AWS Glue ETL パフォーマンス・チューニング② チューニングパターン編

AWS Black Belt Online Seminar

林田 千瑛
Solution Architect
2021/08

AWS 公式 Webinar
https://amzn.to/JPWebinar

過去資料
https://amzn.to/JPArchive

AWS Black Belt Online Seminarとは

- 「サービス別」「ソリューション別」「業種別」のそれぞれのテーマに分け、アマゾン ウェブ サービス ジャパン株式会社が主催するオンラインセミナーシリーズです。
- AWSの技術担当者が、AWSの各サービスについてテーマごとに動画を公開します
- お好きな時間、お好きな場所でご受講いただけるオンデマンド形式です
- 動画を一時停止・スキップすることで、興味がある分野・項目だけの聴講も可能、スキマ時間の学習にもお役立ていただけます

内容についての注意点

- 本資料では2021年8月時点のサービス内容および価格についてご説明しています。
  最新の情報はAWS公式ウェブサイト(http://aws.amazon.com)にてご確認ください。
- 資料作成には十分注意しておりますが、資料内の価格とAWS公式ウェブサイト記載の価格に相違があった場合、AWS公式ウェブサイトの価格を優先とさせていただきます。
- 価格は税抜表記となっています。
  日本居住者のお客様には別途消費税をご請求させていただきます。

- AWS does not offer binding price quotes. AWS pricing is publicly available and is subject to change in accordance with the AWS Customer Agreement available at http://aws.amazon.com/agreement/. Any pricing information included in this document is provided only as an estimate of usage charges for AWS services based on certain information that you have provided. Monthly charges will be based on your actual use of AWS services, and may vary from the estimates provided.

自己紹介

林田 千瑛（Chie Hayashida）

アマゾンウェブサービスジャパン
ソリューションアーキテクト

aws

はじめに

- 本セッションは、
  『**AWS Glue ETL パフォーマンス・チューニング**』
  シリーズの後半、『**チューニングパターン編**』です。
- 前半の、『**基礎知識編**』の続きです。
- 後半となる本セミナーでは、具体的なAWS Glue ETLのパフォーマンスチューニングのチューニングパターンを解説します
- 本セッションは、AWS Glue 2.0（Spark 2.4.3）とGlue 3.0（Spark 3.1.1）を対象としています。

はじめに

- 本セッションは、
  『AWS Glue ETL パフォーマンス・チューニング① 基礎知識編』 の続編です

本セミナーの対象者

- AWS GlueのAWS Black Belt Online Seminarの内容が理解できている
- AWS Glue ETL パフォーマンス・チューニング 基礎知識編の内容が理解できている
- Sparkアプリケーションを書いたことがある
- 今あるAWS Glue ETL ジョブを改善したい

※ AWS Glueのお客様のうち多くがPySparkをご利用のため、
本資料ではコードはPySparkで書いています

アジェンダ

- AWS Glue ETL ジョブのチューニングの基本戦略
- AWS Glue ETL ジョブのチューニングパターン

aws

AWS Glue ETL ジョブのチューニングの基本戦略

- 新しいバージョンを利用する
- データのI/O負荷を小さくする
- シャッフルを最小限にする
- タスク単位の処理を高速化する
- 並列化する

aws

パフォーマンス・チューニングのサイクル[再掲]

1. パフォーマンス目標を決める
2. メトリクスを測定する
3. ボトルネックを特定する
4. ボトルネックの影響を軽減する
5. 目標を達成するまで2. から 4. を繰り返す
6. パフォーマンス目標の達成

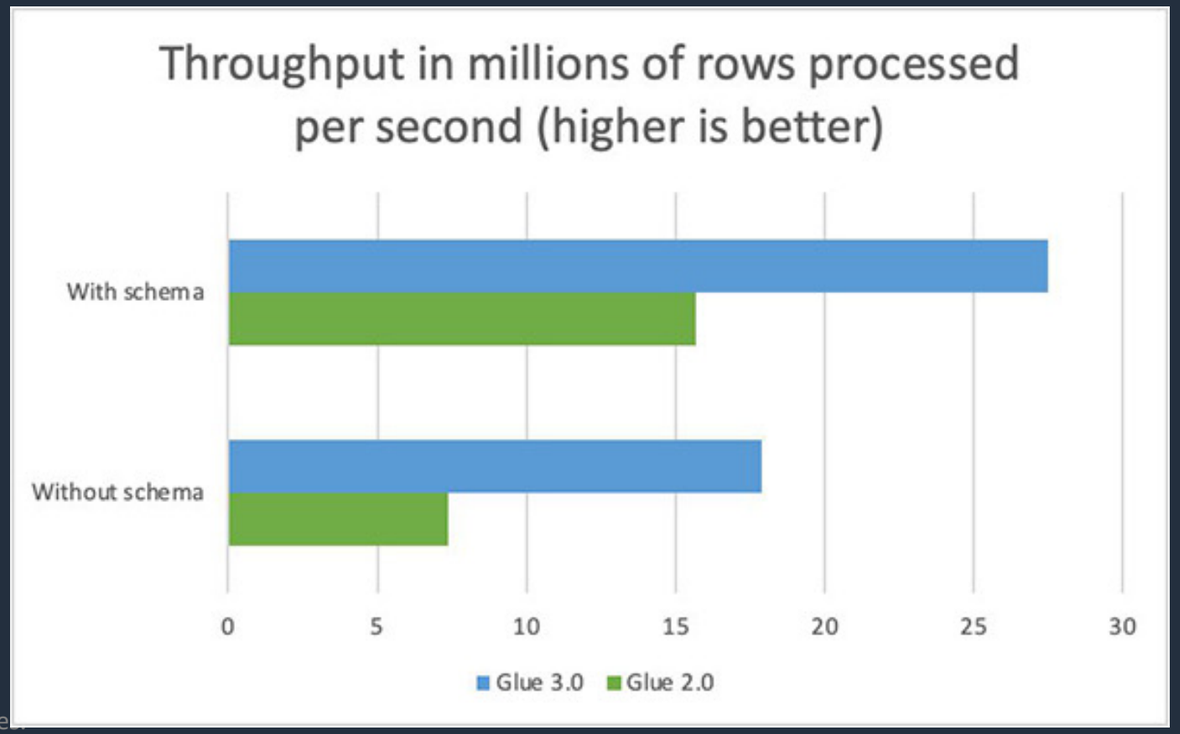
aws

新しいバージョンを利用する

aws

新しいバージョンを利用する

- 処理が遅いと感じるときに、ジョブ実行環境を最新バージョンに置き換えるだけで、処理が高速化する場合がある。
- AWS GlueもApache Sparkも、パフォーマンスだけでなくあらゆる点で進化しています。できるだけ新しいバージョンを利用する。



https://aws.amazon.com/jp/blogs/big-data/introducing-aws-glue-3-0-with-optimized-apache-spark-3-1-runtime-for-faster-data-integration/

AWS Glue 2.0、3.0を利用して起動時間を削減

AWS Glue ETL ジョブの起動に要する時間を大幅に短縮

- 従来、AWS Glue 0.9/1.0 ではコールドスタートに 8分程度かかっていた
- 新しい AWS Glue 2.0 では 1分未満に

データI/Oを最小限にする

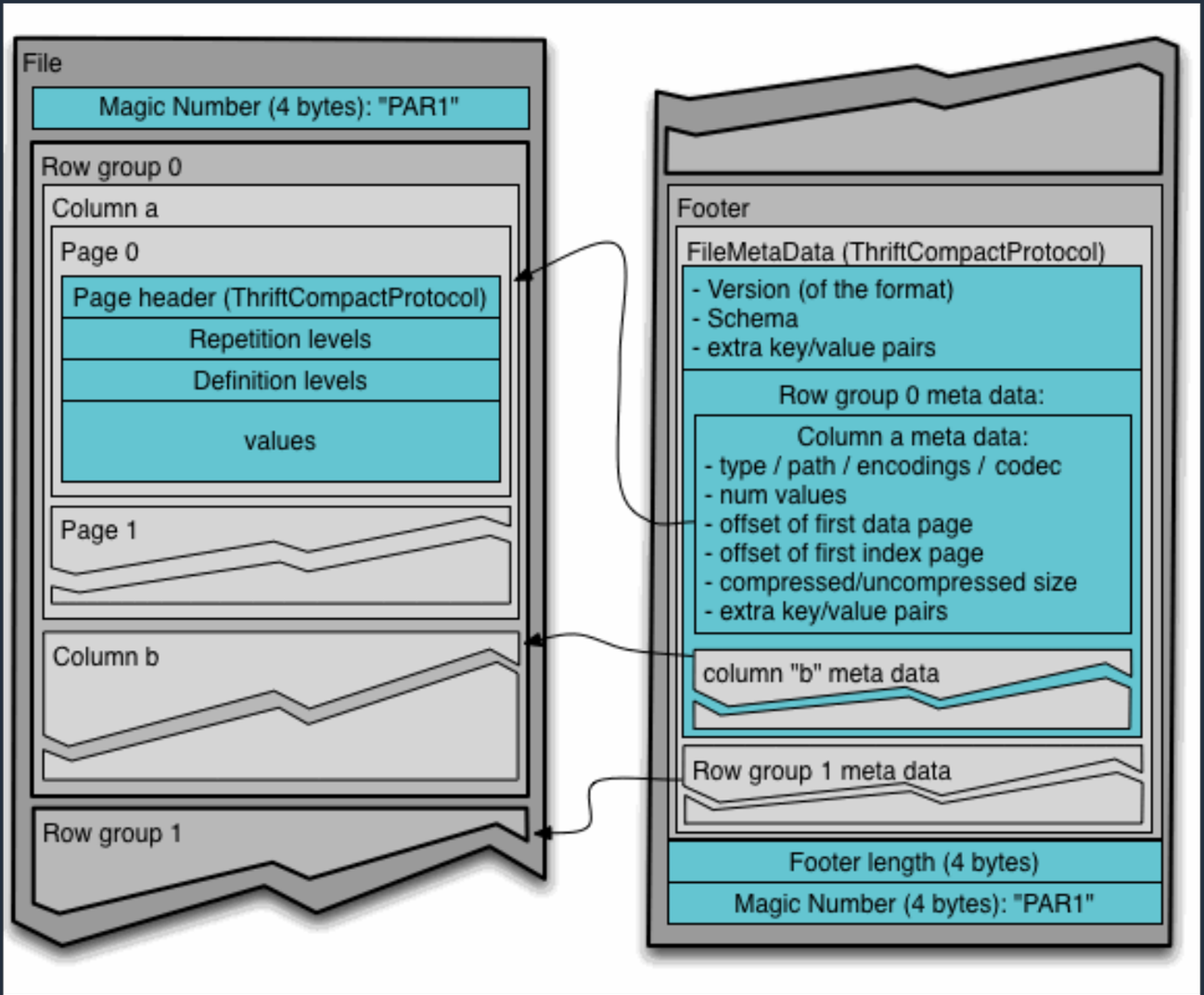
aws

データI/O負荷を最小限にする方法

- 必要なデータだけ読む
- 1taskで読むデータ量をコントロールする
- 適切な圧縮形式を選ぶ

aws

Apache Parquetを使用する



- 列指向フォーマットであり、分析用途に向いたデータ配置
- データ型が保持される
- 圧縮が効く
- 不要なデータの読み飛ばしやメタデータを利用した集計
- SparkエンジンではApache Parquetを効率的に利用できるインテグレーションが行われている

https://parquet.apache.org/documentation/latest/

Partition FilteringとFilter Pushdown

読み取りデータ量を削減する機能

- Partition Filtering
  - filter句やWhere句で指定されたパーティション内のファイルのみを読み取る機能
  - Text/CSV/JSON/ORC/Parquetで利用可能

