aws

Apache Spark 作业性能调整 AWS Glue 的最佳实践

: Apache Spark 作业性能调整 AWS Glue 的最佳实践

Table of Contents

Apache Spark 作业性能调整 AWS Glue 的最佳实践

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

2023 年 12 月（文档历史记录）

AWS Glue 为调整性能提供了不同的选项。本指南定义了调整 Apache Spark AWS Glue 的关键主题。然后，它提供了一个基准策略，供您在调整 Apache Spark 作业时遵循这些 AWS Glue 策略。使用本指南学习如何通过解释中提供的指标来识别性能问题 AWS Glue。然后采用策略来解决这些问题，最大限度地提高性能并最大限度地降低成本。

本指南涵盖以下调整实践：

- 扩展集群容量
- 使用最新 AWS Glue 版本
- 减少数据扫描量
- 并行化任务
- 最大限度减少规划开销
- 优化洗牌
- 优化用户定义的函数

Apache Spark 中的关键话题

本节介绍了 Apache Spark 的基本概念和调整 Apache Spark 性能 AWS Glue 的关键主题。在讨论现实世界的调整策略之前，了解这些概念和主题很重要。

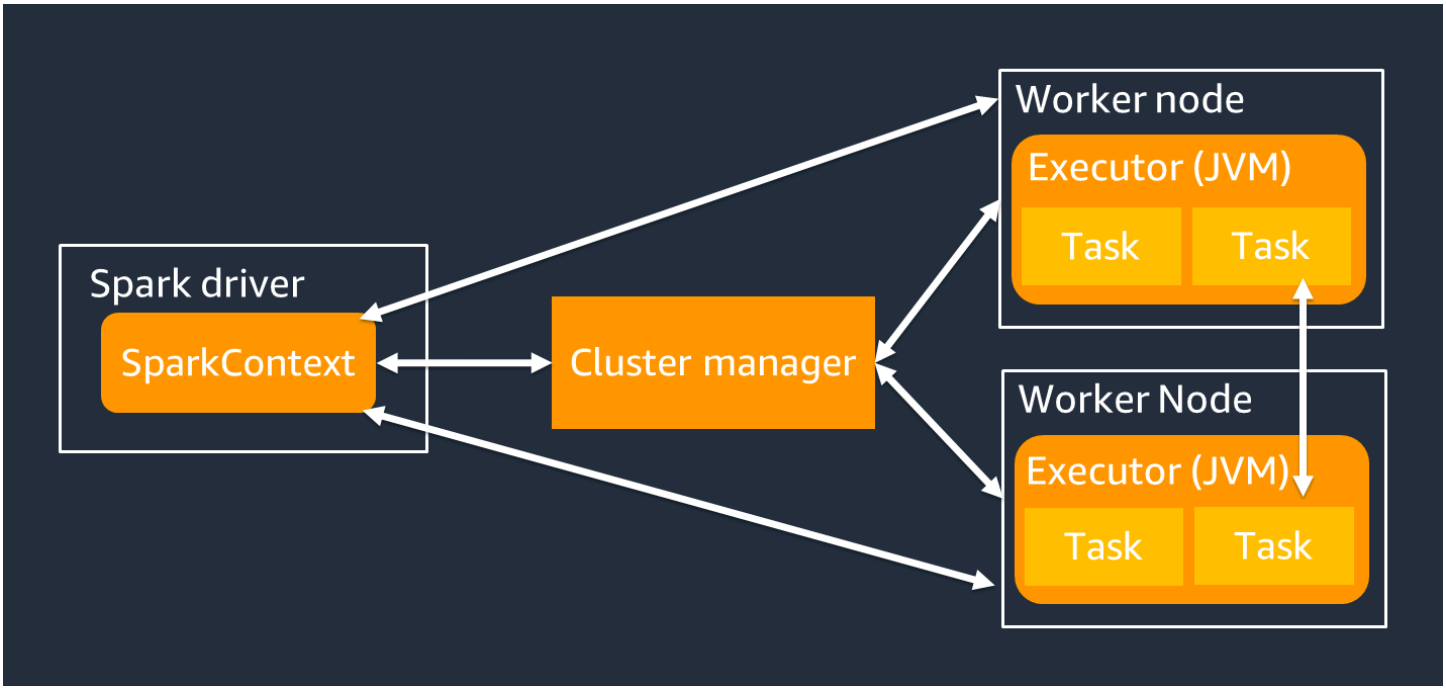
架构

Spark 驱动程序主要负责将您的 Spark 应用程序拆分为可由单个工作人员完成的任务。Spark 驱动程序有以下职责：

- `main()`在你的代码中运行
- 生成执行计划
- 与集群管理器一起配置 Spark 执行器，集群管理器管理集群上的资源
- 为 Spark 执行者安排任务和请求任务
- 管理任务进度和恢复

在作业运行中，您可以使用`SparkContext`对象与 Spark 驱动程序进行交互。

Spark 执行器是用于保存数据和运行从 Spark 驱动程序传递的任务的工作程序。Spark 执行者的数量将随着集群的大小而增加和减少。

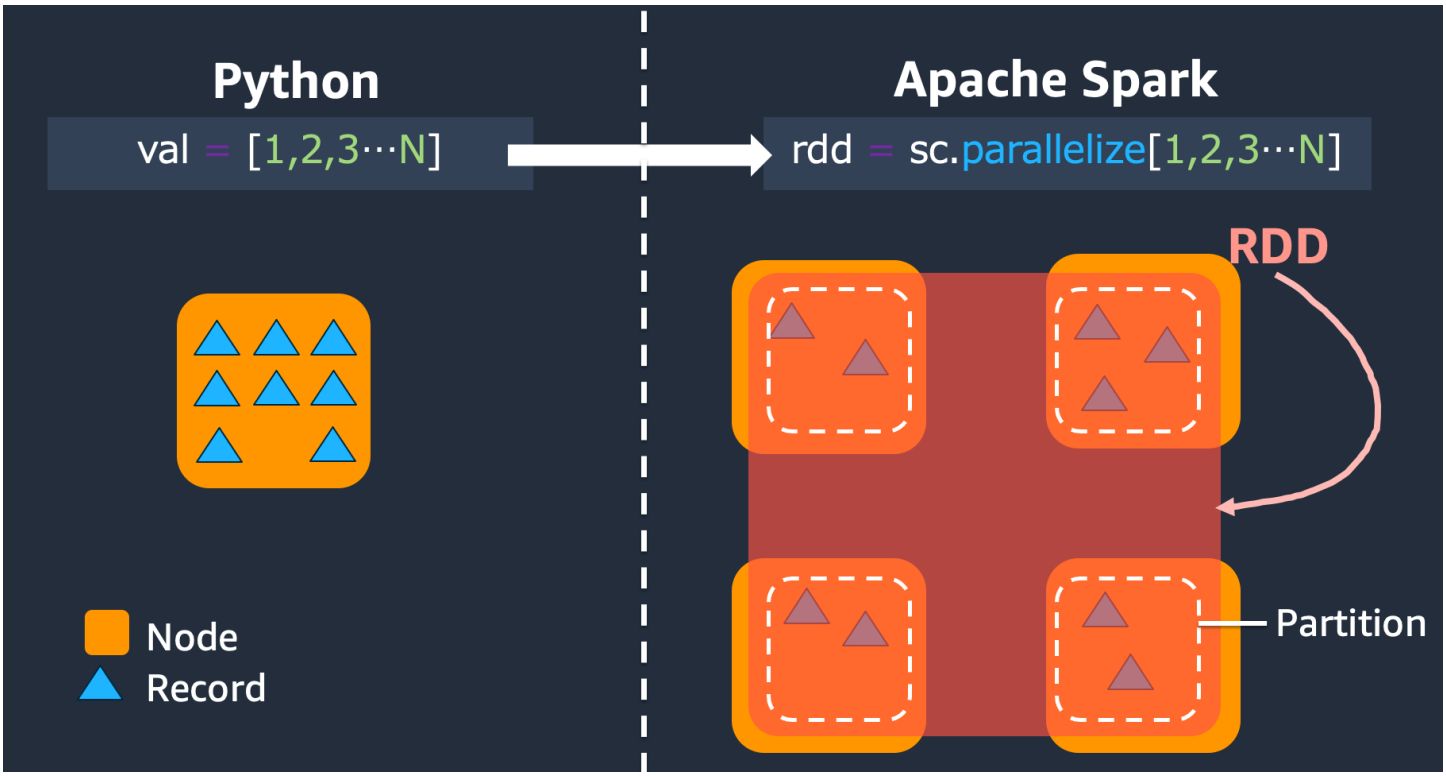


> **Note**
>
> Spark 执行器有多个插槽，因此可以并行处理多个任务。默认情况下，Spark 支持每个虚拟 CPU (vCPU) 内核执行一项任务。例如，如果执行器有四个 CPU 内核，则它可以同时运行四个任务。

弹性分布式数据集

Spark 负责存储和跟踪 Spark 执行器间的大型数据集的复杂工作。在为 Spark 作业编写代码时，无需考虑存储的细节。Spark 提供了弹性分布式数据集 (RDD) 抽象，这是一组可以并行操作的元素，可以跨集群的 Spark 执行器进行分区。

下图显示了 Python 脚本在典型环境中运行和在 Spark 框架 (PySpark) 中运行时，如何将数据存储在内存中的差异。



- Python — 使用 Python 脚本编写`val = [1,2,3...N]`可将数据保存在运行代码的单台计算机上的内存中。

- PySpark— Spark 提供 RDD 数据结构，用于加载和处理分布在多个 Spark 执行器上的内存中的数据。您可以使用诸如之类的代码生成 RDD`rdd = sc.parallelize[1,2,3...N]`，并且 Spark 可以自动在多个 Spark 执行器之间分发数据并将其保存到内存中。

  在许多 AWS Glue 工作中，你可以使用 RDD AWS Glue DynamicFrames和 Spark DataFrames。这些抽象允许您定义 RDD 中的数据架构，并使用这些附加信息执行更高级别的任务。由于它们在内部使用 RDD，因此在以下代码中，数据会透明地分发并加载到多个节点：

  - 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>")
    ```

RDD 具有以下功能：

- RDD 由分成多个部分的数据组成，称为分区。每个 Spark 执行器在内存中存储一个或多个分区，数据分布在多个执行器上。
- RDD 是不可变的，这意味着它们在创建后无法更改。要更改 a DataFrame，可以使用下一节中定义的转换。
- RDD 跨可用节点复制数据，因此它们可以自动从节点故障中恢复。

懒惰的评估

RDD 支持两种类型的操作：变换（从现有数据集创建新数据集）和操作（在对数据集运行计算后向驱动程序返回值）。

- 转换 — 由于 RDD 是不可变的，因此只能通过使用转换来更改它们。

  例如，`map`是一种转换，它将每个数据集元素传递给一个函数，然后返回一个表示结果的新 RDD。请注意，该`map`方法不返回输出。Spark 存储未来的抽象转换，而不是让你与结果进行交互。在你调用操作之前，Spark 不会对变换采取行动。

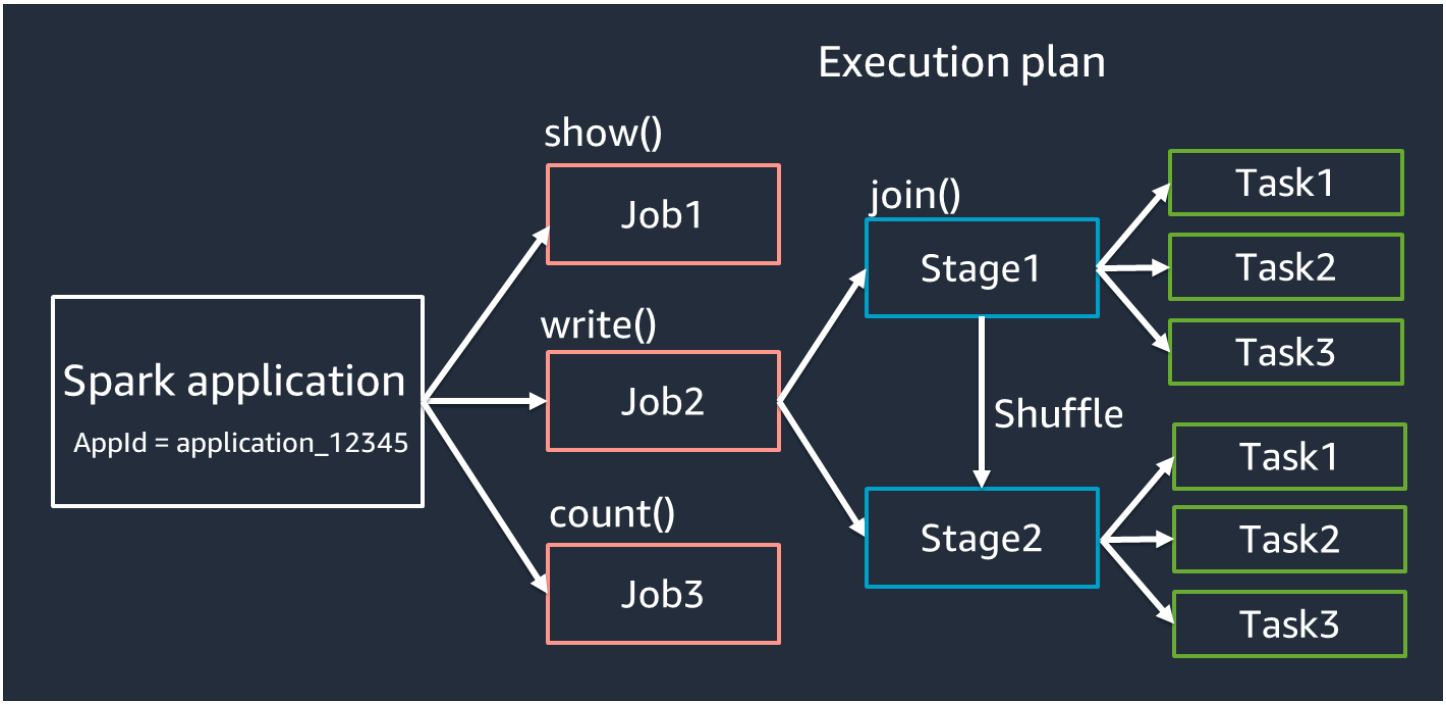
- 操作-使用转换，您可以制定逻辑转换计划。要启动计算，请运行诸如、writecountshow、或collect之类的操作。

  Spark 中的所有转换都是懒惰的，因为它们不会立即计算结果。相反，Spark 会记住应用于某些基础数据集的一系列转换，例如亚马逊简单存储服务 (Amazon S3) Service 对象。只有当操作要求将结果返回给驱动程序时，才会计算变换。这种设计使Spark能够更高效地运行。例如，假设通过map转换创建的数据集仅被大幅减少行数的转换所消耗，例如reduce。然后，您可以将经过两次转换的较小数据集传递给驱动程序，而不是传递较大的映射数据集。

Spark 应用程序的术语

本节介绍了 Spark 应用程序的术语。Spark 驱动程序创建执行计划，并以多种抽象形式控制应用程序的行为。以下术语对于使用 Spark UI 进行开发、调试和性能调整非常重要。

- 应用程序-基于 Spark 会话（Spark 上下文）。由唯一 ID 标识，例如<application_XXX>。
- 作业-基于为 RDD 创建的操作。一项作业由一个或多个阶段组成。
- 阶段 — 基于为 RDD 创建的洗牌。一个阶段由一个或多个任务组成。shuffle 是 Spark 用于重新分配数据的机制，以便在 RDD 分区中对数据进行不同的分组。某些变换（例如join()）需要随机播放。在 “优化洗牌” 调整练习中更详细地讨论了随机播放。
- 任务-任务是 Spark 计划的最小处理单位。为每个 RDD 分区创建任务，任务数是该阶段同时执行的最大数量。

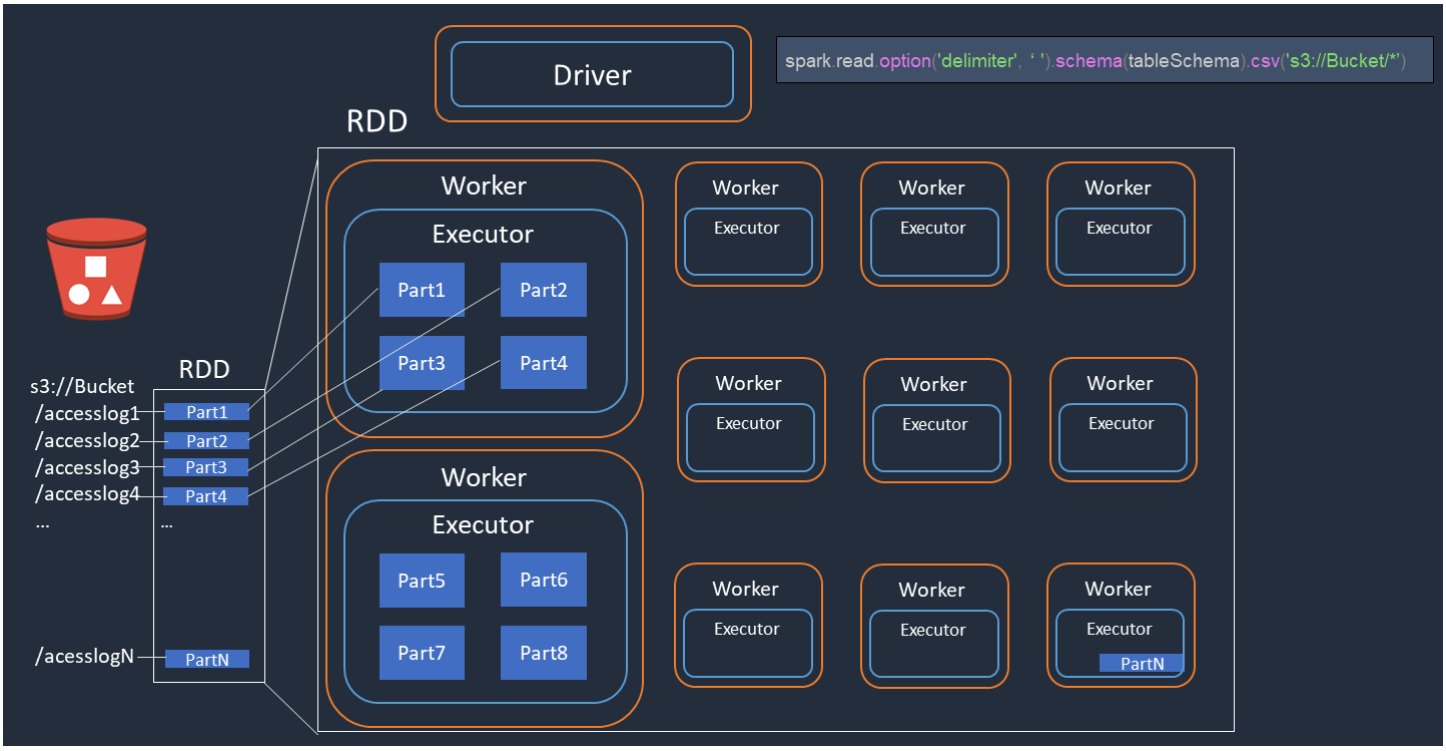


> **Note**
>
> 在优化并行度时，任务是需要考虑的最重要因素。任务的数量随RDD的数量而变化

并行

Spark 对加载和转换数据的任务进行并行处理。

举一个示例，您在 Amazon S3 上对访问日志文件（命名`accesslog1 ... accesslogN`）执行分布式处理。下图显示了分布式处理流程。

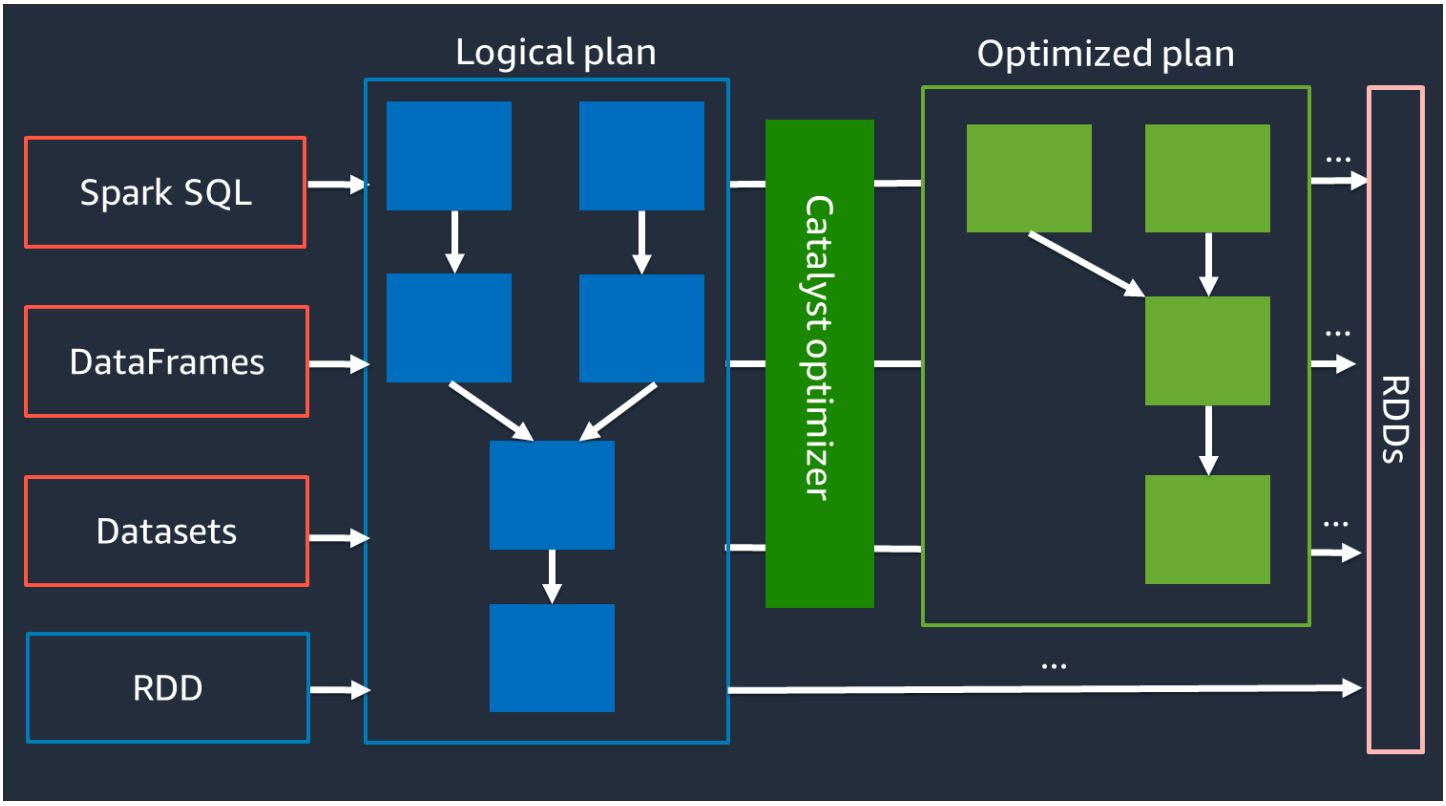


1. Spark 驱动程序创建执行计划，用于在许多 Spark 执行器之间进行分布式处理。
2. Spark 驱动程序根据执行计划为每个执行者分配任务。默认情况下，Spark 驱动程序会为每个 S3 对象 () 创建 RDD 分区（每个分区对应一个 Spark 任务`Part1 ... N`）。然后，Spark 驱动程序将任务分配给每个执行者。
3. 每个 Spark 任务都会下载其分配的 S3 对象，并将其存储在 RDD 分区的内存中。这样，多个 Spark 执行器就可以并行下载和处理其分配的任务。

