# AWS Glue Apache Spark ジョブのパフォーマンスチューニングのベストプラクティス

# : AWS Glue Apache Spark ジョブのパフォーマンスチューニングのベストプラクティス

# Table of Contents

序章 ..... 1
主な トピック ..... 2
アーキテクチャ ..... 2
回復力のある分散データセット ..... 3
遅延評価 ..... 5
Spark アプリケーションの用語 ..... 6
並列処理 ..... 7
カタリストオプティマイザ ..... 8
パフォーマンスの問題の調査 ..... 10
Spark UI を使用してボトルネックを特定する ..... 10
パフォーマンスを調整するための戦略 ..... 12
パフォーマンスチューニングのベースライン戦略 ..... 12
Spark ジョブパフォーマンスのチューニング方法 ..... 13
クラスター容量のスケーリング ..... 14
CloudWatch メトリクス ..... 14
Spark UI ..... 15
最新バージョンを使用する ..... 16
データスキャンの量を減らす ..... 17
CloudWatch メトリクス ..... 17
Spark UI ..... 18
タスクの並列化 ..... 26
CloudWatch メトリクス ..... 27
Spark UI ..... 27
シャッフルの最適化 ..... 33
CloudWatch メトリクス ..... 34
Spark UI ..... 34
計画オーバーヘッドを最小限に抑える ..... 43
CloudWatch メトリクス ..... 43
Spark UI ..... 44
ユーザー定義関数の最適化 ..... 44
標準 Python UDF ..... 46
ベクトル化 UDF ..... 46
Spark SQL ..... 47
ビッグデータに pandas を使用する ..... 48

リソース ..... 49
ドキュメント履歴 ..... 50
用語集 ..... 51
# ..... 51
A ..... 52
B ..... 55
C ..... 57
D ..... 60
E ..... 64
F ..... 66
G ..... 67
H ..... 68
I ..... 69
L ..... 71
M ..... 72
O ..... 76
P ..... 79
Q ..... 81
R ..... 82
S ..... 84
T ..... 88
U ..... 89
V ..... 90
W ..... 90
Z ..... 91
..... xciii

# AWS Glue Apache Spark ジョブのパフォーマンスチューニングのベストプラクティス

アマゾン ウェブ サービス (AWS) のローママヤス、オニキュラ· クリタカ

2023 年 12 月 (ドキュメント履歴)

AWS Glue には、パフォーマンスを調整するためのさまざまなオプションが用意されています。このガイドでは、Apache Spark のチューニング AWS Glue に関する重要なトピックを定義します。次に、Apache Spark ジョブ AWS Glue のこれらを調整する際に従うべきベースライン戦略を提供します。このガイドでは、 で利用可能なメトリクスを解釈してパフォーマンスの問題を特定する方法について説明します AWS Glue。次に、これらの問題に対処するための戦略を組み込んで、パフォーマンスを最大化し、コストを最小限に抑えます。

このガイドでは、以下のチューニングプラクティスについて説明します。

- クラスター容量のスケーリング
- AWS Glue 最新バージョンを使用する
- データスキャンの量を減らす
- タスクの並列化
- 計画オーバーヘッドを最小限に抑える
- シャッフルの最適化
- ユーザー定義関数の最適化

# Apache Spark の主なトピック

このセクションでは、Apache Spark のパフォーマンスを調整するための Apache AWS Glue Spark の基本概念と主要なトピックについて説明します。実際の調整戦略について説明する前に、これらの概念とトピックを理解することが重要です。

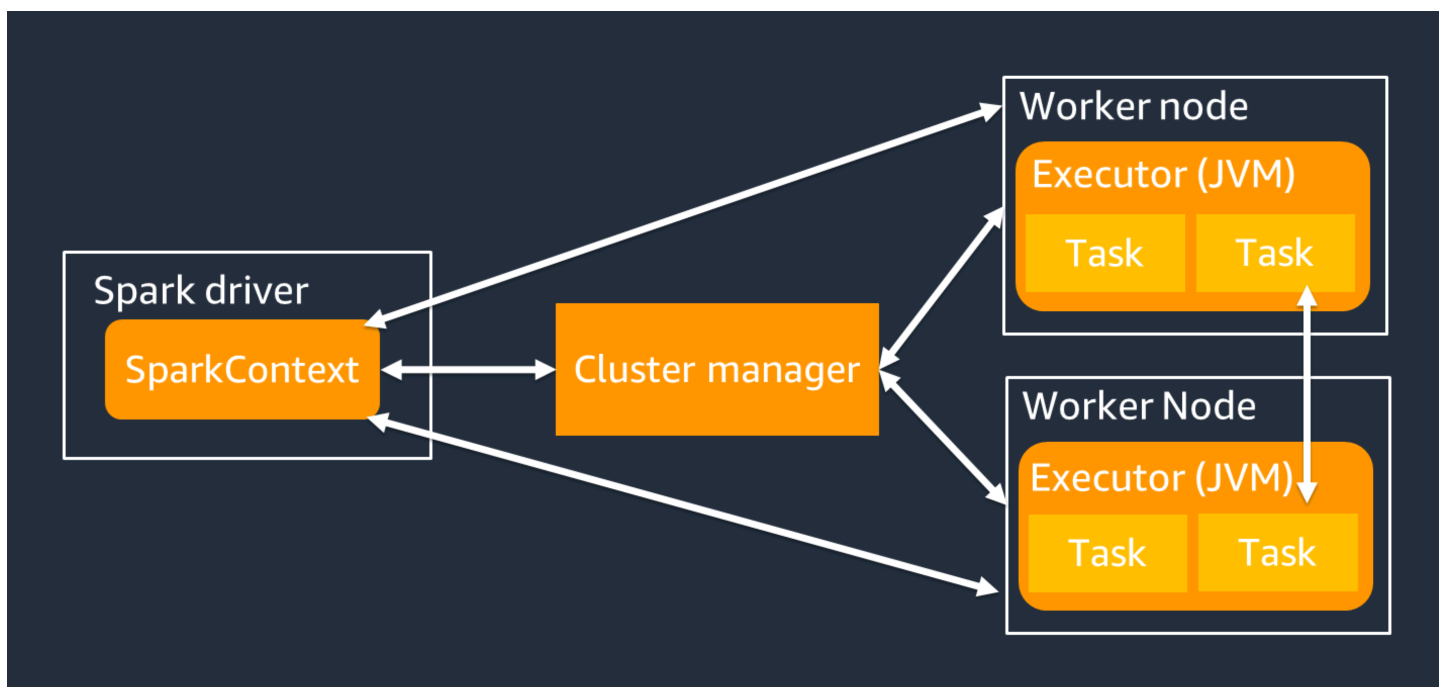
## アーキテクチャ

Spark ドライバーは主に、Spark アプリケーションを個々のワーカーで達成できるタスクに分割する役割を担います。Spark ドライバーには以下の責任があります。

- コード`main()`での実行
- 実行プランの生成
- クラスター上のリソースを管理するクラスターマネージャーと連動した Spark エグゼキューターのプロビジョニング
- Spark エグゼキューターのタスクのスケジュール設定とタスクのリクエスト
- タスクの進行状況と復旧の管理

`SparkContext` オブジェクトを使用して、ジョブ実行用の Spark ドライバーを操作します。

Spark エグゼキューターは、Spark ドライバーから渡されるデータを保持し、タスクを実行するワーカーです。Spark エグゼキューターの数は、クラスターのサイズに応じて増減します。

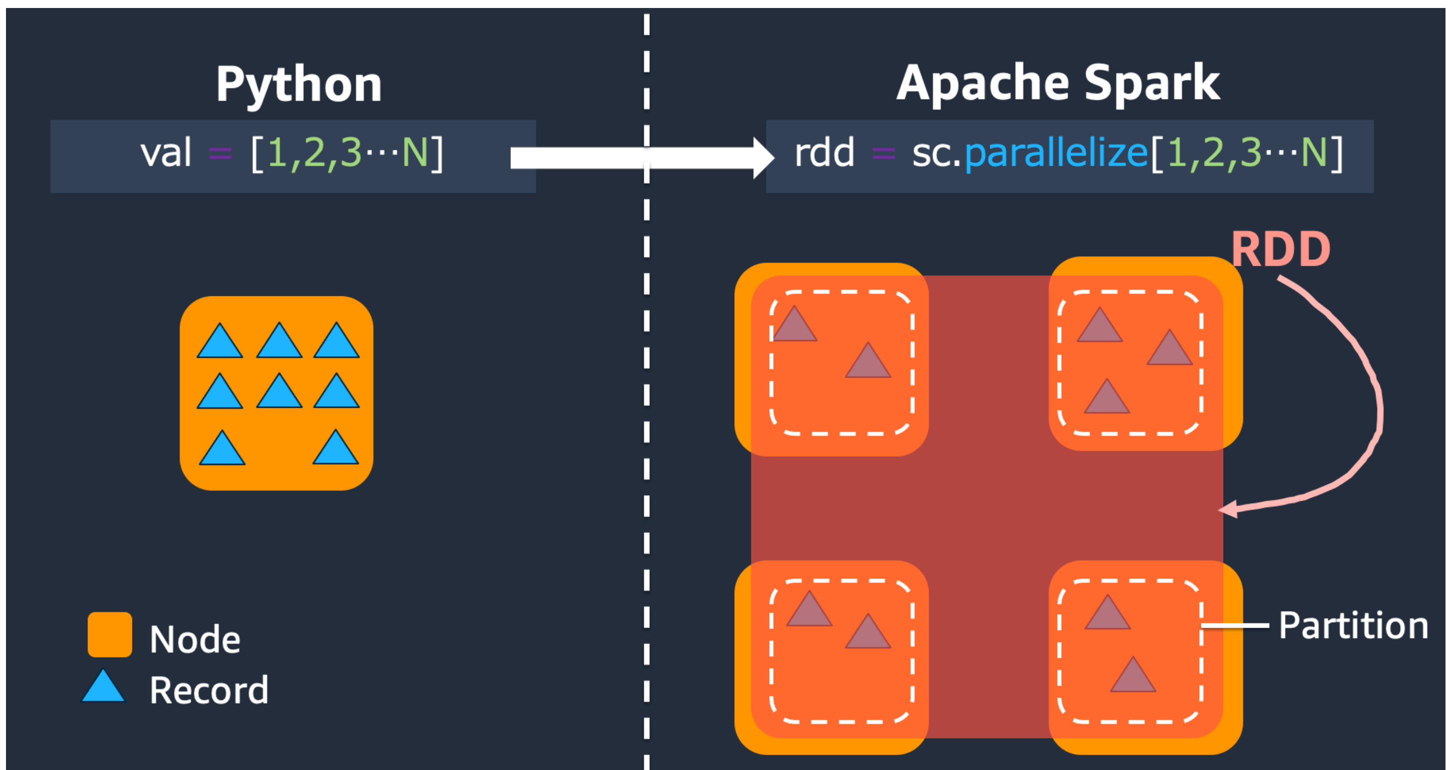



> **Note**
>
> Spark エグゼキューターには複数のスロットがあるため、複数のタスクを並行して処理できます。Spark は、デフォルトで仮想 CPU (vCPU) コアごとに 1 つのタスクをサポートします。例えば、エグゼキューターに 4 つの CPU コアがある場合、4 つの同時タスクを実行できます。

## 回復力のある分散データセット

Spark は、Spark エグゼキューター全体で大規模なデータセットを保存および追跡する複雑なジョブを実行します。Spark ジョブのコードを記述するときは、ストレージの詳細について考える必要はありません。Spark は、回復力のある分散データセット (RDD) 抽象化を提供します。RDD 抽象化は、並列で操作でき、クラスターの Spark エグゼキュター間でパーティション化できる要素のコレクションです。

次の図は、Python スクリプトが一般的な環境で実行されているときと、Spark フレームワーク () で実行されているときに、データをメモリに保存する方法の違いを示していますPySpark。




- Python – Python スクリプト`val = [1,2,3...N]`に書き込むと、コードが実行されている単一マシンのメモリにデータが保持されます。
- PySpark – Spark は、複数の Spark エグゼキューターのメモリ全体に分散されたデータをロードして処理するための RDD データ構造を提供します。などのコードで RDD を生成`rdd = sc.parallelize[1,2,3...N]`できます。Spark は複数の Spark エグゼキューター間でデータをメモリに自動的に分散して保持できます。

多くの AWS Glue ジョブでは、 DynamicFramesと Spark を介して AWS Glue RDDs を使用しますDataFrames。これらは、RDD 内のデータのスキーマを定義し、その追加情報を使用して上位レベルのタスクを実行できるようにする抽象化です。RDDs は内部的に使用されるため、データは透過的に分散され、次のコードの複数のノードにロードされます。

- 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 には次の機能があります。

- RDDsは、パーティション と呼ばれる複数のパートに分割されたデータで構成されます。各 Spark エグゼキューターは 1 つ以上のパーティションをメモリに保存し、データは複数のエグゼキューターに分散されます。
- RDDs はイミュータブルな です。つまり、作成後に変更することはできません。を変更するには DataFrame、次のセクションで定義されている変換 を使用できます。
- RDDs利用可能なノード間でデータをレプリケートするため、ノード障害から自動的に復旧できます。

## 遅延評価

RDDs、既存のデータセットから新しいデータセットを作成する変換 と、データセットで計算を実行した後にドライバープログラムに値を返すアクション の 2 種類のオペレーションをサポートします。

- 変換 — RDDsはイミュータブルであるため、変換を使用してのみ変更できます。

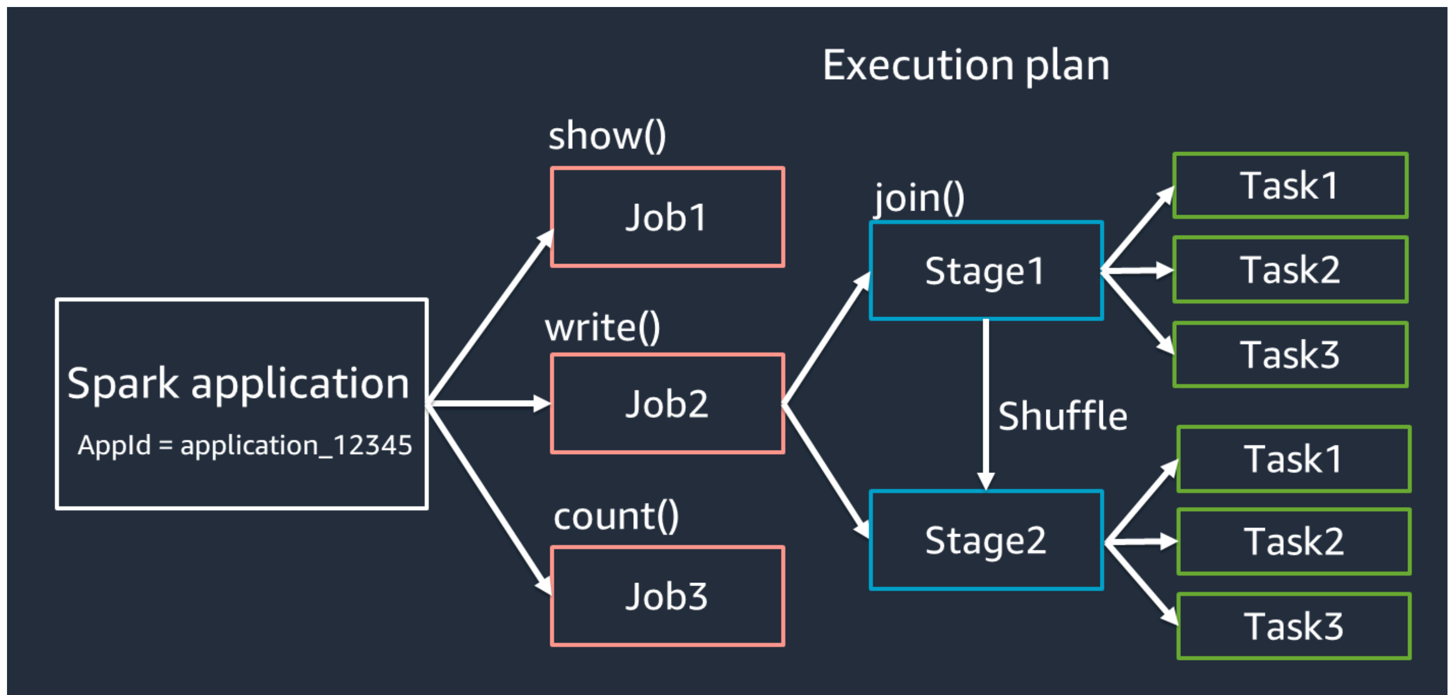
  例えば、 `map`は各データセット要素を関数に渡し、結果を表す新しい RDD を返す変換です。`map` メソッドは出力を返さないことに注意してください。Spark は、結果を操作するのではなく、将来の抽象変換を保存します。Spark は、 アクションを呼び出すまで変換を処理しません。
- アクション — 変換を使用して、論理的な変換計画を構築します。計算を開始するには、、`write`、、 `count` `show`などのアクションを実行します`collect`。

  Spark のすべての変換は、すぐに結果を計算しないという点で遅延しています。代わりに、Spark は Amazon Simple Storage Service (Amazon S3) オブジェクトなど、一部のベースデータセットに適用された一連の変換を記憶します。変換は、アクションで結果をドライバーに返す必要がある場合にのみ計算されます。この設計により、Spark をより効率的に実行できます。例えば、`map`変換によって作成されたデータセットが、 などの行数を大幅に減らす変換によってのみ消費される状況を考えてみましょう`reduce`。その後、マッピングされた大きなデータセットを渡す代わりに、両方の変換を行った小さなデータセットをドライバーに渡すことができます。

# Spark アプリケーションの用語

このセクションでは、Spark アプリケーションの用語について説明します。Spark ドライバーは実行プランを作成し、いくつかの抽象化でアプリケーションの動作を制御します。以下の用語は、Spark UI での開発、デバッグ、パフォーマンスチューニングに重要です。

- アプリケーション — Spark セッション (Spark コンテキスト) に基づきます。などの一意の ID で識別されます`<application_XXX>`。
- ジョブ – RDD 用に 作成されたアクションに基づきます。ジョブは 1 つ以上のステージ で構成されます。
- ステージ – RDD 用に作成されたシャッフル に基づきます。ステージは 1 つ以上のタスクで構成されます。シャッフルは、データを RDD パーティション間で異なる方法でグループ化するように再分散するための Spark のメカニズムです。など、特定の変換にはシャッフル`join()`が必要です。シャッフルについては、シャッフルの最適化のチューニングの実践で詳しく説明します。
- タスク – タスクは、Spark によってスケジュールされる処理の最小単位です。タスクは RDD パーティションごとに作成され、タスクの数はステージ内の同時実行の最大数です。

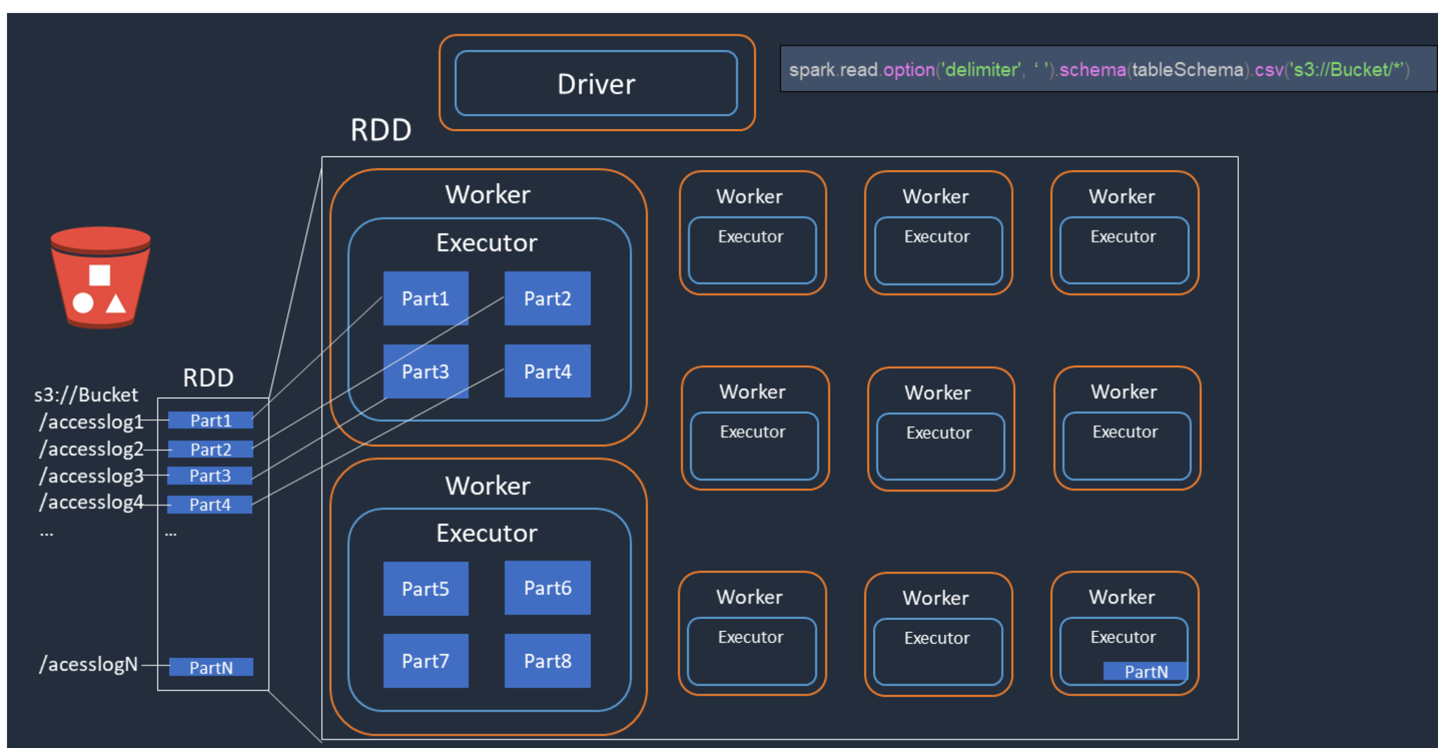


> **Note**
>
> タスクは並列処理を最適化する際に考慮すべき最も重要なものです。RDD の数に応じてスケーリングされるタスクの数

## 並列処理

Spark は、データをロードおよび変換するためのタスクを並列化します。

Amazon S3 でアクセスログファイル ( という名前`accesslog1 ... accesslogN`) の分散処理を実行する例を考えてみましょう。次の図は、分散処理フローを示しています。

