Databricks.Associate-Developer-Apache-Spark-3.5.v2026-05-26.q73

試験コード:	Associate-Developer-Apache-Spark-3.5
試験名称:	Databricks Certified Associate Developer for Apache Spark 3.5 - Python
認定資格:	Databricks
無料問題数:	73
バージョン:	v2026-05-26
アクセス数:	107
ページビュー数:	730
https://www.jpnpdf.com/Databricks.Associate-Developer-Apache-Spark-3.5.v2026-05-26.q73-mondaishu.html	

最新問題: **1**
エンジニアは2つのDataFrame（df1（小）とdf2（大））を持っています。ブロードキャスト結合を使用します。
パイソン
コピー編集
pyspark.sql.functionsからブロードキャストをインポートする
結果 = df2.join(broadcast(df1), on='id', how='inner')
このシナリオでbroadcast() を使用する目的は何ですか?
オプション:
A. 結合を実行する前に ID 値をフィルタリングします。
B. df1 と df2 のパーティション サイズを増加させます。
C. 小さい DataFrame をすべてのノードに複製することで、シャッフル操作の数を減らします。
D. ID 値が同一の場合にのみ結合が行われるようにします。
Answer: C (メーッセージを残す**)**
broadcast(df1) は、小さな DataFrame (df1) をすべてのワーカー ノードに送信するように Spark に指示します。
これにより、結合中に df1 をシャッフルする必要がなくなります。
ブロードキャスト結合は、1 つの大きなテーブルと 1 つの小さなテーブルがあるシナリオに最適化されています。

最新問題: **2**
あるデータエンジニアは、毎日最新のデータで上書きされるParquetテーブルの作成を依頼されました。このParquetテーブルの下流の利用者には、このテーブル内のすべてのレコードがmarket_timeフィールドでソートされた状態でデータが生成されることが必須要件となっています。
これらの要件を満たす Parquet テーブルを生成するのは、Spark コードのどの行ですか?
A. 最終df \
.sort("market_time") \
。書く \
.format("寄木細工") \
.mode("上書き") \

.saveAsTable("output.market_events")
B. ファイナル_df \
.orderBy("market_time") \
。書く \
.format("寄木細工") \
.mode("上書き") \
.saveAsTable("output.market_events")
C. ファイナル_df \
.sort("market_time") \
.coalesce(1) \
。書く \
.format("寄木細工") \
.mode("上書き") \
.saveAsTable("output.market_events")
D. 最終_df \
.sortWithinPartitions("market_time") \
。書く \
.format("寄木細工") \
.mode("上書き") \
.saveAsTable("output.market_events")

Answer: D (メーッセージを残す)
ディスクに書き出されるデータがソートされていることを確認するには、SparkがParquetテーブルに保存する際にデータを書き込む方法を考慮することが重要です。.sort() または .orderBy() メソッドはグローバルソートを適用しますが、特定の条件（例えば、スケーラブルではない .coalesce(1) による単一パーティションなど）が満たされない限り、最終出力ファイルでソートが維持されることは保証されません。
代わりに、分散 Spark 処理で行が書き出されるときにそれぞれのパーティション内でソートされるようにするための適切な方法は次のとおりです。
.sortWithinPartitions("列名")
Apache Spark のドキュメントによると:
「sortWithinPartitions() は、各パーティションが指定された列でソートされることを保証します。これは、ソートされたファイルを必要とする下流のシステムに役立ちます。」このメソッドは分散設定で効率的に動作し、グローバルソート（orderBy() や .sort() など）のパフォーマンスボトルネックを回避し、各出力パーティションにソートされたレコードが含まれることを保証します。これにより、一貫してソートされたデータという要件が満たされます。
したがって：
オプション A と B では、保存されたファイルの内容がソートされることは保証されません。
オプション C は、.coalesce(1) (単一パーティション) によってボトルネックが発生します。
オプション D はパーティション内でソートを正しく適用し、スケーラブルです。

最新問題: **3**
あるeコマース企業のデータサイエンティストは、サブスクライバーデータベースから取得したユーザーデータを処理しており、そのデータをDataFrame df_userに格納しています。このデータをさらに処理する前に、データサイエンティストは別のDataFrame df_user_non_piiを作成し、このDataFrameにPII以外の列のみを格納したいと考えています。df_userのPII列は、first_name、last_name、email、birthdateです。
この要件を満たすために使用できるコード スニペットはどれですか?

A. df_user_non_pii = df_user.drop("名", "姓", "メールアドレス", "生年月日")
B. df_user_non_pii = df_user.drop("名", "姓", "メールアドレス", "生年月日")
C. df_user_non_pii = df_user.dropfields("first_name", "last_name", "email", "birthdate")
D. df_user_non_pii = df_user.dropfields("名、姓、メールアドレス、生年月日")
Answer: (解答を表示する**)**
PySpark DataFrameから特定の列を削除するには、drop()メソッドを使用します。このメソッドは、指定された列を含まない新しいDataFrameを返します。複数の列を削除する正しい構文は、各列名をdrop()メソッドに個別の引数として渡すことです。
正しい使い方:
df_user_non_pii = df_user.drop("first_name", "last_name", "email", "birthdate") このコード行は、指定された PII 列を除外した新しい DataFrame df_user_non_pii を返します。
オプションの説明:
A . 正解です。drop() メソッドを使用し、複数の列名を個別の引数として渡します。これは、PySpark における標準的で正しい使用法です。
B 。オプションAと似ているように見えますが、列名が引用符で囲まれていない場合や構文エラー（引用符の欠落、変数名の誤りなど）がある場合はエラーになります。ただし、記述通りであればオプションAと同一であり、正しいです。
C . 不正解です。dropfields() メソッドは、PySpark の DataFrame クラスのメソッドではありません。このメソッドは、StructType 列で使用され、ネストされた構造からフィールドを削除するものであり、最上位の DataFrame 列ではありません。
D . 不正解です。コンマで区切られた列名を含む単一の文字列を dropfields() に渡すことは、PySpark では有効な構文ではありません。
参照：
PySpark ドキュメント: DataFrame.drop
Stack Overflow ディスカッション: PySpark DataFrame の列を削除する方法

最新問題: 4
データ エンジニアは、feed と呼ばれる Kafka トピックから読み取るストリーミング データフレームを作成したいと考えています。

```
1. spark
2. .readStream
3. .format("kafka")
4. .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
5. .____________________
6. .load()
```

要件を満たすには、5 行目にどのコード フラグメントを挿入する必要がありますか?
コードコンテキスト:
スパーク\
.readStream \
.format("カフカ") \
.option("kafka.bootstrap.servers", "ホスト1:ポート1,ホスト2:ポート2") \
.[5行目] \
。負荷（）

オプション:
A. .option("subscribe", "feed")
B. .option("subscribe.topic", "feed")
C. .option("kafka.topic", "フィード")
D. .option("トピック", "フィード")
Answer: A (メーッセージを残す)
構造化ストリーミングを使用して特定の Kafka トピックから読み取る場合の正しい構文は次のとおりです。
パイソン
コピー編集
.option("購読", "フィード")
これは Spark ドキュメントで明示的に定義されています。
「subscribe - サブスクライブするKafkaトピック。このオプションには1つのトピックのみ指定できます。」（出典Apache Spark Structured Streaming + Kafka統合ガイド）
「subscribe - サブスクライブする Kafka トピック。このオプションには 1 つのトピックのみ指定できます。」(出典: Apache Spark Structured Streaming + Kafka 統合ガイド) B. 「subscribe.topic」は無効です。
C . 「kafka.topic」は認識されるオプションではありません。
D . 「topic」は、Spark の Kafka ソースでは無効です。

最新問題**: 5**
データ エンジニアは、列国別にパーティション化された DataFramedf を Parquet ファイルに書き込み、宛先パスにある既存のデータを上書きする必要があります。
データ エンジニアは、Apache Spark でこのタスクを実行するためにどのコードを使用すればよいですか。
A. df.write.mode("overwrite").partitionBy("country").parquet("/data/output")
B. df.write.mode("append").partitionBy("country").parquet("/data/output")
C. df.write.mode("overwrite").parquet("/data/output")
D. df.write.partitionBy("country").parquet("/data/output")
Answer: (解答を表示する)
.mode("overwrite") は、パスにある既存のファイルが置き換えられることを保証します。
partitionBy("country") は、パーティション化されたフォルダーにデータを書き込むことでクエリを最適化します。
正しい構文:
df.write.mode("overwrite").partitionBy("country").parquet("/data/output")
- 出典:Spark SQL、データフレーム、データセットガイド

最新問題**: 6**
開発者は、次のコードを使用して、sales.purchases_fct と sales.customer_dim の 2 つのテーブルを結合しようとしています。

```
import pyspark.sql.functions as F

purch_df = spark.table('sales.purchases_fct')
cust_df = spark.table('sales.customer_dim').dropDuplicates(['cust_id'])

fact_df = purch_df.join(cust_df, F.col('customer_id') == F.col('cust_id'))
```


fact_df = purch_df.join(cust_df, F.col('customer_id') == F.col('custid')) 開発者は、customer_dim テーブルに存在しない purchases_fct テーブル内の顧客が結合テーブルから削除されていることを発見しました。
これらの顧客レコードが削除されないようにするには、コードにどのような変更を加える必要がありますか?
A. fact_df = purch_df.join(cust_df, F.col('customer_id') == F.col('custid'), 'left')
B. fact_df = cust_df.join(purch_df, F.col('customer_id') == F.col('custid'))
C. fact_df = purch_df.join(cust_df, F.col('cust_id') == F.col('customer_id'))
D. fact_df = purch_df.join(cust_df, F.col('customer_id') == F.col('custid'), 'right_outer')
Answer: A (メーッセージを残す**)**
Sparkでは、デフォルトの結合タイプは内部結合で、両方のDataFrameで一致するキーを持つ行のみを返します。左側のDataFrame（purch_df）のすべてのレコードを保持し、右側のDataFrame（cust_df）の一致するレコードを含めるには、左外部結合を使用する必要があります。
結合タイプを「左」に指定することで、修正されたコードではpurch_dfのすべてのレコードが保持され、cust_dfの一致するレコードも含まれるようになります。purch_dfのレコードのうち、cust_dfに対応するレコードがない場合、cust_dfの列にはnull値が設定されます。
このアプローチは標準の SQL 結合操作と一致しており、PySpark の DataFrame API でサポートされています。

最新問題**: 7**
55 件中 9 件目。
次のコードフラグメントがあるとします:
pyspark.pandasをpsとしてインポートする
pdf = ps.DataFrame(データ)
Spark DataFrame (pyspark.pandas.DataFrame) 上の Pandas API を標準の PySpark DataFrame (pyspark.sql.DataFrame) に変換するにはどの方法を使用しますか?
A. pdf.to_pandas()
B. pdf.to_spark()
C. pdf.to_dataframe()
D. pdf.spark()
Answer: (解答を表示する**)**
Spark (以前の Koalas) 上の Pandas API では、メソッド .to_spark() が pyspark.pandas.DataFrame を PySpark DataFrame に変換します。
正しい使い方:
spark_df = pdf.to_spark()
これにより、Spark 上の Pandas API と PySpark SQL API 間の相互運用性が実現し、開発者は変換やパフォーマンスの最適化のために両者をシームレスに切り替えることができます。
他のオプションが間違っている理由:
A (to_pandas): PySpark DataFrame ではなく、ローカルの Pandas DataFrame に変換します。

C (to_dataframe): 有効な API メソッドではありません。
D (spark): 既存の DataFrame メソッドではありません。
参照：
PySpark Pandas API リファレンス - DataFrame.to_spark() メソッド。
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark での Pandas API の使用」- DataFrame の変換と相互運用性について説明します。

最新問題**: 8**
55 件中 36 件目。
テーブルを永続化するときにデータをパーティション分割する主な利点は何ですか?
A. ディスク容量を節約するためにデータを圧縮します。
B. 未使用のパーティションを自動的にクリーンアップして、ストレージを最適化します。
C. クエリの実行を高速化するために、データが一度にメモリにロードされることを保証します。
D. より少ないパーティションから関連するデータのサブセットのみを読み取ることで最適化します。
Answer: (解答を表示する**)**
データセットのパーティション分割では、パーティション列の値に基づいてデータが個別のディレクトリに分割されます。クエリがパーティション列でフィルタリングされると、Spark は不要なパーティションを除外します。つまり、フィルタ条件に一致するファイルのみを読み取ります。
アドバンテージ：
関連するデータのサブセットのみをスキャンすることで I/O を削減し、パフォーマンスを向上させます。
例：
/データ/売上/年=2023/月=10/...
/データ/売上/年=2024/月=01/...
WHERE year = 2024 をフィルタリングするクエリは、関連するパーティションのみを読み取ります。
他のオプションが間違っている理由:
A: 圧縮はパーティション分割とは無関係です。
B: 手動で管理されない限り、Spark はパーティションを自動的にクリーンアップしません。
C: パーティショニングによって Spark がデータ全体をメモリにロードするわけではありません。
参照：
Databricks 試験ガイド (2025 年 6 月) : セクション 「Spark SQL の使用」- 最適化されたデータ取得のためのパーティション分割とプルーニング。
Spark SQL ドキュメント - DataFrameWriter の partitionBy() とクエリの最適化。

最新問題**: 9**
データエンジニアは、sensor_id、temperature、timestampの列を持つセンサーの読み取りデータを毎秒受信するストリーミングデータフレームを処理したいと考えています。エンジニアは、データのストリーミング中に、過去5分間の各センサーの平均温度を計算する必要があります。
どのコード実装が要件を満たしていますか?
提供された画像からのオプション:
A.
```
df.withColumn("avg_temp", avg("temperature")
  .over(Window.partitionBy("sensor_id")
```

B.
```
df.groupBy("sensor_id", "timestamp")
  .agg(avg("temperature").alias("avg_temp"))
```
C.
```
df.groupBy("sensor_id").avg("temperature")
df.withWatermark("timestamp", "5 minutes")
```
D.
```
.groupBy("sensor_id", window("timestamp", "5 minutes"))
.agg(avg("temperature").alias("avg_temp"))
```
Answer: (解答を表示する**)**
正解は D です。これは、適切な時間ベースのウィンドウ集計とウォーターマークを使用しているためです。これは、イベント時間データの時間ベースの集計に Spark Structured Streaming で必要なパターンです。
構造化ストリーミングに関する Spark 3.5 ドキュメントより:
イベントタイム列にスライディングウィンドウを定義し、groupByとwindow()を使用してそれらのウィンドウの集計を計算できます。遅延データを処理するには、withWatermark()を使用して、遅延データの到着許容範囲を指定します。(出典: Structured Streaming Programming Guide) オプションDでは、以下の使用法が採用されています。
パイソン
コピー編集
.groupBy("sensor_id", window("timestamp", "5分"))
.agg(avg("温度").alias("avg_temp"))
各sensor_idについて、5分間のイベント時間ウィンドウにわたって平均温度が計算されることを保証します。ロジックを完成させるために、パイプラインの早い段階でwithWatermark("timestamp", "5 minutes")を使用して、遅延イベントを処理することを前提としています。
他のオプションが間違っている理由の説明:
オプション A は、静的 DataFrame またはバッチ クエリに適用される Window.partitionBy を使用しますが、ストリーミング集計には適していません。
オプション B では時間ウィンドウが適用されないため、5 分間の移動平均は計算されません。
オプション C では、集計後に withWatermark() が誤って適用され、時間ウィンドウが含まれていないため、必要な時間ベースのグループ化が欠落しています。
したがって、オプション D は、時間ウィンドウ ストリーミング集約を計算するためのすべての要件を満たす唯一のオプションです。

