- Optimize the parameter by following the formula explained in Connection types and options for ETL in AWS Glue for Spark.

Parallelize data load from Kinesis Data Streams

The number of Spark RDD partitions is determined by the number of shards in the source Amazon Kinesis Data Streams data stream. If you have only a few shards in your data stream, there will be only a few Spark tasks. This can result in low parallelism in downstream processes. To parallelize reads from Kinesis Data Streams, configure the following options:

- Increase the number of shards to obtain more parallelism when loading data from Kinesis Data Streams.
- If your logic in the micro-batch is complex enough, consider repartitioning the data at the beginning of the batch, after dropping unneeded columns.

For more information, see Best practices to optimize cost and performance for AWS Glue streaming ETL jobs.

Parallelize tasks after data load

To parallelize tasks after data load, increase the number of RDD partitions by using the following options:

- Repartition data to generate a greater number of partitions, especially right after initial load if the load itself could not be parallelized.

  Call `repartition()` either on DynamicFrame or DataFrame, specifying the number of partitions. A good rule of thumb is two or three times the number of cores available.

  However, when writing a partitioned table, this can lead to an explosion of files (each partition can potentially generate a file into each table partition). To avoid this, you can repartition your DataFrame by column. This uses the table partition columns so the data is organized before writing. You can specify a higher number of partitions without getting small files on the table partitions. However, be careful to avoid data skew, in which some partition values end up with most of the data and delay the completion of the task.
- When there are shuffles, increase the `spark.sql.shuffle.partitions` value. This also can help with any memory issues when shuffling.

When you have more than 2,001 shuffle partitions, Spark uses a compressed memory format. If you have a number close to that, you might want to set the `spark.sql.shuffle.paritions` value over that limit to get the more efficient representation.

Optimize shuffles

Certain operations, such as `join()` and `groupByKey()`, require Spark to perform a shuffle. The shuffle is Spark's mechanism for redistributing data so that it's grouped differently across RDD partitions. Shuffling can help remediate performance bottlenecks. However, because shuffling typically involves copying data between Spark executors, the shuffle is a complex and costly operation. For example, shuffles generate the following costs:

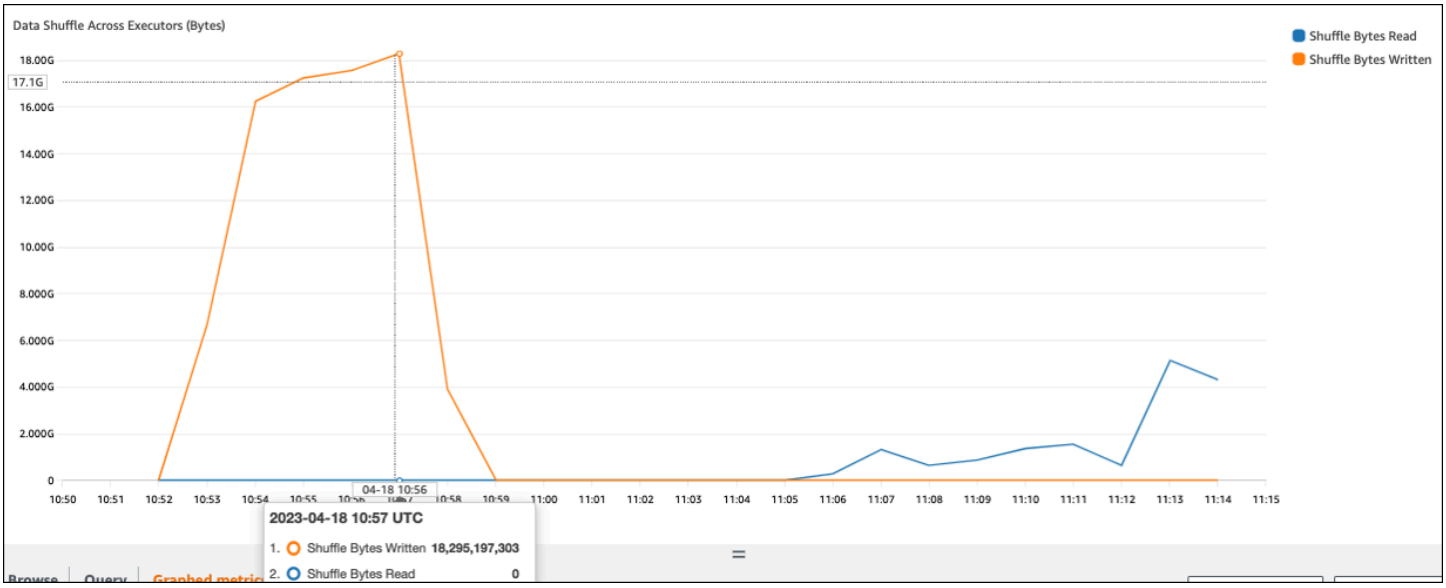
- Disk I/O:
  - Generates a large number of intermediate files on disk.
- Network I/O:
  - Needs many network connections (Number of connections = `Mapper × Reducer`).
  - Because records are aggregated to new RDD partitions that might be hosted on a different Spark executor, a substantial fraction of your dataset might move between Spark executors over the network.
- CPU and memory load:
  - Sorts values and merges sets of data. These operations are planned on the executor, placing a heavy load on the executor.