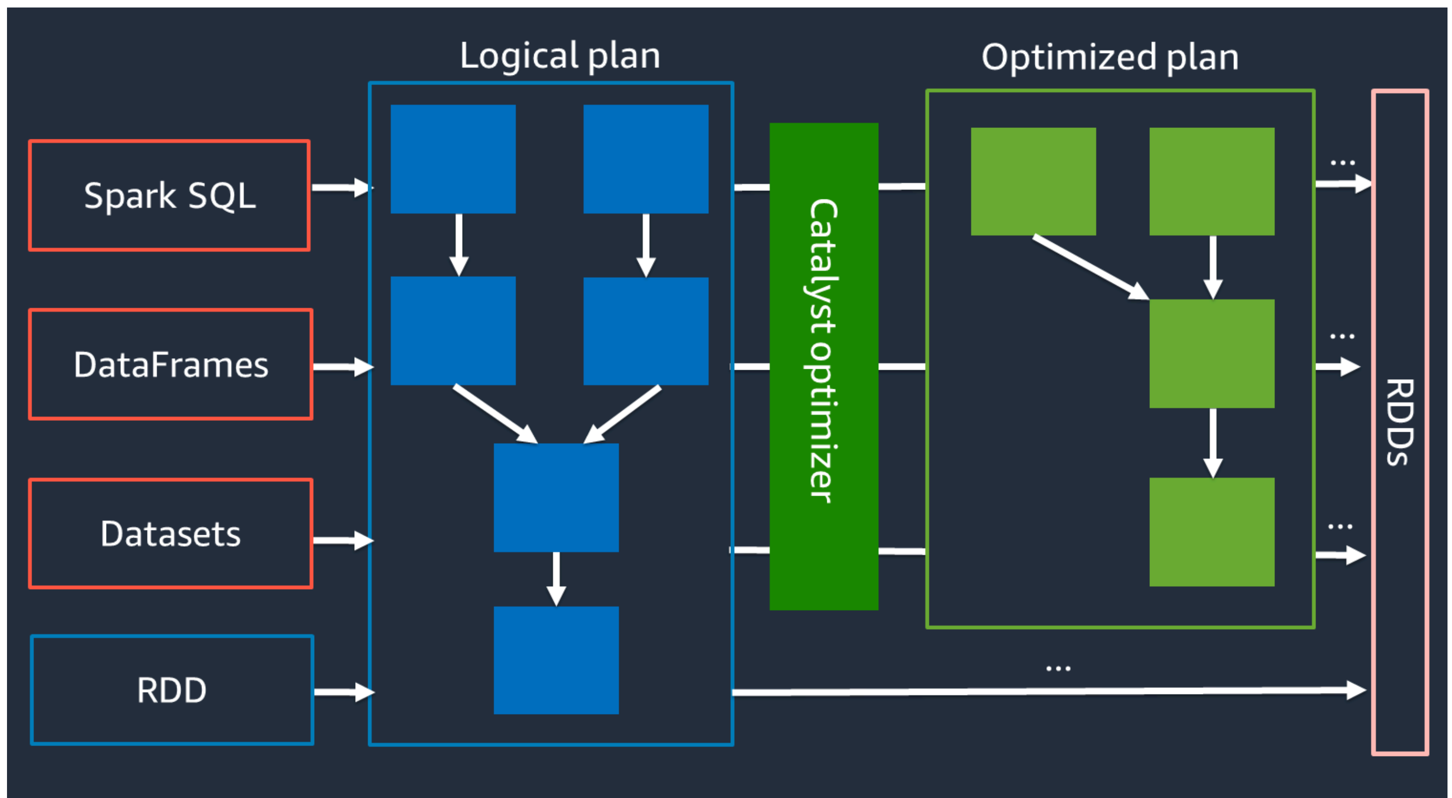



1. Spark ドライバーは、多くの Spark エグゼキューターに分散処理の実行計画を作成します。
2. Spark ドライバーは、実行プランに基づいて各エグゼキューターにタスクを割り当てます。デフォルトでは、Spark ドライバーは S3 オブジェクト () ごとに RDD パーティション (それぞれ Spark タスクに対応) を作成します`Part1 ... N`。次に、Spark ドライバーは各エグゼキューターにタスクを割り当てます。
3. 各 Spark タスクは、割り当てられた S3 オブジェクトをダウンロードし、RDD パーティションのメモリに保存します。このようにして、複数の Spark エグゼキューターが割り当てられたタスクを並行してダウンロードして処理します。

パーティションの初期数と最適化の詳細については、「タスクの並列化」セクションを参照してください。

## カタリストオプティマイザ

内部的には、Spark は Catalyst オプティマイザと呼ばれるエンジンを使用して実行プランを最適化します。Catalyst にはクエリオプティマイザがあり、次の図に示すように、Spark SQL 、、データセット などの高レベルの Spark APIs を実行するときに使用できます。 DataFrame




Catalyst オプティマイザは RDD API と直接連携しないため、高レベル APIs は一般的に低レベル RDD API よりも高速です。複雑な結合の場合、Catalyst オプティマイザはジョブ実行プランを最適化することでパフォーマンスを大幅に向上させることができます。Spark ジョブの最適化された計画は、Spark UI の SQL タブで確認できます。

アダプティブクエリの実行

Catalyst オプティマイザは、アダプティブクエリ実行 と呼ばれるプロセスを通じてランタイム最適化を実行します。アダプティブクエリ実行は、ランタイム統計を使用して、ジョブの実行中にクエリの実行プランを再最適化します。アダプティブクエリの実行には、次のセクションで説明するように、シャッフル後のパーティションの結合、ソートマージ結合のブロードキャスト結合への変換、ス

キュー結合の最適化など、パフォーマンスの課題に対するいくつかのソリューションが用意されています。

アダプティブクエリ実行は AWS Glue 3.0 以降で利用可能で、 AWS Glue 4.0 (Spark 3.3.0) 以降ではデフォルトで有効になっています。コードspark.conf.set("spark.sql.adaptive.enabled", "true")で を使用すると、アダプティブクエリの実行をオンまたはオフにできます。

シャッフル後のパーティションの結合

この機能は、map出力統計に基づいて、シャッフルするたびに RDD パーティション (合体) を減らします。これにより、クエリを実行する際のシャッフルパーティション番号のチューニングが簡素化されます。データセットに合わせてシャッフルパーティション番号を設定する必要はありません。Spark は、シャッフルパーティションの初期数が十分多い場合、実行時に適切なシャッフルパーティション番号を選択できます。

シャッフル後のパーティションの結合は、 spark.sql.adaptive.enabledと の両方spark.sql.adaptive.coalescePartitions.enabledが true に設定されている場合に有効になります。詳細については、「Apache Spark ドキュメント」を参照してください。

ソートマージ結合をブロードキャスト結合に変換する

この機能は、実質的に異なるサイズの 2 つのデータセットを結合する場合を認識し、その情報に基づいてより効率的な結合アルゴリズムを採用します。詳細については、「Apache Spark ドキュメント」を参照してください。結合戦略については、シャッフルの最適化セクションで説明します。

スキュー結合の最適化

データスキューは、Spark ジョブの最も一般的なボトルネックの 1 つです。これは、データが特定の RDD パーティション (ひいては特定のタスク) に偏り、アプリケーションの全体的な処理時間が遅れる状況を示しています。これにより、結合オペレーションのパフォーマンスがダウングレードされることがよくあります。スキュー結合最適化機能は、スキューされたタスクをほぼ同じサイズのタスクに分割 (および必要に応じてレプリケート) することで、ソートマージ結合のスキューを動的に処理します。

この機能は、 が true spark.sql.adaptive.skewJoin.enabled に設定されている場合に有効になります。詳細については、「Apache Spark ドキュメント」を参照してください。データスキューについては、シャッフルの最適化セクションで詳しく説明します。

# Spark UI を使用してパフォーマンスの問題を調査する

ベストプラクティスを適用して AWS Glue ジョブのパフォーマンスを調整する前に、パフォーマンスをプロファイリングし、ボトルネックを特定することを強くお勧めします。これにより、適切なことに集中できます。

迅速な分析のために、Amazon CloudWatch メトリクスはジョブメトリクスの基本ビューを提供します。Spark UI では、パフォーマンスチューニングをより詳細に確認できます。で Spark UI を使用するには AWS Glue、AWS Glue ジョブ で Spark UI を有効にする必要があります。Spark UI に慣れたら、Spark ジョブのパフォーマンスを調整する戦略に従って、検出結果に基づいてボトルネックの影響を特定して軽減します。

## Spark UI を使用してボトルネックを特定する

Spark UI を開くと、Spark アプリケーションがテーブルに一覧表示されます。デフォルトでは、AWS Glue ジョブのアプリケーション名は です`nativespark-<Job Name>-<Job Run ID>`。ジョブ実行 ID に基づいてターゲット Spark アプリを選択し、ジョブタブを開きます。ストリーミングジョブの実行など、不完全なジョブの実行は、「不完全なアプリケーションを表示」に一覧表示されます。

ジョブタブには、Spark アプリケーション内のすべてのジョブの概要が表示されます。ステージまたはタスクの失敗を判断するには、タスクの合計数を確認します。ボトルネックを見つけるには、期間を選択してソートします。説明列に表示されるリンクを選択して、長時間実行されるジョブの詳細までドリルダウンします。

**Spark Jobs** (?)

**User:** spark
**Total Uptime:** 7.7 min
**Scheduling Mode:** FIFO
**Completed Jobs:** 7

▸ Event Timeline

▾ Completed Jobs (7)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

| Job Id | Description | Submitted | Duration ▾ | Stages: Succeeded/Total | Tasks (for all stages): Succeeded/Total |
|---|---|---|---|---|---|
| 3 | parquet at NativeMethodAccessorImpl.java:0<br>parquet at NativeMethodAccessorImpl.java:0 | 2023/03/30 06:49:02 | 6.5 min | 1/1 (1 skipped) | 5/5 (799 skipped) |
| 0 | showString at NativeMethodAccessorImpl.java:0<br>showString at NativeMethodAccessorImpl.java:0 | 2023/03/30 06:48:15 | 29 s | 1/1 | 799/799 |
| 2 | parquet at NativeMethodAccessorImpl.java:0<br>parquet at NativeMethodAccessorImpl.java:0 | 2023/03/30 06:48:48 | 14 s | 1/1 | 799/799 |

ジョブの詳細ページにはステージが一覧表示されます。このページでは、期間、成功したタスクの数と合計タスク数、入力と出力の数、シャッフル読み取りとシャッフル書き込みの量などの全体的なインサイトを確認できます。

**Details for Job 3**

**Status:** SUCCEEDED
**Submitted:** 2023/03/30 06:49:02
**Duration:** 6.5 min
**Associated SQL Query:** 2
**Completed Stages:** 1
**Skipped Stages:** 1

- Event Timeline
- DAG Visualization

**Completed Stages (1)**

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

| Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write |
|---|---|---|---|---|---|---|---|---|
| 5 | parquet at NativeMethodAccessorImpl.java:0 +details | 2023/03/30 06:49:02 | 6.5 min | 5/5 | | 10.2 GiB | 11.9 GiB | |

Executor タブには、Spark クラスターの容量が詳細に表示されます。コアの合計数を確認できます。次のスクリーンショットに示すクラスターには、合計 316 個のアクティブコアと 512 個のコアが含まれています。デフォルトでは、各コアは 1 つの Spark タスクを同時に処理できます。

**Executors**

Show Additional Metrics

**Summary**

| | 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 |

ジョブの詳細ページ5/5に表示される値に基づいて、ステージ 5 が最も長いステージですが、512 のうち 5 つのコアしか使用しません。このステージの並列処理は非常に低いですが、かなりの時間がかかるため、ボトルネックとして識別できます。パフォーマンスを向上させるには、その理由を理解する必要があります。一般的なパフォーマンスのボトルネックの影響を認識して軽減する方法の詳細については、「Spark ジョブのパフォーマンスを調整するための戦略」を参照してください。

# Spark ジョブのパフォーマンスを調整するための戦略

パラメータのチューニングを準備する際は、次のベストプラクティスを使用します。

- 問題の特定を始める前に、パフォーマンス目標を決定します。
- パラメータのチューニングを変更する前に、メトリクスを使用して問題を特定します。

ジョブのチューニングで一貫した結果を得るには、チューニング作業のベースライン戦略を立てます。

## パフォーマンスチューニングのベースライン戦略

通常、パフォーマンスチューニングは次のワークフローで実施します。

1. パフォーマンス目標を決定します。
2. メトリクスを測定します。
3. ボトルネックを特定します。
4. ボトルネックの影響を軽減します。
5. 想定目標を達成するまで、ステップ 2～4 を繰り返します。

まず、パフォーマンス目標を決定します。例えば、目標の 1 つは、3 時間以内に AWS Glue ジョブの実行を完了することです。目標を定義したら、ジョブのパフォーマンスメトリクスを測定します。目標を達成するためのメトリクスとボトルネックの傾向を特定します。特に、トラブルシューティング、デバッグ、パフォーマンスチューニングでは、ボトルネックを特定することが最も重要です。Spark アプリケーションの実行中に、Spark は各タスクのステータスと統計を Spark イベントログに記録します。

では AWS Glue、Spark 履歴サーバーによって提供される Spark Web UI を通じて Spark メトリクスを表示できます。 AWS Glue for Spark ジョブはAmazon S3 で指定した場所に Spark イベントログを送信できます。 AWS Glue また、Amazon EC2インスタンスまたはローカルコンピュータで Spark 履歴サーバーを起動するためのAWS CloudFormation テンプレートと Dockerfile の例も提供されているため、Spark UI をイベントログとともに使用できます。

パフォーマンス目標を決定し、それらの目標を評価するためのメトリクスを特定したら、次のセクションの戦略を使用してボトルネックの特定と修正を開始できます。

# Spark ジョブパフォーマンスのチューニング方法

Spark ジョブのパフォーマンスチューニング AWS Glue には、次の戦略を使用できます。

- AWS Glue リソース：
  - クラスター容量のスケーリング
  - AWS Glue 最新バージョンを使用する
- Spark アプリケーション：
  - データスキャンの量を減らす
  - タスクの並列化
  - シャッフルの最適化
  - 計画オーバーヘッドを最小限に抑える
  - ユーザー定義関数の最適化

これらの戦略を使用する前に、Spark ジョブのメトリクスと設定にアクセスできる必要があります。この情報は、 AWS Glue ドキュメントに記載されています。

AWS Glue リソースの観点からは、 AWS Glue ワーカーを追加し、 AWS Glue 最新バージョンを使用することで、パフォーマンスを向上させることができます。

Apache Spark アプリケーションの観点からは、パフォーマンスを向上させるいくつかの戦略にアクセスできます。不要なデータが Spark クラスターにロードされた場合は、削除してロードされたデータの量を減らすことができます。Spark クラスターリソースの使用率が低く、データ I/O が少ない場合は、並列化するタスクを特定できます。また、結合などの大量のデータ転送操作にかなりの時間がかかる場合は、最適化することもできます。ジョブクエリプランを最適化したり、個々の Spark タスクの計算の複雑さを軽減したりすることもできます。

これらの戦略を効率的に適用するには、メトリクスを参照して適用可能なタイミングを特定する必要があります。詳細については、以下の各セクションを参照してください。これらの手法は、パフォーマンスの調整だけでなく、 out-of-memory（OOM) エラーなどの一般的な問題の解決にも役立ちます。

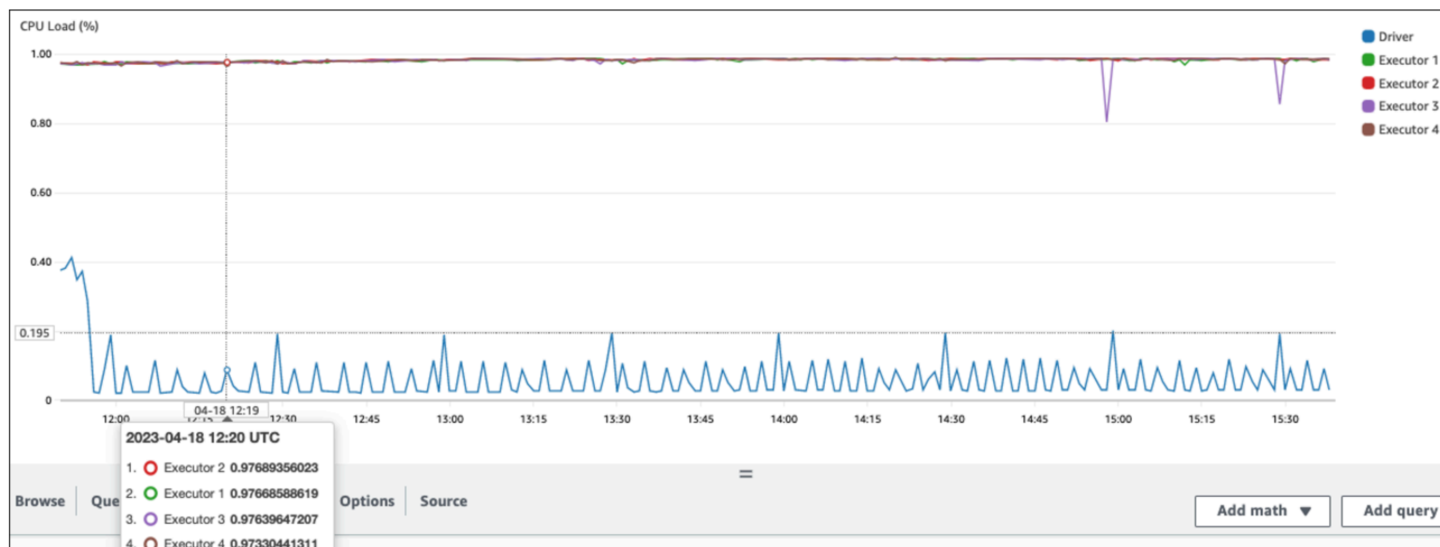
# クラスター容量のスケーリング

ジョブに時間がかかりすぎているが、エグゼキューターが十分なリソースを消費していて、Spark が使用可能なコアに対して大量のタスクを作成している場合は、クラスター容量のスケーリングを検討してください。これが適切かどうかを評価するには、次のメトリクスを使用します。

## CloudWatch メトリクス

- CPU ロードとメモリの使用率をチェックして、エグゼキューターが十分なリソースを消費しているかどうかを確認します。
- ジョブが実行された時間をチェックして、処理時間がパフォーマンス目標を達成できないほど長すぎるかどうかを評価します。

次の例では、4 つのエグゼキューターが 97% を超えるCPU負荷で実行されていますが、約 3 時間後に処理が完了していません。




> **Note**
>
> CPU 負荷が低い場合、クラスター容量のスケーリングによるメリットは得られない可能性があります。

# Spark UI

ジョブタブまたはステージタブで、各ジョブまたはステージのタスク数を確認できます。次の例では、Spark が`58100`タスクを作成しています。

**Stages for All Jobs**

Completed Stages: 1

Completed Stages (1)

| Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Input |
|---|---|---|---|---|---|
| 0 | count at DynamicFrame.scala:1414 +details | 2023/04/18 10:59:10 | 4.8 h | 58100/58100 | 28.4 GB |

エグゼキュタータブには、エグゼキューターとタスクの総数が表示されます。次のスクリーンショットでは、各 Spark エグゼキューターには 4 つのコアがあり、4 つのタスクを同時に実行できます。

**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 |

この例では、Spark タスクの数 (`58100`) は、エグゼキューターが同時に処理できる 16 個のタスク (4 個のエグゼキューター x 4 コア) よりもはるかに大きくなります。

これらの症状が発生した場合は、クラスターのスケーリングを検討してください。次のオプションを使用して、クラスター容量をスケーリングできます。

- Enable AWS Glue Auto Scaling – Auto Scaling は、 AWS Glue バージョン 3.0 以降の AWS Glue 抽出、変換、ロード (ETL) およびストリーミングジョブで使用できます。 は、各ステージのパーティション数またはジョブ実行時にマイクロバッチが生成される速度に応じて、自動的にワーカー AWS Glue をクラスターに追加および削除します。

  Auto Scaling が有効になっていてもワーカー数が増加しない状況に気付いた場合は、ワーカーを手動で追加することを検討してください。ただし、1 つのステージを手動でスケーリングすると、後続のステージで多くのワーカーがアイドル状態になり、パフォーマンスの向上にはより多くのコストがかかる可能性があることに注意してください。

Auto Scaling を有効にすると、エグゼキューターメトリクスに CloudWatch エグゼキューターの数が表示されます。Spark アプリケーションのエグゼキューターの需要をモニタリングするには、次のメトリクスを使用します。

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

メトリクスの詳細については、「Amazon メトリクス AWS Glue を使用したモニタリング CloudWatch 」を参照してください。

- スケールアウト: ワーカー数 AWS Glue を増やす – ワーカー数 AWS Glue を手動で増やすことができます。アイドル状態のワーカーが観察されるまでのみ、ワーカーを追加します。この時点で、ワーカーを追加すると、結果を改善せずにコストが増加します。詳細については、「タスクの並列化」を参照してください。
- スケールアップ: より大きなワーカータイプを使用する – AWS Glue ワーカーのインスタンスタイプを手動で変更して、より多くのコア、メモリ、ストレージを持つワーカーを使用できます。ワーカータイプが大きいほど、メモリを大量に消費するデータ変換、偏った集約、ペタバイト単位のデータを含むエンティティ検出チェックなど、大量のデータ統合ジョブを垂直方向にスケーリングして実行できます。

  スケールアップは、ジョブクエリプランが非常に大きいため、Spark ドライバーでより大きな容量が必要な場合にも役立ちます。ワーカータイプとパフォーマンスの詳細については、 AWS 「ビッグデータブログ」の記事「新しい大きなワーカータイプ G.4X と G.8X を使用して Apache Spark ジョブのスケーリング AWS Glue」を参照してください。

  ワーカー数を大きくすると、必要なワーカーの総数も減り、参加などの集中的な操作でシャッフルを減らすことでパフォーマンスが向上します。

## AWS Glue 最新バージョンを使用する

AWS Glue 最新バージョンを使用することをお勧めします。ジョブのパフォーマンスを自動的に向上させる可能性のある最適化とアップグレードが各バージョンに組み込まれています。例えば、 AWS Glue 4.0 には次の新機能があります。

- 新しい最適化された Apache Spark 3.3.0 ランタイム – AWS Glue 4.0 は Apache Spark 3.3.0 ランタイムに基づいて構築され、オープンソースの Spark と同等のパフォーマンスの向上を実現します。Spark 3.3.0 ランタイムは、Spark 2.x の多くのイノベーションに基づいています。

- 拡張 Amazon Redshift コネクタ – AWS Glue 4.0 以降のバージョンでは、Apache Spark 用の Amazon Redshift 統合が提供されます。統合は既存のオープンソースコネクタ上に構築され、パフォーマンスとセキュリティを強化します。統合により、アプリケーションのパフォーマンスが最大 10 倍速くなります。詳細については、Amazon Redshift と Apache Spark の統合に関するブログ記事を参照してください。
- バージョン 3.0 以降のバージョンではSIMDCSV、 および JSON データを使用したベクトル化された読み取りの ベースの実行により、行ベースのリーダーと比較して全体的なジョブパフォーマンスを大幅に高速化できる最適化されたリーダーが追加されます。 AWS Glue CSV データの詳細については、「ベクトル化されたSIMDCSVリーダーによる読み取りパフォーマンスの最適化」を参照してください。JSON データの詳細については、「Apache Arrow 列形式 でベクトル化されたSIMDJSONリーダーを使用する」を参照してください。

各 AWS Glue バージョンには、コネクタ、ドライバー、ライブラリの更新など、多くの でこの種のアップグレードが含まれます。詳細については、「 AWS Glue バージョン」およびAWS Glue 「ジョブを AWS Glue バージョン 4.0 に移行する」を参照してください。