最新問題**: 10**
あるデータサイエンティストが、PySparkを使用してApache Sparkの大規模データセットに取り組んでいます。データサイエンティストは、user_id、product_id、purchase_amountの列を持つDataFrame dfを所有しており、このデータに対していくつかの操作を効率的に実行する必要があります。
シャッフルを必要とする変換と、それに続くシャッフルを必要としない変換が生じる操作シーケンスはどれですか?
A. df.filter(df.purchase_amount > 100).groupBy("user_id").sum("purchase_amount")
B. df.withColumn("割引", df.purchase_amount * 0.1).select("割引")
C. df.withColumn("purchase_date", current_date()).where("total_purchase > 50")
D. df.groupBy("user_id").agg(sum("purchase_amount").alias("total_purchase")).repartition(10)
Answer: (解答を表示する**)**

シャッフルは、groupBy、reduceByKey、joinなどの操作で発生し、これらの操作ではデータがパーティション間で移動されます。repartition()操作でもシャッフルが発生する可能性がありますが、このコンテキストでは集計の後に発生します。
オプション D では、groupBy の後に agg を実行すると、ノード間のグループ化によりシャッフルが発生します。
再パーティション(10)はパーティション変換ですが、データがすでにグループ化されているため、新たなシャッフルは行われません。
このシーケンス (シャッフル (groupBy)、その後に非シャッフル (repartition)) は正しいです。
オプション A はその逆を行います。フィルターはシャッフルを引き起こしませんが、groupBy はシャッフルを引き起こします。これにより順序が間違ってしまいます。

最新問題**: 11**
データエンジニアは、Spark 3.0 から Spark 3.5 にアップグレードした後、パフォーマンスが向上したことに気づきました。エンジニアは、Adaptive Query Execution (AQE) が有効になっていることを発見しました。
AQE はパフォーマンスを向上させるためにどの操作を実装していますか?
A. 結合戦略を動的に切り替える
B. 永続的なテーブル統計を収集し、将来使用するためにメタストアに保存します。
C. シングルステージSparkジョブのパフォーマンスの向上
D. ディスク上の Delta ファイルのレイアウトを最適化します
Answer: A (メーッセージを残す**)**
アダプティブクエリ実行（AQE）は、実行時にクエリプランを動的に最適化するSpark 3.xの機能です。その中核機能の一つは次のとおりです。
実行時統計に基づいて結合戦略を動的に切り替えます (例: ソートマージからブロードキャストへ)。
その他の AQE 機能には次のものがあります:
シャッフルパーティションの結合
スキュー結合処理
オプションAが正解です。
オプション B は統計収集を指しますが、これは AQE の主な機能ではありません。
オプション C は範囲が広すぎて、AQE に特化していません。
オプション D は、AQE とは関係のない Delta Lake の最適化を指します。
最終答え :A

最新問題**: 12**
55件中47件目。
データ エンジニアは、2 つの DataFrame df1 と df2 を結合する次のコードを作成しました。
df1 = spark.read.csv("sales_data.csv")
df2 = spark.read.csv("product_data.csv")
df_joined = df1.join(df2, df1.product_id == df2.product_id)
DataFrame df1 には約 10 GB の販売データが含まれ、df2 には約 8 MB の製品データが含まれます。
Spark はどの結合戦略を使用しますか?
A. df1 と df2 のサイズ差が大きすぎるため、ブロードキャスト結合を効率的に実行できないため、シャッフル結合を実行します。
B. AQE が有効になっていないため、Spark は静的クエリ プランを使用するため、シャッフル結合を行います。
C. ブロードキャストヒントが提供されなかったため、シャッフル結合します。
D. df2 がデフォルトのブロードキャストしきい値より小さいため、ブロードキャスト参加します。

Answer: (解答を表示する**)**
結合の片側がブロードキャストしきい値内に収まるほど小さい場合、Spark はブロードキャスト ハッシュ結合を自動的に使用します。
デフォルトのしきい値:
spark.sql.autoBroadcastJoinThreshold = 10MB (Spark 3.5 以降)
df2 は 8 MB なので、Spark はそれをすべてのエグゼキューターに自動的にブロードキャストします。これにより、大きなデータセット (df1) のシャッフルが回避され、結合が高速化されます。
他のオプションが間違っている理由:
A: 8 MB < 10 MB しきい値 → ブロードキャスト参加が効率的です。
B: ブロードキャスト結合には AQE は必要ありません。これは静的な最適化です。
C: ブロードキャストヒントはオプションです - Spark は自動的に推測します。
参照：
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark DataFrame API アプリケーションのトラブルシューティングとチューニング」 - ブロードキャスト結合と最適化。
Spark SQL 結合戦略 - ハッシュ結合とシャッフル結合のしきい値をブロードキャストします。

最新問題**: 13**
開発者がSpark SQLクエリを実行したところ、リソースが十分に活用されていないことに気づきました。エグゼキューターはアイドル状態であり、ステージあたりのタスク数も少なくなっています。
クラスターの使用率を向上させるために開発者は何をすべきでしょうか?
A. spark.sql.shuffle.partitionsの値を増やす
B. spark.sql.shuffle.partitionsの値を減らす
C. データセットのサイズを増やしてパーティションを増やす
D. 必要に応じてリソースを拡張するために動的なリソース割り当てを有効にします
Answer: A (メーッセージを残す**)**
タスク数はパーティション数によって制御されます。デフォルトでは、spark.sql.shuffle.partitions は200です。ステージに表示されるタスク数が非常に少ない場合（コア数の合計より少ない場合）、完全な並列処理が活用されていない可能性があります。
Spark チューニング ガイドより:
「特に大規模なクラスターのパフォーマンスを向上させるには、spark.sql.shuffle.partitions を増やして、より多くのタスクと並列処理を作成します。」つまり、次のようになります。
A は正解です。シャッフルパーティションを増やすと並列性が向上します。
Bは間違いです。並列性がさらに低下します。
C は無効です。データセットのサイズを大きくしても、パーティションが増えるとは限りません。D はステージごとのタスク数とは無関係です。最終答え: A

最新問題**: 14**
55 件中 12 件目。
あるデータサイエンティストが、モデルの機能構築のため、ユーザープロファイルデータを調査していました。探索的データ分析の結果、ユーザープロファイルの一部のレコードには、NULL値が多すぎるフィールドが含まれているため、役に立たないことが判明しました。
ユーザー プロファイル テーブルのスキーマは次のようになります。
ユーザーID 文字列、

ユーザー名文字列、
生年月日 DATE、
国 STRING、
created_at タイムスタンプ
データ サイエンティストは、いずれかのレコードのいずれかのフィールドに NULL 値が含まれている場合、さらに処理を進める前にそのレコードを出力から削除することに決めました。
これらの要件を満たすために使用できる Spark コード ブロックはどれですか?
A. フィルターされたユーザー = raw_users.na.drop("any")
B. フィルターされたユーザー = raw_users.na.drop("すべて")
C. フィルターされたユーザー = raw_users.dropna(how="any")
D. フィルターされたユーザー = raw_users.dropna(how="all")
Answer: (解答を表示する**)**
Spark の DataFrame API では、dropna() (または同等の DataFrameNaFunctions.drop()) メソッドは null 値を含む行を削除します。
行動：
how="any" → いずれかの列に null 値がある行を削除します。
how="all" → すべての列が null である行を削除します。
データ サイエンティストは null フィールドを持つレコードを削除したいため、正しいパラメーターは how="any" です。
正しい構文:
フィルターされたユーザー = raw_users.dropna(how="any")
これにより、任意の列に少なくとも 1 つの null 値を持つすべてのレコードが削除されます。
他のオプションが間違っている理由:
A: na.drop("any") を使用しますが、括弧のコンテキストがありません (raw_users.na.drop("any") としてのみ機能し、オプション C と同等です)。
B/D: how="all" は、すべての値が null である行のみを削除します。このユースケースには厳しすぎます。
参照：
PySpark DataFrame API - DataFrameNaFunctions.drop() および DataFrame.dropna()。
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark DataFrame/DataSet API アプリケーションの開発」 - 欠落データの処理と DataFrame のクリーニング操作について説明します。

最新問題**: 15**
55 件中 6 件目。
Apache Spark のアーキテクチャのどのコンポーネントが、割り当てられたときにタスクを実行する責任がありますか?
A. ドライバーノード
B. 執行者
C. CPUコア
D. ワーカーノード
Answer: (解答を表示する**)**
Spark の分散アーキテクチャでは、次のようになります。
ドライバーノードはSparkアプリケーションの実行を調整し、論理プランをステージとタスクの物理プランに変換します。
ワーカー ノード上で実行される Executor は、ドライバーによって割り当てられたタスクを実行し、実行中にデータを (メモリまたはディスクに) 保存する役割を担います。

要点:
エグゼキューターは、データパーティション上で実際の計算を実行するアクティブエージェントです。各エグゼキューターは、利用可能なCPUコアを使用して複数のタスクを並列に実行します。
他のオプションが間違っている理由:
A (ドライバー ノード): ドライバーはタスクをスケジュールしますが、実行はしません。
C (CPU コア): CPU コアはエグゼキュータ内で実行されますが、Spark アーキテクチャ コンポーネントではなくハードウェアです。
D (ワーカー ノード): ワーカー ノードはエグゼキュータをホストしますが、タスクを直接実行しません。エグゼキュータが実行します。
参考資料（Databricks Apache Spark 3.5 - Python / 学習ガイド）：
Spark アーキテクチャ コンポーネント - ドライバー、エグゼキューター、クラスター マネージャー、ワーカー ノード。
Databricks 試験ガイド (2025 年 6 月) : セクション「Apache Spark のアーキテクチャとコンポーネント」- 分散処理におけるドライバー ノードとエグゼキュータ ノードの役割について説明します。

最新問題**: 16**
データ エンジニアは、指定されたストリーミング データを使用して、ストリーミング データフレーム streaming_df を操作しています。

Id	Name	count	timestamp
1	Delhi	20	2024-09-19T10:10:10.000+00:00
1	Delhi	50	2024-09-19T10:10:50.000+00:00
1	Delhi	10	2024-09-19T10:11:10.000+00:00
2	London	50	2024-09-19T10:10:20.000+00:00
3	Paris	30	2024-09-19T10:10:30.000+00:00
3	Paris	20	2024-09-19T10:11:20.000+00:00
4	Washington	10	2024-09-19T10:10:40.000+00:00
4	Washington	40	2024-09-19T10:11:00.000+00:00

streaming_df でサポートされている操作はどれですか?
A. streaming_df.select(countDistinct("名前"))
B. streaming_df.groupby("Id").count()
C. streaming_df.orderBy("timestamp").limit(4)
D. streaming_df.filter(col("count") < 30).show()
Answer: B (メーッセージを残す**)**
包括的かつ詳細な
Explanation:
構造化ストリーミングでは、無制限データの性質上、限られた操作のサブセットのみがサポートされます。
並べ替え (orderBy) やグローバル集計 (countDistinct) などの操作ではデータセットの完全なビューが必要ですが、特定のウォーターマークまたはウィンドウが定義されていない限り、ストリーミング データではこれは不可能です。
各オプションのレビュー:
A). select(countDistinct("名前"))
許可されていません - countDistinct() のようなグローバル集計には完全なデータセットが必要であり、ウォーターマークとウィンドウ ロジックなしではストリーミングで直接サポートされません。

参考: Databricks 構造化ストリーミング ガイド - サポートされていない操作。
B). groupby("Id").count() がサポートされています - キーを介したストリーミング集計 (groupBy("Id") など) がサポートされています。
Spark は各キーの中間状態を維持します。参考: Databricks Docs # 構造化ストリーミングの集計 (https://docs.databricks.com/structured-streaming/aggregation.html)
C). orderBy("timestamp").limit(4) は許可されていません - 並べ替えと制限にはストリームの完全なビュー (無限) が必要なので、ストリーミング データフレームではサポートされていません。参照: Spark Structured Streaming - サポートされていない操作 (ウォーターマーク/ウィンドウなしでの並べ替えは許可されていません)。
D). filter(col("count") < 30).show() は許可されていません - show() はバッチ DataFrame のデバッグに使用されるブロッキング操作です。ストリーミング DataFrame では許可されていません。参照: Structured Streaming Programming Guide
- show() のような出力操作はサポートされていません。
公式ガイドからの参考抜粋:
「orderBy、limit、show、countDistinctなどの操作は、結果を計算するのにデータセット全体を必要とするため、構造化ストリーミングではサポートされていません。増分集計には、代わりにgroupBy(...).agg(...)を使用してください。」 - Databricks構造化ストリーミングプログラミングガイド

最新問題: 17
データ変換に Spark 上の Pandas を使用する利点は何ですか?
オプション:
A. Python でのみ使用できるため、学習曲線が短縮されます。
B. 即時実行を使用して結果を計算するため、簡単に使用できます。
C. メモリバインドされた DataFrame でメモリを活用し、単一ノードでのみ実行されるため、コスト効率が高くなります。
D. クラスター内の利用可能なすべてのコアを使用してクエリをより高速に実行するほか、Pandas の豊富な機能セットも提供します。
Answer: D (メーッセージを残す)
Spark (旧称 Koalas) 上の Pandas API は以下を提供します。
使い慣れたPandas風の構文
Sparkを使った分散実行
クラスター全体の大規模データセットのスケーラビリティ
Pandas の生産性を維持しながら Spark のパワーを提供します。
参考資料: Spark の Pandas API ガイド

