Databricks.Associate-Developer-Apache-Spark.v2023-12-11.q63

試験コード:	Associate-Developer-Apache-Spark
試験名称:	Databricks Certified Associate Developer for Apache Spark 3.0 Exam
認定資格:	Databricks
無料問題数:	63
バージョン:	v2023-12-11
アクセス数:	695
ページビュー数:	630
https://www.jpnpdf.com/Databricks.Associate-Developer-Apache-Spark.v2023-12-11.q63-mondaishu.html	

最新問題: **1**
次のコード ブロックのうち、DataFrametransactionsDf の列 storeId の一意の値の数を返すものはどれですか?
A. transactionsDf.select("storeId").dropDuplicates().count()
B. transactionsDf.select(count("storeId")).dropDuplicates()
C. transactionsDf.select(distinct("storeId")).count()
D. transactionsDf.dropDuplicates().agg(count("storeId"))
E. transactionsDf.distinct().select("storeId").count()
Answer: (解答を表示する**)**
説明
transactionsDf.select("storeId").dropDuplicates().count()
正しい！列storeIdからすべての重複を削除した後、残りの行がカウントされ、列内の固有の値の数を表します。
transactionsDf.select(count("storeId")).dropDuplicates()
いいえ、transactionsDf.select(count("storeId")) は、null 以外の行の数を示す単一行の DataFrame を返すだけです。このコンテキストでは、dropDuplicates() は何の効果もありません。
transactionsDf.dropDuplicates().agg(count("storeId"))
正しくない。transactionsDf.dropDuplicates() は、transactionsDf から重複行を削除しますが、列 storeId のみを考慮して削除するのではなく、代わりに行全体の重複を削除します。
transactionsDf.distinct().select("storeId").count()
間違い。transactionsDf.distinct() は、すべての列にわたって一意の行を識別しますが、列 storeId に関して一意の行だけを識別するわけではありません。これにより、列内に重複した値が残り、カウントがその列内の一意の値の数を表さなくなる可能性があります。

transactionsDf.select(distinct("storeId")).count()
間違い。pyspark.sql.functions には個別のメソッドはありません。

最新問題: **2**
以下に表示されているコード ブロックにはエラーが含まれています。コード ブロックは、列 predError の値が少なくとも 5 である DataFrametransactionsDf の行のみを含む新しい DataFrame を返す必要があります。エラーを見つけます。
コードブロック:
transactionsDf.where("col(predError) >= 5")
A. where メソッドの引数は「predError >= 5」である必要があります。
B. where() の代わりに、filter() を使用する必要があります。
C. この式は、新しい DataFrame ではなく、元の DataFrame のtransactionDf を返します。これを回避するには、コード ブロックを transactionsDf.toNewDataFrame().where("col(predError) >= 5")にする必要があります。
D. where メソッドの引数に文字列を指定することはできません。
E. >= の代わりに、SQL 演算子 GEQ を使用する必要があります。
Answer: A (メーッセージを残す**)**
説明
where メソッドの引数には文字列を指定できません。
文字列でも問題ありません。
where() の代わりに、filter() を使用する必要があります。
いいえ、それは問題ではありません。PySpark では、where() と filter() は同等です。
>= の代わりに、SQL 演算子 GEQ を使用する必要があります。
正しくない。
この式は、新しい DataFrame ではなく、元の DataFrame のtransactionDf を返します。これを回避するには、コード ブロックを transactionsDf.toNewDataFrame().where("col(predError) >= 5")にする必要があります。
いいえ、Spark は新しい DataFrame を返します。
静的ノート | 動的ノートブック: テスト 1 を参照
(https://flrs.github.io/spark_practice_tests_code/#1/27.html 、
https://bit.ly/sparkpracticeexams_import_instructions)

最新問題: **3**
次のコード ブロックのうち、小さな DataFrame トランザクション Df がすべてのエグゼキューターに送信され、それぞれ列 storeId および itemId 上の DataFrame itemsDf と結合される結合を実行するのはどれですか?
A. itemsDf.join(transactionsDf, itemsDf.itemId ==transactionsDf.storeId, "right_outer")
B. itemsDf.join(transactionsDf, itemsDf.itemId ==transactionsDf.storeId, "ブロードキャスト")

C. itemsDf.merge(transactionsDf, "itemsDf.itemId ==transactionsDf.storeId", "ブロードキャスト")
D. itemsDf.join(broadcast(transactionsDf), itemsDf.itemId ==transactionsDf.storeId)
E. itemsDf.join(transactionsDf, ブロードキャスト(itemsDf.itemId ==transactionsDf.storeId))
Answer: (解答を表示する**)**
説明
最後の引数として「ブロードキャスト」を含むすべての回答の問題は、「ブロードキャスト」が有効な結合タイプではないことです。「right_outer」を含むエントリは有効なステートメントですが、ブロードキャスト結合ではありません。等価条件の周りにbroadcast()がラップされている項目は、Sparkでは有効なコードではありません。Broadcast() は、ブロードキャストされる小さな DataFrame の名前をラップする必要があります。
詳細情報: Learning Spark、第 2 版、第 7 章
静的ノート | 動的ノートブック: テスト 1 を参照
と説明は？

最新問題**: 4**
以下に示すコード ブロックは、filePath の場所に保存されている CSV ファイル内の列の数を返す必要があります。
CSV ファイルからは、# 文字で始まらない行のみを読み取る必要があります。これを達成するには、コード ブロック内の空白を正しく埋める答えを選択してください。
コードブロック:
__1__(__2__.__3__.csv(ファイルパス, __4__).__5__)
A. 1. サイズ
2.スパーク
3.read()
4.エスケープ='#'
5. コラム
B. 1. データフレーム
2.スパーク
3.read()
4.エスケープ='#'
5. 形状[0]
C. 1.レン
2.パイスパーク
3. データフレームリーダー
4.コメント='#'
5. コラム
D. 1. サイズ
2.パイスパーク

3. データフレームリーダー
4.コメント='#'
5. コラム
E. 1.レン
2.スパーク
3.読む
4.コメント='#'
5. コラム
Answer: (解答を表示する**)**
説明
正しいコードブロック:
len(spark.read.csv(filePath, comment='#').columns)
これは、DataFrame と DataFrameReader の間の境界という、通常とは異なる状況における困難を伴う難しい質問です。この難易度の問題が試験に出題されることはまずありません。ただし、これを解決すると、試験で扱われる可能性が高い DataFrameReader にもっと慣れることができます。
内側の括弧を処理する前に、外側の括弧であるギャップ 1 と 5 を理解する方が簡単です。コード ブロックを考えると、ギャップ 5 のオブジェクトはギャップ 1 のオブジェクトによって評価され、ギャップ 1 の列数を返す必要があります。読み込んだCSV。1 つの回答オプションには、ギャップ 1 に DataFrame が含まれ、ギャップ 2 にshape[0] が含まれています。DataFrame はshape[0] の評価に使用できないため、この回答オプションは破棄できます。
他の回答オプションにはギャップ 1 のサイズが含まれます。size() は組み込みの Python コマンドではないため、使用する場合は他の場所から取得する必要があります。pyspark.sql.functions には size() メソッドが含まれていますが、このメソッドは列内に格納されている配列またはマップの長さを返すだけです (以下にリンクされているドキュメント)。
したがって、ここでは size() メソッドを使用することはできません。これにより、有効な可能性のある答えが 2 つ残ります。
ギャップ 2 と 3 が、spark.read か pyspark.DataFrameReader のどちらかを選択する必要があります。ドキュメント (以下にリンク) を見ると、DataFrameReader は実際には pyspark.sql の子クラスであり、pyspark.DataFrameReader を使用してインポートできないことを意味します。さらに、Databricks では、spark は現在の Spark セッション (pyspark.sql.SparkSession) を参照し、spark.read が DataFrameReader を返すため、spark.read は理にかなっています (以下のドキュメントも参照)。最後に、正解の選択肢は 1 つだけ残ります。
より詳しい情報：
- pyspark.sql.functions.size - PySpark 3.1.2 ドキュメント
- pyspark.sql.DataFrameReader.csv - PySpark 3.1.2 ドキュメント
- pyspark.sql.SparkSession.read - PySpark 3.1.2 ドキュメント

静的ノート | 動的ノートブック: テスト 3 を参照

最新問題: **5**
データセット API の特徴を説明しているものは次のうちどれですか?
A. データセット API は非構造化データをサポートしません。
B. Python では、Dataset API は主に Pandas の DataFrame API に似ています。
C. Python では、データセット API のスキーマはタイプ ヒントを介して構築されます。
D. データセット API は Scala では使用できますが、Python では使用できません。
E. データセット API はコンパイル時の型安全性を提供しません。
Answer: D (メーッセージを残す)
説明
Dataset API は Scala では利用できますが、Python では利用できません。
正しい。データセット API は固定型を使用し、通常はオブジェクト指向プログラミングに使用されます。これは、Spark が Scala プログラミング言語で使用されている場合に使用できますが、Python では使用できません。Python では、Dataset API に基づいた DataFrame API を使用します。
Dataset API はコンパイル時の型安全性を提供しません。
いいえ、実際には、ユースケースによっては、Dataset API が提供するタイプ セーフティが利点となります。
データセット API は非構造化データをサポートしません。
違います。Dataset API は構造化データと非構造化データをサポートしています。
Python では、データセット API のスキーマはタイプ ヒントを介して構築されます。
いいえ、データセット API は Python では使用できないため、これは適用されません。
Python では、Dataset API は主に Pandas の DataFrame API に似ています。
Dataset API は Python には存在せず、Scala と Java にのみ存在します。

最新問題: **6**
クラスターマネージャーの特徴は次のうちどれですか?
A. 各クラスター マネージャーはデータの単一パーティションで動作します。
B. クラスター マネージャーは、SparkContext を通じてドライバーから入力を受け取ります。
C. スタンドアロンモードではクラスタマネージャが存在しません。
D. クラスター マネージャーはジョブを DAG に変換します。
E. クライアント モードでは、クラスター マネージャーはエッジ ノード上で実行されます。
Answer: (解答を表示する)
説明
クラスター マネージャーは、SparkContext を通じてドライバーから入力を受け取ります。

正しい。ドライバーがクラスター マネージャーに接続するために、ドライバーは SparkContext を起動します。次に、ドライバーはクラスター マネージャーにエグゼキューターを起動するためのリソースを要求します。
クライアント モードでは、クラスター マネージャーはエッジ ノード上で実行されます。
いいえ。クライアント モードでは、クラスター マネージャーはエッジ ノードから独立しており、クラスター内で実行されます。
クラスター マネージャーはスタンドアロン モードには存在しません。
違います。クラスター マネージャーはスタンドアロン モードでも存在します。スタンドアロン モードはクラスター全体に Spark をデプロイする簡単な手段ですが、いくつかの制限があることに注意してください。たとえば、スタンドアロン モードでは、他のフレームワークは Spark と並行して実行できません。ただし、クラスター マネージャーはスタンドアロン デプロイメントでは Spark の一部であり、クラスター全体でリソースを起動および維持するのに役立ちます。
クラスター マネージャーはジョブを DAG に変換します。
いいえ、ジョブを DAG に変換するのは Spark ドライバーのタスクです。
各クラスター マネージャーは、データの単一パーティションで動作します。
いいえ、クラスターマネージャーはパーティションを直接操作しません。彼らの仕事は、クラスター リソースを調整して、クラスター リソースが Spark ドライバーによって要求され、Spark ドライバーに割り当てられるようにすることです。
詳細情報: コア Spark コンセプトの紹介 * BigData