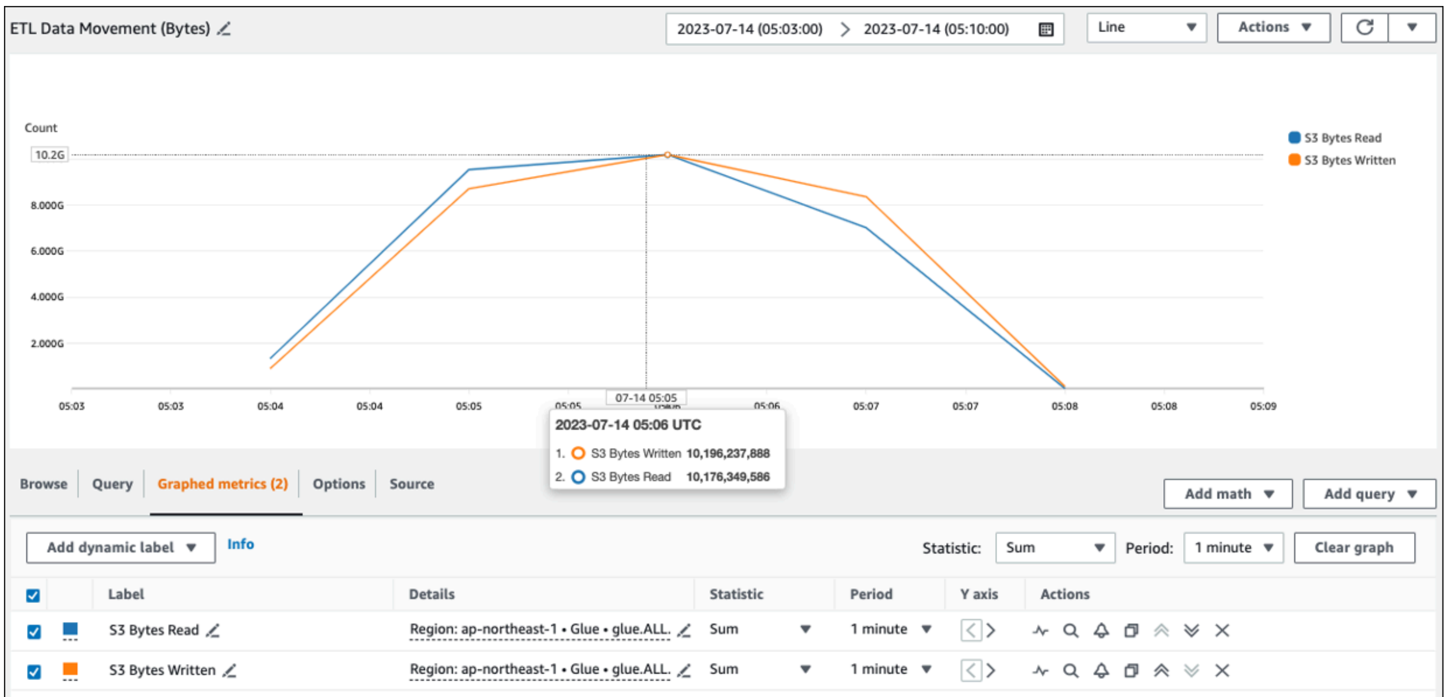
# データスキャンの量を減らす

まず、必要なデータのみをロードすることを検討してください。各データソースの Spark クラスターにロードされるデータの量を減らすだけで、パフォーマンスを向上させることができます。このアプローチが適切かどうかを評価するには、次のメトリクスを使用します。

Spark UI セクションで説明されているように、CloudWatchメトリクスの Amazon S3 からの読み取りバイト数と Spark UI の詳細を確認できます。

## CloudWatch メトリクス

Amazon S3 からのおおよその読み込みサイズは、ETLデータ移動 (バイト ) で確認できます。このメトリクスは、前のレポート以降のすべてのエグゼキューターによって Amazon S3 から読み取られたバイト数を示します。これを使用して Amazon S3 からのETLデータ移動をモニタリングし、読み取りと外部データソースからの取り込みレートを比較できます。




S3 バイト読み取りデータポイントが予想よりも大きい場合は、次の解決策を検討してください。

## Spark UI

for Spark UI AWS Glue のステージタブには、入力サイズと出力サイズが表示されます。次の例では、ステージ 2 は 47.4 GiB の入力と 47.7 GiB の出力を読み取り、ステージ 5 は 61.2 MiB の入力と 56.6 MiB の出力を読み取ります。

### Stages for All Jobs

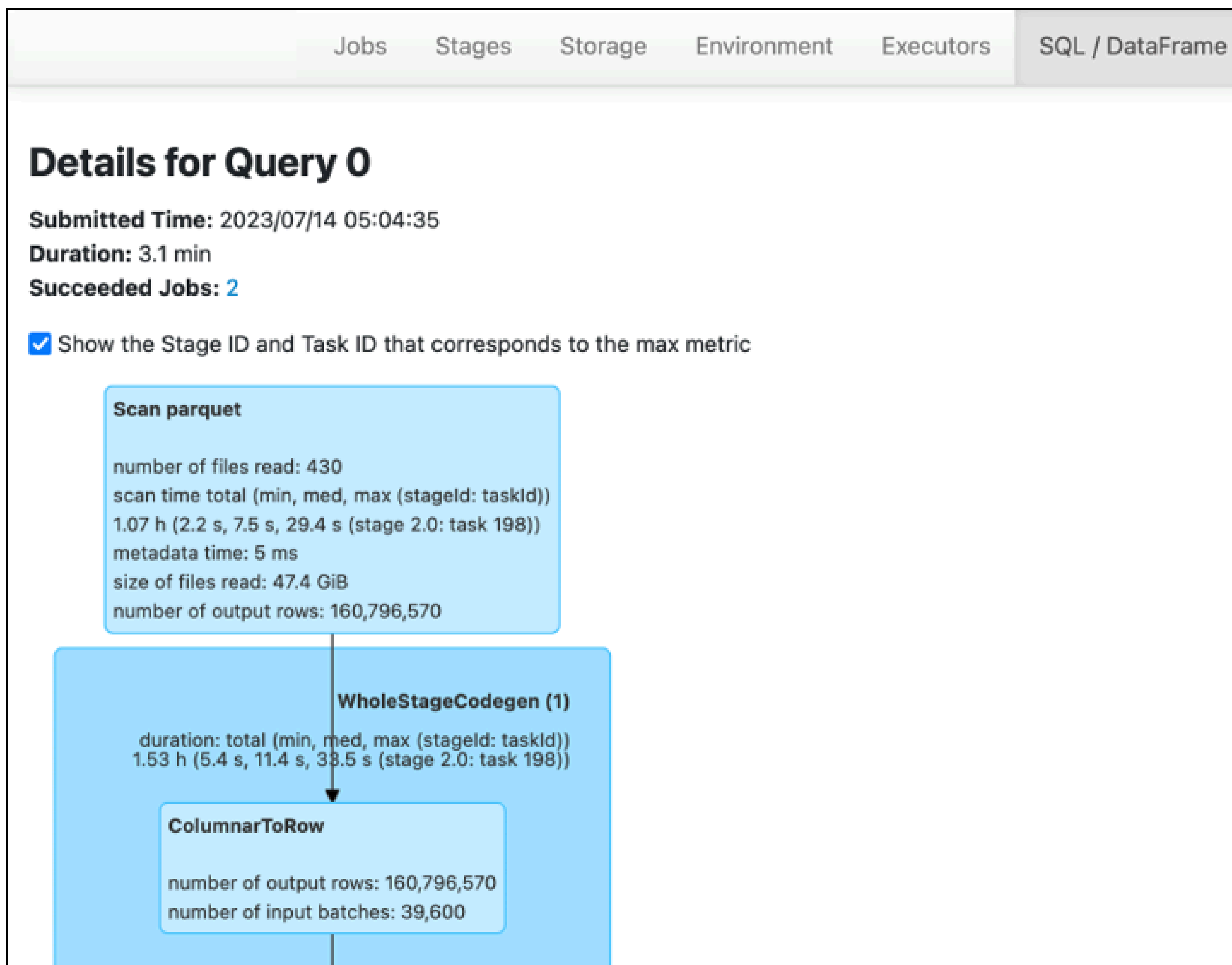
**Completed Stages:** 6

**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 | | |

AWS Glue ジョブで Spark SQLまたは DataFrame アプローチを使用すると、SQL/D ataFrame タブにこれらのステージに関する統計情報が表示されます。この場合、ステージ 2 には、読み取りファイル数: 430、読み取りファイルサイズ: 47.4 GiB、出力行数: 160,796,570 が表示されます。




読み込むデータと使用しているデータのサイズに大きな違いがあることがわかった場合は、次のソリューションを試してください。

## Amazon S3

Amazon S3 から読み取るときにジョブにロードされるデータの量を減らすには、データセット のファイルサイズ、圧縮、ファイル形式、ファイルレイアウト (パーティション) を考慮してください。Spark ジョブ AWS Glue の は、ETL多くの場合 raw データの に使用されますが、効率的な分散処理のためには、データソース形式の機能を検査する必要があります。

- ファイルサイズ – 入力と出力のファイルサイズは、中程度の範囲 (128 MB など) にしておくことをお勧めします。ファイルが小さすぎたり、大きすぎたりすると、問題が発生する可能性があります。

  小さなファイルが多数あると、次の問題が発生します。

  - 多くのオブジェクト (同じ量のデータを保存する少数のオブジェクトと比較してHead) に対してリクエストを行うために必要なオーバーヘッド (List、Get、または など) により、Amazon S3 でのネットワーク I/O 負荷が高くなります。
  - Spark ドライバーの負荷の高い I/O と処理負荷。これにより、多くのパーティションとタスクが生成され、過剰な並列処理が発生します。

  一方、ファイルタイプが分割可能でなく (gzip など ) 、ファイルが大きすぎる場合、Spark アプリケーションは 1 つのタスクでファイル全体の読み取りが完了するまで待機する必要があります。

  小さなファイルごとに Apache Spark タスクを作成するときに発生する過剰な並列処理を減らすには、 のファイルグループ DynamicFramesを使用します。このアプローチにより、Spark ドライバーからOOM例外が発生する可能性が低くなります。ファイルグループを設定するには、 groupFilesおよび groupSizeパラメータを設定します。次のコード例では、 AWS Glue DynamicFrame APIこれらのパラメータを持つ ETLスクリプトで を使用します。

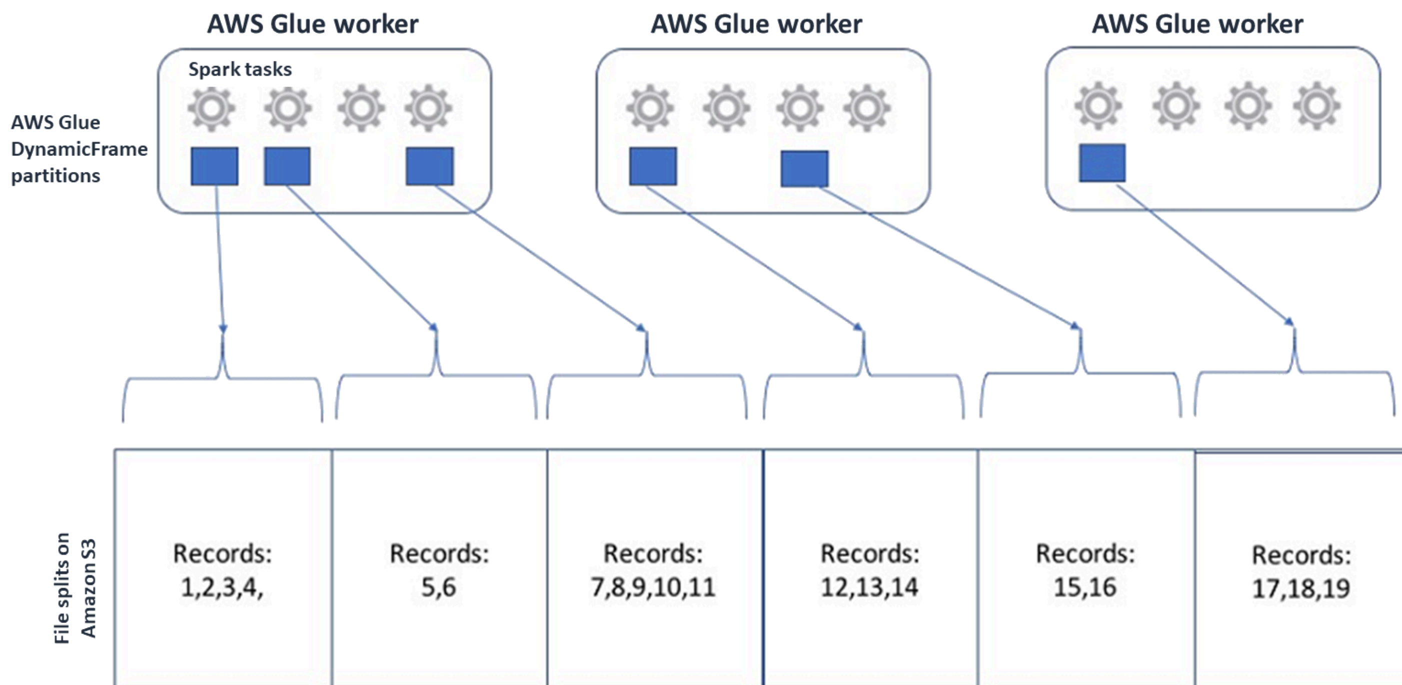
```
dyf = glueContext.create_dynamic_frame_from_options("s3",
    {'paths': ["s3://input-s3-path/"],
    'recurse':True,
    'groupFiles': 'inPartition',
    'groupSize': '1048576'},
    format="json")
```

- 圧縮 — S3 オブジェクトが数百メガバイト内にある場合は、圧縮することを検討してください。圧縮形式にはさまざまな種類があり、大まかに 2 つのタイプに分類できます。

  - gzip などの分割不可能な 圧縮形式では、ファイル全体が 1 人のワーカーによって解凍される必要があります。
  - bzip2 や LZO (インデックス付き) などの分割可能な圧縮形式を使用すると、ファイルを部分的に解凍でき、並列化できます。

  Spark (およびその他の一般的な分散処理エンジン) では、ソースデータファイルをエンジンが並行して処理できるチャンクに分割します。これらの単位は、分割 と呼ばれることがよくあります。データが分割可能な形式になると、最適化された AWS Glue リーダーは、特定のブロックのみを

取得するRangeオプションGetObjectAPIを に提供することで、S3 オブジェクトから分割を取得できます。次の図を参照して、これが実際にどのように機能するかを確認してください。




圧縮されたデータは、ファイルが最適なサイズであるか、ファイルが分割可能である限り、アプリケーションを大幅に高速化できます。データサイズが小さいほど、Amazon S3 からスキャンされるデータと、Amazon S3 から Spark クラスターへのネットワークトラフィックが減少します。一方、データを圧縮および解凍するには、さらに多くの CPUが必要です。必要なコンピューティングの量は、圧縮アルゴリズムの圧縮率でスケーリングされます。分割可能な圧縮形式を選択するときは、このトレードオフを考慮してください。

> **Note**
>
> gzip ファイルは一般に分割できませんが、gzip を使用して個々の Parquet ブロックを圧縮し、それらのブロックを並列化できます。

- ファイル形式 – 列指向形式を使用します。Apache Parquet と Apache ORCは一般的な列指向データ形式です。列ベースの圧縮、データ型に基づく各列のエンコードと圧縮を採用することで、データを効率的に解析してORC保存します。Parquet エンコーディングの詳細については、「Parquet エンコーディング定義」を参照してください。Parquet ファイルも分割可能です。

  Columnar は、値を列ごとにグループ化し、ブロックにまとめて保存します。列指向形式を使用する場合、使用する予定のない列に対応するデータのブロックをスキップできます。Spark アプリ

ケーションは、必要な列のみを取得できます。一般的に、圧縮率が高いか、データブロックをスキップするとAmazon S3から読み取るバイト数が少なくなり、パフォーマンスが向上します。どちらの形式も、I/O を減らすための以下のプッシュダウンアプローチをサポートしています。

- 射影プッシュダウン — 射影プッシュダウンは、アプリケーションで指定された列のみを取得する手法です。次の例に示すように、Spark アプリケーションで列を指定します。
  - DataFrame 例: `df.select("star_rating")`
  - Spark SQLの例: `spark.sql("select start_rating from <table>")`
- 述語のプッシュダウン — 述語のプッシュダウンは、 `WHERE`および `GROUP BY`句を効率的に処理するための手法です。どちらの形式にも、列の値を表すデータのブロックがあります。各ブロックは、最大値や最小値など、ブロックの統計を保持します。Spark は、これらの統計を使用して、アプリケーションで使用されるフィルター値に応じてブロックを読み取るかスキップするかを決定できます。この機能を使用するには、次の例に示すように、条件にフィルターを追加します。
  - DataFrame 例: `df.select("star_rating").filter("star_rating < 2")`
  - Spark SQLの例: `spark.sql("select * from <table> where star_rating < 2")`

- ファイルレイアウト – S3 データを、データの使用方法に基づいて異なるパスのオブジェクトに保存することで、関連するデータを効率的に取得できます。詳細については、Amazon S3ドキュメント」の「プレフィックスを使用してオブジェクトを整理する」を参照してください。 AWS Glue は、キーと値を 形式で Amazon S3 プレフィックスに保存し`key=value`、Amazon S3 パスでデータをパーティション化することをサポートしています。データをパーティション化することで、各ダウンストリームの分析アプリケーションでスキャンされるデータ量を制限し、パフォーマンスを向上させ、コストを削減できます。詳細については、「 でのETL出力のパーティションの管理 AWS Glue」を参照してください。

  パーティション分割はテーブルをさまざまな部分に分割し、次の例に示すように、年 、月 、日 などの列値に基づいて関連するデータをグループ化されたファイルに保持します。

```
# 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
...
```

データセットのパーティションは、 のテーブルでモデル化することで定義できます AWS Glue Data Catalog。その後、次のようにパーティションプルーニングを使用してデータスキャンの量を制限できます。

- には AWS Glue DynamicFrame、 `push_down_predicate` (または `catalogPartitionPredicate`) を設定します。

```
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'",
)
```

- Spark の場合 DataFrame、パーティションをプルーニングする固定パスを設定します。

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

- Spark ではSQL、データカタログからパーティションをプルーニングする where 句を設定できます。

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

- を使用してデータを書き込むときに日付でパーティション化するには AWS Glue、次のように列partitionKeysの日付情報 DataFrame を使用して DynamicFrame または partitionBy() を に設定します。
  - 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>/')
```

これにより、出力データのコンシューマーのパフォーマンスを向上させることができます。

入力データセットを作成するパイプラインを変更するためのアクセス権がない場合、パーティショニングはオプションではありません。代わりに、glob パターンを使用して不要な S3 パスを除外できます。で読み取るときに除外を設定します DynamicFrame。例えば、次のコードでは、2023 年の 01～09 か月の日数を除外しています。

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

データカタログのテーブルプロパティで除外を設定することもできます。

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

- Amazon S3 パーティションが多すぎる – 数千の値を含む ID 列など、幅広い値を含む列で Amazon S3 データをパーティション化することは避けてください。これにより、可能なパーティションの数はパーティション分割したすべてのフィールドの積であるため、バケット内のパーティションの数が大幅に増加する可能性があります。パーティションが多すぎると、以下が発生する可能性があります。
  - Data Catalog からパーティションメタデータを取得する際のレイテンシーの増加
  - Amazon S3 APIリクエスト (、`List`、`Get`および `Head`) をさらに必要とする小さなファイルの数の増加

例えば、 `partitionBy`または で日付タイプを設定すると`partitionKeys`、 などの日付レベルのパーティショニング`yyyy/mm/dd`は多くのユースケースに適しています。ただし、 は多数のパーティションを生成するため、パフォーマンス全体に悪影響を与える`yyyy/mm/dd/<ID>`可能性があります。

一方、リアルタイム処理アプリケーションなどの一部のユースケースでは、 などの多くのパーティションが必要です`yyyy/mm/dd/hh`。ユースケースでかなりのパーティションが必要な場合は、AWS Glue パーティションインデックスを使用して Data Catalog からパーティションメタデータを取得する際のレイテンシーを短縮することを検討してください。

## データベースと JDBC

データベースから情報を取得するときにデータスキャンを減らすには、SQLクエリで述where語 (または 句) を指定します。SQL インターフェイスを提供しないデータベースは、クエリまたはフィルタリングのための独自のメカニズムを提供します。

Java Database Connectivity (JDBC) 接続を使用する場合は、次のパラメータの where句を含む Select クエリを指定します。

- には DynamicFrame、 sampleQueryオプションを使用します。を使用する場合はcreate_dynamic_frame.from_catalog、引additional_options数を次のように設定します。

```
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"
)
```

  の場合using create_dynamic_frame.from_options、引connection_options数を次のように設定します。

```
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
    }
```

```
)
```

- には DataFrame、 クエリオプションを使用します。

```
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()
```

- Amazon Redshift では、 AWS Glue 4.0 以降を使用して Amazon Redshift Spark コネクタ のプッシュダウンサポートを活用します。