Shuffle is one of the most substantial factors in degraded performance of your Spark application. While storing the intermediate data, it can exhaust space on the executor's local disk, which causes the Spark job to fail.

You can assess your shuffle performance in CloudWatch metrics and in the Spark UI.

CloudWatch metrics

If the **Shuffle Bytes Written** value is high compared with **Shuffle Bytes Read**, your Spark job might use shuffle operations such as `join()` or `groupByKey()`.



Spark UI

On the **Stage** tab of the Spark UI, you can check the **Shuffle Read Size / Records** values. You can also see it on the **Executors** tab.

In the following screenshot, each executor exchanges approximately 18.6GB/4020000 records with the shuffle process, for a total shuffle read size of about 75 GB).

The **Shuffle Spill (Disk)** column shows a large amount of data spill memory to disk, which might cause a full disk or a performance issue.

Aggregated Metrics by Executor

Executor ID ▲	Address	Shuffle Read Size / Records	Shuffle Spill (Memory)	Shuffle Spill (Disk)
1	172.35.205.23:46731	18.6 GB / 40210300	98.1 GB	16.8 GB
2	172.35.195.173:46185	18.7 GB / 40246767	117.2 GB	17.3 GB
3	172.36.135.106:35913	18.6 GB / 40253921	101.6 GB	16.6 GB
4	172.34.131.223:46879	18.6 GB / 40190741	99.5 GB	16.4 GB

If you observe these symptoms and the stage takes too long when compared to your performance goals, or it fails with `Out Of Memory` or `No space left on device` errors, consider the following solutions.

Optimize the join

The `join()` operation, which joins tables, is the most commonly used shuffle operation, but it's often a performance bottleneck. Because join is a costly operation, we recommend not using it

unless it's essential to your business requirements. Double-check that you are making efficient use of your data pipeline by asking the following questions:

- Are you recomputing a join that is also performed in other jobs you can reuse?
- Are you joining to resolve foreign keys to values that aren't used by the consumers of your output?

After you confirm that your join operations are essential to your business requirements, see the following options for optimizing your join in a way that meets your requirements.

Use pushdown before join

Filter out unnecessary rows and columns in the DataFrame before performing a join. This has the following advantages:

- Reduces the amount of data transfer during shuffle
- Reduces the amount of processing in the Spark executor
- Reduces the amount of data scan

```
# Default
df_joined  = df1.join(df2, ["product_id"])

# Use Pushdown
df1_select =
 df1.select("product_id","product_title","star_rating").filter(col("star_rating")>=4.0)
df2_select = df2.select("product_id","category_id")
df_joined  = df1_select.join(df2_select, ["product_id"])
```

Use DataFrame Join

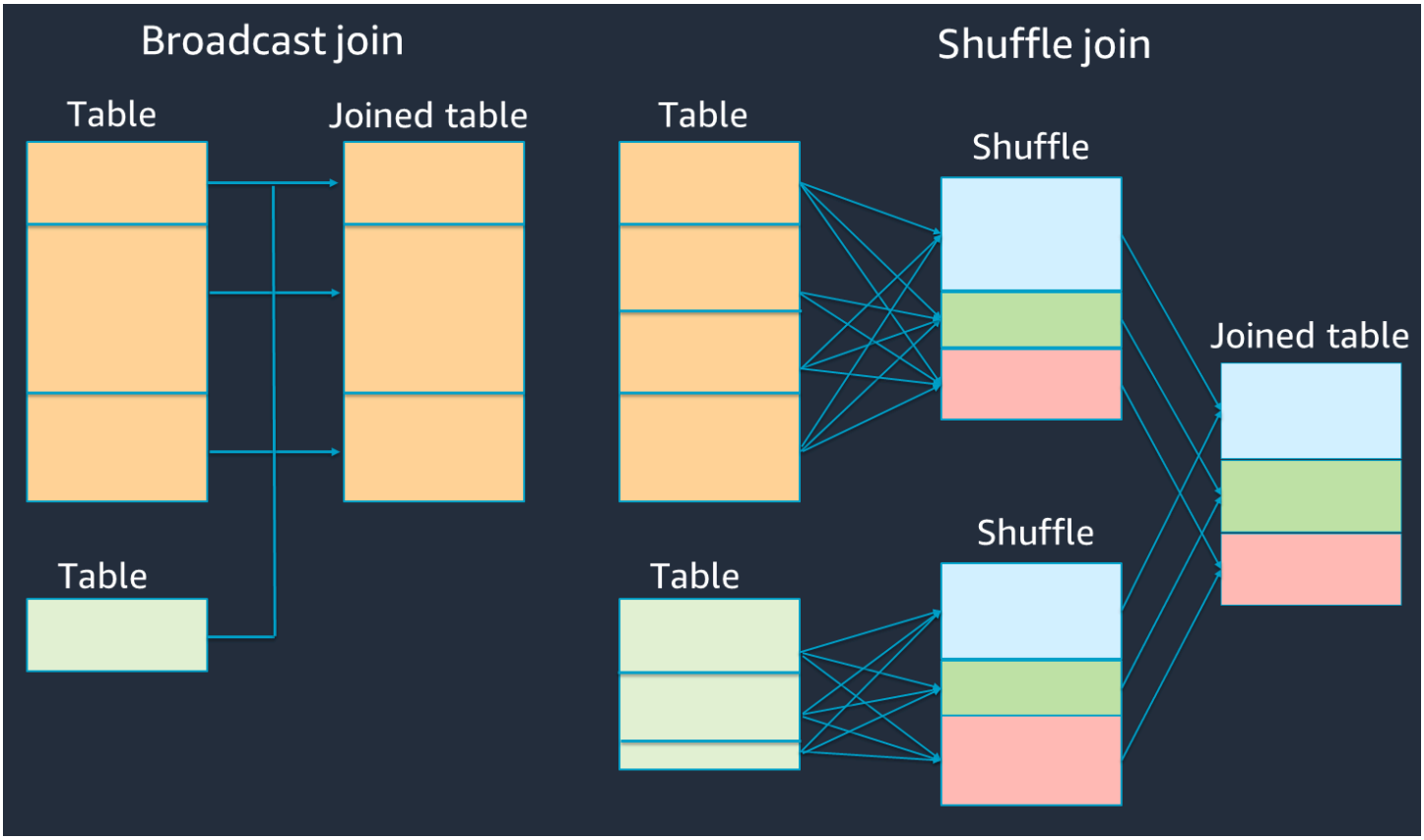
Try using a Spark high-level API such as SparkSQL, DataFrame, and Datasets instead of the RDD API or DynamicFrame join. You can convert DynamicFrame to DataFrame with a method call such as `dyf.toDF()`. As discussed in the Key topics in Apache Spark section, these join operations internally take advantage of query optimization by the Catalyst optimizer.

