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

: Apache Spark ジョブのパフォーマンスチューニング AWS Glue 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
耐障害性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
Catalyst オプティマイザ ... 8
パフォーマンス問題の調査 ... 10
Spark UI を使用してボトルネックを特定する ... 10
パフォーマンスをチューニングするための戦略 ... 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プラクティス ... 13
クラスター容量をスケールする ... 14
CloudWatch メトリクス ... 14
Spark UI ... 15
の最新バージョンを使用する ... 16
データスキャン量を削減する ... 17
CloudWatch メトリクス ... 17
Spark UI ... 18
タスクを並列化する ... 26
CloudWatch メトリクス ... 27
Spark UI ... 27
シャッフルを最適化する ... 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
ユーザー定義関数を最適化する ... 45
標準の Python UDF ... 46
ベクトル化された UDF ... 47
Spark SQL ... 48
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 54
C 56
D 59
E 63
F 66
G 67
H 68
I 70
L 72
M 73
O 77
P 80
Q 83
R 83
S 86
T 90
U 91
V 92
W 92
Z 93
.... xciv

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

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

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 AWS Glue のチューニングに関する主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とき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、 AWS Glue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。次に、これらの問題に対処するための戦略を組み込み、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。

- クラスター容量をスケールする
- AWS Glue 最新バージョンを使用する
- データスキャン量を削減する
- タスクを並列化する
- 計画オーバーヘッドを最小限に抑える
- シャッフルを最適化する
- ユーザー定義関数を最適化する

Apache Spark の主要なトピック

このセクションでは、Apache Spark の基本概念と、AWS Glue for Apache Spark のパフォーマンスをチューニングするための主要なトピックについて説明します。実際のチューニング戦略について話し合う前に、これらの概念とトピックを理解しておくことが重要です。

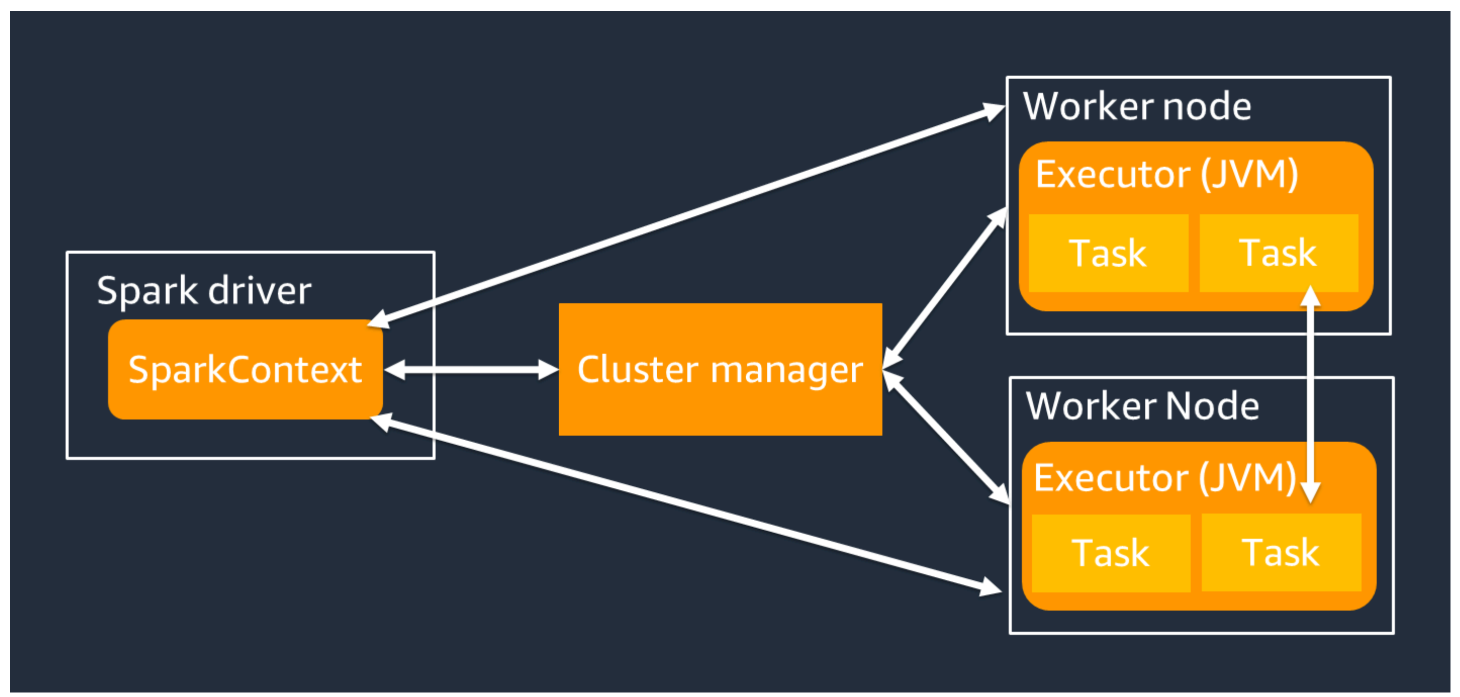
アーキテクチャ

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実行できるタスクに分割します。Spark ドライバーには次の責任があります。

- コード内の `main()` を実行する
- 実行プランを生成する
- クラスターのリソースを管理するクラスターマネージャーと連携して、Spark エグゼキューターをプロビジョニングする
- Spark エグゼキューターに対するタスクをスケジュールし、リクエストする
- タスクの進行状況と復旧を管理する

ジョブ実行では、`SparkContext` オブジェクトを使用して 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。

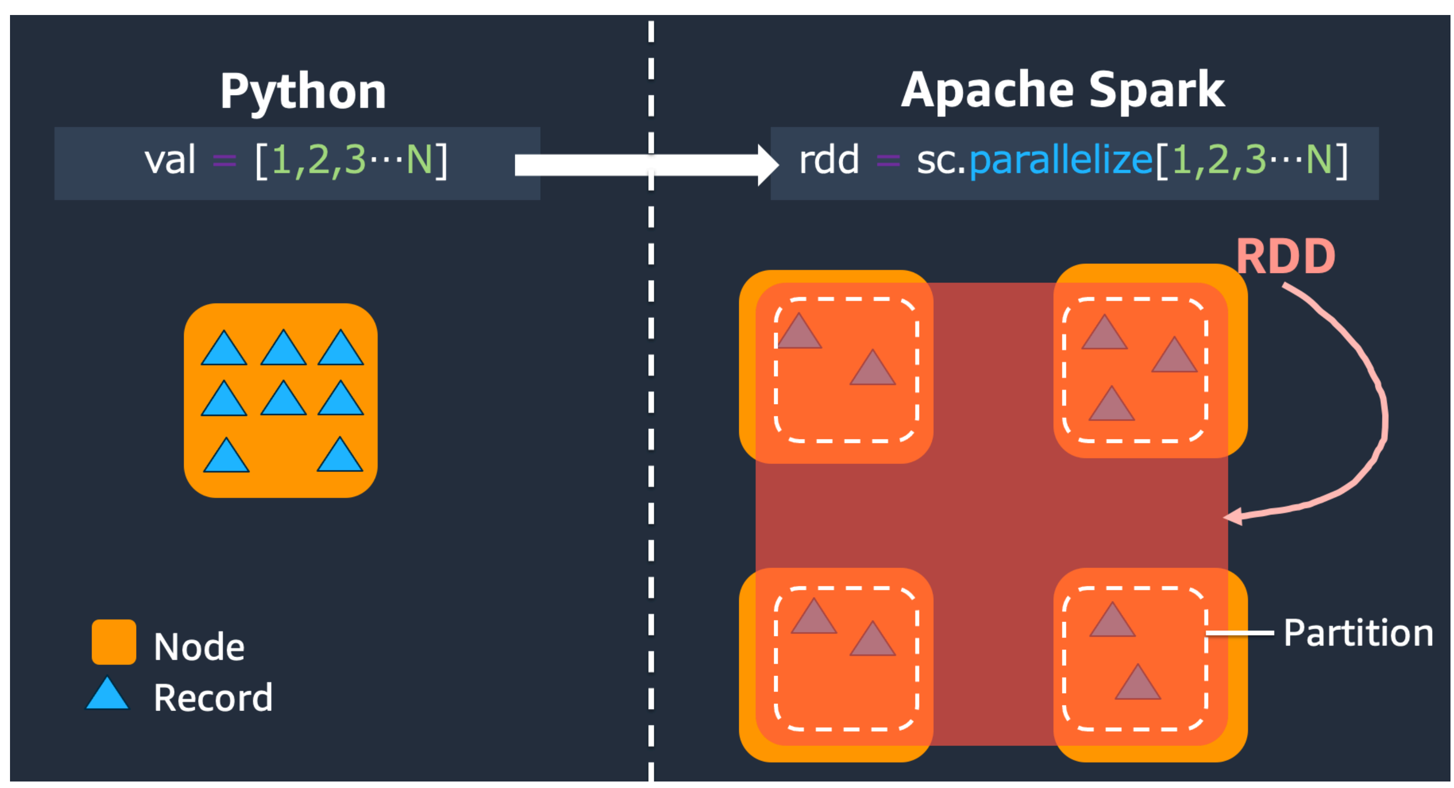


> **Note**
>
> Spark エグゼキューターには複数のスロットがあるため、複数のタスクを並列に処理できます。Spark は、デフォルトで仮想 CPU (vCPU) コアごとに 1 つのタスクをサポートします。例えば、エグゼキューターに 4 つの CPU コアがある場合、4 つのタスクを同時に実行できます。

耐障害性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、耐障害性のある分散データセット (RDD) という抽象化を提供します。これは、並列に操作でき、クラスターの Spark エグゼキューター全体に分割できる要素のコレクションです。

次の図は、Python スクリプトを一般的な環境で実行した場合と、Spark フレームワーク (PySpark) で実行した場合で、メモリ内にデータを保存する方法の違いを示しています。



- Python – Python スクリプトで `val = [1,2,3...N]` のように記述すると、データはコードが実行されている単一のマシンのメモリに保持されます。
- 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。例えば、`rdd = sc.parallelize[1,2,3...N]` のようなコードで RDD を生成でき、Spark はデータを複数の Spark エグゼキューターの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、AWS GlueDynamicFrames と Spark DataFrames を介して RDD を使用します。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高レベルのタスクを実行できるようにする抽象化です。これらは内部的に RDD を使用するため、次のコードではデータが複数ノードに透過的に分散およびロードされます。

  - 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。

- RDD は、パーティションと呼ばれる複数の部分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。
- RDD はイミュータブルです。つまり、作成後に変更することはできません。DataFrame を変更するには、次のセクションで定義されている変換を使用できます。
- RDD は使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

RDD は、2 種類のオペレーションをサポートします。一つは、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、もう一つは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です。

- 変換 – RDD 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。

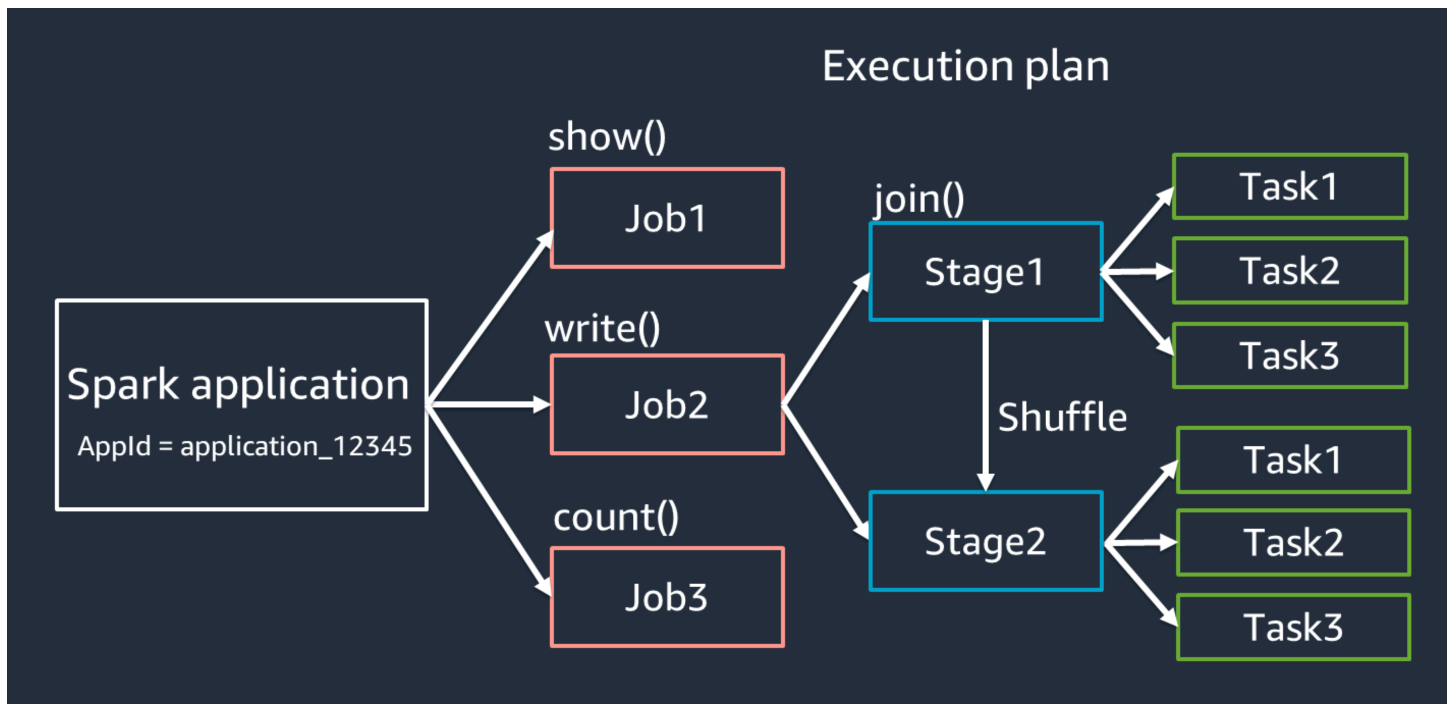
  例えば、`map` は各データセット要素を関数に渡し、結果を表す新しい RDD を返す変換です。`map` メソッドは出力を返さない点に注意してください。Spark は、結果を返す代わりに、抽象的な変換を将来のために保存します。Spark は、アクションが呼び出されるまで、変換を実行しません。
- アクション – 変換を使用して、論理変換プランを構築します。計算を開始するには、`write`、`count`、`show`、`collect` などのアクションを実行します。

  Spark のすべての変換は遅延評価であり、結果はすぐには計算されません。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。これらの変換は、アクションが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 はより効率的に処理を実行できます。例えば、`map` 変換によって作成されたデータセットが、`reduce` のように行数を大幅に削減する変換によってのみ使用される状況を考えてみましょう。その場合、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、いくつかの抽象化を通じてアプリケーションの動作を制御します。Spark UI を使用した開発、デバッグ、パフォーマンスチューニングにおいては、以下の用語が重要です。