有关初始分区数量和优化的更多详细信息，请参阅 Parallelize 任务部分。

催化剂优化器

在内部，Spark 使用名为 Catalyst 优化器的引擎来优化执行计划。Catalyst 有一个查询优化器，您可以在运行高级 Spark API（例如 Spark SQL 和数据集）时使用该优化器，如下图所示。DataFrame



由于 Catalyst 优化器不能直接与 RDD API 配合使用，因此高级别 API 通常比低级 RDD API 更快。对于复杂联接，Catalyst 优化器可以通过优化作业运行计划来显著提高性能。你可以在 Spark 用户界面的 SQL 选项卡上查看 Spark 作业的优化计划。

自适应查询执行

Catalyst 优化器通过名为“自适应查询执行”的过程执行运行时优化。自适应查询执行使用运行时统计信息在作业运行时重新优化查询的运行计划。Adaptive Query Execution 为性能挑战提供了多种解决方案，包括合并洗牌后的分区、将排序合并联接转换为广播联接以及倾斜联接优化，如以下各节所述。

自适应查询执行在 AWS Glue 3.0 及更高版本中可用，在 AWS Glue 4.0（Spark 3.3.0）及更高版本中默认启用。可以在代码`spark.conf.set("spark.sql.adaptive.enabled", "true")`中使用来开启和关闭自适应查询执行。

合并洗牌后的分区

此功能根据输出统计信息在每次洗牌后减少 RDD 分区（合并）。`map`它简化了运行查询时对 shuffle 分区号的调整。您无需设置 shuffle 分区号即可适合您的数据集。在你有足够大的初始洗牌分区数之后，Spark 可以在运行时选择正确的洗牌分区号。

当`spark.sql.adaptive.enabled`和`spark.sql.adaptive.coalescePartitions.enabled`都设置为 true 时，将启用合并后洗牌分区。有关更多信息，请参阅 Apache Spark 文档。

将排序合并联接转换为广播联接

此功能可识别您何时连接两个大小截然不同的数据集，并根据该信息采用更有效的联接算法。有关更多详细信息，请参阅 Apache Spark 文档。“优化洗牌” 部分讨论了加入策略。

倾斜连接优化

数据倾斜是 Spark 作业最常见的瓶颈之一。它描述了一种情况，即数据偏向特定 RDD 分区（以及随之而来的特定任务），这会延迟应用程序的总体处理时间。这通常会降低联接操作的性能。倾斜联接优化功能通过将倾斜的任务拆分（并在需要时复制）为大小大致相等的任务来动态处理排序合并联接中的偏差。

此功能在设置`spark.sql.adaptive.skewJoin.enabled`为 true 时启用。有关更多详细信息，请参阅 Apache Spark 文档。数据偏斜将在优化洗牌一节中进一步讨论。

使用 Spark 用户界面调查性能问题

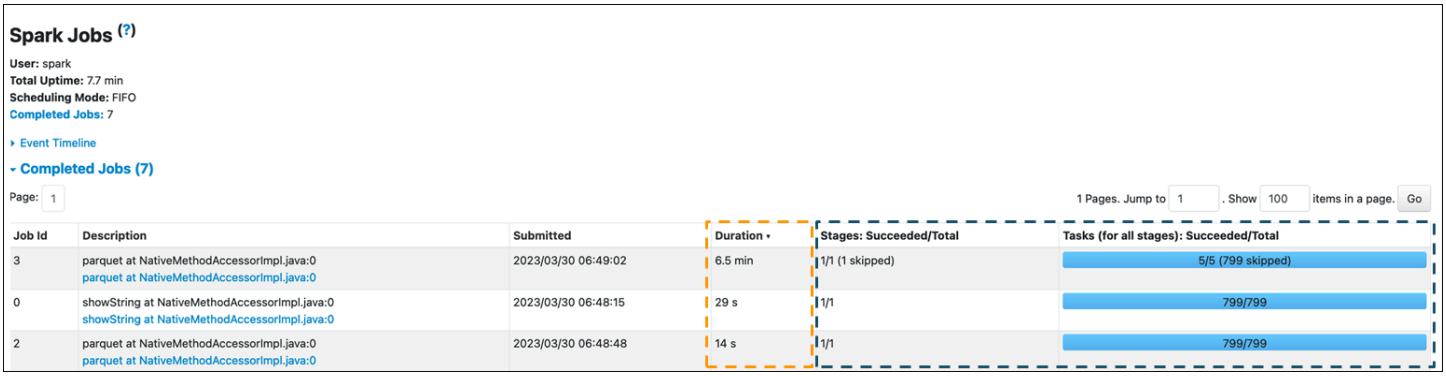
在应用任何最佳实践来调整 AWS Glue 作业性能之前，我们强烈建议您对性能进行分析并确定瓶颈。这将帮助你专注于正确的事情。

为了便于快速分析，Amazon CloudWatch 指标提供了您的工作指标的基本视图。Spark 用户界面为性能调整提供了更深入的视图。要将 Spark 用户界面与配合使用 AWS Glue，必须为 AWS Glue 作业启用 Spark 用户界面。熟悉 Spark 用户界面后，请遵循调整 Spark 作业性能的策略，根据您的发现识别和减少瓶颈的影响。

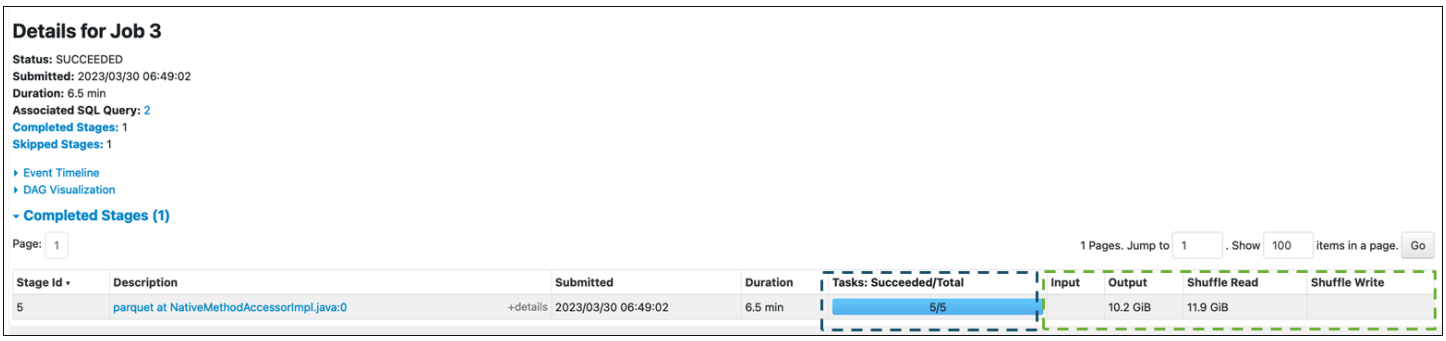
使用 Spark 用户界面识别瓶颈

当你打开 Spark 用户界面时，Spark 应用程序会在表格中列出。默认情况下， AWS Glue 作业的应用程序名称为`nativespark-<Job Name>-<Job Run ID>`。根据作业运行 ID 选择目标 Spark 应用程序以打开 “作业” 选项卡。未完成的作业运行（例如流式处理作业运行）列在显示未完成的应用程序中。

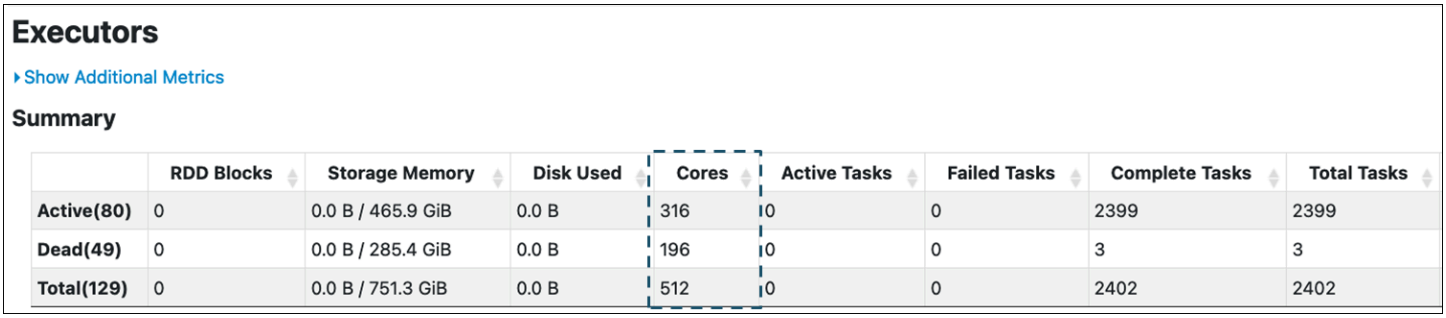
“作业” 选项卡显示 Spark 应用程序中所有作业的摘要。要确定任何阶段或任务失败，请检查任务总数。要找到瓶颈，请选择 “持续时间” 进行排序。选择 “描述” 列中显示的链接，深入了解长时间运行的作业的详细信息。



Job 的详细信息页面列出了各个阶段。在此页面上，您可以看到整体见解，例如持续时间、成功任务数量和任务总数、输入和输出数量以及随机读取和随机写入量。



“执行器” 选项卡详细显示 Spark 集群容量。您可以查看内核总数。以下屏幕截图中显示的集群共包含 316 个活动内核和 512 个内核。默认情况下，每个内核可以同时处理一个 Spark 任务。



根据 Job 详细信息页面上**5/5**显示的值，第 5 阶段是最长的阶段，但它只使用 512 个内核中的 5 个。由于此阶段的并行度非常低，但需要花费大量时间，因此您可以将其识别为瓶颈。为了提高性能，您需要了解原因。要详细了解如何识别和减少常见性能瓶颈的影响，请参阅调整 Spark 作业性能的策略。

调整 Spark 作业绩效的策略

准备优化参数时，请遵循以下最佳实践：

- 应首先确定性能目标，然后再开始确定性能问题。
- 应首先使用指标来确定问题，然后再尝试更改优化参数。

为确保在优化作业时获得稳定一致的结果，应为优化工作制定基线策略。

性能优化的基准策略

性能优化的工作流通常如下：

1. 确定性能目标。
2. 衡量指标。
3. 识别瓶颈。
4. 减少瓶颈的影响。
5. 重复第 2-4 步，直到达到预期目标为止。

首先，确定您的绩效目标。例如，您的目标之一可能是在 3 小时内完成 AWS Glue 作业。定义目标后，衡量工作绩效指标。确定指标的趋势和瓶颈以实现目标。特别是，识别瓶颈对于故障排除、调试和性能调整最为重要。在 Spark 应用程序运行期间，Spark 会在 Spark 事件日志中记录每个任务的状态和统计信息。

在中 AWS Glue，你可以通过 Spark 历史服务器提供的 Spark Web 用户界面查看 Spark 指标。 AWS Glue for Spark 任务可以将 Spark 事件日志发送到你在 Amazon S3 中指定的位置。 AWS Glue 还提供了示例AWS CloudFormation 模板和 Dockerfile，用于在亚马逊EC2实例或本地计算机上启动 Spark 历史服务器，因此您可以将 Spark 用户界面与事件日志一起使用。

在确定绩效目标并确定评估这些目标的指标后，您可以使用以下各节中的策略开始识别和修复瓶颈。

Spark 作业性能的调整实践

您可以使用以下策略对 Spark 作业 AWS Glue 进行性能调整：

- AWS Glue 资源：

  - 扩展集群容量
  - 使用最新 AWS Glue 版本
- 火花应用程序：
  - 减少数据扫描量
  - 并行化任务
  - 优化洗牌
  - 最大限度减少规划开销
  - 优化用户定义的函数

在使用这些策略之前，您必须有权访问 Spark 作业的指标和配置。你可以在AWS Glue 文档中找到这些信息。

从 AWS Glue 资源的角度来看，您可以通过添加 AWS Glue 工作人员和使用最新 AWS Glue 版本来提高性能。

从 Apache Spark 应用程序的角度来看，你可以使用几种可以提高性能的策略。如果将不必要的数据加载到 Spark 集群中，则可以将其移除以减少加载的数据量。如果您的 Spark 集群资源未得到充分利用，并且数据 I/O 较低，则可以确定要并行处理的任务。如果连接等繁重的数据传输操作需要大量时间，则可能还需要对其进行优化。您还可以优化作业查询计划或降低单个 Spark 任务的计算复杂性。

要有效地应用这些策略，您必须通过查阅您的指标来确定它们何时适用。有关更多详细信息，请参阅以下各节。这些技术不仅适用于性能调整，还可以解决诸如 out-of-memory (OOM) 错误之类的典型问题。

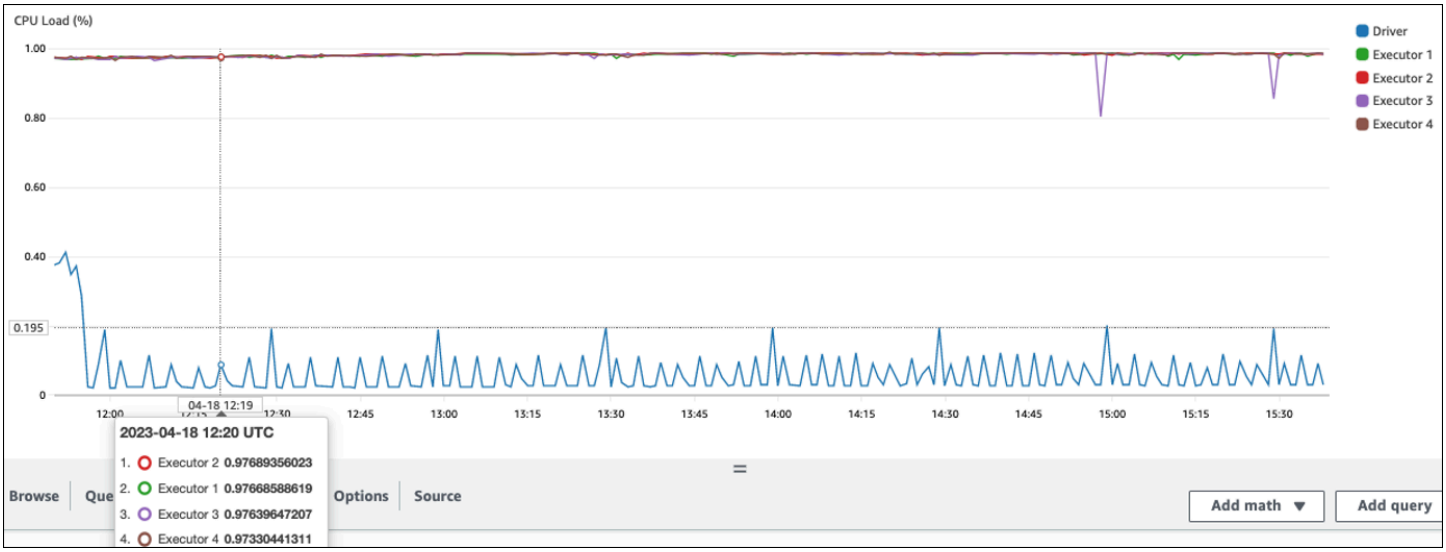
扩展集群容量

如果你的任务花费了太多时间，但执行者消耗了足够的资源，而且 Spark 创建的任务量与可用内核相比很大，可以考虑扩展集群容量。要评估这是否合适，请使用以下指标。

CloudWatch 指标

- 检查CPU负载和内存利用率以确定执行程序是否消耗了足够的资源。
- 检查作业运行了多长时间，以评估处理时间是否过长，无法实现您的绩效目标。

在以下示例中，四个执行器以超过 97% 的CPU负载运行，但是大约三个小时后处理仍未完成。

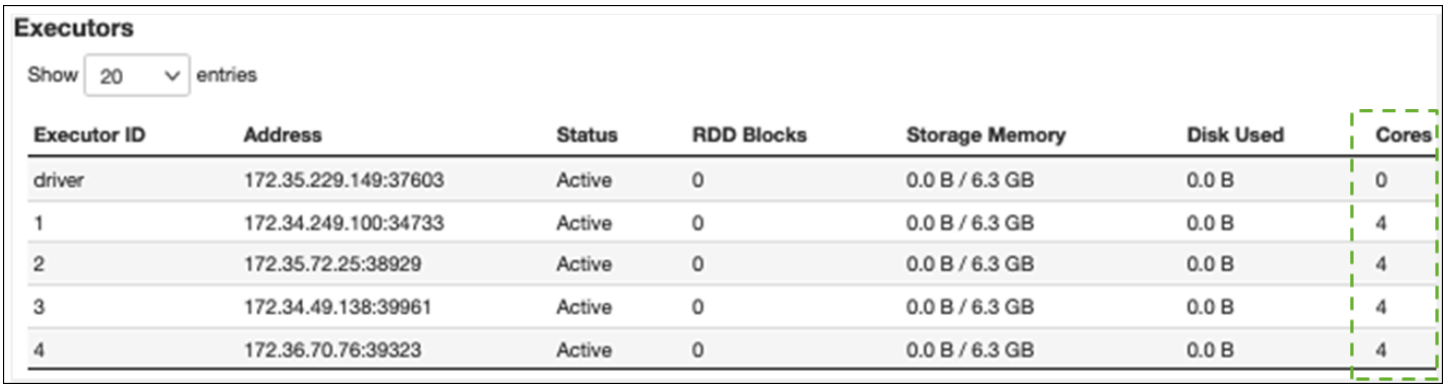


> **Note**
>
> 如果CPU负载较低，您可能无法从扩展集群容量中受益。

Spark UI

在 Jo b 选项卡或 St age 选项卡上，您可以看到每个作业或阶段的任务数。在以下示例中，Spark 创建了58100任务。

在 Executor 选项卡上，您可以看到执行者和任务的总数。在以下屏幕截图中，每个 Spark 执行器都有四个内核，可以同时执行四个任务。



在此示例中，Spark 任务的数量（58100)远大于执行器可以同时处理的 16 个任务（4 个执行器 × 4 个内核）。

如果您发现这些症状，请考虑扩展集群。您可以使用以下选项扩展集群容量：

- 启用 AWS Glue Auto Sc aling — 在 3.0 或更高 AWS Glue 版本中，A uto Scaling 可用于 AWS Glue 提取、转换和加载 (ETL) 以及流式处理作业。 AWS Glue 根据每个阶段的分区数量或作业运行时生成微批量的速率，自动在集群中添加和移除工作程序。

  如果您发现即使启用了 Auto Scaling，工作人员数量也不会增加，请考虑手动添加工作线程。但是，请注意，手动扩展一个阶段可能会导致许多工作人员在后期阶段处于闲置状态，而性能提升为零的成本更高。