Shuffle and broadcast hash joins and hints

Spark supports two types of join: shuffle join and broadcast hash join. A broadcast hash join doesn't require shuffling, and it can require less processing than a shuffle join. However, it's

applicable only when joining a small table to a large one. When joining a table that can fit in the memory of a single Spark executor, consider using a broadcast hash join.

The following diagram shows the high-level structure and steps of a broadcast hash join and a shuffle join.



The details of each join are as follows:

- Shuffle join:
  - The shuffle hash join joins two tables without sorting and distributes the join between the two tables. It's suitable for joins of small tables that can be stored in the Spark executor's memory.
  - The sort-merge join distributes the two tables to be joined by key and sorts them before joining. It's suitable for joins of large tables.
- Broadcast hash join:
  - A broadcast hash join pushes the smaller RDD or table to each of the worker nodes. Then it does a map-side combine with each partition of the larger RDD or table.

It's suitable for joins when one of your RDDs or tables can fit in memory or can be made to fit in memory. It's beneficial to do a broadcast hash join when possible, because it doesn't require a shuffle. You can use a join hint to request a broadcast join from Spark as follows.

```
# DataFrame
from pySpark.sql.functions import broadcast
df_joined= df_big.join(broadcast(df_small), right_df[key] == left_df[key],
 how='inner')

-- SparkSQL
SELECT /*+ BROADCAST(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```

For more information about join hints, see Join hints.

In AWS Glue 3.0 and later, you can take advantage of broadcast hash joins automatically by enabling Adaptive Query Execution and additional parameters. Adaptive Query Execution converts a sort-merge join to a broadcast hash join when the runtime statistics of either join side is smaller than the adaptive broadcast hash join threshold.

In AWS Glue 3.0, you can enable Adaptive Query Execution by setting `spark.sql.adaptive.enabled=true`. Adaptive Query Execution is enabled by default in AWS Glue 4.0.

You can set additional parameters related to shuffles and broadcast hash joins:

- `spark.sql.adaptive.localShuffleReader.enabled`
- `spark.sql.adaptive.autoBroadcastJoinThreshold`

For more information about related parameters, see Converting sort-merge join to broadcast join.

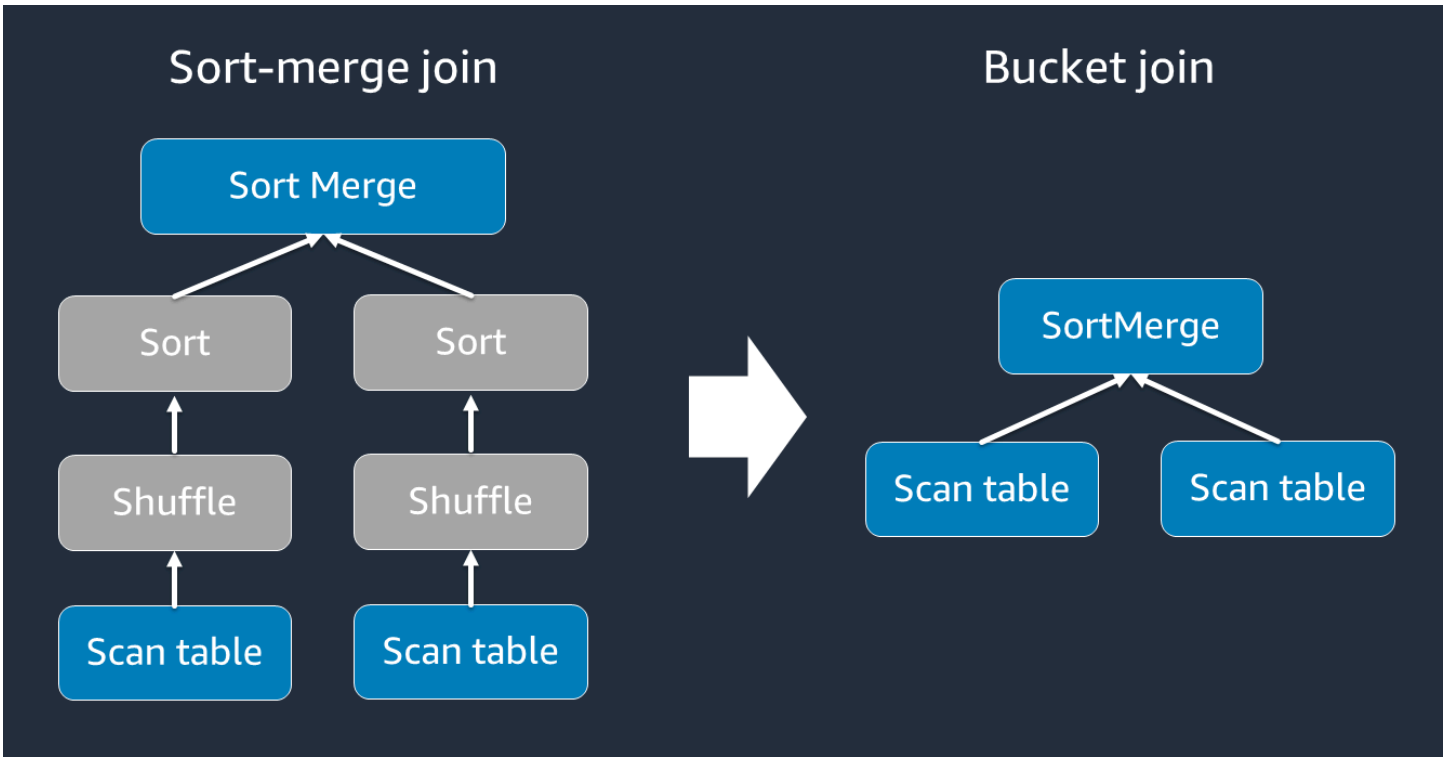
In AWS Glue 3.0 and or later, you can use other join hints for shuffle to tune your behavior.