- Filter Pushdown
  - パーティション列に利用されていない列に対するfilter句やwhere句にヒットするブロックのみを読み取る機能
  - AWS GlueではParquetを利用した場合に自動的に適用される

aws

Partition FilterとFilter Pushdown

```
In [9]: from pyspark.sql.functions import col
(
    spark.read.parquet(
        's3://[redacted]/amazon_customer_reviews_product_partitioned')
    .select('product_category', 'star_rating')
    .filter('star_rating>3').filter(col('product_category')=='Video')
).explain()
```

```
== Physical Plan ==
*(1) Project [product_category#173, star_rating#165]
+- *(1) Filter (isnotnull(star_rating#165) && (star_rating#165 > 3))
   +- *(1) FileScan parquet [star_rating#165,product_category#173] Batched: true, Format: Parquet, Location: InMemoryFileIndex[s3://[redacted]/amazon_customer_reviews_product_partitioned], PartitionCount: 1, PartitionFilters: [isnotnull(product_category#173), (product_category#173 = Video)], PushedFilters: [IsNotNull(star_rating), GreaterThan(star_rating,3)], ReadSchema: struct<star_rating:int>
```

Partition Filtering

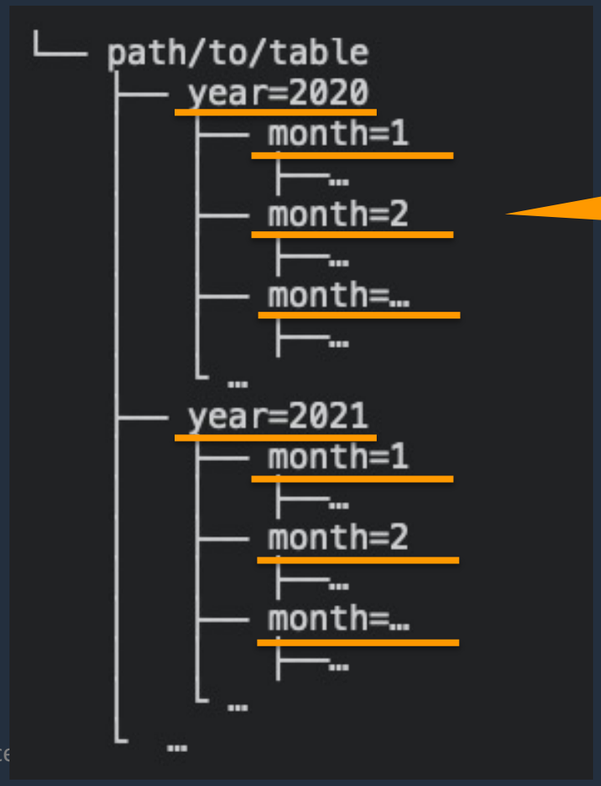
Filter Pushdown

aws

Partition FilterとFilter Pushdown

Partition Filterはパーティションディレクトリが作成されている場合に利用可能。DataframeやDynamicFrameの書き込みでは以下のようにpartitionByオプションを利用することでパーティションディレクトリが作成される

```
df.write.parquet(path=path, partitionBy='col_name')
```



filter句で利用される頻度が高い列を上位のパーティションにするほうがよりパフォーマンス効率がよくなる可能性がある

DynamicFrameのpush_down_predicateの利用

DynamicFrameをAWS Glue データカタログから読み込む際にfilter句やwhere句で指定したデータが格納されるパーティションに含まれるファイルのみを読み取る機能

```
partitionPredicate =”(product_category == ’Video’)”
datasource = glue_context.create_dynamic_frame.from_catalog(
    database = "githubarchive_month",
    table_name = "data",
    push_down_predicate = partitionPredicate)
```


aws

用途に応じて圧縮コーデックを選ぶ

- データ書き込み時に圧縮コーデックを選ぶことができる
- 圧縮率と圧縮/解凍速度はトレードオフ
- bzip2, lzo, snappyで圧縮されたファイルは読み込み時に分割して処理することができるが、gzip(※)で圧縮されたファイルは分割できない
- 非圧縮の場合は圧縮/解凍の時間がかからないが、データ転送コストがボトルネックになる可能性がある
- 処理スピードを重視するのであればsnappyかlzoを選択する

ex. df.write.csv("path/to/csv", compression="gzip")

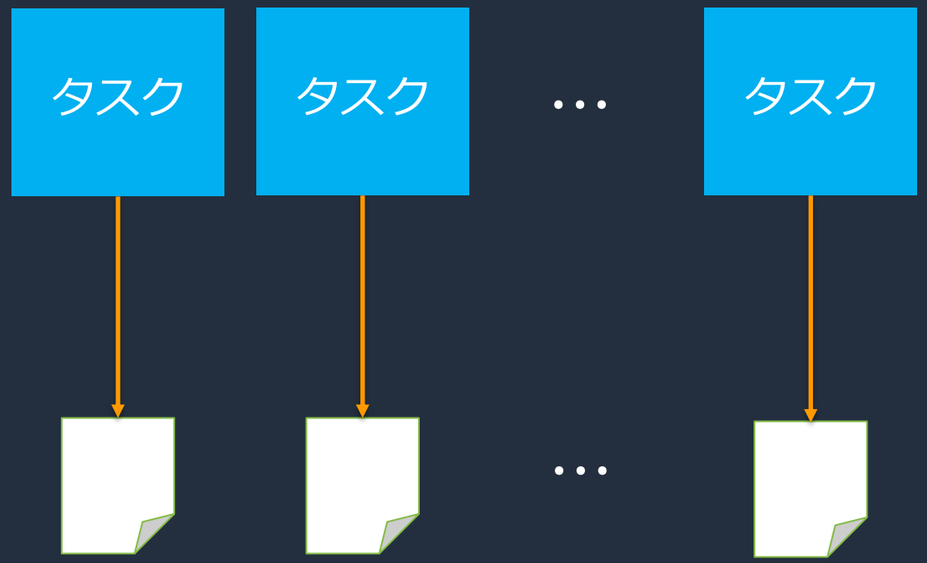
	gzip	bzip2	lzo	snappy
file extension	.gzip	.bz2	.lzo	.snappy
Compression Level	High	Highest	Average	Average
Speed	Medium	Slow	Fast	Fast
CPU usage	Medium	High	Low	Low
Is Splittable	No	No(※)	Yes, if indexed	No

※ Parquetの場合はgzipでも分割できる

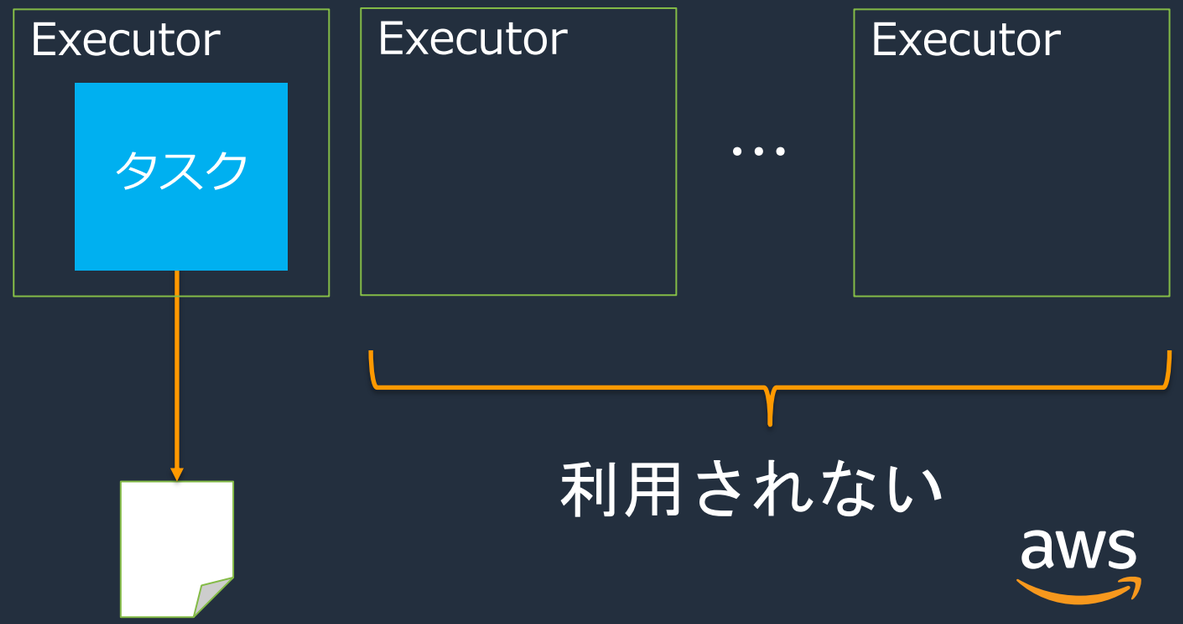
適切なファイルサイズでデータを格納する

- データの読み取り/書き込みのタスクは、基本的に1ファイルに紐づく
  (ファイルがsplitableな場合は1ファイルを複数タスクに分割できる)
- AWS Glueで扱うファイルサイズは128MB-512MBを推奨