  启用 Auto Scaling 后，你可以在执行者指标中看到 CloudWatch 执行者的数量。使用以下指标来监控 Spark 应用程序中对执行者的需求：

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

  有关指标的更多信息，请参阅AWS Glue 使用 Amazon CloudWatch 指标进行监控。
- 向外扩展：增加 AWS Glue 员工人数 — 您可以手动增加 AWS Glue 员工人数。仅在观察到闲置工作人员之前添加工作人员。那时，增加更多员工将增加成本，而不会改善业绩。有关更多信息，请参阅并行化任务。
- 向上扩展：使用更大的工作器类型 — 您可以手动更改 AWS Glue 工作程序的实例类型，以使用具有更多内核、内存和存储空间的工作程序。较大的工作线程类型使您可以垂直扩展和运行密集型数据集成作业，例如内存密集型数据转换、倾斜聚合以及涉及 PB 级数据的实体检测检查。

  在 Spark 驱动程序需要更大容量的情况下，向上扩展还有帮助，例如，因为任务查询计划非常大。有关工作人员类型和绩效的更多信息，请参阅 AWS 大数据博客文章使用新的大型工作器 AWS Glue 类型 G.4X 和 G.8X 来扩展 Apache Spark 作业。

  使用较大的工作人员还可以减少所需的员工总数，从而减少密集型操作（例如加入）中的混乱，从而提高绩效。

使用最新 AWS Glue 版本

我们建议使用最新 AWS Glue 版本。每个版本中都内置了多项优化和升级，可以自动提高工作性能。例如， AWS Glue 4.0 提供了以下新功能：

- 全新优化的 Apache Spark 3.0 运行时 — AWS Glue 4.0 在 Apache Spark 3.3.0 运行时的基础上构建，为开源 Spark 带来了与开源 Spark 相当的性能改进。Spark 3.0 运行时建立在 Spark 2.x 的许多创新之上。
- 增强的亚马逊 Redshift 连接器 — AWS Glue 4.0 及更高版本为 Apache Spark 提供了亚马逊 Redshift 集成。该集成建立在现有的开源连接器之上，并增强了其性能和安全性。该集成可帮助应用程序的运行速度提高多达 10 倍。有关更多信息，请参阅有关亚马逊 Redshift 与 Apache Spark 集成的博客文章。
- SIMD基于执行的矢量化读取CSV和JSON数据 — 3.0 及更高 AWS Glue 版本添加了优化的读取器，与基于行的读取器相比，可以显著提高整体作业性能。有关CSV数据的更多信息，请参阅使用矢量化SIMDCSV阅读器优化读取性能。有关JSON数据的更多信息，请参阅使用具有 Apache Arrow 列SIMDJSON格式的矢量化阅读器。

每个 AWS Glue 版本都将包括此类升级，包括连接器、驱动程序和库更新。有关更多信息，请参阅AWS Glue 版本和将 AWS Glue 任务迁移到 AWS Glue 版本 4.0。

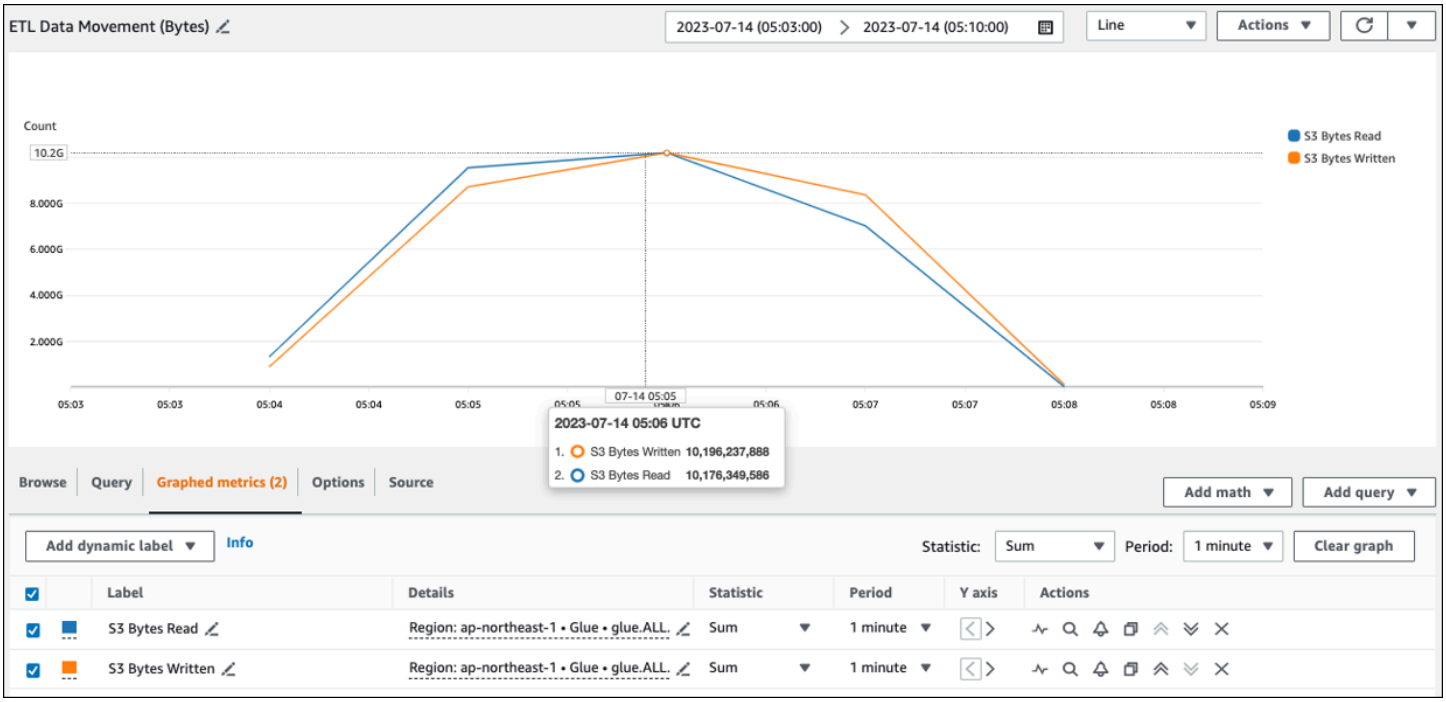
减少数据扫描量

首先，请考虑仅加载所需的数据。只需减少每个数据源加载到 Spark 集群中的数据量，即可提高性能。要评估这种方法是否合适，请使用以下指标。

您可以在 Spark 用户界面中查看从 Amazon S3 读取的字节的CloudWatch指标和更多详细信息，如Spark 用户界面部分所述。

CloudWatch 指标

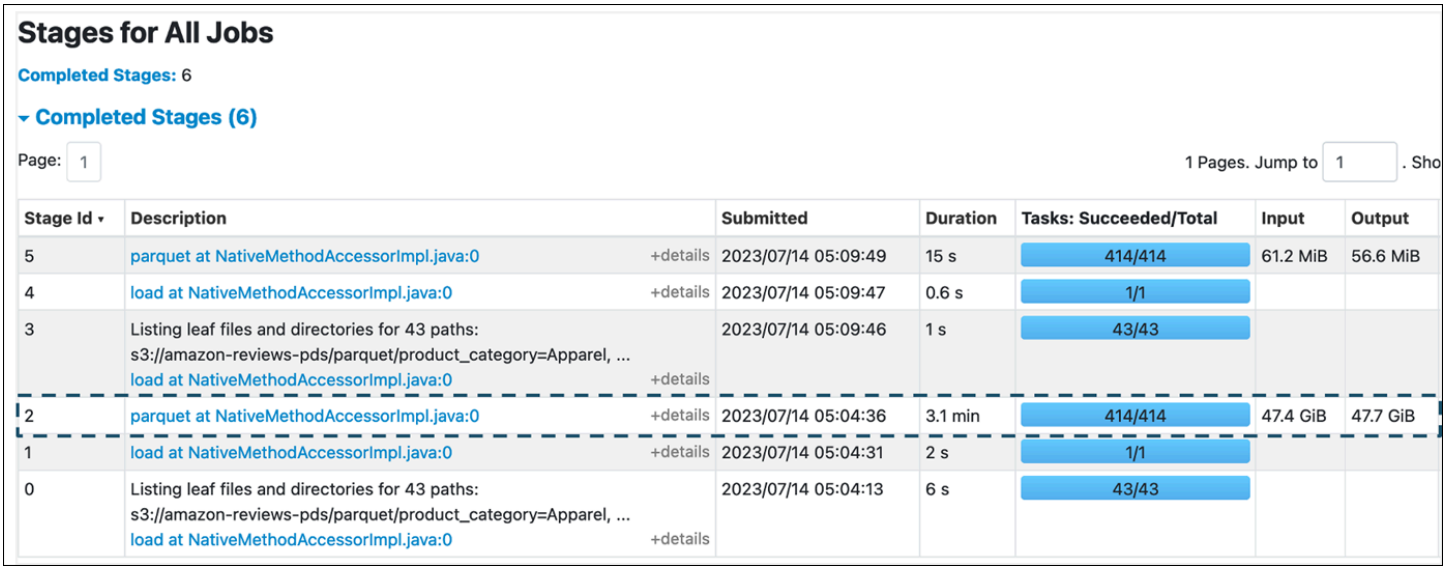
您可以在ETL数据移动（字节）中看到从 Amazon S3 读取的大致大小。该指标显示自上次报告以来所有执行者从 Amazon S3 读取的字节数。您可以使用它来监控来自 Amazon S3 ETL 的数据移动，也可以将外部数据源的读取率与摄取速率进行比较。



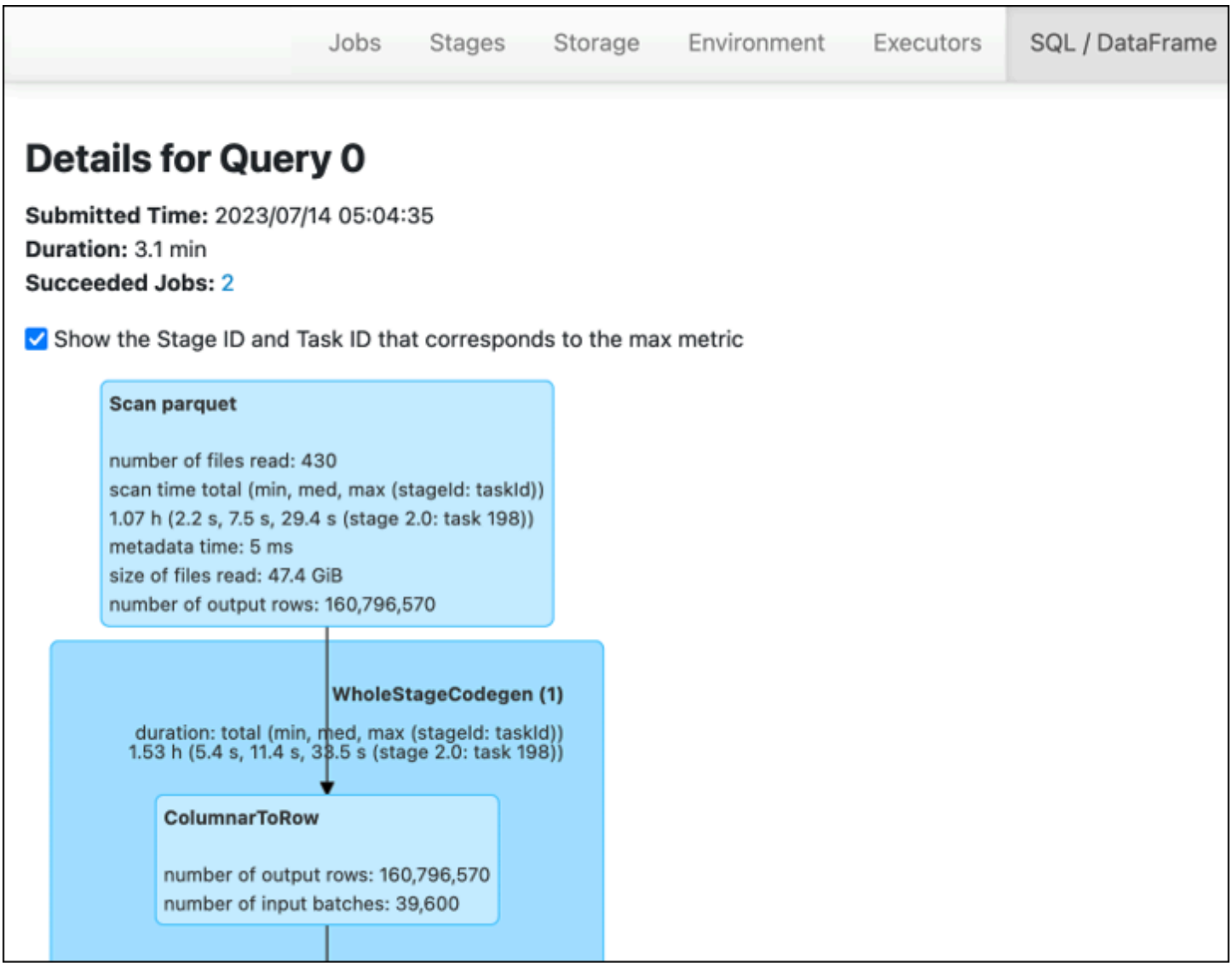
如果您观察到的 S3 字节读取数据点比预期的要大，请考虑以下解决方案。

Spark UI

在 for Spark 用户界面的舞台选项卡上，你可以看到输入和输出的大小。 AWS Glue 在以下示例中，第 2 阶段读取 47.4 GiB 的输入和 47.7 GiB 的输出，而第 5 阶段读取 61.2 MiB 的输入和 56.6 MiB 的输出。



当您在 AWS Glue 工作中使用 Spark SQL 或 DataFrame 方法时，SQL/D ataFrame 选项卡会显示有关这些阶段的更多统计信息。在本例中，第 2 阶段显示读取的文件数：430，读取的文件大小：4 7.4 GiB，输出行数：160 ,796,570。



如果您发现正在读取的数据和正在使用的数据在大小上存在很大差异，请尝试以下解决方案。

Amazon S3

要减少从 Amazon S3 读取数据时加载到任务中的数据量，请考虑数据集的文件大小、压缩、文件格式和文件布局（分区）。 AWS Glue for Spark 作业通常用于ETL处理原始数据，但是为了实现高效的分布式处理，您需要检查数据源格式的特征。

- 文件大小-我们建议将输入和输出的文件大小保持在适中的范围内（例如 128 MB）。文件太小和文件太大可能会导致问题。

大量小文件会导致以下问题：

- Amazon S3 上的网络 I/O 负载很大，这是因为向许多对象发出请求（例如`ListGet`、或`Head`）需要开销（相比之下，只有少数对象存储相同数量的数据）。
- Spark 驱动程序承受沉重的 I/O 和处理负载，这将生成许多分区和任务，并导致并行度过高。

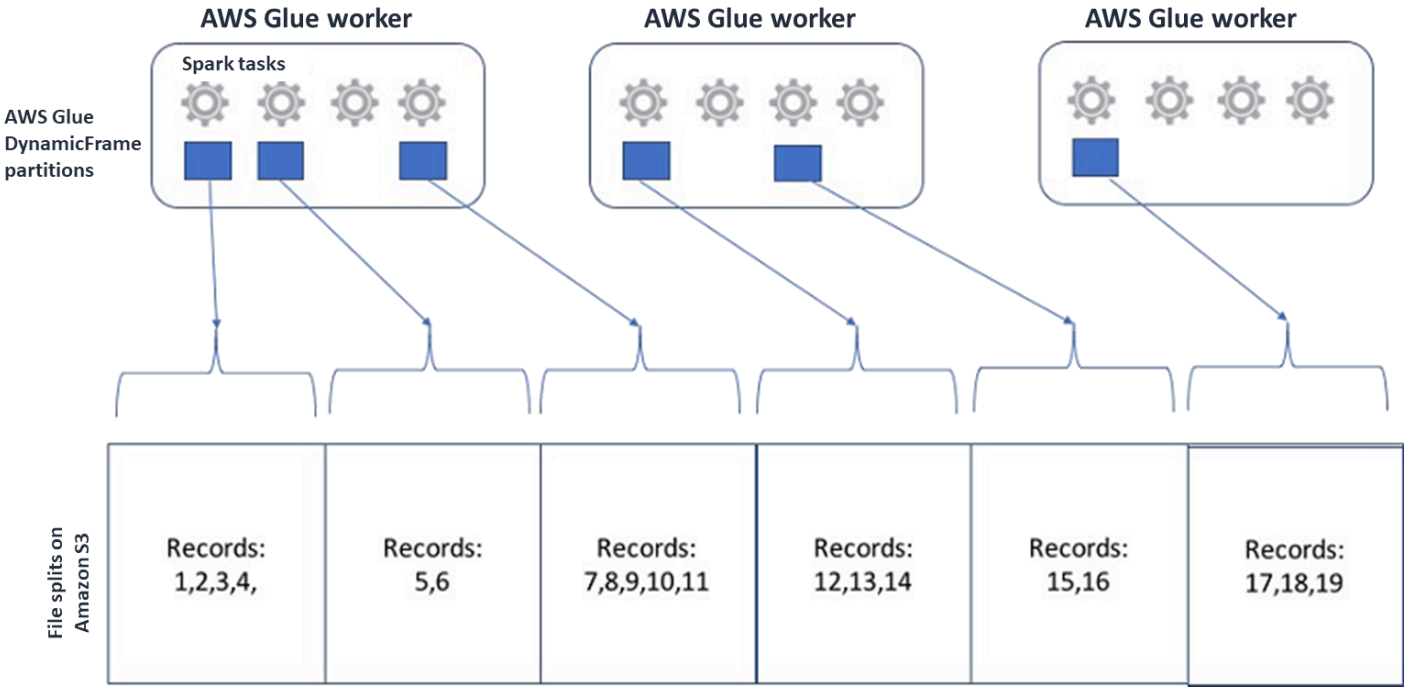
另一方面，如果您的文件类型不可拆分（例如 gzip），并且文件太大，则 Spark 应用程序必须等到单个任务完成对整个文件的读取。

要减少为每个小文件创建 Apache Spark 任务时产生的过多并行度，请使用文件分组。DynamicFrames这种方法减少了 Spark 驱动程序出现OOM异常的机会。要配置文件分组，请设置`groupFiles`和`groupSize`参数。以下代码示例使用带有这些参数的ETL脚本 AWS Glue DynamicFrame API中的。

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

- 压缩-如果您的 S3 对象以数百兆字节为单位，请考虑对其进行压缩。有多种压缩格式，可以大致分为两种类型：
  - 不可拆分的压缩格式（例如 gzip）要求由一个工作程序解压缩整个文件。
  - 可拆分的压缩格式，例如 bzip2 或LZO（已编入索引），允许对文件进行部分解压缩，这种解压缩可以并行化。

  对于 Spark（以及其他常见的分布式处理引擎），您需要将源数据文件拆分为引擎可以并行处理的块。这些单位通常被称为拆分单元。在您的数据采用可拆分格式后，经过优化的 AWS Glue 读取器可以通过提供仅检索特定块的`Range`选项`GetObject`API来检索 S3 对象的拆分。请考虑下图，看看这在实践中是如何运作的。



只要文件大小最优或者文件是可拆分的，压缩数据就可以显著加快应用程序的速度。较小的数据大小会减少从 Amazon S3 扫描的数据以及从 Amazon S3 到 Spark 集群的网络流量。另一方面，压缩和解压缩数据需要更多CPU精力。所需的计算量随压缩算法的压缩率而变化。在选择可拆分压缩格式时，请考虑这种权衡取舍。

> **Note**
>
> 虽然 gzip 文件通常不可拆分，但你可以使用 gzip 压缩单个拼花方块，这些块可以并行化。

- 文件格式-使用分栏格式。Apache Parq uet 和 Apache ORC 是流行的列式数据格式。通过采用基于列的压缩、根据每列的数据类型对每列进行编码和压缩，从而有效地ORC存储和存储数据。有关 Parquet 编码的更多信息，请参阅 Par quet 编码定义。Parquet 文件也可以拆分。

  列格式按列对值进行分组，并将它们一起存储在块中。使用列式格式时，可以跳过与不打算使用的列相对应的数据块。Spark 应用程序只能检索您需要的列。通常，更好的压缩率或跳过数据块意味着从 Amazon S3 读取更少的字节，从而提高性能。这两种格式还支持以下下推方法来减少 I/O：

  - 投影下推-投影下推是一种仅检索应用程序中指定的列的技术。您可以在 Spark 应用程序中指定列，如以下示例所示：
    - DataFrame 示例：`df.select("star_rating")`
    - 火花SQL示例：`spark.sql("select start_rating from <table>")`

- 谓词下推 — 谓词下推是一种高效处理和子句的技术。WHERE GROUP BY两种格式都有表示列值的数据块。每个区块都包含该区块的统计信息，例如最大值和最小值。Spark 可以根据应用程序中使用的过滤器值使用这些统计信息来确定是应该读取还是跳过该块。要使用此功能，请在条件中添加更多过滤器，如以下示例所示，如下所示：
  - DataFrame 示例：`df.select("star_rating").filter("star_rating < 2")`
  - 火花SQL示例：`spark.sql("select * from <table> where star_rating < 2")`
- 文件布局-通过根据数据的使用方式将 S3 数据存储到不同路径中的对象，您可以高效地检索相关数据。有关更多信息，请参阅 Amazon S3 文档中的使用前缀组织对象。 AWS Glue 支持以格式将密钥和值存储到 Amazon S3 前缀中`key=value`，按照 Amazon S3 路径对数据进行分区。通过对数据进行分区，您可以限制每个下游分析应用程序扫描的数据量，从而提高性能并降低成本。有关更多信息，请参阅中的管理ETL输出分区 AWS Glue。