```
-- Join Hints for shuffle sort merge join
SELECT /*+ SHUFFLE_MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGEJOIN(t2) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle hash join
```

```
SELECT /*+ SHUFFLE_HASH(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle-and-replicate nested loop join
SELECT /*+ SHUFFLE_REPLICATE_NL(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```

Use bucketing

The sort-merge join requires two phases, shuffle and sort, and then merge. These two phases can overload the Spark executor and cause OOM and performance issues when some of the executors are merging and others are sorting simultaneously. In such cases, it might be possible to efficiently join by using bucketing. Bucketing will pre-shuffle and pre-sort your input on join keys, and then write that sorted data to an intermediary table. The cost of the shuffle and sort steps can be reduced when joining large tables by defining the sorted intermediary tables in advance.



Bucketed tables are useful for the following:

- Data joined frequently over the same key, such as `account_id`
- Loading daily cumulative tables, such as base and delta tables that could be bucketed on a common column

You can create a bucketed table by using the following code.

```
df.write.bucketBy(50, "account_id").sortBy("age").saveAsTable("bucketed_table")
```

Repartition DataFrames on join keys before the join

To repartition the two DataFrames on the join keys before the join, use the following statements.

```
df1_repartitioned = df1.repartition(N,"join_key")
df2_repartitioned = df2.repartition(N,"join_key")
df_joined  = df1_repartitioned.join(df2_repartitioned,"product_id")
```

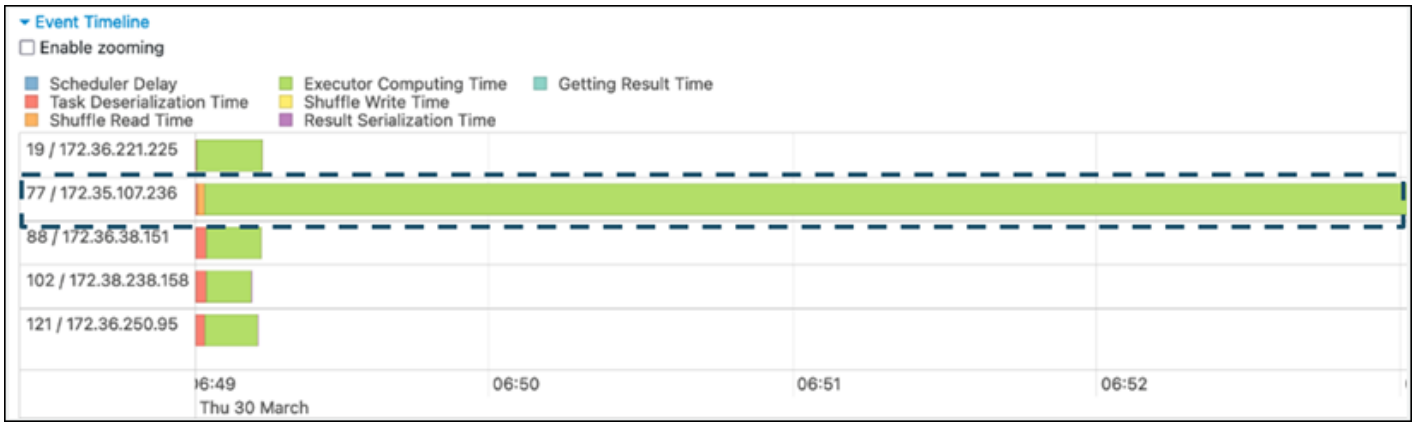
This will partition two (still separate) RDDs on the join key before initiating the join. If the two RDDs are partitioned on the same key with the same partitioning code, RDD records that your plan to join together will have a high likelihood of being co-located on the same worker before shuffling for the join. This might improve performance by reducing network activity and data skew during the join.