データが小さすぎるとき
- 小さいタスクが大量に発生することでオーバヘッドになる



大きなunspilitableデータが1ファイルにあるとき
- 1つのノード上でメモリにデータが乗り切らない
- 分散処理されない



DynamicFrameでBounded Executionを利用する

読み込みデータが多い場合に、ジョブブックマークと同時にBounded Executionを利用することで、未処理のデータを一度に読み込まずに処理を分割することができる

```
glueContext.create_dynamic_frame.from_catalog(
    database = "database",
    tableName = "table_name",
    redshift_tmp_dir = "",
    transformation_ctx = "datasource0",
    additional_options = {
        "boundedFiles" : "500", # need to be string
        # "boundedSize" : "1000000000" unit is byte
    }
)
```


aws

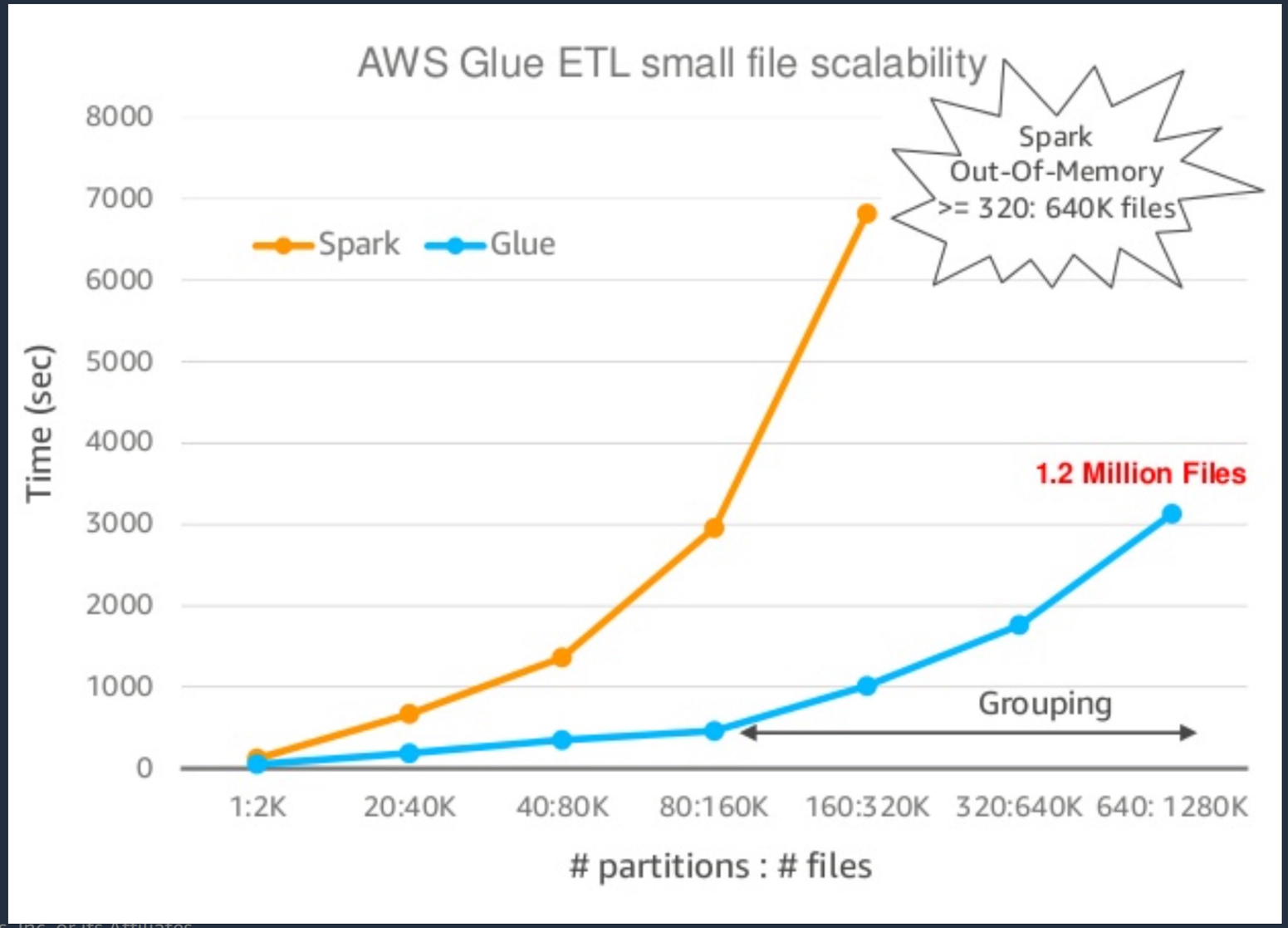
DynamicFrameのgroupFiles, groupSizeの利用

- 小さいファイルをまとめて1つのタスクで読み取ることで、オーバヘッドを解消する
- Kinesis Data Firehoseで数分毎に出力されるデータに対する処理を行うときなどに有効
- S3パーティション内のデータをグループ化する際はgroupFiles、読み取るグループのサイズをgroupSizeオプションで指定する

```
df =  glueContext.create_dynamic_frame_from_options(
   's3', {'paths': ['s3://s3path/'],
   'recurse':True, 'groupFiles': 'inPartition', 'groupSize': '1048576'}, format='json')
```


aws

DataFrameとDynamicFrameでのファイル数と処理時間



DynamicFrameのS3ListImplementationの利用

- 小さいファイルが大量にある場合、大量のタスクによってDriverのOOMを引き起こす可能性がある
- S3ListImplementationがTrueの場合、S3 listの結果を1000ずつのバッチで読み込んで処理を行うため、S3 listingによるドライバメモリの逼迫を防ぐことができる。

```
datasource = glue_context.create_dynamic_frame.from_catalog(
    database = ”my_database",
    table_name = ”my_table",
    push_down_predicate = partitionPredicate,
    additional_options = {"useS3ListImplementation":True} )
```

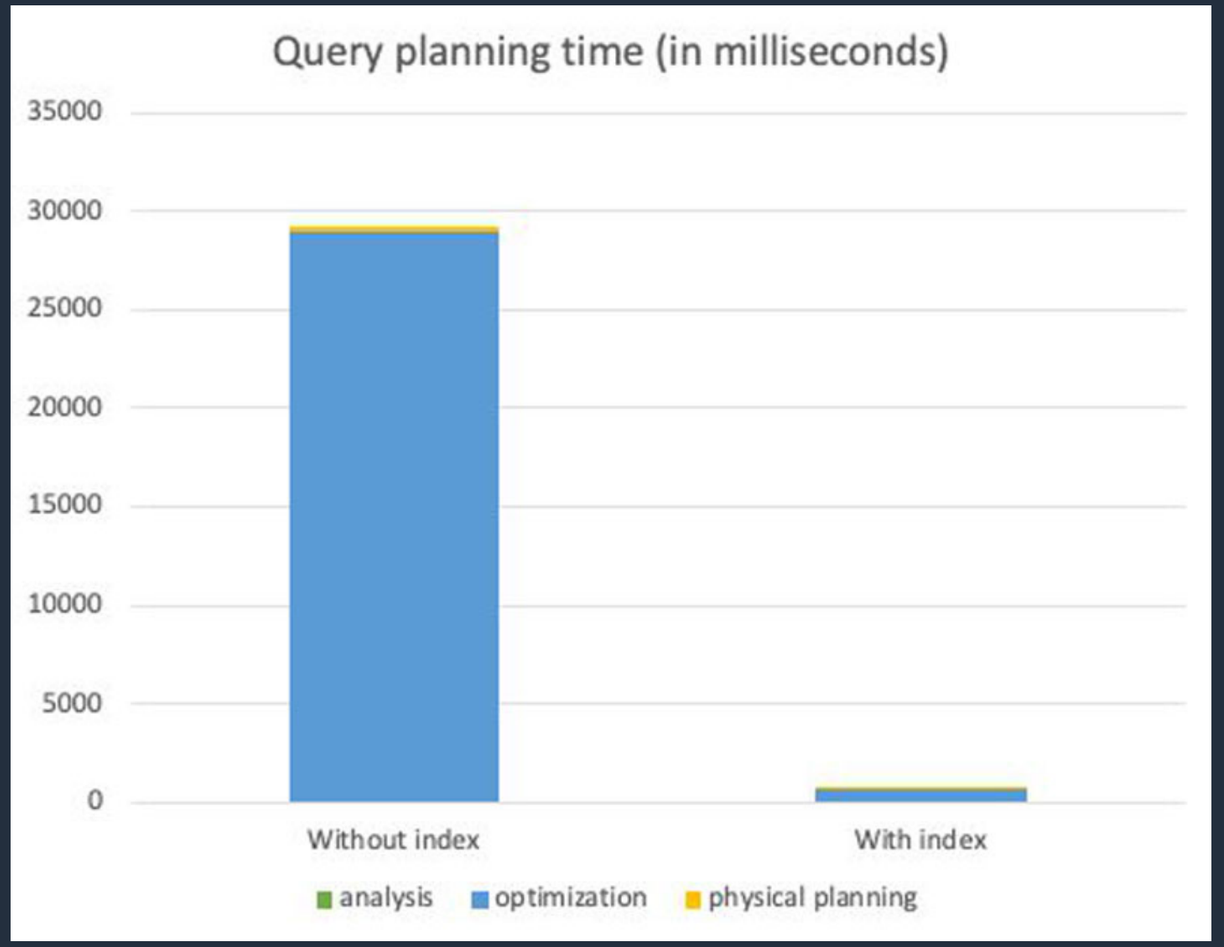

aws

Partition Indexを設定する

複数のパーティションキーからなる多くのパーティションをもつデータソースから、DataFrameをAWS Glueカタログから読み込むときに、Partition Indexを設定しておくことで、対象のパーティションに対するfilter句やwhere句がある場合に読み込みパーティションをフェッチする時間を削減することができる

https://aws.amazon.com/jp/blogs/big-data/improve-query-performance-using-aws-glue-partition-indexes/