- 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。`<application_XXX>` のような一意の ID で識別されます。
- ジョブ – RDD に対して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 に対して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、RDD パーティション全体でデータを再分散し、異なるグループにまとめ直すための Spark のメカニズムです。`join()` などの特定の変換ではシャッフルが必要です。シャッフルの詳細については、「Optimize shuffles」チューニングプラクティスを参照してください。
- タスク – タスクは、Spark によってスケジュールされた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。

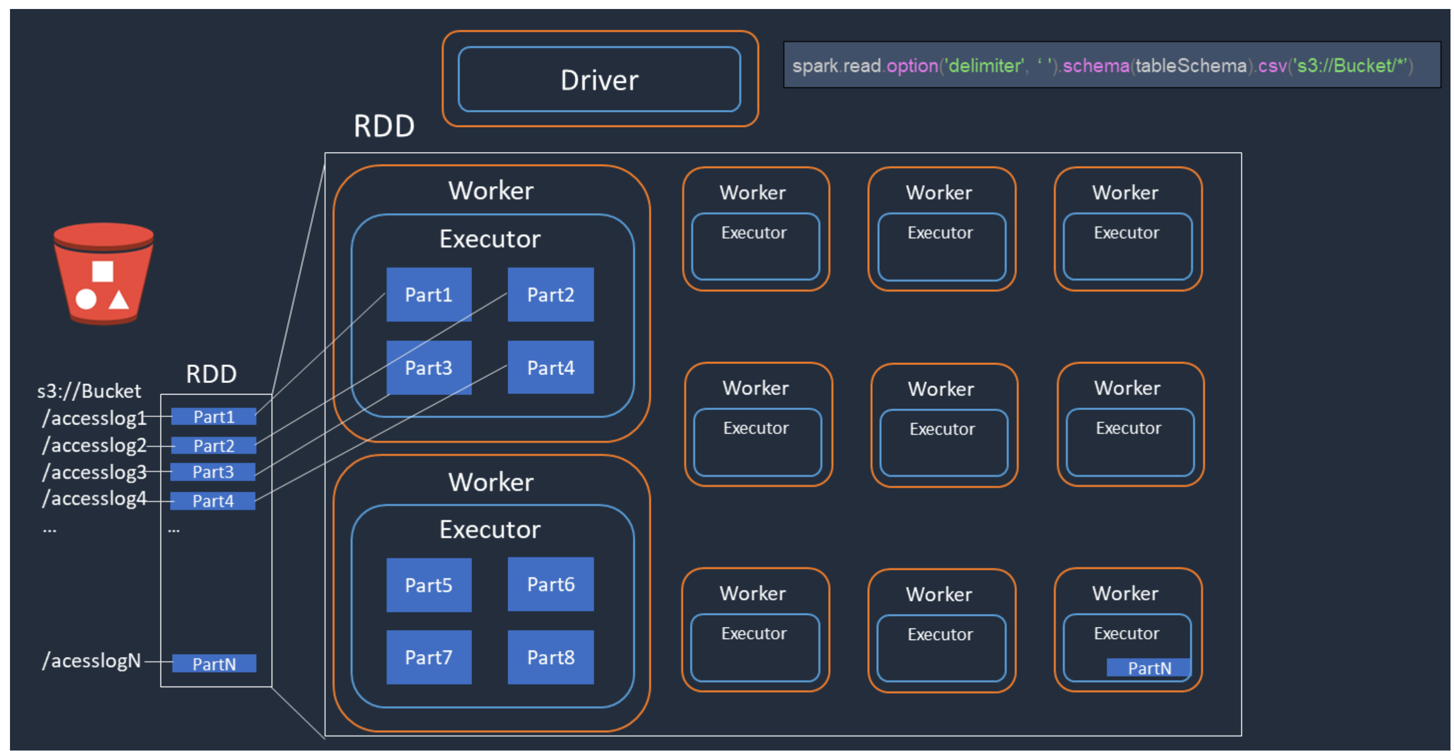


> Note
>
> 並列処理を最適化するときに考慮すべき最も重要なものはタスクです。タスクの数は、RDD の数に応じてスケールします。

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処理します。

Amazon S3 でアクセスログファイル (`accesslog1 ... accesslogN` という名前) 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。



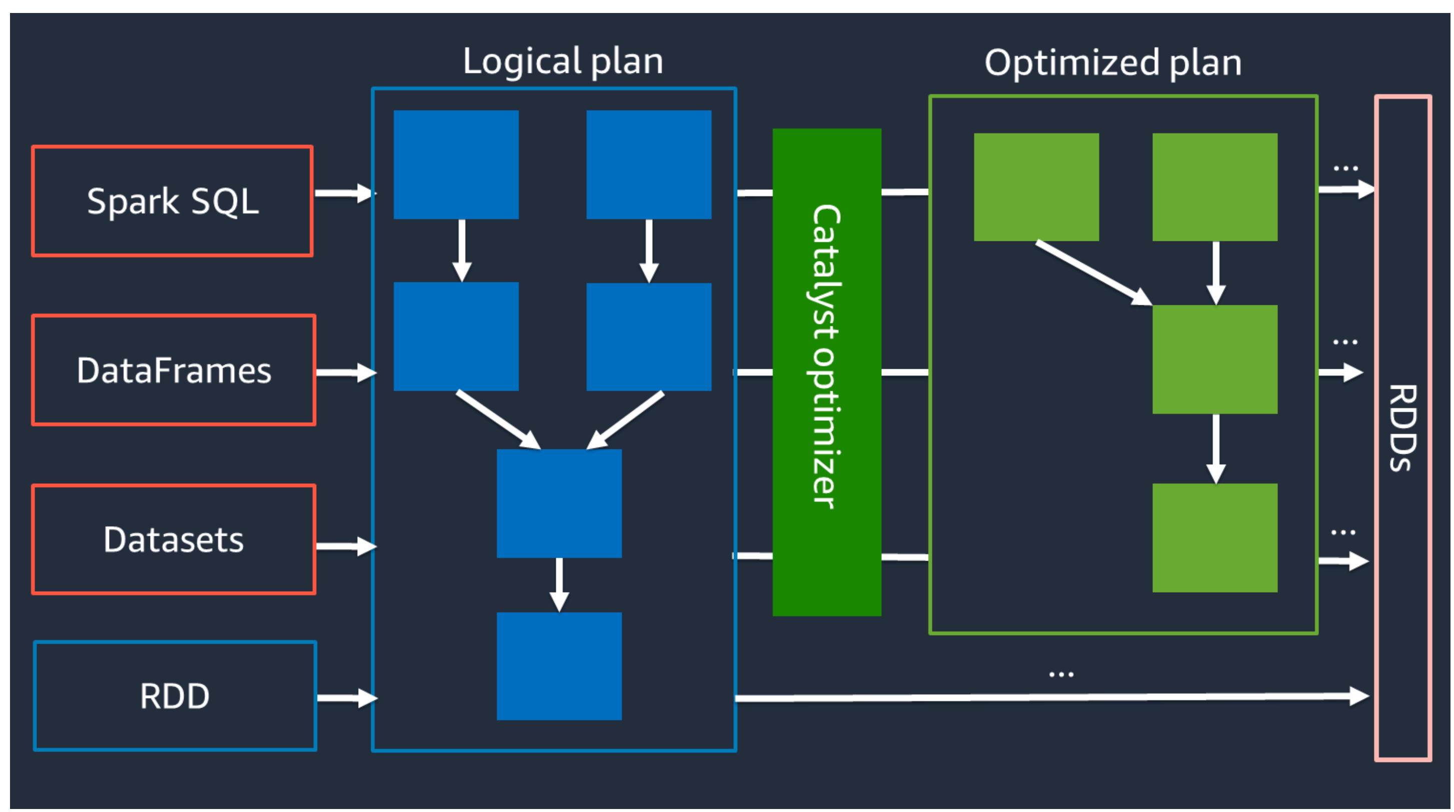
1. Spark ドライバーは、多くの Spark エグゼキューターにまたがる分散処理の実行計画を作成します。
2. Spark ドライバーは、実行計画に基づいて各エグゼキューターにタスクを割り当てます。デフォルトでは、Spark ドライバーは、S3 オブジェクト (`Part1 ... N`) ごとに RDD パーティション (それぞれが 1 つの Spark タスクに対応) を作成します。次に、Spark ドライバーは各エグゼキューターにタスクを割り当てます。

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、それぞれに割り当てられたタスクを並列に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、「Parallelize tasks」セクションを参照してください。

Catalyst 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、実行計画を最適化します。Catalyst にはクエリオプティマイザが搭載されており、次の図に示すように、Spark SQL、DataFrame、Datasets などの高レベルの Spark API を実行するときに使用できます。



Catalyst オプティマイザは RDD API と直接やり取りしないため、高レベル API は一般に、低レベルな 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の実行計画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計画を再最適化します。アダプティブクエリ実行は、以下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を提供します。

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。コード内で `spark.conf.set("spark.sql.adaptive.enabled", "true")` を使用することで、アダプティブクエリ実行を有効または無効にできます。

シャッフル後のパーティションの合体

この機能は、`map` 出力統計に基づいて、各シャッフル後に RDD パーティションを削減 (合体) します。これにより、クエリ実行時のシャッフルパーティション数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数を設定する必要はありません。初期のシャッフルパーティション数が十分に多ければ、Spark が実行時に適切な数を自動的に選択します。

シャッフル後のパーティションの合体は、`spark.sql.adaptive.enabled` と `spark.sql.adaptive.coalescePartitions.enabled` の両方が 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 のドキュメントを参照してください。

ソートマージ結合のブロードキャスト結合への変換

この機能は、サイズが大きく異なる 2 つのデータセットを結合している場合を認識し、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 のドキュメントを参照してください。結合戦略については、「Optimize shuffles」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) に偏ることで、アプリケーション全体の処理時間が遅延する状況を指します。これにより、多くの場合、結合オペレーションのパフォーマンスが低下します。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均等な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、`spark.sql.adaptive.skewJoin.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 の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、「Optimize shuffles」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

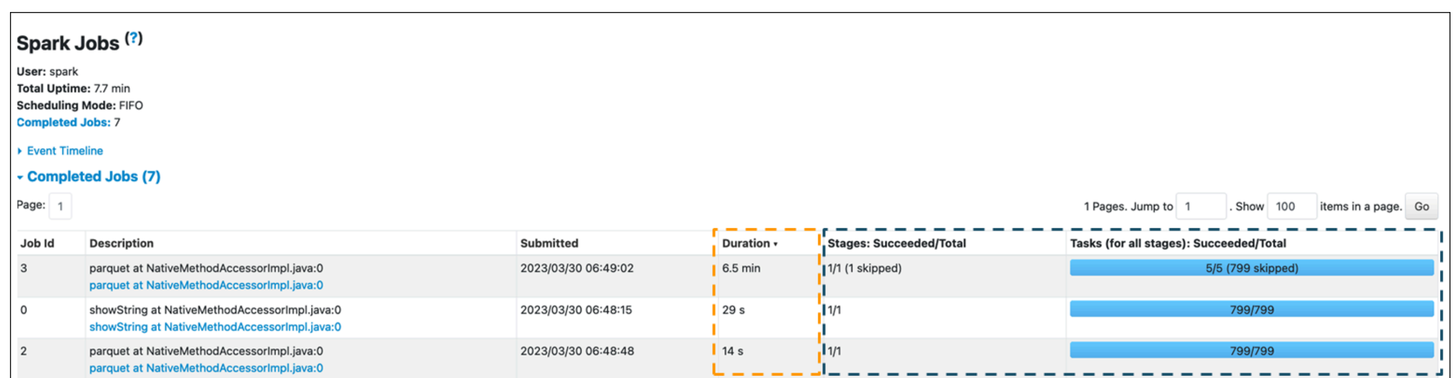
AWS Glue ジョブのパフォーマンスをチューニングするためのベストプラクティスを適用する前に、パフォーマンスをプロファイリングし、ボトルネックを特定することを強くお勧めします。これにより、適切なことに集中できます。

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的なビューを提供します。Spark UI は、パフォーマンスチューニングに関してより詳細なビューを提供します。AWS Glue で Spark UI を使用するには、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チューニングするための戦略」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。

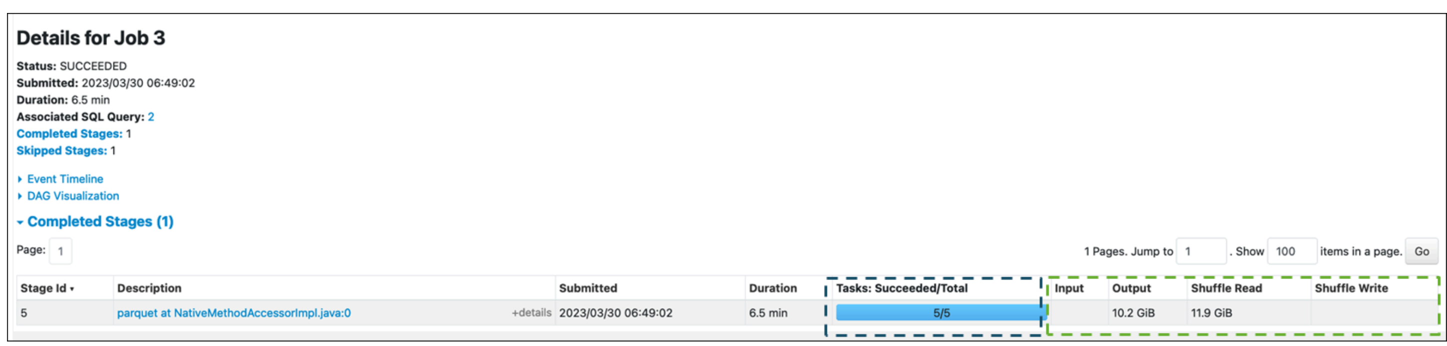
Spark UI を使用してボトルネックを特定する

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 `nativespark-<Job Name>-<Job Run ID>` です。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] に表示されています。