Overcome data skew

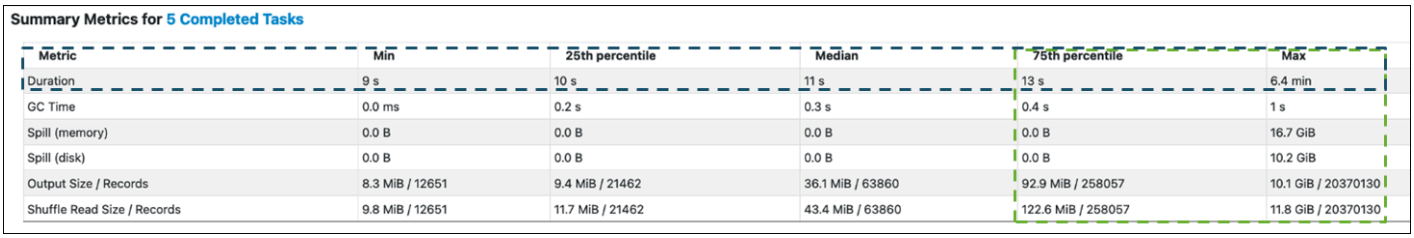
Data skew is one of the most common causes of a bottleneck for Spark jobs. It occurs when data isn't uniformly distributed across RDD partitions. This causes tasks for that partition to take much longer than others, delaying the overall processing time of the application.

To identify data skew, assess the following metrics in the Spark UI:

- On the **Stage** tab in the Spark UI, examine the **Event Timeline** page. You can see an uneven distribution of tasks in the following screenshot. Tasks that are distributed unevenly or are taking too long to run can indicate data skew.



- Another important page is **Summary Metrics**, which shows statistics for Spark tasks. The following screenshot shows metrics with percentiles for **Duration**, **GC Time**, **Spill (memory)**, **Spill (disk)**, and so on.



Summary Metrics for 5 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	9 s	10 s	11 s	13 s	6.4 min
GC Time	0.0 ms	0.2 s	0.3 s	0.4 s	1 s
Spill (memory)	0.0 B	0.0 B	0.0 B	0.0 B	16.7 GiB
Spill (disk)	0.0 B	0.0 B	0.0 B	0.0 B	10.2 GiB
Output Size / Records	8.3 MiB / 12651	9.4 MiB / 21462	36.1 MiB / 63860	92.9 MiB / 258057	10.1 GiB / 20370130
Shuffle Read Size / Records	9.8 MiB / 12651	11.7 MiB / 21462	43.4 MiB / 63860	122.6 MiB / 258057	11.8 GiB / 20370130

  When the tasks are evenly distributed, you will see similar numbers in all the percentiles. When there is data skew, you will see very biased values in each percentile. In the example, task duration is less than 13 seconds in **Min**, **25th percentile**, **Median**, and **75th percentile**. While the **Max** task processed 100 times more data than the **75th percentile**, its duration of 6.4 minutes is about 30 times longer. It means that at least one task (or up to 25 percent of the tasks) took far longer than the rest of the tasks.

If you see data skew, try the following:

- If you use AWS Glue 3.0, enable Adaptive Query Execution by setting `spark.sql.adaptive.enabled=true`. Adaptive Query Execution is enabled by default in AWS Glue 4.0.

  You can also use Adaptive Query Execution for data skew introduced by joins by setting the following related parameters:

  - `spark.sql.adaptive.skewJoin.skewedPartitionFactor`
  - `spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes`
  - `spark.sql.adaptive.advisoryPartitionSizeInBytes=128m (128 mebibytes or larger should be good)`
  - `spark.sql.adaptive.coalescePartitions.enabled=true (when you want to coalesce partitions)`

  For more information, see the Apache Spark documentation.

- Use keys with a large range of values for the join keys. In a shuffle join, partitions are determined for each hash value of a key. If a join key's cardinality is too low, the hash function is more likely

to do a bad job of distributing your data across partitions. Therefore, if your application and business logic support it, consider using a higher cardinality key or a composite key.

```
# Use Single Primary Key
df_joined = df1_select.join(df2_select, ["primary_key"])

# Use Composite Key
df_joined = df1_select.join(df2_select, ["primary_key","secondary_key"])
```

Use cache

When you use repetitive DataFrames, avoid additional shuffle or computation by using `df.cache()` or `df.persist()` to cache the calculation results in each Spark executor's memory and on disk. Spark also supports persisting RDDs on disk or replicating across multiple nodes (storage level).

For example, you can persist the DataFrames by adding `df.persist()`. When the cache is no longer needed, you can use `unpersist` to discard the cached data.

```
df = spark.read.parquet("s3://<Bucket>/parquet/product_category=Books/")
df_high_rate = df.filter(col("star_rating")>=4.0)
df_high_rate.persist()

df_joined1 = df_high_rate.join(<Table1>, ["key"])
df_joined2 = df_high_rate.join(<Table2>, ["key"])
df_joined3 = df_high_rate.join(<Table3>, ["key"])
...
df_high_rate.unpersist()
```

Remove unneeded Spark actions

Avoid running unnecessary actions such as `count`, `show` , or `collect`. As discussed in the Key topics in Apache Spark section, Spark is lazy. Each transformed RDD might be recomputed each time you run an action on it. When you use many Spark actions, multiple source accesses, task calculations, and shuffle runs for each action are being called.