Partition Indexを利用した場合としない場合のクエリプランニング時間の違いの例



https://aws.amazon.com/jp/blogs/big-data/improve-query-performance-using-aws-glue-partition-indexes/

DataFrameのJDBC接続における並列データ読み取り

- spark.read.jdbc()はデフォルトで1つのExecutorのみがターゲットデータベースにアクセスする。
- 並列読み取りを行うためにはpartitionColumn, lowBound, upperBound, numPartitionsの指定が必要。このPartitionColumnは数値・日付・タイムスタンプにいずれかの型である必要がある

```
df = spark.read.jdbc(
    url=jdbcUrl, table="sample",
    partitionColumn="col1",
    lowerBound=1L,
    upperBound=100000L,
    numPartitions=100,
    fetchsize=1000,
    connectionProperties=connectionProperties)
```


aws

DynamicFrameのJDBC接続における並列データ読み取り

- DynamicFrameとしてJDBC接続からデータを読み取る場合は、hashfield/hashexpressionの指定を行う必要がある。
- hashfieldでは文字列などの列もパーティション列として利用できる。

```
glueContext.create_dynamic_frame.from_catalog(
    database = "my_database",
    tableName = "my_table_name",
    transformation_ctx = "my_transformation_context",
    additional_options = { 'hashfield’: ‘customer_name’, 'hashpartitions': '5’ } )
```

https://docs.aws.amazon.com/glue/latest/dg/run-jdbc-parallel-read-job.html

aws

シャッフルを最小限にする

aws

シャッフルを最小限にする

- cacheをうまく利用する
- filter処理をなるべく前段で行う
- データを小さく保つようにjoinの順番を工夫する
- joinを使い分ける
- データの偏りへの対処
- self joinによるデータ加工の代わりにWindow処理を利用する

aws

シャッフルを最小限にする

DataFrameで記述された処理は、Catalyst Optimizerによって最適化が行われる。
しかし、以下の観点で完璧ではない

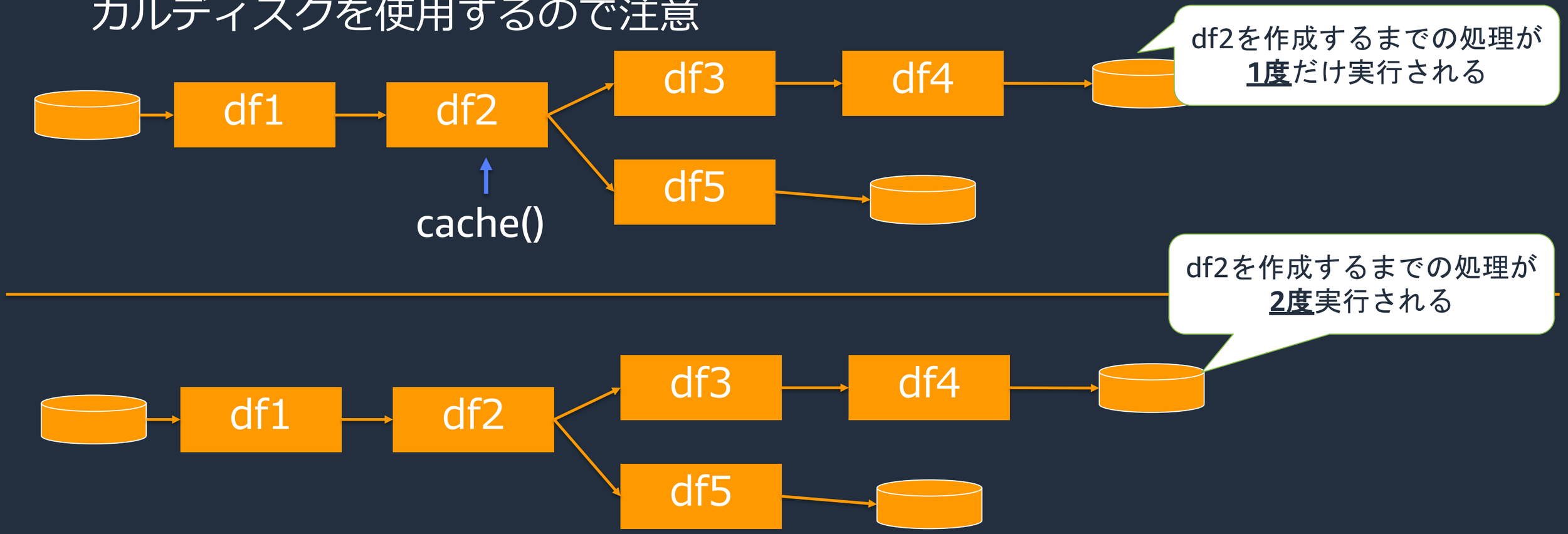
- 間にcache()などが挟まるとその前後を含めた最適化はきかない
- AWS Glue 1.0や2.0で利用されるSpark2.4.3は、デフォルトでコストベースオプティマイザは無効

filterやjoinの順番やストラテジをマニュアルで変更することで、シャッフルを減らすことができる

aws

cacheをうまく利用する

- ひとつのDataframeに対する処理を分岐させて複数のアウトプットを行う場合は、分岐の直前にcache()を入れることで再計算を防ぐことができる
- cacheを利用しないほうが速い場合もあり、またcacheを多用しすぎるとローカルディスクを使用するので注意



cacheをうまく利用する

- キャッシュされたデータはデフォルトでは最初にキャッシュに利用できるように割り当てられたメモリ上に、メモリ上にのらない分はローカルディスク上に永続化される。
- オプションでメモリのみやディスクのみに保存することも選択できる

メモリ上にのみcacheする例：df.cache(MEMORY_ONLY)

aws

使わなくなったcacheは削除する

- cacheしたDataframeはメモリやローカルディスクを占有し続ける
- 不要になったDataframeのキャッシュを削除することで、メモリやディスクを節約できる

```
df.unpersist()
```


aws

filter処理をなるべく前段で行う

In [2]:
```
df1=spark.read.parquet(
    's3://chie-us-east-2/tmp/filtered_amazon_customer_reviewed')
df1.persist()
df2 = df1.filter(df1.product_category == 'Video')
df2.explain()
```

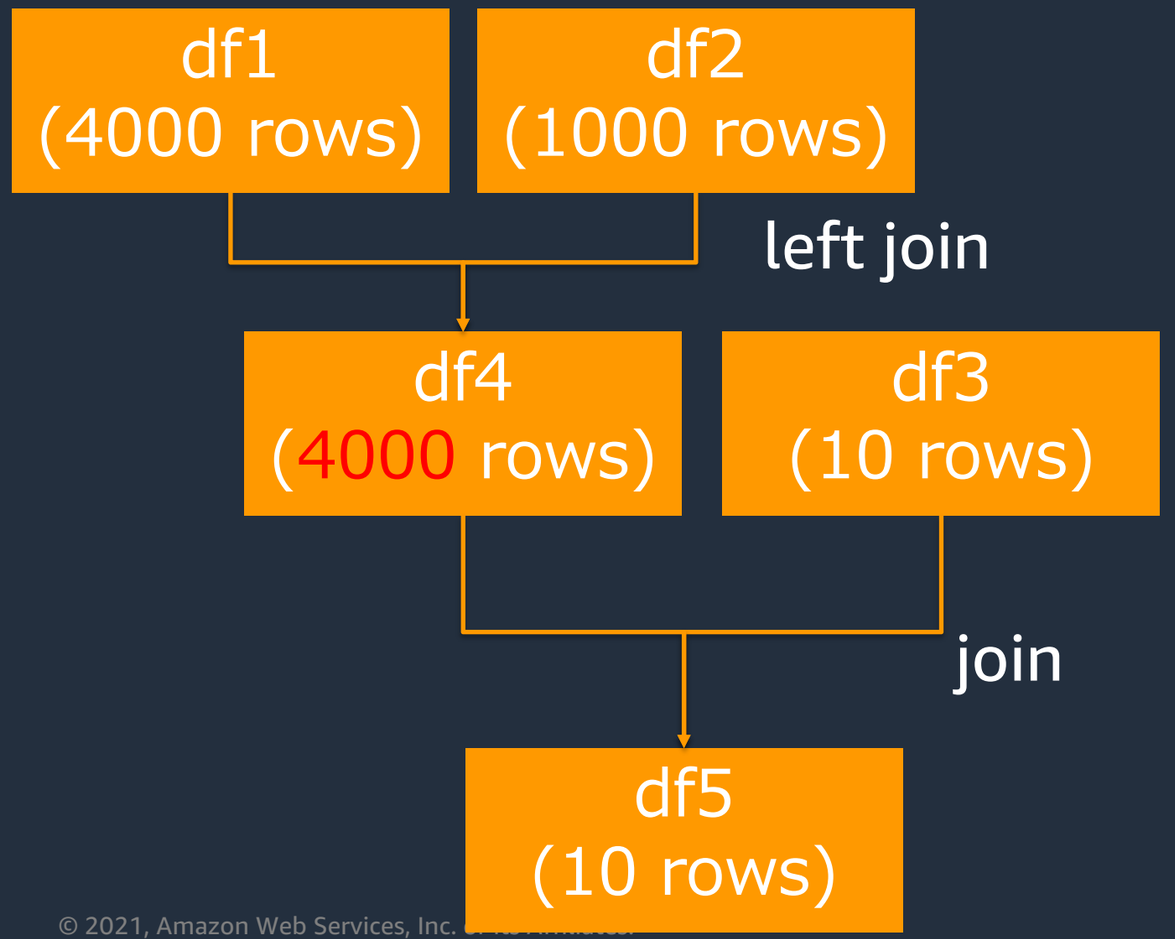
```
== Physical Plan ==
*(1) Filter (isnotnull(product_category#3) && (product_category#3 = Video))
+- InMemoryTableScan [product_title#0, customer_id#1, review_date#2, product_category#3], [isnotnull(product_category#3), (product_category#
3 = Video)]
      +- InMemoryRelation [product_title#0, customer_id#1, review_date#2, product_category#3], StorageLevel(disk, memory, 1 replicas)
            +- *(1) FileScan parquet [product_title#0,customer_id#1,review_date#2,product_category#3] Batched: true, Format: Parquet, Locati
on: InMemoryFileIndex[s3://chie-us-east-2/tmp/filtered_amazon_customer_reviewed], PartitionCount: 39, PartitionFilters: [], PushedFilters:
[], ReadSchema: struct<product_title:string,customer_id:int,review_date:timestamp>
```