```
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"}
)
```

- 他のデータベースについては、そのデータベースのドキュメントを参照してください。

## AWS Glue オプション

- すべての連続ジョブ実行の完全なスキャンを回避し、最後のジョブ実行時に存在しなかったデータのみを処理するために、ジョブブックマーク を有効にします。
- 処理する入力データの量を制限するには、ジョブブックマークを使用して制限付き実行を有効にします。これにより、ジョブ実行ごとにスキャンされるデータの量を減らすことができます。

# タスクの並列化

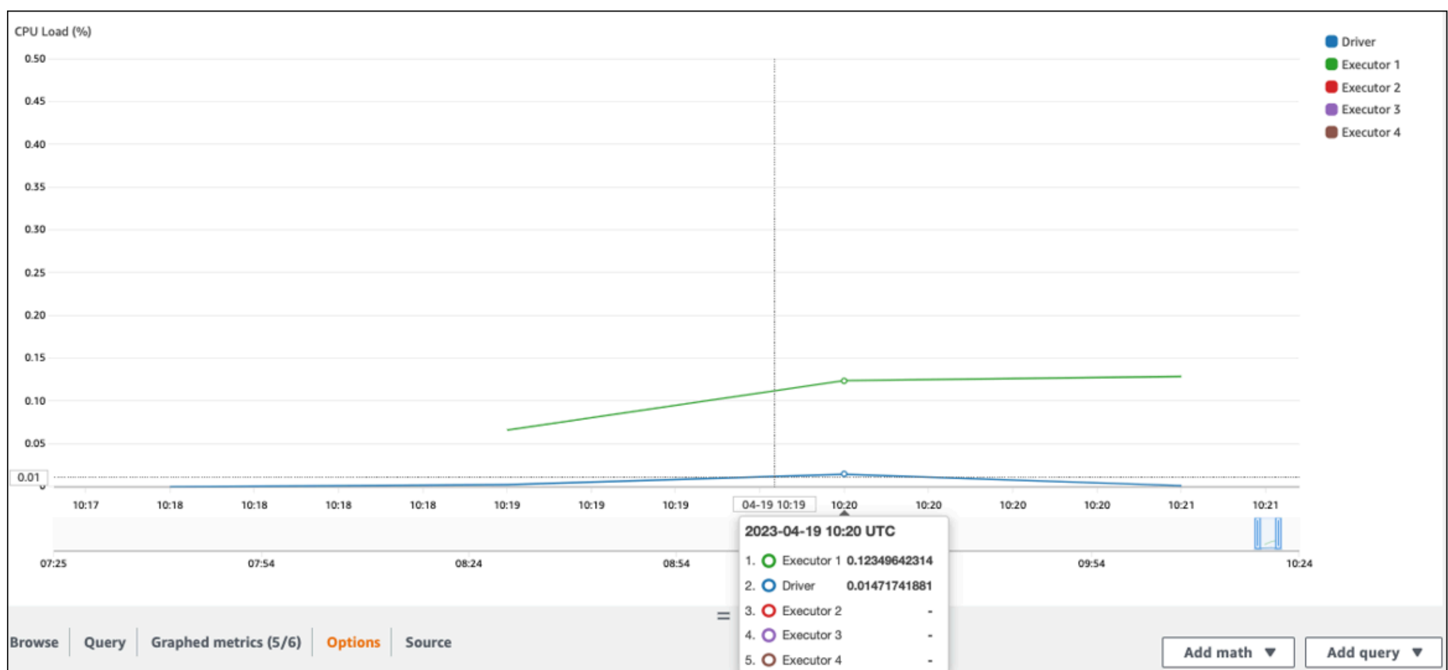
パフォーマンスを最適化するには、データのロードと変換のタスクを並列化することが重要です。Apache Spark の「主要なトピック」で説明したように、並列処理の程度を決定するため、回復力のある分散データセット (RDD) パーティションの数は重要です。Spark が作成する各タスク

は、1:1 ベースでRDDパーティションに対応します。最高のパフォーマンスを実現するには、RDD パーティション数の決定方法とその数の最適化方法を理解する必要があります。

十分な並列処理がない場合、以下の症状はCloudWatchメトリクスと Spark UI に記録されます。

## CloudWatch メトリクス

CPU ロードとメモリの使用率を確認します。一部のエグゼキューターがジョブのフェーズで処理されていない場合は、並列処理を改善するのが適切です。この場合、視覚化された期間中に、エグゼキューター 1 はタスクを実行していましたが、残りのエグゼキューター (2、3、4) は実行しませんでした。これらのエグゼキューターには Spark ドライバーによってタスクが割り当てられていないと推測できます。



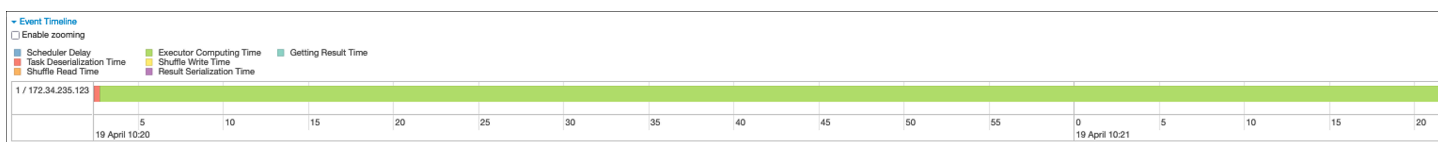

## Spark UI

Spark UI のステージタブで、ステージ内のタスク の数を確認できます。 この場合、Spark は 1 つのタスクのみを実行しています。

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 |

さらに、イベントタイムラインには、Executor 1 が 1 つのタスクを処理することが表示されます。つまり、この段階の作業は 1 つのエグゼキューターで完全に実行され、他のエグゼキューターはアイドル状態でした。




これらの症状が発生した場合は、データソースごとに次のソリューションを試してください。

## Amazon S3 からのデータロードを並列化する

Amazon S3 からのデータロードを並列化するには、まずパーティションのデフォルト数を確認します。その後、パーティションのターゲット数を手動で決定できますが、パーティションが多すぎないように注意してください。

### パーティションのデフォルト数を決定する

Amazon S3 の場合、Spark RDDパーティションの初期数 (それぞれが Spark タスクに対応) は、Amazon S3 データセットの特徴 (形式、圧縮、サイズなど) によって決まります。Amazon S3 に保存されているCSVオブジェクト DataFrame から または Spark を作成する AWS Glue DynamicFrame場合、RDDパーティションの初期数 (`NumPartitions`) を次のようにほぼ計算できます。 Amazon S3

- オブジェクトサイズ <= 64 MB: `NumPartitions = Number of Objects`
- オブジェクトサイズ > 64 MB: `NumPartitions = Total Object Size / 64 MB`
- 分割不可 (gzip): `NumPartitions = Number of Objects`

「データスキャン量を減らす」セクションで説明したように、Spark は大きな S3 オブジェクトを分割して並列処理できます。オブジェクトが分割サイズより大きい場合、Spark はオブジェクトを分割し、分割ごとにRDDパーティション (およびタスク) を作成します。Spark の分割サイズはデータ形式とランタイム環境に基づいていますが、これは妥当な開始近似値です。一部のオブジェクトは gzip などの分割不可能な圧縮形式を使用して圧縮されるため、Spark はそれらを分割できません。

`NumPartitions` 値は、データ形式、圧縮、 AWS Glue バージョン、 AWS Glue ワーカー数、Spark 設定によって異なる場合があります。

例えば、Spark を使用して単一の 10 GB `csv.gz` オブジェクトをロードすると DataFrame、gzip は分割できないため、Spark ドライバーはRDDパーティション (`NumPartitions=1`) を 1 つだけ作成します。これにより、次の図に示すように、特定の Spark エグゼキューターに負荷がかかり、残りのエグゼキューターにタスクが割り当てられません。

Spark Web UI ステージタブでステージの実際のタスク数 (NumPartitions) を確認するか、コードdf.rdd.getNumPartitions()で を実行して並列処理を確認します。

10 GB の gzip ファイルが発生した場合は、そのファイルを生成するシステムが分割可能な形式で生成できるかどうかを確認します。これがオプションでない場合は、ファイルを処理するためにクラスター容量をスケーリングする必要がある場合があります。ロードしたデータに対して変換を効率的に実行するには、再パーティションを使用してクラスター内のワーカーRDD間で を再調整する必要があります。

パーティションのターゲット数を手動で決定する

データのプロパティと Spark による特定の機能の実装によっては、基盤となる作業を引き続き並列化できる場合でも、低いNumPartitions値になる可能性があります。NumPartitions が小さすぎる場合は、 df.repartition(N)を実行してパーティションの数を増やし、処理を複数の Spark エグゼキューターに分散できるようにします。

この場合、 の実行df.repartition(100)は 1 NumPartitionsから 100 に増加し、それぞれに他のエグゼキューターに割り当てることができるタスクを含む 100 個のデータのパーティションが作成されます。

オペレーションは、データ全体を均等にrepartition(N)分割し (10 GB/100 パーティション = 100 MB/パーティション）、特定のパーティションへのデータスキューを回避します。

> **Note**
>
> などのシャッフルオペレーションjoinを実行すると、パーティションの数は spark.sql.shuffle.partitionsまたは の値に応じて動的に増減しますspark.default.parallelism。これにより、Spark エグゼキューター間のデータのより効率的な交換が容易になります。詳細については、Spark ドキュメント を参照してください。

パーティションのターゲット数を決定する際の目標は、プロビジョニングされた AWS Glue ワーカーの使用を最大化することです。 AWS Glue ワーカーの数と Spark タスクの数は、 の数によって関連しますvCPUs。Spark は vCPU コアごとに 1 つのタスクをサポートします。 AWS Glue バージョン 3.0 以降では、次の式を使用してパーティションのターゲット数を計算できます。

```
# 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
```

この例では、各 G.1X ワーカーは Spark エグゼキューター () に 4 つの vCPU コアを提供します`spark.executor.cores = 4`。Spark は vCPU Core ごとに 1 つのタスクをサポートするため、G.1X Spark エグゼキューターは 4 つのタスクを同時に実行できます (`numSlotPerExecutor` )。この数のパーティションは、タスクに同じ時間がかかった場合にクラスターを最大限に活用します。ただし、一部のタスクは他のタスクよりも時間がかかり、アイドル状態のコアが作成されます。この場合、2 または 3 `numPartitions` を乗算してボトルネックタスクを分割し、効率的にスケジュールすることを検討してください。

パーティションが多すぎる

パーティションの数が多すぎると、タスクの数が多すぎます。これにより、管理タスクや Spark エグゼキューター間のデータ交換などの分散処理に関連するオーバーヘッドにより、Spark ドライバーに負荷がかかります。

ジョブ内のパーティション数がターゲットパーティション数よりも大幅に大きい場合は、パーティション数を減らすことを検討してください。次のオプションを使用してパーティションを減らすことができます。

- ファイルサイズが非常に小さい場合は、 を使用します AWS Glue groupFiles。Apache Spark タスクの起動による過剰な並列処理を減らして、各ファイルを処理できます。
- `coalesce(N)` を使用してパーティションをマージします。これは低コストのプロセスです。パーティションの数を減らす場合、 `coalesce(N)`は よりも優先されます。 `repartition(N)`はシャッフルを実行して各パーティションのレコード量を均等に分散`repartition(N)`するためです。これにより、コストと管理オーバーヘッドが増加します。
- Spark 3.x アダプティブクエリ実行を使用します。Apache Spark の「キートピック」セクションで説明したように、アダプティブクエリ実行は、パーティションの数を自動的に結合する関数を提供します。このアプローチは、実行するまでパーティションの数がわからない場合に使用できます。

## からのデータロードを並列化する JDBC

Spark RDDパーティションの数は設定によって決まります。デフォルトでは、SELECTクエリを通じてソースデータセット全体をスキャンするために実行されるタスクは 1 つだけであることに注意してください。

AWS Glue DynamicFrames と Spark はどちらも、複数のタスクにわたる並列JDBCデータロード DataFrames をサポートしています。これは、where述語を使用して 1 つのSELECTクエリを複数のクエリに分割することによって行われます。からの読み取りを並列化するにはJDBC、次のオプションを設定します。

- には AWS Glue DynamicFrame、 hashfield (または hashexpression)と ) を設定しますhashpartition。詳細については、「並列 でのJDBCテーブルからの読み取り」を参照してください。

```
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"
)
```

- Spark の場合は DataFrame、numPartitions、、partitionColumn、lowerBoundおよび を設定しますupperBound。詳細については、JDBC「他のデータベースへ」を参照してください。

```
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/")
```

## ETL コネクタの使用時に DynamoDB からのデータロードを並列化する

Spark RDDパーティションの数は、 `dynamodb.splits`パラメータによって決まります。Amazon DynamoDB からの読み取りを並列化するには、次のオプションを設定します。

- の値を増やします`dynamodb.splits`。
- for Spark の「 の接続タイプとオプションETL AWS Glue 」で説明されている式に従って、 パラメータを最適化します。

## Kinesis Data Streams からのデータロードを並列化する

Spark RDDパーティションの数は、ソース Amazon Kinesis Data Streams データストリーム内のシャードの数によって決まります。データストリームにシャードがわずかしかない場合、Spark タスクはわずかです。これにより、ダウンストリームプロセスでの並列処理が低くなる可能性があります。Kinesis Data Streams からの読み取りを並列化するには、次のオプションを設定します。

- Kinesis Data Streams からデータをロードするときに、シャードの数を増やして並列性を高めます。
- マイクロバッチのロジックが十分に複雑な場合は、不要な列を削除した後、バッチの最初にデータを再パーティション化することを検討してください。

詳細については、AWS Glue 「ストリーミングETLジョブのコストとパフォーマンスを最適化するためのベストプラクティス」を参照してください。

## データロード後にタスクを並列化する

データロード後にタスクを並列化するには、次のオプションを使用してRDDパーティションの数を増やします。

- データを再パーティションして、特にロード自体を並列化できなかった場合は、最初のロードの直後に、より多くのパーティションを生成します。

パーティションの数を指定して DataFrame、 DynamicFrame または `repartition()`で を呼び出します。適切な経験則は、使用可能なコア数の 2～3 倍です。

ただし、パーティション化されたテーブルを記述すると、ファイルが急増する可能性があります (各パーティションは、各テーブルパーティションにファイルを生成する可能性があります ) 。これを回避するには、列ごとに を再パーティション化します DataFrame。これにより、テーブルパーティション列が使用され、書き込み前にデータが整理されます。テーブルパーティションで小さなファイルを取得せずに、より多くのパーティションを指定できます。ただし、一部のパーティション値がほとんどのデータで終了し、タスクの完了が遅延するデータスキューを避けるように注意してください。

- シャッフルがある場合は、`spark.sql.shuffle.partitions`値を増やします。これは、シャッフル時のメモリの問題にも役立ちます。

  2,001 個を超えるシャッフルパーティションがある場合、Spark は圧縮メモリ形式を使用します。それに近い数値がある場合は、その制限よりも`spark.sql.shuffle.paritions`値を設定して、より効率的な表現を取得できます。

# シャッフルの最適化

`join()` や などの特定のオペレーションでは`groupByKey()`、Spark でシャッフルを実行する必要があります。シャッフルは、データがRDDパーティション間で異なるようにグループ化されるように、Spark がデータを再分散するメカニズムです。シャッフルは、パフォーマンスのボトルネックを修正するのに役立ちます。ただし、シャッフルには通常 Spark エグゼキューター間でのデータのコピーが含まれるため、シャッフルは複雑でコストがかかる操作です。例えば、シャッフルでは次のコストが発生します。

- ディスク I/O:
  - ディスク上に多数の中間ファイルを生成します。
- ネットワーク I/O:
  - 多くのネットワーク接続が必要です (接続数 = `Mapper × Reducer` ) 。
  - レコードは別の Spark エグゼキューターでホストされている可能性のある新しいRDDパーティションに集約されるため、データセットの大部分がネットワーク経由で Spark エグゼキューター間で移動する可能性があります。
- CPU およびメモリ負荷：

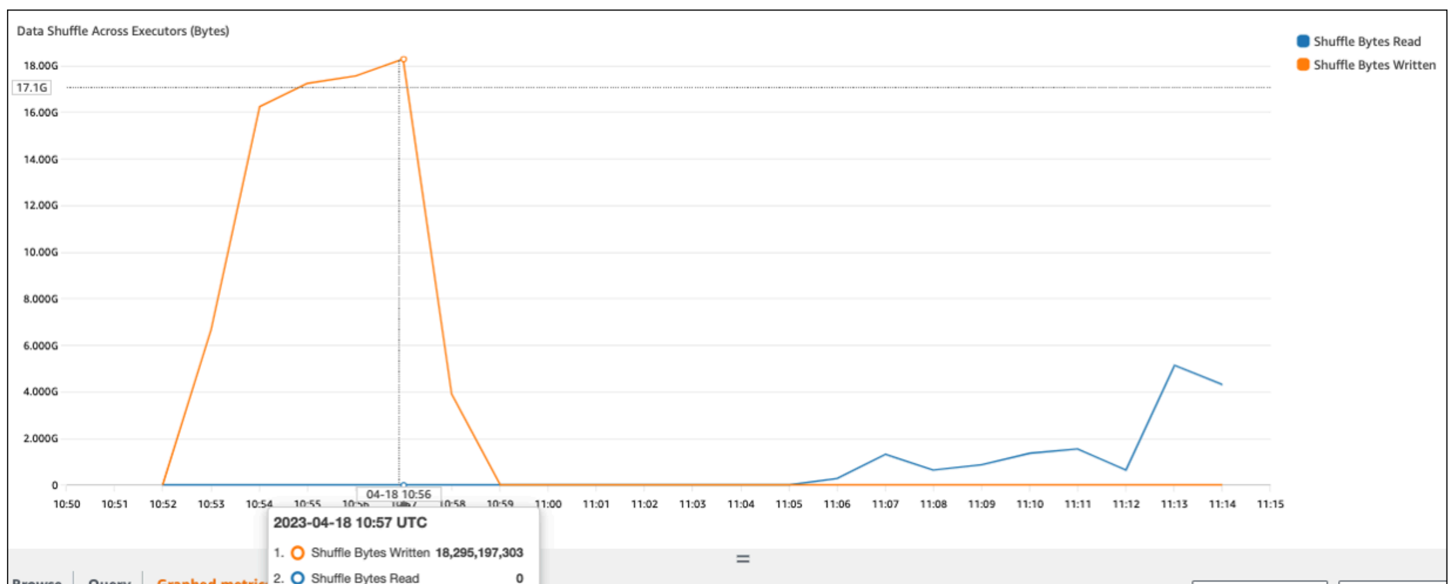
- 値をソートし、データセットをマージします。これらのオペレーションはエグゼキューターで計画され、エグゼキューターに負荷がかかります。

シャッフルは、Spark アプリケーションのパフォーマンス低下における最も重要な要因の 1 つです。中間データの保存中に、エグゼキューターのローカルディスクの領域が枯渇し、Spark ジョブが失敗する可能性があります。

シャッフルパフォーマンスは、 CloudWatch メトリクスと Spark UI で評価できます。

## CloudWatch メトリクス

シャッフルバイト書き込み値がシャッフルバイト読み取り と比較して高い場合、Spark ジョブは `join()`や などの[シャッフルオペレーション](#)を使用することがあります`groupByKey()`。




## Spark UI

Spark UI のステージタブで、シャッフル読み取りサイズ/レコードの値を確認できます。エグゼキュータータブでも確認できます。

次のスクリーンショットでは、各エグゼキューターはシャッフルプロセスで約 18.6 GB/4020000 レコードを交換でき、合計シャッフル読み取りサイズは約 75 GB) です。

Shuffle Spill (Disk) 列には大量のデータスピルメモリがディスクに表示され、ディスクがいっぱいになったり、パフォーマンスの問題が発生したりする可能性があります。

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 |

これらの症状を観察し、パフォーマンス目標と比較してステージに時間がかかりすぎる場合、または `Out Of Memory`または `No space left on device` エラーで失敗する場合は、次の解決策を検討してください。

## 結合の最適化

テーブルを結合する `join()`オペレーションは、最も一般的に使用されるシャッフルオペレーションですが、多くの場合、パフォーマンスのボトルネックです。結合はコストのかかる操作であるため、ビジネス要件に不可欠な場合を除き、使用しないことをおすすめします。次の質問をして、データパイプラインを効率的に使用していることを再確認します。

- 再利用できる他のジョブでも実行される結合を再計算していますか？
- 外部キーを出力のコンシューマーが使用していない値に解決するために参加していますか？

参加オペレーションがビジネス要件に不可欠であることを確認したら、要件を満たす方法で参加を最適化するための以下のオプションを参照してください。

参加前にプッシュダウンを使用する

結合を実行する DataFrame 前に、 で不要な行と列を除外します。これには次の利点があります。

- シャッフル中のデータ転送量を削減
- Spark エグゼキューターの処理量を削減します。
- データスキャンの量を削減