最新問題: 18

データエンジニアは、JSONイベントのストリームをリアルタイムで処理するApache Spark™ Structured Streamingアプリケーションを構築しています。エンジニアは、アプリケーションにフォールトトレラント性を持たせ、障害発生時に最後に正常に処理されたレコードから処理を再開したいと考えています。これを実現するために、データエンジニアはチェックポイントを実装することにしました。
データ エンジニアはどのコード スニペットを使用すべきでしょうか?
A. クエリ = streaming_df.writeStream \
.format("コンソール") \
.option("チェックポイント", "/path/to/checkpoint") \
.outputMode("追加") \
。始める（）
B. クエリ = streaming_df.writeStream \
.format("コンソール") \
.outputMode("追加") \
.option("checkpointLocation", "/path/to/checkpoint") \
。始める（）
C. クエリ = streaming_df.writeStream \
.format("コンソール") \
.outputMode("完了") \
。始める（）
D. クエリ = streaming_df.writeStream \
.format("コンソール") \
.outputMode("追加") \
。始める（）
Answer: (解答を表示する**)**
正確な抜粋からの包括的かつ詳細な説明：
フォールト トレランスを有効にし、障害発生後に Spark が最後にコミットされたオフセットから再開できるようにするには、正しいオプション キー「checkpointLocation」を使用してチェックポイントの場所を構成する必要があります。
公式 Spark Structured Streaming ガイドより:
ストリーミング クエリをフォールト トレラントかつ回復可能にするには、チェックポイント ディレクトリを指定する必要があります。
オプション("checkpointLocation", "/path/to/dir")."
オプションの説明:
オプション A は無効なオプション名「checkpoint」（正しくは「checkpointLocation」）を使用しています。オプション B は正しく、checkpointLocation が適切に設定されています。オプション C にはチェックポイントがないため、障害後に再開されません。オプション D にもチェックポイントの設定がありません。
参考: Apache Spark 3.5 ドキュメント # 構造化ストリーミング # フォールトトレランス セマンティクス

最新問題**: 19**
次のコードがあるとします:

```
inputStream
.withWatermark("event_time", "10 minutes")
.groupBy(window("event_time", "15 minutes"))
.count()
```

.withWatermark("event_time", "10分")
.groupBy(window("イベント時間", "15分"))
。カウント（）
ウォーターマークしきい値後に到着したデータはどうなるのでしょうか?
オプション:
A. ウォーターマークしきい値 (10 分) より後に到着したレコードは、15 分間のウィンドウ内であれば自動的に集計に含められます。
B. ウォーターマークしきい値から 10 分以上経過してから到着したデータは無視され、集計に含まれません。
C. 最新のウォーターマークから 10 分以上経過して到着したデータは集計に含まれますが、次のウィンドウに配置されます。
D. ウォーターマークにより、最新の event_time から 10 分以内に到着する遅延データが処理され、ウィンドウ集計に含まれるようになります。
Answer: B (メーッセージを残す**)**
Spark の透かしルールによると:
「ウォーターマークよりも古いレコード（イベント時間 < 現在のウォーターマーク）は遅すぎるとみなされ、削除されます。」したがって、レコードの event_time が（これまでに確認された最大 event_time - 10 分）よりも早い場合、そのレコードは破棄されます。

最新問題: 20
開発者は SparkSession を初期化します。

```
spark = SparkSession.builder \
.appName("Analytics Application") \
.getOrCreate()
```

spark = SparkSession.builder \
.appName("アナリティクスアプリケーション") \
.getOrCreate()
Spark の SparkSession について説明している記述はどれですか?
A. getOrCreate() メソッドは、既存の SparkSession を明示的に破棄し、新しい SparkSession を作成します。
B. SparkSession は各 appName ごとに一意であり、同じ名前で getOrCreate() を呼び出すと、作成済みの既存の SparkSession が返されます。
C. SparkSession がすでに存在する場合、このコードは新しいセッションを作成する代わりに既存のセッションを返します。
D. getOrCreate() メソッドが呼び出されるたびに新しい SparkSession が作成されます。
Answer: C (メーッセージを残す**)**
PySpark API ドキュメントによると:
「getOrCreate(): 既存のSparkSessionを取得します。既存のSparkSessionが存在しない場合は、このビルダーで設定されたオプションに基づいて新しいSparkSessionを作成します。」これは、SparkがJVMプロセス内でグローバルなシングルトンセッションを維持することを意味します。getOrCreate()を繰り返し呼び出しても、明示的に停止されない限り、同じセッションが返されます。
オプション A は不正解です。このメソッドはセッションを破棄しません。

オプション B では、一意性が appName に誤って結び付けられていますが、これはセッションの再利用性には影響しません。
オプション D は不正解です。getOrCreate() の基本的な動作と矛盾しています。
(出典: PySpark SparkSession API ドキュメント)

最新問題**: 21**
55 件中 7 件目。
開発者はSparkアプリケーションの問題をデバッグするよう依頼されました。開発者は、CSVファイルからロードされたデータがDataFrameに正しく読み込まれていないことを発見しました。
CSV ファイルは、次の Spark SQL ステートメントを使用して読み取られています。
CREATE TABLEの場所
csvの使用
オプション (パス '/data/locations.csv')
コマンド SELECT * FROM locations の最初の行は次のようになります。
| 都市 | 緯度 | 経度 |
|アルティ シドニー | -33... | ... |
CSV データを再度正しく読み取るために、開発者は CREATE TABLE ステートメントの OPTIONS 句にどのパラメータを追加できますか?
A. 'ヘッダー' 'true'
B. 'ヘッダー' '偽'
C. 'sep' ','
D. 'sep' '|'
Answer: A (メーッセージを残す**)**
Spark SQLまたはDataFrame APIを使用してCSVファイルを読み取る場合、Sparkはデフォルトでファイルの最初の行をヘッダーではなくデータとみなします。最初の行を列名として解釈するには、headerオプションをtrueに設定する必要があります。
正しい構文:
CREATE TABLEの場所
csvの使用
オプション（
パス '/data/locations.csv'、
ヘッダー「rue」
）;
これにより、Spark は最初の行を列ヘッダーとして読み取り、city、lat、long などの列を正しくマッピングします。
他のオプションが間違っている理由:
B (ヘッダー 'false'): デフォルトの動作。ヘッダーをデータとして読み取り続けます。
C / D (sep): 区切り文字を指定するために使用されます。ファイルで別の区切り文字 (例: |) が使用されない限り、関係ありません。
参考資料（Databricks Apache Spark 3.5 - Python / 学習ガイド）：
PySpark SQL データ ソース - CSV オプション (header、inferSchema、sep)。
Databricks 試験ガイド (2025 年 6 月) : セクション「Spark SQL の使用」- Spark SQL および DataFrame API を使用して、さまざまな形式のファイルからデータを読み取ります。

最新問題**: 22**

MLOpsエンジニアが、英語の文字列をスペイン語に翻訳する言語モデルを適用するPandas UDFを構築しています。初期コードでは、UDFを呼び出すたびにモデルをロードするため、データパイプラインのパフォーマンスが低下しています。
初期コードは次のとおりです。

```
def in_spanish_inner(df: pd.Series) -> pd.Series:
  model = get_translation_model(target_lang='es')
  return df.apply(model)

in_spanish = sf.pandas_udf(in_spanish_inner, StringType())
```

定義 in_spanish_inner(df: pd.Series) -> pd.Series:
モデル = get_translation_model(ターゲット言語 = 'es')
df.apply(model) を返す
in_spanish = sf.pandas_udf(in_spanish_inner, 文字列型())
MLOps エンジニアは、このコードをどのように変更すれば、言語モデルの読み込み回数を減らすことができますか?
A. Pandas UDF を PySpark UDF に変換する
B. Pandas UDF を Series → Series UDF から Series → Scalar UDF に変換します。
C. mapInPandas()関数呼び出しでin_spanish_inner()関数を実行します。
D. Pandas UDF を Series → Series UDF から Iterator[Series] → Iterator[Series] UDF に変換します。
Answer: D (メーッセージを残す)
提供されたコードは、Series-to-Series型のPandas UDFを定義しており、呼び出しごとに言語モデルの新しいインスタンスが作成されます（これはバッチごとに発生します）。これは非効率的であり、モデルの初期化が繰り返されるため、大きなオーバーヘッドが発生します。
モデルのロード頻度を減らすには、エンジニアはUDFをイテレータベースのPandas UDF（Iterator[pd.Series] -> Iterator[pd.Series]）に変換する必要があります。これにより、モデルは呼び出しごとに1回ではなく、エグゼキュータごとに1回ロードされ、複数のバッチで再利用できるようになります。
公式 Databricks ドキュメントより:
「シリーズの反復子からシリーズの反復子への UDF は、UDF の初期化にコストがかかる場合に役立ちます...たとえば、行/バッチごとに 1 回ではなく、エグゼキュータごとに 1 回 ML モデルをロードする場合などです。」
- Databricks 公式ドキュメント: Pandas UDF
正しい実装は次のようになります。
パイソン
コピー編集
@pandas_udf("文字列")
def translate_udf(batch_iter: イテレータ[pd.Series]) -> イテレータ[pd.Series]:
モデル = get_translation_model(ターゲット言語 = 'es')
batch_iter内のバッチの場合:
バッチを適用します(モデル)
このリファクタリングにより、get_translation_model() がバッチごとではなく、エグゼキュータ プロセスごとに 1 回呼び出されるようになり、パイプラインのパフォーマンスが大幅に向上します。

最新問題: **23**

開発者は、データセット内のすべてのシャッフル後のパーティションが spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold に設定された値よりも小さいことに気付きました。
この場合、Adaptive Query Execution (AQE) はどのタイプの結合を選択しますか?
A. 直交座標結合
B. シャッフルされたハッシュ結合
C. ブロードキャストネストループ結合
D. ソートマージ結合
Answer: B (メーッセージを残す)
アダプティブクエリ実行（AQE）は、実行時の実際のデータサイズに基づいて結合戦略を動的に選択します。シャッフル後のパーティションのサイズが、以下のしきい値を下回っている場合、
spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold
Spark ではシャッフルされたハッシュ結合の使用が優先されます。
Spark のドキュメントから:
「AQE は、シャッフル後のデータのサイズが設定されたしきい値内に収まるほど小さい場合に、シャッフルされたハッシュ結合を選択し、よりコストのかかるソートマージ結合を回避します。」したがって、次のようになります。
A は間違いです。カルテシアン結合は結合条件がない場合にのみ使用されます。
B が正解です。これは、AQE の下での小さなパーティション化されたシャッフル データの最適化された結合です。
C と D は他のシナリオでは使用されますが、この場合には使用されません。
最終回答：B

最新問題: 24
与えられた条件:
パイソン
コピー編集
スパーク.sparkContext.setLogLevel("<LOG_LEVEL>")
どのセットに Spark ドライバー LOG_LEVEL に適した構成設定が含まれていますか?
A. すべて、デバッグ、失敗、情報
B. エラー、警告、トレース、オフ
C. 警告、なし、エラー、致命的
D. 致命的、なし、情報、デバッグ
Answer: B (メーッセージを残す)
正確な抜粋からの包括的かつ詳細な説明：
SparkContextのsetLogLevel()メソッドは、ドライバーのログレベルを設定します。これにより、ジョブ実行中に出力されるログの詳細度が制御されます。サポートされるレベルはlog4jから継承され、以下のものが含まれます。
全て
デバッグ
エラー
致命的
情報
オフ

トレース
警告
公式の Spark および Databricks ドキュメントによると:
「有効なログ レベルは次のとおりです: ALL、DEBUG、ERROR、FATAL、INFO、OFF、TRACE、WARN。」提供されている選択肢のうち、オプション B (ERROR、WARN、TRACE、OFF) のみが 4 つの有効なログ レベルを含み、「FAIL」や「NONE」などの無効なレベルを除外します。
参考: Apache Spark API ドキュメント # SparkContext.setLogLevel

最新問題: **25**
アダプティブ クエリ実行 (AQE) の利点は何ですか?
A. Spark は実行前にクエリ プランを最適化できますが、実行時には適応しません。
B. 実行時にクエリ プランを調整し、偏ったデータを処理し、結合戦略を最適化し、全体的なクエリ パフォーマンスを向上させることができます。
C. タスクを並列化することでクエリ実行を最適化し、データスキューなどの実行時メトリックに基づいて戦略を調整しません。
D. クラスター内のノード間でタスクを自動的に分散し、クエリ プランに対して実行時の調整を実行しません。
Answer: B (メーッセージを残す**)**
アダプティブクエリ実行（AQE）は、Apache Spark 3.0で導入され、Spark 3.2以降ではデフォルトで有効化されている強力な最適化フレームワークです。実行時統計に基づいてクエリ実行プランを動的に調整することで、パフォーマンスを大幅に向上させます。AQEの主なメリットは以下のとおりです。
動的結合戦略の選択：AQEは実行時に結合戦略を切り替えることができます。例えば、結合の片側がブロードキャストできるほど小さいことを検出した場合、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することで、結合操作を最適化します。
偏ったデータの処理：AQEは結合操作中に偏ったパーティションを検出し、それらを小さなパーティションに分割します。このアプローチにより、タスク間のワークロードのバランスが取られ、データの偏りによって特定のタスクの実行時間が大幅に長くなるシナリオを回避できます。
シャッフル後のパーティションの結合: AQE は、実際のデータ サイズに基づいて小さなシャッフル パーティションをより大きなパーティションに動的に結合し、多数の小さなタスクを管理するオーバーヘッドを削減し、全体的なクエリ パフォーマンスを向上させます。
これらのランタイム最適化により、Spark はクエリ実行中に実際のデータ特性に適応できるようになり、リソースの使用効率が向上し、クエリ処理時間が短縮されます。

最新問題: **26**
データエンジニアは、上流のストリーミングソースから重複レコードが送信されていることに気づきました。重複レコードは同じキーを共有しており、event_timestamp のタイムスタンプの差は最大30分です。エンジニアは次のように付け加えました。
dropDuplicatesWithinWatermark("event_timestamp", "30分")
結果はどうなりましたか？
A. このシナリオでは重複排除を処理できません
B. ウォーターマークで指定された30分以内に到着した重複を削除します。
C. いつ到着したかに関係なく、すべての重複を削除します。
D. 数秒で透かしを受け入れ、コードはエラーになります
Answer: (解答を表示する**)**
正確な抜粋からの包括的かつ詳細な説明：
構造化ストリーミングのメソッドdropDuplicatesWithinWatermark()は、指定された列とウォーターマークウィンドウに基づいて重複レコードを削除します。ウォーターマークは、遅延データが有効とみなされるしきい値を定義します。
Spark のドキュメントから:

dropDuplicatesWithinWatermark は、イベント時間のウォーターマークウィンドウ内で発生した重複を削除します。」この場合、Spark は最初の発生を保持し、30 分のウォーターマークウィンドウ内の後続のレコードを削除します。
最終回答 :B