最新問題**: 7**
次のコード ブロックのうち、スキーマ fileSchema を使用して、filePath の場所にある寄木細工のファイルを DataFrame に読み取るのはどれですか?
A. spark.read.schema(fileSchema).format("parquet").load(filePath)
B. spark.read.schema("fileSchema").format("parquet").load(filePath)
C. spark.read().schema(fileSchema).parquet(filePath)
D. spark.read().schema(fileSchema).format(parquet).load(filePath)
E. spark.read.schema(fileSchema).open(filePath)
Answer: (解答を表示する**)**
説明
ここで、どの変数が引用されているかに注意してください。fileSchema は変数であるため、引用符で囲むことはできません。
parquet は変数ではないため、引用符で囲む必要があります。
SparkSession.read (ここでは、spark.read として参照) は、後続のすべての呼び出しで参照される DataFrameReader を返します。DataFrameReader は呼び出し可能ではないため、ここでは括弧を使用しないでください。
最後に、PySpark にはオープン メソッドがありません。メソッド名はloadです。
静的ノート | 動的ノートブック: テスト 1 を参照

最新問題: **8**
次のコード ブロックは、DataFrametransactionsDf のすべての列
と、DataFrametransactionsDf の列 predError の 2 乗値である追加の列 predErrorSquared
を含む DataFrame を返します。
A. transactionsDf.withColumn("predError", pow(col("predErrorSquared"), 2))
B. transactionsDf.withColumnRenamed("predErrorSquared", pow(predError, 2))
C. transactionsDf.withColumn("predErrorSquared", pow(col("predError"), lit(2)))
D. transactionsDf.withColumn("predErrorSquared", pow(predError, lit(2)))
E. transactionsDf.withColumn("predErrorSquared", "predError"**2)
Answer: C (メーッセージを残す**)**
説明
これらのコード ブロックのうち機能するのは 1 つだけですが、DataFrame API は pow()
メソッドに列を受け入れる点で非常に柔軟です。次のコード ブロックも機能します。
transactionsDf.withColumn("predErrorSquared", pow("predError", 2))
transactionsDf.withColumn("predErrorSquared", pow("predError", lit(2))) 静的ノートブック
| 動的ノートブック: テスト 1 (https://flrs.github.io/spark_practice_tests_code/#1/26.html)
を参照してください。
https://bit.ly/sparkpracticeexams_import_instructions)

最新問題: **9**
アダプティブ クエリ実行の機能ではないものは次のうちどれですか?
A. 必要に応じて、ソート ･マージ結合をブロードキャスト結合に置き換えます。
B. パーティションを結合してデータ処理を高速化します。
C. パーティションの処理時間の差を避けるために、偏ったパーティションをより小さな
パーティションに分割します。
D. エグゼキュータに障害が発生した場合にクエリを再ルーティングします。
E. クエリの実行中にランタイム統計を収集します。
Answer: (解答を表示する**)**
説明
エグゼキュータに障害が発生した場合にクエリを再ルーティングします。
正しい。この機能は Spark に存在しますが、Adaptive Query Execution の機能ではありま
せん。クラスター マネージャーはエグゼキューターを追跡し、ドライバーと連携してエグ
ゼキューターを起動し、失敗したエグゼキューターのワークロードをそれに割り当てます
(以下のリンクも参照)。
必要に応じて、ソート ･マージ結合をブロードキャスト結合に置き換えます。
いいえ、これは適応クエリ実行の機能です。
パーティションを結合してデータ処理を高速化します。
違います。Adaptive Query Execution がこれを実行します。
クエリの実行中にランタイム統計を収集します。

不正解です。Adaptive Query Execution (AQE) は、クエリ プランを調整するためにこれらの統計を収集します。このフィードバック ループは、AQE を介してクエリを高速化するために不可欠な部分です。
パーティションの処理時間の差を避けるために、偏ったパーティションをより小さなパーティションに分割します。
いいえ、これは実際に Adaptive Query Execution の機能です。詳細については、以下のリンク先の Databricks ブログ投稿をご覧ください。
詳細情報: Learning Spark、第 2 版、第 12 章、Spark の RDD はどの方法でフォールト トレランスを完了しますか?
- スタック オーバーフロー、アダプティブ クエリ実行で SQL クエリを高速化する方法

最新問題**: 10**
ProductId 列と itemId 列の DataFrame のtransactionDf と itemsDf の内部結合の結果、DataFrame の列値が空ではないレコードの数を返すには、以下に示すコード ブロックをどの順序で実行する必要がありますか?
1. .filter(~isnull(col('value')))
2. .count()
3.transactionsDf.join(itemsDf,col("transactionsDf.productId")==col("itemsDf.itemId"))
4.transactionsDf.join(itemsDf,transactionsDf.productId==itemsDf.itemId,how='inner')
5. .filter(col('value').isnotnull())
6. .sum(col('value'))
A. 4、1、2
B. 3、1、6
C. 3、1、2
D. 3、5、2
E. 4、6
Answer: A (メーッセージを残す**)**
説明
正しいコードブロック:
transactionsDf.join(itemsDf,transactionsDf.productId==itemsDf.itemId,
how='inner').filter(~isnull(col('value'))).count()
式col("transactionsDf.productId")およびcol("itemsDf.itemId")は無効です。Col() は DataFrame の名前を受け入れず、列名のみを受け入れます。
静的ノート | 動的ノートブック: テスト 2 を参照

最新問題**: 11**
次のコード ブロックのうち、列名と型の両方を含むデータフレームの構造をツリー状に示しているものはどれですか?
A. 1.print(itemsDf.columns)
2.print(itemsDf.types)

B. itemsDf.printSchema()
C. spark.schema(itemsDf)
D. itemsDf.rdd.printSchema()
E. itemsDf.print.schema()
Answer: (解答を表示する**)**
説明
itemsDf.printSchema()
正しい！itemsDf.printSchema() が示す例を次に示します。列名と型の両方を含むツリー状の構造がわかります。
根
|-- itemId: 整数 (null 許容 = true)
|-- 属性: 配列 (null 許容 = true)
| |-- 要素: 文字列 (Null = true を含む)
|-- サプライヤー: 文字列 (null 可能 = true)
itemsDf.rdd.printSchema()
いいえ、DataFrame の基礎となる RDD には printSchema() メソッドがありません。
スパーク.スキーマ(itemsDf)
不正解です。spark.schema コマンドはありません。
print(itemsDf.columns)
print(itemsDf.dtypes)
間違い。このコード ブロックの出力には列名と列タイプの両方が含まれていますが、情報はツリー状に配置されていません。
itemsDf.print.schema()
いいえ、DataFrame には print メソッドがありません。
静的ノート | 動的ノートブック: テスト 3 を参照

最新問題**: 12**
次のコード ブロックは、CSV ファイル ヘッダーで定義された列名を使用して、ディレクトリ filePath 内のすべての CSV ファイルを単一のデータフレームに読み取りますか?
ディレクトリファイルパスの内容:
1._成功
2._committed_2754546451699747124
3._開始_2754546451699747124
4.part-00000-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-298-1-c000.csv.gz
5.part-00001-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-299-1-c000.csv.gz
6.part-00002-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-300-1-c000.csv.gz

7.part-00003-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-301-1-c000.csv.gz spar.option("header",True).csv(filePath)

A.
spark.read.format("csv").option("header",True).option("compression","zip").load(filePath)

B. spark.read().option("header",True).load(filePath)

C. spark.read.format("csv").option("header",True).load(filePath)

D. spark.read.load(ファイルパス)

Answer: (解答を表示する**)**

説明

ディレクトリ filePath 内のファイルは、gzip 圧縮を使用してエクスポートされた DataFrame のパーティションです。

Spark はこの状況を自動的に認識し、CSV ファイルを個別のパーティションとして単一の DataFrame にインポートします。ただし、Spark が header オプションを使用して CSV 内のファイル ヘッダーをロードするように指定する必要があります。このオプションはデフォルトで False に設定されています。

最新問題**: 13**

以下に表示されているコード ブロックにはエラーが含まれています。コード ブロックは、2 つの列の情報を使用して DataFrame トランザクション Df の行を順序どおりに配置する必要があります。最初に列の値ごとに配置し、上部に小さい数値を表示し、下部に大きい数値を表示します。次に列 predError ごとに配置し、すべての値を表示する必要があります。列値の項目の順序と逆の方法で配置されます。エラーを見つけてください。

コードブロック:

transactionsDf.orderBy('value', asc_nulls_first(col('predError')))

A. 1 つの orderBy ステートメントに両方の列を含めるのではなく、個々の列を呼び出す 2 つの orderBy ステートメントをチェーンする必要があります。

B. 列値は、col() 演算子でラップする必要があります。

C. 列 predError は降順でソートされ、null が最後に配置される必要があります。

D. 列 predError は、代わりに desc_nulls_first() によってソートされる必要があります。

E. orderBy の代わりに、sort を使用する必要があります。

Answer: C (メーッセージを残す**)**

説明

正しいコードブロック:

transactionsDf.orderBy('value', desc_nulls_last('predError'))

predError 列は降順でソートし、null を最後に置く必要があります。

正しい！デフォルトでは、Spark は昇順にソートし、null を最初に配置します。したがって、デフォルトのソートの逆ソートは実際には desc_nulls_last です。

orderBy の代わりに、sort を使用する必要があります。

いいえ、DataFrame.sort() はパーティションごとにデータを順序付けしますが、グローバルな順序は保証しません。このため、ここでは orderBy がより適切な演算子となります。

列の値は、col() 演算子でラップする必要があります。
正しくない。DataFrame.sort() は、文字列オブジェクトと列オブジェクトの両方を受け入れます。
列 predError は、代わりに desc_nulls_first() でソートする必要があります。
間違い。Spark のデフォルトのソート順序は asc_nulls_first() と一致するため、反転する場合は null が最後に来る必要があります。
1 つの orderBy ステートメントに両方の列を含めるのではなく、個々の列を呼び出す 2 つの orderBy ステートメントをチェーンする必要があります。
いいえ、これは DataFrame を最後の列で並べ替えるだけであり、質問に記載されているように、両方の列からの情報は考慮されません。
詳細: pyspark.sql.DataFrame.orderBy - PySpark 3.1.2 ドキュメント、pyspark.sql.functions.desc_nulls_last - PySpark 3.1.2 ドキュメント、Spark の sort() と orderBy() | データサイエンスに向けて 静的ノートブック | 動的ノートブック: テスト 3 を参照

最新問題**: 14**
次のコード ブロックは、filePath に格納されている 2 つのパーティションの寄木細工ファイルを読み取り、各パーティションのスキーマが異なる場合でもすべての列が 1 回だけ含まれることを確認しますか?
最初のパーティションのスキーマ:
1.ルート
2. |-- トランザクション ID: 整数 (null 許容 = true)
3. |-- predError: 整数 (null 許容 = true)
4. |-- 値: 整数 (null 許容 = true)
5. |--storeId: 整数 (null 許容 = true)
6. |-- productId: 整数 (null 許容 = true)
7. |-- f: 整数 (null 許容 = true)
2 番目のパーティションのスキーマ:
1.ルート
2. |-- トランザクション ID: 整数 (null 許容 = true)
3. |-- predError: 整数 (null 許容 = true)
4. |-- 値: 整数 (null 許容 = true)
5. |--storeId: 整数 (null 許容 = true)
6. |-- rollId: 整数 (null 許容 = true)
7. |-- f: 整数 (null 許容 = true)
8. |--tax_id: 整数 (null 可能 = false)
A. spark.read.parquet(filePath, mergeSchema='y')
B. spark.read.option("mergeSchema", "true").parquet(filePath)
C. spark.read.parquet(ファイルパス)
D. 1.nx = 0