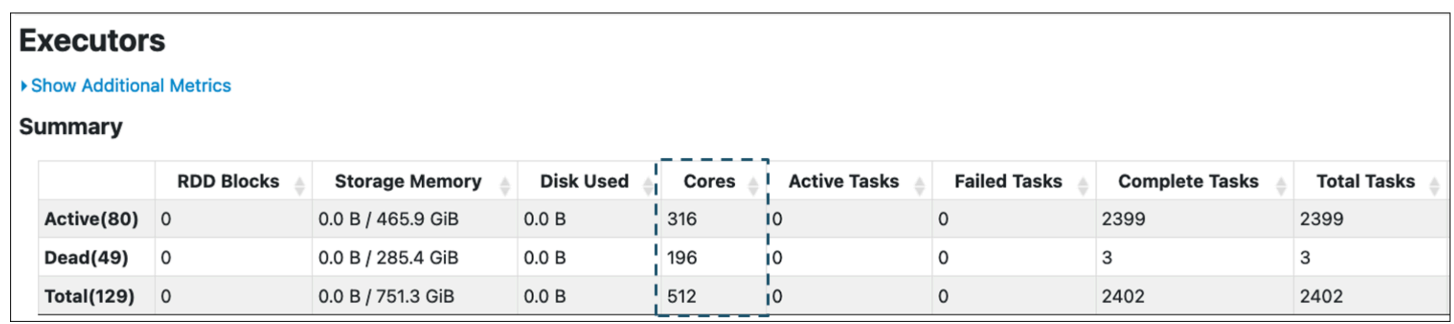
[ジョブ] タブには、Spark アプリケーション内のすべてのジョブの概要が表示されます。ステージまたはタスクの失敗を判断するには、タスクの合計数を確認します。ボトルネックを見つけるには、[期間] を選択してソートします。[説明] 列に表示されるリンクを選択して、長時間実行されるジョブの詳細にドリルダウンします。



[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の数、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。



[エグゼキューター] タブには、Spark クラスターの容量の詳細が表示されます。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、316 個のアクティブコアと合計 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。



	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] ページに表示される 5/5 という値に基づくと、ステージ 5 が最も長いステージですが、512 個のうち 5 個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特定できます。パフォーマンスを向上させるには、その原因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チューニングするための戦略」を参照してください。

Spark ジョブのパフォーマンスをチューニングするための戦略

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。たとえば、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に、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には、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 を介して Spark メトリクスを表示できます。Spark ジョブ AWS Glue 用 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。 には、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サンプルAWS CloudFormation テンプレートと Dockerfile AWS Glue も用意されているため、イベントログで Spark UI を使用できます。

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復を開始できます。

Spark ジョブパフォーマンスのチューニングプラクティス

AWS Glue Spark ジョブのパフォーマンス調整には、次の戦略を使用できます。

- AWS Glue リソース:
  - クラスター容量をスケールする
  - AWS Glue 最新バージョンを使用する
- Spark アプリケーション:
  - データスキャン量を削減する
  - タスクを並列化する
  - シャッフルを最適化する
  - 計画オーバーヘッドを最小限に抑える
  - ユーザー定義関数を最適化する

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 のドキュメントで確認できます。

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を使用できます。不要なデータが Spark クラスターにロードされた場合は、それを削除してロードされたデータ量を減らすことができます。Spark クラスターのリソースが十分に使用されておらず、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オペレーションにかなりの時間がかかっている場合は、それらを最適化することもできます。ジョブの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、メトリクスを確認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、次の各セクションを参照してください。これらの手法は、パフォーマンスのチューニングだけでなく、メモリ不足 (OOM) エラーなどの一般的な問題の解決にも役立ちます。

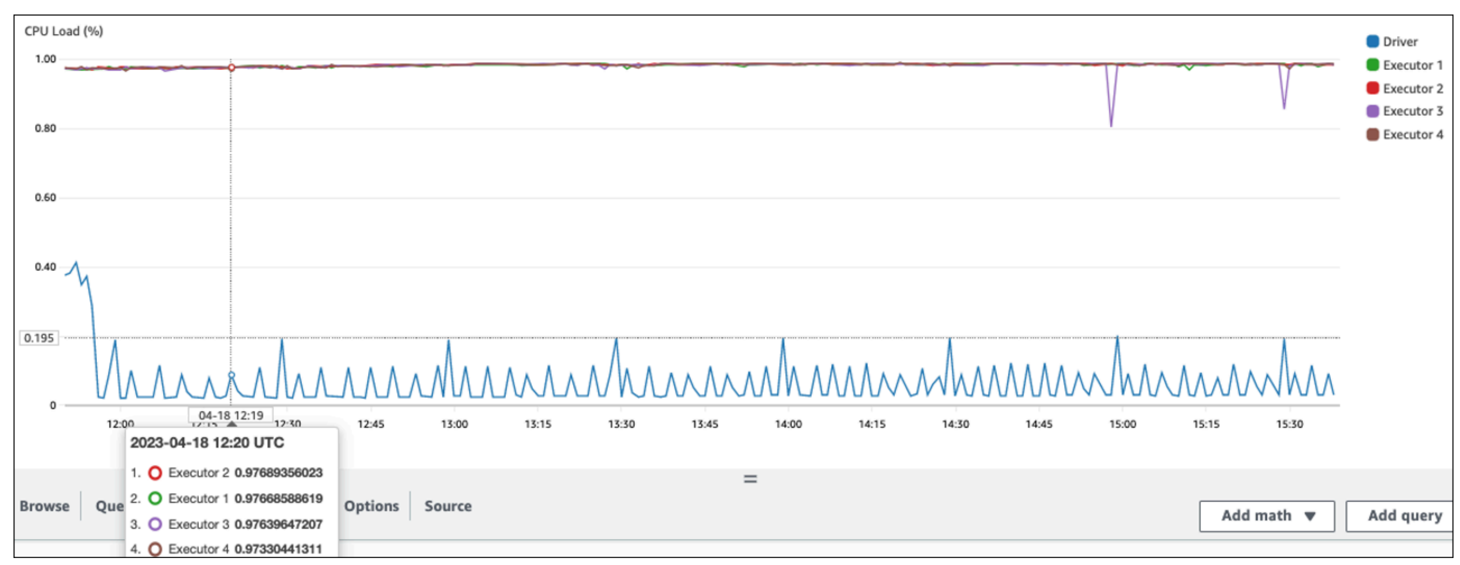
クラスター容量をスケールする

ジョブの処理に時間がかかっているものの、エグゼキューターが十分なリソースを消費しており、Spark が使用可能なコア数に対して多数のタスクを生成している場合は、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

- 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判断するには、CPU 負荷とメモリ使用率を確認します。
- 処理時間がパフォーマンス目標を満たすには長すぎるかどうかを評価するには、ジョブの実行時間を確認します。

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 時間経過しても処理は完了していません。



> **Note**
>
> CPU 負荷が低い場合、クラスター容量をスケーリングしてもおそらく効果は見込めません。

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` 個の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

[エグゼキューター] タブには、エグゼキューターとタスクの合計数が表示されます。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 個のコア) を大きく上回っています。

このような状況が発生した場合は、クラスターのスケーリングを検討してください。クラスター容量は、次のオプションを使用して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) ジョブと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、クラスターからワーカー AWS Glue を自動的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が発生した場合は、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 つのステージで手動スケーリングを行うと、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がないままコストが増加す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、CloudWatch エグゼキューターメトリクスにエグゼキューターの数が表示されます。Spark アプリケーションにおける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 CloudWatch メトリクス AWS Glue を使用したモニタリング」を参照してください。

- スケールアウトする: AWS Glue ワーカーの数を増やす – AWS Glue ワーカーの数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確認されるまでワーカーを追加します。この時点でさらにワーカーを追加しても、結果が改善されずコストが増加します。詳細については、「タスクを並列化する」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。より大きなワーカータイプを使用すると、メモリ集約型のデータ変換、偏りのある集約、ペタバイト規模のデータを含むエンティティ検出チェックなど、負荷の高いデータ統合ジョブを垂直スケーリングで実行できます。

  スケールアップは、例えばジョブのクエリプランがかなり大きいことが原因で、Spark ドライバーにより大きな容量が必要な場合にも有効です。ワーカータイプとパフォーマンスの詳細については、 AWS Big Data Blog の記事「Scale your AWS Glue for Apache Spark jobs with new larger worker types G.4X and G.8X」を参照してください。

  より大きなワーカータイプを使用すると、必要なワーカーの合計数を減らすこともできます。これにより、結合などの高負荷な処理でシャッフルを減らし、パフォーマンスを向上させます。

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。各バージョンには、ジョブのパフォーマンスを自動的に向上させる可能性がある最適化とアップグレードが組み込まれています。たとえば、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 Spark に同等のパフォーマンスの向上をもたらします。Spark 3.3.0 ランタイムは、Spark 2.x の多くのイノベーションを継承しています。

- 拡張された Amazon Redshift コネクタ – AWS Glue 4.0 以降のバージョンでは、Apache Spark と Amazon Redshift を統合できます。この統合は既存のオープンソースコネクタを基盤としており、パフォーマンスとセキュリティが強化されています。この統合により、アプリケーションのパフォーマンスが最大 10 倍向上します。詳細については、「Amazon Redshift integration with Apache Spark」に関するブログ記事を参照してください。
- CSV 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 SIMD ベースの実行 – AWS Glue バージョン 3.0 以降では、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ています。CSV データの詳細については、「ベクトル化された SIMD CSV リーダーで読み取りパフォーマンスを最適化する」を参照してください。JSON データの詳細については、「Apache Arrow 列指向形式によりベクトル化された SIMD JSON リーダーの使用」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 の中で、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「 バージョン 4.0 への AWS GlueAWS Glue ジョブの移行」を参照してください。

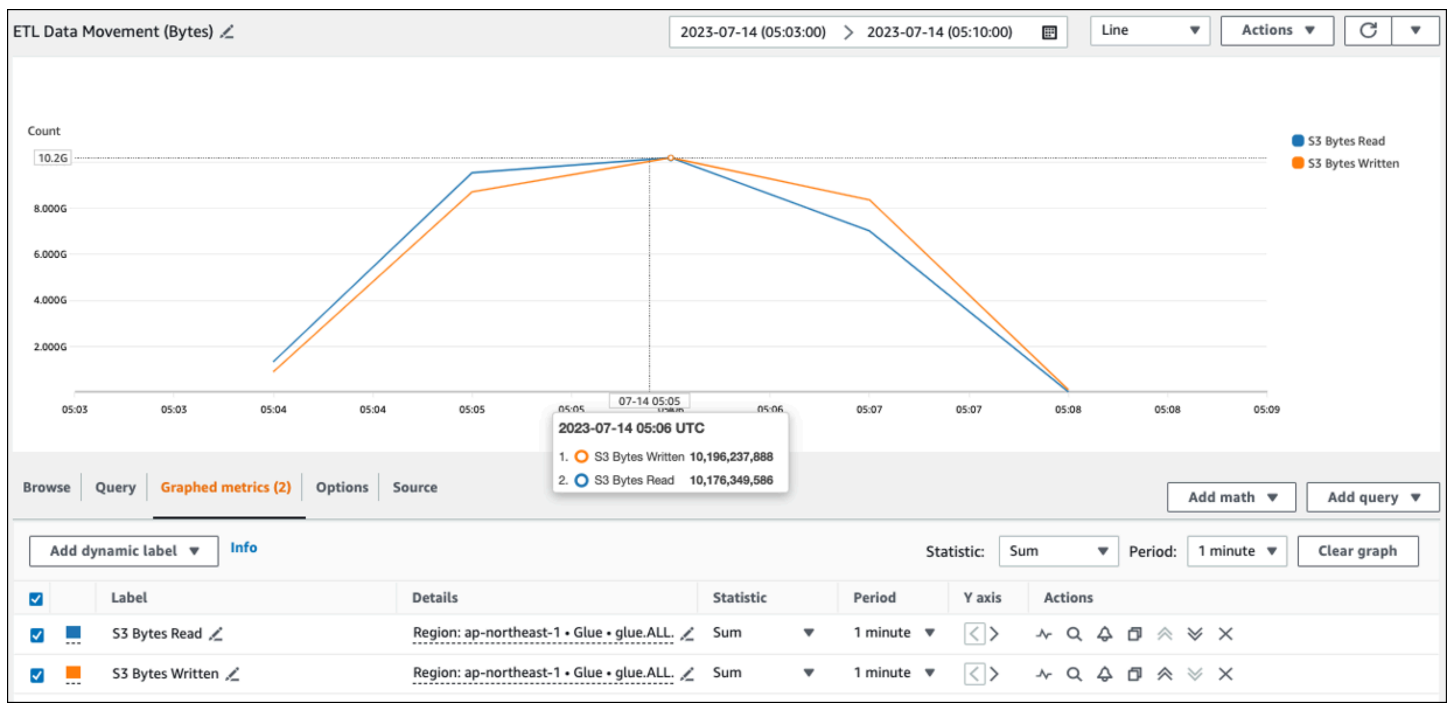
データスキャン量を削減する

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。

Amazon S3 からの読み取りバイト数は CloudWatch メトリクスで確認できる他、「Spark UI」セクションで説明されているように Spark UI でも詳細を確認できます。

CloudWatch メトリクス

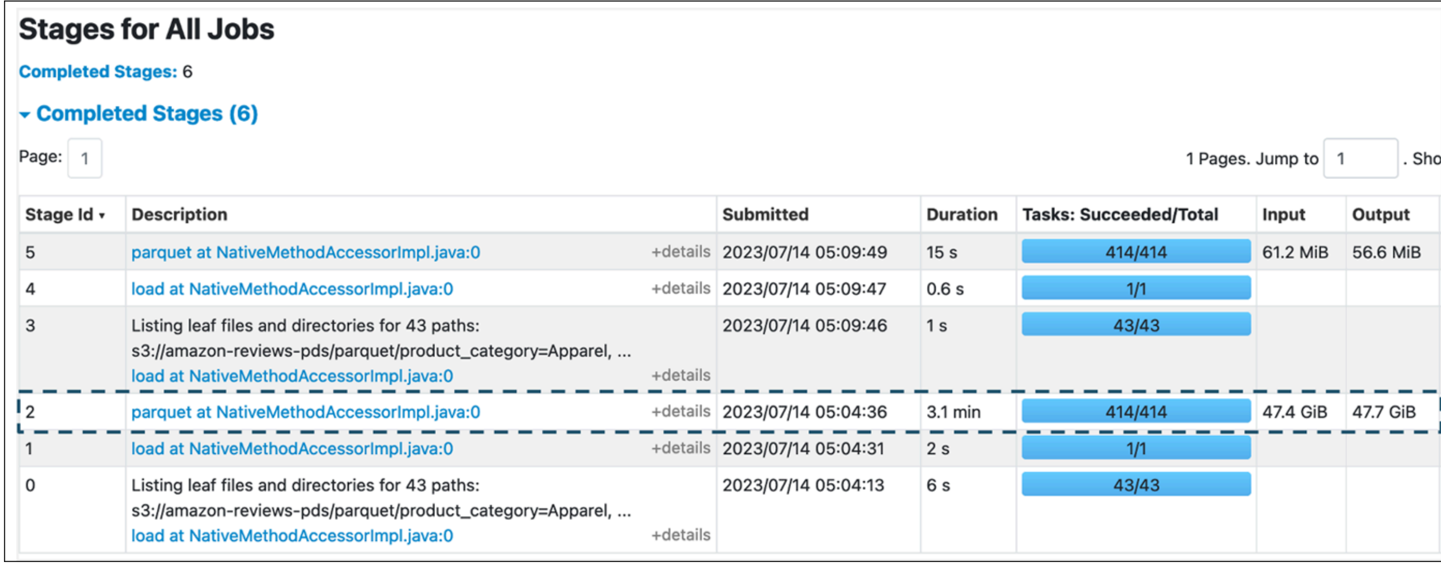
Amazon S3 からのおおよその読み取りサイズは、ETL データ移動 (バイト) で確認できます。このメトリクスは、前回のレポート以降に、すべてのエグゼキューターによって Amazon S3 から読み取られたバイト数を示します。これを使用して、Amazon S3 からの ETL データ移動をモニタリングでき、外部データソースからの取り込みレートと読み取り量を比較できます。