最新問題**: 27**
55 件中 25 件目。
データ アナリストは、employees_df で作業しており、給与に対して 10% の税金を計算する新しい列を追加する必要があります。
さらに、DataFrame には不要な列 age が含まれています。
どのコードフラグメントが税金列を追加し、年齢列を削除しますか?
A. employees_df = employees_df.withColumn("税金", col("給与") * 0.1).drop("年齢")
B. employees_df = employees_df.withColumn("税金", lit(0.1)).drop("年齢")
C. 従業員df = 従業員df.dropField("年齢").withColumn("税金", col("給与") * 0.1)
D. employees_df = employees_df.withColumn("税金", col("給与") + 0.1).drop("年齢")
Answer: A (メーッセージを残す**)**
Spark で新しい計算列を作成するには、.withColumn() メソッドを使用します。
不要な列を削除するには、.drop() メソッドを使用します。
正しい構文:
pyspark.sql.functionsからcolをインポートする
従業員df = 従業員df.withColumn("税金", col("給与") * 0.1).drop("年齢")
.withColumn("tax", col("salary") * 0.1) → 税金 = 給与の 10% となる新しい列を追加します。
.drop("age") → DataFrame から age 列を削除します。
他のオプションが間違っている理由:
B: lit(0.1) は計算された税金ではなく定数値を作成します。
C: .dropField() は DataFrame API メソッドではありません (構造体フィールドの操作でのみ使用されます)。
D: 10% を計算する代わりに、給与に 0.1 を加算します。
参照 :
PySpark DataFrame API - withColumn()、drop()、および col()。
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark DataFrame/DataSet API アプリケーションの開発」 - 列の操作、名前変更、および削除。

最新問題**: 28**
アダプティブ クエリ実行 (AQE) の利点は何ですか?
A. Spark は実行前にクエリ プランを最適化できますが、実行時には適応しません。
B. 実行時にクエリ プランを調整し、偏ったデータを処理し、結合戦略を最適化し、全体的なクエリ パフォーマンスを向上させることができます。
C. タスクを並列化することでクエリ実行を最適化し、データスキューなどの実行時メトリックに基づいて戦略を調整しません。
D. クラスター内のノード間でタスクを自動的に分散し、クエリ プランに対して実行時の調整を実行しません。
Answer: B (メーッセージを残す**)**
正確な抜粋からの包括的かつ詳細な説明 :
アダプティブクエリ実行 （AQE）は、Apache Spark 3.0で導入され、Spark 3.2以降ではデフォルトで有効化されている強力な最適化フレームワークです。実行時統計に基づいてクエリ実行プランを動的に調整することで、パフォーマンスを大幅に向上させます。AQEの主なメリットは以下のとおりです。

動的結合戦略の選択 :AQEは実行時に結合戦略を切り替えることができます。例えば、結合の片側がブロードキャストできるほど小さいことを検出した場合、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することで、結合操作を最適化します。
偏ったデータの処理 :AQEは結合操作中に偏ったパーティションを検出し、それらを小さなパーティションに分割します。このアプローチにより、タスク間のワークロードのバランスが取られ、データの偏りによって特定のタスクの実行時間が大幅に長くなるシナリオを回避できます。
シャッフル後のパーティションの結合: AQE は、実際のデータ サイズに基づいて小さなシャッフル パーティションをより大きなパーティションに動的に結合し、多数の小さなタスクを管理するオーバーヘッドを削減し、全体的なクエリ パフォーマンスを向上させます。
これらのランタイム最適化により、Spark はクエリ実行中に実際のデータ特性に適応できるようになり、リソースの使用効率が向上し、クエリ処理時間が短縮されます。

最新問題**: 29**
あるデータサイエンティストは、大量の構造化データの処理、SQLクエリの実行、機械学習アルゴリズムの適用を必要とするプロジェクトに取り組んでいます。このタスクにApache Sparkの使用を検討しています。
このシナリオでは、データ サイエンティストはどの Apache Spark モジュールの組み合わせを使用する必要がありますか?
オプション:
A. Spark DataFrames、構造化ストリーミング、GraphX
B. Spark SQL、Spark 上の Pandas API、構造化ストリーミング
C. Spark ストリーミング、GraphX、および Spark 上の Pandas API
D. Spark DataFrames、Spark SQL、MLlib
Answer: D (メーッセージを残す**)**
包括的な説明 :
Apache Spark で構造化データ処理、SQL クエリ、機械学習をカバーするには、コンポーネントの正しい組み合わせは次のとおりです。
Spark DataFrames: 構造化データ処理用
Spark SQL: 構造化データに対してSQLクエリを実行する
MLlib: Sparkのスケーラブルな機械学習ライブラリ
このトリオはまさにこのタイプの使用ケース向けに設計されています。
他のオプションが間違っている理由:
A: GraphX はグラフ処理用であり、ここでは必要ありません。
B: Spark 上の Pandas API は便利ですが、ML には MLlib が不可欠であり、このオプションでは省略されます。
C: Spark Streaming はレガシーです。GraphX はここでは無関係です。
参考:Apache Spark モジュールの概要

最新問題**: 30**
55 件中 48 件目。
データ エンジニアは複数の DataFrame を結合する必要があり、次のコードを記述しました。
pyspark.sql.functionsからブロードキャストをインポートする
データ1 = [(1, "A"), (2, "B")]
データ2 = [(1, "X"), (2, "Y")]
データ3 = [(1, "M"), (2, "N")]
df1 = spark.createDataFrame(data1, ["id", "val1"])
df2 = spark.createDataFrame(data2, ["id", "val2"])

df3 = spark.createDataFrame(data3, ["id", "val3"])
df_joined = df1.join(broadcast(df2), "id", "inner") \
.join(ブロードキャスト(df3)、"id", "inner")
このコードの出力は何でしょうか?
A. コードは正常に動作し、2 つのブロードキャスト結合を同時に実行して df1 を df2 と結合し、その結果を df3 と結合します。
B. 一度に実行できるブロードキャスト参加は 1 つだけなので、コードは失敗します。
C. 2 番目の結合条件 (df2.id == df3.id) が正しくないため、コードは失敗します。
D. broadcast() はインラインではなく結合の前に呼び出す必要があるため、コードはエラーになります。
Answer: (解答を表示する**)**
Spark は、ブロードキャストされた各 DataFrame が構成されたしきい値以下に収まるほど小さい限り、単一のクエリ プランで複数のブロードキャスト結合をサポートします。
実行計画:
Spark は df2 をすべての実行者にブロードキャストします。
df1 (big) をブロードキャストされた df2 に結合します。
次に、df3 をブロードキャストし、中間結果を使用して別の結合を実行します。
結果は効率的であり、大きなデータのシャッフルを回避します。
他のオプションが間違っている理由:
B: Spark 3.x では複数のブロードキャスト結合がサポートされています。
C: すべてが id をキーとして使用しているため、結合条件は正しいです。
D: broadcast() はインラインで使用できます。これは有効な構文です。
参照：
PySpark SQL 関数 - broadcast() の使用法。
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark DataFrame/DataSet API アプリケーションの開発」 - 複数のブロードキャスト結合の最適化。

最新問題**: 31**
このビュー定義があるとします:
df.createOrReplaceTempView("users_vw")
セッションが終了した後に users_vw ビューをクエリするにはどの方法を使用できますか?
オプション:
A. Sparkを使用してusers_vwをクエリする
B. users_vw データをテーブルとして保存します
C. users_vwを再作成し、Sparkを使用してデータをクエリします。
D. Sparkを使用してusers_vw定義とクエリを保存します。
Answer: B (メーッセージを残す**)**
createOrReplaceTempView のような一時ビューはセッション スコープです。
Spark セッションが終了すると消えます。
セッション間でデータを保持するには、データを永続化する必要があります。
df.write.saveAsTable("users_vw")
したがって、セッション終了後もビューを保持するには、ビューをテーブルとして保存する必要があります。

最新問題: **32**
どの Spark 構成が、エグゼキュータ上で並列実行できるタスクの数を制御しますか?
オプション:
A. spark.executor.cores
B. spark.task.maxFailures
C. spark.driver.cores
D. spark.executor.memory
Answer: A (メーッセージを残す)
spark.executor.cores は、エグゼキュータが実行できる同時タスクの数を決定します。
たとえば、4 に設定すると、各エグゼキュータは最大 4 つのタスクを並行して実行できます。
その他の設定:
spark.task.maxFailures はタスクの再試行ロジックを制御します。
spark.driver.cores は、executor 用ではなく、ドライバー用です。
spark.executor.memory はタスクの同時実行性ではなく、メモリの制限を設定します。
参考:Apache Spark の構成

最新問題: **33**
あるデータサイエンティストが、PySparkを使用してApache Sparkの大規模データセットに取り組んでいます。データサイエンティストは、user_id、product_id、purchase_amountの列を持つDataFramedfを所有しており、このデータに対していくつかの操作を効率的に実行する必要があります。
シャッフルを必要とする変換と、それに続くシャッフルを必要としない変換が生じる操作シーケンスはどれですか?
A. df.filter(df.purchase_amount > 100).groupBy("user_id").sum("purchase_amount")
B. df.withColumn("割引", df.purchase_amount * 0.1).select("割引")
C. df.withColumn("purchase_date", current_date()).where("total_purchase > 50")
D. df.groupBy("user_id").agg(sum("purchase_amount").alias("total_purchase")).repartition(10)
Answer: (解答を表示する)
正確な抜粋からの包括的かつ詳細な説明：
シャッフルは、groupBy、reduceByKey、joinなどの操作で発生し、これらの操作ではデータがパーティション間で移動されます。partition()操作でもシャッフルが発生する可能性がありますが、このコンテキストでは集計の後に発生します。
オプション D では、groupBy の後に aggres を実行すると、ノード間のグループ化によりシャッフルが発生します。
パーティション(10)はパーティショニング変換ですが、データがすでにグループ化されているため、新たなシャッフルは行われません。

このシーケンス (シャッフル (groupBy)、その後に非シャッフル (repartition)) は正しいです。
オプション A はその逆を行います。フィルターはシャッフルを引き起こしませんが、groupBy はシャッフルを引き起こします。これにより順序が間違ってしまいます。

最新問題: **34**
Apache Spark での実行中のジョブ、ステージ、タスク間の関係は何ですか?
オプション:
A. ジョブには複数のステージが含まれ、各ステージには複数のタスクが含まれます。
B. ジョブには複数のタスクが含まれ、各タスクには複数のステージが含まれます。
C. ステージには複数のジョブが含まれ、各ジョブには複数のタスクが含まれます。
D. ステージには複数のタスクが含まれ、各タスクには複数のジョブが含まれます。
Answer: A (メーッセージを残す)
Sparkjob はアクション (カウント、表示など) によってトリガーされます。
ジョブはステージに分割され、通常はシャッフル境界ごとに 1 つになります。
各ステージは複数のタスクに分割され、ワーカー ノード間に分散されます。
参考:Spark実行モデル

最新問題: **35**
55件中34件目。
データ エンジニアは、スケジュールされたバッチ ジョブ中に十分に活用されていない Spark クラスターを調査しています。
Spark ログを確認したところ、タイムアウト エラーによりタスクが頻繁に強制終了されていること、またログにリソース不足に関する警告がいくつか記録されていることに気付きました。
十分に活用されていない問題を解決するために、エンジニアはどのようなアクションを取る必要がありますか?
A. spark.network.timeout プロパティを設定すると、タスクが強制終了されることなく完了するまでの時間が長くなります。
B. Spark 構成でエグゼキュータのメモリ割り当てを増やします。
C. タスクのスケジュールを改善するために、データ パーティションのサイズを縮小します。
D. より多くの同時タスクを処理するために、エグゼキュータ インスタンスの数を増やします。
Answer: D (メーッセージを残す)
タイムアウト警告を伴う使用率の低さは、多くの場合、並列処理が不十分であることを示します。つまり、すべてのタスクを同時に処理するのに十分なエグゼキューターがないことを意味します。
解決：
実行プログラムの数を増やすと、より多くのタスクを並列実行でき、リソースの使用率が向上します。
構成例:
--conf spark.executor.instances=8
これにより、ワークロードがクラスター ノード間でより効率的に分散され、保留中のタスクのアイドル時間が短縮されます。
他のオプションが間違っている理由:
A: タイムアウトを延長すると、根本的な原因 (実行者の不足) ではなく症状が隠れてしまいます。
B: 実行者あたりのメモリを増やしても、スケジュールのボトルネックは解消されません。
C: パーティション サイズを縮小するとオーバーヘッドが増加する可能性があり、リソースの不均衡は修正されません。

参照：
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark DataFrame API アプリケーションのトラブルシューティングとチューニング」- エグゼキューターとクラスターの使用率のチューニング。
Spark 構成 - 実行インスタンスとリソースのスケーリング。

最新問題: **36**
与えられた条件:
パイソン
コピー編集
スパーク.sparkContext.setLogLevel("<LOG_LEVEL>")
どのセットに Spark ドライバー LOG_LEVEL に適した構成設定が含まれていますか?
A. すべて、デバッグ、失敗、情報
B. エラー、警告、トレース、オフ
C. 警告、なし、エラー、致命的
D. 致命的、なし、情報、デバッグ
Answer: B (メーッセージを残す**)**
SparkContext の setLogLevel() メソッドは、ドライバーのログレベルを設定します。これにより、ジョブ実行中に出力されるログの詳細度が制御されます。
サポートされるレベルは log4j から継承され、以下のものが含まれます。
全て
デバッグ
エラー
致命的
情報
オフ
トレース
警告
公式の Spark および Databricks ドキュメントによると:
「有効なログ レベルは次のとおりです: ALL、DEBUG、ERROR、FATAL、INFO、OFF、TRACE、WARN。」提供されている選択肢のうち、オプション B (ERROR、WARN、TRACE、OFF) のみが 4 つの有効なログ レベルを含み、「FAIL」や「NONE」などの無効なレベルを除外します。

最新問題: **37**
データエンジニアは、ファイルベースのデータソースを特定の場所に永続化する必要があります。しかし、Spark はデフォルトでウェアハウスディレクトリ（例/user/hive/warehouse）に書き込みます。これをオーバーライドするには、エンジニアはファイルパスを明示的に定義する必要があります。
データが特定の場所に保存されることを保証するコード行はどれですか?
オプション:
A. users.write(path="/some/path").saveAsTable("default_table")
B. users.write.saveAsTable("default_table").option("path", "/some/path")
C. users.write.option("path", "/some/path").saveAsTable("default_table")
D. users.write.saveAsTable("default_table", path="/some/path")
Answer: C (メーッセージを残す**)**