```
# 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"])
```

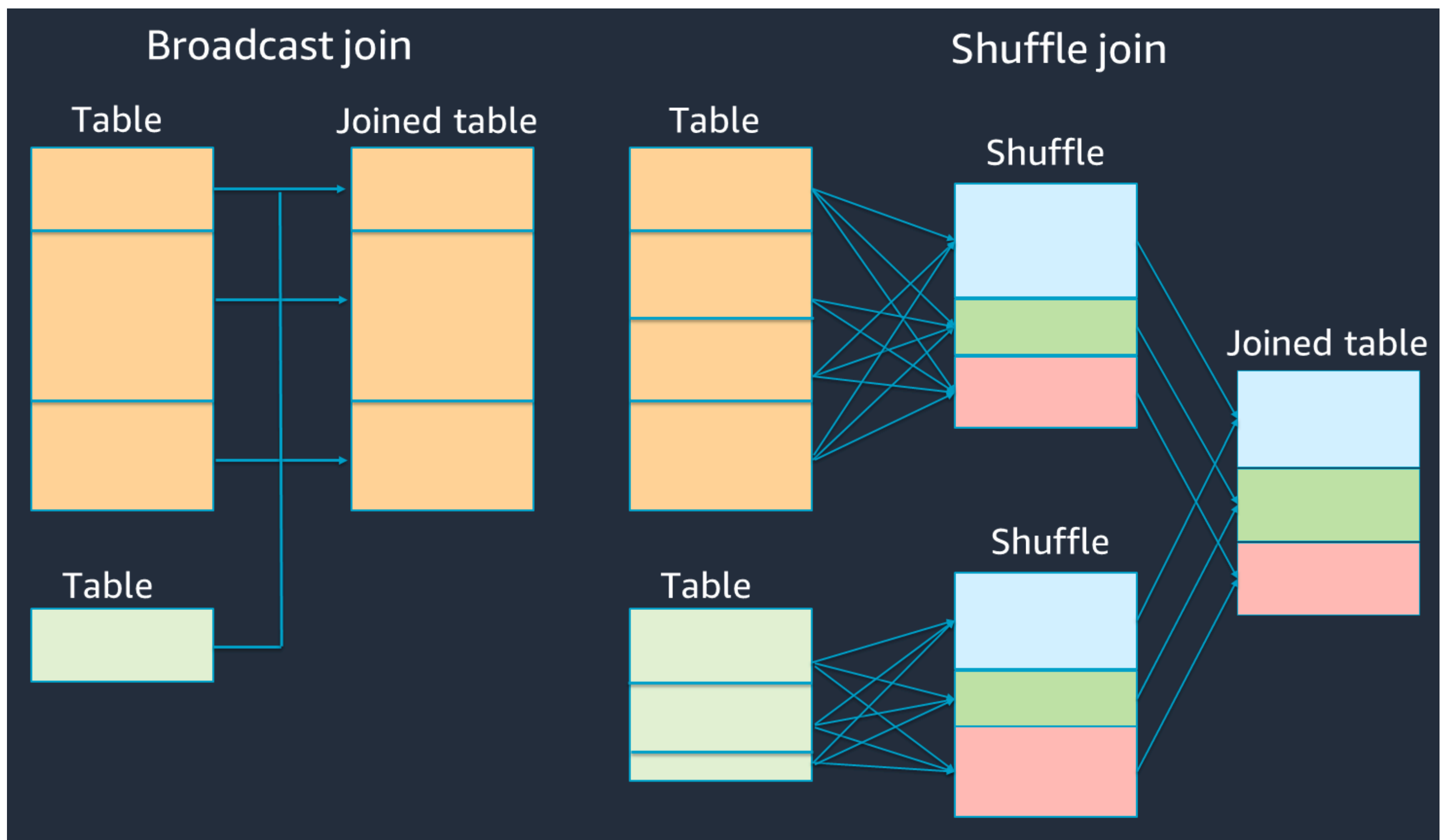
Join DataFrame を使用する

RDD API または DynamicFrame 結合の代わりに Spark 、、データセットなどの Spark の高レベルAPIの使用を試してください。SQL DataFrameなどのメソッド呼び出し DataFrame を使用して DynamicFrame に変換できます`dyf.toDF()`。Apache Spark の「キートピック」セクションで説明したように、これらの結合オペレーションは、Catalyst オプティマイザによるクエリ最適化を内部的に活用します。

ハッシュ結合とヒントのシャッフルとブロードキャスト

Spark は、シャッフル結合とブロードキャストハッシュ結合の 2 種類の結合をサポートしています。ブロードキャストハッシュ結合にはシャッフルは不要で、シャッフル結合よりも処理が少なくて済みます。ただし、小さなテーブルを大きなテーブルに結合する場合にのみ適用されます。単一の Spark エグゼキューターのメモリに収まるテーブルを結合する場合は、ブロードキャストハッシュ結合の使用を検討してください。

次の図は、ブロードキャストハッシュ結合とシャッフル結合の大まかな構造とステップを示しています。



各結合の詳細は次のとおりです。

- シャッフル結合：
  - シャッフルハッシュ結合は、ソートせずに 2 つのテーブルを結合し、2 つのテーブル間に結合を分散します。Spark エグゼキューターのメモリに保存できる小さなテーブルの結合に適しています。
  - ソートマージ結合は、結合する 2 つのテーブルをキーで分散し、結合する前にソートします。大きなテーブルの結合に適しています。
- ブロードキャストハッシュ結合：
  - ブロードキャストハッシュ結合は、小さい RDDまたは テーブルを各ワーカーノードにプッシュします。次に、大きな RDDまたはテーブルの各パーティションとマップ側の結合を行います。

    これは、 RDDsまたは テーブルのいずれかがメモリに収まるか、メモリに収まるように作成できる結合に適しています。ブロードキャストハッシュ結合はシャッフルを必要としないため、可能な限り行うことをお勧めします。結合ヒントを使用して、次のように Spark からブロードキャスト結合をリクエストできます。

    ```
    # 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;
    ```

    結合ヒントの詳細については、「結合ヒント」を参照してください。

AWS Glue 3.0 以降では、アダプティブクエリの実行と追加のパラメータを有効にすることで、ブロードキャストハッシュ結合を自動的に利用できます。アダプティブクエリの実行は、いずれかの結合側のランタイム統計がアダプティブブロードキャストハッシュ結合しきい値よりも小さい場合に、ソートマージ結合をブロードキャストハッシュ結合に変換します。

AWS Glue 3.0 では、 を設定することで、アダプティブクエリの実行を有効にできます`spark.sql.adaptive.enabled=true`。アダプティブクエリの実行は、 Glue AWS 4.0 でデフォルトで有効になっています。

シャッフルとブロードキャストハッシュ結合に関連する追加のパラメータを設定できます。

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

関連するパラメータの詳細については、「ソートマージ結合をブロードキャスト結合に変換する」を参照してください。

AWS Glue 3.0 以降では、シャッフルに他の結合ヒントを使用して動作を調整できます。

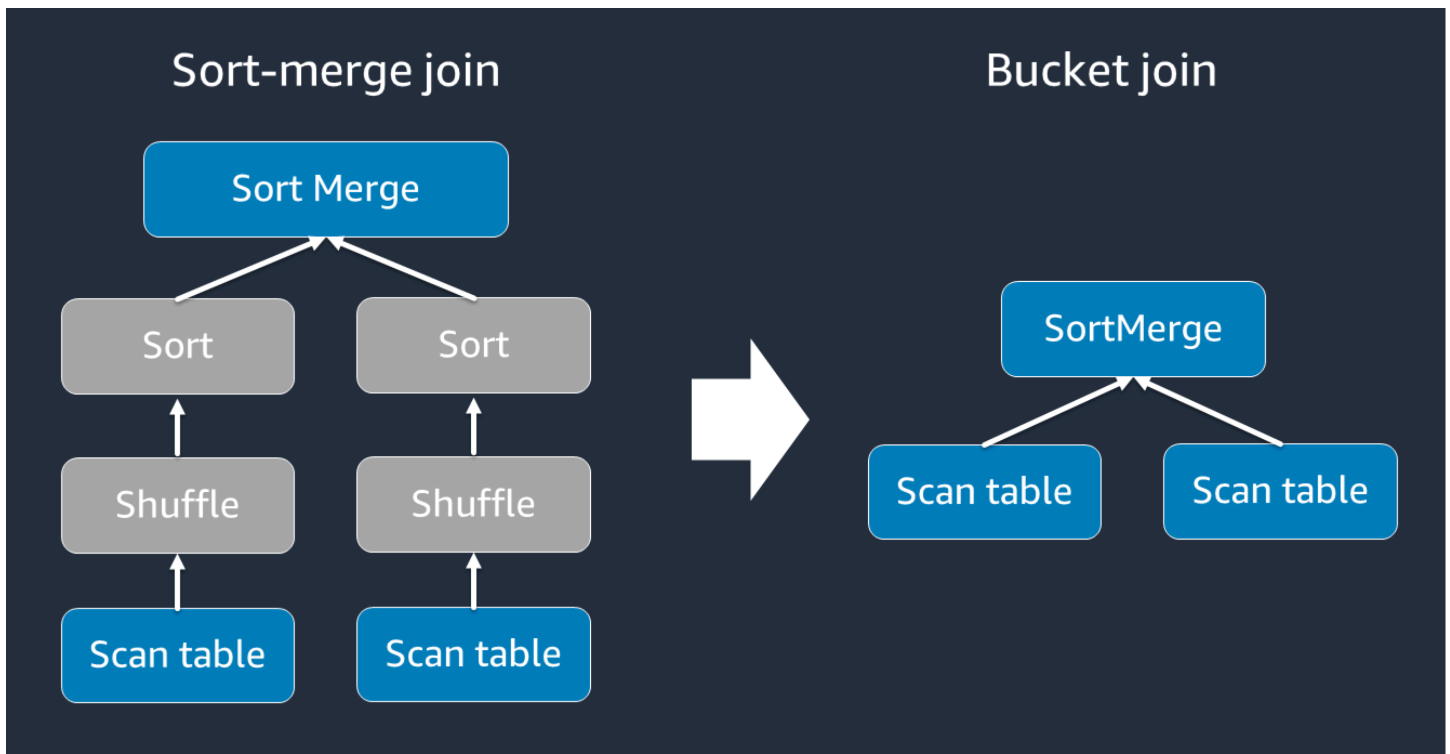
```
-- Join Hints for shuffle sort merge join
SELECT /*+ SHUFFLE_MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGEJOIN(t2) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- 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;
```

バケット化を使用する

ソートマージ結合には、シャッフルとソートの 2 つのフェーズが必要で、その後マージします。これらの 2 つのフェーズでは、Spark エグゼキューターが過負荷になり、一部のエグゼキューターがマージしていて、他のエグゼキューターが同時にソートされている場合に および OOMのパフォーマンスの問題が発生する可能性があります。このような場合、バケット化 を使用して効率的に結合できる場合があります。バケット化は、結合キーの入力を事前にシャッフルして事前にソートし、ソートされたデータを中間テーブルに書き込みます。ソートされた中間テーブルを事前に定義することで、大きなテーブルを結合する場合のシャッフルステップとソートステップのコストを削減できます。




バケット化されたテーブルは、次の場合に便利です。

- などの同じキーで頻繁に結合されるデータ account_id
- 共通列にバケット化できるベーステーブルやデルタテーブルなどの日次累積テーブルをロードする

次のコードを使用して、バケット化されたテーブルを作成できます。

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

結合前の結合キー DataFrames の再パーティション

結合の前に結合キー DataFrames で 2 つの を再パーティションするには、次のステートメントを使用します。

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

これにより、結合を開始する前に、結合キーRDDsで 2 つ (まだ分離されています) がパーティション化されます。2 つの RDDsが同じパーティション化コードを持つ同じキーでパーティション化されている場合、結合する計画RDDが結合のシャッフル前に同じワーカーにコロケーションされる可能性

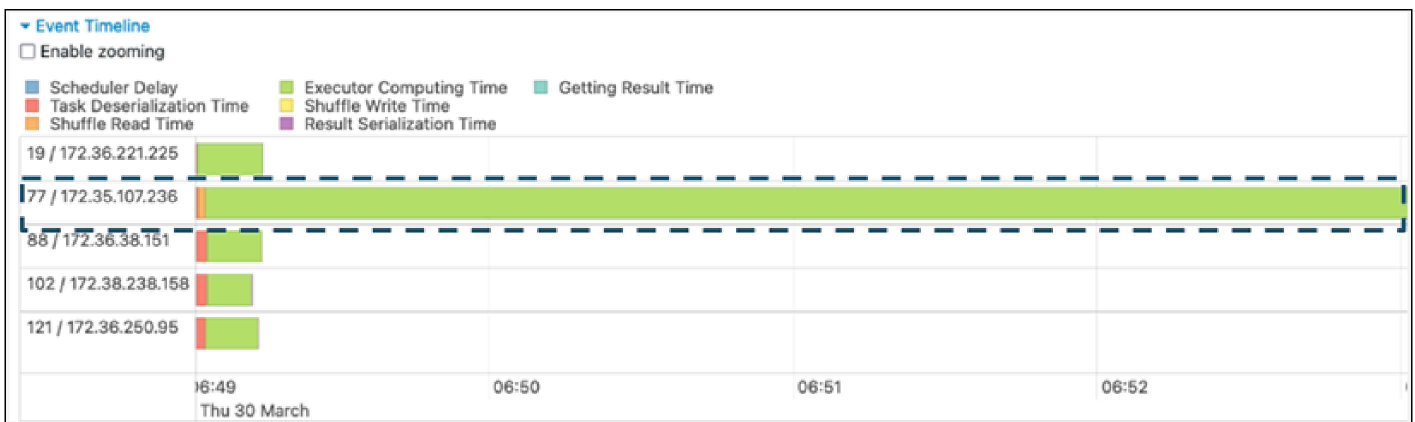
が高くなります。これにより、結合中のネットワークアクティビティとデータスキューが減少し、パフォーマンスが向上する可能性があります。

## データスキューを克服する

データスキューは、Spark ジョブのボトルネックの最も一般的な原因の 1 つです。これは、データがRDDパーティション間で均一に分散されていない場合に発生します。これにより、そのパーティションのタスクが他のパーティションよりもはるかに時間がかかり、アプリケーションの全体的な処理時間が遅延します。

データスキューを特定するには、Spark UI で次のメトリクスを評価します。

- Spark UI のステージタブで、イベントタイムラインページを確認します。次のスクリーンショットでは、タスクの不均等な分布を確認できます。不均等に分散されているタスクや実行に時間がかかりすぎるタスクは、データスキューを示している可能性があります。




- もう 1 つの重要なページは、Spark タスクの統計を表示する概要メトリクス です。次のスクリーンショットは、期間 、GC 時間 、スピル (メモリ ) 、スピル (ディスク) などのパーセンタイルを持つメトリクスを示しています。

**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 |

タスクが均等に分散されると、すべてのパーセンタイルに同様の数値が表示されます。データスキューがあると、各パーセンタイルに非常にバイアスされた値が表示されます。この例では、最小 、25 番目のパーセンタイル 、中央値 、および 75 番目のパーセンタイル のタスク期間は 13 秒

未満です。 Max タスクは 75 パーセンタイル の 100 倍多くのデータを処理しましたが、6.4 分の期間は約 30 倍長くなります。つまり、少なくとも 1 つのタスク (またはタスクの 25% まで) が残りのタスクよりもはるかに長くかかったということです。

データスキューが表示された場合は、以下を試してください。

- AWS Glue 3.0 を使用する場合は、 を設定してアダプティブクエリの実行を有効にします`spark.sql.adaptive.enabled=true`。アダプティブクエリの実行は AWS Glue 4.0 でデフォルトで有効になっています。

  次の関連パラメータを設定することで、結合によって導入されたデータスキューに Adaptive Query Execution を使用することもできます。

  - `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)`

  詳細については、「Apache Spark ドキュメント」を参照してください。

- 結合キーには、幅広い値を持つキーを使用します。シャッフル結合では、パーティションはキーのハッシュ値ごとに決定されます。結合キーのカーディナリティが低すぎる場合、ハッシュ関数はパーティション間でデータを分散する不正なジョブを実行する可能性が高くなります。したがって、アプリケーションとビジネスロジックでサポートされている場合は、より高い基数キーまたは複合キーを使用することを検討してください。

```
# 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"])
```

## キャッシュを使用する

反復的な を使用する場合は DataFrames、 または を使用して計算結果を各 Spark エグゼキューターのメモリとディスクにdf.persist()キャッシュすることで、追加のシャッフルdf.cache()や計算を避けます。Spark は、ディスクRDDsでの保持または複数のノード (ストレージレベル) でのレプリケーションもサポートしています。

例えば、 DataFrames を追加して を保持できますdf.persist()。キャッシュが不要になった場合は、 unpersistを使用してキャッシュされたデータを破棄できます。

```
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()
```

## 不要な Spark アクションを削除する

count、、showまたは などの不要なアクションを実行することは避けてくださいcollect。「Apache Spark の主要トピック」セクションで説明したように、Spark は遅延しています。変換された各 は、アクションを実行するたびに再計算RDDされる場合があります。Spark アクションを多数使用すると、アクションごとに複数のソースアクセス、タスク計算、シャッフル実行が呼び出されます。

商用環境で collect()やその他のアクションが必要ない場合は、それらを削除することを検討してください。

> **Note**
>
> Spark collect()は商用環境でできるだけ使用しないでください。collect() アクションは、Spark エグゼキューターの計算のすべての結果を Spark ドライバーに返します。これにより、Spark ドライバーがOOMエラーを返す可能性があります。OOM エラーを回避するために、Spark spark.driver.maxResultSize = 1GBはデフォルトで を設定します。これにより、Spark ドライバーに返される最大データサイズが 1 GB に制限されます。

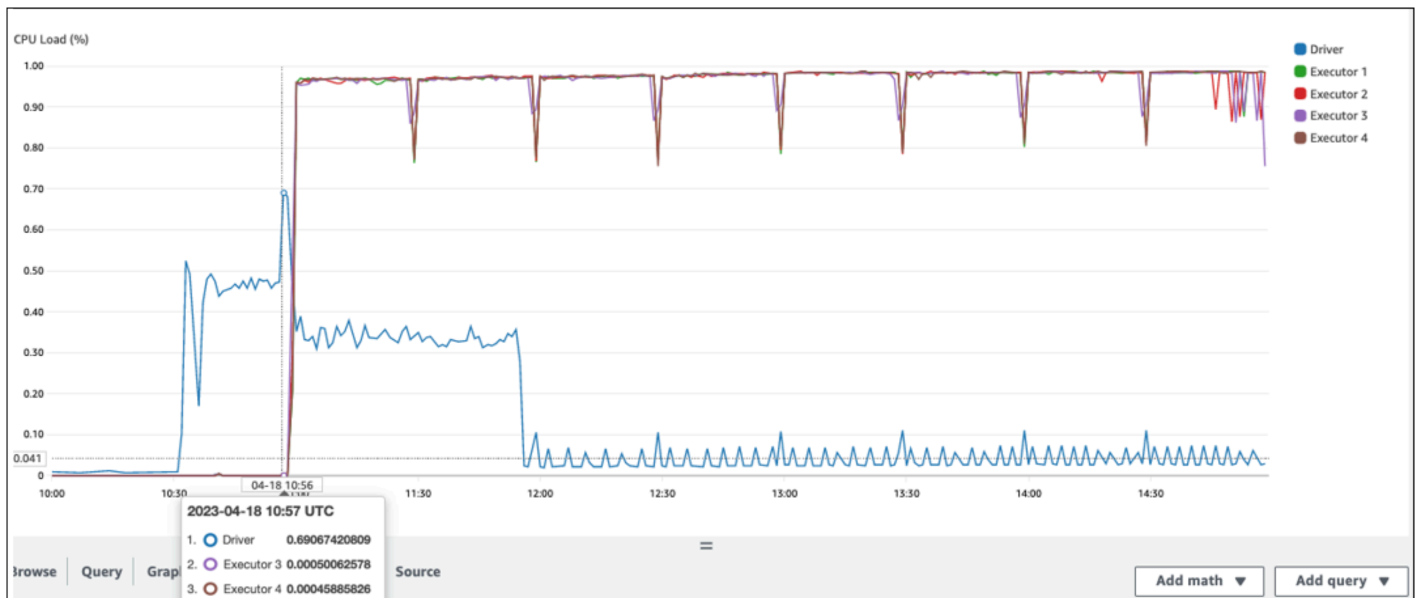
# 計画オーバーヘッドを最小限に抑える

Apache Spark で説明したように、Spark ドライバーは実行プランを生成します。その計画に基づいて、タスクは分散処理のために Spark エグゼキューターに割り当てられます。ただし、小さなファイルが多数ある場合や、 に AWS Glue Data Catalog 多数のパーティションが含まれている場合、Spark ドライバーはボトルネックになる可能性があります。計画オーバーヘッドが高いことを特定するには、次のメトリクスを評価します。

## CloudWatch メトリクス

CPU ロードとメモリの使用率をチェックして、次の状況を確認します。

- Spark ドライバーのCPU負荷とメモリ使用率は高いと記録されます。通常、Spark ドライバーはデータを処理しないため、CPU負荷とメモリの使用率が急増することはありません。ただし、Amazon S3 データソースの小さなファイルが多すぎる場合、すべての S3 オブジェクトを一覧表示し、多数のタスクを管理すると、リソース使用率が高くなる可能性があります。
- Spark エグゼキューターで処理が開始されるまでに長いギャップがあります。次のスクリーンショット例では、 AWS Glue ジョブが 10:00 に開始されたにもかかわらず、Spark エグゼキューターのCPUロードが 10:57 まで低すぎます。これは、Spark ドライバーが実行プランの生成に時間がかかる可能性があることを示します。この例では、Data Catalog 内の多数のパーティションを取得し、Spark ドライバー内の多数の小さなファイルを一覧表示するのに時間がかかります。



# Spark UI

Spark UI の ジョブ タブで、送信日時を確認できます。次の例では、ジョブが 10:00:00 に開始されたにもかかわらず、Spark ドライバーは 10:56:46 に AWS Glue job0 を開始しました。

- Completed Jobs (1)

| Job Id ▾ | Description | Submitted | Duration | Stages: Succeeded/Total | Tasks (for all stages): Succeeded/Total |
| --- | --- | --- | --- | --- | --- |
| 0 | count at DynamicFrame.scala:1414<br>count at DynamicFrame.scala:1414 | 2023/04/18 10:56:46 | 4.9 h | 1/1 | 58100/58100 |

タスク (すべてのステージ): 成功/合計時間をジョブタブで確認することもできます。この場合、タスクの数は として記録されます58100。Parallelize tasks ページの Amazon S3 セクションで説明されているように、タスクの数は S3 オブジェクトの数にほぼ対応しています。つまり、Amazon S3 には約 58,100 個のオブジェクトがあります。

このジョブとタイムラインの詳細については、ステージタブを参照してください。Spark ドライバーでボトルネックが発生した場合は、次の解決策を検討してください。

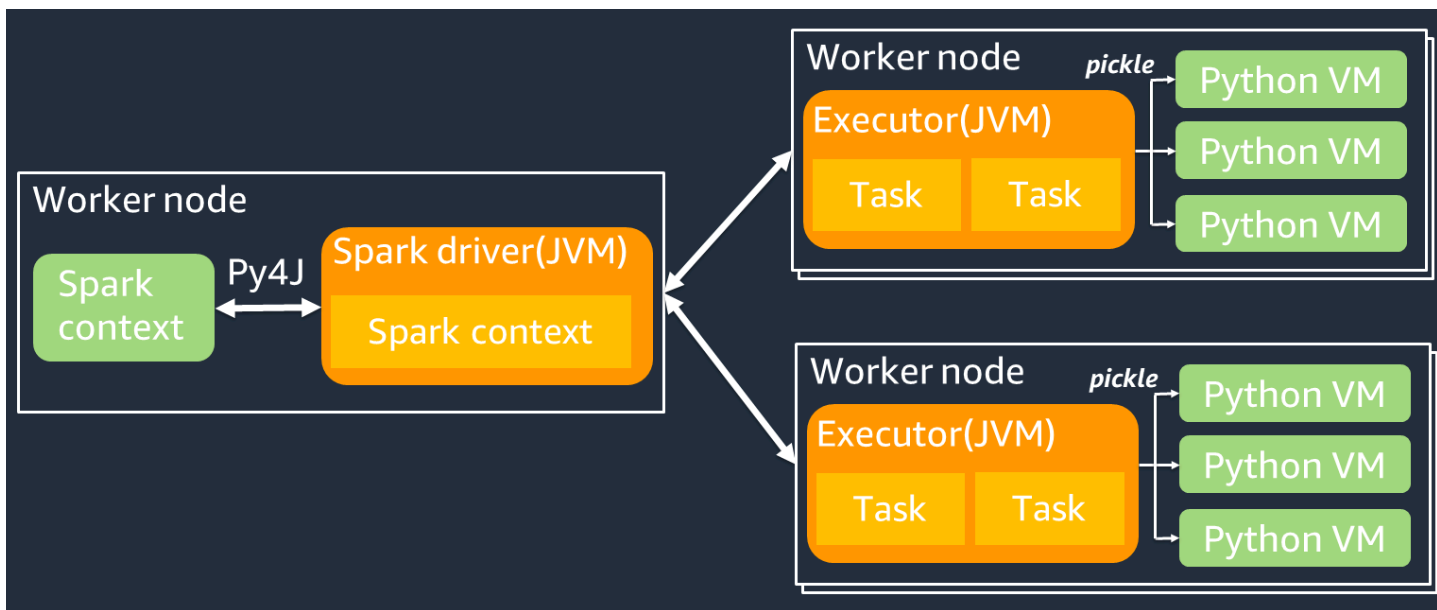
- Amazon S3 のファイルが多すぎる場合は、並列処理タスクページの「パーティションが多すぎます」セクションにある過剰な並列処理に関するガイダンスを検討してください。
- Amazon S3 のパーティションが多すぎる場合は、データスキャン量を減らすページのAmazon S3 パーティションが多すぎる」セクションで、過剰なパーティション分割に関するガイダンスを検討してください。パーティションが多い場合はパーティションAWS Glue インデックスを有効にして、Data Catalog からパーティションメタデータを取得する際のレイテンシーを短縮します。詳細については、AWS Glue 「パーティションインデックス を使用したクエリパフォーマンスの向上」を参照してください。
- JDBC にパーティションが多すぎる場合は、`hashpartition`値を小さくします。
- DynamoDB のパーティションが多すぎる場合は、 `dynamodb.splits`値を小さくします。
- ストリーミングジョブのパーティションが多すぎる場合は、シャードの数を減らします。

# ユーザー定義関数の最適化

のユーザー定義関数 (UDFs) と `RDD.map` では、パフォーマンスが大幅に低下することが PySpark よくあります。これは、Spark の基盤となる Scala 実装で Python コードを正確に表すために必要なオーバーヘッドが原因です。

次の図は、 PySpark ジョブのアーキテクチャを示しています。を使用する場合 PySpark、Spark ドライバーは Py4j ライブラリを使用して Python から Java メソッドを呼び出します。Spark SQLまたは DataFrame 組み込み関数を呼び出す場合、Python と Scala のパフォーマンスの違いはほとんどあ

りません。関数は、最適化された実行プランJVMを使用して各エグゼキューターの で実行されるためです。




の使用など、独自の Python ロジックを使用する場合map/ mapPartitions/ udf、タスクは Python ランタイム環境で実行されます。2 つの環境を管理すると、オーバーヘッドコストが発生します。さらに、JVMランタイム環境の組み込み関数で使用するには、メモリ内のデータを変換する必要があります。Pickle は、デフォルトで JVMと Python ランタイム間の交換に使用されるシリアル化形式です。ただし、このシリアル化と逆シリアル化のコストは非常に高くなるため、Java または Scala でUDFs記述される方が Python よりも高速ですUDFs。

のシリアル化と逆シリアル化のオーバーヘッドを回避するには PySpark、次の点を考慮してください。

- 組み込みの Spark SQL関数を使用する — 独自の UDFまたはマップ関数を Spark SQLまたは DataFrame組み込み関数に置き換えることを検討してください。Spark SQLまたは DataFrame 組み込み関数を実行する場合、タスクは各エグゼキューターの で処理されるため、Python と Scala のパフォーマンスの違いはほとんどありませんJVM。
- Scala または Java UDFsで実装する — Java または Scala で記述UDFされた を使用することを検討してください。これらは で実行されるためですJVM。
- ベクトル化されたワークロードUDFsに Apache Arrow ベースを使用する – Arrow ベースの の使用を検討してくださいUDFs。この機能はベクトル化 UDF (Pandas ) とも呼ばれますUDF。Apache Arrow は言語に依存しないインメモリデータ形式であり、 を使用して JVMと Python プロセス間でデータを効率的に転送 AWS Glue できます。これは現在、Pandas または NumPyデータを操作する Python ユーザーにとって最も有益です。

矢印は列指向 (ベクトル化) 形式です。その使用は自動ではなく、設定やコードにわずかな変更を加えることで、互換性を最大限に引き出すことができます。詳細と制限については、「」の「Apache Arrow PySpark」を参照してください。

次の例では、UDF標準の Python、ベクトル化された 、UDFSpark の基本的な増分を比較します SQL。

## 標準 Python UDF

時間の例は 3.20 (秒) です。

コードの例

```
# 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()
```

実行計画

```
== 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)
```

## ベクトル化 UDF

時間の例は 0.59 (秒) です。

ベクトル化された UDFは、前のUDF例の 5 倍高速です。を確認するとPhysical Plan、 が表示されます。これはArrowEvalPython、このアプリケーションが Apache Arrow によってベクトル化されていることを示しています。ベクトル化された を有効にするにはUDF、コードspark.sql.execution.arrow.pyspark.enabled = trueで を指定する必要があります。

コードの例

```
# 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()
```

実行計画

```
== 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

時間の例は 0.087 (秒) です。

タスクSQLは Python ランタイム JVMなしで各エグゼキューターの で実行されるためUDF、Spark は ベクトル化された よりもはるかに高速です。UDF を組み込み関数に置き換えることができる場合は、置き換えることをお勧めします。