2.dbutils.fs.ls(filePath) のファイル:
3. file.name.endswith(".parquet") でない場合:
4. 続ける
5. df_temp =spark.read.parquet(file.path)
6. nx == 0の場合:
7. df = df_temp
8. その他:
9. df = df.union(df_temp)
10. nx = nx+1
11.df
E. 1.nx = 0
2.dbutils.fs.ls(filePath) のファイル:
3. file.name.endswith(".parquet") でない場合:
4. 続ける
5. df_temp =spark.read.parquet(file.path)
6. nx == 0の場合:
7. df = df_temp
8. その他:
9. df = df.join(df_temp, how="outer")
10. nx = nx+1
11.df
Answer: (解答を表示する**)**
説明
これは非常に難しい質問であり、マージに関する知識と、寄木細工のファイルを読み取るときのスキーマの両方の知識が必要です。
smile.read.option("mergeSchema", "true").parquet(filePath)
正しい。両方のパーティションに表示される列のデータ型が一致するため、Spark の DataFrameReader の mergeSchema オプションはここでうまく機能します。両方のパーティションに存在する同じ名前の 1 つ以上の列のデータ型が異なる場合、mergeSchema は失敗することに注意してください。
スパーク.read.parquet(ファイルパス)
正しくない。これにより両方のパーティションからデータが読み込まれますが、最初に読み込まれる寄木細工ファイルのスキーマのみが考慮されるため、2 番目のパーティションにのみ表示される一部の列 (tax_id など) は失われます。
nx = 0
dbutils.fs.ls(filePath) のファイルの場合:
file.name.endswith(".parquet") でない場合:
続く
df_temp =spark.read.parquet(file.path)
nx == 0の場合:

df = df_temp
それ以外：
df = df.union(df_temp)
nx = nx+1
DF
間違い。このソリューションの重要なアイデアは DataFrame.union() コマンドです。このコマンドはすべてのデータをマージしますが、両方のパーティションに同じデータ型のまったく同じ数の列が必要です。
smile.read.parquet(filePath, mergeSchema="y")
間違い。mergeSchema オプションを使用することがこの問題を解決する正しい方法であり、コード ブロックのように DataFrameReader.parquet() で呼び出すこともできますが、値 True をブール値または文字列変数として受け入れます。ただし、ȳ」は有効なオプションではありません。
nx = 0
dbutils.fs.ls(filePath) のファイルの場合:
file.name.endswith(".parquet") でない場合:
続く
df_temp =spark.read.parquet(file.path)
nx == 0の場合:
df = df_temp
それ以外：
df = df.join(df_temp, how="アウター")
nx = nx+1
DF
いいえ。これにより完全外部結合が引き起こされます。結果の DataFrame には両方のパーティションのすべての列が含まれますが、両方のパーティションに表示される列は重複します。質問では、パーティションに含まれるすべての列が 1 回だけ表示される必要があると述べています。
詳細: Apache Spark での異なるスキーマのマージ | チアゴ ·コルドン著 | データアリーナ | 中型静的ノートブック | 動的ノートブック: テスト 3 を参照

最新問題**: 15**
メモリ不足エラーの削減に関する次の記述のうち、間違っているものはどれですか?
A. 複数の文字列列を 1 つの列に連結すると、メモリ不足エラーを防ぐことができます。
B. パーティション サイズを小さくすると、メモリ不足エラーを防ぐことができます。
C. 結合で自動的にブロードキャストされるデータの量を制限すると、メモリ不足エラーを防ぐことができます。
D. ドライバーに返されるシリアル化されたデータの最大サイズに制限を設定すると、メモリ不足エラーを防ぐことができる場合があります。

E. 各エグゼキュータで使用できるコアの数を減らすと、メモリ不足エラーを防ぐことができます。
Answer: A (メーッセージを残す**)**
説明
複数の文字列列を 1 つの列に連結すると、メモリ不足エラーを防ぐことができます。
まさに、これは不正解です！文字列列を連結してもデータのサイズは減りません。構造が異なるだけです。これは Spark によるデータの処理方法にはほとんど影響せず、メモリ不足エラーは明らかに減りません。
パーティション サイズを小さくすると、メモリ不足エラーを防ぐことができます。
いいえ、これは間違いではありません。エグゼキューターはパーティションを処理する前にメモリにロードする必要があるため、パーティション サイズを減らすことはメモリ不足エラーを防ぐための有効な方法です。これを行うのに十分なメモリがエグゼキュータにない場合、メモリ不足エラーがスローされます。したがって、パーティション サイズを減らすことは、それを防ぐのに非常に役立ちます。
各エグゼキュータで使用できるコアの数を減らすと、メモリ不足エラーを防ぐことができます。
いいえ、これは間違いではありません。パーティションを処理するには、このパーティションをエグゼキュータのメモリにロードする必要があります。すべてのエグゼキュータのすべてのコアが、他のエグゼキュータと並行してパーティションを処理すると想像すると、エグゼキュータをホストしているマシン上のメモリがすぐにいっぱいになることが想像できます。したがって、特に複数のパーティションが同時に処理される場合、エグゼキュータのメモリ使用量が問題になります。パフォーマンスとメモリ使用量のバランスをとるには、コアの数を減らすとメモリ不足エラーを防ぐことができる場合があります。
ドライバーに返されるシリアル化されたデータの最大サイズに制限を設定すると、メモリ不足エラーの防止に役立つ場合があります。
いいえ、これは間違いではありません。クラスターからドライバーへの潜在的に大量のデータの送信をトリガーするcollect() などのコマンドを使用すると、ドライバーでメモリ不足エラーが発生する可能性があります。これを回避する 1 つの戦略は、ドライバーに大量のデータを送り返すcollect() などのコマンドの使用に注意することです。もう 1 つの戦略は、パラメーター spar.driver.maxResultSize を設定することです。ドライバーに送信されるデータがパラメーターで指定されたしきい値を超えると、Spark はジョブを中止するため、メモリー不足エラーが発生するのを防ぎます。
結合で自動的にブロードキャストされるデータの量を制限すると、メモリ不足エラーの防止に役立ちます。
違います、これは間違いではありません。Spark の内部最適化の一環として、Spark は (通常は比較的小さい) テーブルをエグゼキューターにブロードキャストすることで操作を高速化することを選択する場合があります。このブロードキャストはドライバーから行われるため、最初にすべてのブロードキャスト テーブルがドライバーにロードされます。これらのテーブルが比較的大きい場合、または複数の中サイズのテーブルがブロードキャスト

されている場合、メモリ不足エラーが発生する可能性があります。Spark がブロードキャストを考慮する最大テーブル サイズは、spark.sql.autoBroadcastJoinThreshold パラメーターによって設定されます。
詳細情報: 構成 - Spark 3.1.2 ドキュメントおよび Spark OOM エラー - クローズアップ。次のような場合に見覚えはありますか? アミット ・シン ・ラソール著 | スタートアップ | 中くらい

最新問題: **16**
次のコード ブロックのうち、ブール値を返す Python 関数 EvaluateTestSuccess をユーザー定義関数として DataFrametransactionsDf の列 storeId に適用するものはどれですか?
A. 1.pyspark.sql からタイプを T としてインポートします
2.evaluateTestSuccessUDF = udf(evaluateTestSuccess, T.BooleanType())
3.transactionsDf.withColumn("結果",evaluateTestSuccessUDF(col("storeId")))
B. 1.evaluateTestSuccessUDF = udf(evaluateTestSuccess)
2.transactionsDf.withColumn("結果",evaluateTestSuccessUDF(storeId))
C. 1.pyspark.sql からタイプを T としてインポートします
2.evaluateTestSuccessUDF = udf(evaluateTestSuccess, T.IntegerType())
3.transactionsDf.withColumn("結果",evaluateTestSuccess(col("storeId")))
D. 1.evaluateTestSuccessUDF = udf(evaluateTestSuccess)
2.transactionsDf.withColumn("結果",evaluateTestSuccessUDF(col("storeId")))
E. 1.pyspark.sql からタイプを T としてインポートします
2.evaluateTestSuccessUDF = udf(evaluateTestSuccess, T.BooleanType())
3.transactionsDf.withColumn("結果",evaluateTestSuccess(col("storeId")))
Answer: (解答を表示する**)**
説明
この問題を解決するには、UDF 仕様では戻り値の型 (デフォルトの文字列でない限り) が必要であることを認識することが重要です。さらに、Python 関数 (evaluateTestSuccess) ではなく、生成された UDF (evaluateTestSuccessUDF) が列 storeId に適用されていることを確認する必要があります。
詳細: pyspark.sql.functions.udf - PySpark 3.1.2 ドキュメント
静的ノート | 動的ノートブック: テスト 2 を参照

最新問題: **17**
以下に示すコード ブロックは、列 itemNameElements に少なくとも 3 つの項目を持つ DataFrame itemsDf のすべての行を返す必要があります。これを達成するには、コード ブロック内の空白を正しく埋める答えを選択してください。
DataFrame itemsDf の例:
1.+------+----------+----- ----------+---------- -------
+
2.|itemId|itemName|supplier|itemNameElements|
3.+------+----------+----- ----------+---------- -------
+
4.|1 |雪中ウォーキング用厚手コート|スポーツカンパニー株式会社 |厚手、コート、ウォーキング、イン、ザ、スノー]|
5.|2 |エレガントアウトドアサマーワンピース |YetiX |[エレガント,アウトドア,夏,ワンピース] |
6.|3 |アウトドア バックパック |株式会社スポーツカンパニー |[アウトドア,バックパック] |
7.+-----+----------+----- ----------+---------- -------
+ コードブロック:
itemsDf.__1__(__2__(__3__)__4__)
A. 1. を選択します。
2. 数える
3.col("アイテム名要素")
4.>3
B. 1.フィルター
2. 数える
3. 項目名要素
4. >=3
C. 1.選択
2. 数える
3. 「アイテム名要素」
4.>3
D. 1. フィルター
2.サイズ
3. 「アイテム名要素」
4. >=3
（正しい）
E. 1. を選択します。
2.サイズ

3. 「アイテム名要素」
4.>3
Answer: D (メーッセージを残す)
説明
正しいコードブロック:
itemsDf.filter(size("itemNameElements")>3)
コードブロックの出力:
+------+---------------------+------- ----------------------+-- -----+
|itemId|itemName |サプライヤー |itemNameElements |
+------+---------------------+------- ----------------------+-- -----+
|1 |雪中ウォーキング用厚手コート|スポーツカンパニー株式会社 |[厚手、コート、ウォーキング、イン、ザ、スノー]|
|2 |エレガントアウトドアサマーワンピース |YetiX | [エレガント,アウトドア,夏,ワンピース] |
+------+---------------------+------- ----------------------+-- --+ この質問の大きな難点は、カウントとサイズの違いを理解することです (以下のドキュメントを参照してください)。size は、行ごとに配列内の要素の数を返すため、ここで選択するのに正しい関数です。
この質問を解決するためのもう 1 つの考慮事項は、選択とフィルターの違いです。元の DataFrame の行を返したいので、フィルターが正しい選択です。select を使用する場合は、次のように、基準に一致する行を示す単一列の DataFrame を取得するだけです。
+--------------------------+
|(サイズ(項目名要素) > 3)|
+--------------------------+
|本当 |
|本当 |
|偽|
+--------------------------+
より詳しい情報：
カウントのドキュメント: pyspark.sql.functions.count - PySpark 3.1.1 ドキュメント サイズのドキュメント: pyspark.sql.functions.size - PySpark 3.1.1 ドキュメント 静的ノートブック | 動的ノートブック: テスト 1 を参照

最新問題**: 18**
次のコード ブロックのうち、文字 X を含まない DataFrame itemsDf の列サプライヤーのすべての値の 1 列の DataFrame を返すものはどれですか? DataFrame では、すべての値は 1 回だけリストされる必要があります。
DataFrame itemsDf のサンプル:
1.+-----+--------------------+------------------- ----------------------+
2.|アイテムID| アイテム名 |属性 |サプライヤー|

3.+------+--------------------+-------------------- ----------------------+
4.| 1|和装向け厚手コート|ブルー、冬、着心地]|株式会社スポーツカンパニー|
5.| 2|エレガントなアウトドア ...|[赤、夏、フレ...| YetiX|
6.| 3| アウトドアリュック|グリーン、夏、夏...|株式会社スポーツカンパニー|
7.+-----+--------------------+-------------------- ----------------------+
A. itemsDf.filter(col(サプライヤー).not_contains('X')).select(サプライヤー).distinct()
B. itemsDf.select(~col('supplier').contains('X')).distinct()
C. itemsDf.filter(not(col('supplier').contains('X'))).select('supplier').unique()
D. itemsDf.filter(~col('サプライヤー').contains('X')).select('サプライヤー').distinct()
E. itemsDf.filter(!col('サプライヤー').contains('X')).select(col('サプライヤー')).unique()
Answer: (解答を表示する**)**
説明
正しいコード ブロックの出力:
+----------------------+
| サプライヤー|
+----------------------+
|株式会社スポーツカンパニー|
+----------------------+
この質問に対処するための鍵は、操作の逆を行うためにどの演算子を使用するかを理解することです
- ~ (not) 演算子。さらに、unique() メソッドがないことを知っておく必要があります。
静的ノート | 動的ノートブック: テスト 1 を参照