テーブルを永続化し、保存パスを指定するには、次を使用します。
users.write.option("path","/some/path").saveAsTable("default_table")
saveAsTable を呼び出す前に .option("path", ...) を適用する必要があります。
オプション A では無効な構文 (write(path=...)) が使用されています。
オプション B は、saveAsTable() の後に.option() を適用しますが、これは遅すぎます。
オプション D では誤った構文が使用されています (saveAsTable にパス パラメータがありません)。
参考:Spark SQL - テーブルとして保存

最新問題**: 38**
55 件中 5 件目。
Apache Spark での実行中のジョブ、ステージ、タスク間の関係は何ですか?
A. ジョブには複数のタスクが含まれ、各タスクには複数のステージが含まれます。
B. ステージには複数のジョブが含まれ、各ジョブには複数のタスクが含まれます。
C. ステージには複数のタスクが含まれ、各タスクには複数のジョブが含まれます。
D. ジョブには複数のステージが含まれ、各ステージには複数のタスクが含まれます。
Answer: D (メーッセージを残す)
Apache Spark の実行階層では、関係は次のように構造化されます。
ジョブ: RDD または DataFrame でアクション (例: count()、collect()、save()) がトリガーされたときに作成されます。
ステージ: 各ジョブは、シャッフル境界 (例: reduceByKey または join の後) によって区切られた 1 つ以上のステージに分割されます。
タスク: 各ステージは、パーティションごとに 1 つずつ、エグゼキュータで並列に実行される複数のタスクで構成されます。
実行階層:
ジョブ → ステージ → タスク
したがって、ジョブには複数のステージが含まれ、各ステージには複数のタスクが含まれます。
他のオプションが間違っている理由:
A: ジョブにはステージのないタスクが直接含まれません。
B: ステージには複数のジョブを含めることはできません。ステージは 1 つのジョブに属します。
C: タスクにジョブが含まれていません。
参考資料（Databricks Apache Spark 3.5 - Python / 学習ガイド）：
Spark アーキテクチャの概要 - 実行階層: ジョブ、ステージ、およびタスク。
Databricks 試験ガイド (2025 年 6 月) : セクション「Apache Spark のアーキテクチャとコンポーネント」- 実行階層と遅延評価について説明します。

最新問題**: 39**
開発者は、Spark 3.5.0で導入された組み込み関数を活用するために、古いSparkコードをリファクタリングしたいと考えています。既存のコードでは、配列操作を手動で実行しています。次のコードスニペットのうち、Spark 3.5.0の新しい組み込み関数を配列操作に利用しているのはどれですか？

```
import pyspark.sql.functions as F

min_price = 110.50

result_df = prices_df \
  .filter(F.col("spot_price") >= F.lit(min_price)) \
  .agg(F.count("*"))
```

A.

```
result_df = prices_df \
  .withColumn("valid_price", F.when(F.col("spot_price") > F.lit(min_price), 1).otherwise(0))
```

result_df = prices_df \
.withColumn("valid_price", F.when(F.col("spot_price") > F.lit(min_price), 1).otherwise(0))

```
result_df = prices_df \
  .agg(F.count_if(F.col("spot_price") >= F.lit(min_price)))
```

B.

result_df = prices_df \
.agg(F.count_if(F.col("スポット価格") >= F.lit(最小価格)))

```
result_df = prices_df \
  .agg(F.min("spot_price"), F.max("spot_price"))
```

C.

result_df = prices_df \
.agg(F.min("スポット価格"), F.max("スポット価格"))

```
result_df = prices_df \
  .agg(F.count("spot_price").alias("spot_price")) \
  .filter(F.col("spot_price") > F.lit("min_price"))
```

D.

result_df = prices_df \
.agg(F.count("スポット価格").alias("スポット価格")) \
.filter(F.col("スポット価格") > F.lit("最小価格"))

Answer: B (メーッセージを残す)

count_if(condition) は、指定されたブール条件を満たす行の数をカウントします。

この例では、spot_price >= min_price が true と評価される回数を直接カウントし、when/otherwise とフィルタリングまたは合計の古い冗長な組み合わせを置き換えます。

公式の Spark 3.5.0 ドキュメントには、この種のロジックを簡素化するために count_if が追加されたことが記されています。

ブール条件が成立する行のみをカウントする count_if 集計関数を追加しました (SPARK-43773)。その他のオプションが間違っている、または古くなっている理由:

A は、フラグ列を追加するレガシー スタイルの方法 (when().otherwise()) を使用しますが、これは count_if と比較すると冗長です。

C は単純な最小/最大集計を実行します。これは便利ですが、条件付き配列操作や更新された機能とは関係ありません。

D は .agg() の後に .filter() を誤って適用し、エラーを引き起こし、変数ではなく文字列 "min_price" を誤用します。
したがって、Spark 3.5.0 の新しい機能を正しく効率的に活用できるのは B だけです。
Explanation:
正解は B です。これは、集計内の条件付きカウントを簡素化する、Spark 3.5.0 で導入された新しい関数 count_if を使用しているためです。

最新問題: **40**
データ エンジニアは、2 つの DataFramesdf1 と df2 を結合する次のコードを記述します。
df1 = spark.read.csv("sales_data.csv") # 約10GB
df2 = spark.read.csv("product_data.csv") # 約 8 MB
結果 = df1.join(df2, df1.product_id == df2.product_id)

```
df1 = spark.read.csv("sales_data.csv")
df2 = spark.read.csv("product_data.csv")
result = df1.join(df2, df1.product_id == df2.product_id)
```

Spark はどの結合戦略を使用しますか?
A. シャッフル結合。AQE が有効になっていないため、Spark は静的クエリ プランを使用します。
B. df2 がデフォルトのブロードキャストしきい値より小さいため、ブロードキャスト参加
C. df1とdf2のサイズ差が大きすぎるため、ブロードキャスト結合を効率的に実行できないため、シャッフル結合を実行します。
D. ブロードキャストヒントが提供されなかったため、シャッフル結合しました
Answer: (解答を表示する)
正確な抜粋からの包括的かつ詳細な説明 :
Spark のデフォルトのブロードキャスト参加しきい値は次のとおりです。
spark.sql.autoBroadcastJoinThreshold = 10MB
df2 はわずか 8 MB (10 MB 未満) なので、Spark は明示的なヒントを必要とせずにブロードキャスト結合を自動的に適用します。
Spark のドキュメントから:
結合の片側がブロードキャストしきい値より小さい場合、Spark はそれをすべてのエグゼキューターに自動的にブロードキャストします。」A は不正解です。Spark は静的プランでも自動ブロードキャストをサポートしているからです。
B が正解です。Spark は df2 を自動的にブロードキャストします。
C と D は、Spark のデフォルトのロジックがこの最適化を処理するため、正しくありません。
最終回答 :B

最新問題: **41**
データエンジニアは、新しいマネージドテーブルを作成するSparkジョブを作成したいと考えています。テーブルが既に存在する場合、ジョブは失敗し、何も変更されません。
どの保存モードと方法を使用すればよいですか?
A. ErrorIfExists モードの saveAsTable
B. 上書きモードで saveAsTable を保存
C. 無視モードで保存
D. ErrorIfExists モードで保存
Answer: A (メーッセージを残す)

メソッド saveAsTable() は新しいテーブルを作成し、テーブルが存在する場合はオプションで失敗します。
Spark のドキュメントより:
「モード 'ErrorIfExists' (デフォルト) では、テーブルがすでに存在する場合にエラーが発生します。」つまり、
オプションAが正解です。
オプション B (上書き) は既存のデータを上書きするため、ここでは受け入れられません。
オプション C と D は save() を使用しますが、メタストアにメタデータを含む管理テーブルは作成されません。
最終答え :A

最新問題**: 42**
あるデータエンジニアが、Apache Spark Structured Streamingを用いたリアルタイム分析パイプラインの開発に取り組んでいます。エンジニアは、入力データを処理し、クエリ実行時にトリガーが適切に制御されるようにしたいと考えています。システムは、5秒間隔でマイクロバッチ処理を行い、データを処理する必要があります。
データ エンジニアがこの要件を満たすために使用できるコード スニペットはどれですか?
A)

```
query = df.writeStream \
    .outputMode("append") \
    .trigger(continuous='5 seconds') \
    .start()
```

B)

```
query = df.writeStream \
    .outputMode("append") \
    .trigger() \
    .start()
```

C)

```
query = df.writeStream \
    .outputMode("append") \
    .trigger(processingTime='5 seconds') \
    .start()
```

ダ）

```
query = df.writeStream \
    .outputMode("append") \
    .trigger(processingTime=5000) \
    .start()
```

オプション:
A. トリガー(連続='5秒')を使用します - 連続処理モード。
B. trigger() を使用します - 間隔のないデフォルトのマイクロバッチトリガー。
C. トリガー(processingTime='5 seconds') を使用します - 間隔付きの正しいマイクロバッチ トリガーです。
D. trigger(processingTime=5000) を使用します - processingTime には文字列が必要なので無効です。

Answer: C (メーッセージを残す)
マイクロバッチ間隔を定義する正しい構文は次のとおりです。
クエリ = df.writeStream \
.outputMode("追加") \
.trigger(処理時間='5秒') \
。始める（）
これにより、クエリが 5 秒ごとに実行されるようにスケジュールされます。
連続モード (オプション A で使用) は実験的なモードであり、シンクのサポートが制限されています。
オプション D は、processingTime が整数ではなく文字列である必要があるため、正しくありません。
オプション B は、間隔制御なしでできるだけ早くトリガーします。

最新問題**: 43**
55 件中 40 件目。
開発者は、Spark 3.5 で導入された組み込み関数を活用するために、古い Spark コードをリファクタリングしたいと考えています。
元のコード:
pyspark.sqlから関数をFとしてインポートする
最小価格 = 110.50
result_df = prices_df.filter(F.col("price") > min_price).agg(F.count("*")) 開発者はコードをリファクタリングするためにどのコード ブロックを使用する必要がありますか?
A. result_df = prices_df.filter(F.col("price") > F.lit(min_price)).agg(F.count("*"))
B. result_df = prices_df.where(F.lit("price") > min_price).groupBy().count()
C. result_df = prices_df.withColumn("valid_price", when(col("price") > F.lit(min_price), True))
D. result_df = prices_df.filter(F.lit(min_price) > F.col("price")).count()
Answer: A (メーッセージを残す)
DataFrame 式内の列の値を Python リテラル定数と比較するには、F.lit() を使用して Spark リテラルに変換します。
正しいリファクタリング:
pyspark.sqlから関数をFとしてインポートする
最小価格 = 110.50
result_df = prices_df.filter(F.col("price") > F.lit(min_price)).agg(F.count("*")) これにより、型の不一致が回避され、Spark がクラスター上でフィルター式を実行することが保証されます。
他のオプションが間違っている理由:
B: where() 構文は有効ですが、F.lit("price") は正しくありません。列ではなく文字列リテラルをラップします。
C: withColumn は、この集計に必要のない列を追加します。
D: 比較ロジックが逆になっています。
参照：
PySpark SQL 関数 - lit()、col()、および DataFrame フィルター。
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark DataFrame/DataSet API アプリケーションの開発」 - フィルタリング、リテラル、集計。

最新問題**: 44**
55 件中 24 件目。

events.parquet の場所に保存されている Parquet ファイルのスキーマを表示するには、どのコードを使用する必要がありますか?
A. spark.sql("events.parquet から * を選択").show()
B. spark.read.format("parquet").load("events.parquet").show()
C. spark.read.parquet("events.parquet").printSchema()
D. spark.sql("events.parquetからスキーマを選択").show()
Answer: (解答を表示する**)**
Parquet ファイルのスキーマを表示するには、DataFrameReader を使用して Parquet データを読み込み、.printSchema() メソッドを呼び出す必要があります。
正しい構文:
spark.read.parquet("events.parquet").printSchema()
このコマンドは、ファイルのメタデータを読み込み (完全な読み取りをトリガーせずに)、列名、データ型、および NULL 値可能性情報をツリー形式で出力します。
他のオプションが間違っている理由:
A/D: SQL クエリはファイル スキーマを直接イントロスペクトできません。
B: .show() はスキーマではなくデータ行を表示します。
参照 :
PySpark DataFrameReader API - read.parquet() および DataFrame.printSchema()。
Databricks 試験ガイド (2025 年 6 月) : セクション 「Spark SQL の使用」 - Spark SQL および DataFrame API でのファイルの読み取りとスキーマの調査について説明します。

最新問題: 45
データエンジニアは、sensor_id、temperature、timestampの列を持つセンサーの読み取りデータを毎秒受信するストリーミングデータフレームを処理したいと考えています。エンジニアは、データのストリーミング中に、過去5分間の各センサーの平均温度を計算する必要があります。
どのコード実装が要件を満たしていますか?
提供された画像からのオプション:

A.
```
df.withColumn("avg_temp", avg("temperature")
  .over(Window.partitionBy("sensor_id")
```

B.
```
df.groupBy("sensor_id", "timestamp")
  .agg(avg("temperature").alias("avg_temp"))
```

C.
```
df.groupBy("sensor_id").avg("temperature")
```

D.
```
df.withWatermark("timestamp", "5 minutes")
 .groupBy("sensor_id", window("timestamp", "5 minutes"))
 .agg(avg("temperature").alias("avg_temp"))
```

Answer: D (メーッセージを残す**)**
正確な抜粋からの包括的かつ詳細な説明 :
正解は D です。これは、適切な時間ベースのウィンドウ集計とウォーターマークを使用するためです。これは、イベント時間データの時間ベースの集計に Spark Structured Streaming で必要なパターンです。

構造化ストリーミングに関する Spark 3.5 ドキュメントより:
イベント時間列にスライディングウィンドウを定義し、groupBをwindow()とともに使用して、それらのウィンドウの集計を計算できます。遅延データを処理するには、withWatermark()を使用して、遅延データの到着許容範囲を指定します。(出典:Structured Streaming Programming Guide) オプションDでは、以下の使用法があります。
パイソン
コピー編集
groupBy("sensor_id", window("timestamp","5分"))
agg(avg("温度").alias("avg_temp"))
各sensor_idについて、5分間のイベント時間ウィンドウにわたって平均温度が計算されることを保証します。ロジックを完成させるために、パイプラインの早い段階でwithWatermark("timestamp", "5 minutes")を使用して、遅いイベントを処理することを前提としています。
他のオプションが間違っている理由の説明:
オプション A は静的 DataFrame またはバッチ クエリに適用され、ストリーミング集計には適していない Window.partitionBy を使用します。
オプション B では時間ウィンドウが適用されないため、5 分間の移動平均は計算されません。
オプション C は集計後に誤って ApplyWithWatermark() を適用し、時間ウィンドウが含まれていないため、必要な時間ベースのグループ化が欠落しています。
したがって、オプション D は、時間ウィンドウ ストリーミング集約を計算するためのすべての要件を満たす唯一のオプションです。

最新問題: **46**
開発者は、次のような小さな Parquet テーブルに保存されたデータを使用して Python 辞書を作成する必要があります。

```
+---------+------------+
|region_id|region      |
+---------+------------+
|1        |AMERICA     |
|3        |EUROPE      |
|0        |AFRICA      |
|4        |MIDDLE EAST|
|2        |ASIA        |
+---------+------------+
```

結果の Python 辞書には、最小の 3 つの region_id 値を含む region -> region id のマッピングが含まれている必要があります。
どのコードフラグメントが要件を満たしていますか?
A)

```
regions = dict(
  regions_df \
    .select('region', 'region_id') \
    .sort('region_id') \
    .take(3)
)
```

B)

C)

ダ）

結果のPython辞書には、最小のregion -> region_idのマッピングが含まれている必要があります。
3つのregion_id値。
どのコードフラグメントが要件を満たしていますか?