読み取り S3 バイト数の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 のステージタブで、入力サイズと出力サイズを確認できます。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

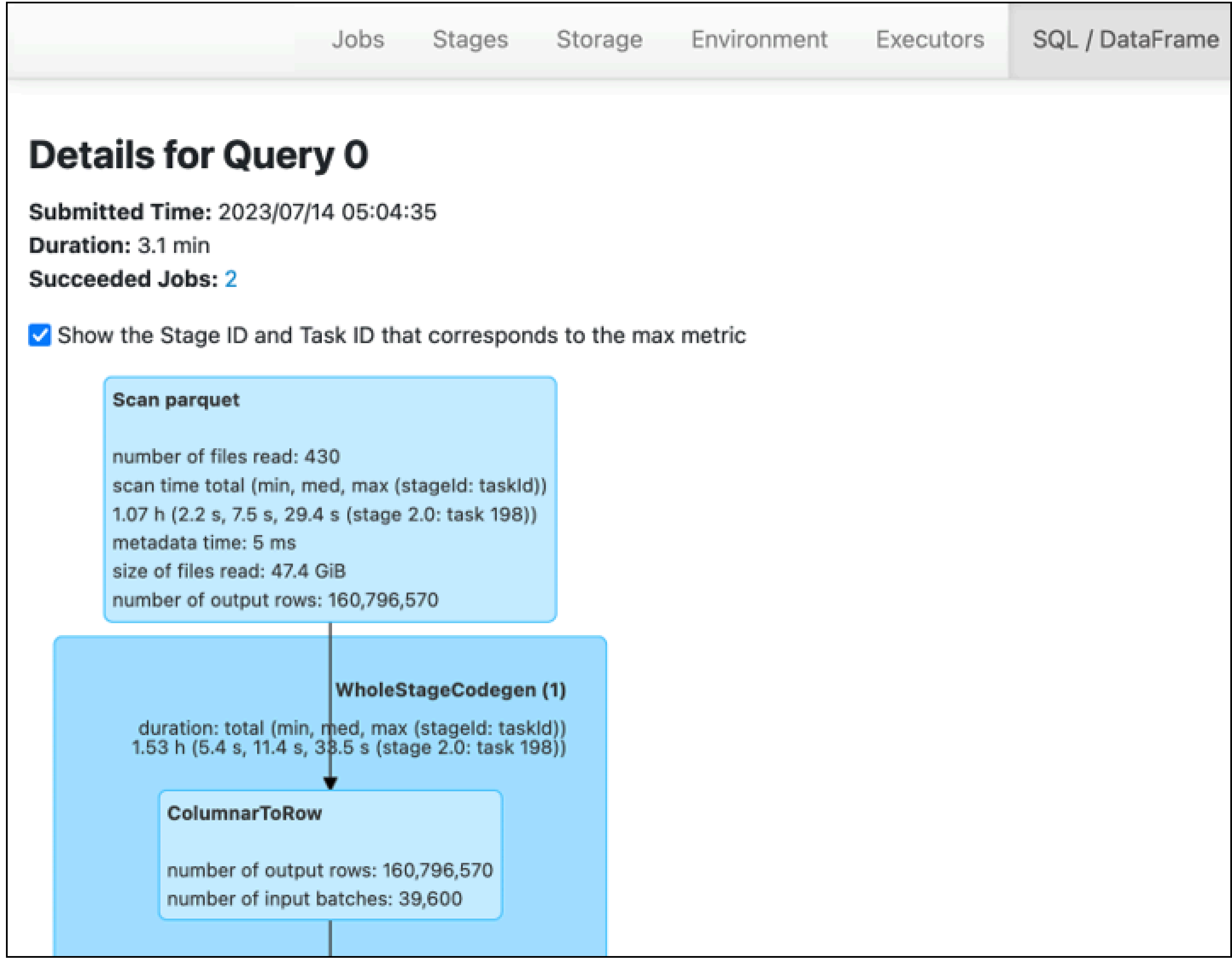
Completed Stages: 6

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場合、SQL /D ataFrame タブには、これらのステージに関するより多くの統計が表示されます。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。



読み込んでいるデータと使用しているデータのサイズに大きな差がある場合、次の解決策を試してください。

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 の場合、raw データの ETL によく使用されますが、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を中程度の範囲 (128 MB など) に維持することをお勧めします。ファイルが小さすぎても大きすぎても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。

  - 多数のオブジェクトに対してリクエスト (`List`、`Get`、`Head`など) を行う際に発生するオーバーヘッドにより、Amazon S3 のネットワーク I/O に大きな負荷がかかります (同じデータの量を少数のオブジェクトに格納する場合と比較して)。
  - 多数のパーティションやタスクが生成されるため、Spark ドライバーにおいて I/O および処理負荷が増大し、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不可 (gzip など) でファイルが大きすぎる場合、Spark アプリケーションは 1 つのタスクがファイル全体の読み取りを完了するまで待つ必要があります。

  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、DynamicFrames のファイルグループを使用します。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。ファイルグループ化を設定するには、`groupFiles` および `groupSize` パラメータを設定します。次のコード例では、これらのパラメータを含む ETL スクリプトで AWS Glue DynamicFrame API を使用しています。

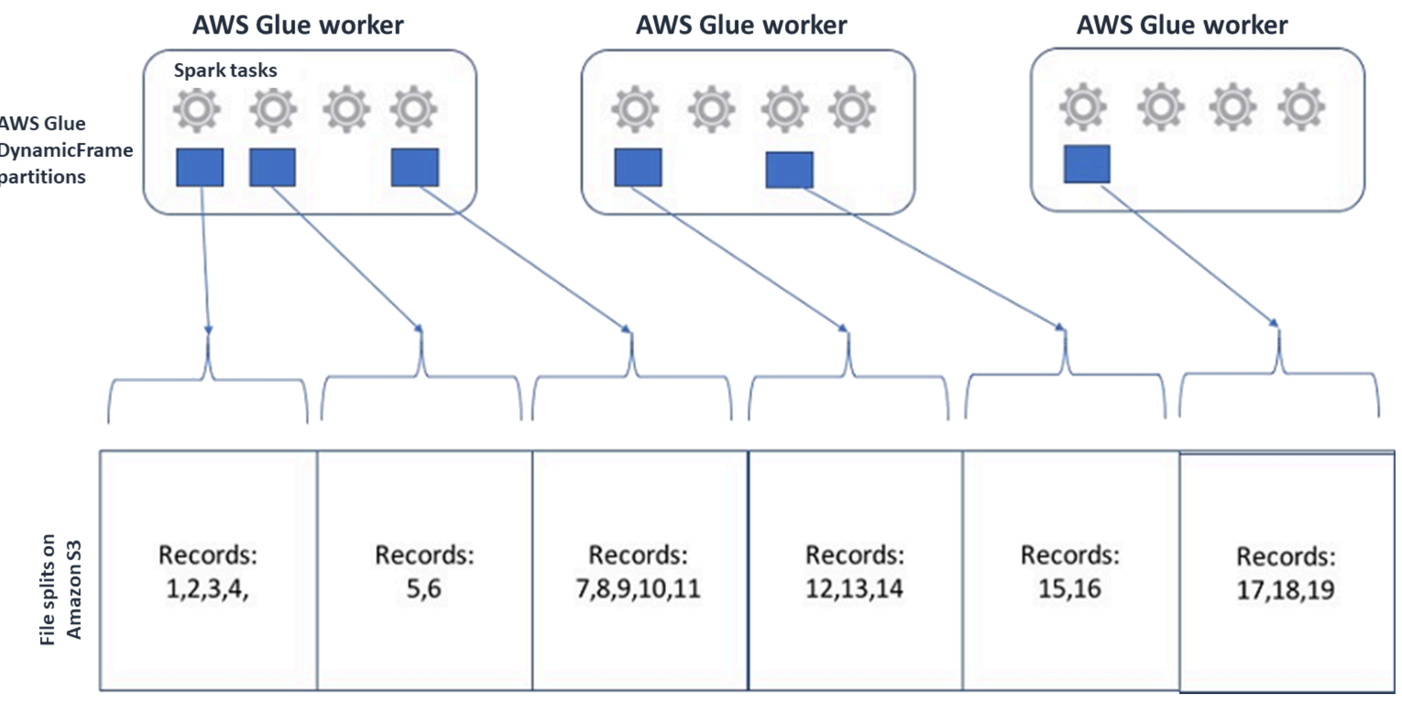
```
dyf = glueContext.create_dynamic_frame_from_options("s3",
    {'paths': ["s3://input-s3-path/"],
    'recurse':True,
    'groupFiles': 'inPartition',
    'groupSize': '1048576'},
    format="json")
```

- 圧縮 — S3 オブジェクトのサイズが数百メガバイトある場合は、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ますが、大きく 2 種類に分類できます。

  - gzip などの分割不可の圧縮形式では、1 つのワーカーがファイル全体を解凍する必要があります。
  - bzip2 や LZO (インデックス付き) などの分割可能な圧縮形式では、ファイルの一部を解凍できるため、並列化が可能です。

  Spark (およびその他の一般的な分散処理エンジン) では、エンジンが並列で処理できるチャンクにソースデータファイルを分割します。これらの単位はスプリット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オプションを GetObject API 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や、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削減されます。一方で、データを圧縮および解凍するには、より多くの 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に応じてスケールされます。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、個々の Parquet ブロックを gzip で圧縮することで、それらのブロックを並列化できます。

- ファイル形式 – 列形式を使用します。Apache Parquet と Apache ORC は、一般的な列指向のデータ形式です。Parquet と ORC は、列ベースの圧縮を使用し、各列をそのデータ型に基づいてエンコードおよび圧縮することで、データを効率的に保存します。Parquet エンコーディングの詳細については、「Parquet encoding definitions」を参照してください。Parquet ファイルは分割可能でもあります。

  列形式では、値を列ごとにフォーマットし、ブロックにまとめて保存します。列形式を使用すると、使用し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 アプリケーションでは、必要な列のみを取得できます。一般に、圧縮率の向上やデータのブロックスキップによって Amazon S3 から読み取るバイト数が減少し、パフォーマンスが向上します。どちらの形式も、I/O を削減する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した列のみを取得する手法です。次の例に示すように、Spark アプリケーション内で列を指定します。
    - DataFrame の例: `df.select("star_rating")`
    - Spark SQL の例: `spark.sql("select start_rating from <table>")`
  - 述語プッシュダウン – 述語プッシュダウンは、`WHERE` 句および `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、そのブロックの統計を保持します。Spark は、アプリケーションで使用されるフィルター値に応じて、そのブロックを読み取るかスキップするかを判断するために、これらの統計を使用ます。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 の例: `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 – データの使用方法に基づいて、S3 データを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、`key=value` の形式でキーと値を Amazon S3 のプレフィックスに保存し、Amazon S3 パスによって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、各下流の分析アプリケーションによってスキャンされるデータの量を制限できるようになるため、パフォーマンスが向上し、コストが削減されます。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。

  パーティション化では、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、 AWS Glue Data Catalogのテーブルでモデル化することで定義できます。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。

- For AWS Glue DynamicFrame の場合は、 `push_down_predicate` (または) を設定しま す`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'",
)
```

- Spark DataFrame の場合は、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 SQL の場合は、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分割するには AWS Glue、次のように DataFrame DynamicFrame の DynamicFrame または partitionBy() に partitionKeys を列の日付情報とともに設定します。
  - 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アクセス権限がない場合、パーティショニングは使用できません。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。DynamicFrame で読み取るときに除外を設定します。例えば、次のコードでは、2023 年の 01～09 月の日付を除外しています。

```
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場合 – 数千の値を持つ ID 列など、値の範囲が広い列で 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が、パーティション化したすべてのフィールドの積になるため、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。
  - データカタログ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が増加する
  - 小さなファイルの数が増え、Amazon S3 API へのリクエスト (`List`、`Get`、`Head`) が増加する

  例えば、`partitionBy` または `partitionKeys` で日付タイプを設定する場合、`yyyy/mm/dd` のような日付レベルのパーティショニングは、多くのユースケースに適しています。ただし、`yyyy/mm/dd/<ID>` のような形式は、非常に多くのパーティション数を生成する可能性があり、パフォーマンス全体に悪影響を与える可能性があります。

  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、`yyyy/mm/dd/hh` のような多くのパーティションが必要になります。ユースケースで大量のパーティションが必要な場合は、データカタログ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ために、AWS Glue のパーティションインデックスを使用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 where 述語 (または句) を指定できます。SQL インターフェイスを提供しないデータベースは、クエリや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、次のパラメータに対して where 句を含む select クエリを指定します。

- DynamicFrame の場合は、sampleQuery オプションを使用します。create_dynamic_frame.from_catalog を使用する場合は、additional_options 引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"
)
```

  using create_dynamic_frame.from_options を使用する場合は、connection_options 引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
    }
```

```
)
```

- DataFrame の場合は、query 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 の場合は、 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に対して完全なスキャンを回避し、前回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には、ジョブの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、ジョブの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。