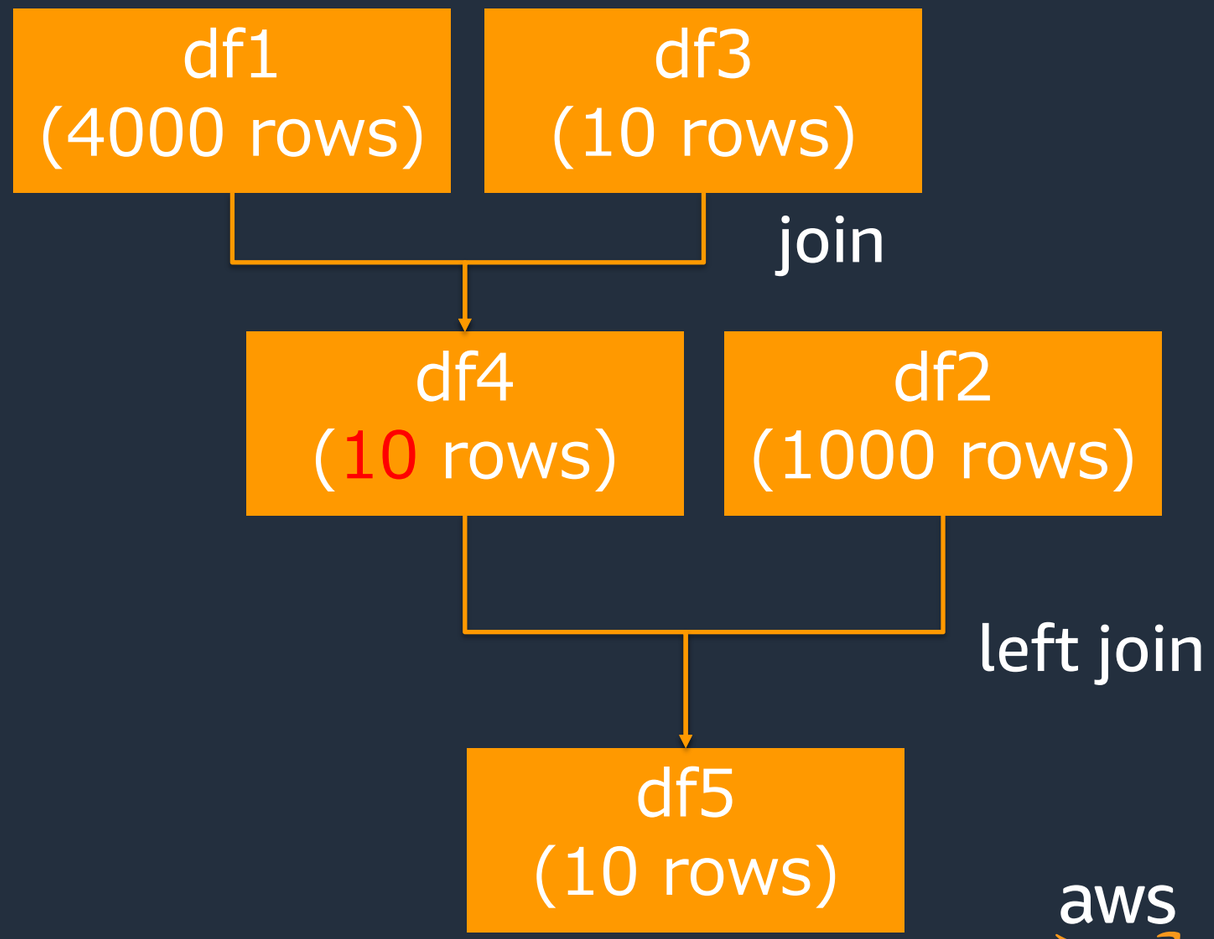
In [2]:
```
df1=spark.read.parquet(
    's3://chie-us-east-2/tmp/filtered_amazon_customer_reviewed')
df2 = df1.filter(df1.product_category == 'Video')
df2.persist()
df2.explain()
```

```
== Physical Plan ==
InMemoryTableScan [product_title#0, customer_id#1, review_date#2, product_category#3]
   +- InMemoryRelation [product_title#0, customer_id#1, review_date#2, product_category#3], StorageLevel(disk, memory, 1 replicas)
         +- *(1) FileScan parquet [product_title#0,customer_id#1,review_date#2,product_category#3] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[s3://chie-us-east-2/tmp/filtered_amazon_customer_reviewed], PartitionCount: 1, PartitionFilters: [isnotnull(product_catego
ry#3), (product_category#3 = Video)], PushedFilters: [], ReadSchema: struct<product_title:string,customer_id:int,review_date:timestamp>
```

joinの順番を工夫する

- 最終的な結果は同じだが途中のDataFrameのデータサイズが異なる
- Glue 3.0(Spark3.1.1)ではコストベースオプティマイザにより、データ量も考慮されてjoinの順番も最適化される





joinの使い分け

Sort Merge Join
- join対象の2つの表をそれぞれキーごとに分散させ、ソートした上でjoinする
- 大きなテーブル同士のジョインに適している

Broadcast Join
- 片方のテーブルを全てのExecutorに転送し、もう片方のテーブルを分散させていjoinする
- 片方のテーブルが小さい場合に適している

Shuffle Hash Join
- join対象の2つの表をソートせずに分散させてjoinする
- そこまで大きくないテーブル同士のjoinに適している

aws

joinの使い分け

- デフォルトでは、テーブルサイズがspark.sql.autoBroadcastJoinThresholdで指定した値（デフォルト10MB）以下の場合はBroadcast Joinが採用される
- 利用されているJoinストラテジはSpark UIやexplain()によって確認できる
- join性能がボトルネックになっている場合に、手動でjoinストラテジーを変更することでパフォーマンスがあがることがある

```
df1.join( broadcast(df2), df1(”col1") <=> df2(”col1") ).explain()
```

```
== Physical Plan == BroadcastHashJoin [coalesce(col1#6, )], [coalesce(col1#21, )], Inner, BuildRight, (col1#6 <=> col1#21)
 :- LocalTableScan [first_name#5, col1#6]
+- BroadcastExchangeHashedRelationBroadcastMode(List(coalesce(input[0, string, true], ))) +- LocalTableScan [col1#21, col2#22, population#23]
```

https://spark.apache.org/docs/2.4.0/sql-performance-tuning.html#broadcast-hint-for-sql-queries

aws

coalesce

以下の理由で、処理途中でパーティションが細かく分割されてしまう場合がある。

- 小さい大量のファイルの読み込みを行う
- カーディナリティの高い列でgroupByを行う

このような場合、次の処理を行う前にパーティションをマージして少なくしたほうが後続処理でオーバヘッドが少なくなる

repartitionはシャッフルを伴うので、coalesceを利用することが望ましい場合がある
ただし単純なマージのためcoalesce後のデータは偏りが発生する可能性がある

Glue 3.0ではAdaptive Query Executionという新機能により、自動的にcoalesceによってパーティション数を最適化してくれる

df.repartition(2)

df.coalesce(2)

aws

self joinとデータ集計の代わりにWindow処理を利用する

- 1つのログデータから集計データを作成してjoinする処理を行っている場合、Window処理を行うことでjoinの負荷をなくすことができる

```
df_agg = df.groupBy(‘gender’, ‘age’).agg(
    F.mean(‘height’).alias(‘avg_height’), F.mean(‘weight’).alias(‘avg_weight’))
df = df.join(df_agg, on=[‘gender‘, ‘age’])
```

```
w = Window.partitionBy(‘gender’, ‘age’)
df = df.withColumn(
    ‘avg_height’, F.mean(col(‘height’)).over(w)
).withColumn(‘avg_weight’, F.mean(col(‘weight’)).over(w))
```

タスク単位の処理を高速化する

aws

Scalaを利用する

DataFrameの殆どの処理は、PySparkで記述しても内部的にはJavaに変換してJVM上で動作するが、以下についてはPythonを利用すると処理が遅くなる。

上記がボトルネックになっている場合は、Scalaを利用することで処理が高速化する

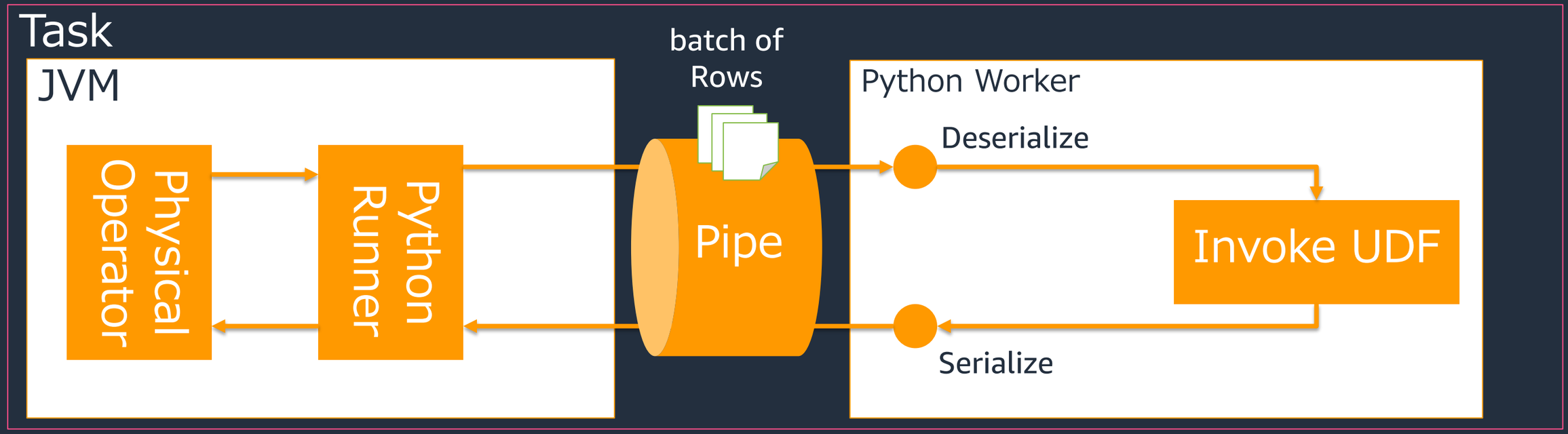
- RDDで処理を記述している部分
    - RDDで記述された処理はオプティマイザによる最適化が行われない
- UDFを利用している部分
    - 後述