A. 地域 = 辞書(
地域_df
.select('地域', '地域ID')
.sort('地域ID')
.take(3)
)

B. 地域 = 辞書(
地域_df
.select('region_id', 'region')
.sort('地域ID')
.take(3)
)

C. 領域 = dict(
地域_df
.select('region_id', 'region')
.limit(3)

。集める（）
）
D. 地域 = 辞書(
地域_df
.select('地域', '地域ID')
.sort(desc('region_id'))
.take(3)
）
Answer: A (メーッセージを残す**)**
正確な抜粋からの包括的かつ詳細な説明：
この問題では、キーが地域値、値が対応する地域ID整数である辞書を作成する必要があります。さらに、最小の地域ID値3つだけを取得するように求められています。
主な観察事項:
select('region', 'region_id') は、dict() によって期待されるとおりに列の順序を設定します。最初の列がキーになり、2 番目の列が値になります。
sort('region_id') は、最小の ID が最初に表示されるように昇順で並べ替えます。
take(3)はちょうど3行を取得します。
結果をラップすると、必要なPython辞書が正しく構築されます:{ 'AFRICA': 0, 'AMERICA': 1,
'アジア': 2 }。
誤ったオプション:
オプション B は順序を反転して region_idfirst とし、整数キーを持つ辞書を作成しますが、これは要求されたものではありません。
オプション C はソートなしで .limit(3) を使用するため、パーティション レイアウトに基づいて行が非決定的になります。
オプション D は降順で並べ替え、最小の region_id ではなく最大の region_id を返します。
したがって、オプション A はすべての要件を正確に満たしています。

最新問題: **47**
エンジニアは、/file/test_data.orc にある大きな ORC ファイルを持っており、メモリ使用量を削減するために特定の列のみを読み取りたいと考えています。
読み取りプロセス中に列 (col1、col2) を選択するコード フラグメントはどれですか。
A. spark.read.orc("/file/test_data.orc").filter("col1 = '値' ").select("col2")
B. spark.read.format("orc").select("col1", "col2").load("/file/test_data.orc")
C. spark.read.orc("/file/test_data.orc").selected("col1", "col2")
D. spark.read.format("orc").load("/file/test_data.orc").select("col1", "col2")
Answer: D (メーッセージを残す**)**

正確な抜粋からの包括的かつ詳細な説明：
ORC ファイルから特定の列をロードする正しい方法は、まず .load() を使用してファイルをロードし、次に適用することです。
結果のDataFrameに対してselect()を実行します。これは.read.format("orc")またはショートカット.read.orc()で有効です。
df = spark.read.format("orc").load("/file/test_data.orc").select("col1","col2") 他のコードが間違っている理由:
フィルタリング後に選択を実行しますが、ロード時にメモリを最小限に抑えるという意図と一致しません。
Bin は誤って .select() を .load() の前に使用しようとしますが、これは無効です。
存在しない.selected() メソッドを使用します。
正しく読み込まれてから選択されます。
参考資料:Apache Spark SQL API - ORC 形式

最新問題**: 48**
データ エンジニアは、ストリーミング データフレームを Parquet ファイルとして書き込む必要があります。
次のコードが与えられます:

```
df
.writeStream
// -- insert code here --
.checkpointLocation("path/to/checkpoint/dir")
.start()
```


要件を満たすにはどのコードフラグメントを挿入する必要がありますか?
A)

```
.format("parquet")
.option("location", "path/to/destination/dir")
```

B)

```
.option("format", "parquet")
.option("destination", "path/to/destination/dir")
```

C)

```
.option("format", "parquet")
.option("location", "path/to/dest
```

ダ）

```
.format("parquet")
.option("path", "path/to/destinat
```

要件を満たすにはどのコードフラグメントを挿入する必要がありますか?
A. .format("parquet")
.option("location", "path/to/destination/dir")
B. コピー編集
.option("format", "parquet")
.option("destination", "path/to/destination/dir")
C. .option("format", "parquet")

.option("location", "path/to/destination/dir")
D. .format("parquet")
.option("path", "path/to/destination/dir")
Answer: D (メーッセージを残す)
正確な抜粋からの包括的かつ詳細な説明：
構造化ストリーミング DataFrame を Parquet ファイルに書き込むには、形式と出力ディレクトリを指定する正しい方法は次のとおりです。
書き込みストリーム
フォーマット("寄木細工")
オプション("パス", "パス/宛先/ディレクトリ")
Spark のドキュメントによると:
ファイルベースのシンクに書き込む場合（Parquetなど）、.option("path", ...)メソッドを使用してパスを指定する必要があります。バッチ書き込みとは異なり、.save()はサポートされていません。オプションAでは、.option("location", ...)が誤って使用されています（Parquetシンクでは無効です）。
オプション B は、正しい方法ではないにもかかわらず、via.option("format", ...) を使用して誤ってフォーマットを設定しています。
オプション C は同じ問題を繰り返します。
オプション D が正解です: .format("parquet")+.option("path", ...) が必須の構文です。
最終回答：D

最新問題: **49**
Spark アプリケーション開発者は、どの操作がシャッフルを引き起こし、Spark 実行プランの新しいステージにつながるのかを特定したいと考えています。
どの操作によりシャッフルが行われ、新しいステージが作成されますか?
A. DataFrame.groupBy().agg()
B. DataFrame.filter()
C. DataFrame.withColumn()
D. データフレーム.select()
Answer: (解答を表示する)
パーティション間でのデータ移動をトリガーする操作 (groupBy、join、repartition など) により、シャッフルと新しいステージが生成されます。
Spark のドキュメントより:
「groupBy と集計により、パーティション間でデータがシャッフルされ、同じキーを持つ行が結合されます。」オプション A (groupBy + agg) → シャッフルが発生します。
オプション B、C、および D (filter、withColumn、select) → シャッフルを必要としない変換。依存関係が狭いです。
最終答え：A

最新問題: **50**
データ エンジニアは構造化ストリーミング パイプラインを構築しており、パイプラインが中断したところから続行することで、パイプラインが障害または意図的なシャットダウンから回復することを望んでいます。
これを実現するにはどうすればよいでしょうか?
A. オプションcheckpointLocationduringreadStreamを設定することで
B. SparkSessionの初期化中にオプションrecoveryLocationを設定することで
C. オプションrecoveryLocationduringwriteStreamを設定することで
D. オプションcheckpointLocationduringwriteStreamを設定することで

Answer: (解答を表示する**)**
正確な抜粋からの包括的かつ詳細な説明：
構造化ストリーミングクエリを障害や意図的なシャットダウンから回復させるには、writeStream操作中にcheckpointLocationオプションを指定することが不可欠です。このチェックポイント位置にはストリーミングクエリの進行状況情報が保存され、中断した時点から再開できるようになります。
Databricks のドキュメントによると:
次の例のように、ストリーミング クエリを実行する前に checkpointLocation オプションを指定する必要があります。
オプション("checkpointLocation", "/path/to/checkpoint/dir")
toTable("catalog.schema.table")
- Databricks ドキュメント: 構造化ストリーミング チェックポイント
checkpointLocationduringwriteStream を設定することにより、Spark は状態情報を維持し、信頼性の高いストリーミング アプリケーションにとって重要な 1 回限りの処理セマンティクスを保証できます。

最新問題**: 51**
55 件中 35 件目。
データ エンジニアは構造化ストリーミング パイプラインを構築しており、中断したところから続行することで障害または意図的なシャットダウンから回復したいと考えています。
これを実現するにはどうすればよいでしょうか?
A. SparkSession の初期化中にオプション recoveryLocation を構成します。
B. readStream 中にオプション checkpointLocation を構成することによって。
C. writeStream 中にオプション checkpointLocation を構成することによって。
D. writeStream 中にオプション recoveryLocation を構成することによって。
Answer: (解答を表示する**)**
構造化ストリーミングでは、チェックポイントは、障害または再起動後にストリームを再開するために必要な状態情報 (オフセット、進行状況、メタデータ) を保存します。
正しい使い方:
ストリーミング出力を書き込むときに checkpointLocation オプションを設定します。
ストリーミング_df.writeStream \
.format("デルタ") \
.option("checkpointLocation", "/path/to/checkpoint/dir") \
.start("/path/to/output")
Spark は、このチェックポイント ディレクトリを使用して、進行状況を自動的に回復し、正確に 1 回のセマンティクスを維持します。
他のオプションが間違っている理由:
A/D: recoveryLocation は有効な Spark 構成オプションではありません。
B: チェックポイントは、readStream 中ではなく、writeStream 中に構成する必要があります。
参照：
PySpark 構造化ストリーミング ガイド - チェックポイントとリカバリ。
Databricks 試験ガイド (2025 年 6 月) : セクション 構造化ストリーミング」- チェックポイントとフォールト トレラント ストリーミング回復について説明します。

最新問題**: 52**

DataFrame を書き込むときに既存の JSON ファイルを上書きするコマンドはどれですか?
A. df.write.mode("overwrite").json("ファイルへのパス")
B. df.write.overwrite.json("ファイルへのパス")
C. df.write.json("ファイルへのパス", overwrite=True)
D. df.write.format("json").save("ファイルへのパス", モード="上書き")
Answer: A (メーッセージを残す)
DataFrameWriter を使用して既存のファイルを上書きする正しい方法は次のとおりです。
df.write.mode("上書き").json("ファイルへのパス")
オプション D も技術的には有効ですが、オプション A は最も簡潔で慣用的な PySpark 構文です。
リファレンス:PySpark DataFrameWriter API

最新問題**: 53**
エンジニアは2つのDataFrame dfとdf1（小）df2（大）を持っています。ブロードキャスト結合を使用します。
パイソン
コピー編集
pyspark.sql.functionsimportbroadcastから
結果 = df2.join(broadcast(df1), on='id', how='inner')
このシナリオでbroadcast() を使用する目的は何ですか?
オプション:
A. 結合を実行する前に ID 値をフィルタリングします。
B. df1 と df2 のパーティション サイズを増加させます。
C. 小さい DataFrame をすべてのノードに複製することで、シャッフル操作の数を減らします。
D. ID 値が同一の場合にのみ結合が行われるようにします。
Answer: C (メーッセージを残す)
broadcast(df1) は、小さな DataFrame (df1) をすべてのワーカー ノードに送信するように Spark に指示します。
これにより、結合中に df1 をシャッフルする必要がなくなります。
ブロードキャスト結合は、1 つの大きなテーブルと 1 つの小さなテーブルがあるシナリオに最適化されています。
参考資料:Spark SQL パフォーマンスチューニングガイド - ブロードキャスト結合

最新問題**: 54**
Spark Connect の機能は何ですか?
A. DataStreamReader、DataStreamWriter、StreamingQuery、Streaming APIをサポートします。
B. DataFrame、関数、列、SparkContext PySpark API をサポート
C. PySparkアプリケーションのみをサポートします
D. 認証機能が組み込まれています
Answer: A (メーッセージを残す)
正確な抜粋からの包括的かつ詳細な説明：
Spark Connect は、Apache Spark 3.4 で導入されたクライアント サーバー アーキテクチャであり、クライアントを Spark ドライバーから分離して、Spark クラスターへのリモート接続を可能にするように設計されています。
Spark 3.5.5 のドキュメントによると:

DataStreamReader、DataStreamWriter、StreamingQuery、StreamingQueryListener など、ストリーミング API の大部分がサポートされています。」これは、Spark Connect が Structured Streaming の主要コンポーネントをサポートし、堅牢なストリーミング データ処理機能を可能にすることを示しています。
その他のオプションについて:
B)Spark Connect は DataFrame、Functions、および Column API をサポートしていますが、SparkContext および RDD API はサポートしていません。
C).Spark Connect は、PySpark だけでなく、PySpark や Scala など複数の言語をサポートしています。
D).Spark Connect には認証機能が組み込まれていませんが、既存の認証インフラストラクチャとシームレスに連携するように設計されています。

最新問題: **55**
55 件中 23 件目。
あるデータサイエンティストは、単一マシンのメモリ容量を超える大規模なデータセットを扱っています。データサイエンティストは、標準的なPythonスクリプトなどの従来の単一マシン言語の代わりに、Apache Spark™の使用を検討しています。
このシナリオでは、Apache Spark™ は通常のシングルマシン言語に比べてどのような 2 つの利点がありますか? (回答を 2 つ選択してください)
A. データ処理タスクをマシンのクラスター全体に分散し、水平スケーラビリティを実現します。
B. 実行するには特殊なハードウェアが必要なので、汎用ハードウェア クラスターには適していません。
C. ディスク ストレージ上でのみデータを処理するため、メモリ リソースの必要性が削減されます。
D. コードを記述する必要がなく、すべてのデータ処理が自動的に処理されます。
E. フォールト トレランスが組み込まれているため、計算中にノード障害が発生してもシームレスに回復できます。
Answer: A,E (メーッセージを残す**)**
Apache Spark は、大規模なクラスターベースの計算用に設計された分散データ処理エンジンです。
利点:
水平スケーラビリティ: Spark は、タスクを多数のマシンに分散し、単一ノードのメモリよりも大きなデータセットを処理できます。
フォールト トレランス: Spark は、系統グラフ (RDD 回復メカニズム) と再試行ロジックを使用して、ノードまたはタスクの障害から自動的に回復します。
これら 2 つの機能により、1 台のマシンに制限され、単一ノード エラーで失敗する標準の Python スクリプトとは異なり、Spark は巨大なデータセットを効率的かつ確実に処理できます。
他のオプションが間違っている理由:
B: Spark は一般的なハードウェア上で実行されるため、特殊なマシンは必要ありません。
C: Spark は、ディスクのみの操作ではなく、メモリ内処理を重視します。
D: Spark では、依然として Python、Scala、SQL、または Java のユーザー コードが必要です。
参照：
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark のアーキテクチャとコンポーネント」- 利点、クラスター実行、フォールト トレランス。
Apache Spark の概要 - 分散処理と回復力設計。