  分区将表分成不同的部分，并根据年、月和日等列值将相关数据保存在分组文件中，如以下示例所示。

```
# 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
...
```

  您可以使用中的表对数据集进行建模，从而为数据集定义分区 AWS Glue Data Catalog。然后，您可以使用分区修剪来限制数据扫描量，如下所示：

  - 对于 AWS Glue DynamicFrame，设置`push_down_predicate`（或`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'",
)
```

  - 对于 Spark DataFrame，请设置一个固定的路径来修剪分区。

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

  - 对于 SparkSQL，您可以将 where 子句设置为从数据目录中删除分区。

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

- 要在写入数据时按日期进行分区 AWS Glue，partitionKeys请在 DynamicFrame 或 partitionBy() 中 DataFrame 设置列中的日期信息，如下所示。
  - 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>/')
```

这可以提高输出数据使用者的性能。

如果您无权更改创建输入数据集的管道，则无法选择分区。相反，您可以使用 glob 模式排除不需要的 S3 路径。在 DynamicFrame读入时设置排除项。例如，以下代码不包括 2023 年第 1 个月到 09 个月中的天数。

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

您还可以在数据目录的表属性中设置排除项：

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

- Amazon S3 分区过多 — 避免将 Amazon S3 数据分区到包含各种值的列上，例如包含数千个值的 ID 列。这可能会大大增加存储桶中的分区数量，因为可能的分区数量是您分区所依据的所有字段的乘积。分区过多可能会导致以下情况：
  - 从数据目录中检索分区元数据的延迟增加
  - 小文件数量增加，这需要更多的 Amazon S3 API 请求（`ListGet`、和`Head`）

  例如，当您在`partitionBy`或中设置日期类型时`partitionKeys`，诸如之类的日期级分区`yyyy/mm/dd`适用于许多用例。但是，`yyyy/mm/dd/<ID>`可能会生成太多的分区，以至于会对整体性能产生负面影响。

  另一方面，某些用例（例如实时处理应用程序）需要许多分区，例如`yyyy/mm/dd/hh`。如果您的用例需要大量分区，请考虑使用AWS Glue 分区索引来减少从数据目录检索分区元数据的延迟。

数据库和 JDBC

要减少从数据库检索信息时的数据扫描，可以在查询中指定`where`谓词（或子句）。SQL不提供SQL接口的数据库将提供自己的查询或筛选机制。

使用 Java 数据库连接 (JDBC) 连接时，请提供包含以下参数`where`子句的选择查询：

- 对于 DynamicFrame，请使用sampleQuery选项。使用时`create_dynamic_frame.from_catalog`，按如下方式配置`additional_options`参数。

```
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"
)
```

  何时`using create_dynamic_frame.from_options`，按如下方式配置`connection_options`参数。

```
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
    }
)
```

- 对于 DataFrame，请使用查询选项。

```
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()
```

- 对于亚马逊 Redshift，请使用 AWS Glue 4.0 或更高版本来利用亚马逊 Redshift Spark 连接器中的下推支持。

```
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"}
)
```

- 对于其他数据库，请查阅该数据库的文档。

AWS Glue 选项

- 要避免对所有连续运行的作业进行全面扫描，并仅处理上次作业运行期间不存在的数据，请启用作业书签。
- 要限制要处理的输入数据的数量，请使用作业书签启用限定执行。这有助于减少每次作业运行的扫描数据量。

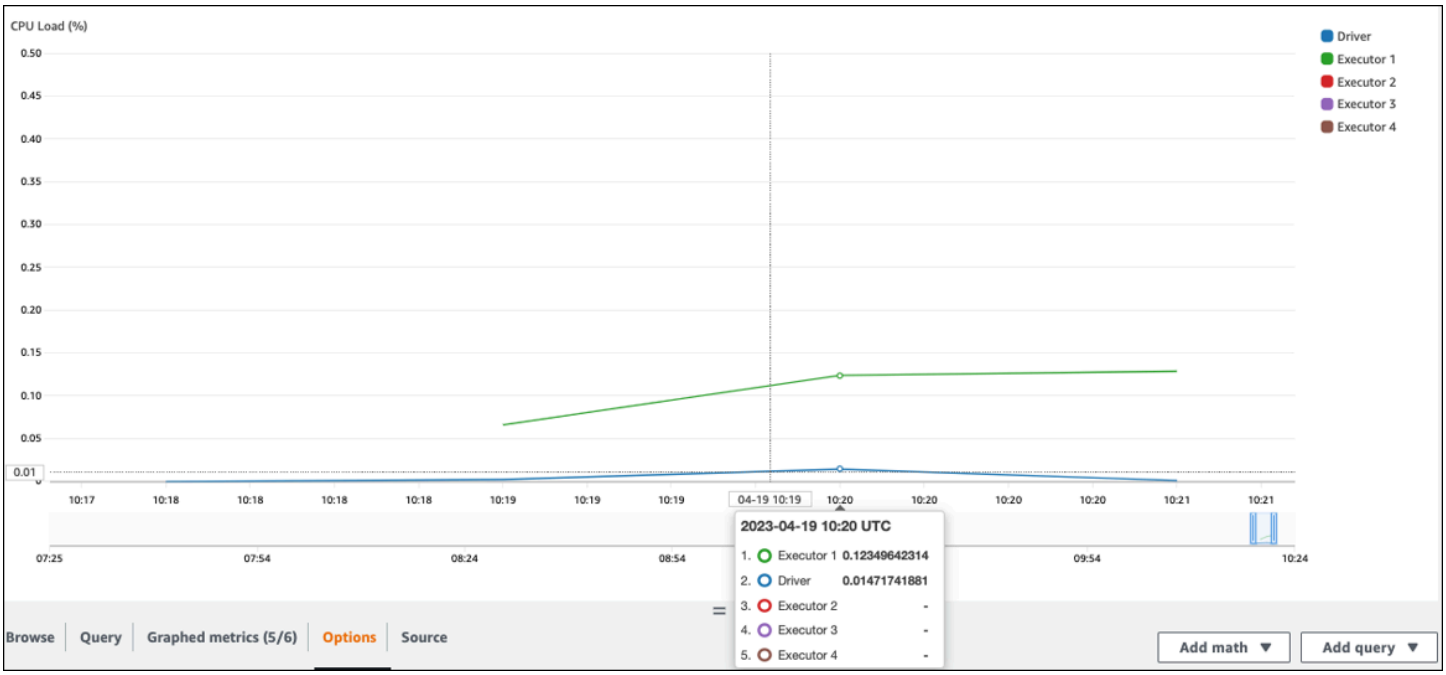
并行化任务

要优化性能，必须对数据加载和转换的任务进行并行处理。正如我们在 Apache Spark 的关键主题中所讨论的那样，弹性分布式数据集 (RDD) 分区的数量很重要，因为它决定了并行度。Spark 创建的每个任务都对应于一个 1:1 的RDD分区。要获得最佳性能，您需要了解RDD分区数量是如何确定的，以及如何优化分区数量。

如果您没有足够的并行度，则将在CloudWatch指标和 Spark UI 中记录以下症状。

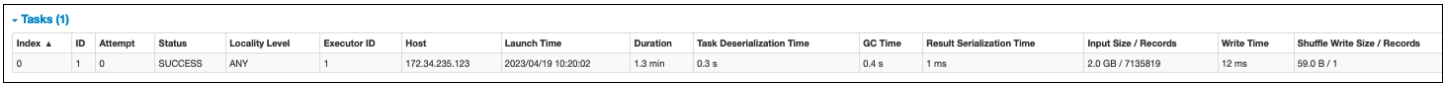
CloudWatch 指标

检查负CPU载和内存利用率。如果某些执行器在工作的某个阶段没有进行处理，那么改进并行度是合适的。在本例中，在可视化时间范围内，E xecutor 1 正在执行一项任务，但其余的执行者（2、3 和 4）却没有。你可以推断出 Spark 驱动程序没有为这些执行者分配任务。



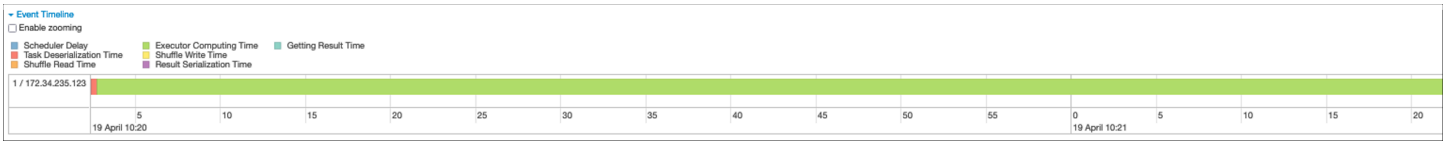
Spark UI

在 Spark 用户界面的舞台选项卡上，您可以看到一个阶段中的任务数量。在本例中，Spark 只执行了一项任务。


Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	Task Deserialization Time	GC Time	Result Serialization Time	Input Size / Records	Write Time	Shuffle Write Size / Records
0	1	0	SUCCESS	ANY	1	172.34.235.123	2023/04/19 10:20:02	1.3 min	0.3 s	0.4 s	1 ms	2.0 GB / 7135819	12 ms	59.0 B / 1

此外，事件时间轴显示 E xecutor 1 正在处理一项任务。这意味着该阶段的工作完全由一个执行人完成，而其他执行人则处于闲置状态。



如果您发现这些症状，请针对每个数据源尝试以下解决方案。

并行处理来自 Amazon S3 的数据加载

要并行处理从 Amazon S3 加载的数据，请先检查默认分区数。然后，您可以手动确定分区的目标数量，但一定要避免分区过多。

确定默认分区数

对于 Amazon S3，Spark RDD 分区的初始数量（每个分区对应于一个 Spark 任务）由您的 Amazon S3 数据集的特征（例如格式、压缩和大小）决定。当您使用存储在 Amazon S3 DataFrame 中的CSV对象创建 AWS Glue DynamicFrame或 Spark 时，初始RDD分区数 (NumPartitions) 可以大致计算如下：

- 对象大小 <= 64 MB：`NumPartitions = Number of Objects`
- 对象大小 > 64 MB：`NumPartitions = Total Object Size / 64 MB`
- 不可拆分 (gzip)：`NumPartitions = Number of Objects`

如减少数据扫描量部分所述，Spark 将大型 S3 对象分成可以并行处理的拆分。当对象大于拆分大小时，Spark 会拆分对象，并为每次拆RDD分创建一个分区（和任务）。Spark 的拆分大小取决于您的数据格式和运行时环境，但这是一个合理的起始近似值。有些对象使用不可拆分的压缩格式（例如 gzip）进行压缩，因此 Spark 无法对其进行拆分。

该`NumPartitions`值可能会有所不同，具体取决于您的数据格式、压缩率、 AWS Glue 版本、 AWS Glue 工作器数量和 Spark 配置。

例如，当你使用 Spark 加载一个 10 GB 的`csv.gz`对象时 DataFrame，Spark 驱动程序将只创建一个 RDD分区 (`NumPartitions=1`)，因为 gzip 是不可拆分的。这会导致一个特定 Spark 执行器承受沉重的负担，并且不会向其余执行器分配任何任务，如下图所示。

在 Spark Web UI Stage 选项卡上查看该阶段的实际任务数 (`NumPartitions`)，或者`df.rdd.getNumPartitions()`在代码中运行以检查并行度。

遇到 10 GB 的 gzip 文件时，请检查生成该文件的系统是否可以将其生成为可拆分的格式。如果这不是一个选项，则可能需要扩展集群容量来处理文件。要对加载的数据高效运行转换，您需要使用重新分区来重新平衡集群中的RDD所有工作节点。

手动确定目标分区数

根据数据的属性和Spark对某些功能的实现，尽管底层工作仍然可以并行化，但最终的`NumPartitions`价值可能会很低。如果太小，`NumPartitions`则运行`df.repartition(N)`以增加分区的数量，以便可以将处理分布在多个 Spark 执行器上。

在这种情况下，运行`df.repartition(100)`将`NumPartitions`从 1 增加到 100，从而创建 100 个数据分区，每个分区都有可以分配给其他执行者的任务。

该操作`repartition(N)`将所有数据平均分开（ 10 GB/100 个分区 = 100 MB/分区 ），从而避免数据偏向某些分区。

> **Note**
>
> 当运行诸如之类的洗牌操作`join`时，分区的数量会根据或的值动态增加`spark.sql.shuffle.partitions`或`spark.default.parallelism`减少。这便于在 Spark 执行者之间更有效地交换数据。有关更多信息，请参阅 Spark 文档。

在确定目标分区数量时，您的目标是最大限度地利用已配置 AWS Glue 的工作程序。 AWS Glue 工作人员的数量和 Spark 任务的数量通过数量相关联vCPUs。Spark 支持每个 v CPU 内核执行一项任务。在 3.0 或更高 AWS Glue 版本中，您可以使用以下公式计算目标分区数。

```
# 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
```

在此示例中，每个 G.1X 工作程序向 Spark 执行器 () 提供四CPU个 v 内核。`spark.executor.cores = 4`Spark 支持每个 v CPU Core 执行一个任务，因此 G.1X Spark 执行器可以同时运行四个任务（ ）。`numSlotPerExecutor`如果任务花费的时间相等，则此数量的分区可以充分利用群集。但是，有些任务会比其他任务花费更长的时间，从而导致内核处于空闲状态。如果发生这种情况，可以考虑`numPartitions`乘以 2 或 3 来分解并有效地安排瓶颈任务。

分区太多

分区数量过多会导致任务数量过多。由于与分布式处理（例如管理任务和 Spark 执行器之间的数据交换）相关的开销，这会导致 Spark 驱动程序负载过重。

如果作业中的分区数大于目标分区数，请考虑减少分区数量。您可以使用以下选项减少分区：

- 如果您的文件大小非常小，请使用 AWS Glue groupFiles。您可以减少因启动 Apache Spark 任务来处理每个文件而导致的过度并行度。
- 用于`coalesce(N)`将分区合并在一起。这是一个低成本的过程。减少分区数量时，优`coalesce(N)`于`repartition(N)`，因为`repartition(N)`执行洗牌可以平均分配每个分区中的记录量。这会增加成本和管理开销。
- 使用 Spark 3.x 自适应查询执行。正如 Apache Spark 中的关键主题部分所述，自适应查询执行提供了一种自动合并分区数量的功能。当你在执行之前无法知道分区数量时，你可以使用这种方法。

并行化从中加载数据 JDBC

Spark RDD 分区的数量由配置决定。请注意，默认情况下，通过`SELECT`查询仅运行一个任务来扫描整个源数据集。

两者 AWS Glue DynamicFrames 和 Spark 都 DataFrames 支持跨多个任务的并行JDBC数据加载。这是通过使用`where`谓词将一个查询拆分为多个`SELECT`查询来完成的。要并行化读取数据JDBC，请配置以下选项：

- 对于 AWS Glue DynamicFrame，设置`hashfield`（或`hashexpression`）和`hashpartition`。要了解更多信息，请参阅 paral JDBCIel 从表中读取。

```
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"
)
```

- 对于 Spark DataFramenumPartitions，请设置partitionColumnlowerBound、、和upperBound。要了解更多信息，请参阅JDBC到其他数据库。

```
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/")
```

使用连接器时，从 DynamoDB 并行加载数据 ETL

Spark RDD 分区的数量由dynamodb.splits参数决定。要并行处理来自 Amazon DynamoDB 的读取，请配置以下选项：

- 增加的价值dynamodb.splits。
- 按照 Spark 中的 “连接类型和选项” 中所述的ETL公式 AWS Glue 来优化参数。

并行处理来自 Kinesis Data Streams 的数据加载

Spark RDD 分区的数量由源 Amazon Kinesis Data Streams 数据流中的分片数量决定。如果您的数据流中只有几个分片，则只有几个 Spark 任务。这可能会导致下游流程的并行度降低。要并行处理来自 Kinesis Data Streams 的读取，请配置以下选项：

- 从 Kinesis Data Streams 加载数据时，增加分片数量以获得更多的并行度。
- 如果您在微批处理中的逻辑足够复杂，请考虑在删除不需要的列之后，在批次开始时对数据进行重新分区。

有关更多信息，请参阅优化 AWS Glue 流式ETL作业成本和性能的最佳实践。

数据加载后并行执行任务

要在数据加载后并行处理任务，请使用以下选项增加RDD分区数量：

- 重新分区数据以生成更多的分区，尤其是在初始加载之后如果无法并行处理负载本身。

  在`repartition()` DynamicFrame 或上调用 DataFrame，指定分区数。一个好的经验法则是可用内核数量的两到三倍。

  但是，在写入分区表时，这可能会导致文件爆炸式增长（每个分区都可能在每个表分区中生成一个文件）。为避免这种情况，您可以 DataFrame按列重新分区。它使用表分区列，因此在写入之前对数据进行整理。您可以指定更多数量的分区，而无需在表分区上放置小文件。但是，请注意避免数据倾斜，在这种倾斜中，某些分区值最终会包含大部分数据，从而延迟任务的完成。
- 出现洗牌时，增加该值。`spark.sql.shuffle.partitions`这也可以帮助解决洗牌时的任何内存问题。

  当你的洗牌分区超过 2,001 个时，Spark 会使用压缩的内存格式。如果您有一个接近该值的数字，则可能需要将该`spark.sql.shuffle.paritions`值设置为超过该限制以获得更有效的表示形式。

优化洗牌

某些操作（例如`join()`和`groupByKey()`）需要 Spark 执行随机播放。shuffle 是 Spark 用于重新分配数据的机制，以便在分区之间RDD对数据进行不同的分组。洗牌可以帮助修复性能瓶颈。但是，由于洗牌通常涉及在 Spark 执行器之间复制数据，因此洗牌是一项复杂且成本高昂的操作。例如，洗牌会产生以下成本：

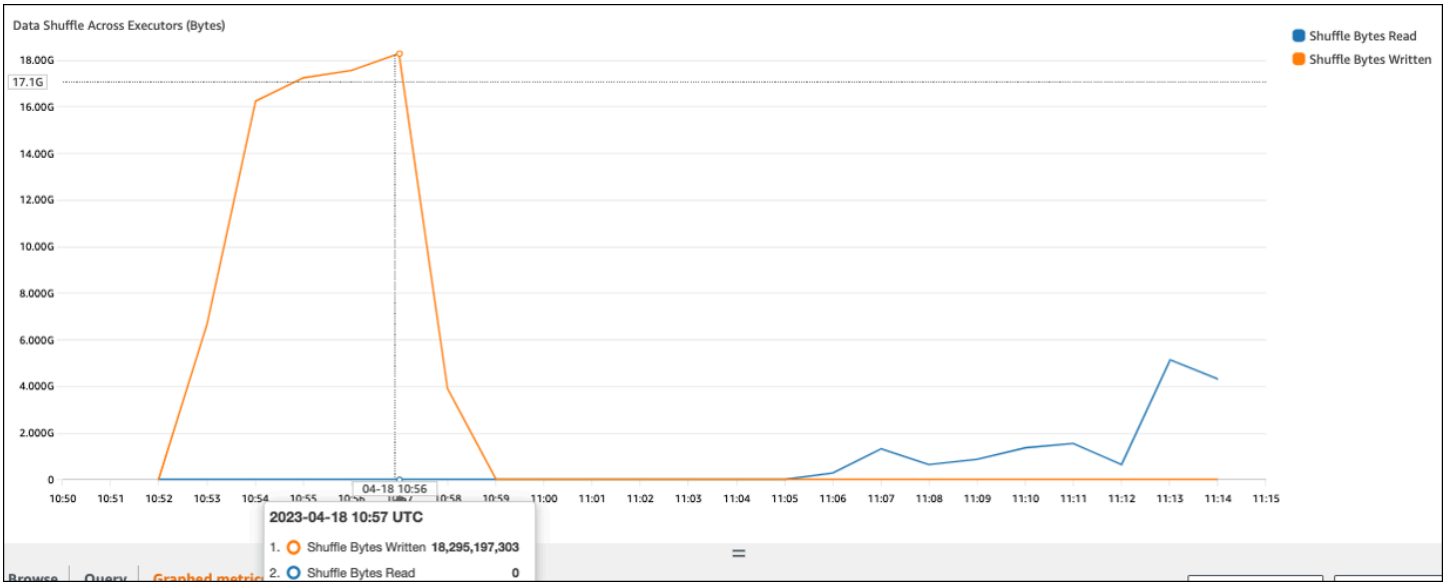
- 磁盘 I/O：
  - 在磁盘上生成大量中间文件。
- 网络 I/O：
  - 需要多个网络连接（连接数 =`Mapper × Reducer`）。
  - 由于记录会聚合到可能托管在不同的 Spark 执行器上的新RDD分区，因此数据集的很大一部分可能会通过网络在 Spark 执行器之间移动。
- CPU和内存负载：
  - 对值进行排序并合并数据集。这些操作是在执行者身上计划的，这给执行者带来了沉重的负担。

Shuffle 是导致 Spark 应用程序性能下降的最重要因素之一。在存储中间数据时，它可能会耗尽执行程序本地磁盘上的空间，从而导致 Spark 作业失败。

你可以通过 CloudWatch 指标和 Spark UI 来评估你的 shuffle 表现。

CloudWatch 指标

如果 Shuffle Bytes W rited 值与 Shuffle Bytes Re ad 相比高，则你的 Spark 作业可能会使用洗牌操作，例如或。`join()` `groupByKey()`