コードの例

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

## ビッグデータに pandas を使用する

すでに pandas に精通していて、ビッグデータに Spark を使用する場合は、Spark. AWS Glue 4.0 API以降の pandas を使用できます。開始するには、公式ノートブック Quickstart: Pandas API on Spark を使用できます。詳細については、 PySpark ドキュメントを参照してください。

# リソース

- AWS Glue
- パフォーマンスチューニング (Spark SQL ガイド）
- AWS Glue 最適化ワークショップ

# ドキュメント履歴

以下の表は、本ガイドの重要な変更点について説明したものです。今後の更新に関する通知を受け取る場合は、RSS フィード をサブスクライブできます。

| 変更 | 説明 | 日付 |
| --- | --- | --- |
| 初版発行 | — | 2024 年 1 月 2 日 |

# AWS 規範的ガイダンスの用語集

以下は、 AWS 規範的ガイダンスが提供する戦略、ガイド、パターンで一般的に使用される用語です。エントリを提案するには、用語集の最後のフィードバックの提供リンクを使用します。

## 数字

7 Rs

アプリケーションをクラウドに移行するための 7 つの一般的な移行戦略。これらの戦略は、ガートナーが 2011 年に特定した 5 Rs に基づいて構築され、以下で構成されています。

- リファクタリング/アーキテクチャの再設計 — クラウドネイティブ特徴を最大限に活用して、俊敏性、パフォーマンス、スケーラビリティを向上させ、アプリケーションを移動させ、アーキテクチャを変更します。これには、通常、オペレーティングシステムとデータベースの移植が含まれます。例: オンプレミスの Oracle データベースを Amazon Aurora PostgreSQL 互換エディションに移行します。
- リプラットフォーム (リフトアンドリシェイプ) — アプリケーションをクラウドに移行し、クラウド機能を活用するための最適化レベルを導入します。例: オンプレミスの Oracle データベースを の Oracle 用 Amazon Relational Database Service (Amazon RDS) に移行します AWS クラウド。
- 再購入 (ドロップアンドショップ) — 通常、従来のライセンスから SaaS モデルに移行して、別の製品に切り替えます。例: カスタマーリレーションシップ管理 (CRM) システムを Salesforce.com に移行します。
- リホスト (リフトアンドシフト) — クラウド機能を活用するための変更を加えずに、アプリケーションをクラウドに移行します。例: オンプレミスの Oracle データベースを の EC2 インスタンス上の Oracle に移行します AWS クラウド。
- 再配置 (ハイパーバイザーレベルのリフトアンドシフト) – 新しいハードウェアを購入したり、アプリケーションを書き換えたり、既存の運用を変更したりすることなく、インフラストラクチャをクラウドに移行できます。サーバーをオンプレミスプラットフォームから同じプラットフォームのクラウドサービスに移行します。例: Microsoft Hyper-Vアプリケーションを に移行します AWS。
- 保持 (再アクセス) — アプリケーションをお客様のソース環境で保持します。これには、主要なリファクタリングを必要とするアプリケーションや、お客様がその作業を後日まで延期したいアプリケーション、およびそれらを移行するためのビジネス上の正当性がないため、お客様が保持するレガシーアプリケーションなどがあります。

- 使用停止 — お客様のソース環境で不要になったアプリケーションを停止または削除します。

# A

ABAC

「属性ベースのアクセスコントロール」を参照してください。

抽象化されたサービス

「 マネージドサービス」を参照してください。

ACID

「原子性、一貫性、分離性、耐久性」を参照してください。

アクティブ - アクティブ移行

(双方向レプリケーションツールまたは二重書き込み操作を使用して) ソースデータベースとターゲットデータベースを同期させ、移行中に両方のデータベースが接続アプリケーションからのトランザクションを処理するデータベース移行方法。この方法では、1 回限りのカットオーバーの必要がなく、管理された小規模なバッチで移行できます。アクティブ/パッシブ移行よりも柔軟ですが、より多くの作業が必要です。

アクティブ - パッシブ移行

ソースデータベースとターゲットデータベースを同期させながら、データがターゲットデータベースにレプリケートされている間、接続しているアプリケーションからのトランザクションをソースデータベースのみで処理するデータベース移行の方法。移行中、ターゲットデータベースはトランザクションを受け付けません。

集計関数

行のグループを操作し、グループの単一の戻り値を計算する SQL 関数。集計関数の例としては、 `SUM`や などがあります`MAX`。

AI

「人工知能」を参照してください。

AIOps

「人工知能オペレーション」を参照してください。

匿名化

データセット内の個人情報を完全に削除するプロセス。匿名化は個人のプライバシー保護に役立ちます。匿名化されたデータは、もはや個人データとは見なされません。

アンチパターン

繰り返し起こる問題に対して頻繁に用いられる解決策で、その解決策が逆効果であったり、効果がなかったり、代替案よりも効果が低かったりするもの。

アプリケーションコントロール

マルウェアからシステムを保護するために、承認されたアプリケーションのみを使用できるようにするセキュリティアプローチ。

アプリケーションポートフォリオ

アプリケーションの構築と維持にかかるコスト、およびそのビジネス価値を含む、組織が使用する各アプリケーションに関する詳細情報の集まり。この情報は、ポートフォリオの検出と分析プロセス の需要要素であり、移行、モダナイズ、最適化するアプリケーションを特定し、優先順位を付けるのに役立ちます。

人工知能 (AI)

コンピューティングテクノロジーを使用し、学習、問題の解決、パターンの認識など、通常は人間に関連づけられる認知機能の実行に特化したコンピュータサイエンスの分野。詳細については、「人工知能 (AI) とは何ですか?」を参照してください。

AI オペレーション (AIOps)

機械学習技術を使用して運用上の問題を解決し、運用上のインシデントと人の介入を減らし、サービス品質を向上させるプロセス。 AWS 移行戦略での AIOps の使用方法については、オペレーション統合ガイド を参照してください。

非対称暗号化

暗号化用のパブリックキーと復号用のプライベートキーから成る 1 組のキーを使用した、暗号化のアルゴリズム。パブリックキーは復号には使用されないため共有しても問題ありませんが、プライベートキーの利用は厳しく制限する必要があります。

原子性、一貫性、分離性、耐久性 (ACID)

エラー、停電、その他の問題が発生した場合でも、データベースのデータ有効性と運用上の信頼性を保証する一連のソフトウェアプロパティ。

属性ベースのアクセス制御 (ABAC)

部署、役職、チーム名など、ユーザーの属性に基づいてアクセス許可をきめ細かく設定する方法。詳細については、 AWS Identity and Access Management (IAM) ドキュメントの「 の ABAC AWS」を参照してください。

信頼できるデータソース

最も信頼性のある情報源とされるデータのプライマリーバージョンを保存する場所。匿名化、編集、仮名化など、データを処理または変更する目的で、信頼できるデータソースから他の場所にデータをコピーすることができます。

アベイラビリティーゾーン

他のアベイラビリティーゾーンの障害から AWS リージョン 隔離され、同じリージョン内の他のアベイラビリティーゾーンへの低コストで低レイテンシーのネットワーク接続を提供する 内の別の場所。

AWS クラウド導入フレームワーク (AWS CAF)

組織がクラウドに正常に移行 AWS するための効率的で効果的な計画を立てるのに役立つ、 のガイドラインとベストプラクティスのフレームワーク。 AWS CAF は、ビジネス、人材、ガバナンス、プラットフォーム、セキュリティ、運用という 6 つの重点分野にガイダンスを編成します。ビジネス、人材、ガバナンスの観点では、ビジネススキルとプロセスに重点を置き、プラットフォーム、セキュリティ、オペレーションの視点は技術的なスキルとプロセスに焦点を当てています。例えば、人材の観点では、人事 (HR)、人材派遣機能、および人材管理を扱うステークホルダーを対象としています。この観点から、 AWS CAF は、クラウド導入を成功させるための組織の準備に役立つ人材開発、トレーニング、コミュニケーションに関するガイダンスを提供します。詳細については、AWS CAF ウェブサイト と AWS CAF のホワイトペーパー を参照してください。

AWS ワークロード認定フレームワーク (AWS WQF)

データベース移行ワークロードを評価し、移行戦略を推奨し、作業見積もりを提供するツール。AWS WQF は AWS Schema Conversion Tool （AWS SCT) に含まれています。データベーススキーマとコードオブジェクト、アプリケーションコード、依存関係、およびパフォーマンス特性を分析し、評価レポートを提供します。

# B

不正なボット

個人または組織に混乱や損害を与えることを目的としたボット。

BCP

「事業継続計画」を参照してください。

動作グラフ

リソースの動作とインタラクションを経時的に示した、一元的なインタラクティブビュー。Amazon Detective の動作グラフを使用すると、失敗したログオンの試行、不審な API 呼び出し、その他同様のアクションを調べることができます。詳細については、Detective ドキュメントのData in a behavior graphを参照してください。

ビッグエンディアンシステム

最上位バイトを最初に格納するシステム。エンディアンネス も参照してください。

二項分類

バイナリ結果 (2 つの可能なクラスのうちの 1 つ) を予測するプロセス。例えば、お客様の機械学習モデルで「この E メールはスパムですか、それともスパムではありませんか」などの問題を予測する必要があるかもしれません。または「この製品は書籍ですか、車ですか」などの問題を予測する必要があるかもしれません。

ブルームフィルター

要素がセットのメンバーであるかどうかをテストするために使用される、確率的でメモリ効率の高いデータ構造。

ブルー/グリーンデプロイ

2 つの異なる同一の環境を作成するデプロイ戦略。現在のアプリケーションバージョンは 1 つの環境 (ブルー) で実行し、新しいアプリケーションバージョンは他の環境 (グリーン) で実行します。この戦略は、影響を最小限に抑えて迅速にロールバックするのに役立ちます。

ボット

インターネット経由で自動タスクを実行し、人間のアクティビティやインタラクションをシミュレートするソフトウェアアプリケーション。インターネット上の情報のインデックスを作成するウェブクローラーなど、一部のボットは有用または有益です。悪質なボット と呼ばれる他のボッ

トの中には、個人や組織に混乱を与えたり、損害を与えたりすることを意図しているものがあります。

ボットネット

マルウェアに感染し、ボットのヘルダーまたはボットオペレーター と呼ばれる、単一関係者の管理下にあるボットのネットワーク。ボットは、ボットとその影響をスケールするための最もよく知られているメカニズムです。

ブランチ

コードリポジトリに含まれる領域。リポジトリに最初に作成するブランチは、メインブランチといいます。既存のブランチから新しいブランチを作成し、その新しいブランチで機能を開発したり、バグを修正したりできます。機能を構築するために作成するブランチは、通常、機能ブランチと呼ばれます。機能をリリースする準備ができたら、機能ブランチをメインブランチに統合します。詳細については、「ブランチについて (GitHub ドキュメント）」を参照してください。

ブレークグラスアクセス

例外的な状況や承認されたプロセスを通じて、ユーザーが通常アクセス許可を持たない AWS アカウント にすばやくアクセスできるようにします。詳細については、 Well-Architected ガイダンスの「ブレークグラスプロシージャの実装」インジケータ AWS を参照してください。

ブラウンフィールド戦略

環境の既存インフラストラクチャ。システムアーキテクチャにブラウンフィールド戦略を導入する場合、現在のシステムとインフラストラクチャの制約に基づいてアーキテクチャを設計します。既存のインフラストラクチャを拡張している場合は、ブラウンフィールド戦略とグリーンフィールド戦略を融合させることもできます。

バッファキャッシュ

アクセス頻度が最も高いデータが保存されるメモリ領域。

ビジネス能力

価値を生み出すためにビジネスが行うこと (営業、カスタマーサービス、マーケティングなど)。マイクロサービスのアーキテクチャと開発の決定は、ビジネス能力によって推進できます。詳細については、ホワイトペーパー AWSでのコンテナ化されたマイクロサービスの実行 の ビジネス機能を中心に組織化 セクションを参照してください。

ビジネス継続性計画 (BCP)

大規模移行など、中断を伴うイベントが運用に与える潜在的な影響に対処し、ビジネスを迅速に再開できるようにする計画。

# C

CAF

AWS 「クラウド導入フレームワーク」を参照してください。

Canary デプロイ

エンドユーザーへのバージョンの低速かつ増分的なリリース。確信できたら、新しいバージョンをデプロイし、現在のバージョン全体を置き換えます。

CCoE

「Cloud Center of Excellence」を参照してください。

CDC

「データキャプチャの変更」を参照してください。

変更データキャプチャ (CDC)

データソース (データベーステーブルなど) の変更を追跡し、その変更に関するメタデータを記録するプロセス。CDC は、ターゲットシステムでの変更を監査またはレプリケートして同期を維持するなど、さまざまな目的に使用できます。

カオスエンジニアリング

障害や破壊的なイベントを意図的に導入して、システムの耐障害性をテストします。AWS Fault Injection Service （AWS FIS） を使用して、 AWS ワークロードに負荷を与え、その応答を評価する実験を実行できます。

CI/CD

「継続的インテグレーションと継続的デリバリー」を参照してください。

分類

予測を生成するのに役立つ分類プロセス。分類問題の機械学習モデルは、離散値を予測します。離散値は、常に互いに区別されます。例えば、モデルがイメージ内に車があるかどうかを評価する必要がある場合があります。

クライアント側の暗号化

ターゲットがデータ AWS サービス を受信する前に、ローカルでデータを暗号化します。

Cloud Center of Excellence (CCoE)

クラウドのベストプラクティスの作成、リソースの移動、移行のタイムラインの確立、大規模変革を通じて組織をリードするなど、組織全体のクラウド導入の取り組みを推進する学際的なチーム。詳細については、 AWS クラウド エンタープライズ戦略ブログの CCoE の投稿を参照してください。

クラウドコンピューティング

リモートデータストレージと IoT デバイス管理に通常使用されるクラウドテクノロジー。クラウドコンピューティングは、一般的にエッジコンピューティングテクノロジーに接続されています。

クラウド運用モデル

IT 組織において、1 つ以上のクラウド環境を構築、成熟、最適化するために使用される運用モデル。詳細については、「クラウド運用モデルの構築」を参照してください。

導入のクラウドステージ

組織が に移行するときに通常実行する 4 つのフェーズ AWS クラウド :

- プロジェクト — 概念実証と学習を目的として、クラウド関連のプロジェクトをいくつか実行する
- 基礎固め — お客様のクラウドの導入を拡大するための基礎的な投資 (ランディングゾーンの作成、CCoE の定義、運用モデルの確立など)
- 移行 — 個々のアプリケーションの移行
- 再発明 — 製品とサービスの最適化、クラウドでのイノベーション

これらのステージは、 AWS クラウド エンタープライズ戦略ブログのブログ記事「クラウドファーストへのジャーニー」と「導入のステージ」で Stephen Orban によって定義されました。移行戦略とどのように関連しているかについては、 AWS 「移行準備ガイド」を参照してください。

CMDB

「設定管理データベース」を参照してください。

コードリポジトリ

ソースコードやその他の資産 (ドキュメント、サンプル、スクリプトなど) が保存され、バージョン管理プロセスを通じて更新される場所。一般的なクラウドリポジトリには、 GitHub または が含まれます AWS CodeCommit。コードの各バージョンはブランチと呼ばれます。マイクロサー

ビスの構造では、各リポジトリは 1 つの機能専用です。1 つの CI/CD パイプラインで複数のリポジトリを使用できます。

コールドキャッシュ

空である、または、かなり空きがある、もしくは、古いデータや無関係なデータが含まれているバッファキャッシュ。データベースインスタンスはメインメモリまたはディスクから読み取る必要があり、バッファキャッシュから読み取るよりも時間がかかるため、パフォーマンスに影響します。

コールドデータ

めったにアクセスされず、通常は過去のデータです。この種類のデータをクエリする場合、通常は低速なクエリでも問題ありません。このデータを低パフォーマンスで安価なストレージ階層またはクラスに移動すると、コストを削減することができます。

コンピュータビジョン (CV)

機械学習を使用してデジタルイメージやビデオなどのビジュアル形式から情報を分析および抽出する AI の分野。例えば、 はオンプレミスのカメラネットワークに CV を追加するデバイス AWS Panorama を提供し、Amazon SageMaker は CV の画像処理アルゴリズムを提供します。

設定ドリフト

ワークロードの場合、設定は想定した状態から変化します。これにより、ワークロードが非準拠になる可能性があり、通常は段階的かつ意図的ではありません。

構成管理データベース (CMDB)

データベースとその IT 環境 (ハードウェアとソフトウェアの両方のコンポーネントとその設定を含む) に関する情報を保存、管理するリポジトリ。通常、CMDB のデータは、移行のポートフォリオの検出と分析の段階で使用します。

コンフォーマンスパック

コンプライアンスチェックとセキュリティチェックをカスタマイズするためにアセンブルできる AWS Config ルールと修復アクションのコレクション。YAML テンプレートを使用して、コンフォーマンスパックを AWS アカウント および リージョンに、または組織全体に 1 つのエンティティとしてデプロイできます。詳細については、 AWS Config ドキュメントの「コンフォーマンスパック」を参照してください。

継続的インテグレーションと継続的デリバリー (CI/CD)

ソフトウェアリリースプロセスのソース、ビルド、テスト、ステージング、本番の各ステージを自動化するプロセス。CI/CD は一般的にパイプラインと呼ばれます。プロセスの自動化、生産性

の向上、コード品質の向上、配信の加速化を可能にします。詳細については、「継続的デリバリーの利点」を参照してください。CD は継続的デプロイ (Continuous Deployment) の略語でもあります。詳細については「継続的デリバリーと継続的なデプロイ」を参照してください。

CV

「コンピュータビジョン」を参照してください。

## D

保管中のデータ

ストレージ内にあるデータなど、常に自社のネットワーク内にあるデータ。

データ分類

ネットワーク内のデータを重要度と機密性に基づいて識別、分類するプロセス。データに適した保護および保持のコントロールを判断する際に役立つため、あらゆるサイバーセキュリティのリスク管理戦略において重要な要素です。データ分類は、 AWS Well-Architected フレームワークのセキュリティの柱のコンポーネントです。詳細については、データ分類を参照してください。

データドリフト

実稼働データと ML モデルのトレーニングに使用されたデータとの間に有意な差異が生じたり、入力データが時間の経過と共に有意に変化したりすることです。データドリフトは、ML モデル予測の全体的な品質、精度、公平性を低下させる可能性があります。

転送中のデータ

ネットワーク内 (ネットワークリソース間など) を活発に移動するデータ。

データメッシュ

一元化された管理とガバナンスにより、分散型の分散型データ所有権を提供するアーキテクチャフレームワーク。

データ最小化

厳密に必要なデータのみを収集し、処理するという原則。でデータ最小化を実践 AWS クラウドすることで、プライバシーリスク、コスト、分析のカーボンフットプリントを削減できます。

データ境界

AWS 環境内の一連の予防ガードレール。信頼できる ID のみが、期待されるネットワークから信頼できるリソースにアクセスしていることを確実にします。詳細については、「 でのデータ境界の構築 AWS」を参照してください。

データの前処理

raw データをお客様の機械学習モデルで簡単に解析できる形式に変換すること。データの前処理とは、特定の列または行を削除して、欠落している、矛盾している、または重複する値に対処することを意味します。

データ出所

データの生成、送信、保存の方法など、データのライフサイクル全体を通じてデータの出所と履歴を追跡するプロセス。

データ件名

データを収集、処理している個人。

データウェアハウス

分析などのビジネスインテリジェンスをサポートするデータ管理システム。データウェアハウスには通常、大量の履歴データが含まれており、クエリや分析によく使用されます。

データベース定義言語 (DDL)

データベース内のテーブルやオブジェクトの構造を作成または変更するためのステートメントまたはコマンド。

データベース操作言語 (DML)

データベース内の情報を変更 (挿入、更新、削除) するためのステートメントまたはコマンド。

DDL

「データベース定義言語」を参照してください。

ディープアンサンブル

予測のために複数の深層学習モデルを組み合わせる。ディープアンサンブルを使用して、より正確な予測を取得したり、予測の不確実性を推定したりできます。

ディープラーニング

人工ニューラルネットワークの複数層を使用して、入力データと対象のターゲット変数の間のマッピングを識別する機械学習サブフィールド。

defense-in-depth

一連のセキュリティメカニズムとコントロールをコンピュータネットワーク全体に層状に重ねて、ネットワークとその内部にあるデータの機密性、整合性、可用性を保護する情報セキュリティの手法。この戦略を に採用するときは AWS、 AWS Organizations 構造の異なるレイヤーに複数のコントロールを追加して、リソースの安全性を確保します。例えば、 defense-in-depth アプローチでは、多要素認証、ネットワークセグメンテーション、暗号化を組み合わせることができます。

委任管理者

では AWS Organizations、互換性のあるサービスが AWS メンバーアカウントを登録して組織のアカウントを管理し、そのサービスのアクセス許可を管理できます。このアカウントを、そのサービスの委任管理者と呼びます。詳細、および互換性のあるサービスの一覧は、 AWS Organizations ドキュメントのAWS Organizationsで使用できるサービスを参照してください。

デプロイメント

アプリケーション、新機能、コードの修正をターゲットの環境で利用できるようにするプロセス。デプロイでは、コードベースに変更を施した後、アプリケーションの環境でそのコードベースを構築して実行します。

開発環境

「環境」を参照してください。

検出管理

イベントが発生したときに、検出、ログ記録、警告を行うように設計されたセキュリティコントロール。これらのコントロールは副次的な防衛手段であり、実行中の予防的コントロールをすり抜けたセキュリティイベントをユーザーに警告します。詳細については、Implementing security controls on AWSのDetective controlsを参照してください。

開発バリューストリームマッピング (DVSM)

ソフトウェア開発ライフサイクルのスピードと品質に悪影響を及ぼす制約を特定し、優先順位を付けるために使用されるプロセス。DVSM は、もともとリーンマニュファクチャリング･プラクティスのために設計されたバリューストリームマッピング･プロセスを拡張したものです。ソフトウェア開発プロセスを通じて価値を創造し、動かすために必要なステップとチームに焦点を当てています。

デジタルツイン

建物、工場、産業機器、生産ラインなど、現実世界のシステムを仮想的に表現したものです。デジタルツインは、予知保全、リモートモニタリング、生産最適化をサポートします。

ディメンションテーブル

スタースキーマ では、ファクトテーブル内の量的データに関するデータ属性を含む小さなテーブル。ディメンションテーブル属性は通常、テキストフィールドまたはテキストのように動作する離散数値です。これらの属性は、クエリの制約、フィルタリング、結果セットのラベル付けに一般的に使用されます。

ディザスタ

ワークロードまたはシステムが、導入されている主要な場所でのビジネス目標の達成を妨げるイベント。これらのイベントは、自然災害、技術的障害、または意図しない設定ミスやマルウェア攻撃などの人間の行動の結果である場合があります。

ディザスタリカバリ (DR)

災害によるダウンタイムとデータ損失を最小限に抑えるために使用する戦略とプロセス。詳細については、 AWS Well-Architected フレームワークの「 でのワークロードのディザスタリカバリ AWS: クラウドでのリカバリ」を参照してください。

DML

「データベース操作言語」を参照してください。

ドメイン駆動型設計

各コンポーネントが提供している変化を続けるドメイン、またはコアビジネス目標にコンポーネントを接続して、複雑なソフトウェアシステムを開発するアプローチ。この概念は、エリック·エヴァンスの著書、Domain-Driven Design: Tackling Complexity in the Heart of Software (ドメイン駆動設計:ソフトウエアの中心における複雑さへの取り組み) で紹介されています (ボストン: Addison-Wesley Professional、2003)。strangler fig パターンでドメイン駆動型設計を使用する方法の詳細については、コンテナと Amazon API Gateway を使用して、従来の Microsoft ASP.NET (ASMX) ウェブサービスを段階的にモダナイズ を参照してください。