最新問題: **19**
Spark のクラスター実行モードとクライアント実行モードの違いを説明しているものは次のうちどれですか?
A. クラスター モードではクラスター マネージャーはワーカー ノードに常駐しますが、クライアント モードではクラスター マネージャーはエッジ ノードに常駐します。
B. クラスター モードでは、実行プロセスはワーカー ノードで実行されますが、クライアント モードではゲートウェイ ノードで実行されます。
C. クラスター モードではドライバーはワーカー ノードに常駐しますが、クライアント モードではドライバーはエッジ ノードに常駐します。
D. クラスター モードではゲートウェイ マシンがドライバーをホストしますが、クライアント モードではゲートウェイ マシンがエグゼキューターと同じ場所に配置されます。
E. クラスター モードでは、Spark ドライバーはクラスター マネージャーと同じ場所に配置されませんが、クライアント モードでは同じ場所に配置されます。
Answer: C (メーッセージを残す**)**
説明
クラスター モードではドライバーはワーカー ノードに常駐しますが、クライアント モードではドライバーはエッジ ノードに常駐します。

正しい。Spark のクライアント モードの考え方は、クラスターの外側にあるエッジ ノード (ゲートウェイ マシンとも呼ばれる) からワークロードを実行できるというものです。ただし、Spark を実行する最も一般的な方法はクラスター モードであり、ドライバーはワーカー ノード上に常駐します。
実際には、クライアント モードでは、クラスター内のワーカー ノード間のデータ転送速度と比較して、データ転送速度に関して厳しい制約があります。また、エッジ ノードに障害が発生すると、クライアント モードで実行されるジョブはすべて失敗します。これらの理由により、クライアント モードは通常、運用環境では使用されません。
クラスター モードでは、クラスター マネージャーはワーカー ノードに常駐しますが、クライアント実行モードではクラスター マネージャーはエッジ ノードに常駐します。
いいえ。どちらの実行モードでも、クラスター マネージャーはワーカー ノードに常駐しますが、クライアント モードのエッジ ノードには常駐しません。
クラスター モードでは、実行プロセスはワーカー ノードで実行されますが、クライアント モードではゲートウェイ ノードで実行されます。
これは間違いです。ドライバーのみがクライアント モードのゲートウェイ ノード (匸ッジ ノード」とも呼ばれます) で実行され、エグゼキューター プロセスは実行されません。
クラスター モードでは、Spark ドライバーはクラスター マネージャーと同じ場所に配置されませんが、クライアント モードでは同じ場所に配置されます。
いいえ、クライアント モードでは、Spark ドライバーはドライバーと同じ場所に配置されません。クライアント モードの重要な点は、ドライバーがクラスターの外部にあり、クラスターを管理するリソース (クラスター マネージャーを実行するマシン) に関連付けられていないことです。
クラスター モードでは、ゲートウェイ マシンがドライバーをホストしますが、クライアント モードではゲートウェイ コンピューターがエグゼキューターと同じ場所に配置されます。
いいえ、まったく逆です。クラスター モードにはゲートウェイ マシンはありませんが、クライアント モードではドライバーがホストされます。

最新問題: **20**
Spark がフォールト トレランスを実現する方法を説明しているのは次のうちどれですか?
A. Spark は、MEMORY_AND_DISK ストレージ レベル オプションを提供することで、ワーカー障害が発生した場合のデータの迅速な回復に役立ちます。
B. RDD の計算中にワーカー ノード上のエグゼキュータが失敗した場合、その RDD はリネージュを使用して別のエグゼキュータによって再計算できます。
C. Spark は、レガシー RDD データ システム上にフォールト トレラント レイヤーを構築しますが、それ自体はフォールト トレラントではありません。
D. 変換後の DataFrame は可変であるため、ワーカー ノードに障害が発生した場合、Spark は観察されたリネージを使用して DataFrame を再現します。

E. Spark は、この機能がspark.fault_recovery.enabled プロパティによって明示的に有効にされている場合にのみフォールト トレラントになります。
Answer: B (メーッセージを残す**)**
説明
変換後の DataFrame は可変であるため、ワーカー ノードに障害が発生した場合、Spark は観察されたリネージを使用して DataFrame を再作成します。
間違い - 変換間では、DataFrame は不変です。Spark はリネージも記録するため、障害が発生した場合でも Spark は任意の DataFrame を再現できます。これら 2 つの側面は、Spark のフォールト トレランスを理解するための鍵となります。
Spark は、レガシー RDD データ システム上にフォールト トレラント レイヤーを構築しますが、それ自体はフォールト トレラントではありません。
間違い。RDD は Resilient Distributed Dataset の略で、Spark の中核であり、「レガシー システム」ではありません。
設計によりフォールトトレラントになっています。
Spark は、MEMORY_AND_DISK ストレージ レベル オプションを提供することで、ワーカー障害が発生した場合のデータの迅速な回復に役立ちます。
本当じゃない。ワーカー障害が発生した場合のリカバリをサポートするために、Spark は「2」、「3」などのストレージ レベル オプション (MEMORY_AND_DISK_2 など) を提供します。これらのストレージ レベルは、データの複製を複数のノードに保持するように特別に設計されています。これにより、最初にデータを再計算する必要がなく、データのコピーをすぐに使用できるため、ワーカー障害が発生した場合に時間を節約できます。
Spark は、この機能がspark.fault_recovery.enabled プロパティによって明示的に有効にされている場合にのみフォールト トレラントになります。
いいえ、Spark は設計上フォールト トレラントです。

最新問題: **21**
次のコード ブロックのうち、列 productId が 0 以下または 3 であるすべての行を DataFrame トランザクション Df から選択するものはどれですか?
A. transactionsDf.filter(productId==3 または productId<1)
B. transactionsDf.filter((col("productId")==3) または (col("productId")<1))
C. transactionsDf.filter(col("productId")==3 |col("productId")<1)
D. transactionsDf.where("productId"=3).or("productId"<1))
E. transactionsDf.filter((col("productId")==3) | (col("productId")<1))
Answer: (解答を表示する**)**
説明
この質問は、フィルター条件をチェーンする方法に関する知識を対象としています。各フィルタリング条件は括弧内に入れる必要があります。「or」の正しい演算子はパイプ文字 (|) であり、単語 or ではありません。もう 1 つの重要な演算子は等価演算子です。比較のために、等しいことは 2 つの等号 (==) で表されます。

静的ノート | 動的ノートブック: テスト 2 を参照

最新問題: **22**
アキュムレータの使用に関する問題は次のうちどれですか?
A. Spark UI では名前のないアキュムレータのみを検査できます。
B. アキュムレータで使用できるのは数値のみです。
C. アキュムレータの値はドライバーのみが読み取ることができ、エグゼキュータは読み取ることができません。
D. アキュムレータは遅延評価に従いません。
E. アキュムレータは、ハードウェア障害によりタスクを再実行する必要があるかどうかに関係なく、一度だけ更新されるため、デバッグに使用するのは困難です。
Answer: C (メーッセージを残す**)**
説明
アキュムレータの値はドライバーのみが読み取ることができ、エグゼキュータは読み取ることができません。
正しい。したがって、たとえば、エグゼキュータ間のワークロードを調整するためにアキュムレータ変数を使用することはできません。アキュムレータ値の典型的な正規の使用例は、デバッグ目的などでデータをドライバーに報告することです。たとえば、デバッグ目的で UDF 内の特定の条件に一致する値をカウントしたい場合、アキュムレータはそれを行うための優れた方法を提供します。
アキュムレータでは数値のみを使用できます。
いいえ。pySpark のアキュムレータは数値 (int と float を考えてください) のみをサポートしますが、AccumulatorParam インターフェイス (以下にリンクされているドキュメント) を介してカスタム型のアキュムレータを定義できます。
アキュムレータは遅延評価に従いません。
不正解 - アキュムレータは遅延評価に従います。これは実際に影響を及ぼします。アキュムレータが変換でカプセル化されると、そのアキュムレータは後続のアクションが実行されるまで変更されません。
アキュムレータは、ハードウェア障害によりタスクを再実行する必要があるかどうかに関係なく、一度だけ更新されるため、デバッグに使用するのは困難です。
間違い。アキュムレータに関する懸念は、実際、特定の条件下では各タスクに対して複数回実行される可能性があることです。たとえば、アキュムレータ変数が増加した後、タスクが終了する前に、タスク中にハードウェア障害が発生し、障害に応じて Spark が別のワーカーでタスクを起動した場合、すでに実行されたアキュムレータ変数の増加が繰り返されます。
Spark UI では、名前のないアキュムレータのみを検査できます。
いいえ。現在、PySpark では、Spark UI でアキュムレーターを検査できません。Spark の Scala インターフェイスでは、名前付きアキュムレーターのみが Spark UI で検査できます。

詳細: Spark アキュムレータを使用した結果の集計 | Sparkour、RDD プログラミング ガイド - Spark 3.1.2 ドキュメント、pyspark.Accumulator - PySpark 3.1.2 ドキュメント、および pyspark.AccumulatorParam - PySpark 3.1.2 ドキュメント

最新問題**: 23**
DataFrame itemsDf の列サプライヤーのそれぞれの値の隣にある列属性のすべての値のテーブルを作成するには、以下に示すコード ブロックをどの順序で実行する必要がありますか?
1. itemsDf.createOrReplaceView("itemsDf")
2.spark.sql("FROM itemsDf SELECT 'サプライヤー',explode('属性')")
3.spark.sql("FROM itemsDf SELECT サプライヤー、爆発(属性)")
4. itemsDf.createOrReplaceTempView("itemsDf")
A. 4、3
B. 1、3
C. 2
D. 4、2
E. 1、2
Answer: (解答を表示する**)**
説明
静的ノート | 動的ノートブック: テスト 1 を参照

最新問題**: 24**
データ スキューに関する次の記述のうち、間違っているものはどれですか?
A. Spark は、デフォルトではスキュー結合を自動的に最適化しません。
B. ブロードキャスト結合は、ソートマージ結合よりも偏ったデータの結合パフォーマンスを向上させる実行可能な方法です。
C. 偏ったデータフレームでは、最大のパーティションと最小のパーティションで消費されるメモリ量が大きく異なります。
D. スキューを軽減するために、Spark は結合時にキーの null 値を自動的に無視します。
E. ソルティングによりデータ スキューを解決できます。
Answer: D (メーッセージを残す**)**
説明
スキューを軽減するために、Spark は結合時にキーの null 値を自動的に無視します。
このステートメントは正しくないため、質問に対する正しい答えになります。NULL 値を含むキーの結合は、データ スキューに関して特に懸念されます。
実際のアプリケーションでは、結合キーとして使用される列に値が割り当てられていないレコードがテーブルに多数含まれる場合があります。結合中、データは大きく偏る危険があります。これは、NULL 値の結合キーを持つすべてのレコードが単一の大きなパーティションとして評価され、NULL キー以外のレコードの潜在的に多様なキー値 (したがって小さなパーティション) とはまったく対照的であるためです。