タスクを並列化する

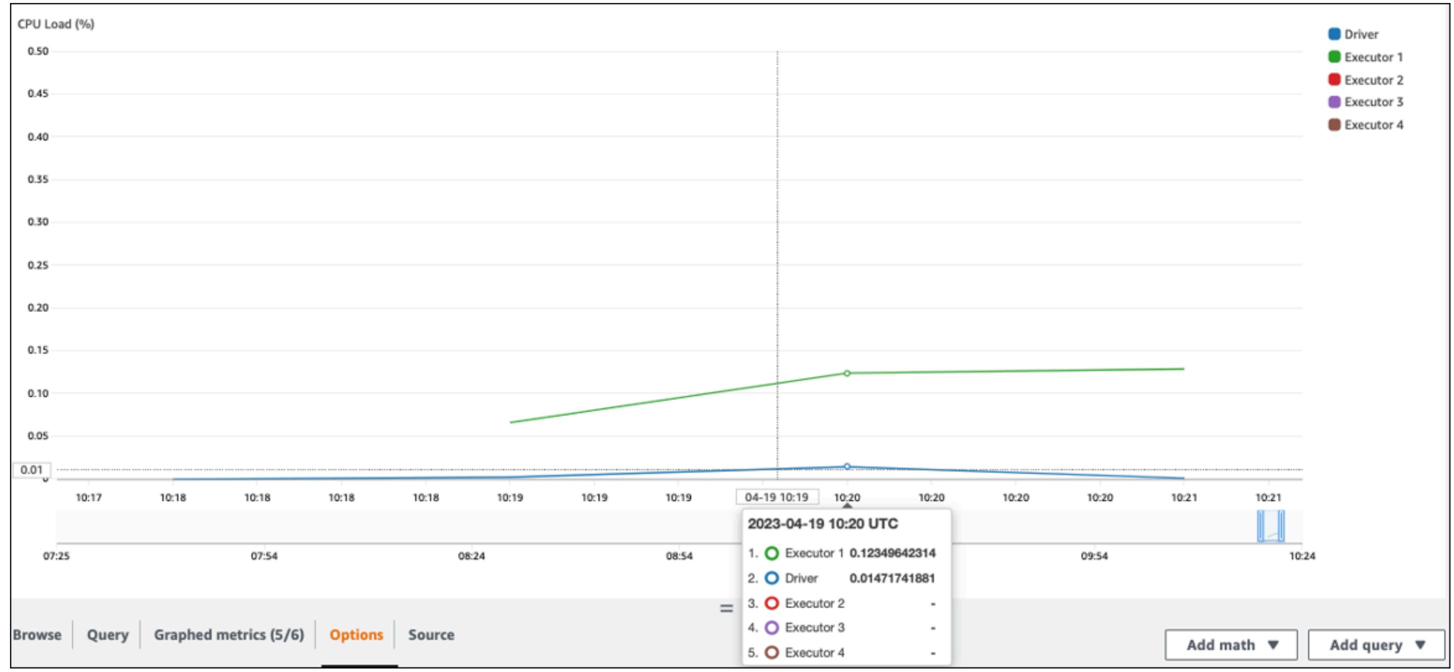
パフォーマンスを最適化するには、データのロードと変換のタスクを並列化することが重要です。「Key topics in Apache Spark」で説明したように、回復力のある分散データセット (RDD) のパーティション数は並列処理の程度を決定するため重要です。Spark が作成する各タスクは、RDD パー

ティションと 1:1 で対応します。最適なパフォーマンスを実現するには、RDD パーティション数がどのように決定され、その数がどのように最適化されるかを理解しておく必要があります。

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

CloudWatch メトリクス

CPU 負荷とメモリ使用率を確認します。ジョブのあるフェーズで一部のエグゼキューターが処理を行っていない場合は、並列処理を改善することをお勧めします。この場合、視覚化された時間枠内ではエグゼキューター 1 が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) は実行していませんでした。これらのエグゼキューターには、Spark ドライバーからタスクが割り当てられていなかったと推測できます。



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、エグゼキューター 1 が 1 つのタスクを処理している様子が表示されます。つまり、このステージの処理はすべて 1 つのエグゼキューターで実行され、他のエグゼキューターはアイドル状態のままでした。

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、まずデフォルトのパーティション数を確認します。その後、ターゲットパーティション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。

デフォルトのパーティション数を決定する

Amazon S3 の場合、Spark RDD の初期パーティション数 (各パーティションは 1 つの Spark タスクに対応) は、Amazon S3 データセットの特性 (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 オブジェクトから AWS Glue DynamicFrame または Spark DataFrame を作成すると、RDD パーティションの初期数 (`NumPartitions`) を次のように計算できます。

- 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`

「Reduce the amount of data scan」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、並列処理が可能なスプリットにします。オブジェクトがスプリットサイズより大きい場合、Spark はオブジェクトを分割し、各スプリットに対して RDD パーティション (およびタスク) を作成します。Spark のスプリット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、おおよその開始値としては妥当です。一部のオブジェクトは gzip などの分割不可な圧縮形式で圧縮されているため、Spark では分割できません。

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

Spark Web UI の [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、コード内で `df.rdd.getNumPartitions()` を実行して並列処理を確認します。

10 GB の gzip ファイルを扱う場合は、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。これが不可能な場合は、ファイルを処理するためにクラスター容量をスケール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間で RDD を再分散する必要があります。

ターゲットパーティション数を手動で決定する

データのプロパティや Spark における特定機能の実装によっては、基盤となる処理を並列化できる場合でも、`NumPartitions` の値が低くなることがあります。`NumPartitions` が少なすぎる場合は、`df.repartition(N)` を実行してパーティション数を増やすことで、処理を複数の Spark エグゼキューターに分散できます。

この場合、`df.repartition(100)` を実行すると `NumPartitions` が 1 から 100 に増加し、データの 100 個のパーティションが作成され、それぞれに他のエグゼキューターへ割り当て可能なタスクが割り当てられます。

`repartition(N)` オペレーションは、データ全体を均等に分割し (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**
>
> `join` などのシャッフルオペレーションを実行すると、`spark.sql.shuffle.partitions` または `spark.default.parallelism` の値に応じて、パーティションの数が動的に増減されます。これにより、Spark エグゼキュター間でのデータ交換をより効率的に実行できるようになります。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、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 (`spark.executor.cores = 4`) に 4 つの vCPU コアを提供します。Spark は vCPU コアごとに 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`)。このパーティション数は、タスクに同じ時間がかかる場合、クラスターを最大限に活用できます。ただし、一部のタスクは他のタスクより時間がかかるため、アイドルコアが発生します。このような場合は、ボトルネックとなるタスクを細分化して効率的にスケジュールするために、`numPartitions` を 2～3 倍に増やすことを検討してください。

パーティション数が多すぎる

パーティション数が過剰になると、タスク数も過剰になります。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、分散処理に関連するオーバーヘッドによって Spark ドライバーに大きな負荷がかかります。

ジョブ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著しく多い場合は、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することで、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、 AWS Glue groupFiles を使用します。これにより、各ファイルを処理するために Apache Spark タスクを起動することで発生する過剰な並列処理を抑えられます。
- `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。パーティション数を減らす場合は、`repartition(N)` より `coalesce(N)` を優先します。`repartition(N)` はシャッフルを発生させて各パーティション内のレコード数を均等に分散するため、コストと管理オーバーヘッドが増加します。
- Spark 3.x のアダプティブクエリ実行を使用します。「Key topics in Apache Spark」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数を自動的に結合する機能を提供します。このアプローチは、ジョブを実行するまでパーティション数がわからない場合に使用できます。

JDBC からのデータロードを並列化する

Spark RDD のパーティション数は設定によって決まります。デフォルトでは、1 件の SELECT クエリでソースデータセット全体をスキャンするタスクが 1 つだけ実行されることに注意してください。

AWS Glue DynamicFrames と Spark DataFrames はどちらも、複数のタスクにわたる並列 JDBC データロードをサポートしています。これは、where 述語を使用して 1 件の 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。JDBC 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。

- For AWS Glue DynamicFrame、 hashfield (または hashexpression)と hashpartition。詳細については、「JDBC テーブルからの並列読み取り」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 DataFrame の場合は、numPartitions、partitionColumn、lowerBound、および upperBound を設定します。詳細については、「JDBC To Other Databases」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 のパーティション数は、`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。

- `dynamodb.splits` の値を増やします。
- AWS Glue for Spark の ETL の接続タイプとオプション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 のパーティション数は、ソースの Amazon Kinesis Data Streams データストリームにおけるシャードの数によって決まります。データストリームのシャードが少ない場合、Spark タスクも少なくなるため、ダウンストリームプロセスにおける並列処理が低下す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際の並列処理を高めるために、シャード数を増やします。
- マイクロバッチ内のロジックが複雑な場合は、不要な列を削除した後、バッチの開始時点で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 パーティション数を増やします。

- 特にロード処理自体を並列化できなかった場合は、初期ロード直後にデータを再パーティション化して、より多くのパーティションを生成します。

DynamicFrame または DataFrame に対して `repartition()` を呼び出し、パーティション数を指定します。使用可能なコア数の 2～3 倍を目安とするのが一般的です。

ただし、パーティションテーブルに書き込む際、ファイルが大量に生成される可能性があります (各パーティションが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るため)。これを回避するには、DataFrame を列単位で再パーティション化します。これは、テーブルのパーティション列を使用するため、書き込み前にデータが整理されます。この方法であれば、テーブルパーティション上に小さなファイルができるのを避けつつ、より多くのパーティション数を指定できます。ただし、一部のパーティション値にデータが集中してタスクの完了が遅れる「データスキュー」が発生しないよう注意が必要です。

- シャッフルが発生する場合は、`spark.sql.shuffle.partitions` の値を増やします。これは、シャッフル時のメモリに関する問題の回避にも役立ちます。

  シャッフルパーティションが 2,001 を超えると、Spark は圧縮メモリ形式を使用します。値がこの上限に近い場合は、より効率的な表現を得るために、`spark.sql.shuffle.partitions` の値をそれ以上に設定することを検討します。

シャッフルを最適化する

`join()` や `groupByKey()` などの特定のオペレーションでは、Spark によるシャッフルの実行が必要です。シャッフルは、RDD パーティション全体でデータを再分散し、異なるグループにまとめ直すための Spark のメカニズムです。シャッフルによって、パフォーマンスのボトルネックを解消できることがあります。ただし、シャッフルでは通常、Spark エグゼキューター間でデータをコピーする必要があるため、複雑でコストのかかる処理となります。例えば、シャッフルによって次のようなコストが発生します。

- ディスク I/O:
  - ディスク上に大量の中間ファイルが生成されます。
- ネットワーク I/O:
  - 多くのネットワーク接続が必要になります (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、データセットの多くが Spark エグゼキューター間でネットワーク越しに移動する可能性があります。
- CPU とメモリの負荷:

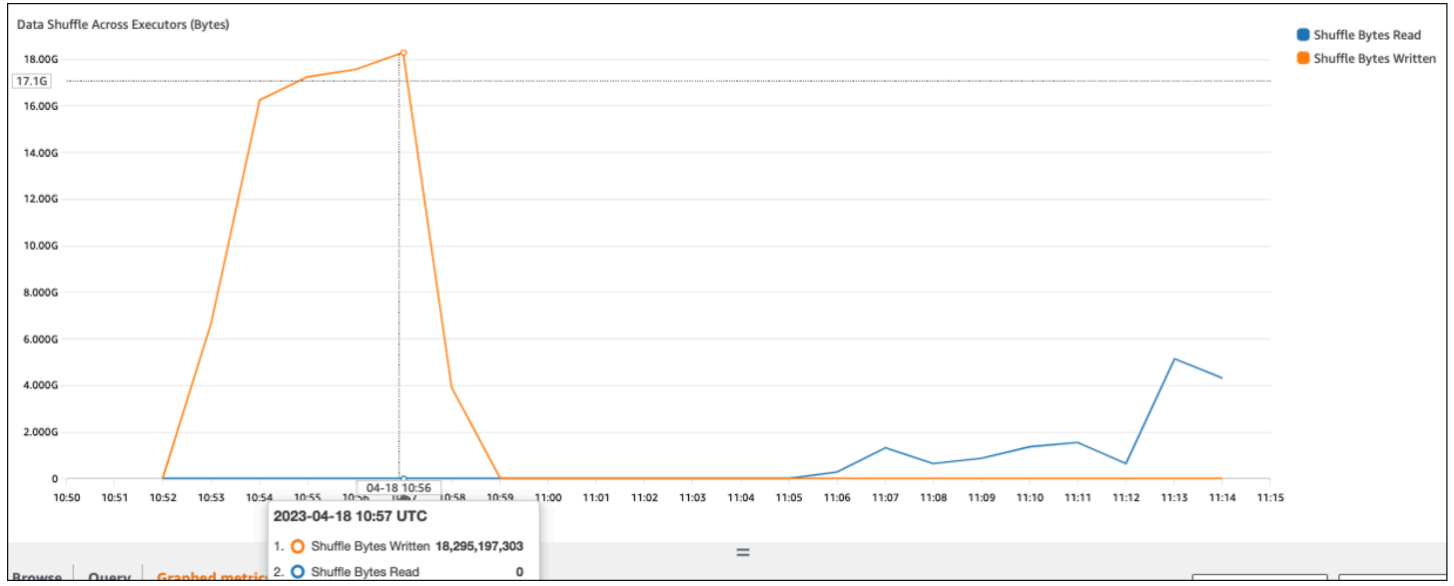
- 値をソートし、データセットをマージします。これらのオペレーションはエグゼキューターで計画され、エグゼキューターに大きな負荷がかかります。

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つながる最も重要な要因の 1 つです。中間データを保存する際にエグゼキューターのローカルディスクのスペースが枯渇し、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

[書き込み済みシャッフルバイト数] の値が [読み取り済みシャッフルバイト数] よりも大きい場合、Spark ジョブで `join()`や `groupByKey()` などのシャッフルオペレーションが使用されている可能性があります。



Spark UI

Spark UI の [ステージ] タブで、[シャッフル読み取りサイズ/レコード数] 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が約 18.6 GB/4020000 件のレコードをシャッフルプロセスとやり取りしており、合計のシャッフル読み取りサイズは約 75 GB になります。

[シャッフルスピル (ディスク)] 列には、大量のメモリスピルデータがディスクに書き込まれていることが示されており、ディスクのスペース不足やパフォーマンスの問題を引き起こす可能性があります。

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

これらの症状が見られ、ステージの処理がパフォーマンス目標と比べて時間がかかりすぎる場合、または `Out Of Memory` や `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。

結合を最適化する

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になります。結合はコストのかかるオペレーションであるため、ビジネス要件に不可欠でない限り、使用しないことをお勧めします。データパイプラインを効率的に使用できているか、次の質問で再確認します。

- 他のジョブでも実行され、再利用できる結合を再計算していませんか?
- 出力のコンシューマーが使用しない値を取得するためだけに、外部キーを解決する結合をしていませんか?