DR

「ディザスタリカバリ」を参照してください。

ドリフト検出

ベースライン設定からの偏差の追跡。例えば、 AWS CloudFormation を使用してシステムリソース のドリフトを検出したり、 を使用して AWS Control Tower ガバナンス要件への準拠に影響を与える可能性のあるランディングゾーンの変更を検出したりできます。

DVSM

「開発値ストリームマッピング」を参照してください。

# E

EDA

「探索的データ分析」を参照してください。

エッジコンピューティング

IoT ネットワークのエッジにあるスマートデバイスの計算能力を高めるテクノロジー。クラウドコンピューティング と比較すると、エッジコンピューティングは通信レイテンシーを短縮し、応答時間を短縮できます。

暗号化

人間が読み取り可能なプレーンテキストデータを暗号文に変換するコンピューティングプロセス。

暗号化キー

暗号化アルゴリズムが生成した、ランダム化されたビットからなる暗号文字列。キーの長さは決まっておらず、各キーは予測できないように、一意になるように設計されています。

エンディアン

コンピュータメモリにバイトが格納される順序。ビッグエンディアンシステムでは、最上位バイトが最初に格納されます。リトルエンディアンシステムでは、最下位バイトが最初に格納されます。

エンドポイント

「サービスエンドポイント」を参照してください。

エンドポイントサービス

仮想プライベートクラウド (VPC) 内でホストして、他のユーザーと共有できるサービス。を使用してエンドポイントサービスを作成し AWS PrivateLink 、他の AWS アカウント または AWS Identity and Access Management (IAM) プリンシパルにアクセス許可を付与できます。これらのアカウントまたはプリンシパルは、インターフェイス VPC エンドポイントを作成することで、エンドポイントサービスにプライベートに接続できます。詳細については、Amazon Virtual Private Cloud (Amazon VPC) ドキュメントの「エンドポイントサービスを作成する」を参照してください。

エンタープライズリソースプランニング (ERP)

エンタープライズの主要なビジネスプロセス (アカウンティング、MES、プロジェクト管理など) を自動化および管理するシステム。

エンベロープ暗号化

暗号化キーを、別の暗号化キーを使用して暗号化するプロセス。詳細については、 AWS Key Management Service ( AWS KMS) ドキュメントの「エンベロープ暗号化」を参照してください。

環境

実行中のアプリケーションのインスタンス。クラウドコンピューティングにおける一般的な環境の種類は以下のとおりです。

- 開発環境 — アプリケーションのメンテナンスを担当するコアチームのみが使用できる、実行中のアプリケーションのインスタンス。開発環境は、上位の環境に昇格させる変更をテストするときに使用します。このタイプの環境は、テスト環境と呼ばれることもあります。
- 下位環境 — 初期ビルドやテストに使用される環境など、アプリケーションのすべての開発環境。
- 本番環境 — エンドユーザーがアクセスできる、実行中のアプリケーションのインスタンス。CI/CD パイプラインでは、本番環境が最後のデプロイ環境になります。
- 上位環境 — コア開発チーム以外のユーザーがアクセスできるすべての環境。これには、本番環境、本番前環境、ユーザー承認テスト環境などが含まれます。

エピック

アジャイル方法論で、お客様の作業の整理と優先順位付けに役立つ機能カテゴリ。エピックでは、要件と実装タスクの概要についてハイレベルな説明を提供します。例えば、 AWS CAF セキュリティエピックには、ID とアクセスの管理、検出コントロール、インフラストラクチャセキュリティ、データ保護、インシデント対応が含まれます。 AWS 移行戦略のエピックの詳細については、プログラム実装ガイド を参照してください。

ERP

「エンタープライズリソース計画」を参照してください。

探索的データ分析 (EDA)

データセットを分析してその主な特性を理解するプロセス。お客様は、データを収集または集計してから、パターンの検出、異常の検出、および前提条件のチェックのための初期調査を実行します。EDA は、統計の概要を計算し、データの可視化を作成することによって実行されます。

# F

ファクトテーブル

スタースキーマ の中央テーブル。事業運営に関する定量的データを保存します。通常、ファクトテーブルには、メジャーを含む列とディメンションテーブルへの外部キーを含む列の 2 種類の列が含まれます。

フェイルファスト

頻繁で段階的なテストを使用して開発ライフサイクルを短縮する哲学。これはアジャイルアプローチの重要な部分です。

障害分離境界

では AWS クラウド、障害の影響を制限し AWS リージョン、ワークロードの耐障害性を向上させるアベイラビリティーゾーン、コントロールプレーン、データプレーンなどの境界です。詳細については、AWS 「障害分離境界」を参照してください。

機能ブランチ

「ブランチ」を参照してください。

特徴量

お客様が予測に使用する入力データ。例えば、製造コンテキストでは、特徴量は製造ラインから定期的にキャプチャされるイメージの可能性もあります。

特徴量重要度

モデルの予測に対する特徴量の重要性。これは通常、Shapley Additive Deskonations (SHAP) や積分勾配など、さまざまな手法で計算できる数値スコアで表されます。詳細については、「 を使用した機械学習モデルの解釈可能性：AWS」を参照してください。

機能変換

追加のソースによるデータのエンリッチ化、値のスケーリング、単一のデータフィールドからの複数の情報セットの抽出など、機械学習プロセスのデータを最適化すること。これにより、機械学習モデルはデータの恩恵を受けることができます。例えば、「2021-05-27 00:15:37」の日付を「2021 年」、「5 月」、「木」、「15」に分解すると、学習アルゴリズムがさまざまなデータコンポーネントに関連する微妙に異なるパターンを学習するのに役立ちます。

FGAC

「きめ細かなアクセスコントロール」を参照してください。

きめ細かなアクセス制御 (FGAC)

複数の条件を使用してアクセス要求を許可または拒否すること。

フラッシュカット移行

段階的なアプローチを使用するのではなく、変更データキャプチャによる継続的なデータレプリケーションを使用して、可能な限り短時間でデータを移行するデータベース移行方法。目的はダウンタイムを最小限に抑えることです。

# G

ジオブロッキング

「地理的制限」を参照してください。

地理的制限 (ジオブロッキング)

Amazon では CloudFront、特定の国のユーザーがコンテンツディストリビューションにアクセスできないようにするオプションです。アクセスを許可する国と禁止する国は、許可リストまたは禁止リストを使って指定します。詳細については、 CloudFront ドキュメントの「コンテンツの地理的ディストリビューションの制限」を参照してください。

Gitflow ワークフロー

下位環境と上位環境が、ソースコードリポジトリでそれぞれ異なるブランチを使用する方法。Gitflow ワークフローはレガシーと見なされ、トランクベースのワークフローはモダンで推奨されるアプローチです。

グリーンフィールド戦略

新しい環境に既存のインフラストラクチャが存在しないこと。システムアーキテクチャにグリーンフィールド戦略を導入する場合、既存のインフラストラクチャ (別名ブラウンフィールド) との互換性の制約を受けることなく、あらゆる新しいテクノロジーを選択できます。既存のインフラストラクチャを拡張している場合は、ブラウンフィールド戦略とグリーンフィールド戦略を融合させることもできます。

ガードレール

組織単位 (OU) 全般のリソース、ポリシー、コンプライアンスを管理するのに役立つ概略的なルール。予防ガードレールは、コンプライアンス基準に一致するようにポリシーを実施します。これらは、サービスコントロールポリシーと IAM アクセス許可の境界を使用して実装

されます。検出ガードレールは、ポリシー違反やコンプライアンス上の問題を検出し、修復のためのアラートを発信します。これらは、 AWS Config、、Amazon AWS Security Hub、GuardDuty、Amazon Inspector AWS Trusted Advisor、およびカスタム AWS Lambda チェックを使用して実装されます。

# H

HA

「高可用性」を参照してください。

異種混在データベースの移行

別のデータベースエンジンを使用するターゲットデータベースへお客様の出典データベースの移行 (例えば、Oracle から Amazon Aurora)。異種間移行は通常、アーキテクチャの再設計作業の一部であり、スキーマの変換は複雑なタスクになる可能性があります。AWS は、スキーマの変換に役立つ AWS SCTを提供します。

ハイアベイラビリティ (HA)

課題や災害が発生した場合に、介入なしにワークロードを継続的に運用できること。HA システムは、自動的にフェイルオーバーし、一貫して高品質のパフォーマンスを提供し、パフォーマンスへの影響を最小限に抑えながらさまざまな負荷や障害を処理するように設計されています。

ヒストリアンのモダナイゼーション

製造業のニーズによりよく応えるために、オペレーションテクノロジー (OT) システムをモダナイズし、アップグレードするためのアプローチ。ヒストリアンは、工場内のさまざまなソースからデータを収集して保存するために使用されるデータベースの一種です。

同種データベースの移行

お客様の出典データベースを、同じデータベースエンジンを共有するターゲットデータベース (Microsoft SQL Server から Amazon RDS for SQL Server など) に移行する。同種間移行は、通常、リホストまたはリプラットフォーム化の作業の一部です。ネイティブデータベースユーティリティを使用して、スキーマを移行できます。

ホットデータ

リアルタイムデータや最近の翻訳データなど、頻繁にアクセスされるデータ。通常、このデータには高速なクエリ応答を提供する高性能なストレージ階層またはクラスが必要です。

ホットフィックス

本番環境の重大な問題を修正するために緊急で配布されるプログラム。緊急性のため、通常、修正は一般的な DevOps リリースワークフローの外で行われます。

ハイパーケア期間

カットオーバー直後、移行したアプリケーションを移行チームがクラウドで管理、監視して問題に対処する期間。通常、この期間は 1～4 日です。ハイパーケア期間が終了すると、アプリケーションに対する責任は一般的に移行チームからクラウドオペレーションチームに移ります。

## I

IaC

「Infrastructure as Code」を参照してください。

ID ベースのポリシー

AWS クラウド 環境内のアクセス許可を定義する 1 つ以上の IAM プリンシパルにアタッチされたポリシー。

アイドル状態のアプリケーション

90 日間の平均的な CPU およびメモリ使用率が 5～20% のアプリケーション。移行プロジェクトでは、これらのアプリケーションを廃止するか、オンプレミスに保持するのが一般的です。

IIoT

「産業モノのインターネット」を参照してください。

イミュータブルインフラストラクチャ

既存のインフラストラクチャを更新、パッチ適用、または変更する代わりに、本番ワークロード用の新しいインフラストラクチャをデプロイするモデル。イミュータブルなインフラストラクチャは、本質的にミュータブルなインフラストラクチャ よりも一貫性、信頼性、予測性が高くなります。詳細については、 AWS Well-Architected フレームワークの「変更不可能なインフラストラクチャを使用したデプロイ」のベストプラクティスを参照してください。

インバウンド (受信) VPC

AWS マルチアカウントアーキテクチャでは、アプリケーションの外部からネットワーク接続を受け入れ、検査し、ルーティングする VPC。AWS Security Reference Architecture では、アプリ

ケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

増分移行

アプリケーションを 1 回ですべてカットオーバーするのではなく、小さい要素に分けて移行するカットオーバー戦略。例えば、最初は少数のマイクロサービスまたはユーザーのみを新しいシステムに移行する場合があります。すべてが正常に機能することを確認できたら、残りのマイクロサービスやユーザーを段階的に移行し、レガシーシステムを廃止できるようにします。この戦略により、大規模な移行に伴うリスクが軽減されます。

インダストリー 4.0

接続、リアルタイムデータ、自動化、分析、AI/ML の進歩を通じて、 のビジネスプロセスのモダナイゼーションを指すために 2016 年に Klaus Schwab によって導入された用語。

インフラストラクチャ

アプリケーションの環境に含まれるすべてのリソースとアセット。

Infrastructure as Code (IaC)

アプリケーションのインフラストラクチャを一連の設定ファイルを使用してプロビジョニングし、管理するプロセス。IaC は、新しい環境を再現可能で信頼性が高く、一貫性のあるものにするため、インフラストラクチャを一元的に管理し、リソースを標準化し、スケールを迅速に行えるように設計されています。

産業分野における IoT (IIoT)

製造、エネルギー、自動車、ヘルスケア、ライフサイエンス、農業などの産業部門におけるインターネットに接続されたセンサーやデバイスの使用。詳細については、「Building an industrial Internet of Things (IIoT) digital transformation strategy」を参照してください。

インスペクション VPC

AWS マルチアカウントアーキテクチャでは、VPC (同一または異なる 内 AWS リージョン ) 、インターネット、オンプレミスネットワーク間のネットワークトラフィックの検査を管理する一元化された VPCs。AWS Security Reference Architecture では、アプリケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

IoT

インターネットまたはローカル通信ネットワークを介して他のデバイスやシステムと通信する、センサーまたはプロセッサが組み込まれた接続済み物理オブジェクトのネットワーク。詳細については、「IoT とは」を参照してください。

解釈可能性

機械学習モデルの特性で、モデルの予測がその入力にどのように依存するかを人間が理解できる度合いを表します。詳細については、「AWS を使用した機械学習モデルの解釈」を参照してください。

IoT

「モノのインターネット」を参照してください。

IT 情報ライブラリ (ITIL)

IT サービスを提供し、これらのサービスをビジネス要件に合わせるための一連のベストプラクティス。ITIL は ITSM の基盤を提供します。

IT サービス管理 (ITSM)

組織の IT サービスの設計、実装、管理、およびサポートに関連する活動。クラウドオペレーションと ITSM ツールの統合については、オペレーション統合ガイド を参照してください。

ITIL

「IT 情報ライブラリ」を参照してください。

ITSM

「IT サービス管理」を参照してください。

# L

ラベルベースアクセス制御 (LBAC)

強制アクセス制御 (MAC) の実装で、ユーザーとデータ自体にそれぞれセキュリティラベル値が明示的に割り当てられます。ユーザーセキュリティラベルとデータセキュリティラベルが交差する部分によって、ユーザーに表示される行と列が決まります。

ランディングゾーン

ランディングゾーンは、スケーラブルで安全な、適切に設計されたマルチアカウント AWS 環境です。これは、組織がセキュリティおよびインフラストラクチャ環境に自信を持ってワークロー

ドとアプリケーションを迅速に起動してデプロイできる出発点です。ランディングゾーンの詳細については、安全でスケーラブルなマルチアカウント AWS 環境のセットアップ を参照してください。

大規模な移行

300 台以上のサーバの移行。

LBAC

「ラベルベースのアクセスコントロール」を参照してください。

最小特権

タスクの実行には必要最低限の権限を付与するという、セキュリティのベストプラクティス。詳細については、IAM ドキュメントの最小特権アクセス許可を適用するを参照してください。

リフトアンドシフト

「7 R」を参照してください。

リトルエンディアンシステム

最下位バイトを最初に格納するシステム。エンディアンネス も参照してください。

下位環境

「環境」を参照してください。

## M

機械学習 (ML)

パターン認識と学習にアルゴリズムと手法を使用する人工知能の一種。ML は、モノのインターネット (IoT) データなどの記録されたデータを分析して学習し、パターンに基づく統計モデルを生成します。詳細については、「機械学習」を参照してください。

メインブランチ

「ブランチ」を参照してください。

マルウェア

コンピュータのセキュリティまたはプライバシーを侵害するように設計されているソフトウェア。マルウェアは、コンピュータシステムの中断、機密情報の漏洩、不正アクセスにつながる

可能性があります。マルウェアの例としては、ウイルス、ワーム、ランサムウェア、トロイの木馬、スパイウェア、キーロガーなどがあります。

マネージドサービス

AWS サービス がインフラストラクチャレイヤー、オペレーティングシステム、プラットフォーム AWS を運用し、ユーザーがエンドポイントにアクセスしてデータを保存および取得します。Amazon Simple Storage Service (Amazon S3) と Amazon DynamoDB は、 マネージドサービスの例です。これらは抽象化されたサービス とも呼ばれます。

製造実行システム (MES)

生産プロセスを追跡、モニタリング、文書化、制御するためのソフトウェアシステム。これにより、加工品を現場の完成製品に変換します。

MAP

「移行促進プログラム」を参照してください。

メカニズム

ツールを作成し、ツールの導入を推進し、調整のために結果を検査する完全なプロセス。メカニズムとは、動作中にそれ自体を強化して改善するサイクルです。詳細については、 AWS Well-Architected フレームワークの「メカニズムの構築」を参照してください。

メンバーアカウント

の組織の一部である管理アカウント AWS アカウント を除くすべての AWS Organizations。アカウントが組織のメンバーになることができるのは、一度に 1 つのみです。

MES

「製造実行システム」を参照してください。

メッセージキューイングテレメトリトランスポート (MQTT)

リソースに制約のある IoT デバイス用の、パブリッシュ/サブスクライブパターンに基づく軽量の machine-to-machine (M2M) 通信プロトコル。

マイクロサービス

明確に定義された API を介して通信し、通常は小規模な自己完結型のチームが所有する、小規模で独立したサービスです。例えば、保険システムには、販売やマーケティングなどのビジネス機能、または購買、請求、分析などのサブドメインにマッピングするマイクロサービスが含まれる場合があります。マイクロサービスの利点には、俊敏性、柔軟なスケーリング、容易なデプロ

イ、再利用可能なコード、回復力などがあります。詳細については、AWS 「サーバーレスサービスを使用したマイクロサービスの統合」を参照してください。

マイクロサービスアーキテクチャ

各アプリケーションプロセスをマイクロサービスとして実行する独立したコンポーネントを使用してアプリケーションを構築するアプローチ。これらのマイクロサービスは、軽量 API を使用して、明確に定義されたインターフェイスを介して通信します。このアーキテクチャの各マイクロサービスは、アプリケーションの特定の機能に対する需要を満たすように更新、デプロイ、およびスケーリングできます。詳細については、「 でのマイクロサービスの実装 AWS」を参照してください。

Migration Acceleration Program (MAP)

コンサルティングサポート、トレーニング、サービスを提供する AWS プログラム。組織がクラウドへの移行のための強固な運用基盤を構築し、移行の初期コストを相殺するのに役立ちます。MAP には、組織的な方法でレガシー移行を実行するための移行方法論と、一般的な移行シナリオを自動化および高速化する一連のツールが含まれています。

大規模な移行

アプリケーションポートフォリオの大部分を次々にクラウドに移行し、各ウェーブでより多くのアプリケーションを高速に移動させるプロセス。この段階では、以前の段階から学んだベストプラクティスと教訓を使用して、移行ファクトリー チーム、ツール、プロセスのうち、オートメーションとアジャイルデリバリーによってワークロードの移行を合理化します。これは、AWS 移行戦略 の第 3 段階です。

移行ファクトリー

自動化された俊敏性のあるアプローチにより、ワークロードの移行を合理化する部門横断的なチーム。移行ファクトリーチームには、通常、オペレーション、ビジネスアナリストと所有者、移行エンジニア、デベロッパー、スプリントに取り組む DevOps プロフェッショナルが含まれます。エンタープライズアプリケーションポートフォリオの 20～50% は、ファクトリーのアプローチによって最適化できる反復パターンで構成されています。詳細については、このコンテンツセットの移行ファクトリーに関する解説とCloud Migration Factory ガイドを参照してください。

移行メタデータ

移行を完了するために必要なアプリケーションおよびサーバーに関する情報。移行パターンごとに、異なる一連の移行メタデータが必要です。移行メタデータの例には、ターゲットサブネット、セキュリティグループ、 AWS アカウントなどがあります。

移行パターン

移行戦略、移行先、および使用する移行アプリケーションまたはサービスを詳述する、反復可能な移行タスク。例: Application Migration Service を使用して Amazon EC2 AWS への移行をリホストします。

Migration Portfolio Assessment (MPA)

に移行するためのビジネスケースを検証するための情報を提供するオンラインツール AWS クラウド。MPA は、詳細なポートフォリオ評価 (サーバーの適切なサイジング、価格設定、TCO 比較、移行コスト分析) および移行プラン (アプリケーションデータの分析とデータ収集、アプリケーションのグループ化、移行の優先順位付け、およびウェーブプランニング) を提供します。MPA ツール (ログインが必要) は、すべての AWS コンサルタントと APN パートナーコンサルタントが無料で利用できます。

移行準備状況評価 (MRA)

AWS CAF を使用して、組織のクラウド準備状況に関するインサイトを取得し、長所と短所を特定し、特定されたギャップを埋めるためのアクションプランを構築するプロセス。詳細については、移行準備状況ガイド を参照してください。MRA は、AWS 移行戦略の第一段階です。

移行戦略

ワークロードを に移行するために使用されるアプローチ AWS クラウド。詳細については、この用語集の「7 Rs エントリ」と「組織を動員して大規模な移行を加速する」を参照してください。

ML

「機械学習」を参照してください。

モダナイゼーション

古い (レガシーまたはモノリシック) アプリケーションとそのインフラストラクチャをクラウド内の俊敏で弾力性のある高可用性システムに変換して、コストを削減し、効率を高め、イノベーションを活用します。詳細については、「」の「アプリケーションをモダナイズするための戦略 AWS クラウド」を参照してください。

モダナイゼーション準備状況評価

組織のアプリケーションのモダナイゼーションの準備状況を判断し、利点、リスク、依存関係を特定し、組織がこれらのアプリケーションの将来の状態をどの程度適切にサポートできるかを決定するのに役立つ評価。評価の結果として、ターゲットアーキテクチャのブループリント、モダナイゼーションプロセスの開発段階とマイルストーンを詳述したロードマップ、特定され

たギャップに対処するためのアクションプランが得られます。詳細については、「」の「アプリケーションのモダナイゼーション準備状況の評価 AWS クラウド」を参照してください。

モノリシックアプリケーション (モノリス)

緊密に結合されたプロセスを持つ単一のサービスとして実行されるアプリケーション。モノリシックアプリケーションにはいくつかの欠点があります。1 つのアプリケーション機能エクスペリエンスの需要が急増する場合は、アーキテクチャ全体をスケーリングする必要があります。モノリシックアプリケーションの特徴を追加または改善することは、コードベースが大きくなると複雑になります。これらの問題に対処するには、マイクロサービスアーキテクチャを使用できます。詳細については、モノリスをマイクロサービスに分解する を参照してください。

MPA

「移行ポートフォリオ評価」を参照してください。

MQTT

「Message Queuing Telemetry Transport」を参照してください。

多クラス分類

複数のクラスの予測を生成するプロセス (2 つ以上の結果の 1 つを予測します)。例えば、機械学習モデルが、「この製品は書籍、自動車、電話のいずれですか?」 または、「このお客様にとって最も関心のある商品のカテゴリはどれですか?」と聞くかもしれません。

変更可能なインフラストラクチャ

本番ワークロードの既存のインフラストラクチャを更新および変更するモデル。Well-Architected AWS Framework では、一貫性、信頼性、予測可能性を向上させるために、イミュータブルなインフラストラクチャの使用をベストプラクティスとして推奨しています。

# O

OAC

「オリジンアクセスコントロール」を参照してください。

OAI

「オリジンアクセスアイデンティティ」を参照してください。

OCM

「組織変更管理」を参照してください。

オフライン移行

移行プロセス中にソースワークロードを停止させる移行方法。この方法はダウンタイムが長くなるため、通常は重要ではない小規模なワークロードに使用されます。

OI

「 オペレーション統合」を参照してください。

OLA

「運用レベルの契約」を参照してください。

オンライン移行

ソースワークロードをオフラインにせずにターゲットシステムにコピーする移行方法。ワークロードに接続されているアプリケーションは、移行中も動作し続けることができます。この方法はダウンタイムがゼロから最小限で済むため、通常は重要な本番稼働環境のワークロードに使用されます。

OPC-UA

「Open Process Communications - Unified Architecture」を参照してください。

オープンプロセス通信 - 統合アーキテクチャ (OPC-UA)