Spark は特にこれを自動的に処理しません。ただし、Null 値を一時的に破棄し、後でマージし直すなど、この問題を軽減するための戦略がいくつかあります (下記の最後のリンクを参照)。
スキューのある DataFrame では、最大のパーティションと最小のパーティションで消費されるメモリ量が大きく異なります。
この発言は正しいです。実際、パーティション サイズが大きく異なることが、まさにスキューの定義です。最大のパーティションが単一のエグゼキュータを長時間占有するため、スキューによって Spark のパフォーマンスが低下する可能性があります。これは Spark ジョブをブロックし、大きなパーティションが処理されるまで小さなパーティションを処理する他のエグゼキューターをアイドル状態にする必要があるため、リソースの非効率的な使用になります。
ソルティングによりデータの偏りを解決できます。
この発言は正しいです。ソルト化の目的は、ソルト化されたパーティション化キーに基づいて、データを同様のサイズのパーティションに再分割する機会を Spark に提供することです。
ソルト分割キーは通常、均一に分散された乱数で構成される列です。パーティション化キー列の一意のエントリの数は、希望するパーティション数の数と一致する必要があります。ソルト付きキーで再パーティション化した後は、すべてのパーティションのサイズがほぼ同じになるはずです。
Spark は、デフォルトではスキュー結合を自動的に最適化しません。
この発言は正しいです。自動スキュー結合最適化は、Adaptive Query Execution (AQE) の機能です。
デフォルトでは、AQE は Spark で無効になっています。これを有効にするには、Spark の spark.sql.adaptive.enabled 構成オプションをデフォルトの false のままにするのではなく、true に設定する必要があります。
スキュー結合を自動的に最適化するには、Spark のspark.sql.adaptive.skewJoin.enabled オプションも true (デフォルト) に設定する必要があります。
スキュー結合の最適化が有効になっている場合、Spark はスキュー結合を認識し、大きなパーティションを小さなパーティションに分割することで最適化するため、パフォーマンスが向上します。
ブロードキャスト結合は、ソートマージ結合よりも偏ったデータの結合パフォーマンスを向上させる実行可能な方法です。
この発言は正しいです。実際、ブロードキャスト結合は、条件によっては、偏ったデータの結合パフォーマンスを向上させるのに役立ちます。結合される DataFrame の 1 つは、他の DataFrame からのパーティションに沿って、各エグゼキュータのメモリに収まるのに十分な大きさである必要があります。この場合、ブロードキャスト結合はソートマージ結合よりも結合パフォーマンスを向上させます。
その理由は、偏ったデータを含むソートマージ結合には過剰なシャッフルが含まれるためです。シャッフル中にデータがクラスター内に送信され、最終的に Spark アプリケーショ

ンの速度が低下します。偏ったデータの場合、データ量とそれに伴う速度の低下が特に大きくなります。
ただし、ブロードキャスト結合はデータのシャッフルを減らすのに役立ちます。小さいテーブルはすべてのエグゼキューターに直接保存されるため、大量のネットワーク トラフィックが排除され、最終的にソートマージ結合と比較して結合パフォーマンスが向上します。
小さい方の DataFrame がデフォルトで設定されている 10 MB より大きい場合、スキュー結合の動作を最適化するために、Spark のspark.sql.autoBroadcastJoinThreshold 構成プロパティを手動で調整することが合理的である可能性があることに注意してください。
より詳しい情報：
- パフォーマンス チューニング - Spark 3.0.0 ドキュメント
- Spark のパフォーマンスを向上させるデータ スキューとガベージ コレクション
- セクション 1.2 - 歪んだデータの結合 * GitBook

最新問題**: 25**
以下に示すコード ブロックは、列間の区切り文字としてタブ (\t 文字) を使用し、欠損値を文字列 n/a として表現し、列名のヘッダー行を省略して、DataFrame トランザクション Df を単一の CSV ファイルとしてパス csvPath のディスクに書き込む必要があります。これを達成するには、コード ブロック内の空白を正しく埋める答えを選択してください。
transactionsDf.__1__.write.__2__(__3__, " ").__4__.__5__(csvPath)
A. 1.合体(1)
2.オプション
3. 「月」
4. オプション("ヘッダー", True)
5. パス
B. 1.合体(1)
2.オプション
3. 「ロルセップ」
4. オプション("nullValue", "n/a")
5. パス
C. 1. 再パーティション(1)
2.オプション
3. 「月」
4. オプション("nullValue", "n/a")
5.CSV
（正しい）
D. 1.csv
2.オプション
3. 「月」
4. オプション("空の値", "n/a")

5. パス
* 1. 再パーティション(1)
2.モード
3. 9月」
4.mode("nullValue", "n/a")
5.CSV
Answer: C (メーッセージを残す)
説明
正しいコードブロック:
transactionsDf.repartition(1).write.option("sep", "\t").option("nullValue", "n/a").csv(csvPath)
ここで、質問が具体的に求めていることを理解することが重要です。DataFrame を単一の CSV ファイルとして書き込みます。これは、パーティションについて考えるきっかけとなるはずです。デフォルトでは、すべてのパーティションは個別のファイルとして書き込まれるため、呼び出しに repatio(1) を含める必要があります。ここでも、coalesce(1) が機能します。
次に、この質問は、DataFrameWriter.csv (以下のリンク) で動作する Spark ドキュメント内のパラメーターを検索するよう促すものです。これらのパラメーターを適用するには、option() ステートメントが必要であることも知っておく必要があります。
最後の懸念は、一般的な呼び出し構造に関するものです。DataFrame の accessed write を呼び出すと、オプションが続き、csv を使用して DataFrame を書き込みます。ここでは csv(csvPath) の代わりに、save(csvPath, format='csv') を使用することもできます。
詳細: pyspark.sql.DataFrameWriter.csv - PySpark 3.1.1 ドキュメント 静的ノートブック | 動的ノートブック: テスト 1 を参照

最新問題**: 26**
以下に表示されるコード ブロックには複数のエラーが含まれています。コード ブロックは、DataFrametransactionsDf から列transactionDateを削除し、列transactionTimestampを追加する必要があります。この列では、DataFrametransactionDfの列transactionDateで文字列として表現される日付がUNIXタイムスタンプに変換されます。エラーを見つけてください。
DataFrame トランザクション Df のサンプル:
1.+---------------+----------+-----+----------+--------- +-----+---------------+
2.|transactionId|predError|value|storeId|productId| f| 取引日|
3.+-------------+----------+-----+----------+--------- +-----+---------------+
4.| 1| 3| 4| 25| 1|ヌル|2020-04-2615:35|
5.| 2| 6| 7| 2| 2|ヌル|2020-04-1322:01|
6.| 3| 3| null| 25| 3|ヌル|2020-04-0210:53|
7.+---------------+----------+-----+----------+--------- +-----+---------------+ コードブロック:
1.transactionsDf =transactionsDf.drop("transactionDate")

2.transactionsDf["transactionTimestamp"] = unix_timestamp("transactionDate", "yyyy-MM-dd")
A. 列transactionDateは、transactionTimestampが書き込まれた後に削除する必要があります。日付形式を示す文字列を調整する必要があります。既存の列割り当ての代わりに withColumn 演算子を使用する必要があります。unix_timestamp() の代わりに演算子 to_unixtime() を使用する必要があります。
B. 列transactionDateは、transactionTimestampが書き込まれた後に削除する必要があります。既存の列割り当ての代わりに withColumn 演算子を使用する必要があります。列 transactionDateはcol()演算子でラップする必要があります。
C. 列transactionDateはcol()演算子でラップする必要があります。
D. 日付形式を示す文字列を調整する必要があります。コード ブロック内のドロップ アンド アサイン パターンの代わりに withColumnReplaced 演算子を使用して、列 transactionDateを新しい列transactionTimestampに置き換える必要があります。
E. 列transactionDateは、transactionTimestampが書き込まれた後に削除する必要があります。日付形式を示す文字列を調整する必要があります。既存の列割り当ての代わりに withColumn 演算子を使用する必要があります。
Answer: (解答を表示する**)**
説明
この質問を正しく理解するには、多くの思考が必要です。この問題を解決するには、テスト中に提供されるデジタル メモ帳を活用するとよいでしょう。おそらく、コード ブロックに複数のエラーが含まれているのを見たことがあるでしょう。テストでは通常、エラーが 1 つだけ含まれるコード ブロックに直面します。ただし、ここで練習しているので、この挑戦的な複数エラーの問題により、実際の試験で単一エラーの問題に簡単に対処できるようになります。
列transactionDateは、transactionTimestampが書き込まれた後にのみ削除する必要があることが明確にわかります。これは、列transactionTimestampを生成するには、Sparkが列 transactionDateから値を読み取る必要があるためです。
元のtransactionsDf DataFrameの列transactionDateの値は、2020-04-26 15:35のようになります。したがって、これらを正しく変換するには、yyyy-MM-dd HH:mm を渡す必要があります。言い換えると：
日付形式を示す文字列を調整する必要があります。
(from_unixtime() 演算子に沿って) unix_timestamp() を to_unixtime() に変更したくなるかもしれませんが、この関数は Spark には存在しません。unix_timestamp() は、ここで使用する正しい演算子です。
また、DataFrame.withColumnReplaced() 演算子もありません。同様の演算子として DataFrame.withColumnRenamed() があります。
Col() を使用するかどうかは unix_timestamp() には無関係です。このコマンドはどちらでも問題ありません。

最後に、Spark では、transactionsDf["columnName"] = ... のような列を割り当てることはできません。これは Pandas 構文 (Pandas はデータ分析用の人気のある Python パッケージです) ですが、Spark ではサポートされていません。
したがって、代わりに Spark の DataFrame.withColumn() 構文を使用する必要があります。
詳細: pyspark.sql.functions.unix_timestamp - PySpark 3.1.2 ドキュメント 静的ノートブック | 動的ノートブック: テスト 3 を参照

最新問題**: 27**
次のコード ブロックのうち、各行に 0 から DataFrame トランザクション Df の列 predError で指定された数値を含むすべての整数の配列が含まれる 1 列の DataFrame を返します。predError が null の場合は null を返します。
DataFrame トランザクション Df のサンプル:
1.+---------------+----------+-----+----------+--------- +----+
2.|transactionId|predError|value|storeId|productId| f|
3.+-------------+----------+-----+----------+--------- +----+
4.| 1| 3| 4| 25| 1|ヌル|
5.| 2| 6| 7| 2| 2|ヌル|
6.| 3| 3| null| 25| 3|ヌル|
7.| 4| null| null| 3| 2|ヌル|
8.| 5| null| null| null| 2|ヌル|
9.| 6| 3| 2| 25| 2|ヌル|
10.+---------------+----------+-----+----------+--------- +----+
A. 1.def count_to_target(ターゲット):
2. ターゲットが "なし"の場合:
3. 戻る
4.
5. 結果 = [範囲(ターゲット)]
6. 結果を返す
7。
8.count_to_target_udf = udf(count_to_target, ArrayType[IntegerType])
9.
10.transactionsDf.select(count_to_target_udf(col('predError')))
B. 1.def count_to_target(ターゲット):
2. ターゲットが "なし"の場合:
3. 戻る
4.
5.結果 = リスト(範囲(ターゲット))
6. 結果を返す
7。
8.transactionsDf.select(count_to_target(col('predError')))