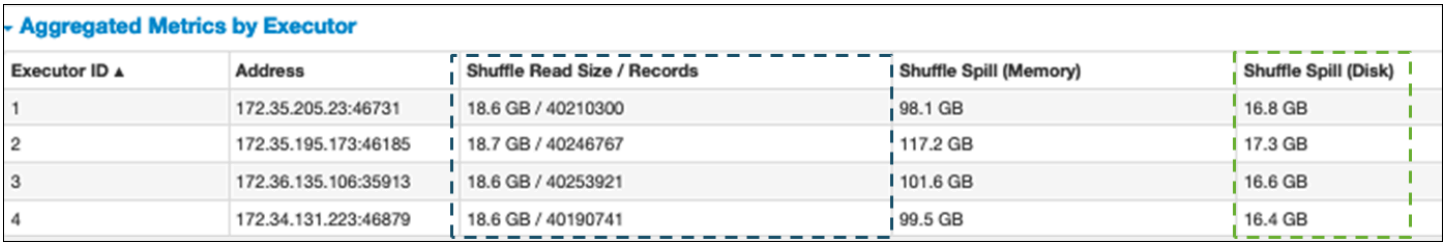


Spark UI

在 Spark 用户界面的舞台选项卡上，你可以查看 Shuffle 读取大小/记录值。您也可以在 “执行者” 选项卡上看到它。

在以下屏幕截图中，每个执行者与洗牌过程交换了大约 18.6GB/40200000 条记录，总随机读取大小约为 75 GB）。

Shuffle Spill（磁盘）列显示大量数据溢出到磁盘，这可能会导致磁盘已满或性能问题。



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

如果您观察到这些症状，并且该阶段与您的绩效目标相比花费的时间太长，或者失败并`Out Of MemoryNo space left on device`出现错误，请考虑以下解决方案。

优化联接

连接表的`join()`操作是最常用的洗牌操作，但它通常是性能瓶颈。由于加入是一项昂贵的操作，因此除非它对您的业务需求至关重要，否则我们建议不要使用它。通过询问以下问题，仔细检查您的数据管道是否得到有效利用：

- 您是否正在重新计算在其他可以重复使用的作业中执行的联接？
- 您是否加入是为了将外键解析为输出使用者未使用的值？

确认您的联接操作对您的业务需求至关重要，请查看以下选项，以满足您的要求来优化您的联接。

加入前使用下推

在执行联接 DataFrame 之前，过滤掉中不必要的行和列。这具有以下优点：

- 减少随机播放期间的数据传输量
- 减少 Spark 执行器中的处理量
- 减少数据扫描量

```
# 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"])
```

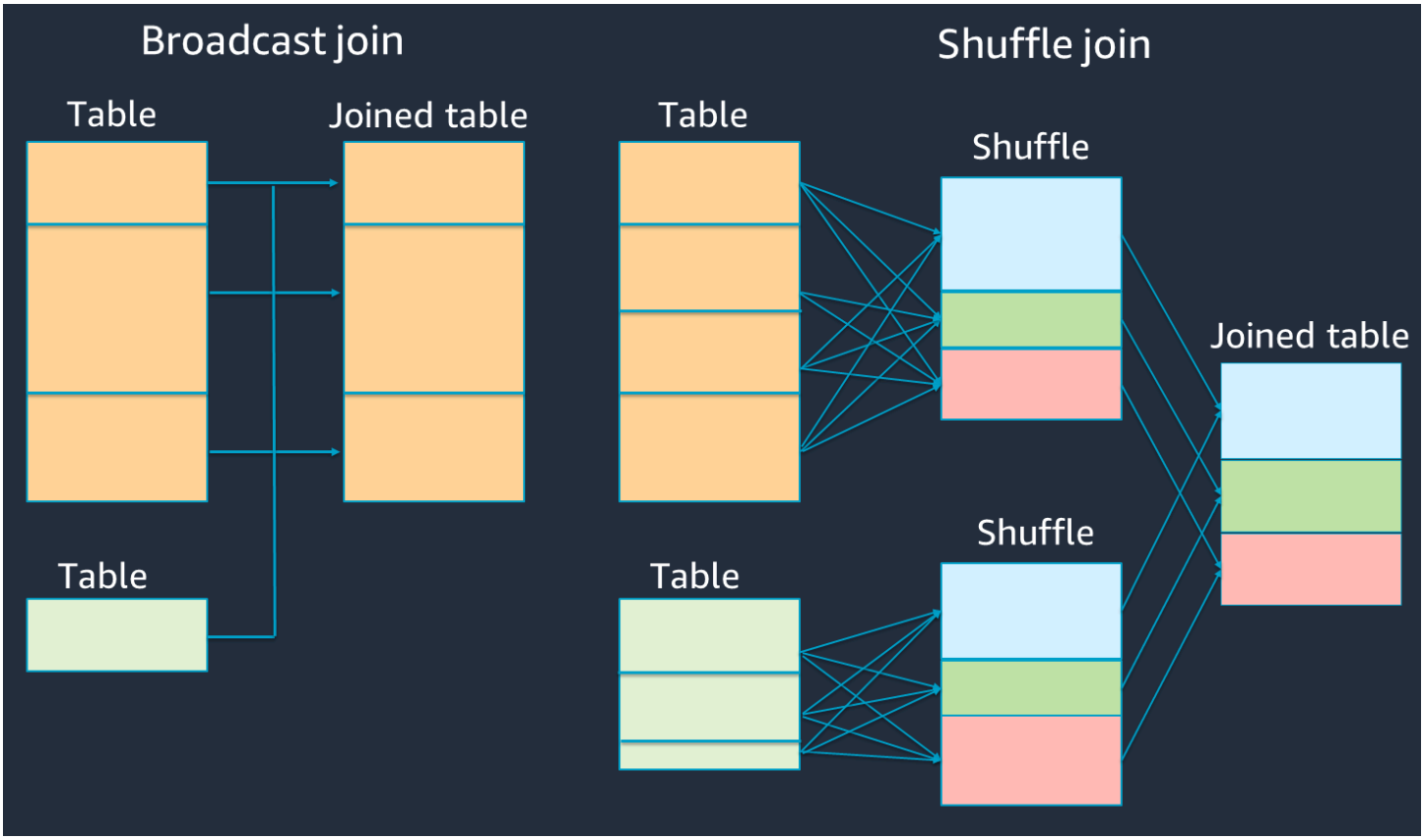
使用 “ DataFrame 加入”

尝试使用 Spark 高级别 API（例如 Spark SQL、 DataFrame、和数据集）来代替RDDAPI 或 DynamicFrame 联接。您可以使用诸如 DynamicFrame 之类 DataFrame 的方法调用转换为`dyf.toDF()`。正如 Apache Spark 的关键主题部分所述，这些联接操作在内部利用了 Catalyst 优化器的查询优化。

随机播放和广播哈希连接和提示

Spark 支持两种类型的加入：随机连接和广播哈希联接。广播哈希联接不需要洗牌，而且与随机连接相比，它需要的处理量更少。但是，它仅在将小表与大表连接时才适用。在加入可容纳单个 Spark 执行器内存的表时，可以考虑使用广播哈希联接。

下图显示了广播哈希联接和随机连接的高级结构和步骤。



每个联接的详细信息如下：

- 随机加入：
  - shuffle 哈希联接在不进行排序的情况下连接两个表，并在两个表之间分配联接。它适用于连接可以存储在 Spark 执行器内存中的小表。
  - 排序合并联接将按键分配要连接的两个表，并在联接之前对它们进行排序。它适用于大型表的联接。
- 广播哈希加入：
  - 广播哈希联接将较小的RDD或表推送到每个工作节点。然后它对较大的RDD或表的每个分区进行地图端合并。

    当您的一个或一个表可以容纳在内存中RDDs或可以放入内存中时，它适用于联接。尽可能进行广播哈希连接是有益的，因为它不需要洗牌。您可以使用加入提示从 Spark 请求广播加入，如下所示。

    ```
    # 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;
    ```

    有关联接提示的更多信息，请参阅联接提示。

在 AWS Glue 3.0 及更高版本中，您可以通过启用自适应查询执行和其他参数来自动利用广播哈希联接。当任一联接方的运行时统计数据小于自适应广播哈希联接阈值时，自适应查询执行会将排序合并联接转换为广播哈希联接。

在 AWS Glue 3.0 中，您可以通过设置启用自适应查询执行`spark.sql.adaptive.enabled=true`。AWSGlue 4.0 中默认启用自适应查询执行。

您可以设置与洗牌和广播哈希联接相关的其他参数：

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

有关相关参数的更多信息，请参阅将排序合并联接转换为广播联接。

在 AWS Glue 3.0 及更高版本中，你可以使用 shuffle 的其他联接提示来调整你的行为。

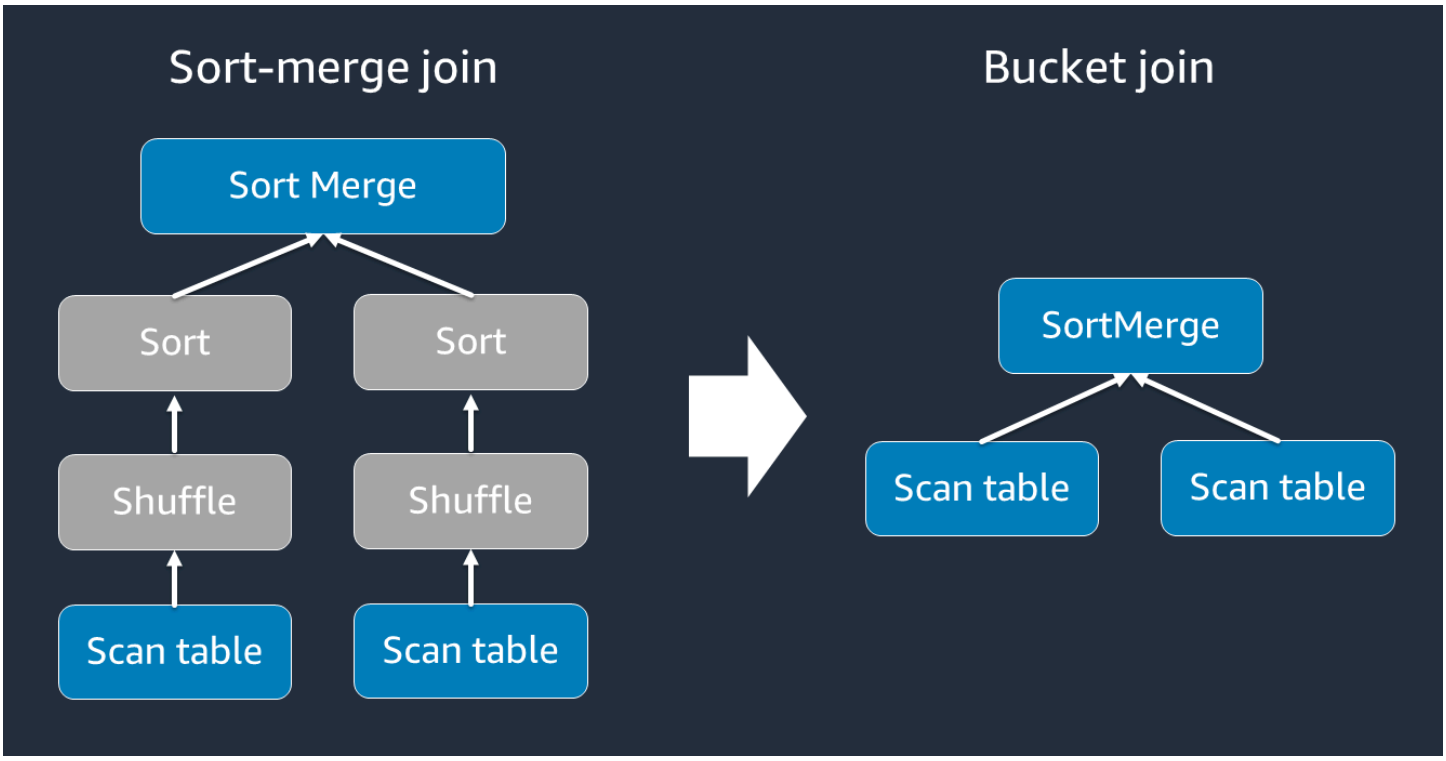
```
-- Join Hints for shuffle sort merge join
SELECT /*+ SHUFFLE_MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGEJOIN(t2) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- 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;
```

使用分桶

排序合并联接需要两个阶段：洗牌和排序，然后合并。当一些执行器合并而其他执行器同时排序时，这两个阶段可能会使 Spark 执行器过载，并导致OOM性能问题。在这种情况下，也许可以通过使用分桶来高效地加入。Bucketing 将对连接键上的输入进行预洗和预排序，然后将排序后的数据写入中间表。通过预先定义排序的中间表，可以降低联接大型表时洗牌和排序步骤的成本。



存储桶表可用于以下用途：

- 数据经常通过同一个密钥联接，例如 account_id
- 加载每日累积表，例如可以存储在公共列上的基表和增量表

您可以使用以下代码创建存储桶表。

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

DataFrames 在加入之前对联接键进行重新分区

要在联接之前在联接键 DataFrames 上对两者进行重新分区，请使用以下语句。

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

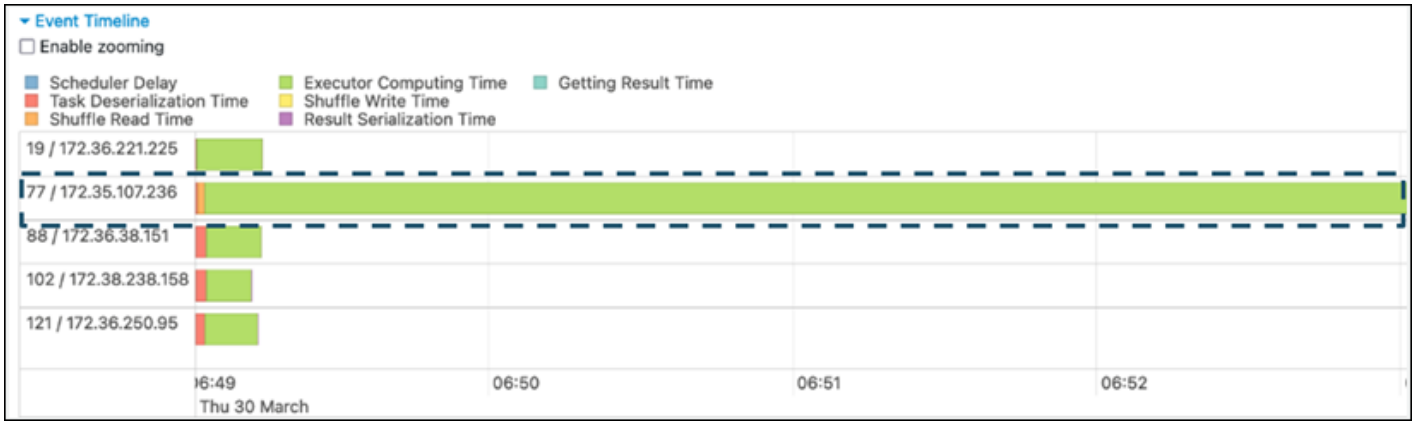
在启动联接之前，这将在连接键RDDs上分成两个（仍然是分开的）。如果两RDDs者使用相同的分区代码在同一个密钥上进行分区，则RDD记录在一起的计划很有可能在洗牌加入之前位于同一个工作线程上。这可能会减少加入期间的网络活动和数据偏差，从而提高性能。

克服数据偏差

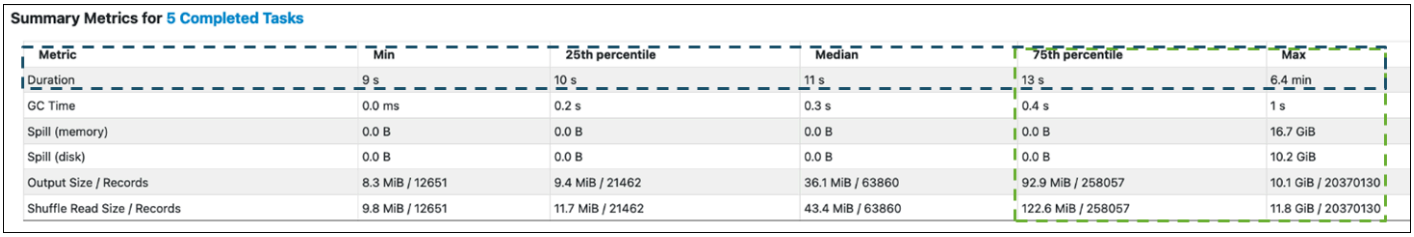
数据倾斜是 Spark 作业出现瓶颈的最常见原因之一。当数据在RDD分区之间分布不均匀时，就会发生这种情况。这会导致该分区的任务比其他分区的任务花费的时间长得多，从而延迟了应用程序的总体处理时间。

要识别数据偏差，请在 Spark 用户界面中评估以下指标：

- 在 Spark 用户界面的 “舞台” 选项卡上，查看 “事件时间表” 页面。您可以在以下屏幕截图中看到任务分布不均匀。分布不均匀或运行时间过长的任务可能表明数据存在偏差。



- 另一个重要的页面是摘要指标，它显示了 Spark 任务的统计信息。以下屏幕截图显示了持续时间、GC 时间、溢出（内存）、溢出（磁盘）等的百分位数指标。



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

当任务均匀分布时，您将在所有百分位数中看到相似的数字。当数据出现偏差时，您将在每个百分位数中看到非常有偏差的值。在示例中，任务持续时间在最小值、第 25 个百分位、中位数和第 75 个百分位数中小于 13 秒。虽然 Max 任务处理的数据量是第 75 个百分位数的 100 倍，但其 6.4 分钟的持续时间大约长了 30 倍。这意味着至少一项任务（或最多占任务的25%）花费的时间比其余任务长得多。

如果您看到数据偏斜，请尝试以下方法：

- 如果您使用 AWS Glue 3.0，请通过设置启用自适应查询执行`spark.sql.adaptive.enabled=true`。在 AWS Glue 4.0 中，自适应查询执行功能默认处于启用状态。

  您还可以通过设置以下相关参数，使用自适应查询执行来处理联接引入的数据偏差：

  - `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)`

  有关更多信息，请参阅 Apache Spark 文档。

- 使用值范围较大的键作为联接键。在 shuffle 联接中，分区是为密钥的每个哈希值确定的。如果联接键的基数太低，则哈希函数更有可能在分区之间分配数据方面做得不好。因此，如果您的应用程序和业务逻辑支持它，请考虑使用更高的基数键或复合键。

```
# 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"])
```

使用缓存

重复使用时 DataFrames，请使用或`df.persist()`将计算结果缓存在每个 Spark 执行程序的内存和磁盘上，从而避免额外的随机处理`df.cache()`或计算。Spark 还支持RDDs在磁盘上持久化或跨多个节点复制（存储级别）。

例如，您可以 DataFrames 通过添加来保留`df.persist()`。当不再需要缓存时，您可以使用`unpersist`丢弃缓存的数据。

```
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()
```

移除不需要的 Spark 动作

避免运行不必要的操作`count`，例如`show`、或`collect`。正如 Apache Spark 的关键主题部分所讨论的那样，Spark 很懒惰。每次对其执行操作时，RDD可能会重新计算每个变换。当你使用许多 Spark 操作时，会为每个操作调用多个源代码访问、任务计算和随机运行。

如果您不需要`collect()`在商业环境中执行其他操作，请考虑将其删除。

> **Note**
>
> 尽量避免`collect()`在商业环境中使用 Spark。该`collect()`操作将 Spark 执行器中的所有计算结果返回给 Spark 驱动程序，这可能会导致 Spark 驱动程序返回OOM错误。为避免OOM错误，Spark `spark.driver.maxResultSize = 1GB` 默认设置为将返回给 Spark 驱动程序的最大数据大小限制为 1 GB。

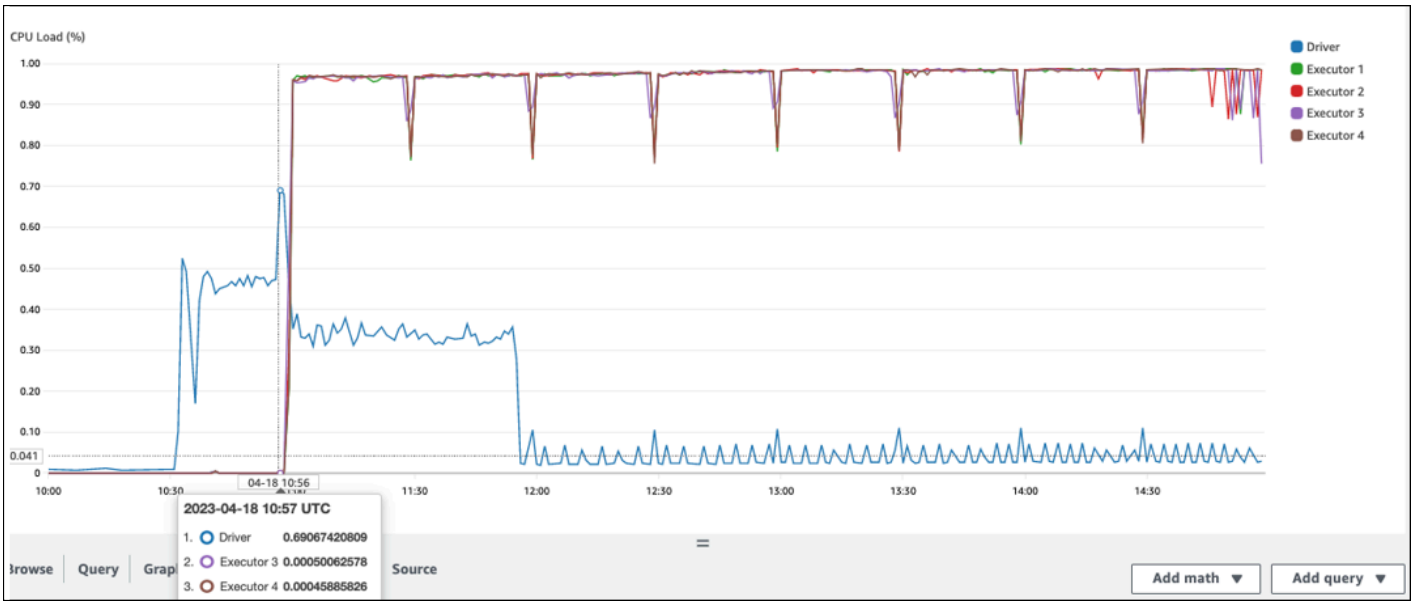
最大限度减少规划开销

正如 Apache Spark 中讨论的关键主题，Spark 驱动程序会生成执行计划。根据该计划，将任务分配给 Spark 执行器进行分布式处理。但是，如果存在大量小文件或 AWS Glue Data Catalog 包含大量分区，Spark 驱动程序可能会成为瓶颈。要确定高昂的计划开销，请评估以下指标。