aws

PySparkのUDFは避ける

パフォーマンスの問題点

- Iteratorごとにシリアライゼーションと Pythonプロセスへのパイプが発生
- Pythonプロセスのメモリは JVMの制御が行われない



aws

PythonUDFよりPandasUDFを利用する

Python UDF

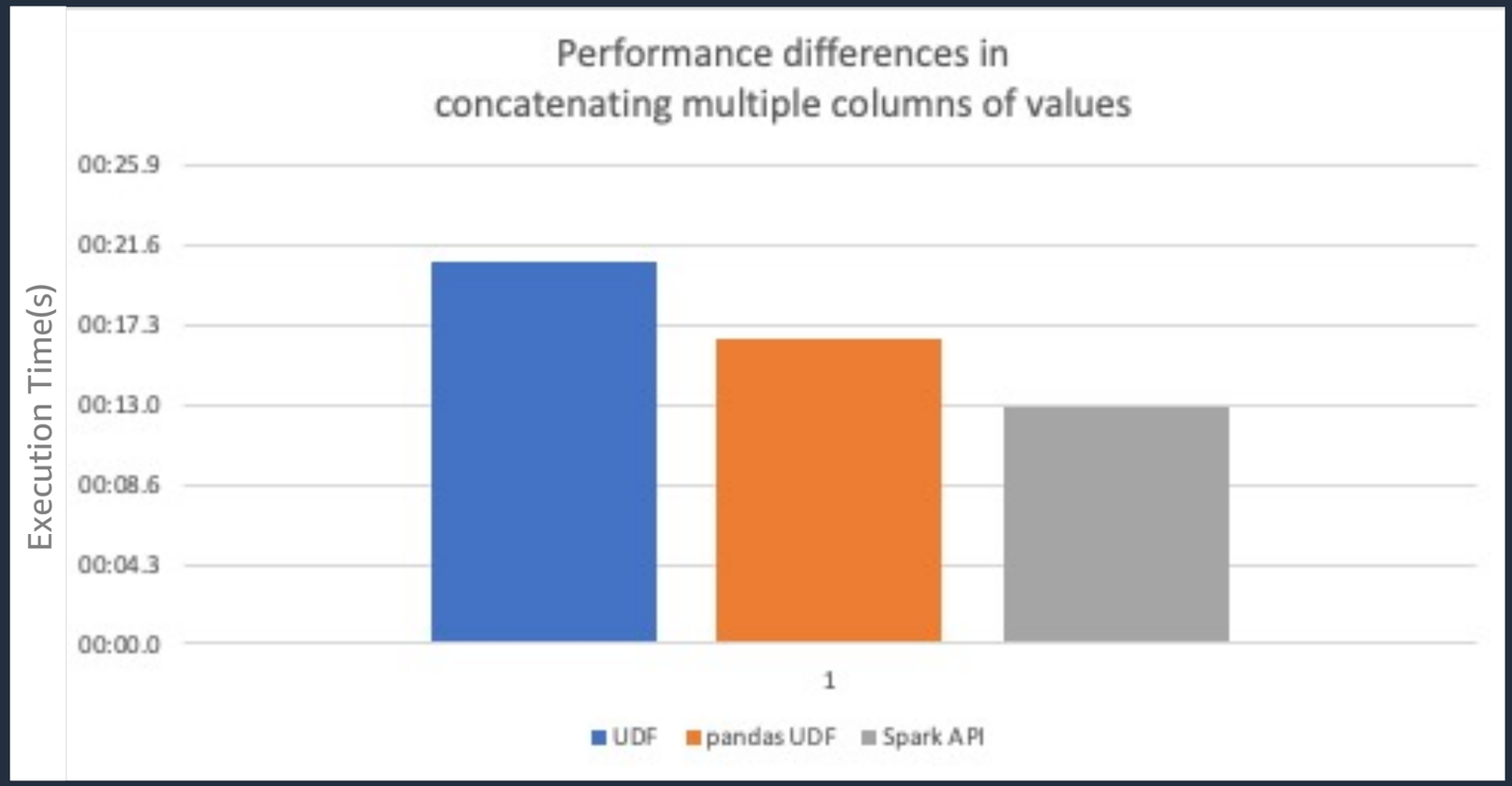
- シリアライズ/デシリアライズは**Pickling**で行われる
- データはブロックごとにfetchされるが、UDF処理は**行毎**に実行される

Pandas UDF

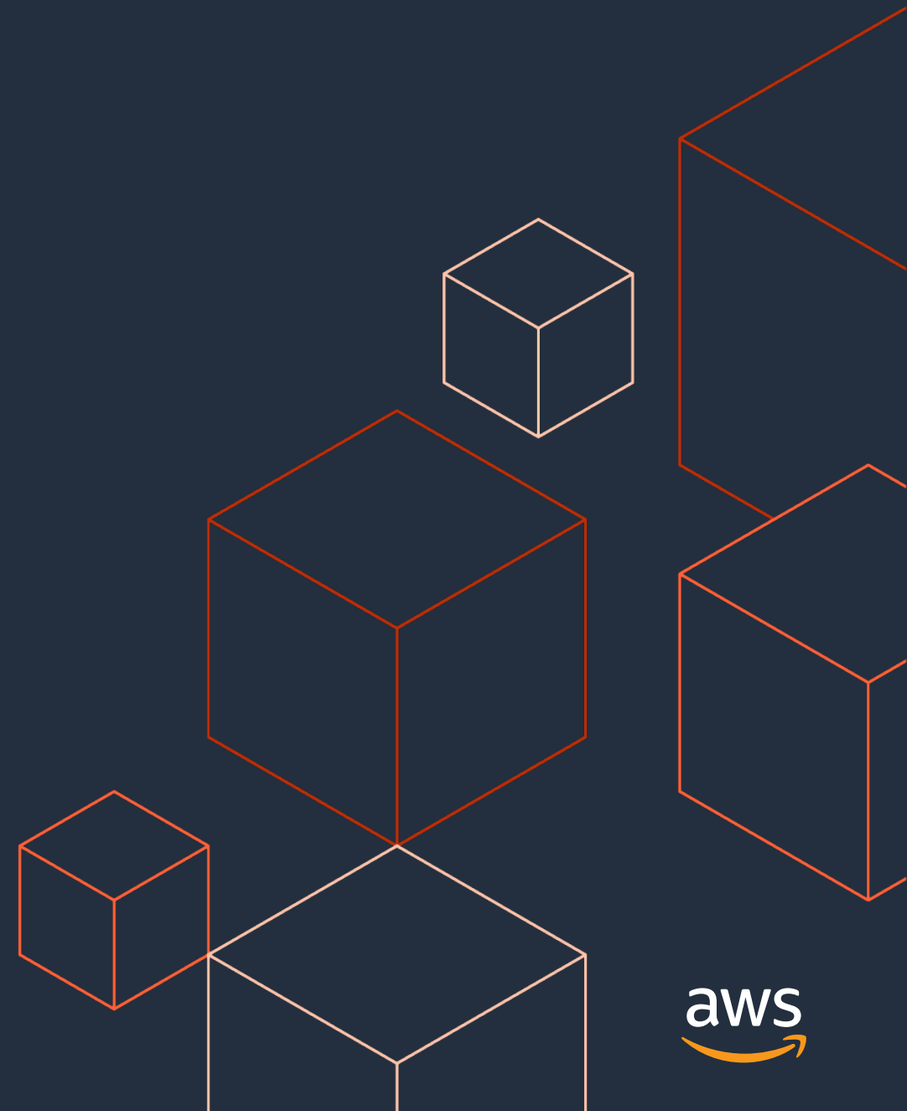
- シリアライズ/デシリアライズは**Apache Arrow**で行われる
- データのfetchもUDF処理も**複数行まとめて**に実行される

aws

AWS Glue ETLにおけるPython UDF、Pandas UDF、Spark APIのパフォーマンスの違いの例

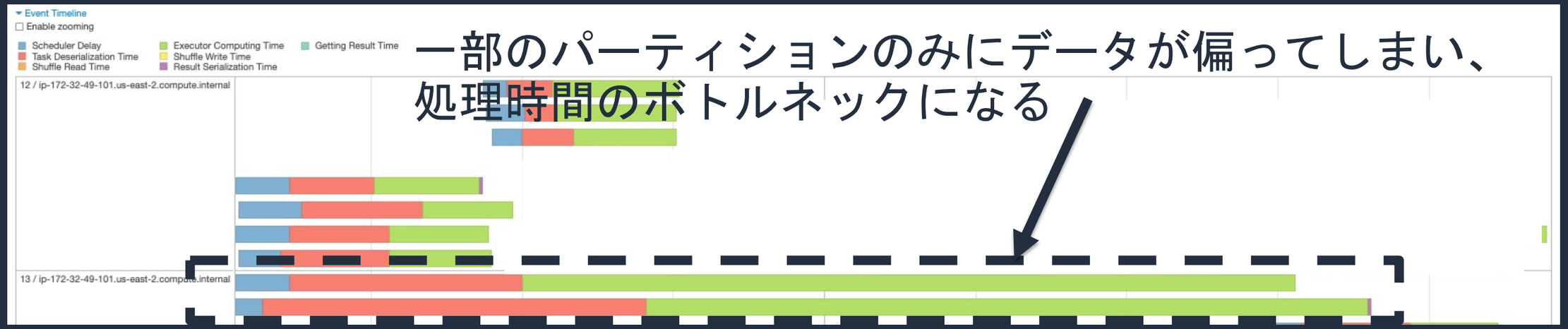


並列化する



データの偏り（Skewness）への対処

パーティションごとにデータのばらつきがあると、大きいパーティションの処理を行う一部のタスクのみに負荷が偏り、処理遅延の原因となる



どういうときに起こるか

- 読み取るファイルサイズに偏りがある場合
- joinキーごとにレコード数に差があるデータでjoinする場合
- df.groupBy()を行ったときのキー毎のレコード数のばらつき