If you don't need `collect()` or other actions in your commercial environment, consider removing them.

> **Note**
>
> Avoid using Spark `collect()` in commercial environments as much as possible. The `collect()` action returns all the results of a calculation in the Spark executor to the Spark driver, which might cause the Spark driver to return an OOM error. To avoid an OOM error, Spark sets `spark.driver.maxResultSize = 1GB` by default, which limits the maximum data size returned to the Spark driver to 1 GB.

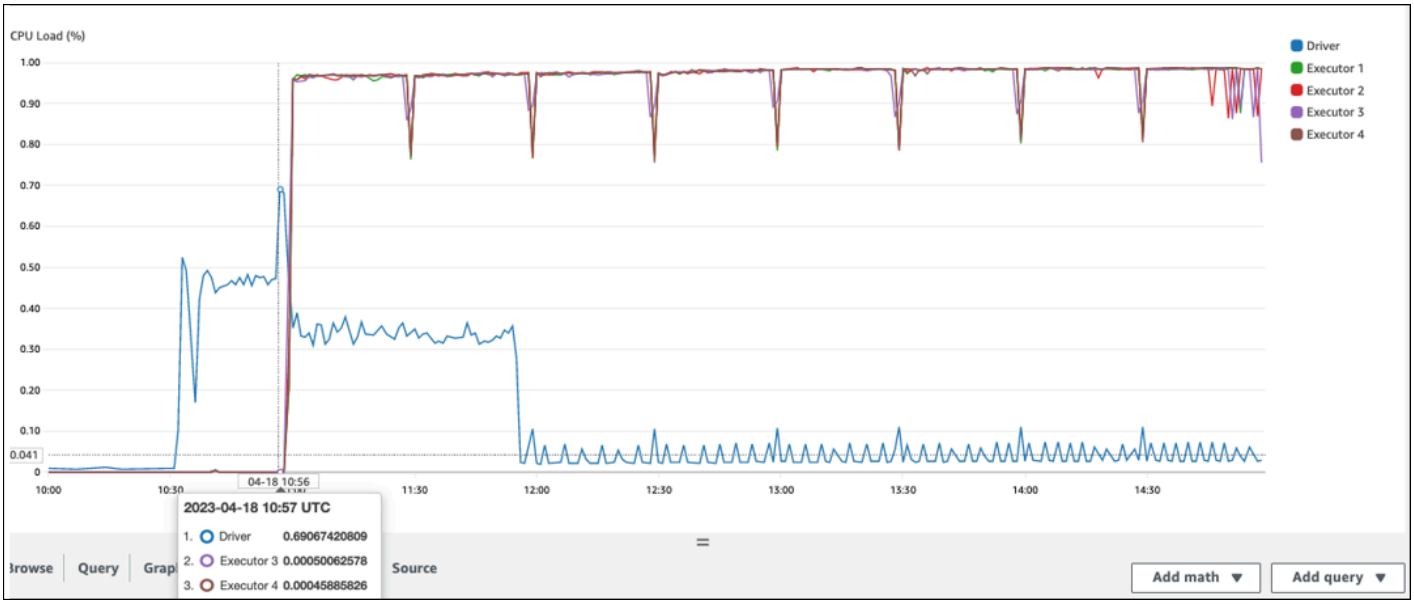
Minimize planning overhead

As discussed Key topics in Apache Spark, the Spark driver generates the execution plan. Based on that plan, tasks are assigned to the Spark executor for distributed processing. However, the Spark driver can become a bottleneck if there is a large number of small files or if the AWS Glue Data Catalog contains a large number of partitions. To identify high planning overhead, assess the following metrics.

CloudWatch metrics

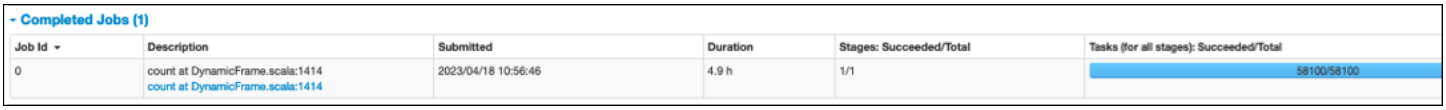
Check **CPU Load** and **Memory Utilization** for the following situations:

- Spark driver **CPU Load** and **Memory Utilization** are recorded as high. Normally, the Spark driver doesn't process your data, so CPU load and memory utilization don't spike. However, if the Amazon S3 data source has too many small files, listing all the S3 objects and managing a large number of tasks might cause resource utilization to be high.
- There is a long gap before processing starts in Spark executor. In the following example screenshot, the Spark executor's CPU Load is too low until 10:57, even though the AWS Glue job started at 10:00. This indicates that the Spark driver might be taking a long time to generate an execution plan. In this example, retrieving the large number of partitions in the Data Catalog and listing the large number of small files in the Spark driver is taking a long time.