CloudWatch 指标

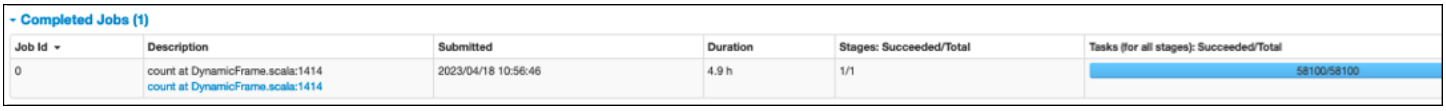
检查CPU负载和内存利用率以了解以下情况：

- Spark 驱动程序CPU负载和内存利用率记录为高。通常，Spark 驱动程序不会处理您的数据，因此CPU负载和内存利用率不会激增。但是，如果 Amazon S3 数据源有太多小文件，则列出所有 S3 对象并管理大量任务可能会导致资源利用率过高。
- 在 Spark 执行器中开始处理之前还有很长的间隔。在以下示例屏幕截图中，尽管 AWS Glue 任务从 10:00 开始，但 Spark 执行程序的CPU负载在 10:57 之前还是太低了。这表明 Spark 驱动程序可能需要很长时间才能生成执行计划。在此示例中，检索数据目录中的大量分区并列出 Spark 驱动程序中的大量小文件需要很长时间。



Spark UI

在 Spark 用户界面的 J ob 选项卡上，你可以看到提交时间。在以下示例中，Spark 驱动程序在 10:56:46 启动了 job0，尽管该作业是在 10:00:00 开始的。 AWS Glue



您还可以在 Job 选项卡上查看 “任务（适用于所有阶段）：成功/总时间”。在这种情况下，任务数记录为58100。如并行化任务页面的 Amazon S3 部分所述，任务数量与 S3 对象的数量大致对应。这意味着亚马逊 S3 中大约有 58,100 个对象。

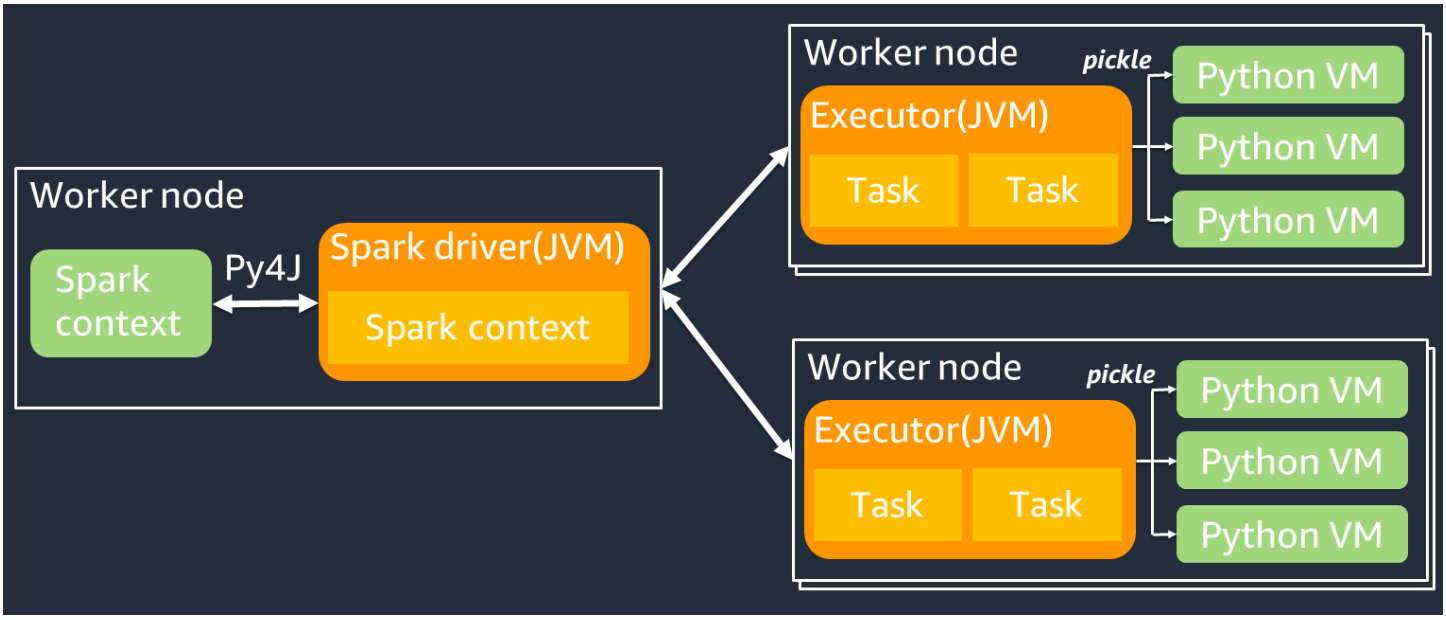
有关此任务和时间表的更多详细信息，请查看 “阶段” 选项卡。如果您发现 Spark 驱动程序存在瓶颈，请考虑以下解决方案：

- 当 Amazon S3 的文件过多时，请考虑并行化任务页面的 “分区过多” 部分中有关过度并行度的指导。
- 当 Amazon S3 的分区过多时，请考虑减少数据扫描量页面的 “过多 Amazon S3 分区” 部分中有关过度分区的指南。如果有许多AWS Glue 分区，请启用分区索引，以减少从数据目录检索分区元数据的延迟。有关更多信息，请参阅使用 AWS Glue 分区索引提高查询性能。
- 当分区JDBC过多时，请降低该hashpartition值。
- 当 DynamoDB 的分区过多时，请降低该值。dynamodb.splits
- 当流式处理作业的分区过多时，请减少分片的数量。

优化用户定义的函数

用户定义的函数 (UDFs) 和 RDD.map in PySpark 通常会显著降低性能。这是因为在 Spark 的底层 Scala 实现中准确表示你的 Python 代码需要开销。

下图显示了 PySpark 作业的架构。当你使用时 PySpark，Spark 驱动程序会使用 Py4J 库从 Python 中调用 Java 方法。在调用 Spark SQL 或 DataFrame 内置函数时，Python 和 Scala 之间几乎没有性能差异，因为这些函数JVM使用优化的执行计划在每个执行器上运行。



如果您使用自己的 Python 逻辑（例如使用）`map/ mapPartitions/ udf`，则该任务将在 Python 运行时环境中运行。管理两个环境会产生管理成本。此外，必须对内存中的数据进行转换，以供JVM运行时环境的内置函数使用。Pickle 是一种序列化格式，默认用于和 JVM Python 运行时之间的交换。但是，这种序列化和反序列化的成本非常高，因此用 Java 或 Scala UDFs 编写的速度比 Python 快。UDFs

为避免序列化和反序列化开销 PySpark，请考虑以下几点：

- 使用内置的 Spark SQL 函数 — 考虑用 Spark UDF 或 DataFrame内置函数替换您自己的函数SQL 或映射函数。在运行 Spark SQL 或 DataFrame 内置函数时，Python 和 Scala 之间几乎没有性能差异，因为任务是在每个执行器上处理的。JVM
- UDFs在 Scala 或 Java 中实现 — 考虑使用用 Java 或 Scala 编写的，因为它们在上运行。UDF JVM
- 使用基于 Apache Arrow 的矢量化工作负载 — 考虑使用基UDFs于 Arrow 的内容。UDFs此功能也被称为矢量化UDF（PandasUDF）。Apache Arrow 是一种与语言无关的内存数据格式， AWS Glue 可用于在和 Python 进程之间高效地传输数据。JVM目前，这对使用 Pandas 或 NumPy数据的 Python 用户最有利。

  箭头是一种柱状（矢量化）格式。它的使用不是自动的，可能需要对配置或代码进行一些细微的更改才能充分利用并确保兼容性。有关更多详细信息和限制，请参阅中的 Apache Arrow。 PySpark

  以下示例比较了标准 Python UDF 中的基本增量、矢量化UDF版本和 Spark 中的基本增量。SQL

标准 Python UDF

示例时间为 3.20（秒）。

示例代码

```
# 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()
```

执行计划

```
== 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)
```

矢量化 UDF

示例时间为 0.59（秒）。

向量化UDF比上一个UDF示例快 5 倍。你可以看到 Physical PlanArrowEvalPython，这表明这个应用程序是由 Apache Arrow 向量化的。要启用 Vectori UDF zed，必须在代码spark.sql.execution.arrow.pyspark.enabled = true中指定。

示例代码

```
# 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()
```

执行计划

```
== 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)
```

火花 SQL

示例时间为 0.087（秒）。

Spark 比 Vector SQL ized 快得多UDF，因为任务是在没有 Python 运行时的情况下在每个执行器上运行的JVM。如果你能用内置函数替换你UDF的，我们建议你这样做。

示例代码

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

使用熊猫来处理大数据

如果你已经熟悉熊猫并想使用 Spark 来处理大数据，你可以在 Spark API 上使用熊猫。 AWS Glue 4.0 及更高版本支持它。首先，你可以使用官方笔记本快速入门：Spark API 上的 Pandas。有关更多信息，请参阅PySpark 文档。

资源

- AWS Glue
- 性能优化（Spark SQL 指南）
- AWS Glue 优化研讨会

文档历史记录

下表介绍了本指南的一些重要更改。如果您希望收到有关未来更新的通知，可以订阅 RSS 源。

变更	说明	日期
初次发布	—	2024 年 1 月 2 日

AWS 规范性指导词汇表

以下是 AWS 规范性指导提供的策略、指南和模式中的常用术语。若要推荐词条，请使用术语表末尾的提供反馈链接。

数字

7 R

将应用程序迁移到云中的 7 种常见迁移策略。这些策略以 Gartner 于 2011 年确定的 5 R 为基础，包括以下内容：

- 重构/重新架构 - 充分利用云原生功能来提高敏捷性、性能和可扩展性，以迁移应用程序并修改其架构。这通常涉及到移植操作系统和数据库。示例：将您的本地 Oracle 数据库迁移到兼容 Amazon Aurora PostgreSQL 的版本。
- 更换平台 - 将应用程序迁移到云中，并进行一定程度的优化，以利用云功能。示例：在中将您的本地 Oracle 数据库迁移到适用于 Oracle 的亚马逊关系数据库服务 (Amazon RDS) AWS Cloud。
- 重新购买 - 转换到其他产品，通常是从传统许可转向 SaaS 模式。示例：将您的客户关系管理 (CRM) 系统迁移到 Salesforce.com。
- 更换主机（直接迁移）- 将应用程序迁移到云中，无需进行任何更改即可利用云功能。示例：在中的 EC2 实例上将您的本地 Oracle 数据库迁移到 Oracle AWS Cloud。
- 重新定位（虚拟机监控器级直接迁移）：将基础设施迁移到云中，无需购买新硬件、重写应用程序或修改现有操作。您可以将服务器从本地平台迁移到同一平台的云服务。示例：将Microsoft Hyper-V应用程序迁移到 AWS。
- 保留（重访）- 将应用程序保留在源环境中。其中可能包括需要进行重大重构的应用程序，并且您希望将工作推迟到以后，以及您希望保留的遗留应用程序，因为迁移它们没有商业上的理由。
- 停用 - 停用或删除源环境中不再需要的应用程序。

A

ABAC

请参阅基于属性的访问控制。

抽象服务

参见托管服务。

酸

参见原子性、一致性、隔离性、耐久性。

主动-主动迁移

一种数据库迁移方法，在这种方法中，源数据库和目标数据库保持同步（通过使用双向复制工具或双写操作），两个数据库都在迁移期间处理来自连接应用程序的事务。这种方法支持小批量、可控的迁移，而不需要一次性割接。与主动-被动迁移相比，它更灵活，但需要更多的工作。

主动-被动迁移

一种数据库迁移方法，在这种方法中，源数据库和目标数据库保持同步，但在将数据复制到目标数据库时，只有源数据库处理来自连接应用程序的事务。目标数据库在迁移期间不接受任何事务。

聚合函数

一个 SQL 函数，它对一组行进行操作并计算该组的单个返回值。聚合函数的示例包括SUM和MAX。

AI

参见人工智能。

AIOps

参见人工智能操作。

匿名化

永久删除数据集中个人信息的过程。匿名化可以帮助保护个人隐私。匿名化数据不再被视为个人数据。

反模式

一种用于解决反复出现的问题的常用解决方案，而在这类问题中，此解决方案适得其反、无效或不如替代方案有效。

应用程序控制

一种安全方法，仅允许使用经批准的应用程序，以帮助保护系统免受恶意软件的侵害。

应用程序组合

有关组织使用的每个应用程序的详细信息的集合，包括构建和维护该应用程序的成本及其业务价值。这些信息是产品组合发现和分析过程的关键，有助于识别需要进行迁移、现代化和优化的应用程序并确定其优先级。

人工智能（AI）

计算机科学领域致力于使用计算技术执行通常与人类相关的认知功能，例如学习、解决问题和识别模式。有关更多信息，请参阅什么是人工智能？

人工智能运营（AIOps）

使用机器学习技术解决运营问题、减少运营事故和人为干预以及提高服务质量的过程。有关如何在 AWS 迁移策略中使用 AIOps 的更多信息，请参阅运营集成指南。

非对称加密

一种加密算法，使用一对密钥，一个公钥用于加密，一个私钥用于解密。您可以共享公钥，因为它不用于解密，但对私钥的访问应受到严格限制。

原子性、一致性、隔离性、持久性（ACID）

一组软件属性，即使在出现错误、电源故障或其他问题的情况下，也能保证数据库的数据有效性和操作可靠性。

基于属性的访问权限控制（ABAC）

根据用户属性（如部门、工作角色和团队名称）创建精细访问权限的做法。有关更多信息，请参阅 AWS Identity and Access Management (I AM) 文档 AWS中的 AB AC。

权威数据源

存储主要数据版本的位置，被认为是最可靠的信息源。您可以将数据从权威数据源复制到其他位置，以便处理或修改数据，例如对数据进行匿名化、编辑或假名化。

可用区

中的一个不同位置 AWS 区域 ，不受其他可用区域故障的影响，并向同一区域中的其他可用区提供低成本、低延迟的网络连接。

AWS 云采用框架 (AWS CAF)

该框架包含指导方针和最佳实践 AWS ，可帮助组织制定高效且有效的计划，以成功迁移到云端。AWS CAF将指导分为六个重点领域，称为视角：业务、人员、治理、平台、安全和运营。业务、人员和治理角度侧重于业务技能和流程；平台、安全和运营角度侧重于技术技能和流程。例如，人员角度针对的是负责人力资源（HR）、人员配置职能和人员管理的利益相关者。从这个角度来看，AWS CAF 为人员发展、培训和沟通提供了指导，以帮助组织为成功采用云做好准备。有关更多信息，请参阅 AWS CAF 网站和 AWS CAF 白皮书。

AWS 工作负载资格框架 (AWS WQF)

一种评估数据库迁移工作负载、推荐迁移策略和提供工作估算的工具。 AWS WQF 包含在 AWS Schema Conversion Tool (AWS SCT) 中。它用来分析数据库架构和代码对象、应用程序代码、依赖关系和性能特征，并提供评测报告。

B

坏机器人

旨在破坏个人或组织或对其造成伤害的机器人。

BCP

参见业务连续性计划。

行为图

一段时间内资源行为和交互的统一交互式视图。您可以使用 Amazon Detective 的行为图来检查失败的登录尝试、可疑的 API 调用和类似的操作。有关更多信息，请参阅 Detective 文档中的行为图中的数据。

大端序系统

一个先存储最高有效字节的系统。另请参见字节顺序。

二进制分类

一种预测二进制结果（两个可能的类别之一）的过程。例如，您的 ML 模型可能需要预测诸如“该电子邮件是否为垃圾邮件？”或“这个产品是书还是汽车？”之类的问题

bloom 筛选条件

一种概率性、内存高效的数据结构，用于测试元素是否为集合的成员。

蓝/绿部署

一种部署策略，您可以创建两个独立但完全相同的环境。在一个环境中运行当前的应用程序版本（蓝色），在另一个环境中运行新的应用程序版本（绿色）。此策略可帮助您在影响最小的情况下快速回滚。

自动程序

一种通过互联网运行自动任务并模拟人类活动或互动的软件应用程序。有些机器人是有用或有益的，例如在互联网上索引信息的网络爬虫。其他一些被称为恶意机器人的机器人旨在破坏个人或组织或对其造成伤害。

僵尸网络

被恶意软件感染并受单方（称为机器人牧民或机器人操作员）控制的机器人网络。僵尸网络是最著名的扩展机器人及其影响力的机制。

分支

代码存储库的一个包含区域。在存储库中创建的第一个分支是主分支。您可以从现有分支创建新分支，然后在新分支中开发功能或修复错误。为构建功能而创建的分支通常称为功能分支。当功能可以发布时，将功能分支合并回主分支。有关更多信息，请参阅关于分支（GitHub 文档）。

破碎的玻璃通道

在特殊情况下，通过批准的流程，用户 AWS 账户 可以快速访问他们通常没有访问权限的内容。有关更多信息，请参阅 Well -Architected 指南中的 “实施破碎玻璃程序” 指示 AWS 器。

棕地策略

您环境中的现有基础设施。在为系统架构采用棕地策略时，您需要围绕当前系统和基础设施的限制来设计架构。如果您正在扩展现有基础设施，则可以将棕地策略和全新策略混合。

缓冲区缓存

存储最常访问的数据的内存区域。

业务能力

企业如何创造价值（例如，销售、客户服务或营销）。微服务架构和开发决策可以由业务能力驱动。有关更多信息，请参阅在 AWS上运行容器化微服务白皮书中的围绕业务能力进行组织部分。

业务连续性计划（BCP）

一项计划，旨在应对大规模迁移等破坏性事件对运营的潜在影响，并使企业能够快速恢复运营。

C

CAF

参见AWS 云采用框架。

金丝雀部署

向最终用户缓慢而渐进地发布版本。当你有信心时，你可以部署新版本并全部替换当前版本。

CCoE

参见云卓越中心。

CDC

参见变更数据捕获。

更改数据捕获（CDC）

跟踪数据来源（如数据库表）的更改并记录有关更改的元数据的过程。您可以将 CDC 用于各种目的，例如审计或复制目标系统中的更改以保持同步。

混沌工程

故意引入故障或破坏性事件来测试系统的弹性。您可以使用 AWS Fault Injection Service (AWS FIS) 来执行实验，对您的 AWS 工作负载施加压力并评估其响应。

CI/CD

查看持续集成和持续交付。

分类

一种有助于生成预测的分类流程。分类问题的 ML 模型预测离散值。离散值始终彼此不同。例如，一个模型可能需要评估图像中是否有汽车。

客户端加密

在目标 AWS 服务 收到数据之前，对数据进行本地加密。

云卓越中心（CCoE）

一个多学科团队，负责推动整个组织的云采用工作，包括开发云最佳实践、调动资源、制定迁移时间表、领导组织完成大规模转型。有关更多信息，请参阅 AWS Cloud 企业战略博客上的 CCoE 帖子。

云计算

通常用于远程数据存储和 IoT 设备管理的云技术。云计算通常与边缘计算技术相关。

云运营模型

在 IT 组织中，一种用于构建、完善和优化一个或多个云环境的运营模型。有关更多信息，请参阅构建您的云运营模型。

云采用阶段

组织迁移到以下阶段时通常会经历四个阶段 AWS Cloud：

- 项目 - 出于概念验证和学习目的，开展一些与云相关的项目
- 基础 - 进行基础投资以扩大云采用率（例如，创建登录区、定义 CCoE、建立运营模型）

- 迁移 - 迁移单个应用程序
- 重塑 - 优化产品和服务，在云中创新

Stephen Orban在 AWS Cloud 企业战略博客的博客文章《云优先之旅和采用阶段》中定义了这些阶段。有关它们与 AWS 迁移策略的关系的信息，请参阅迁移准备指南。

CMDB

参见配置管理数据库。

代码存储库

通过版本控制过程存储和更新源代码和其他资产（如文档、示例和脚本）的位置。常见的云存储库包括 GitHub 或 AWS CodeCommit。每个版本的代码都称为一个分支。在微服务结构中，每个存储库都专门用于一个功能。单个 CI/CD 管道可以使用多个存储库。