aws

データの偏りへの対処

- インプットファイルを作成時にファイルサイズが均一になるようにする
- repartitionする
- broadcast join
- saltingする

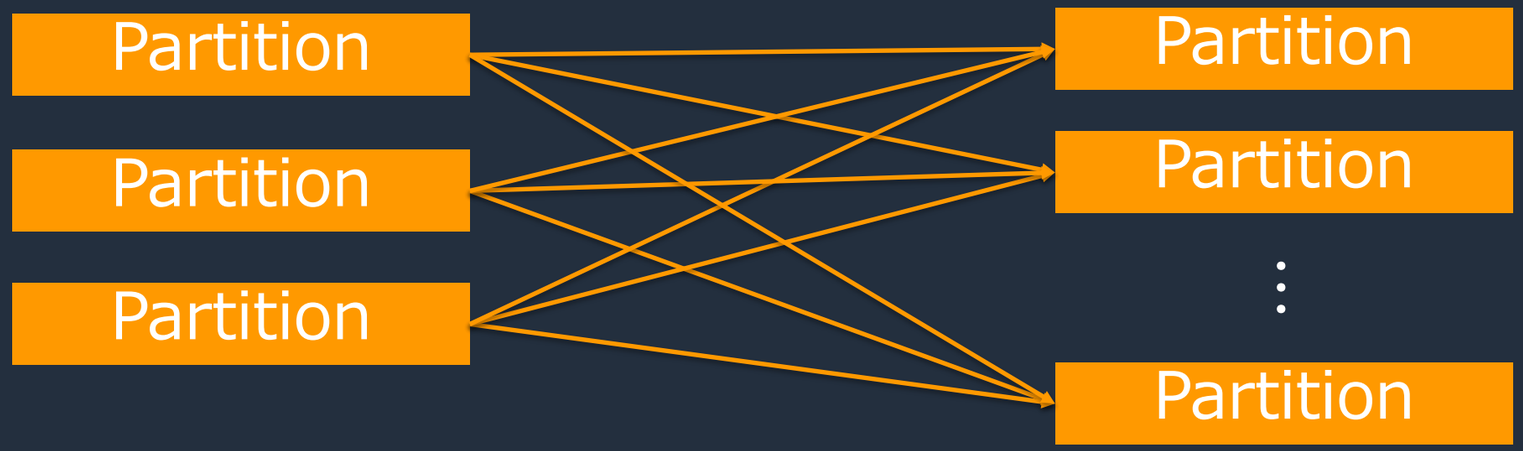
aws

Skewness（データの偏り）への対処

repartitionする

後続処理がキー毎の処理（日付ごとにパーティショニングしてデータ格納、キー毎のWindow処理など）でない場合は、repartitionによりSkewを解消する

df.repartition(200)

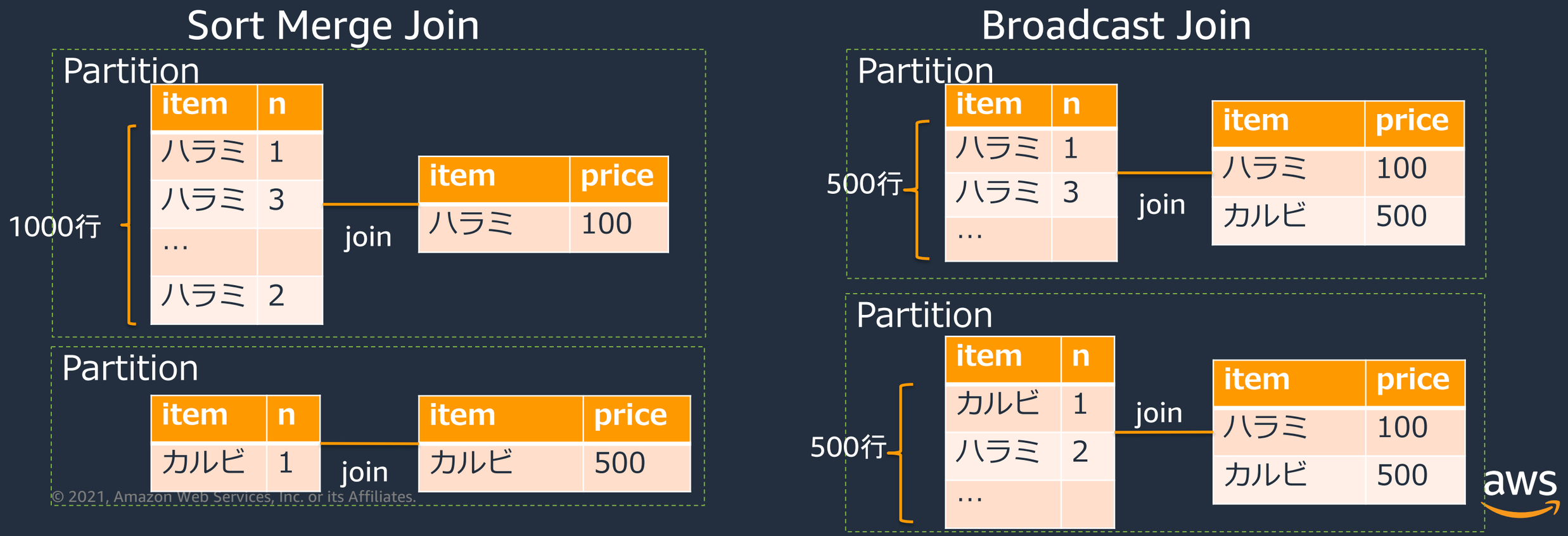


aws

Skewness（データの偏り）への対処

broadcast join

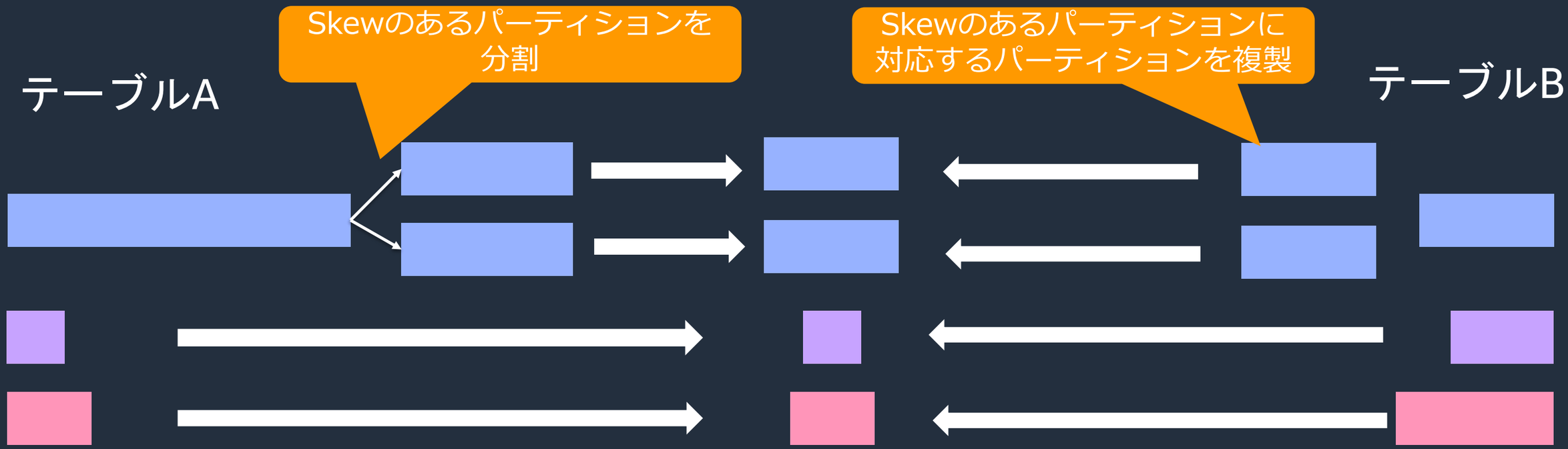
- 片方のDataFrameのサイズが1Executorに全てのるほど小さく、もう片方のDataFrameがジョインキーの列がSkewのある巨大なデータの場合、Joinのストラテジを Broadcast Join にすることで処理性能を向上することができる



Skewness（データの偏り）への対処

Saltingする

片方にSkewがある十分に大きいデータ同士のjoinの場合、「Salt」を行うことで負荷の偏りを解消することができる

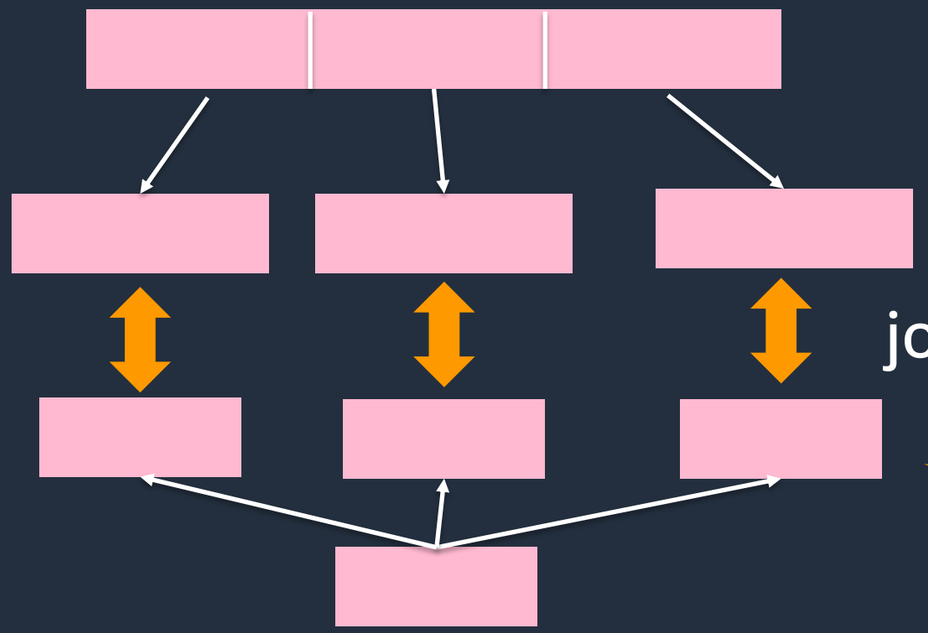


Skewness（データの偏り）への対処

Saltingする

片方にSkewがある十分に大きいデータ同士のjoinの場合、「Salt」を行うことで負荷の偏りを解消することができる

テーブルA



テーブルB

Skewのあるパーティションを分割

Skewのあるパーティションに対応するパーティションを複製

aws

Skew Joinの方法

- Glue 2.0（Spark 2.4.3）では、手動でSaltingのコードを記述する必要がある
- Glue 3.0（Spark 3.1.1）では、Adaptive Query Executionという新機能により、動的にSkew Joinを行われる

```
# Salting the skewed column
df_big = df_big.withColumn(‘shop_salt', F.concat(df[‘shop'], F.lit('_'), F.lit(F.floor(F.rand() * numPartition) +
1)))

# Explode the column
df_medium = df_medium.withColumn(‘shop_exploded', F.explode(F.array([F.lit(i) for i in
range(1,numPartition+1)])))
df_medium = df_medium.withColumn(
    ‘shop_exploded', F.concat(df_medium[‘shop'], F.lit('_’),
    df_medium[‘shop_exploded']))

# Joining
df_join = df_big.join(df_medium df_big.‘shop_salted’ == df_medium.shop_exploded)
```

https://spark.apache.org/docs/latest/sql-performance-tuning.html#optimizing-skew-join