産業用オートメーション用の machine-to-machine (M2M) 通信プロトコル。OPC-UA は、データの暗号化、認証、認可スキームを備えた相互運用性標準を提供します。

オペレーショナルレベルアグリーメント (OLA)

サービスレベルアグリーメント (SLA) をサポートするために、どの機能的 IT グループが互いに提供することを約束するかを明確にする契約。

運用準備状況レビュー (ORR)

インシデントや潜在的な障害の理解、評価、防止、または範囲の縮小に役立つ質問とそれに関連するベストプラクティスのチェックリスト。詳細については、 AWS Well-Architected フレームワークの「運用準備状況レビュー (ORR)」を参照してください。

運用テクノロジー (OT)

産業運用、機器、インフラストラクチャを制御するために物理環境と連携するハードウェアおよびソフトウェアシステム。製造では、OT と情報技術 (IT) システムの統合が、Industry 4.0 トランスフォーメーションの主要な焦点です。

オペレーション統合 (OI)

クラウドでオペレーションをモダナイズするプロセスには、準備計画、オートメーション、統合が含まれます。詳細については、オペレーション統合ガイド を参照してください。

組織の証跡

の組織 AWS アカウント 内のすべての のすべてのイベントをログ AWS CloudTrail に記録する によって作成された証跡 AWS Organizations。証跡は、組織に含まれている各 AWS アカウント に作成され、各アカウントのアクティビティを追跡します。詳細については、 ドキュメントの「組織の証跡の作成」を参照してください。 CloudTrail

組織変更管理 (OCM)

人材、文化、リーダーシップの観点から、主要な破壊的なビジネス変革を管理するためのフレームワーク。OCM は、変化の導入を加速し、移行問題に対処し、文化や組織の変化を推進することで、組織が新しいシステムと戦略の準備と移行するのを支援します。 AWS 移行戦略では、クラウド導入プロジェクトに必要な変化のスピードから、このフレームワークは人材アクセラレーション と呼ばれます。詳細については、OCM ガイド を参照してください。

オリジンアクセスコントロール (OAC)

では CloudFront、Amazon Simple Storage Service (Amazon S3) コンテンツを保護するためのアクセスを制限するための拡張オプションです。OAC は、すべての のすべての S3 バケット AWS リージョン、 AWS KMS (SSE-KMS) によるサーバー側の暗号化、S3 バケットへの動的 PUTおよび DELETEリクエストをサポートします。

オリジンアクセスアイデンティティ (OAI)

では CloudFront、Amazon S3 コンテンツを保護するためのアクセスを制限するオプションです。OAI を使用すると、 は Amazon S3 が認証できるプリンシパル CloudFront を作成します。認証されたプリンシパルは、特定の CloudFront ディストリビューションを介してのみ S3 バケット内のコンテンツにアクセスできます。OACも併せて参照してください。OAC では、より詳細な、強化されたアクセスコントロールが可能です。

ORR

「運用準備状況レビュー」を参照してください。

OT

「運用技術」を参照してください。

アウトバウンド (送信) VPC

AWS マルチアカウントアーキテクチャでは、アプリケーション内から開始されるネットワーク接続を処理する VPC。AWS Security Reference Architecture では、アプリケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

# P

アクセス許可の境界

ユーザーまたはロールが使用できるアクセス許可の上限を設定する、IAM プリンシパルにアタッチされる IAM 管理ポリシー。詳細については、IAM ドキュメントのアクセス許可の境界を参照してください。

個人を特定できる情報 (PII)

直接閲覧した場合、または他の関連データと組み合わせた場合に、個人の身元を合理的に推測するために使用できる情報。PII の例には、氏名、住所、連絡先情報などがあります。

PII

個人を特定できる情報を参照してください。

プレイブック

クラウドでのコアオペレーション機能の提供など、移行に関連する作業を取り込む、事前定義された一連のステップ。プレイブックは、スクリプト、自動ランブック、またはお客様のモダナイズされた環境を運用するために必要なプロセスや手順の要約などの形式をとることができます。

PLC

「プログラム可能なロジックコントローラー」を参照してください。

PLM

「製品ライフサイクル管理」を参照してください。

ポリシー

アクセス許可の定義 (アイデンティティベースのポリシー を参照）、アクセス条件の指定 (リソースベースのポリシー を参照）、または の組織内のすべてのアカウントに対する最大アクセス許可の定義 AWS Organizations （ サービスコントロールポリシー を参照) が可能なオブジェクト。

多言語の永続性

データアクセスパターンやその他の要件に基づいて、マイクロサービスのデータストレージテクノロジーを個別に選択します。マイクロサービスが同じデータストレージテクノロジーを使用している場合、実装上の問題が発生したり、パフォーマンスが低下する可能性があります。マイクロサービスは、要件に最も適合したデータストアを使用すると、より簡単に実装でき、パフォーマンスとスケーラビリティが向上します。詳細については、マイクロサービスでのデータ永続性の有効化 を参照してください。

ポートフォリオ評価

移行を計画するために、アプリケーションポートフォリオの検出、分析、優先順位付けを行うプロセス。詳細については、「移行準備状況ガイド」を参照してください。

述語

`true` または を返すクエリ条件。`false`通常は `WHERE`句にあります。

述語プッシュダウン

転送前にクエリ内のデータをフィルタリングするデータベースクエリ最適化手法。これにより、リレーショナルデータベースから取得して処理する必要があるデータの量が減少し、クエリのパフォーマンスが向上します。

予防的コントロール

イベントの発生を防ぐように設計されたセキュリティコントロール。このコントロールは、ネットワークへの不正アクセスや好ましくない変更を防ぐ最前線の防御です。詳細については、Implementing security controls on AWSのPreventative controlsを参照してください。

プリンシパル

アクションを実行し AWS 、リソースにアクセスできる のエンティティ。このエンティティは通常、、IAM ロール AWS アカウント、またはユーザーのルートユーザーです。詳細については、IAM ドキュメントのロールに関する用語と概念内にあるプリンシパルを参照してください。

プライバシーバイデザイン

エンジニアリングプロセス全体を通してプライバシーを考慮に入れたシステムエンジニアリングのアプローチ。

プライベートホストゾーン

1 つ以上の VPC 内のドメインとそのサブドメインへの DNS クエリに対し、Amazon Route 53 がどのように応答するかに関する情報を保持するコンテナ。詳細については、Route 53 ドキュメントの「プライベートホストゾーンの使用」を参照してください。

プロアクティブコントロール

非準拠のリソースのデプロイを防止するように設計されたセキュリティコントロール。これらのコントロールは、プロビジョニング前にリソースをスキャンします。リソースがコントロールに準拠していない場合、プロビジョニングされません。詳細については、 AWS Control Tower ドキュメントの「 コントロールリファレンスガイド」および「 でのセキュリティコントロールの実装」の「プロアクティブコントロール」を参照してください。 AWS

製品ライフサイクル管理 (PLM)

設計、開発、発売から成長と成熟まで、製品のデータとプロセスのライフサイクル全体にわたる管理。

本番環境

「環境」を参照してください。

プログラミング可能ロジックコントローラー (NAL)

製造では、マシンをモニタリングし、承認プロセスを自動化する、信頼性が高く適応可能なコンピュータです。

仮名化

データセット内の個人識別子をプレースホルダー値に置き換えるプロセス。仮名化は個人のプライバシー保護に役立ちます。仮名化されたデータは、依然として個人データとみなされます。

パブリッシュ/サブスクライブ (pub/sub)

マイクロサービス間の非同期通信を可能にするパターン。スケーラビリティと応答性を向上させます。例えば、マイクロサービスベースの MES では、マイクロサービスは他のマイクロサービスがサブスクライブできるチャネルにイベントメッセージを発行できます。システムは、公開サービスを変更せずに新しいマイクロサービスを追加できます。

# Q

クエリプラン

SQL リレーショナルデータベースシステムのデータにアクセスするために使用される手順などの一連のステップ。

クエリプランのリグレッション

データベースサービスのオプティマイザーが、データベース環境に特定の変更が加えられる前に選択されたプランよりも最適性の低いプランを選択すること。これは、統計、制限事項、環境設

定、クエリパラメータのバインディングの変更、およびデータベースエンジンの更新などが原因である可能性があります。

## R

RACI マトリックス

責任、説明責任、相談、通知 (RACI) を参照してください。

ランサムウェア

決済が完了するまでコンピュータシステムまたはデータへのアクセスをブロックするように設計された、悪意のあるソフトウェア。

RASCI マトリックス

責任、説明責任、相談、通知 (RACI) を参照してください。

RCAC

「行と列のアクセスコントロール」を参照してください。

リードレプリカ

読み取り専用に使用されるデータベースのコピー。クエリをリードレプリカにルーティングして、プライマリデータベースへの負荷を軽減できます。

再構築

「7 Rs」を参照してください。

目標復旧時点 (RPO)

最後のデータリカバリポイントからの最大許容時間です。これにより、最後の回復時点からサービスが中断されるまでの間に許容できるデータ損失の程度が決まります。

目標復旧時間 (RTO)

サービスの中断から復旧までの最大許容遅延時間。

リファクタリング

「7 Rs」を参照してください。

リージョン

地理的エリア内の AWS リソースのコレクション。各 AWS リージョン は、耐障害性、安定性、耐障害性を提供するために、他の とは分離され、独立しています。詳細については、AWS リージョン 「 を使用できるアカウントを指定する」を参照してください。

回帰

数値を予測する機械学習手法。例えば、「この家はどれくらいの値段で売れるでしょうか?」という問題を解決するために、機械学習モデルは、線形回帰モデルを使用して、この家に関する既知の事実 (平方フィートなど) に基づいて家の販売価格を予測できます。

リホスト

「7 Rs」を参照してください。

リリース

デプロイプロセスで、変更を本番環境に昇格させること。

再配置

「7 R」を参照してください。

プラットフォーム変更

「7 R」を参照してください。

再購入

「7 R」を参照してください。

回復性

中断に耐えたり、中断から回復したりするアプリケーションの機能。で障害耐性を計画する場合、高可用性とディザスタリカバリが一般的な考慮事項です AWS クラウド。詳細については、AWS クラウド 「レジリエンス」を参照してください。

リソースベースのポリシー

Amazon S3 バケット、エンドポイント、暗号化キーなどのリソースにアタッチされたポリシー。このタイプのポリシーは、アクセスが許可されているプリンシパル、サポートされているアクション、その他の満たすべき条件を指定します。

実行責任者、説明責任者、協業先、報告先 (RACI) に基づくマトリックス

移行活動とクラウド運用に関わるすべての関係者の役割と責任を定義したマトリックス。マトリックスの名前は、マトリックスで定義されている責任の種類、すなわち責任 (R) 、説明責任

(A) 、協議 (C) 、情報提供 (I) に由来します。サポート (S) タイプはオプションです。サポートを含めると、そのマトリックスは RASCI マトリックスと呼ばれ、サポートを除外すると RACI マトリックスと呼ばれます。

レスポンシブコントロール

有害事象やセキュリティベースラインからの逸脱について、修復を促すように設計されたセキュリティコントロール。詳細については、Implementing security controls on AWSのResponsive controlsを参照してください。

保持

「7 R」を参照してください。

廃止

「7 Rs」を参照してください。

ローテーション

攻撃者が認証情報にアクセスすることをより困難にするために、シークレットを定期的に更新するプロセス。

行と列のアクセス制御 (RCAC)

アクセスルールが定義された、基本的で柔軟な SQL 表現の使用。RCAC は行権限と列マスクで構成されています。

RPO

「目標復旧時点」を参照してください。

RTO

「目標復旧時間」を参照してください。

ランブック

特定のタスクを実行するために必要な手動または自動化された一連の手順。これらは通常、エラー率の高い反復操作や手順を合理化するために構築されています。

# S

SAML 2.0

多くの ID プロバイダー (IdPs) が使用するオープンスタンダード。この機能により、フェデレーティッドシングルサインオン (SSO) が有効になるため、ユーザーは にログイン AWS

Management Console したり、組織内のすべてのユーザーを IAM で作成しなくても AWS API オペレーションを呼び出すことができます。SAML 2.0 ベースのフェデレーションの詳細については、IAM ドキュメントのSAML 2.0 ベースのフェデレーションについてを参照してください。

SCADA

「監視コントロールとデータ収集」を参照してください。

SCP

「サービスコントロールポリシー」を参照してください。

シークレット

では AWS Secrets Manager、暗号化された形式で保存するパスワードやユーザー認証情報などの機密情報または制限付き情報。シークレット値とそのメタデータで構成されます。シークレット値は、バイナリ、1 つの文字列、または複数の文字列にすることができます。詳細については、Secrets Manager ドキュメントの「Secrets Manager シークレットの内容」を参照してください。

セキュリティコントロール

脅威アクターによるセキュリティ脆弱性の悪用を防止、検出、軽減するための、技術上または管理上のガードレール。セキュリティコントロールには、予防的 、検出的 、???応答的 、プロアクティブ の 4 つの主なタイプがあります。

セキュリティ強化

アタックサーフェスを狭めて攻撃への耐性を高めるプロセス。このプロセスには、不要になったリソースの削除、最小特権を付与するセキュリティのベストプラクティスの実装、設定ファイル内の不要な機能の無効化、といったアクションが含まれています。

Security Information and Event Management (SIEM) システム

セキュリティ情報管理 (SIM) とセキュリティイベント管理 (SEM) のシステムを組み合わせたツールとサービス。SIEM システムは、サーバー、ネットワーク、デバイス、その他ソースからデータを収集、モニタリング、分析して、脅威やセキュリティ違反を検出し、アラートを発信します。

セキュリティレスポンスの自動化

セキュリティイベントに自動的に応答または修正するように設計された、事前定義されたプログラムされたアクション。これらの自動化は、セキュリティのベストプラクティスの実装に役立つ検出的または応答的な AWS セキュリティコントロールとして機能します。自動レスポンスア

クションの例としては、VPC セキュリティグループの変更、Amazon EC2 インスタンスへのパッチ適用、認証情報のローテーションなどがあります。

サーバー側の暗号化

送信先にあるデータの、それを受け取る AWS サービス による暗号化。

サービスコントロールポリシー (SCP)

AWS Organizationsの組織内の、すべてのアカウントのアクセス許可を一元的に管理するポリシー。SCP は、管理者がユーザーまたはロールに委任するアクションに、ガードレールを定義したり、アクションの制限を設定したりします。SCP は、許可リストまたは拒否リストとして、許可または禁止するサービスやアクションを指定する際に使用できます。詳細については、 AWS Organizations ドキュメントの「サービスコントロールポリシー」を参照してください。

サービスエンドポイント

のエントリポイントの URL AWS サービス。ターゲットサービスにプログラムで接続するには、エンドポイントを使用します。詳細については、AWS 全般のリファレンスの「AWS サービス エンドポイント」を参照してください。

サービスレベルアグリーメント (SLA)

サービスのアップタイムやパフォーマンスなど、IT チームがお客様に提供すると約束したものを明示した合意書。

サービスレベルインジケータ (SLI)

エラー率、可用性、スループットなど、サービスのパフォーマンス側面の測定。

サービスレベルの目標 (SLO)

サービスレベルのインジケータ によって測定される、サービスの状態を表すターゲットメトリクス。

責任共有モデル

クラウドのセキュリティとコンプライアンス AWS について と共有する責任を説明するモデル。AWS はクラウドのセキュリティを担当しますが、お客様はクラウドのセキュリティを担当します。詳細については、責任共有モデルを参照してください。

SIEM

「セキュリティ情報とイベント管理システム」を参照してください。

単一障害点 (SPOF)

システムを中断させる可能性のあるアプリケーションの単一の重要なコンポーネントの障害。

SLA

「サービスレベルアグリーメント」を参照してください。

SLI

「サービスレベルインジケータ」を参照してください。

SLO

「サービスレベルの目標」を参照してください。

split-and-seed モデル

モダナイゼーションプロジェクトのスケーリングと加速のためのパターン。新機能と製品リリースが定義されると、コアチームは解放されて新しい製品チームを作成します。これにより、お客様の組織の能力とサービスの拡張、デベロッパーの生産性の向上、迅速なイノベーションのサポートに役立ちます。詳細については、「」の「アプリケーションをモダナイズするための段階的アプローチ AWS クラウド」を参照してください。

SPOF

単一障害点 を参照してください。

star スキーマ

トランザクションデータまたは測定データを保存するために 1 つの大きなファクトテーブルを使用し、データ属性を保存するために 1 つ以上の小さなディメンションテーブルを使用するデータベースの組織構造。この構造は、データウェアハウスまたはビジネスインテリジェンスの目的で使用するように設計されています。

strangler fig パターン

レガシーシステムが廃止されるまで、システム機能を段階的に書き換えて置き換えることにより、モノリシックシステムをモダナイズするアプローチ。このパターンは、宿主の樹木から根を成長させ、最終的にその宿主を包み込み、宿主に取って代わるイチジクのつるを例えています。そのパターンは、モノリシックシステムを書き換えるときのリスクを管理する方法として Martin Fowler により提唱されました。このパターンの適用方法の例については、コンテナと Amazon API Gateway を使用して、従来の Microsoft ASP.NET (ASMX) ウェブサービスを段階的にモダナイズを参照してください。

サブネット

VPC 内の IP アドレスの範囲。サブネットは、1 つのアベイラビリティーゾーンに存在する必要があります。

監視統制とデータ収集 (SCADA)

製造では、ハードウェアとソフトウェアを使用して物理アセットと生産オペレーションをモニタリングするシステム。

対称暗号化

データの暗号化と復号に同じキーを使用する暗号化のアルゴリズム。

合成テスト

ユーザーインタラクションをシミュレートして潜在的な問題を検出したり、パフォーマンスをモニタリングしたりする方法でシステムをテストします。Amazon CloudWatch Synthetics を使用してこれらのテストを作成できます。

# T

タグ

AWS リソースを整理するためのメタデータとして機能するキーと値のペア。タグは、リソースの管理、識別、整理、検索、フィルタリングに役立ちます。詳細については、「AWS リソースのタグ付け」を参照してください。

ターゲット変数

監督された機械学習でお客様が予測しようとしている値。これは、結果変数 のことも指します。例えば、製造設定では、ターゲット変数が製品の欠陥である可能性があります。

タスクリスト

ランブックの進行状況を追跡するために使用されるツール。タスクリストには、ランブックの概要と完了する必要のある一般的なタスクのリストが含まれています。各一般的なタスクには、推定所要時間、所有者、進捗状況が含まれています。

テスト環境

「環境」を参照してください。

トレーニング

お客様の機械学習モデルに学習するデータを提供すること。トレーニングデータには正しい答えが含まれている必要があります。学習アルゴリズムは入力データ属性をターゲット (お客様が予測したい答え) にマッピングするトレーニングデータのパターンを検出します。これらのパター

ンをキャプチャする機械学習モデルを出力します。そして、お客様が機械学習モデルを使用して、ターゲットがわからない新しいデータでターゲットを予測できます。

トランジットゲートウェイ

VPC とオンプレミスネットワークを相互接続するために使用できる、ネットワークの中継ハブ。詳細については、 AWS Transit Gateway ドキュメントの「トランジットゲートウェイとは」を参照してください。

トランクベースのワークフロー

デベロッパーが機能ブランチで機能をローカルにビルドしてテストし、その変更をメインブランチにマージするアプローチ。メインブランチはその後、開発環境、本番前環境、本番環境に合わせて順次構築されます。

信頼されたアクセス

ユーザーに代わって AWS Organizations およびそのアカウントで組織内でタスクを実行するために指定するサービスへのアクセス許可を付与します。信頼されたサービスは、サービスにリンクされたロールを必要なときに各アカウントに作成し、ユーザーに代わって管理タスクを実行します。詳細については、 ドキュメントの「 AWS Organizations を他の AWS のサービスで使用する AWS Organizations 」を参照してください。

チューニング

機械学習モデルの精度を向上させるために、お客様のトレーニングプロセスの側面を変更する。例えば、お客様が機械学習モデルをトレーニングするには、ラベル付けセットを生成し、ラベルを追加します。これらのステップを、異なる設定で複数回繰り返して、モデルを最適化します。

ツーピザチーム

2 つのピザを食べることができる小さな DevOps チーム。ツーピザチームの規模では、ソフトウェア開発におけるコラボレーションに最適な機会が確保されます。

# U

不確実性

予測機械学習モデルの信頼性を損なう可能性がある、不正確、不完全、または未知の情報を指す概念。不確実性には、次の 2 つのタイプがあります。認識論的不確実性は、限られた、不完全なデータによって引き起こされ、弁論的不確実性 は、データに固有のノイズとランダム性によって引き起こされます。詳細については、深層学習システムにおける不確実性の定量化 ガイドを参照してください。

未分化なタスク

へビーリフティングとも呼ばれ、アプリケーションの作成と運用には必要だが、エンドユーザーに直接的な価値をもたらさなかったり、競争上の優位性をもたらしたりしない作業です。未分化なタスクの例としては、調達、メンテナンス、キャパシティプランニングなどがあります。

上位環境

「環境」を参照してください。

## V

バキューミング

ストレージを再利用してパフォーマンスを向上させるために、増分更新後にクリーンアップを行うデータベースのメンテナンス操作。

バージョンコントロール

リポジトリ内のソースコードへの変更など、変更を追跡するプロセスとツール。

VPC ピアリング

プライベート IP アドレスを使用してトラフィックをルーティングできる、2 つの VPC 間の接続。詳細については、Amazon VPC ドキュメントの「VPC ピア機能とは」を参照してください。

脆弱性

システムのセキュリティを脅かすソフトウェアまたはハードウェアの欠陥。

## W

ウォームキャッシュ

頻繁にアクセスされる最新の関連データを含むバッファキャッシュ。データベースインスタンスはバッファキャッシュから、メインメモリまたはディスクからよりも短い時間で読み取りを行うことができます。

ウォームデータ

アクセス頻度の低いデータ。この種類のデータをクエリする場合、通常は適度に遅いクエリでも問題ありません。

ウィンドウ関数

現在のレコードに関連する行のグループに対して計算を実行する SQL 関数。ウィンドウ関数は、移動平均の計算や、現在の行の相対位置に基づく行の値へのアクセスなどのタスクの処理に役立ちます。

ワークロード

ビジネス価値をもたらすリソースとコード (顧客向けアプリケーションやバックエンドプロセスなど) の総称。

ワークストリーム

特定のタスクセットを担当する移行プロジェクト内の機能グループ。各ワークストリームは独立していますが、プロジェクト内の他のワークストリームをサポートしています。たとえば、ポートフォリオワークストリームは、アプリケーションの優先順位付け、ウェーブ計画、および移行メタデータの収集を担当します。ポートフォリオワークストリームは、これらの設備を移行ワークストリームで実現し、サーバーとアプリケーションを移行します。

WORM

「書き込み 1 回」を参照し、多くの を読み取ります。

WQF

「AWS ワークロード認定フレームワーク」を参照してください。

Write Once, Read Many (WORM)

データを 1 回書き込み、データの削除や変更を防ぐストレージモデル。承認されたユーザーは、必要な回数だけデータを読み取ることができますが、変更することはできません。このデータストレージインフラストラクチャはイミュータブルな と見なされます。

# Z

ゼロデイエクスプロイト

ゼロデイ脆弱性 を利用する攻撃、通常はマルウェア。

ゼロデイ脆弱性

実稼働システムにおける未解決の欠陥または脆弱性。脅威アクターは、このような脆弱性を利用してシステムを攻撃する可能性があります。開発者は、よく攻撃の結果で脆弱性に気付きます。

## ゾンビアプリケーション

平均 CPU およびメモリ使用率が 5% 未満のアプリケーション。移行プロジェクトでは、これらのアプリケーションを廃止するのが一般的です。

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。