C. 1.def count_to_target(ターゲット):
2. ターゲットが "なし"の場合:
3. 戻る
4.
5.結果 = リスト(範囲(ターゲット))
6. 結果を返す
7。
8.count_to_target_udf = udf(count_to_target, ArrayType(IntegerType()))
9.
10.transactionsDf.select(count_to_target_udf('predError'))
（正しい）
D. 1.def count_to_target(ターゲット):
2.結果 = リスト(範囲(ターゲット))
3. 結果を返す
4.
5.count_to_target_udf = udf(count_to_target, ArrayType(IntegerType()))
6.
7.df =transactionsDf.select(count_to_target_udf('predError'))
E. 1.def count_to_target(ターゲット):
2. ターゲットが "なし"の場合:
3. 戻る
4.
5.結果 = リスト(範囲(ターゲット))
6. 結果を返す
7。
8.count_to_target_udf = udf(count_to_target)
9.
10.transactionsDf.select(count_to_target_udf('predError'))
Answer: (解答を表示する**)**
説明
正しいコードブロック:
def count_to_target(ターゲット):
ターゲットがなしの場合:
戻る
結果 = リスト(範囲(ターゲット))
結果を返す
count_to_target_udf = udf(count_to_target, ArrayType(IntegerType()))
transactionsDf.select(count_to_target_udf('predError'))
正しいコード ブロックの出力:
+----------------------------------+

|count_to_target(predError)|
+----------------------------------+
| [0, 1, 2]|
| [0、1、2、3、4、5]|
| [0, 1, 2]|
| null|
| null|
| [0, 1, 2]|
+----------------------------------+
この質問は決して簡単ではありません。UDF (ユーザー定義関数) に関する構文に精通している必要があります。具体的には、この質問では、正しい型を udf メソッドに渡すことが重要です。単一の型ではなく特定の型の配列を返すということは、型の影響について通常よりも注意深く考える必要があることを意味します。
Spark では、常に ArrayType(IntegerType) ではなく、ArrayType(IntegerType()) のようなインスタンス化された方法で型を渡すことに注意してください。ここで重要なのは括弧 () です。忘れないように注意してください。
また、実際には、Python メソッド count_to_target ではなく、UDF count_to_target_udf を select() 演算子に渡していることにも注意する必要があります。
最後に、UDF では null 値は常に扱いにくいケースです。したがって、コードがそれらを正しく処理できるように注意してください。
詳細情報: Python 関数を PySpark 関数 (UDF) に変換する方法 - Chang Hsin Lee - 私の考えを言葉に表します。
静的ノート | 動的ノートブック: テスト 3 を参照

最新問題**: 28**
クラスター上で実行されているアプリケーションが 1 つだけである場合、大量のデータを処理するときに Spark のパフォーマンスを向上させる実行可能な方法は次のうちどれですか?
A. プロパティspark.default.Parallelismおよびspark.sql.shuffle.partitionsの値を増やします。
B. プロパティspark.default.Parallelismおよびspark.sql.partitionsの値を減らします。
C. プロパティspark.sql.Parallelismおよびspark.sql.partitionsの値を増やします。
D. プロパティspark.sql.Parallelismおよびspark.sql.shuffle.partitionsの値を増やします。
E. プロパティspark.dynamicAllocation.maxExecutors、spark.default.Parallelism、およびspark.sql.shuffle.partitionsの値を増やします。
Answer: A (メーッセージを残す**)**
説明
プロパティspark.default.Parallelismおよびspark.sql.partitionsの値を減らします。いいえ、これらの値を増やす必要があります。

プロパティspark.sql.Parallelismおよびspark.sql.partitionsの値を増やします。間違っています。spark.sql.Parallelismプロパティはありません。
プロパティspark.sql.Parallelismおよびspark.sql.shuffle.partitionsの値を増やします。上記を参照してください。
プロパティspark.dynamicAllocation.maxExecutors、spark.default.Parallelism、およびspark.sql.shuffle.partitionsの値を増やします。プロパティspark.dynamicAllocation.maxExecutorsは、spark.dynamicAllocation.enabledプロパティを使用して動的割り当てが有効になっている場合にのみ有効です。。デフォルトでは無効になっています。動的割り当ては、同じクラスター上で複数のアプリケーションを並行して実行する場合に役立ちます。ただし、この場合、クラスター上で実行されているアプリケーションは 1 つだけであるため、動的割り当てを有効にしてもパフォーマンスの向上は得られません。
詳細情報: データ サイエンティストのための実践的な Spark のヒント | Experfy.com と Apache Spark 構成設定の基本 | ハリル ·エルタン著 | データサイエンスに向けて (https://bit.ly/3gA0A6w 、 https://bit.ly/2QxhNTr)

最新問題**: 29**
Spark の実行階層で最も深いレベルは次のうちどれですか?
A. 仕事
B. タスク
C. 執行者
D. スロット
E. ステージ
Answer: B (メーッセージを残す**)**
説明
階層は上から下に、ジョブ、ステージ、タスクです。
エグゼキュータとスロットはタスクの実行を容易にしますが、それらは階層の直接の一部ではありません。エグゼキュータは、特定の Spark アプリケーションを実行する目的で、ワーカー ノード上のドライバーによって起動されます。スロットは、Spark の作業の並列化に役立ちます。エグゼキュータには複数のスロットを持たせることができ、これにより複数のタスクを並行して処理できます。

最新問題**: 30**
Spark のスタンドアロン デプロイメント モードについて説明しているのは次のうちどれですか?
A. スタンドアロン モードでは、単一の JVM を使用して Spark ドライバーとエグゼキューターのプロセスを実行します。
B. スタンドアロン モードは、クラスターにドライバーが含まれていないことを意味します。

C. スタンドアロン モードは、Spark が YARN および Mesos クラスター上で実行される方法です。
D. スタンドアロン モードでは、アプリケーションごとにワーカーごとに 1 つのエグゼキュータのみを使用します。
E. スタンドアロン モードは、Spark だけでなく、複数のフレームワークを実行するクラスターにとって実行可能なソリューションです。
Answer: D (メーッセージを残す**)**
説明
スタンドアロン モードでは、アプリケーションごとのワーカーごとに 1 つのエグゼキュータのみを使用します。
これは正しいことであり、Spark のスタンドアロン モードの制限です。
スタンドアロン モードは、複数のフレームワークを実行するクラスターにとって有効なソリューションです。
正しくない。スタンドアロン モードの制限は、Apache Spark がクラスター上で実行される唯一のフレームワークである必要があることです。同じクラスター上で複数のフレームワーク (Apache Spark と Apache Flink など) を並行して実行する場合は、YARN デプロイメント モードを検討します。
スタンドアロン モードでは、単一の JVM を使用して Spark ドライバーとエグゼキューターのプロセスを実行します。
いいえ、これがローカル モードの動作です。
スタンドアロン モードは、Spark が YARN および Mesos クラスター上で実行される方法です。
いいえ。YARN モードと Mesos モードは、スタンドアロン モードとは異なる 2 つのデプロイメント モードです。これらのモードを使用すると、Spark をクラスター上の他のフレームワークと並行して実行できます。Spark がスタンドアロン モードで実行されている場合、クラスター上で実行できるのは Spark フレームワークのみです。
スタンドアロン モードは、クラスターにドライバーが含まれていないことを意味します。
不正解です。クライアント モードではクラスターにドライバーが含まれていませんが、スタンドアロン モードではドライバーがクラスター内のノードで実行されます。
詳細情報: Learning Spark、第 2 版、第 1 章

最新問題**: 31**
有効な実行モードは次のうちどれですか?
A. Kubernetes、ローカル、クライアント
B. クライアント、クラスター、ローカル
C. サーバー、スタンドアロン、クライアント
D. クラスター、サーバー、ローカル
E. スタンドアロン、クライアント、クラスター
Answer: B (メーッセージを残す**)**

説明
実行モードと展開モードは混同されやすいため、これを正しく理解するのは難しい質問です。文学の中でも、両方の用語が同じ意味で使用されることがあります。
Spark には、クライアント、クラスター、ローカル実行モードの 3 つの有効な実行モードのみがあります。実行モードは特定のフレームワークを参照するのではなく、インフラストラクチャが相互に配置される場所を参照します。
クライアント モードでは、ドライバーはクラスターの外部のマシン上に配置されます。クラスター モードでは、ドライバーはクラスター内のマシン上に配置されます。最後に、ローカル モードでは、すべての Spark インフラストラクチャが単一のコンピューター上の単一の JVM (Java 仮想マシン) で開始され、ドライバーも含まれます。
デプロイメント モードは、多くの場合、Spark をクラスター モードでデプロイする方法と、Spark 外部の特定のフレームワークを使用する方法を指します。有効なデプロイメント モードは、スタンドアロン、Apache YARN、Apache Mesos、および Kubernetes です。
クライアント、クラスター、ローカル
正解です。これらはすべて Spark の有効な実行モードです。
スタンドアロン、クライアント、クラスター
いいえ、スタンドアロンは有効な実行モードではありません。ただし、これは有効な展開モードです。
Kubernetes、ローカル、クライアント
いいえ、Kubernetes はデプロイメント モードですが、実行モードではありません。
クラスター、サーバー、ローカル
いいえ、サーバーは実行モードではありません。
サーバー、スタンドアロン、クライアント
いいえ、スタンドアロンとサーバーは実行モードではありません。
詳細情報: Apache Spark 内部 - 学習ジャーナル

最新問題: 32

以下に表示されているコード ブロックにはエラーが含まれています。以下のコード ブロックが実行されると、列storeId とtransactionDate (この順序) に基づいて、DataFrame の transactionDf が 14 の部分に分割されているはずです。エラーを見つけてください。
コードブロック:
transactionDf.coalesce(14, ("storeId", "transactionDate"))
A. 列名の周囲のかっこを削除し、コード ブロックに .select() を追加する必要があります。
B. 演算子の結合を再パーティションに置き換え、列名の周囲のかっこを削除し、コード ブロックに .count() を追加する必要があります。
（正しい）
C. 演算子の合体を再パーティションに置き換え、列名の周囲のかっこを削除し、コード ブロックに .select() を追加する必要があります。
D. 演算子 coalesce は repartition に置き換える必要があり、列名を囲む括弧は角括弧に置き換える必要があります。
E. オペレーター合体を再分割によって置き換える必要があります。
Answer: B (メーッセージを残す**)**
説明
正しいコードブロック:
transactionDf.repartition(14, "storeId", "transactionDate").count()
DataFrametransactionDf のパーティション数がわからないため、現在のパーティション数が 14 より小さい場合は何も変更されないため、Coalesce を安全に使用することはできません。
したがって、再パーティションを使用する必要があります。
Spark のドキュメントでは、再パーティションの呼び出し構造は次のように示されています。
DataFrame.repartition(numPartitions, *cols)。* 演算子は、numPartitions の後の引数が列として解釈されることを意味します。したがって、ブラケットを取り外す必要があります。
最後に、質問では、実行後に DataFrame を分割する必要があると指定しています。したがって、この質問は間接的に、コード ブロックにアクションを追加するように求めています。.select() は変換なので。ここで可能な唯一の選択肢は .count() です。
詳細: pyspark.sql.DataFrame.repartition - PySpark 3.1.1 ドキュメント 静的ノートブック | 動的ノートブック: テスト 1 を参照

最新問題**: 33**
以下に表示されているコード ブロックにはエラーが含まれています。コード ブロックは、Python メソッド find_most_freq_letter を使用して、DataFrame itemsDf の列 itemName に最も多く存在する文字を検索し、それを新しい列 most_frequent_letter で返す必要があります。エラーを見つけてください。
コードブロック:
1. find_most_freq_letter_udf = udf(find_most_freq_letter)
2. itemsDf.withColumn("most_frequent_letter", find_most_freq_letter("itemName"))

A. Spark は UDF メソッドを正しく使用していません。
B. 戻り値の型が欠落しているため、UDF メソッドが正しく登録されていません。
C. 「itemName」式は、col() でラップする必要があります。
D. UDF は PySpark に存在しません。
E. Spark は列を追加していません。
Answer: A (メッセージを残す)
説明
正しいコードブロック:
find_most_freq_letter_udf = udf(find_most_frequent_letter)
itemsDf.withColumn("most_frequent_letter", find_most_freq_letter_udf("itemName")) Spark は、ここで以前に登録された find_most_freq_letter_udf メソッドを使用する必要がありますが、元のコードブロックではそれを行っていません。そこでは、Python メソッドの非 UDF バージョンを使用するだけです。
通常、udf() の戻り値の型を指定する必要があることに注意してください。この場合を除きます。udf() のデフォルトの戻り値の型は文字列であり、これがここで期待されるものです。代わりに整数変数を返したい場合は、find_most_freq_letter_udf = udf(find_most_freq_letter, IntegerType()) を使用して、Python 関数を UDF として登録する必要があります。
詳細: pyspark.sql.functions.udf - PySpark 3.1.1 ドキュメント