結合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結合を最適化するための以下のオプションを検討してください。

結合前にプッシュダウンを使用する

結合を実行する前に、DataFrame 内の不要な行と列を除外します。これには以下の利点があります。

- シャッフル中のデータ転送量が削減される
- Spark エグゼキューターでの処理量が削減される
- データスキャン量が削減される

```
# 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"])
```

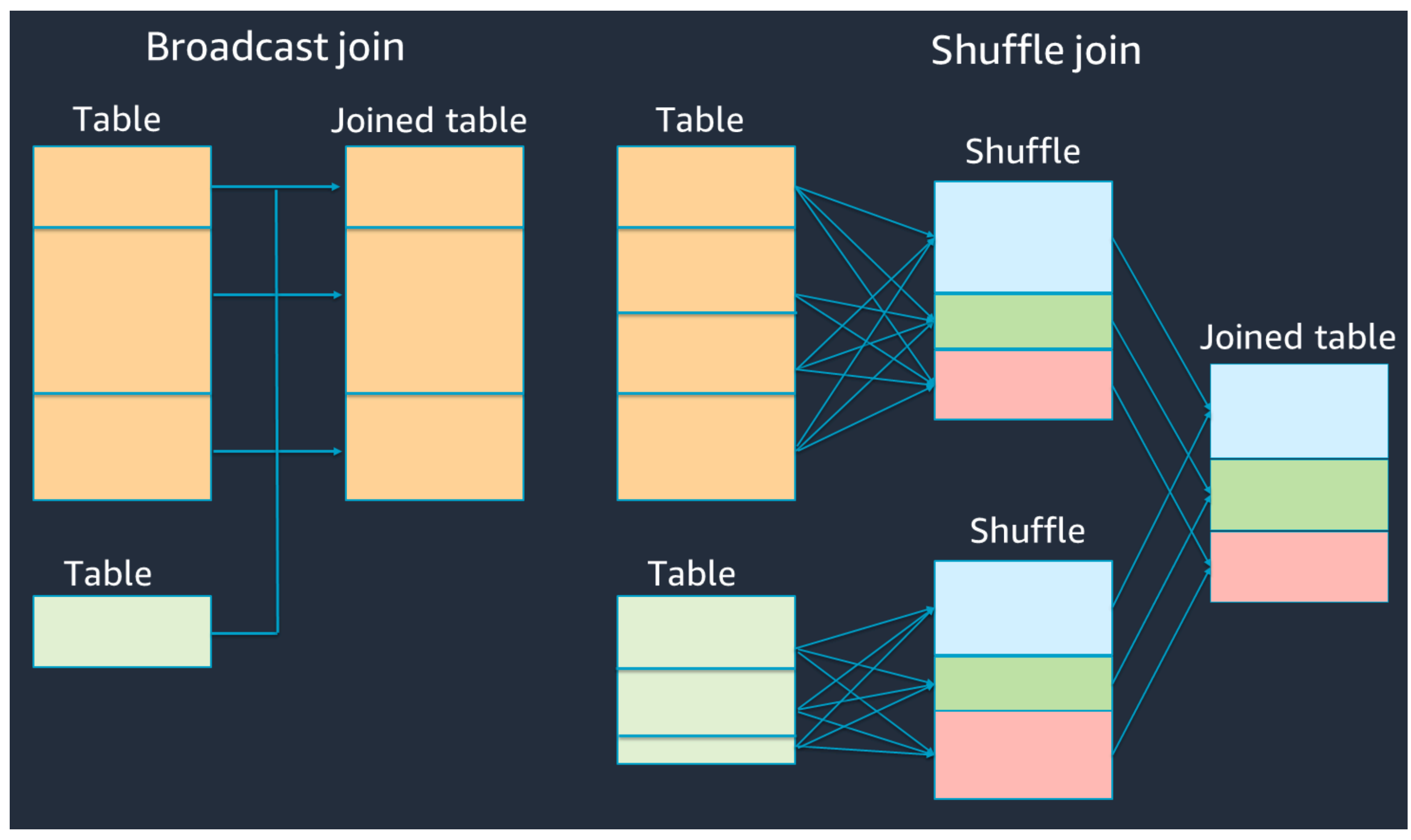
DataFrame 結合を使用する

RDD API や DynamicFrame の結合ではなく、SparkSQL、DataFrame、Datasets などの Spark 高レベル API の使用を検討します。`dyf.toDF()` などのメソッド呼び出しを使用して、DynamicFrame を DataFrame に変換できます。「Key topics in Apache Spark」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ています。

シャッフル結合とブロードキャストハッシュ結合、およびヒントの活用

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ではシャッフルが不要となり、シャッフル結合よりも処理量を抑えられる場合があります。ただし、これは小さいテーブルを大きなテーブルに結合する場合にのみ適用できます。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全体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収まる小規模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。大規模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送信し、続いて大きな RDD またはテーブルの各パーティションとマップ側で結合します。

    RDD またはテーブルの一方がメモリに収まる場合、またはメモリに収まるように調整できる場合の結合に適しています。シャッフルが不要になるため、可能な場合はブロードキャストハッシュ結合を使用することが有益です。結合ヒントを使用すると、次のように Spark にブロードキャスト結合を要求できます。

```
# 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活用できます。アダプティブクエリ実行は、どちらか一方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のしきい値を下回る場合に、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 Adaptive Query Execution を有効にできます`spark.sql.adaptive.enabled=true`。AWS Glue 4.0 では、アダプティブクエリ実行がデフォルトで有効になっています。

シャッフル結合とブロードキャストハッシュ結合に関連する、以下の追加パラメータを設定できます。

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

関連パラメータの詳細については、「ソートマージ結合のブロードキャスト結合への変換」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。

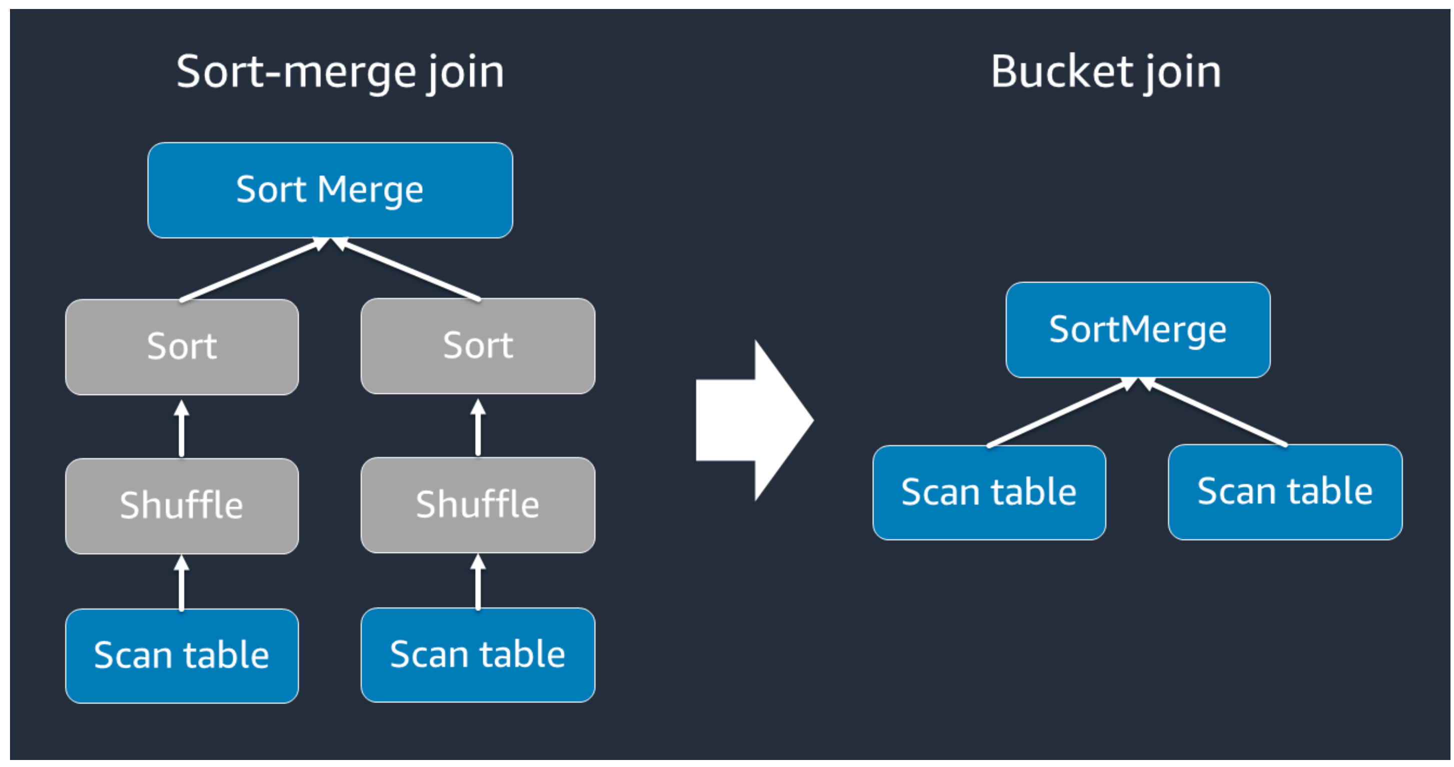
```
-- Join Hints for shuffle sort merge join
SELECT /*+ SHUFFLE_MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGEJOIN(t2) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- 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、そしてマージという 2 つのフェーズが必要です。これらの 2 つのフェーズでは、一部のエグゼキューターがマージを実行し、他のエグゼキューターが同時にソートを行っていると、Spark エグゼキューターが過負荷となり、OOM やパフォーマンスの問題が発生する可能性があります。このような場合には、バケット化を使用することで効率的に結合できる可能性があります。バケット化は、結合キーに基づいて入力をあらかじめシャッフルおよびソートし、そのソート済みデータを中間テーブルに書き込みます。ソート済みの中間テーブルを事前に定義しておくことで、大規模なテーブルを結合する際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。



バケットテーブルは、次の場合に便利です。

- account_id など、同じキーで頻繁に結合されるデータ
- 共通の列で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、日次の累積テーブルをロードする場合