Spark UI

On the **Job** tab in the Spark UI, you can see the **Submitted** time. In the following example, the Spark driver started job0 at 10:56:46, even though the AWS Glue job started at 10:00:00.



You can also see the **Tasks (for all stages): Succeeded/Total** time on the **Job** tab. In this case, the number of tasks is recorded as 58100. As explained in the Amazon S3 section of the Parallelize tasks page, the number of tasks approximately corresponds to the number of S3 objects. This means that there are about 58,100 objects in Amazon S3.

For more details about this job and timeline, review the **Stage** tab. If you observe a bottleneck with the Spark driver, consider the following solutions:

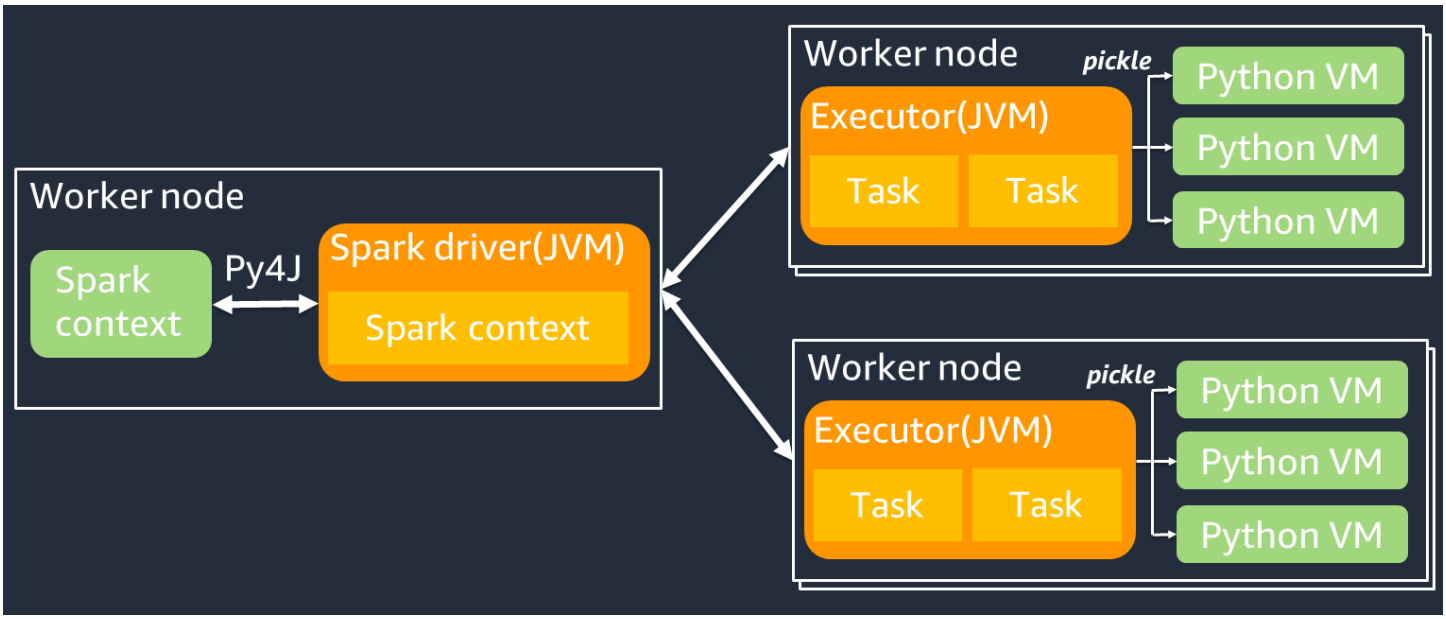
- When Amazon S3 has too many files, consider the guidance on excessive parallelism in the *Too many partitions* section of the Parallelize tasks page.
- When Amazon S3 has too many partitions, consider the guidance on excessive partitioning in the *Too many Amazon S3 partitions* section of the Reduce the amount of data scan page. Enable AWS Glue partition indexes if there are many partitions to reduce latency for retrieving partition metadata from the Data Catalog. For more information, see Improve query performance using AWS Glue partition indexes.

- When JDBC has too many partitions, lower the `hashpartition` value.
- When DynamoDB has too many partitions, lower the `dynamodb.splits` value.
- When streaming jobs have too many partitions, lower the number of shards.

Optimize user-defined functions

User-defined functions (UDFs) and `RDD.map` in PySpark often degrade performance significantly. This is because of the overhead required to accurately represent your Python code in Spark's underlying Scala implementation.

The following diagram shows the architecture of PySpark jobs. When you use PySpark, the Spark driver uses the Py4j library to call Java methods from Python. When calling Spark SQL or DataFrame built-in functions, there is little performance difference between Python and Scala because the functions run on each executor's JVM using an optimized execution plan.



If you use your own Python logic, such as using `map/ mapPartitions/ udf`, the task will run in a Python runtime environment. Managing two environments creates an overhead cost. Additionally, your data in memory must be transformed for use by the JVM runtime environment's built-in functions. *Pickle* is a serialization format used by default for the exchange between the JVM and Python runtimes. However, the cost of this serialization and deserialization cost is very high, so UDFs written in Java or Scala are faster than Python UDFs.