最新問題**: 34**
以下に表示されているコード ブロックにはエラーが含まれています。コード ブロックは、結合または集計のためにエグゼキューター間でデータを交換するときに、データを 20 の部分に分割するように Spark を構成する必要があります。エラーを見つけてください。
コードブロック:
スパーク.conf.set(spark.sql.shuffle.partitions, 20)
A. コード ブロックは、オプションを設定するために間違ったコマンドを使用しています。
B. コード ブロックが間違ったオプションを設定しています。
C. コード ブロックでオプションが正しく表現されていません。
D. コード ブロックが間違ったパーツ数を設定しています。
E. コード ブロックにパラメータがありません。
Answer: C (メッセージを残す)
説明
正しいコードブロック:
スパーク.conf.set("spark.sql.shuffle.partitions", 20)
コード ブロックはオプションを誤って表現しています。
正しい！オプションは文字列として表現する必要があります。
コード ブロックが間違ったオプションを設定しています。
いいえ、spark.sql.shuffle.partitions は質問のユースケースの正しいオプションです。

コード ブロックが間違ったパーツ数を設定しています。
違います。コード ブロックには 20 個の部分が正しく記載されています。
コード ブロックは、オプションを設定するために間違ったコマンドを使用しています。
いいえ、PySpark では、spark.conf.set() がオプションを設定するための正しいコマンドです。
コード ブロックにパラメーターがありません。
不正解です。spark.conf.set() は 2 つのパラメータを取ります。
詳細情報: 構成 - Spark 3.1.2 ドキュメント

最新問題: **35**
ブロードキャスト変数に関する次の記述のうち、正しいものはどれですか?
A. ブロードキャスト変数は、単一タスクごとにシリアル化されます。
B. ブロードキャスト変数は、メモリに収まらないテーブルによく使用されます。
C. ブロードキャスト変数は不変です。
D. ブロードキャスト変数はタスクごとに動的に更新されることがあります。
E. ブロードキャスト変数はワーカー ノードに対してローカルであり、クラスター全体で共有されません。
Answer: C (メーッセージを残す)
説明
ブロードキャスト変数はワーカー ノードに対してローカルであり、クラスター全体では共有されません。
ブロードキャスト変数はクラスター全体で共有されることを意図しているため、これは誤りです。したがって、それらはワーカー ノードに対してローカルなだけではなく、すべてのワーカー ノードで利用できます。
ブロードキャスト変数は、メモリに収まらないテーブルによく使用されます。
ブロードキャスト変数は小さくてメモリに収まるためにのみブロードキャストできるため、これは誤りです。
ブロードキャスト変数は、単一タスクごとにシリアル化されます。
これは誤りです。なぜなら、タスクはクラスター内のすべてのマシンにキャッシュされ、単一タスクごとにシリアル化する必要がなくなるからです。
ブロードキャスト変数は、タスクごとに動的に更新されることがあります。
ブロードキャスト変数は不変であり、決して更新されないため、これは誤りです。
詳細情報: Spark - 決定版ガイド、第 14 章

最新問題: **36**
Spark 実行階層におけるタスクの役割を説明しているものは次のうちどれですか?
A. タスクは実行階層の最小要素です。
B. 1 つのタスク内で、スロットはデータの各パーティションに対して実行される作業の単位です。

C. タスクは実行階層内で 2 番目に小さい要素です。
D. 依存関係が狭いステージを 1 つのタスクにグループ化できます。
E. 幅広い依存関係を持つタスクを 1 つのステージにグループ化できます。
Answer: A (メーッセージを残す**)**
説明
依存関係が狭いステージは 1 つのタスクにグループ化できます。
違います。依存関係が狭いタスクは 1 つのステージにグループ化できます。
幅広い依存関係を持つタスクは 1 つのステージにグループ化できます。
ワイド変換を行うと常にステージの境界を示すシャッフルが発生するため、間違っています。したがって、幅広い依存関係を持つ複数のタスクを1 つのステージにバンドルすることはできません。
タスクは、実行階層内で 2 番目に小さい要素です。
いいえ、それらは実行階層の最小要素です。
1 つのタスク内で、スロットはデータの各パーティションに対して実行される作業の単位です。
いいえ、タスクはパーティションごとに実行される作業の単位です。スロットは、Spark の作業の並列化に役立ちます。エグゼキュータには複数のスロットを持たせることができ、これにより複数のタスクを並行して処理できます。

最新問題**: 37**
Spark が Hadoop と比較して持つパフォーマンス上の大きな利点の 1 つは次のうちどれですか?
A. Spark はデータを DAG 形式で保存することで優れたパフォーマンスを実現しますが、Hadoop は寄木細工のファイルしか使用できません。
B. Spark は、Hadoop とは異なり、Kubernetes 上にデプロイできるため、クエリに対する高い復元力を実現します。
C. Spark はメモリ内にデータを保存し、計算を実行することで優れたパフォーマンスを実現しますが、Hadoop での大規模なジョブには、比較的遅いディスク I/O 操作が大量に必要になります。
D. Spark はデータを HDFS 形式で保存することで優れたパフォーマンスを実現しますが、Hadoop は寄木細工のファイルしか使用できません。
E. Spark は、ユーザー フレンドリーな API で Hadoop の DataFrame を拡張することで、開発者のパフォーマンスの向上を実現します。
Answer: C (メーッセージを残す**)**
説明
Spark はデータを DAG 形式で保存することで優れたパフォーマンスを実現しますが、Hadoop は寄木細工のファイルしか使用できません。

違います。「DAG 形式」はありません。DAG は「有向非巡回グラフ」の略です。DAG は、Spark で計算ステップを表す手段です。ただし、Hadoop が DAG を使用しないのは事実です。
Spark での DAG の導入は、ディスクへのデータの書き込みとディスクからの読み取りを継続的に行う必要があるという Hadoop のマップ リデュース フレームワークの制限の結果でした。
Apache Spark のグラフ DAG - DataFlair
Spark はデータを HDFS 形式で保存することで優れたパフォーマンスを実現しますが、Hadoop は寄木細工のファイルしか使用できません。
いいえ、Spark は確かにデータを HDFS (および他の形式) に保存できますが、これは Hadoop に比べてパフォーマンス上の重要な利点ではありません。Hadoop は、Parquet だけでなく、複数のファイル形式を使用できます。
Spark は、Hadoop とは異なり、Kubernetes 上にデプロイできるため、クエリに対してより高い復元力を実現します。
いいえ、質問では回復力は求められていません。質問はパフォーマンスの向上についてです。
Hadoop と Spark はどちらも Kubernetes にデプロイできます。
Spark は、ユーザー フレンドリーな API で Hadoop の DataFrame を拡張することで、開発者のパフォーマンスを向上させます。
いいえ、DataFrame は Spark の概念ですが、Hadoop の概念ではありません。

最新問題: **38**
以下に表示されているコード ブロックにはエラーが含まれています。コード ブロックは、結合の実行時に最大 20 MB のサイズの DataFrame がすべてのワーカー ノードにブロードキャストされるように Spark を構成する必要があります。
エラーを見つけてください。
コードブロック:
A. spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 20)
B. Spark は、デフォルト値よりもはるかに小さい DataFrame のみをブロードキャストします。
C. 構成を書き込むための正しいオプションは、spark.conf ではなく、spark.config を使用することです。
D. Spark は制限をしきい値結合にのみ適用し、他の結合には適用しません。
E. 渡された制限の変数タイプが間違っています。
F. コマンドは遅延評価され、その後にアクションが続く必要があります。
Answer: (解答を表示する**)**
説明
これは難しい質問です。さまざまな答えを 1 つずつ評価してみましょう。

Spark は、デフォルト値よりもはるかに小さい DataFrame のみをブロードキャストします。
これは正しいです。デフォルト値は 10 MB (10485760 バイト) です。Spark.sql.autoBroadcastJoinThreshold の構成では (メガバイトではなく) バイト単位の数値が必要であるため、コード ブロックは、要求された 20 * 1024 * 1024 (= 20971520) バイトではなく、制限を単に 20 バイトに設定します。
コマンドは遅延評価されるため、その後にアクションを実行する必要があります。
いいえ、このコマンドはすぐに評価されます。
Spark は制限をしきい値結合にのみ適用し、他の結合には適用しません。
「しきい値結合」がないため、このオプションは意味がありません。
構成を書き込むための正しいオプションは、spark.conf ではなく、spark.config を使用することです。
いいえ、それは確かにspark.confです!
渡された制限の変数タイプが間違っています。
この構成では、バイト数 (数値) が入力として期待されます。したがって、コード ブロックで指定されている 20 は問題ありません。

最新問題**: 39**
以下に示すコード ブロックは、列transactionDateFormをDataFrametransactionsDfに追加する必要があります。この列は、transactionDate 列の Unix 形式のタイムスタンプを 4 月 26 日 (日曜日) のように文字列型で表現する必要があります。これを達成するには、コードブロック内の空白を正しく埋める答えを選択してください。
トランザクションDf.__1__(__2__, from_unixtime(__3__, __4__))
A. 1. withColumn
2. 「トランザクション日付フォーム」
3. 「MMM d (EEEE)」
4. 「取引日」
B. 1. を選択します。
2. 「取引日」
3. 「トランザクション日付フォーム」
4. 「MMM d (EEEE)」
C. 1. withColumn
2. 「トランザクション日付フォーム」
3. 「取引日」
4. 「MMM d (EEEE)」
D. 1.withColumn
2. 「トランザクション日付フォーム」
3. 「取引日」
4. 「MM d (EEE)」

E. 1.withColumnRenamed
2. 取引日」
3. 「トランザクション日付フォーム」
4. 「MM d (EEE)」
Answer: (解答を表示する**)**
説明
正しいコードブロック:
transactionDf.withColumn("transactionDateForm", from_unixtime("transactionDate", "MMM d (EEEE)")) この質問は、特に列の 「追加」について尋ねています。提示されたすべての回答のコンテキストでは、DataFrame.withColumn() がこれに対する正しいコマンドです。理論的には、既存の列をすべて選択して新しい列を追加する場合、DataFrame.select() をこの目的に使用することもできます。
DataFrame.withColumnRenamed() は、既存の列の名前を変更することしかできず、新しい列を追加したり、列の値を変更したりすることはできないため、適切なコマンドではありません。
DataFrame.withColumn() を選択すると、ドキュメント (以下を参照) で、メソッドへの最初の入力引数は新しい列の列名にする必要があることがわかります。
最後の難関は日付形式です。この質問は、日付形式として 4 月 26 日 (日曜日) が必要であることを示しています。答えには 「MM d (EEEE)」と 「MM d (EEEE)」が選択肢として表示されます。Spark で使用される日付形式の詳細を知るのは難しい場合があります。特に、MMM と MM の違いを知ることは、おそらく毎日対処することではありません。ただし、違いを覚える簡単な方法があります。通常、M (1 文字) は最も短い形式で、4 月を表す 4 です。MM にはパディングが含まれます: 4 月の場合は 04。MMM (3 文字) は 3 文字の月の略称で、4 月を表す Apr です。MMMM は可能な限り長い形式です: April です。この 4 文字の順序を知っておくと、ここで正しいオプションを選択するのに役立ちます。
詳細: pyspark.sql.DataFrame.withColumn - PySpark 3.1.2 ドキュメント 静的ノートブック | 動的ノートブック: テスト 3 を参照

最新問題**: 40**
以下に表示されているコード ブロックにはエラーが含まれています。コード ブロックは、一意の storeId によってグループ化された列値の行の平均を返す必要があります。エラーを見つけてください。
コードブロック:
transactionsDf.agg("ストアID").avg("値")
A. avg("value") の代わりに、avg(col("value")) を使用する必要があります。
B. avg("value") は、agg() に追加するのではなく、2 番目の引数として指定する必要があります。
C. すべての列名は、col() 演算子で囲む必要があります。
D. agg は groupBy に置き換える必要があります。

E. 「storeId」と「value」を交換する必要があります。
Answer: D (メッセージを残す**)**
説明
静的ノート | 動的ノートブック: テスト 1 を参照
(https://flrs.github.io/spark_practice_tests_code/#1/30.html 、
https://bit.ly/sparkpracticeexams_import_instructions)