冷缓存

一种空的、填充不足或包含过时或不相关数据的缓冲区缓存。这会影响性能，因为数据库实例必须从主内存或磁盘读取，这比从缓冲区缓存读取要慢。

冷数据

很少访问的数据，且通常是历史数据。查询此类数据时，通常可以接受慢速查询。将这些数据转移到性能较低且成本更低的存储层或类别可以降低成本。

计算机视觉 (CV)

人工智能领域，使用机器学习来分析和提取数字图像和视频等视觉格式中的信息。例如， AWS Panorama 提供将 CV 添加到本地摄像机网络的设备，而 Amazon 则为 CV SageMaker 提供图像处理算法。

配置偏差

对于工作负载，配置会从预期状态发生变化。这可能会导致工作负载变得不合规，而且通常是渐进的，不是故意的。

配置管理数据库（CMDB）

一种存储库，用于存储和管理有关数据库及其 IT 环境的信息，包括硬件和软件组件及其配置。您通常在迁移的产品组合发现和分析阶段使用来自 CMDB 的数据。

合规性包

一系列 AWS Config 规则和补救措施，您可以汇编这些规则和补救措施，以自定义合规性和安全性检查。您可以使用 YAML 模板将一致性包作为单个实体部署在 AWS 账户 和区域或整个组织中。有关更多信息，请参阅 AWS Config 文档中的一致性包。

持续集成和持续交付（CI/CD）

自动执行软件发布过程的源代码、构建、测试、暂存和生产阶段的过程。CI/CD 通常被描述为管道。CI/CD 可以帮助您实现流程自动化、提高工作效率、改善代码质量并加快交付速度。有关更多信息，请参阅持续交付的优势。CD 也可以表示持续部署。有关更多信息，请参阅持续交付与持续部署。

CV

参见计算机视觉。

D

静态数据

网络中静止的数据，例如存储中的数据。

数据分类

根据网络中数据的关键性和敏感性对其进行识别和分类的过程。它是任何网络安全风险管理策略的关键组成部分，因为它可以帮助您确定对数据的适当保护和保留控制。数据分类是 Well-Architecte AWS d Framework 中安全支柱的一个组成部分。有关详细信息，请参阅数据分类。

数据漂移

生产数据与用来训练机器学习模型的数据之间的有意义差异，或者输入数据随时间推移的有意义变化。数据漂移可能降低机器学习模型预测的整体质量、准确性和公平性。

传输中数据

在网络中主动移动的数据，例如在网络资源之间移动的数据。

数据网格

一种架构框架，可提供分布式、去中心化的数据所有权以及集中式管理和治理。

数据最少化

仅收集并处理绝对必要数据的原则。在中进行数据最小化 AWS Cloud 可以降低隐私风险、成本和分析碳足迹。

数据边界

AWS 环境中的一组预防性防护措施，可帮助确保只有可信身份才能访问来自预期网络的可信资源。有关更多信息，请参阅在上构建数据边界。 AWS

数据预处理

将原始数据转换为 ML 模型易于解析的格式。预处理数据可能意味着删除某些列或行，并处理缺失、不一致或重复的值。

数据溯源

在数据的整个生命周期跟踪其来源和历史的过程，例如数据如何生成、传输和存储。

数据主体

正在收集和处理其数据的个人。

数据仓库

一种支持商业智能（例如分析）的数据管理系统。数据仓库通常包含大量历史数据，通常用于查询和分析。

数据库定义语言（DDL）

在数据库中创建或修改表和对象结构的语句或命令。

数据库操作语言（DML）

在数据库中修改（插入、更新和删除）信息的语句或命令。

DDL

参见数据库定义语言。

深度融合

组合多个深度学习模型进行预测。您可以使用深度融合来获得更准确的预测或估算预测中的不确定性。

深度学习

一个 ML 子字段使用多层人工神经网络来识别输入数据和感兴趣的目标变量之间的映射。

defense-in-depth

一种信息安全方法，经过深思熟虑，在整个计算机网络中分层实施一系列安全机制和控制措施，以保护网络及其中数据的机密性、完整性和可用性。当你采用这种策略时 AWS，你会在 AWS Organizations 结构的不同层面添加多个控件来帮助保护资源。例如，一种 defense-in-depth 方法可以结合多因素身份验证、网络分段和加密。

委托管理员

在中 AWS Organizations，兼容的服务可以注册 AWS 成员帐户来管理组织的帐户并管理该服务的权限。此账户被称为该服务的委托管理员。有关更多信息和兼容服务列表，请参阅 AWS Organizations 文档中使用 AWS Organizations的服务。

部署

使应用程序、新功能或代码修复在目标环境中可用的过程。部署涉及在代码库中实现更改，然后在应用程序的环境中构建和运行该代码库。

开发环境

参见环境。

侦测性控制

一种安全控制，在事件发生后进行检测、记录日志和发出警报。这些控制是第二道防线，提醒您注意绕过现有预防性控制的安全事件。有关更多信息，请参阅在 AWS上实施安全控制中的侦测性控制。

开发价值流映射 (DVSM)

用于识别对软件开发生命周期中的速度和质量产生不利影响的限制因素并确定其优先级的流程。DVSM 扩展了最初为精益生产实践设计的价值流映射流程。其重点关注在软件开发过程中创造和转移价值所需的步骤和团队。

数字孪生

真实世界系统的虚拟再现，如建筑物、工厂、工业设备或生产线。数字孪生支持预测性维护、远程监控和生产优化。

维度表

在星型架构中，一种较小的表，其中包含事实表中定量数据的数据属性。维度表属性通常是文本字段或行为类似于文本的离散数字。这些属性通常用于查询约束、筛选和结果集标注。

灾难

阻止工作负载或系统在其主要部署位置实现其业务目标的事件。这些事件可能是自然灾害、技术故障或人为操作的结果，例如无意的配置错误或恶意软件攻击。

灾难恢复 (DR)

您用来最大限度地减少灾难造成的停机时间和数据丢失的策略和流程。有关更多信息，请参阅 Well-Architected Fr ame AWS work 中的 “工作负载灾难恢复：云端 AWS 恢复”。

DML

参见数据库操作语言。

领域驱动设计

一种开发复杂软件系统的方法，通过将其组件连接到每个组件所服务的不断发展的领域或核心业务目标。Eric Evans 在其著作领域驱动设计：软件核心复杂性应对之道（Boston: Addison-Wesley Professional, 2003）中介绍了这一概念。有关如何将领域驱动设计与 strangler fig 模式结合使用的信息，请参阅使用容器和 Amazon API Gateway 逐步将原有的 Microsoft ASP.NET（ASMX）Web 服务现代化。

DR

参见灾难恢复。

漂移检测

跟踪与基准配置的偏差。例如，您可以使用 AWS CloudFormation 来检测系统资源中的偏差，也可以使用 AWS Control Tower 来检测着陆区中可能影响监管要求合规性的变化。

DVSM

参见开发价值流映射。

E

EDA

参见探索性数据分析。

边缘计算

该技术可提高位于 IoT 网络边缘的智能设备的计算能力。与云计算相比，边缘计算可以减少通信延迟并缩短响应时间。

加密

一种将人类可读的纯文本数据转换为密文的计算过程。

加密密钥

由加密算法生成的随机位的加密字符串。密钥的长度可能有所不同，而且每个密钥都设计为不可预测且唯一。

字节顺序

字节在计算机内存中的存储顺序。大端序系统先存储最高有效字节。小端序系统先存储最低有效字节。

端点

参见服务端点。

端点服务

一种可以在虚拟私有云（VPC）中托管，与其他用户共享的服务。您可以使用其他 AWS 账户 或 AWS Identity and Access Management (IAM) 委托人创建终端节点服务， AWS PrivateLink 并向其授予权限。这些账户或主体可通过创建接口 VPC 端点来私密地连接到您的端点服务。有关更多信息，请参阅 Amazon Virtual Private Cloud（Amazon VPC）文档中的创建端点服务。

企业资源规划 (ERP)

一种自动化和管理企业关键业务流程（例如会计、MES 和项目管理）的系统。

信封加密

用另一个加密密钥对加密密钥进行加密的过程。有关更多信息，请参阅 AWS Key Management Service (AWS KMS) 文档中的信封加密。

environment

正在运行的应用程序的实例。以下是云计算中常见的环境类型：

- 开发环境 — 正在运行的应用程序的实例，只有负责维护应用程序的核心团队才能使用。开发环境用于测试更改，然后再将其提升到上层环境。这类环境有时称为测试环境。
- 下层环境 — 应用程序的所有开发环境，比如用于初始构建和测试的环境。
- 生产环境 — 最终用户可以访问的正在运行的应用程序的实例。在 CI/CD 管道中，生产环境是最后一个部署环境。
- 上层环境 — 除核心开发团队以外的用户可以访问的所有环境。这可能包括生产环境、预生产环境和用户验收测试环境。

epic

在敏捷方法学中，有助于组织工作和确定优先级的功能类别。epics 提供了对需求和实施任务的总体描述。例如， AWS CAF 安全史诗包括身份和访问管理、侦探控制、基础设施安全、数据保护和事件响应。有关 AWS 迁移策略中 epics 的更多信息，请参阅计划实施指南。

ERP

参见企业资源规划。

探索性数据分析（EDA）

分析数据集以了解其主要特征的过程。您收集或汇总数据，并进行初步调查，以发现模式、检测异常并检查假定情况。EDA 通过计算汇总统计数据和创建数据可视化得以执行。

F

事实表

星形架构中的中心表。它存储有关业务运营的定量数据。通常，事实表包含两种类型的列：包含度量的列和包含维度表外键的列。

失败得很快

一种使用频繁和增量测试来缩短开发生命周期的理念。这是敏捷方法的关键部分。

故障隔离边界

在中 AWS Cloud，诸如可用区 AWS 区域、控制平面或数据平面之类的边界，它限制了故障的影响并有助于提高工作负载的弹性。有关更多信息，请参阅AWS 故障隔离边界。

功能分支

参见分支。

特征

您用来进行预测的输入数据。例如，在制造环境中，特征可能是定期从生产线捕获的图像。

特征重要性

特征对于模型预测的重要性。这通常表示为数值分数，可以通过各种技术进行计算，例如 Shapley 加法解释（SHAP）和积分梯度。有关更多信息，请参阅机器学习模型的可解释性：AWS。

功能转换

为 ML 流程优化数据，包括使用其他来源丰富数据、扩展值或从单个数据字段中提取多组信息。这使得 ML 模型能从数据中获益。例如，如果您将“2021-05-27 00:15:37”日期分解为“2021”、“五月”、“星期四”和“15”，则可以帮助学习与不同数据成分相关的算法学习精细模式。

FGAC

请参阅精细的访问控制。

精细访问控制 (FGAC)

使用多个条件允许或拒绝访问请求。

快闪迁移

一种数据库迁移方法，它使用连续的数据复制，通过更改数据捕获在尽可能短的时间内迁移数据，而不是使用分阶段的方法。目标是将停机时间降至最低。

G

地理封锁

请参阅地理限制。

地理限制（地理阻止）

在 Amazon 中 CloudFront，一种阻止特定国家/地区的用户访问内容分发的选项。您可以使用允许列表或阻止列表来指定已批准和已禁止的国家/地区。有关更多信息，请参阅 CloudFront 文档中的限制内容的地理分布。

GitFlow 工作流程

一种方法，在这种方法中，下层和上层环境在源代码存储库中使用不同的分支。Gitflow 工作流程被认为是传统的，而基于主干的工作流程是现代的首选方法。

全新策略

在新环境中缺少现有基础设施。在对系统架构采用全新策略时，您可以选择所有新技术，而不受对现有基础设施（也称为棕地）兼容性的限制。如果您正在扩展现有基础设施，则可以将棕地策略和全新策略混合。

防护机制

一种高级规则，用于跨组织单位（OU）管理资源、策略和合规性。预防性防护机制会执行策略以确保符合合规性标准。它们是使用服务控制策略和 IAM 权限边界实现的。侦测性防护机制会检测策略违规和合规性问题，并生成警报以进行修复。它们通过使用 AWS Config、、Amazon、 AWS Security Hub GuardDuty AWS Trusted Advisor、Amazon Inspector 和自定义 AWS Lambda 支票来实现。

H

HA

参见高可用性。

异构数据库迁移

将源数据库迁移到使用不同数据库引擎的目标数据库（例如，从 Oracle 迁移到 Amazon Aurora）。异构迁移通常是重新架构工作的一部分，而转换架构可能是一项复杂的任务。AWS 提供了 AWS SCT 来帮助实现架构转换。

高可用性 (HA)

在遇到挑战或灾难时，工作负载无需干预即可连续运行的能力。HA 系统旨在自动进行故障转移、持续提供良好性能，并以最小的性能影响处理不同负载和故障。

历史数据库现代化

一种用于实现运营技术 (OT) 系统现代化和升级以更好满足制造业需求的方法。历史数据库是一种用于收集和存储工厂中各种来源数据的数据库。

同构数据库迁移

将源数据库迁移到共享同一数据库引擎的目标数据库（例如，从 Microsoft SQL Server 迁移到 Amazon RDS for SQL Server）。同构迁移通常是更换主机或更换平台工作的一部分。您可以使用本机数据库实用程序来迁移架构。

热数据

经常访问的数据，例如实时数据或近期的转化数据。这些数据通常需要高性能存储层或存储类别才能提供快速的查询响应。

修补程序

针对生产环境中关键问题的紧急修复。由于其紧迫性，修补程序通常是在典型的 DevOps 发布工作流程之外进行的。

hypercare 周期

割接之后，迁移团队立即管理和监控云中迁移的应用程序以解决任何问题的时间段。通常，这个周期持续 1-4 天。在 hypercare 周期结束时，迁移团队通常会将应用程序的责任移交给云运营团队。

I

IaC

参见基础架构即代码。

基于身份的策略

附加到一个或多个 IAM 委托人的策略，用于定义他们在 AWS Cloud 环境中的权限。

空闲应用程序

90 天内平均 CPU 和内存使用率在 5% 到 20% 之间的应用程序。在迁移项目中，通常会停用这些应用程序或将其保留在本地。

IIoT

参见工业物联网。

不可变的基础架构

一种为生产工作负载部署新基础架构，而不是更新、修补或修改现有基础架构的模型。不可变基础架构本质上比可变基础架构更一致、更可靠、更可预测。有关更多信息，请参阅 Well-Architected Framework 中的使用不可变基础架构 AWS 部署最佳实践。

入站（入口）VPC

在 AWS 多账户架构中，一种接受、检查和路由来自应用程序外部的网络连接的 VPC。AWS 安全参考架构建议使用入站、出站和检查 VPC 设置网络账户，保护应用程序与广泛的互联网之间的双向接口。

增量迁移

一种割接策略，在这种策略中，您可以将应用程序分成小部分进行迁移，而不是一次性完整割接。例如，您最初可能只将几个微服务或用户迁移到新系统。在确认一切正常后，您可以逐步迁移其他微服务或用户，直到停用遗留系统。这种策略降低了大规模迁移带来的风险。

工业 4.0

该术语由克劳斯·施瓦布（Klaus Schwab）于2016年推出，指的是通过连接、实时数据、自动化、分析和人工智能/机器学习的进步实现制造流程的现代化。

基础设施

应用程序环境中包含的所有资源和资产。

基础设施即代码（IaC）

通过一组配置文件预置和管理应用程序基础设施的过程。IaC 旨在帮助您集中管理基础设施、实现资源标准化和快速扩展，使新环境具有可重复性、可靠性和一致性。

工业物联网（IIoT）

在工业领域使用联网的传感器和设备，例如制造业、能源、汽车、医疗保健、生命科学和农业。有关更多信息，请参阅制定工业物联网（IIoT）数字化转型策略。

检查 VPC

在 AWS 多账户架构中，一种集中式 VPC，用于管理 VPC（相同或不同 AWS 区域）、互联网和本地网络之间的网络流量检查。AWS 安全参考架构建议使用入站、出站和检查 VPC 设置网络账户，保护应用程序与广泛的互联网之间的双向接口。

物联网（IoT）

由带有嵌入式传感器或处理器的连接物理对象组成的网络，这些传感器或处理器通过互联网或本地通信网络与其他设备和系统进行通信。有关更多信息，请参阅什么是 IoT？

可解释性

它是机器学习模型的一种特征，描述了人类可以理解模型的预测如何取决于其输入的程度。有关更多信息，请参阅使用 AWS 实现机器学习模型的可解释性。

IoT

参见物联网。

IT 信息库（ITIL）

提供 IT 服务并使这些服务符合业务要求的一套最佳实践。ITIL 是 ITSM 的基础。

IT 服务管理（ITSM）

为组织设计、实施、管理和支持 IT 服务的相关活动。有关将云运营与 ITSM 工具集成的信息，请参阅运营集成指南。

ITIL

请参阅 IT 信息库。

ITSM

请参阅 IT 服务管理。

L

基于标签的访问控制 (LBAC)

强制访问控制 (MAC) 的一种实施方式，其中明确为用户和数据本身分配了安全标签值。用户安全标签和数据安全标签之间的交集决定了用户可以看到哪些行和列。

登录区

landing zone 是一个架构精良的多账户 AWS 环境，具有可扩展性和安全性。这是一个起点，您的组织可以从这里放心地在安全和基础设施环境中快速启动和部署工作负载和应用程序。有关登录区的更多信息，请参阅设置安全且可扩展的多账户 AWS 环境。

大规模迁移

迁移 300 台或更多服务器。

LBAC

参见基于标签的访问控制。

最低权限

授予执行任务所需的最低权限的最佳安全实践。有关更多信息，请参阅 IAM 文档中的应用最低权限许可。

直接迁移

见 7 R。

小端序系统

一个先存储最低有效字节的系统。另请参见字节顺序。

下层环境

参见环境。

M

机器学习（ML）

一种使用算法和技术进行模式识别和学习的人工智能。ML 对记录的数据（例如物联网（IoT）数据）进行分析和学习，以生成基于模式的统计模型。有关更多信息，请参阅机器学习。

主分支

参见分支。

恶意软件

旨在危害计算机安全或隐私的软件。恶意软件可能会破坏计算机系统、泄露敏感信息或获得未经授权的访问。恶意软件的示例包括病毒、蠕虫、勒索软件、特洛伊木马、间谍软件和键盘记录器。

托管服务

AWS 服务 它 AWS 运行基础设施层、操作系统和平台，您可以访问端点来存储和检索数据。亚马逊简单存储服务 (Amazon S3) Service 和 Amazon DynamoDB 就是托管服务的示例。这些服务也称为抽象服务。

制造执行系统 (MES)

一种软件系统，用于跟踪、监控、记录和控制在车间将原材料转化为成品的生产过程。

MAP

参见迁移加速计划。

机制

一个完整的过程，在此过程中，您可以创建工具，推动工具的采用，然后检查结果以进行调整。机制是一种在运行过程中自我增强和改进的循环。有关更多信息，请参阅在 Well-Architect AWS ed 框架中构建机制。

成员账户

AWS 账户 除属于组织中的管理账户之外的所有账户 AWS Organizations。一个账户一次只能是一个组织的成员。

MES

参见制造执行系统。

消息队列遥测传输 (MQTT)

一种基于发布/订阅模式的轻量级 machine-to-machine (M2M) 通信协议，适用于资源受限的物联网设备。

微服务

一种小型独立服务，通过明确定义的 API 进行通信，通常由小型独立团队拥有。例如，保险系统可能包括映射到业务能力（如销售或营销）或子域（如购买、理赔或分析）的微服务。微服务的好处包括敏捷、灵活扩展、易于部署、可重复使用的代码和恢复能力。有关更多信息，请参阅使用 AWS 无服务器服务集成微服务。

微服务架构

一种使用独立组件构建应用程序的方法，这些组件将每个应用程序进程作为微服务运行。这些微服务使用轻量级 API 通过明确定义的接口进行通信。该架构中的每个微服务都可以更新、部署和扩展，以满足对应用程序特定功能的需求。有关更多信息，请参阅在上实现微服务。 AWS

迁移加速计划（MAP）

AWS 该计划提供咨询支持、培训和服务，以帮助组织为迁移到云奠定坚实的运营基础，并帮助抵消迁移的初始成本。MAP 提供了一种以系统的方式执行遗留迁移的迁移方法，以及一套用于自动执行和加速常见迁移场景的工具。

大规模迁移

将大部分应用程序组合分波迁移到云中的过程，在每一波中以更快的速度迁移更多应用程序。本阶段使用从早期阶段获得的最佳实践和经验教训，实施由团队、工具和流程组成的迁移工厂，通过自动化和敏捷交付简化工作负载的迁移。这是 AWS 迁移策略的第三阶段。

迁移工厂

跨职能团队，通过自动化、敏捷的方法简化工作负载迁移。迁移工厂团队通常包括运营、业务分析师和所有者、迁移工程师、开发 DevOps 人员和冲刺专业人员。20% 到 50% 的企业应用程序组合由可通过工厂方法优化的重复模式组成。有关更多信息，请参阅本内容集中有关迁移工厂的讨论和云迁移工厂指南。