To avoid serialization and deserialization overhead in PySpark, consider the following:

- **Use the built-in Spark SQL functions** – Consider replacing your own UDF or map function with Spark SQL or DataFrame built-in functions. When running Spark SQL or DataFrame built-in functions, there is little performance difference between Python and Scala because the tasks are handled on each executor's JVM .
- **Implement UDFs in Scala or Java** – Consider using a UDF which is written in Java or Scala, because they run on the JVM.
- **Use Apache Arrow-based UDFs for vectorized workloads** – Consider using Arrow-based UDFs. This feature is also known as Vectorized UDF (Pandas UDF). Apache Arrow is a language-agnostic in-memory data format that AWS Glue can use to efficiently transfer data between JVM and Python processes. This is currently most beneficial to Python users that work with Pandas or NumPy data.

  Arrow is a columnar (vectorized) format. Its usage is not automatic and might require some minor changes to configuration or code to take full advantage and ensure compatibility. For more detail and limitations see Apache Arrow in PySpark.

  The following example compares a basic incremental UDF in standard Python, as a Vectorized UDF, and in Spark SQL.

Standard Python UDF

Example time is 3.20 (sec).

Example code

```
# DataSet
df = spark.range(10000000).selectExpr("id AS a","id AS b")

# UDF Example
def plus(a,b):
    return a+b
spark.udf.register("plus",plus)

df.selectExpr("count(plus(a,b))").collect()
```

Execution plan

```
== Physical Plan ==
```

```
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[count(pythonUDF0#124)])
+- Exchange SinglePartition, ENSURE_REQUIREMENTS, [id=#580]
+- HashAggregate(keys=[], functions=[partial_count(pythonUDF0#124)])
+- Project [pythonUDF0#124]
+- BatchEvalPython [plus(a#116L, b#117L)], [pythonUDF0#124]
+- Project [id#114L AS a#116L, id#114L AS b#117L]
+- Range (0, 10000000, step=1, splits=16)
```

Vectorized UDF

Example time is 0.59 (sec).

The Vectorized UDF is 5 times faster than the previous UDF example. Checking `Physical Plan`, you can see `ArrowEvalPython`, which shows this application is vectorized by Apache Arrow. To enable Vectorized UDF, you must specify `spark.sql.execution.arrow.pyspark.enabled = true` in your code.

Example code

```
# Vectorized UDF
from pyspark.sql.types import LongType
from pyspark.sql.functions import count, pandas_udf

# Enable Apache Arrow Support
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")

# DataSet
df = spark.range(10000000).selectExpr("id AS a","id AS b")

# Annotate pandas_udf to use Vectorized UDF
@pandas_udf(LongType())
def pandas_plus(a,b):
    return a+b
spark.udf.register("pandas_plus",pandas_plus)

df.selectExpr("count(pandas_plus(a,b))").collect()
```

Execution plan

```
== Physical Plan ==
```

```
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[count(pythonUDF0#1082L)],
 output=[count(pandas_plus(a, b))#1080L])
+- Exchange SinglePartition, ENSURE_REQUIREMENTS, [id=#5985]
+- HashAggregate(keys=[], functions=[partial_count(pythonUDF0#1082L)],
 output=[count#1084L])
+- Project [pythonUDF0#1082L]
+- ArrowEvalPython [pandas_plus(a#1074L, b#1075L)], [pythonUDF0#1082L], 200
+- Project [id#1072L AS a#1074L, id#1072L AS b#1075L]
+- Range (0, 10000000, step=1, splits=16)
```

Spark SQL

Example time is 0.087 (sec).

Spark SQL is much faster than Vectorized UDF, because the tasks are run on each executor's JVM without a Python runtime . If you can replace your UDF with a built-in function, we recommend doing so.

Example code

```
df.createOrReplaceTempView("test")
spark.sql("select count(a+b) from test").collect()
```

Using pandas for big data

If you are already familiar with pandas and want to use Spark for big data, you can use the pandas API on Spark. AWS Glue 4.0 and later support it. To get started, you can use the official notebook Quickstart: Pandas API on Spark. For more information, see the PySpark documentation.

Resources

- AWS Glue
- Performance Tuning (Spark SQL Guide)
- AWS Glue Optimization Workshop

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	January 2, 2024

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. This migration scenario is specific to VMware Cloud on AWS, which supports virtual machine (VM) compatibility and workload portability between your on-premises environment and AWS. You can use the VMware Cloud Foundation technologies from your on-premises data centers when you migrate your infrastructure to VMware Cloud on AWS. Example: Relocate the hypervisor hosting your Oracle database to VMware Cloud on AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later

time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or AWS CodeCommit. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision

A field of AI used by machines to identify people, places, and things in images with accuracy at or above human levels. Often built with deep learning models, it automates extraction, analysis, classification, and understanding of useful information from a single image or a sequence of images.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in

an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally

designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts

or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies,

and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with :AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML

model to benefit from the data. For example, if you break down the “2021-05-27 00:15:37” date into “2021”, “May”, “Thu”, and “15”, you can help the learning algorithm learn nuanced patterns associated with different data components.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

G

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries. *Detective guardrails* detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently

more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial

cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

MPA

See Migration Portfolio Assessment.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic PUT and DELETE requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

Privacy by Design

An approach in system engineering that takes privacy into account throughout the whole engineering process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

production environment

See environment.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Managing AWS Regions in *AWS General Reference*.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata. The secret value can be binary, a single string, or multiple strings. For more information, see Secret in the Secrets Manager documentation.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid

innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.