パフォーマンスを考慮したインクリメンタルIDの付与

- Window関数 row_number()を利用して、全てのレコードに対する連続するインクリメンタルIDを付与する場合、全レコードに対するアグリゲーションが行われるため処理が低速になる。
- monotonically_increasing_id()はパーティションをまたがると非連続となることを許容することで、アグリゲーションを行わずにインクリメンタルIDを付与することができる。

df. withColumn(F.rowNumber().over(Window.partitionBy(”col1").orderBy(”col2"))

1	2	3	4	5	6	7	8	9	10	11

df. withColumn(monotonically_increasing_id)

1	2	3	6	7	8	9	10	13	14	15

aws

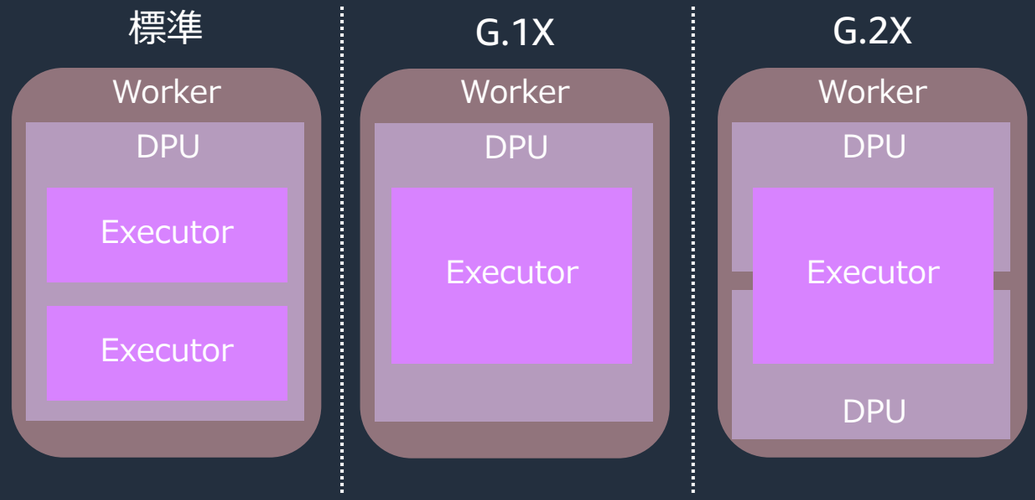
AWS GlueのWorker Typeの選定

- ジョブ実行時に割り当てる処理能力をDPU(Data Processing Unit)という
- 1DPU = 4vCPU、16GBメモリ
- Worker Typeごとにリソース容量や構成が異なる

Worker Type一覧

Worker Type	DPU数 /1Worker	Executor数 /1Worker	メモリ数 /1Worker	ディスク /1Worker
標準	1	2	16GB	50GB
G.1X	1	1	16GB	64GB
G.2X	2	1	32GB	128GB

Worker Type構成イメージ



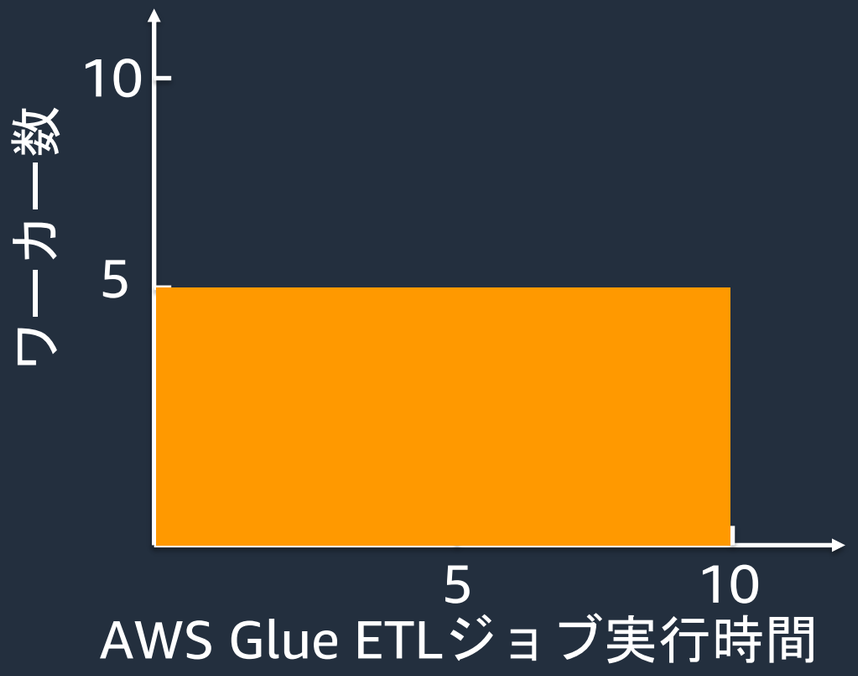
理想的なリソース使用状況

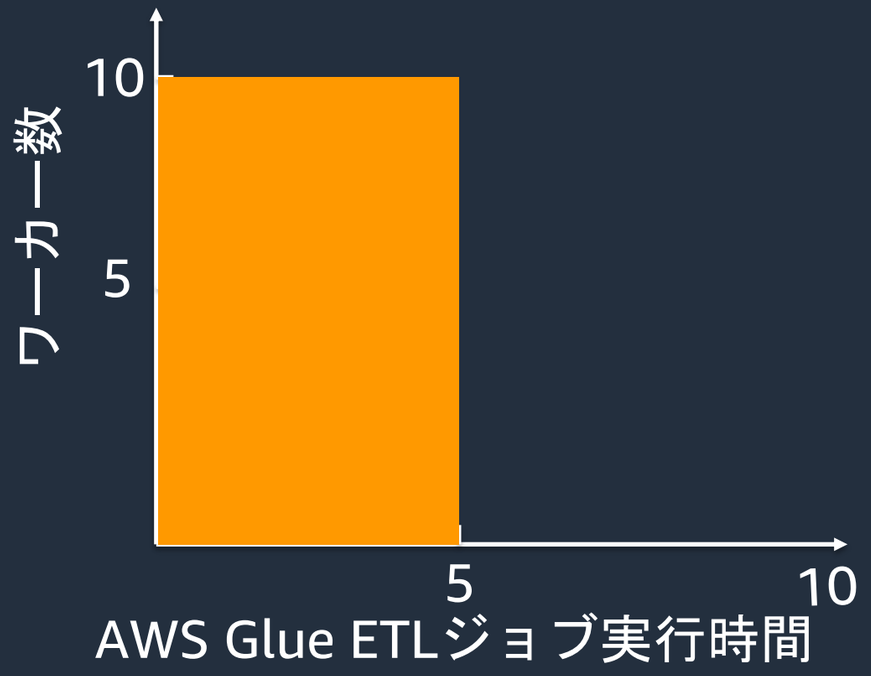
- すべてのExecutorでリソースが無駄なく均等に利用されている状態が望ましい
- そうでない場合はチューニングの余地がある可能性が高い
- リソース傾向をある程度処理内容から予測して最初のWorker Typeを選ぶ
    - 例：
        - 複雑なUDFなどの加工処理がある場合はCPU使用率が高くなりやすい
        - 大量データのjoinなど、シャッフルサイズが大きくなる場合はメモリ使用率が高くなりやすい

aws

ワーカー数とジョブ実行時間のトレードオフ

- リソースが有効利用できるだけの並列数が担保できる限りにおいて、ワーカー数を増やすことでコストを増やすことなくジョブ実行時間が短縮できる





まとめ

- AWS Glue ETL ジョブのチューニングパターンを紹介した
- AWS Glueはそのままでも大量データを高いパフォーマンスで処理できるが、チューニングを行うことでさらに高いパフォーマンスとスケーラビリティを実現できる
- チューニングを行うにはメトリクスを確認しながらボトルネックを特定し、その原因をとりのぞく必要がある

本資料に関するお問い合わせ・ご感想

- 技術的な内容に関しましては、有料のAWSサポート窓口へお問い合わせください
  - https://aws.amazon.com/jp/premiumsupport/
- 料金面でのお問い合わせに関しましては、カスタマーサポート窓口へお問い合わせください（マネジメントコンソールへのログインが必要です）
  - https://console.aws.amazon.com/support/home#/case/create?issueType=customer-service
- 具体的な案件に対する構成相談は、後述する個別技術相談会をご活用ください

ご感想はTwitterへ！ハッシュタグは以下をご利用ください
#awsblackbelt

AWSの日本語資料の場所「AWS 資料」で検索

https://amzn.to/JPArchive

AWSのハンズオン資料の場所「AWS ハンズオン」で検索

https://aws.amazon.com/jp/aws-jp-introduction/aws-jp-webinar-hands-on/

AWS Well-Architected個別技術相談会

- 毎週「W-A個別技術相談会」を実施中
  - AWSのソリューションアーキテクト（SA）に対策などを相談することも可能
- 申込みはイベント告知サイトから
  - https://aws.amazon.com/jp/about-aws/events/

ご視聴ありがとうございました