迁移元数据

有关完成迁移所需的应用程序和服务器的信息。每种迁移模式都需要一套不同的迁移元数据。迁移元数据的示例包括目标子网、安全组和 AWS 账户。

迁移模式

一种可重复的迁移任务，详细列出了迁移策略、迁移目标以及所使用的迁移应用程序或服务。示例：使用 AWS 应用程序迁移服务重新托管向 Amazon EC2 的迁移。

迁移组合评测（MPA）

一种在线工具，可提供信息，用于验证迁移到的业务案例。 AWS Cloud MPA 提供了详细的组合评测（服务器规模调整、定价、TCO 比较、迁移成本分析）以及迁移计划（应用程序数据分析和数据收集、应用程序分组、迁移优先级排序和波次规划）。所有 AWS 顾问和 APN 合作伙伴顾问均可免费使用 MPA 工具（需要登录）。

迁移准备情况评测（MRA）

使用 AWS CAF 深入了解组织的云就绪状态、确定优势和劣势以及制定行动计划以缩小已发现差距的过程。有关更多信息，请参阅迁移准备指南。MRA 是 AWS 迁移策略的第一阶段。

迁移策略

用于将工作负载迁移到的方法 AWS Cloud。有关更多信息，请参阅此词汇表中的 7 R 条目和动员组织以加快大规模迁移。

ML

参见机器学习。

现代化

将过时的（原有的或单体）应用程序及其基础设施转变为云中敏捷、弹性和高度可用的系统，以降低成本、提高效率和利用创新。有关更多信息，请参阅中的应用程序现代化策略。 AWS Cloud

现代化准备情况评估

一种评估方式，有助于确定组织应用程序的现代化准备情况；确定收益、风险和依赖关系；确定组织能够在多大程度上支持这些应用程序的未来状态。评估结果是目标架构的蓝图、详细说明现代化进程发展阶段和里程碑的路线图以及解决已发现差距的行动计划。有关更多信息，请参阅中的评估应用程序的现代化准备情况 AWS Cloud。

单体应用程序（单体式）

作为具有紧密耦合进程的单个服务运行的应用程序。单体应用程序有几个缺点。如果某个应用程序功能的需求激增，则必须扩展整个架构。随着代码库的增长，添加或改进单体应用程序的功能也会变得更加复杂。若要解决这些问题，可以使用微服务架构。有关更多信息，请参阅将单体分解为微服务。

MPA

参见迁移组合评估。

MQTT

请参阅消息队列遥测传输。

多分类器

一种帮助为多个类别生成预测（预测两个以上结果之一）的过程。例如，ML 模型可能会询问“这个产品是书、汽车还是手机？”或“此客户最感兴趣什么类别的产品？”

可变基础架构

一种用于更新和修改现有生产工作负载基础架构的模型。为了提高一致性、可靠性和可预测性，Well-Architect AWS ed Framework 建议使用不可变基础设施作为最佳实践。

O

OAC

请参阅源站访问控制。

OAI

参见源访问身份。

OCM

参见组织变更管理。

离线迁移

一种迁移方法，在这种方法中，源工作负载会在迁移过程中停止运行。这种方法会延长停机时间，通常用于小型非关键工作负载。

OI

参见运营集成。

OLA

参见运营层协议。

在线迁移

一种迁移方法，在这种方法中，源工作负载无需离线即可复制到目标系统。在迁移过程中，连接工作负载的应用程序可以继续运行。这种方法的停机时间为零或最短，通常用于关键生产工作负载。

OPC-UA

参见开放流程通信-统一架构。

开放流程通信-统一架构 (OPC-UA)

一种用于工业自动化的 machine-to-machine （M2M）通信协议。OPC-UA 提供了数据加密、身份验证和授权方案的互操作性标准。

运营级别协议（OLA）

一项协议，阐明了 IT 职能部门承诺相互交付的内容，以支持服务水平协议（SLA）。

运营准备情况审查 (ORR)

一份问题清单和相关的最佳实践，可帮助您理解、评估、预防或缩小事件和可能的故障的范围。有关更多信息，请参阅 Well-Architecte AWS d Frame work 中的运营准备情况评估 (ORR)。

操作技术 (OT)

与物理环境配合使用以控制工业运营、设备和基础设施的硬件和软件系统。在制造业中，OT 和信息技术 (IT) 系统的集成是工业 4.0 转型的重点。

运营整合（OI）

在云中实现运营现代化的过程，包括就绪计划、自动化和集成。有关更多信息，请参阅运营整合指南。

组织跟踪

由 AWS CloudTrail 此创建的跟踪记录组织 AWS 账户 中所有人的所有事件 AWS Organizations。该跟踪是在每个 AWS 账户 中创建的，属于组织的一部分，并跟踪每个账户的活动。有关更多信息，请参阅 CloudTrail文档中的为组织创建跟踪。

组织变革管理（OCM）

一个从人员、文化和领导力角度管理重大、颠覆性业务转型的框架。OCM 通过加快变革采用、解决过渡问题以及推动文化和组织变革，帮助组织为新系统和战略做好准备和过渡。在 AWS 迁移策略中，该框架被称为人员加速，因为云采用项目需要变更的速度。有关更多信息，请参阅 OCM 指南。

来源访问控制（OAC）

在中 CloudFront，一个增强的选项，用于限制访问以保护您的亚马逊简单存储服务 (Amazon S3) 内容。OAC 全部支持所有 S3 存储桶 AWS 区域、使用 AWS KMS (SSE-KMS) 进行服务器端加密，以及对 S3 存储桶的动态PUT和DELETE请求。

来源访问身份（OAI）

在中 CloudFront，一个用于限制访问权限以保护您的 Amazon S3 内容的选项。当您使用 OAI 时，CloudFront 会创建一个 Amazon S3 可以对其进行身份验证的委托人。经过身份验证的委托人只能通过特定 CloudFront 分配访问 S3 存储桶中的内容。另请参阅 OAC，其中提供了更精细和增强的访问控制。

或者

参见运营准备情况审查。

OT

参见运营技术。

出站（出口）VPC

在 AWS 多账户架构中，一种处理从应用程序内部启动的网络连接的 VPC。AWS 安全参考架构建议使用入站、出站和检查 VPC 设置网络账户，保护应用程序与广泛的互联网之间的双向接口。

P

权限边界

附加到 IAM 主体的 IAM 管理策略，用于设置用户或角色可以拥有的最大权限。有关更多信息，请参阅 IAM 文档中的权限边界。

个人身份信息 (PII)

直接查看其他相关数据或与之配对时可用于合理推断个人身份的信息。PII 的示例包括姓名、地址和联系信息。

PII

查看个人身份信息。

playbook

一套预定义的步骤，用于捕获与迁移相关的工作，例如在云中交付核心运营功能。playbook 可以采用脚本、自动化运行手册的形式，也可以是操作现代化环境所需的流程或步骤的摘要。

PLC

参见可编程逻辑控制器。

PLM

参见产品生命周期管理。

策略

一个对象，可以在中定义权限（参见基于身份的策略）、指定访问条件（参见基于资源的策略）或定义组织中所有账户的最大权限 AWS Organizations （参见服务控制策略）。

多语言持久性

根据数据访问模式和其他要求，独立选择微服务的数据存储技术。如果您的微服务采用相同的数据存储技术，它们可能会遇到实现难题或性能不佳。如果微服务使用最适合其需求的数据存储，则可以更轻松地实现微服务，并获得更好的性能和可扩展性。有关更多信息，请参阅在微服务中实现数据持久性。

组合评测

一个发现、分析和确定应用程序组合优先级以规划迁移的过程。有关更多信息，请参阅评估迁移准备情况。

谓词

返回`true`或的查询条件`false`，通常位于子`WHERE`句中。

谓词下推

一种数据库查询优化技术，可在传输前筛选查询中的数据。这减少了必须从关系数据库检索和处理的数据量，并提高了查询性能。

预防性控制

一种安全控制，旨在防止事件发生。这些控制是第一道防线，帮助防止未经授权的访问或对网络的意外更改。有关更多信息，请参阅在 AWS上实施安全控制中的预防性控制。

主体

中 AWS 可以执行操作和访问资源的实体。此实体通常是 IAM 角色的根用户或用户。 AWS 账户有关更多信息，请参阅 IAM 文档中角色术语和概念中的主体。

隐私设计

一种贯穿整个工程化过程考虑隐私的系统工程方法。

私有托管区

私有托管区就是一个容器，其中包含的信息说明您希望 Amazon Route 53 如何响应一个或多个 VPC 中的某个域及其子域的 DNS 查询。有关更多信息，请参阅 Route 53 文档中的私有托管区的使用。

主动控制

一种安全控制措施，旨在防止部署不合规的资源。这些控件会在资源置备之前对其进行扫描。如果资源与控件不兼容，则不会对其进行配置。有关更多信息，请参阅 AWS Control Tower 文档中的控制参考指南，并参见在上实施安全控制中的主动控制 AWS。

产品生命周期管理 (PLM)

在产品的整个生命周期中，从设计、开发和上市，到成长和成熟，再到衰落和移除，对产品进行数据和流程的管理。

生产环境

参见环境。

可编程逻辑控制器 (PLC)

在制造业中，一种高度可靠、适应性强的计算机，用于监控机器并实现制造过程自动化。

假名化

用占位符值替换数据集中个人标识符的过程。假名化可以帮助保护个人隐私。假名化数据仍被视为个人数据。

发布/订阅（发布/订阅）

一种支持微服务间异步通信的模式，以提高可扩展性和响应能力。例如，在基于微服务的 MES 中，微服务可以将事件消息发布到其他微服务可以订阅的频道。系统可以在不更改发布服务的情况下添加新的微服务。

Q

查询计划

一系列步骤，例如指令，用于访问 SQL 关系数据库系统中的数据。

查询计划回归

当数据库服务优化程序选择的最佳计划不如数据库环境发生特定变化之前时。这可能是由统计数据、约束、环境设置、查询参数绑定更改和数据库引擎更新造成的。

R

RACI 矩阵

参见 “负责任、负责、咨询、知情”（RACI）。

勒索软件

一种恶意软件，旨在阻止对计算机系统或数据的访问，直到付款为止。

RASCI 矩阵

参见 “负责任、负责、咨询、知情”（RACI）。

RCAC

请参阅行和列访问控制。

只读副本

用于只读目的的数据库副本。您可以将查询路由到只读副本，以减轻主数据库的负载。

重新架构师

见 7 R。

恢复点目标 (RPO)

自上一个数据恢复点以来可接受的最长时间。这决定了从上一个恢复点到服务中断之间可接受的数据丢失情况。

恢复时间目标 (RTO)

服务中断和服务恢复之间可接受的最大延迟。

重构

见 7 R。

区域

地理区域内的 AWS 资源集合。每一个 AWS 区域 都相互隔离，彼此独立，以提供容错、稳定性和弹性。有关更多信息，请参阅指定 AWS 区域 您的账户可以使用的账户。

回归

一种预测数值的 ML 技术。例如，要解决“这套房子的售价是多少？”的问题 ML 模型可以使用线性回归模型，根据房屋的已知事实（如建筑面积）来预测房屋的销售价格。

重新托管

见 7 R。

版本

在部署过程中，推动生产环境变更的行为。

搬迁

见 7 R。

更换平台

见 7 R。

回购

见 7 R。

故障恢复能力

应用程序抵御中断或从中断中恢复的能力。在中规划弹性时，高可用性和灾难恢复是常见的考虑因素。 AWS Cloud有关更多信息，请参阅AWS Cloud 弹性。

基于资源的策略

一种附加到资源的策略，例如 AmazonS3 存储桶、端点或加密密钥。此类策略指定了允许哪些主体访问、支持的操作以及必须满足的任何其他条件。

责任、问责、咨询和知情（RACI）矩阵

定义参与迁移活动和云运营的所有各方的角色和责任的矩阵。矩阵名称源自矩阵中定义的责任类型：负责 (R)、问责 (A)、咨询 (C) 和知情 (I)。支持 (S) 类型是可选的。如果包括支持，则该矩阵称为 RASCI 矩阵，如果将其排除在外，则称为 RACI 矩阵。

响应性控制

一种安全控制，旨在推动对不良事件或偏离安全基线的情况进行修复。有关更多信息，请参阅在AWS上实施安全控制中的响应性控制。

保留

见 7 R。

退休

见 7 R。

旋转

定期更新密钥以使攻击者更难访问凭据的过程。

行列访问控制 (RCAC)

使用已定义访问规则的基本、灵活的 SQL 表达式。RCAC 由行权限和列掩码组成。

RPO

参见恢复点目标。

RTO

参见恢复时间目标。

运行手册

执行特定任务所需的一套手动或自动程序。它们通常是为了简化重复性操作或高错误率的程序而设计的。

S

SAML 2.0

许多身份提供商 (IdPs) 使用的开放标准。此功能支持联合单点登录 (SSO)，因此用户无需在 IAM 中为组织中的所有人创建用户即可登录 AWS Management Console 或调用 AWS API 操作。有关基于 SAML 2.0 的联合身份验证的更多信息，请参阅 IAM 文档中的关于基于 SAML 2.0 的联合身份验证。

SCADA

参见监督控制和数据采集。

SCP

参见服务控制政策。

secret

在中 AWS Secrets Manager，您以加密形式存储的机密或受限信息，例如密码或用户凭证。它由密钥值及其元数据组成。密钥值可以是二进制、单个字符串或多个字符串。有关更多信息，请参阅 Secret s Manager 密钥中有什么？ 在 Secrets Manager 文档中。

安全控制

一种技术或管理防护机制，可防止、检测或降低威胁行为体利用安全漏洞的能力。安全控制主要有四种类型：预防性、侦测、响应式和主动式。

安全加固

缩小攻击面，使其更能抵御攻击的过程。这可能包括删除不再需要的资源、实施授予最低权限的最佳安全实践或停用配置文件中不必要的功能等操作。

安全信息和事件管理（SIEM）系统

结合了安全信息管理（SIM）和安全事件管理（SEM）系统的工具和服务。SIEM 系统会收集、监控和分析来自服务器、网络、设备和其他来源的数据，以检测威胁和安全漏洞，并生成警报。

安全响应自动化

一种预定义和编程的操作，旨在自动响应或修复安全事件。这些自动化可作为侦探或响应式安全控制措施，帮助您实施 AWS 安全最佳实践。自动响应操作的示例包括修改 VPC 安全组、修补 Amazon EC2 实例或轮换证书。

服务器端加密

在目的地对数据进行加密，由接收方 AWS 服务 进行加密。

服务控制策略（SCP）

一种策略，用于集中控制 AWS Organizations的组织中所有账户的权限。SCP 为管理员可以委托给用户或角色的操作定义了防护机制或设定了限制。您可以将 SCP 用作允许列表或拒绝列表，指定允许或禁止哪些服务或操作。有关更多信息，请参阅 AWS Organizations 文档中的服务控制策略。

服务端点

的入口点的 URL AWS 服务。您可以使用端点，通过编程方式连接到目标服务。有关更多信息，请参阅 AWS 一般参考 中的 AWS 服务 端点。

服务水平协议（SLA）

一份协议，阐明了 IT 团队承诺向客户交付的内容，比如服务正常运行时间和性能。

服务级别指示器 (SLI)

对服务性能方面的衡量，例如其错误率、可用性或吞吐量。

服务级别目标 (SLO)

代表服务运行状况的目标指标，由服务级别指标衡量。

责任共担模式

描述您在云安全与合规方面共同承担 AWS 的责任的模型。 AWS 负责云的安全，而您则负责云中的安全。有关更多信息，请参阅责任共担模式。

暹粒

参见安全信息和事件管理系统。

单点故障 (SPOF)

应用程序的单个关键组件出现故障，可能会中断系统。

SLA

参见服务级别协议。

SLI

参见服务级别指标。

SLO

参见服务级别目标。

split-and-seed 模型

一种扩展和加速现代化项目的模式。随着新功能和产品发布的定义，核心团队会拆分以创建新的产品团队。这有助于扩展组织的能力和服务，提高开发人员的工作效率，支持快速创新。有关更多信息，请参阅中的分阶段实现应用程序现代化的方法。 AWS Cloud

恶作剧

参见单点故障。

星型架构

一种数据库组织结构，它使用一个大型事实表来存储交易数据或测量数据，并使用一个或多个较小的维度表来存储数据属性。此结构专为在数据仓库中使用或用于商业智能目的而设计。

strangler fig 模式

一种通过逐步重写和替换系统功能直至可以停用原有的系统来实现单体系统现代化的方法。这种模式用无花果藤作为类比，这种藤蔓成长为一棵树，最终战胜并取代了宿主。该模式是由 Martin Fowler 提出的，作为重写单体系统时管理风险的一种方法。有关如何应用此模式的示例，请参阅使用容器和 Amazon API Gateway 逐步将原有的 Microsoft ASP.NET（ASMX）Web 服务现代化。

子网

您的 VPC 内的一个 IP 地址范围。子网必须位于单个可用区中。

监控和数据采集 (SCADA)

在制造业中，一种使用硬件和软件来监控有形资产和生产操作的系统。

对称加密

一种加密算法，它使用相同的密钥来加密和解密数据。

综合测试

以模拟用户交互的方式测试系统，以检测潜在问题或监控性能。你可以使用 Amazon S CloudWatch ynthetics 来创建这些测试。

T

标签

键值对，充当用于组织资源的元数据。 AWS 标签可帮助您管理、识别、组织、搜索和筛选资源。有关更多信息，请参阅标记您的 AWS 资源。

目标变量

您在监督式 ML 中尝试预测的值。这也被称为结果变量。例如，在制造环境中，目标变量可能是产品缺陷。

任务列表

一种通过运行手册用于跟踪进度的工具。任务列表包含运行手册的概述和要完成的常规任务列表。对于每项常规任务，它包括预计所需时间、所有者和进度。

测试环境

参见环境。

训练

为您的 ML 模型提供学习数据。训练数据必须包含正确答案。学习算法在训练数据中查找将输入数据属性映射到目标（您希望预测的答案）的模式。然后输出捕获这些模式的 ML 模型。然后，您可以使用 ML 模型对不知道目标的新数据进行预测。

中转网关

中转网关是网络中转中心，您可用它来互连 VPC 和本地网络。有关更多信息，请参阅 AWS Transit Gateway 文档中的什么是公交网关。

基于中继的工作流程

一种方法，开发人员在功能分支中本地构建和测试功能，然后将这些更改合并到主分支中。然后，按顺序将主分支构建到开发、预生产和生产环境。

可信访问权限

向您指定的服务授予权限，该服务可代表您在其账户中执行任务。 AWS Organizations 当需要服务相关的角色时，受信任的服务会在每个账户中创建一个角色，为您执行管理任务。有关更多信息，请参阅 AWS Organizations 文档中的AWS Organizations 与其他 AWS 服务一起使用。

优化

更改训练过程的各个方面，以提高 ML 模型的准确性。例如，您可以通过生成标签集、添加标签，并在不同的设置下多次重复这些步骤来优化模型，从而训练 ML 模型。

双披萨团队

一个小 DevOps 团队，你可以用两个披萨来喂食。双披萨团队的规模可确保在软件开发过程中充分协作。

U

不确定性

这一概念指的是不精确、不完整或未知的信息，这些信息可能会破坏预测式 ML 模型的可靠性。不确定性有两种类型：认知不确定性是由有限的、不完整的数据造成的，而偶然不确定性是由数据中固有的噪声和随机性导致的。有关更多信息，请参阅量化深度学习系统中的不确定性指南。

无差别任务

也称为繁重工作，即创建和运行应用程序所必需的工作，但不能为最终用户提供直接价值或竞争优势。无差别任务的示例包括采购、维护和容量规划。

上层环境

参见环境。

V

vacuum 操作

一种数据库维护操作，包括在增量更新后进行清理，以回收存储空间并提高性能。

版本控制

跟踪更改的过程和工具，例如存储库中源代码的更改。

VPC 对等连接

两个 VPC 之间的连接，允许您使用私有 IP 地址路由流量。有关更多信息，请参阅 Amazon VPC 文档中的什么是 VPC 对等连接。

漏洞

损害系统安全的软件缺陷或硬件缺陷。

W

热缓存

一种包含经常访问的当前相关数据的缓冲区缓存。数据库实例可以从缓冲区缓存读取，这比从主内存或磁盘读取要快。

暖数据

不常访问的数据。查询此类数据时，通常可以接受中速查询。

窗口函数

一个 SQL 函数，用于对一组以某种方式与当前记录相关的行进行计算。窗口函数对于处理任务很有用，例如计算移动平均线或根据当前行的相对位置访问行的值。

工作负载

一系列资源和代码，它们可以提供商业价值，如面向客户的应用程序或后端过程。

工作流

迁移项目中负责一组特定任务的职能小组。每个工作流都是独立的，但支持项目中的其他工作流。例如，组合工作流负责确定应用程序的优先级、波次规划和收集迁移元数据。组合工作流将这些资产交付给迁移工作流，然后迁移服务器和应用程序。

蠕虫

参见一次写入，多读。

WQF

请参阅 AWS 工作负载资格框架。

一次写入，多次读取 (WORM)

一种存储模型，它可以一次写入数据并防止数据被删除或修改。授权用户可以根据需要多次读取数据，但他们无法对其进行更改。这种数据存储基础架构被认为是不可变的。

Z

零日漏洞利用

一种利用未修补漏洞的攻击，通常是恶意软件。

零日漏洞

生产系统中不可避免的缺陷或漏洞。威胁主体可能利用这种类型的漏洞攻击系统。开发人员经常因攻击而意识到该漏洞。

僵尸应用程序

平均 CPU 和内存使用率低于 5% 的应用程序。在迁移项目中，通常会停用这些应用程序。

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。