バケットテーブルを作成するには、次のコードを使用します。

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

結合前に、結合キーで DataFrames を再パーティションする

結合前に 2 つの DataFrame を結合キーで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")
```

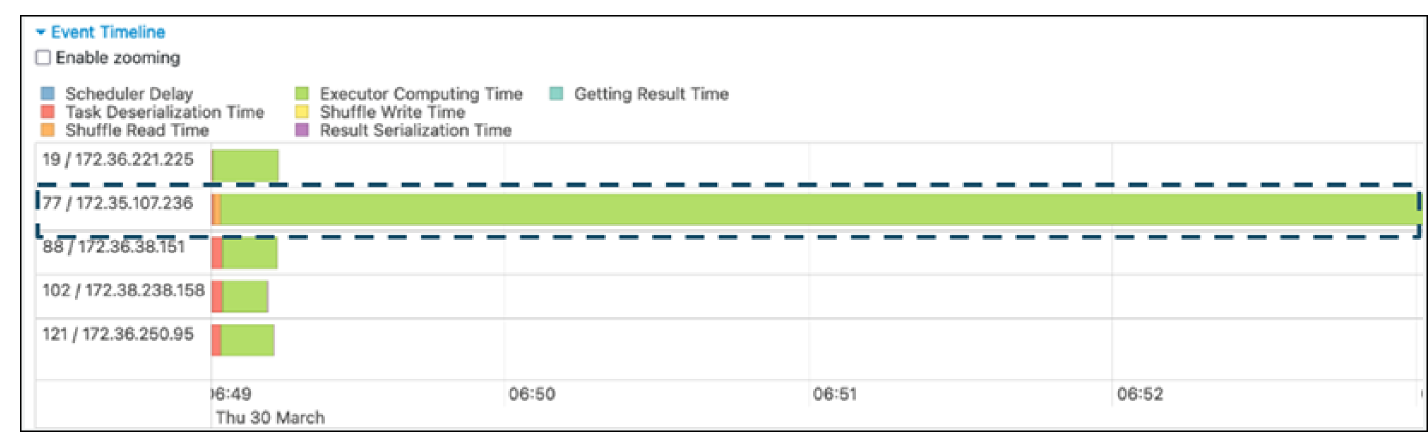
これにより、結合を開始する前に、2 つの (まだ分離された) RDD を結合キーでパーティションします。2 つの RDD が同じキーで同じパーティションコードを用いてパーティションされている場合、結合する予定の RDD レコードは、結合のためのシャッフルが行われる前に、同じワーカー上に配置される可能性が高くなります。これにより、結合時のネットワークアクティビティとデータスキューが減少し、パフォーマンスが向上する可能性があります。

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として最も一般的な原因の 1 つです。これは、データが RDD のパーティション間で均等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に他のタスクよりもはるかに長い時間がかかり、アプリケーション全体の処理時間に遅延が生じます。

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、タスクが不均等に分布している様子が示されています。不均等に分布しているタスクや、実行に時間がかかりすぎているタスクは、データスキューの兆候であ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には、[時間]、[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メトリクスが示されています。

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で類似した数値が表示されます。データスキューがある場合は、各パーセンタイルに大きく偏った値が表示されます。この例では、タスクの処理時間は [最小値]、[25 パーセンタイル]、[中央値]、[75 パーセンタイル] で 13 秒未満です。一方、[最大値] のタスクは [75 パーセンタイル] の 100 倍のデータを処理しており、その処理時間は 6.4 分で、約 30 倍の長さになります。つまり、少なくとも 1 件のタスク (またはタスクの最大 25%) が、残りのタスクよりもはるかに長い時間を要したことになります。

データスキューが見られる場合は、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 Adaptive Query Execution 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、デフォルトで AWS Glue 4.0 で有効になっています。

  結合によって発生するデータスキューにも、次の関連パラメータを設定することで、アダプティブクエリ実行を使用できます。

  - `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 のドキュメントを参照してください。

- 結合キーには、値の範囲が広いキーを使用します。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、ハッシュ関数がデータをパーティション間で均等に分散できない可能性が高くなります。そのため、アプリケーションやビジネスロジックでサポートされている場合は、よりカーディナリティの高いキー、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

繰り返し使用する DataFrames では、`df.cache()` または `df.persist()` を使用して、各 Spark エグゼキューターのメモリとディスクに計算結果をキャッシュすることで、追加のシャッフルや計算を回避します。Spark では、RDD をディスクに永続化したり、複数のノードにレプリケートしたりすること (ストレージレベル) もサポートされています。

例えば、`df.persist()` を追加することで DataFrames を永続化できます。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`、`collect` などの不要なアクションは実行しないでください。「Key topics in Apache Spark」セクションで説明したように、Spark は遅延評価を採用しており、変換された各 RDD は、アクションを実行するたびに再計算される可能性があります。多くの Spark アクションを使用すると、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不要な場合は、それらを削除することを検討してください。

> **Note**
>
> 商用環境では、Spark の `collect()` をできるだけ使用しないでください。`collect()` アクションは、Spark エグゼキューターでのすべての計算結果を Spark ドライバーに返すため、Spark ドライバーで OOM エラーが発生する可能性があります。OOM エラーを回避するため、Spark では `spark.driver.maxResultSize = 1GB` がデフォルトで設定されており、Spark ドライバーに返されるデータサイズは最大 1 GB に制限されています。

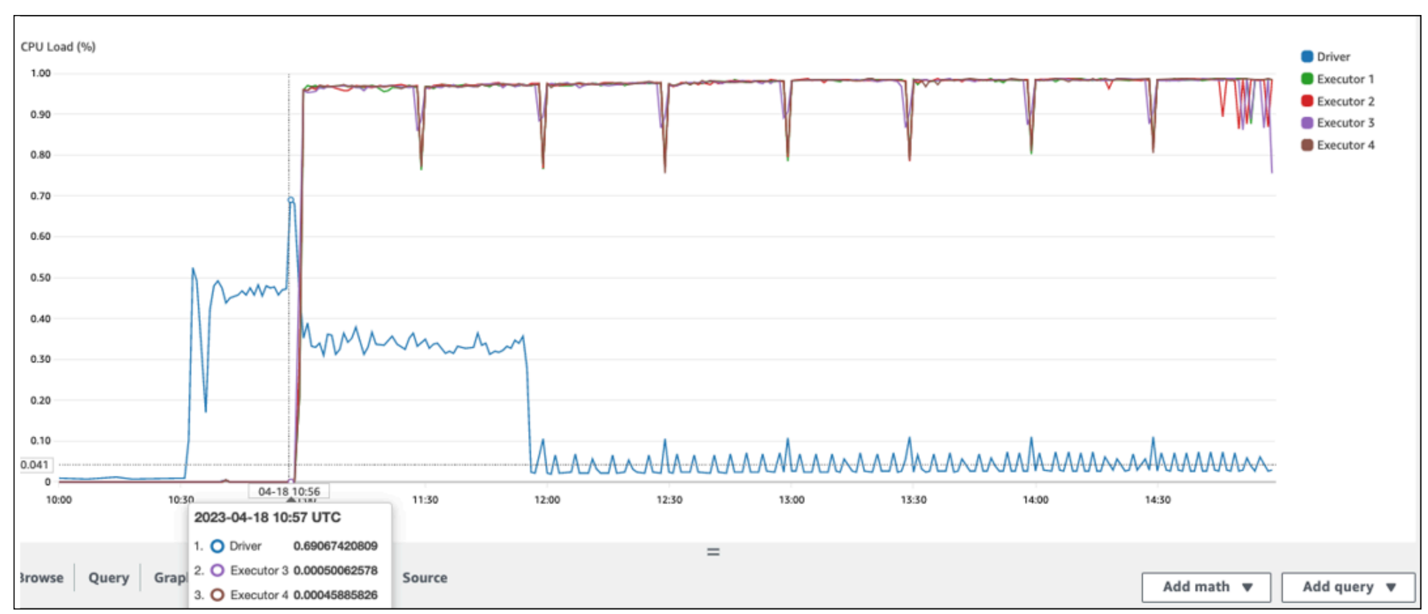
計画オーバーヘッドを最小限に抑える

「Key topics in Apache Spark」で説明されているように、Spark ドライバーは実行計画を生成します。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、 AWS Glue Data Catalog に多数のパーティションが含まれている場合、Spark ドライバーがボトルネックになる可能性があります。計画のオーバーヘッドが大きいかどうかを特定するには、次のメトリクスを評価します。

CloudWatch メトリクス

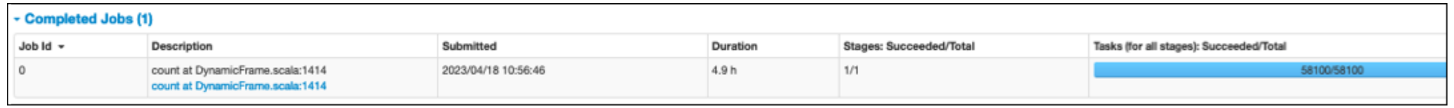
次の状況において、[CPU 負荷] と [メモリ使用率] を確認します。

- Spark ドライバーの [CPU 負荷] と [メモリ使用率] が高く記録されています。通常、Spark ドライバーはデータを処理しないため、CPU 負荷とメモリ使用率が急増することはありません。ただし、Amazon S3 のデータソースに小さなファイルが多数存在する場合、S3 オブジェクトの一覧表示や大量のタスクの管理によって、リソース使用率が高くなる可能性があります。
- Spark エグゼキューターによる処理開始までに長い時間がかかっています。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 負荷が 10:57 まで低すぎます。これは、Spark ドライバーが実行計画の生成に時間を要している可能性を示しています。この例では、Data Catalog から大量のパーティションを取得し、Spark ドライバーで多数の小さなファイルを一覧表示するのに時間がかかっています。



Spark UI

Spark UI の [ジョブ] タブで、[送信] 時刻を確認できます。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 count at DynamicFrame.scala:1414	2023/04/18 10:56:46	4.9 h	1/1	58100/58100

また、[ジョブ] タブでは、[タスク (すべてのステージ): 成功/合計] 時刻を確認することもできます。この場合、タスク数は 58100 と記録されています。「Parallelize tasks」ページの Amazon S3 セクションで説明されているように、タスク数は S3 オブジェクト数にほぼ対応します。つまり、Amazon S3 には約 58,100 個のオブジェクトが存在することになります。

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 ドライバーにボトルネックが発生している場合は、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、「Parallelize tasks」ページの「Too many partitions」セクションに記載されている、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、「Reduce the amount of data scan」ページの「Too many Amazon S3 partitions」セクションに記載されている、過剰なパーティショニング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、AWS Glue パーティションインデックスを有効にして、データカタログからパーティションメタデータを取得する際のレイテ

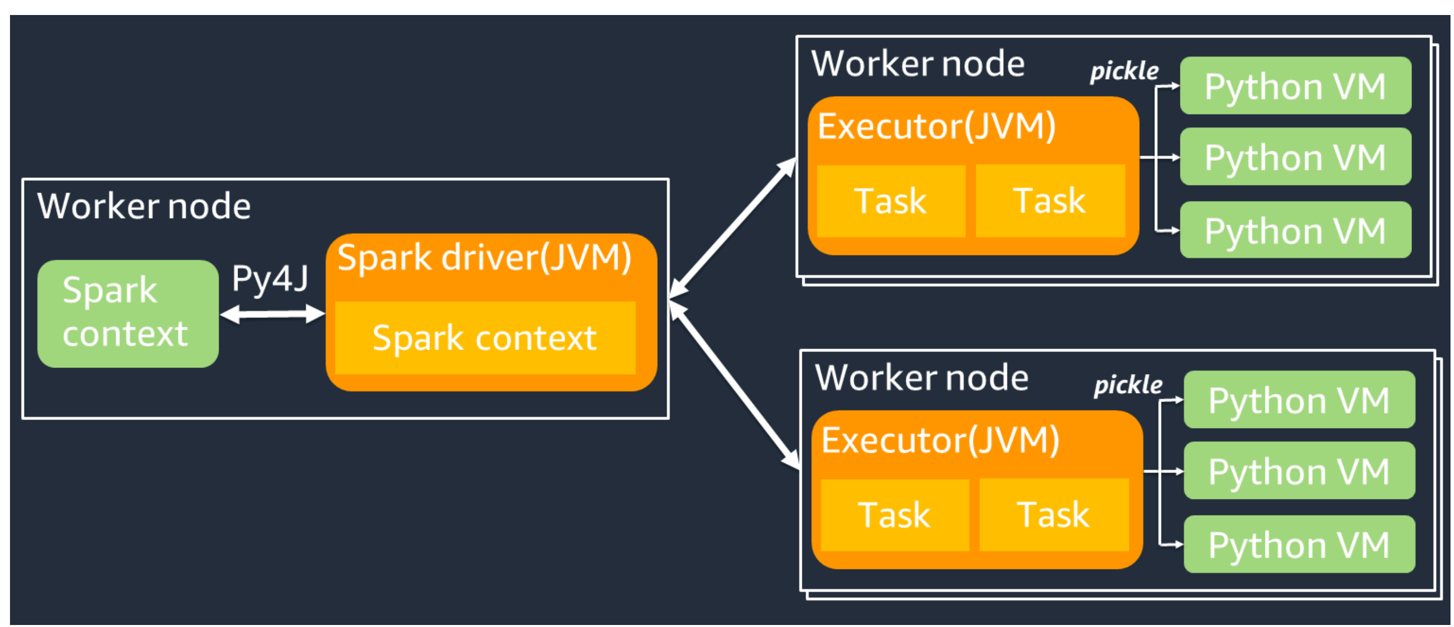
ンシーを低減します。詳細については、AWS Glue 「パーティションインデックスを使用したクエリパフォーマンスの向上」を参照してください。

- JDBC のパーティションが多すぎる場合は、hashpartition の値を小さくします。
- DynamoDB のパーティションが多すぎる場合は、dynamodb.splits の値を小さくします。
- ストリーミングジョブでパーティションが多すぎる場合は、シャード数を減らします。

ユーザー定義関数を最適化する

PySpark のユーザー定義関数 (UDF) と RDD.map は、パフォーマンスを大幅に低下させることがあります。これは、Spark の基盤となる Scala 実装で Python コードを正確に表現するために必要なオーバーヘッドが原因です。

PySpark ジョブのアーキテクチャを次の図に示します。PySpark を使用すると、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 の組み込み関数を呼び出す場合、関数は最適化された実行計画を使用して各エグゼキューターの JVM 上で実行されるため、Python と Scala の間でパフォーマンスの差はほとんどありません。



map/ mapPartitions/ udf の使用などを使った独自の Python ロジックを使用すると、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ことで、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、メモリ上のデータを変換する必要があります。Pickle は、JVM と Python のランタイム間でデータをやり取りする際に、デ

フォルトで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いため、Java または Scala で記述された UDF の方が、Python の UDF よりも高速です。

PySpark における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、次の点を検討してください。

- Spark SQL の組み込み関数を使用する – 独自の UDF やマップ関数を Spark SQL または DataFrame の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 の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 JVM 上で処理されるため、Python と Scala の間でパフォーマンスの差はほとんどありません。
- UDF を Scala または Java で実装する – Java または Scala で記述された UDF は JVM 上で実行されるため、それらの言語による実装を検討してください。
- ベクトル化されたワークロードには Apache Arrow ベースの UDF を使用する – Arrow ベースの UDF の使用を検討してください。この機能は、ベクトル化された UDF (Pandas UDF) とも呼ばれます。Apache Arrow は、JVM と Python プロセス間でデータを効率的に転送するために AWS Glue が使用できる言語に依存しないインメモリデータ形式です。これは現在、Pandas や NumPy のデータを扱う Python ユーザーにとって最も有益です。

  Arrow は列指向 (ベクトル化) 形式です。その使用は自動ではなく、最大限に活用したり互換性を確保したりするには、設定やコードにわずかな変更が必要になる可能性があります。詳細と制限については、「Apache Arrow in PySpark」を参照してください。

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

標準の Python UDF

サンプルの時間 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)
```

ベクトル化された UDF

サンプルの時間 0.59 (秒) です。

ベクトル化された UDF は、前の UDF のサンプルと比べて 5 倍高速です。`Physical Plan` を確認すると、`ArrowEvalPython` が表示されており、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がわかります。ベクトル化された UDF を有効にするには、コードで `spark.sql.execution.arrow.pyspark.enabled = true` 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

サンプルの時間 0.087 (秒) です。

タスクが Python ランタイムを介さずに各エグゼキューターの JVM 上で実行されるため、Spark SQL はベクトル化された UDF よりもはるかに高速です。UDF を組み込み関数に置き換えられる場合は、そうす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

pandas にすでに精通していて、ビッグデータに Spark を使用する場合は、Spark. AWS Glue 4.0 で pandas API 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 のドキュメントを参照してください。

リソース

- AWS Glue
- Performance Tuning (Spark SQL ガイド)
- AWS Glue Optimization Workshop

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 規範ガイダンスの用語集

以下は、 AWS 規範ガイダンスによって提供される戦略、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する。
- 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: お客様のオンプレミスの Oracle データベースを AWS クラウドの Oracle 用の Amazon Relational Database Service (Amazon RDS) に移行する。
- 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: 顧客関係管理 (CRM) システムを Salesforce.com に移行する。
- 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: お客様のオンプレミスの Oracle データベースを AWS クラウドの EC2 インスタンス上の Oracle に移行する。
- 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。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サーバーを移行します。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。
- 廃止 — 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制御」をご覧ください。

抽象化されたサービス

「マネージドユーザー」をご覧ください。

ACID

「原子性、一貫性、分離性、耐久性 (ACID)」をご覧ください。

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な方法ですが、さらに多くの作業が必要となります。

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

複数行に処理を行い、グループ全体を対象に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 MAX などがあります。

AI

「人工知能」をご覧ください。

AIOps

「AI オペレーション」をご覧ください。

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。

アプリケーション制御

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 の重要な要素であり、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?」をご覧ください。

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 (AZ)

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)

組織がクラウドへの移行を成功させるための効率的で効果的な計画を立て AWS るための、 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です。 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整理しています。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

個人や組織に混乱や損害を与えることを目的としたボット。

BCP

「ビジネス継続性計画 (BCP)」をご覧ください。

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「動作グラフのデータ」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。「エンディアン性」もご覧ください。

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

それぞれが独立しているが、同一の環境を 2 つ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を 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を別の環境 (グリーン) で実行します。この戦略は、最小限の影響で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やり取り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、個人や組織を混乱させたり、損害を与えたりすることを意図したものもあります。

ボットネット

マルウェアに感染しており、ボットハーダーまたはボットオペレーターと呼ばれる単一の当事者によって制御されているボットのネットワーク。ボットネットは、ボットとその影響力を拡大する仕組みとして、非常によく知られています。

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、「ブランチの概要」(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

例外的な状況では、承認されたプロセスを通じて、ユーザーが AWS アカウント 通常アクセス許可を持たない 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、 AWS Well-Architected ガイダンスの「ブレークグラス手順の実装」インジケータ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、AWSでのコンテナ化されたマイクロサービスの実行ホワイトペーパーの「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

カナリアデプロイ

エンドユーザーへのバージョンリリースを、時間をかけて段階的に行うこと。確信が持てたら新規バージョンをデプロイして、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

「変更データキャプチャ」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すること。AWS Fault Injection Service (AWS FIS) を使用して、 AWS ワークロードにストレスを与え、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 のサービス を受信する前のローカルでのデータの暗号化。

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 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、 AWS クラウドへの移行時に通常実行する 4 つの段階。

- 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。移行戦略との関連性については、 AWS 「移行準備ガイド」を参照してください。

CMDB

「構成管理データベース (CMDB)」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 や Bitbucket Cloud があります。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といった、ビジュアル形式の情報を分析および抽出する AI の分野。例えば、Amazon SageMaker AI では、CV 用の画像処理アルゴリズムを利用できます。

設定ドリフト

ワークロードにおいて、設定が想定した状態から変化すること。これによって、ワークロードが非準拠になる可能性がありますが、この状態は、徐々に生じ、意図的なものではありません。

構成管理データベース (CMDB)

データベースとその IT 環境 (ハードウエ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 リージョンの単一のエンティティとしてデプロイすることも、組織全体にデプロイすることもできます。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

非一元的で分散型のデータ所有権を持つとともに、一元的な管理およびガバナンスを行え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。信頼された ID のみが、期待されるネットワークから信頼されたリソースにアクセスできるようにします。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、一般的に、大量の履歴データが含まれており、多くの場合、それらはクエリや分析に使用されます。

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ます。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。

深層学習

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。

多層防御

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。たとえば、多層防御アプローチでは、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。

トラブルシューティング

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、「 AWSでのセキュリティコントロールの実装」の「検出的コントロール」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

スタースキーマにおいて、ファクトテーブルの定量データに関するデータ属性が含まれる小さいテーブル。ディメンションテーブルの属性は、通常、テキストフィールド、またはテキストのように扱える個別の数値で示されます。これらの属性は、一般的に、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)