最新問題: **56**
Spark エンジニアは、Spark ジョブに適切なデプロイメント モードを選択する必要があります。
Apache Spark™ でクラスターモードを使用する利点は何ですか?
A. クラスタモードでは、クラスタ上のリソースマネージャからリソースが割り当てられ、大規模なジョブのパフォーマンスとスケーラビリティが向上します。
B. クラスター モードでは、ドライバーはすべてのタスクをワーカー ノード全体に分散せずにローカルで実行します。

C. クラスター モードでは、ドライバーはクライアント マシン上で実行されるため、大規模なデータセットを効率的に処理するアプリケーションの能力が制限される可能性があります。
D. クラスター モードでは、ドライバー プログラムはワーカー ノードの 1 つで実行され、アプリケーションはクラスターの分散リソースを最大限に活用できます。
Answer: D (メーッセージを残す)
正確な抜粋からの包括的かつ詳細な説明：
Apache Spark のクラスター モードの場合:
ドライバープログラムは、クライアントのローカルマシンではなく、クラスターのワーカーノードで実行されます。これにより、ドライバーはデータや他のエグゼキューターに近くなり、ネットワークオーバーヘッドが削減され、本番ジョブのフォールトトレランスが向上します。（出典Apache Sparkドキュメント - クラスターモードの概要）このデプロイメントは、ジョブがゲートウェイノードから送信され、Sparkがクラスター自体でドライバーのライフサイクルを管理する本番環境に最適です。
オプション A は部分的に正しいですが、D ほど具体的ではありません。
オプション B は不正解です。ドライバーはすべてのタスクを実行することはありません。エグゼキューターは分散タスクを処理します。
オプション C は、クラスター モードではなく、クライアント モードについて説明します。

最新問題**: 57**
データエンジニアは、上流チームから毎晩配信されるParquetファイル群の取り込みパイプラインを構築するよう依頼されています。データは「/path/events/data」をベースパスとするディレクトリ構造に保存されています。上流チームは、年/月/日の規則に従って、日次データを下位のサブディレクトリにドロップします。
ディレクトリ構造の例をいくつか挙げます。

次のコード スニペットのうち、ディレクトリ構造内のすべてのデータを読み取るものはどれですか。
A. df = spark.read.option("inferSchema", "true").parquet("/path/events/data/")
B. df = spark.read.option("recursiveFileLookup", "true").parquet("/path/events/data/")
C. df = spark.read.parquet("/path/events/data/*")
D. df = spark.read.parquet("/path/events/data/")
Answer: (解答を表示する)
ネストされたディレクトリ構造内のすべてのファイルを再帰的に読み取るには、Spark で recursiveFileLookup オプションを明示的に有効にする必要があります。Databricks の公式ドキュメントによると、ディレクトリツリー内の深くネストされた Parquet ファイル（この例のように）を処理する場合は、次のように設定する必要があります。
df = spark.read.option("recursiveFileLookup", "true").parquet("/path/events/data/") これにより、Spark は /path/events/data/ の下のすべてのサブディレクトリを検索し、フォルダーの深さに関係なく、見つかった Parquet ファイルをすべて読み取るようになります。

オプション A は、オプションが含まれていますが、inferSchema はここでは無関係であり、再帰的なファイル読み取りを有効にしないため、正しくありません。
オプション C は不正解です。ワイルドカードは、1 つのディレクトリ レベルを超える深いネスト構造に確実に一致しない可能性があります。
オプション D は、/path/events/data/ 内のファイルのみを直接読み取り、/2023/01/01 などのサブディレクトリは読み取らないため、正しくありません。
Databricks ドキュメントリファレンス:
「ネストされたフォルダーからファイルを再帰的に読み取るには、recursiveFileLookup オプションを true に設定します。これは、データが階層的なフォルダー構造で編成されている場合に便利です。」- Parquet ファイルの取り込みとオプションに関する Databricks ドキュメント。

最新問題: **58**
開発者は、次のような小さな Parquet テーブルに保存されたデータを使用して Python 辞書を作成する必要があります。

結果の Python 辞書には、最小の 3 つの region_id 値を含む region -> region id のマッピングが含まれている必要があります。
どのコードフラグメントが要件を満たしていますか?
A)

B)

C)

ダ）

結果の Python 辞書には、最小の 3 つの region_id 値に対する region -> region_id のマッピングが含まれている必要があります。
どのコードフラグメントが要件を満たしていますか?

A. 地域 = 辞書(
地域_df
.select('地域', '地域ID')
.sort('地域ID')
.take(3)
）

B. 地域 = 辞書(
地域_df
.select('region_id', 'region')
.sort('地域ID')
.take(3)
）

C. 領域 = dict(
地域_df
.select('region_id', 'region')
.limit(3)
。集める（）
）

D. 地域 = 辞書(
地域_df
.select('地域', '地域ID')
.sort(desc('region_id'))
.take(3)

)
Answer: (解答を表示する**)**
この問題では、キーが地域値、値が対応するregion_idの整数である辞書を作成する必要があります。さらに、region_idの最小値を3つだけ取得するように求められています。
主な観察事項:
.select('region', 'region_id') は、dict() が期待するとおりに列の順序を設定します。最初の列がキーになり、2 番目の列が値になります。
.sort('region_id') は、最も小さい ID が最初に表示されるように昇順で並べ替えます。
.take(3)はちょうど3行を取得します。
結果を dict(...) でラップすると、必要な Python 辞書が正しく構築されます: { 'AFRICA': 0, 'AMERICA': 1, 'ASIA': 2 }。
誤ったオプション:
オプション B は、順序を反転して region_id を最初にし、整数キーを持つ辞書を作成しますが、これは要求されたものではありません。
オプション C はソートなしで .limit(3) を使用するため、パーティション レイアウトに基づいて行が非決定的になります。
オプション D は降順で並べ替え、最小の region_id ではなく最大の region_id を返します。
したがって、オプション A はすべての要件を正確に満たしています。

最新問題**: 59**
55 件中 27 件目。
データ エンジニアは、1 つのテーブルのすべての行を別のテーブルのすべての行に追加する必要がありますが、最初のテーブルのすべての列が 2 番目のテーブルに存在するわけではありません。
エラーメッセージは次のとおりです:
AnalysisException: UNION は同じ数の列を持つテーブルでのみ実行できます。
既存のコードは次のとおりです。
au_df.union(nz_df)
DataFrame au_df には、DataFrame nz_df には存在しない 1 つの列がありますが、それ以外は両方の DataFrame の列名とデータ型は同じです。
結合された DataFrame が期待どおりに生成されるようにするには、データ エンジニアはコードのどこを修正する必要がありますか?
A. df = au_df.unionByName(nz_df, allowMissingColumns=True)
B. df = au_df.unionAll(nz_df)
C. df = au_df.unionByName(nz_df, allowMissingColumns=False)
D. df = au_df.union(nz_df, allowMissingColumns=True)
Answer: A (メーッセージを残す**)**
2 つの DataFrame に異なる列セットがある場合、両方の列が同じ順序でまったく同じでない限り、通常の union() または unionAll() 関数は失敗します。
解決策: allowMissingColumns=True を指定した unionByName() を使用します。
これにより、列が名前で揃えられ、欠落している列が null 値で自動的に追加されます。
正しい構文:
combined_df = au_df.unionByName(nz_df, allowMissingColumns=True)
これにより、1 つの DataFrame に追加の列または欠落した列があっても、結合が機能するようになります。
他のオプションが間違っている理由:
B: unionAll() は非推奨です。同一のスキーマも必要です。
C: allowMissingColumns=False の場合でも、Spark は不一致エラーをスローします。
D: union() は allowMissingColumns 引数を受け入れません。

参照：
PySpark API - allowMissingColumns オプションを指定した DataFrame.unionByName()。
Databricks 試験ガイド (2025 年 6 月) : 「Apache Spark DataFrame/DataSet API アプリケーションの開発」セクション - DataFrame とスキーマの調整を組み合わせます。

最新問題: **60**
55 件中 11 件目。
どの Spark 構成が、エグゼキュータ上で並列実行できるタスクの数を制御しますか?
A. spark.executor.cores
B. spark.task.maxFailures
C. spark.executor.memory
D. spark.sql.shuffle.partitions
Answer: A (メーッセージを残す**)**
Spark 構成 spark.executor.cores は、単一の executor プロセス内で同時に実行できるタスクの数を定義します。
各エグゼキュータには、いくつかの CPU コアが割り当てられます。
各コアは一度に 1 つのタスクを実行します。
したがって、spark.executor.cores を増やすと、エグゼキュータはより多くのタスクを同時に実行できるようになります。
例：
--conf spark.executor.cores=4
→ 各エグゼキュータは 4 つのタスクを並列実行できます。
他のオプションが間違っている理由:
B (spark.task.maxFailures): 失敗したタスクの再試行回数を設定します。
C (spark.executor.memory): 同時実行性ではなく、エグゼキュータのメモリを設定します。
D (spark.sql.shuffle.partitions): 実行者の同時実行性ではなく、シャッフル パーティションの数を定義します。
参照：
Spark 構成ガイド - Executor コア、タスク、および並列処理。
Databricks 試験ガイド (2025 年 6 月) : セクション 「Apache Spark のアーキテクチャとコンポーネント」 - エグゼキュータ構成、CPU コア、および並列タスク実行。

最新問題: **61**
Sparkエンジニアは、実行中にメモリ不足エラーが発生しているSparkアプリケーションのトラブルシューティングを行っています。Sparkドライバーのログを確認すると、「GCオーバーヘッド制限を超えました」というメッセージが複数回表示されていることに気付きました。
この問題を解決するためにエンジニアはどのようなアクションを取る必要がありますか?
A. DataFrame を再パーティション化してデータ処理ロジックを最適化します。
B. Spark 設定を変更してガベージコレクションを無効にします
C. Spark ドライバーに割り当てられたメモリを増やします。
D. 大きな DataFrame をキャッシュしてメモリ内に保持します。
Answer: (解答を表示する**)**

GCオーバーヘッド制限を超えました」というメッセージは通常、JVMがガベージコレクションに多くの時間を費やし、メモリ回復がほとんど行われていないことを示しています。これは、ドライバまたはエグゼキュータのメモリが不足していることを示唆しています。
最も効果的な解決策は、次の方法でドライバーのメモリを増やすことです。
--ドライバーメモリ 4g
これは Spark の公式トラブルシューティング ドキュメントでも確認されています。
「ドライバー ログに GC オーバーヘッド制限超過エラーが多数記録されている場合は、ドライバーのメモリが不足していることを示しています。」
- スパークチューニングガイド
「ドライバー ログに GC オーバーヘッド制限超過エラーが多数記録されている場合は、ドライバーのメモリが不足していることを示しています。」
- スパークチューニングガイド
他の人が間違っている理由:
A は役立つかもしれませんが、ドライバーのメモリ不足を直接解決するものではありません。
B は有効なアクションではありません。GC を無効にすることはできません。
D はメモリ使用量を増加させ、問題を悪化させます。

最新問題: **62**
あるデータサイエンティストは、大量の構造化データの処理、SQLクエリの実行、機械学習アルゴリズムの適用を必要とするプロジェクトに取り組んでいます。このタスクにApache Sparkの使用を検討しています。
このシナリオでは、データ サイエンティストはどの Apache Spark モジュールの組み合わせを使用する必要がありますか?
オプション:
A. Spark DataFrames、構造化ストリーミング、GraphX
B. Spark SQL、Spark 上の Pandas API、構造化ストリーミング
C. Spark ストリーミング、GraphX、および Spark 上の Pandas API
D. Spark DataFrames、Spark SQL、MLlib
Answer: D (メーッセージを残す**)**
包括的な
Apache Spark で構造化データ処理、SQL クエリ、機械学習をカバーするには、コンポーネントの正しい組み合わせは次のとおりです。
Spark DataFrames: 構造化データ処理用
Spark SQL: 構造化データに対してSQLクエリを実行する
MLlib: Sparkのスケーラブルな機械学習ライブラリ
このトリオはまさにこのタイプの使用ケース向けに設計されています。
他のオプションが間違っている理由:
A: GraphX はグラフ処理用であり、ここでは必要ありません。

B: Spark 上の Pandas API は便利ですが、ML には MLlib が不可欠であり、このオプションでは省略されます。
C: Spark Streaming はレガシーです。GraphX はここでは無関係です。

最新問題**: 63**
データサイエンティストが大規模なデータセットを分析しており、DataFrame に対する複数の変換とアクションを含む PySpark スクリプトを作成しました。スクリプトは、結果を取得するための collect() アクションで終了します。
データ サイエンティストがこのスクリプトを実行すると、Apache Spark™ の実行階層は操作をどのように処理しますか?
A. スクリプトは最初に複数のアプリケーションに分割され、次に各アプリケーションはジョブ、ステージ、最後にタスクに分割されます。
B. スクリプト全体が 1 つのジョブとして扱われ、複数のステージに分割され、各ステージはさらにデータ パーティションに基づいてタスクに分割されます。
C. collect() アクションはジョブをトリガーします。ジョブはシャッフル境界でステージに分割され、各ステージは個々のデータ パーティションで動作するタスクに分割されます。
D. Spark はスクリプト内の各変換とアクションに対して 1 つのタスクを作成し、これらのタスクは依存関係に基づいてステージとジョブにグループ化されます。
Answer: (解答を表示する**)**
正確な抜粋からの包括的かつ詳細な説明：
Apache Spark では、実行階層は次のように構造化されています。
アプリケーション: Spark 上に構築されたユーザー プログラムを表す最上位の単位。
ジョブ: アクション（例collect()、count()）によってトリガーされます。各アクションはジョブに対応します。
ステージ: ジョブはシャッフル境界に基づいてステージに分割されます。各ステージには、並列実行可能なタスクが含まれます。
タスク: データのパーティションに適用される単一の操作を表す、作業の最小単位。
collect()アクションが呼び出されると、Sparkはジョブを開始します。このジョブは、データのシャッフル（つまり、ワイド変換）が必要なポイントでステージに分割されます。各ステージは、クラスターのエグゼキューターに分散され、個々のデータパーティションで実行されるタスクで構成されています。
この階層型実行モデルにより、Spark はタスクを並列化し、リソース使用率を最適化することで、大規模なデータを効率的に処理できます。

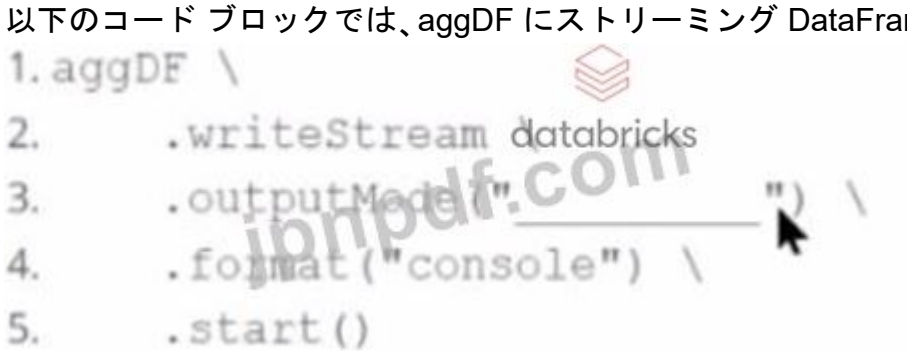
最新問題**: 64**
どの UDF 実装が Spark DataFrame 内の文字列の長さを計算しますか?
A. df.withColumn("長さ", spark.udf("len", StringType()))
B. df.select(length(col("stringColumn")).alias("length"))
C. spark.udf.register("文字列の長さ", lambda s: len(s))
D. df.withColumn("length", udf(lambda s: len(s), StringType()))
Answer: B (メーッセージを残す**)**
正確な抜粋からの包括的かつ詳細な説明：
オプション B では、効率的で Python UDF のオーバーヘッドを回避できる Spark の組み込み SQL 関数 length() を使用します。
pyspark.sql.functionsからlength、colをインポートします
df.select(length(col("stringColumn")).alias("length"))
その他のオプションの説明:
オプション A は構文が正しくありません。spark.udfi はこの方法では呼び出されません。
オプション C は UDF を登録しますが、DataFrame 変換には適用しません。
オプション D は構文的には有効ですが、組み込み関数よりも効率の悪い Python UDF を使用します。

最終回答 :B

最新問題: **65**
データサイエンティストは、ユーザープロファイルテーブル内の一部のレコードに、いずれかのフィールドにnull値が含まれていることを発見しました。これらのレコードは、処理前にデータセットから削除する必要があります。スキーマには、user_id、username、date_of_birth、created_tsなどのフィールドが含まれています。
ユーザー プロファイル テーブルのスキーマは次のようになります。

この要件を満たすために使用できる Spark コード ブロックはどれですか?
オプション:
A. フィルターされたdf = users_raw_df.na.drop(しきい値=0)
B. フィルターされたdf = users_raw_df.na.drop(how='all')
C. フィルターされたdf = users_raw_df.na.drop(how='any')
D. filtered_df = users_raw_df.na.drop(how='all', thresh=None)
Answer: C (メーッセージを残す)
na.drop(how='any') は、少なくとも 1 つの null 値を持つ行を削除します。
これは、完全に完全な記録のみを保持することが目的である場合にまさに必要なことです。
使用法:コピー編集
フィルターされたdf = users_raw_df.na.drop(how='any')
誤ったオプションの説明:
A: thresh=0 は無効です。thresh は # 1 にする必要があります。
B: how='all' は、すべての列が null である行のみを削除します (緩すぎます)。
D: spark.na.drop は、how と thresh をそのように混在させることをサポートしていません。これは間違った構文です。
参考:PySpark DataFrameNaFunctions.drop()

最新問題: **66**
以下のコード ブロックでは、aggDF にストリーミング DataFrame の集計が含まれています。



3 行目のどの出力モードにより、トリガーの実行ごとに結果テーブル全体がコンソールに書き込まれるようになりますか?
A. 完了
B. 追加
C. 置換
D. 集約
Answer: (解答を表示する**)**
各トリガーで完全な更新結果を出力する必要があるストリーミング集計の正しい出力モードは「complete」です。
公式ドキュメントより:
「完全: トリガーが発生するたびに、更新された結果テーブル全体がシンクに出力されます。」これは、キーでグループ化されたカウントや平均など、結果テーブルが時間の経過とともに増分的に変化する集計に最適です。
append: 新しく追加された行のみを出力します
置換と集計: 出力モードに無効な値

最新問題**: 67**
開発者は以下を実行します:

```
df.write.partitionBy("color", "fruit").parquet("/path/to/output")
```

結果はどうなりましたか？
オプション:
A. すべてのデータを 1 つの Parquet ファイルに保存します。
B. いずれかのパーティション列に null 値がある場合はエラーがスローされます。
C. 既存の Parquet ファイルに新しいパーティションを追加します。
D. 色と果物の固有の組み合わせごとに個別のディレクトリを作成します。
Answer: D (メーッセージを残す**)**
Spark の partitionBy() メソッドは、指定された列の一意の組み合わせに基づいて出力をサブディレクトリに整理します。
例えば
/path/to/output/color=red/fruit=apple/part-0000.parquet
/path/to/output/color=green/fruit=banana/part-0001.parquet
これにより、パーティション プルーニングによってクエリ パフォーマンスが向上します。
1 つのファイルに統合されません。
パーティションでは NULL 値が許可されます。
.mode("append") が使用されない限り、「追加」は行われません。
参考:パーティショニングを使用した Spark Write

最新問題**: 68**
データ エンジニアは、毎日最新のデータで上書きされる Parquet テーブルを作成するように依頼されました。
この Parquet テーブルのダウンストリーム コンシューマーには、このテーブル内のデータがすべてのレコードが market_time フィールドでソートされた状態で生成されるという厳格な要件があります。
これらの要件を満たす Parquet テーブルを生成するのは、Spark コードのどの行ですか?

A. 最終df \
.sort("market_time") \
。書く \
.format("寄木細工") \
.mode("上書き") \
.saveAsTable("output.market_events")
B. ファイナル_df \
.orderBy("market_time") \
。書く \
.format("寄木細工") \
.mode("上書き") \
.saveAsTable("output.market_events")
C. ファイナル_df \
.sort("market_time") \
.coalesce(1) \
。書く \
.format("寄木細工") \
.mode("上書き") \
.saveAsTable("output.market_events")
D. 最終_df \
.sortWithinPartitions("market_time") \
。書く \
.format("寄木細工") \
.mode("上書き") \
.saveAsTable("output.market_events")
Answer: D (メーッセージを残す**)**
正確な抜粋からの包括的かつ詳細な説明：
ディスクに書き出されるデータがソートされていることを確認するには、SparkがParquetテーブルに保存する際にデータを書き込む方法を考慮することが重要です。.sort()または.orderBy()メソッドはグローバルソートを適用しますが、特定の条件（例えば、スケーラブルではない.coalesce(1)による単一パーティションなど）が満たされない限り、最終出力ファイルでソートが維持されることは保証されません。
代わりに、分散 Spark 処理で行が書き出されるときにそれぞれのパーティション内でソートされるようにするための適切な方法は次のとおりです。
sortWithinPartitions("列名")
Apache Spark のドキュメントによると:
「sortWithinPartitions() は、各パーティションが指定された列でソートされていることを保証します。これは、ソートされたファイルを必要とする下流システムに役立ちます。」このメソッドは分散設定で効率的に動作し、グローバルソート（.orderBy() や.sort() など）のパフォーマンスボトルネックを回避し、各出力パーティションにソートされたレコードが含まれることを保証します。これにより、一貫してソートされたデータという要件が満たされます。
したがって：
オプション A と B では、保存されたファイルの内容がソートされることは保証されません。
オプション C は、.coalesce(1) (単一パーティション) によってボトルネックが発生します。
オプション D はパーティション内でソートを正しく適用し、スケーラブルです。

参考: Databricks & Apache Spark 3.5 ドキュメント # DataFrame API # sortWithinPartitions()

最新問題: **69**
DataFrame を書き込むときに既存の JSON ファイルを上書きするコマンドはどれですか?
A. df.write.mode("overwrite").json("ファイルへのパス")
B. df.write.overwrite.json("ファイルへのパス")
C. df.write.json("ファイルへのパス", overwrite=True)
D. df.write.format("json").save("ファイルへのパス", モード="上書き")
Answer: A (メーッセージを残す)
DataFrameWriter を使用して既存のファイルを上書きする正しい方法は次のとおりです。
df.write.mode("上書き").json("ファイルへのパス")
オプション D も技術的には有効ですが、オプション A は最も簡潔で慣用的な PySpark 構文です。

最新問題: **70**
Spark開発者がタスクパフォーマンスを監視するアプリを構築しています。ワーカーノードごとの最大タスク処理時間を追跡し、ドライバーに統合して分析する必要があります。
どのテクニックを使うべきでしょうか?
A. reduce()のようなRDDアクションを使用して最大時間を計算します
B. アキュムレータを使用してドライバーの最大時間を記録します
C. ワーカー間で最大時間を共有するための変数をブロードキャストする
D. Spark UI が最大回数を自動的に収集するように設定する
Answer: (解答を表示する)
正確な抜粋からの包括的かつ詳細な説明：
分散ワーカーからドライバーに情報 (最大値など) を集約する正しい方法は、reduce() やaggregate() などの RDD アクションを使用することです。
ドキュメントより:
「分散データに対してグローバル集計を実行するには、min/max/avg などの集計を収集するために、reduce() などのアクションが一般的に使用されます。」アキュムレータ (オプション B) は最大演算を直接サポートしておらず、このような分析には適していません。
ブロードキャスト (オプション C) は、ワーカーからデータを収集するのではなく、ワーカーに送信するために使用されます。
Spark UI (オプション D) は監視ツールであり、分析収集インターフェースではありません。
最終回答：A

最新問題: **71**
55 件中 14 件目。
開発者は、color、fruit、taste の列を持つ DataFrame を作成し、次を使用してデータを Parquet ディレクトリに書き込みました。
df.write.partitionBy("color", "taste").parquet("/path/to/output")
このコードの結果は何でしょうか?
A. 既存の Parquet ファイルに新しいパーティションを追加します。
B. いずれかのパーティション列に null 値がある場合はエラーがスローされます。
C. 色と味の固有の組み合わせごとに個別のディレクトリを作成します。
D. すべてのデータを 1 つの Parquet ファイルに保存します。

Answer: C (メーッセージを残す)
Spark で .partitionBy() を使用して DataFrame を書き込む場合、データはパーティション列の一意の組み合わせに対応するディレクトリ構造に物理的に編成されます。
例：
/path/to/output/color=Red/taste=Sweet/part-0001.parquet
/path/to/output/color=Green/taste=Sour/part-0002.parquet
この構造により、これらの列でフィルタリングするときにパーティションが削減され、クエリのパフォーマンスが向上します。
他のオプションが間違っている理由:
A: 追加には .mode("append") が必要ですが、ここでは使用されません。
B: パーティション列の NULL 値は処理され、エラーは発生しません。
D: パーティション化により、すべてのデータが 1 つのファイルに保存されなくなります。
参照：
PySpark DataFrameWriter API - partitionBy() および .parquet() メソッド。
Databricks 試験ガイド (2025 年 6 月) : セクション 「Spark SQL の使用」- 最適化された出力ファイルのパーティション分割と書き込み。

最新問題**: 72**
my_spark_app.py に次のコード スニペットがあるとします。

```
from pyspark.sql
import SparkSession

spark = SparkSession.builder.appName("CoreComponentsExample").getOrCreate()

data = [("Alice", 34), ("Bob", 36), ("Cathy", 31)]
columns = ["Name", "Age"]

df = spark.createDataFrame(data, columns).withColumn("Status","Pass")
df_filtered = df.filter(df.Age > 35)
df_filtered.show()

spark.stop()
```


ドライバーノードの役割は何ですか?
A. ドライバーノードは、アクションをタスクに変換し、ワーカーノードに配布することで実行を調整します。
B. ドライバーノードは、アプリケーションを監視するためのユーザーインターフェースのみを提供します。
C. ドライバーノードはDataFrameデータを保持し、すべての計算をローカルで実行します。
D. ドライバーノードは、ワーカーノードによる計算が完了した後の最終結果を保存します。
Answer: (解答を表示する)
正確な抜粋からの包括的かつ詳細な説明：
Spark アーキテクチャでは、ドライバー ノードが Spark アプリケーションの実行を調整する役割を担います。

ユーザー定義の変換とアクションを論理プランに変換し、それを物理プランに最適化してから、プランをエグゼキュータ ノードに配布されるタスクに分割します。
Databricks および Spark のドキュメントによると:
「ドライバーノードは、Sparkアプリケーションに関する情報を維持し、ユーザーのプログラムや入力に応答し、エグゼキューター間で作業を分析、分散、スケジュールする役割を担います。」これは次のことを意味します。
ドライバーがジョブ実行をスケジュールおよび調整するため、オプション A が正解です。
オプション B は、ドライバーが UI 監視以上の機能を実行するため、正しくありません。
オプション C は、データと計算がエグゼキュータ ノード間に分散されるため、正しくありません。
オプション D は正しくありません。結果はドライバーに返されますが、ドライバーによって長期的に保存されることはありません。
参考: Databricks 認定開発者 Spark 3.5 ドキュメント # Spark アーキテクチャ # ドライバーとエグゼキューター。

最新問題**: 73**
Spark エンジニアは、Spark ジョブに適切なデプロイメント モードを選択する必要があります。
Apache Spark™ でクラスターモードを使用する利点は何ですか?
A. クラスタモードでは、クラスタ上のリソースマネージャからリソースが割り当てられ、大規模なジョブのパフォーマンスとスケーラビリティが向上します。
B. クラスター モードでは、ドライバーはすべてのタスクをワーカー ノード全体に分散せずにローカルで実行します。
C. クラスター モードでは、ドライバーはクライアント マシン上で実行されるため、大規模なデータセットを効率的に処理するアプリケーションの能力が制限される可能性があります。
D. クラスター モードでは、ドライバー プログラムはワーカー ノードの 1 つで実行され、アプリケーションはクラスターの分散リソースを最大限に活用できます。
Answer: D (メーッセージを残す**)**
Apache Spark のクラスター モードの場合:
ドライバープログラムは、クライアントのローカルマシンではなく、クラスターのワーカーノードで実行されます。これにより、ドライバーはデータや他のエグゼキューターに近接して配置され、ネットワークオーバーヘッドが削減され、本番ジョブのフォールトトレランスが向上します。（出典Apache Spark ドキュメント - クラスターモードの概要）
ドライバープログラムは、クライアントのローカルマシンではなく、クラスターのワーカーノードで実行されます。これにより、ドライバーはデータや他のエグゼキューターに近くなるため、ネットワークオーバーヘッドが削減され、本番ジョブのフォールトトレランスが向上します。（出典Apache Spark ドキュメント - クラスターモードの概要）このデプロイメントは、ジョブがゲートウェイノードから送信され、Sparkがクラスター自体でドライバーのライフサイクルを管理する本番環境に最適です。
オプション A は部分的に正しいですが、D ほど具体的ではありません。
オプション B は不正解です。ドライバーはすべてのタスクを実行することはありません。エグゼキューターは分散タスクを処理します。
オプション C は、クラスター モードではなく、クライアント モードについて説明します。