ディザスタ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の戦略とプロセス。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:ソフトウェ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を追跡します。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するランディングゾーンの変更を検出したりできます。

DVSM

「開発バリュー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。

EDI

「電子データ交換」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、応答時間を改善できます。

電子データ交換 (EDI)

組織間で行う、ビジネスドキュメントの自動交換。詳細については、「電子データ交換とは」を参照してください。

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処理。

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 (会計、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にあるテーブル。ビジネスオペレーションに関する定量的データが保存されます。一般的に、ファクトテーブルは、2 種類の列で構成されます。1 つは測定値が含まれる列、もう 1 つはディメンションテーブルへの外部キーが含まれる列です。

フェイルファスト

開発ライフサイクルを短縮するために、頻繁かつ段階的にテストを行う哲学であり、アジャイルアプローチでは、この考え方がきわめて重要です。

障害分離境界

では AWS クラウド、障害の影響を制限し、ワークロードの耐障害性を高めるのに役立つアベイラビリティーゾーン AWS リージョン、コントロールプレーン、データプレーンなどの境界。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。

数ショットプロンプト

LLM に、タスクと望ましい出力を示す例を少数提示した後に、類似のタスクを実行させせること。この手法は、プロンプトに記述された例 (ショット) からモデルが学習する「インコンテキスト学

習」の一種です。数ショットプロンプトは、特定のフォーマット、推論、専門知識が必要なタスクに効果的です。「ゼロショットプロンプト」も参照してください。

FGAC

「きめ細かなアクセス制御」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

変更データのキャプチャによる継続的なデータ複製を利用して、段階的なアプローチではなく、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。

FM

「基盤モデル」を参照してください。

基盤モデル (FM)

大規模な深層学習ニューラルネットワークであり、一般化およびラベル付けされていないデータからなる大規模データセットでトレーニングされています。FM により、言語理解、テキストおよび画像生成、自然言語での会話といった、一般的な各種タスクを実行できます。詳細については、「基盤モデルとは何ですか?」を参照してください。

G

生成 AI

AI モデルのサブセット。大量のデータでトレーニングされており、シンプルなテキストプロンプトを使用して、画像、動画、テキスト、オーディオなどの新しいコンテンツやアーティファクトを作成できます。詳細については、「生成 AI とは何ですか?」を参照してください。

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)

特定の国のユーザーがコンテンツ配信にアクセスできないようにするための、Amazon CloudFront のオプション。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 ワークフローは古いと見なされている方法であり、トランクベースのワークフローは推奨されている新しい方法です。

ゴールデンイメージ

システムまたはソフトウェアのスナップショットであり、システムまたはソフトウェアの新規インスタンスをデプロイするテンプレートとして使用されます。製造の例で言えば、ゴールデンイメージを使用すると、複数のデバイスにソフトウェアをプロビジョニングして、デバイス製造オペレーションの速度、スケーラビリティ、生産性を向上させることができます。

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 GuardDuty AWS Security Hub CSPM、、 AWS Trusted Advisor Amazon Inspector、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。

高可用性 (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。

ホールドアウトデータ

機械学習モデルのトレーニング用データセットから保留される、ラベル付き履歴データの一部。ホールドアウトデータを使用すると、モデル予測をホールドアウトデータと比較して、モデルのパフォーマンスを評価できます。

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。緊急性が高いため、通常の DevOps のリリースワークフローからは外れた形で実施されます。

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

「インダストリアル IoT」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

既存インフラストラクチャの更新、パッチ適用、変更などを行わずに、本番環境ワークロードに使用する新規インフラストラクチャをデプロイするモデル。本質的に、イミュータブルインフラストラクチャは、ミュータブルインフラストラクチャよりも一貫性、信頼性、予測性に優れています。詳細については、 AWS Well-Architected フレームワークにある「イミュータブルインフラストラクチャを使用してデプロイする」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

2016 年に Klaus Schwab 氏が提唱した用語で、接続、リアルタイムデータ、オートメーション、分析、AI/ML の進歩による、ビジネスプロセスのモダナイズを意味します。

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。

インダストリアル 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、「インダストリアル IoT (IIoT) デジタルトランスフォーメーション戦略の構築」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、「 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。

IoT

「IoT」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。

大規模言語モデル (LLM)

大量のデータで事前トレーニングされた深層学習 AI モデル。LLM では、質問への回答、ドキュメントの要約、他言語へのテキスト翻訳、文を完成させるなど、さまざまなタスクを実行できます。詳細については、「大規模言語モデル (LLM) とは何ですか?」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

「ラベルベースアクセス制御」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 Rs」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。「エンディアン性」もご覧ください。

LLM

「大規模言語モデル」を参照してください。

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やプライバシーを侵害するように設計されたソフトウエア。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、不正アクセスを招く可能性があります。マルウェアの例には、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 のサービス は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、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。マネージドサービスの例として、Amazon Simple Storage Service (Amazon S3) と Amazon DynamoDB が挙げられます。このサービスは、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、制御するソフトウェアシステムであり、工場では、これによって、原材料から製品を完成させます。

MAP

「Migration Acceleration Program」を参照してください。

メカニズム

ツールを作成してその導入を推進し、導入結果を調べて調整を行うための包括的なプロセス。メカニズムとは、運用中にそれ自体を強化し改善するサイクルを意味します。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

組織の一部である管理アカウント AWS アカウント 以外のすべて 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。

Message Queuing Telemetry Transport (MQTT)

発行/サブスクライブのパターンに基づく、軽量のマシンツーマシン (M2M) 通信プロトコルであり、リソースに限りのある IoT デバイスに使用されます。

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)

組織がクラウドに移行するための強力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つ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。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、運用、ビジネスアナリストおよび所有者、移行エンジニア、デベロッパー、およびスプリントで作業する 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としては、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: AWS Application Migration Service を使用して Amazon EC2 への移行をリホストします。

Migration Portfolio Assessment (MPA)

オンラインツール。これによって、 AWS クラウドに移行するビジネスケースの検証に必要な情報を得られます。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 AWS クラウドに移行するために使用するアプローチ。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、「AWS クラウドでのアプリケーションのモダナイズ戦略」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、「AWS クラウドで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評価する」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

「Migration Portfolio Assessment」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。

ミュータブルなインフラストラクチャ

本番ワークロードに使用する既存のインフラストラクチャを更新および変更するためのモデル。Well-Architected AWS フレームワークでは、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

「オリジンアクセス制御」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

「オペレーショナルレベルアグリーメント」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。

Open Process Communications - Unified Architecture (OPC-UA)

産業オートメーション用のマシンツーマシン (M2M) 通信プロトコル。OPC-UA により、相互運用の際に、データ暗号化、認証、認可の各スキームを標準化できます。

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)

質問と関連するベストプラクティスのチェックリスト。インシデントや起こり得る障害を理解、評価、防止したり、その範囲を縮小したりする際に役立ちます。詳細については、 AWS Well-Architected フレームワークの「Operational Readiness Reviews (ORR)」を参照してください。

運用テクノロジー (OT)

産業オペレーション、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させるハードウェアおよびソフトウェアシステム。製造分野では、Industry 4.0 への変革を進める上で、OT と情報技術 (IT) システムの統合に焦点が当てられています。

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

組織 AWS アカウント 内のすべての のすべてのイベント AWS CloudTrail をログに記録する 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、CloudTrail ドキュメントの「組織の証跡の作成」を参照してください。

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により、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。

オリジンアクセス制御 (OAC)

Amazon Simple Storage Service (Amazon S3) コンテンツを保護するための、CloudFront のアクセス制限の強化オプション。OAC は AWS リージョン、すべての S3 バケット、 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)

CloudFront の、Amazon S3 コンテンツを保護するためのアクセス制限オプション。OAI を使用すると、CloudFront が、Amazon S3 に認証可能なプリンシパルを作成します。認証されたプリンシパルは、S3 バケット内のコンテンツに、特定の CloudFront ディストリビューションを介してのみアクセスできます。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制御が可能です。

ORR

「運用準備状況レビュー」を参照してください。

OT

「運用テクノロジー」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た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

「プログラマブル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

次の操作を可能にするオブジェクト: アクセス許可を定義する (ID ベースのポリシーを参照)。アクセス条件を指定する (リソースベースのポリシーを参照)。 AWS Organizations の組織における全アカウントにアクセス許可の上限を定義する (サービスコントロールポリシーを参照)。

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。

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、「移行の準備状況の評価」を参照してください。

述語

`true` または `false` を返すためのクエリ条件。一般的に、`WHERE` 句に記述されます。

述語プッシュダウン

データベースクエリを最適化する手法。これによって、転送前にクエリ内のデータをフィルタリングします。この手法を取ると、リレーショナルデータベースから取得し処理する必要のあるデータの量が減少するため、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、「AWSでのセキュリティコントロールの実装」の「予防的コントロール」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 AWS アカウントロール、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」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

開発プロセス全体を通してプライバシーが考慮されている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

非準拠リソースのデプロイ防止を目的としたセキュリティコントロール。このコントロールにより、プロビジョニング前にリソースをスキャンします。コントロールに準拠していないリソースは、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「 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)

製品の設計、開発、発売から、成長、成熟、衰退、廃棄に至る、製品のライフサイクル全体を通してデータとプロセスを管理すること。

本番環境

「環境」を参照してください。

プログラマブルロジックコントローラー (PLC)

製造分野で使用される、信頼性と適応性に優れたコンピュータであり、これによって、マシンをモニタリングするとともに、製造プロセスを自動化します。

プロンプトチェイニング

1 つの LLM プロンプトによる出力を次のプロンプトの入力に使用して、より良いレスポンスを生成します。この手法を使用すると、複雑なタスクをサブタスクに分割したり、事前レスポンスを繰り返し改良または拡張したりできます。これによって、モデルのレスポンスの精度と関連性が向上し、粒度の高いパーソナライズされた結果を得られます。

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。

発行/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。これにより、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 の場合、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可能なチャネルにイベントメッセージを発行できます。このシステムでは、発行サービスの変更なしに、新規マイクロサービスを追加できます。

Q

クエリプラン

手順などの一連のステップであり、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ます。

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

「実行責任者、説明責任者、協業先、報告先 (RACI)」を参照してください。

RAG

「検索拡張生成」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

「実行責任者、説明責任者、協業先、報告先 (RACI)」を参照してください。

RCAC

「行と列のアクセス制御」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。

リアーキテクト

「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 から分離され、独立しています。詳細については、「アカウントが使用できる AWS リージョン を指定する」を参照してください。

リグレッション

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 Rs」を参照してください。

リプラットフォーム

「7 Rs」を参照してください。

再購入

「7 Rs」を参照してください。

回復性

中断に抵抗または中断から回復するアプリケーションの機能。 AWS クラウドでの回復力を計画する際には、一般的に、高可用性とディザスタリカバリが考慮されます。詳細については、「AWS クラウド の耐障害性」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。サポートが含まれる場合は RASCI マトリックスと呼ばれ、含まれない場合は 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、「 AWSでのセキュリティコントロールの実装」の「レスポンシブコントロール」を参照してください。

保持

「7 Rs」を参照してください。

廃止

「7 Rs」を参照してください。

検索拡張生成 (RAG)

生成 AI の技術。これにより、LLM では、レスポンスの生成前に、トレーニングデータソースの外部にある信頼できるデータソースが参照されます。例えば、RAG モデルによって、組織のナレッジベースまたはカスタムデータのセマンティック検索を実行できる場合があります。細については、「RAG (検索拡張生成) とは何ですか?」を参照してください。

ローテーション

定期的にシークレット情報を更新して、攻撃者が認証情報にアクセスするのをより困難にするプロセス。

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 (IdP) が使用しているオープンスタンダード。この機能を使用すると、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、ユーザーは組織内のすべてのユーザーを IAM で作成しなくても、 AWS マネジメントコンソール にログインしたり 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

「監視制御とデータ取得」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

暗号化された形式で保存する AWS Secrets Manager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。シークレット値には、バイナリ、1 つの文字列、複数の文字列を指定できます。詳細については、Secrets Manager ドキュメントの「Secrets Manager シークレットの概要」を参照してください。

セキュリティバイデザイン

開発プロセス全体を通してセキュリティが考慮されているシステムエンジニアリングのアプローチ。

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、主に 4 つの種類があります。4 つとは、予防、検出、レスポンシブ、プロアクティブです。

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への自動レスポンスまたは自動修復を目的として、事前定義および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を実装するの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には、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、認証情報の更新などがあります。

サーバー側の暗号化

送信先で、それ AWS のサービス を受け取る によるデータの暗号化。

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 のサービス。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 のサービス 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、スループットといった、サービスパフォーマンス面の指標。

サービスレベル目標 (SLO)

サービスレベルインジケータによって測定され、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、 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

「Security Information and Event Management システム」を参照してください。

単一障害点 (SPOF)

特定のアプリケーションを構成する単一の重要なコンポーネントで発生し、システム稼働に支障をきたす可能性のある障害。

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。

スプリットアンドシードモデル

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、「AWS クラウドで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」を参照してください。

SPOF

「単一障害点」を参照してください。

スタースキーマ

データベースの編成構造を意味し、1 つの大きいファクトテーブルにトランザクションデータまたは測定データが保存され、1 つ以上の小さいディメンションテーブルにデータ属性が保存されます。この構造は、データウェアハウスやビジネスインテリジェンスを用途と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。

監視制御とデータ取得 (SCADA)

製造分野において、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本番運用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

ユーザーとのやり取りをシミュレートして、起こり得る問題を検出したり、パフォーマンスをモニタリングしたりすることで、システムをテストします。Amazon CloudWatch Synthetics を使用すると、こうしたテストを作成できます。

システムプロンプト

コンテキスト、指示、ガイドラインなどを提示して、LLM に動作を指示する手法。システムプロンプトは、コンテキストを設定して、ユーザーとやり取りするルールを確立するのに有用です。

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したサービスに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「Using AWS Organizations with other AWS services 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 枚のピザを分け合えることができるくらい小さな 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

ヘ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

現在のレコードに何らかの形で関連している行のグループに計算を実行する SQL 関数。ウィンドウ関数は、移動平均を計算したり、現在の行の相対位置に基づいて他の行の値にアクセスするといった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

「Write-Once-Read-Many」を参照してください。

WQF

「AWS ワークロード資格フレームワーク」を参照してください

Write-Once-Read-Many (WORM)

データを 1 回のみ書き込むことで、データの削除や変更を防ぐストレージモデル。承認済みユーザーは、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

ゼロデイ脆弱性を悪用した攻撃 (一般的にマルウェアによる)。

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。

ゼロショットプロンプト

LLM にタスク実行の手順は提示するが、実行のガイドとして役立つ例 (ショット) は提示しない方法。LLM は、事前トレーニング済みの知識を使用してタスクを処理する必要があります。ゼロショットプロンプトの有効性は、タスクの複雑さとプロンプトの品質によって異なります。「数ショットプロンプト」も参照してください。

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。