最新問題**: 41**
狭い変換を説明しているのは次のうちどれですか?
A. ナロー変換は、パーティション間でデータを交換する操作です。
B. ナロー変換は、複数の RDD からのデータが使用されるプロセスです。
C. ナロー変換は、32 ビットの float 変数を 16 ビットや 8 ビットの float 変数などのより小さな float 変数にキャストするプロセスです。
D. ナロー変換は、クラスター全体でデータを交換する操作です。
E. ナロー変換は、クラスター間でデータが交換されない操作です。
Answer: (解答を表示する**)**
説明
ナロー変換は、クラスター間でデータが交換されない操作です。
正しい！ナロー変換では、これらの変換は適用されるパーティションの外部からのデータを必要としないため、クラスター全体でデータは交換されません。典型的な狭い変換には、フィルタ、ドロップ、合体が含まれます。
ナロー変換は、パーティション間でデータを交換する操作です。
いいえ、それは広い変換の 1 つの定義ではありますが、狭い変換の定義ではありません。通常、広範囲にわたる変換ではシャッフルが発生し、パーティション、エグゼキュータ、クラスター間でデータが交換されます。
ナロー変換は、クラスター全体でデータを交換する操作です。
いいえ、このすぐ上の説明を参照してください。
ナロー変換は、32 ビットの float 変数をより小さな float 変数にキャストするプロセスです。
16 ビットまたは 8 ビットの浮動小数点変数。
いいえ、型変換は Spark の狭い変換とは何の関係もありません。
ナロー変換は、複数の RDD からのデータが使用されるプロセスです。
いいえ。回復力のある分散データセット (RDD) は、パーティションのコレクションとして説明できます。ナロー変換では、パーティション間でデータは交換されません。したがって、RDD 間でデータは交換されません。
ただし、狭い変換、および実際にはどのような変換でも新しい RDD が作成されると言えるでしょう。これは、変換により既存の RDD が変更されるためです (RDD は、DataFrame などの他の Spark データ構造の基礎です)。ただし、RDD は不変であるため、変換によって生じた変更を反映するには、新しい RDD を作成する必要があります。

詳細情報: Spark の変換とアクション: 詳細 | ミスバ ウディン | 2011/11/1 コードX | 中くらい

最新問題**: 42**
filePath に保存されている JSON ファイルを DataFrame として読み取るコード ブロックは次のどれですか?
A. spark.read.json(ファイルパス)
B. spark.read.path(filePath,source="json")
C. spark.read().path(filePath)
D. spark.read().json(ファイルパス)
E. spark.read.path(ファイルパス)
Answer: A (メーッセージを残す**)**
説明
スパーク.read.json(ファイルパス)
正しい。smile.read は Spark の DataFrameReader にアクセスします。次に、Spark は、filePath を DataFrameReader.json() メソッドに渡すことによって、JSON タイプとして読み取られるファイル タイプを識別します。
スパーク.読み取り.パス(ファイルパス)
正しくない。Spark の DataFrameReader にはパス メソッドがありません。ファイルを読み取るための汎用的な方法は、DataFrameReader.load() メソッド (以下のリンク) によって提供されます。
smile.read.path(filePath,source="json")
間違い。DataFrameReader.path() メソッドは存在しません (上記を参照)。
スパーク.read().json(ファイルパス)
正しくない。smile.read は、Spark の DataFrameReader にアクセスする方法です。ただし、DataFrameReader は呼び出し可能ではないため、spark.read() 経由での呼び出しは失敗します。
スパーク.read().path(ファイルパス)
いいえ、Spark の DataFrameReader は呼び出し可能ではありません (上記を参照)。
詳細: pyspark.sql.DataFrameReader.json - PySpark 3.1.2 ドキュメント、pyspark.sql.DataFrameReader.load - PySpark 3.1.2 ドキュメント 静的ノートブック | 動的ノートブック: テスト 3 を参照

最新問題**: 43**
以下に示すコード ブロックは、列 cos が追加された DataFrametransactionDf のコピーを返す必要があります。
この列には、列 value の値が度に変換され、それらの変換された値のコサインが取得され、小数点第 2 位に四捨五入されている必要があります。これを達成するには、コード ブロック内の空白を正しく埋める答えを選択してください。
コードブロック:

トランザクションDf.__1__(__2__,round(__3__(__4__(__5__)),2))
A. 1. withColumn
2.col("cos")
3.cos
4.度
5.transactionDf.value
B. 1.withColumnRenamed
2. ロス」
3.cos
4.度
5. ransactionsDf.value」
C. 1. withColumn
2. ロス」
3.cos
4.度
5.transactionDf.value
D. 1.withColumn
2.col("cos")
3.cos
4.度
5.col("値")
E
。1.列あり
2. ロス」
3.度
4.cos
5.col("値")
Answer: C (メーッセージを残す**)**
説明
正しいコードブロック:
transactionsDf.withColumn("cos",round(cos(degrees(transactionsDf.value)),2)) この質問は、col と "cos" が非常に似ているため、特に混乱を招きます。試験では似たような解答の選択肢が現れる可能性があり、この問題と同様に、正しい解答の選択肢を特定するには細部に注意を払う必要があります。
除外する最初の回答オプションは、withColumnRenamed: 質問 NO: で始まるものです。特に列の追加について説明します。ただし、withColumnRenamed 演算子は既存の列の名前を変更するだけなので、ここでは使用できません。
次に、transactionsDf.withColumn() の最初の引数であるギャップ 2 に何を入れるかを決定する必要があります。

ドキュメント (以下にリンク) を見ると、withColumn の最初の引数は実際には追加する列の名前を含む文字列である必要があることがわかります。したがって、ギャップ 2 のオプションとしてcol("cos") を含む回答は無視できます。
これにより、考えられる答えは 2 つになります。これら 2 つの答えの本当の違いは、cos 法と次数法がどこにあるか (ギャップ 3 と 4、またはその逆) です。質問から、新しい列には 「列 value の値が度に変換され、それらの変換された値のコサインが取られる」必要があることがわかります。これにより、操作の明確な順序が規定されます。まず、値を列の値から度に変換し、次にそれらの値のコサインを取得します。したがって、内側の括弧 (ギャップ 4) には次数法が含まれている必要があり、論理的にはギャップ 3 に cos 法が含まれます。これにより、考えられる正解は 1 つだけになります。
詳細: pyspark.sql.DataFrame.withColumn - PySpark 3.1.2 ドキュメント 静的ノートブック | 動的ノートブック: テスト 3 を参照

最新問題**: 44**
次のコード ブロックのうち、DataFrame のパーティションを 12 から 6 に減らし、フル シャッフルを実行するものはどれですか?
A. DataFrame.repartition(12)
B. DataFrame.coalesce(6).shuffle()
C. DataFrame.coalesce(6)
D. DataFrame.coalesce(6, shuffle=True)
E. DataFrame.repartition(6)
Answer: (解答を表示する**)**
説明
DataFrame.repartition(6)
正しい。repartition() は常に完全なシャッフルをトリガーします (coalesce() とは異なります)。
DataFrame.repartition(12)
いいえ、これでは DataFrame に 6 つのパーティションではなく 12 のパーティションが残るだけです。
DataFrame.coalesce(6)
coalesce はデータの完全なシャッフルを実行しません。「フル シャッフル」を見ると、coalesce() を扱っていないことがわかります。coalesce() は必要に応じて部分的なシャッフルを実行できますが、シャッフル操作を最小限に抑え、エグゼキュータ間で送信されるデータ量を最小限に抑えようとします。
ここでは、2 つのパーティションを 1 つに結合するだけで、12 のパーティションを簡単に 6 つのパーティションに再分割できます。
DataFrame.coalesce(6, shuffle=True) および DataFrame.coalesce(6).shuffle() これらのステートメントは有効な Spark API 構文ではありません。
詳細: Spark の再分割と合体 - Apache Spark での再分割と合体についての説明 - Rock the JVM ブログ

最新問題: **45**
次のコード ブロックのうち、DataFrame itemsDf をストレージ場所 filePath のディスクに書き込み、その場所にある既存のデータを確実に置き換えるものはどれですか?
A. itemsDf.write.mode("上書き").parquet(filePath)
B. itemsDf.write.option("parquet").mode("overwrite").path(filePath)
C. itemsDf.write(filePath, mode="上書き")
D. itemsDf.write.mode("上書き").path(ファイルパス)
E. itemsDf.write().parquet(filePath, mode="overwrite")
Answer: A (メーッセージを残す)
説明
itemsDf.write.mode("上書き").parquet(filePath)
正しい！itemsDf.write は、pyspark.sql.DataFrameWriter インスタンスを返します。その上書き動作は、モード設定を通じて、または parquet() コマンドに mode="overwrite" を渡すことによって変更できます。
この問題を解決するために寄木細工の形式は規定されていませんが、parquet() は Spark を開始してデータをディスクに書き込むための有効な演算子です。
itemsDf.write.mode("上書き").path(ファイルパス)
いいえ。pyspark.sql.DataFrameWriter インスタンスには path() メソッドがありません。
itemsDf.write.option("parquet").mode("overwrite").path(filePath)
不正解です。上記を参照してください。さらに、option() メソッドを介してファイル形式を渡すことはできません。
itemsDf.write(filePath, mode="上書き")
間違い。残念ながら、これは単純すぎます。itemsDf.write を呼び出して、DataFrame の DataFrameWriter へのアクセスを取得する必要があります。これにより、Spark データをディスクに書き込む方法を制御するためのさらなるメソッドを適用できます。ただし、引数を itemsDf.write に直接渡すことはできません。
itemsDf.write().parquet(filePath, mode="上書き")
間違い。上記を参照。
詳細: pyspark.sql.DataFrameWriter.parquet - PySpark 3.1.2 ドキュメント 静的ノートブック | 動的ノートブック: テスト 3 を参照

最新問題: **46**
Spark の実行階層の最上位レベルはどれですか?
A. タスク
B. ステージ
C. 仕事
D. スロット
E. 執行者
Answer: C (メーッセージを残す)

最新問題: **47**
タスクについて説明しているのは次のうちどれですか?
A. タスクは、変換に応じてドライバーからエグゼキューターに送信されるコマンドです。
B. タスクはジョブを DAG に変換します。
C. タスクはスロットの集合です。
D. タスクは行のコレクションです。
E. タスクはドライバーによってエグゼキューターに割り当てられます。
Answer: (解答を表示する**)**
説明
タスクはドライバーによって実行プログラムに割り当てられます。
正しい！言い換えると、エグゼキュータはドライバによって割り当てられたタスクを取得し、パーティション上で実行し、その結果をドライバに報告します。
タスクはジョブを DAG に変換します。
いいえ、このステートメントは Spark 階層内の要素の順序を無視しています。Spark ドライバーはジョブを DAG に変換します。各ジョブは 1 つ以上のステージで構成されます。各ステージには 1 つ以上のタスクが含まれます。
タスクは行のコレクションです。
間違い。パーティションは行のコレクションです。タスクは行のコレクションとはほとんど関係がありません。どちらかというと、タスクは特定のパーティションを処理します。
タスクは、変換に応じてドライバーからエグゼキュータに送信されるコマンドです。
正しくない。変換は遅延評価されるため、Spark ドライバーは変換に応じてエグゼキュータに何も送信しません。したがって、Spark ドライバーは、アクションに応じてのみタスクを実行プログラムに送信します。
タスクはスロットの集合です。
いいえ。エグゼキュータにはタスクを処理するための 1 つ以上のスロットがあり、各スロットにタスクを割り当てることができます。

最新問題: **48**

次のコード ブロックのうち、DataFrame トランザクション Df の列 "value" の平均値をその列 storeId ごとにグループ化して示す DataFrame を返すものはどれですか?
A. transactionsDf.groupBy(col(storeId).avg())
B. transactionsDf.groupBy("storeId").avg(col("value"))
C. transactionsDf.groupBy("storeId").agg(avg("value"))
D. transactionsDf.groupBy("storeId").agg(average("value"))
E. transactionsDf.groupBy("値").average()
Answer: C (メーッセージを残す)
説明
この質問では、Spark での groupBy パターンと agg パターンの使用方法に関する知識をテストします。ドキュメントを参照すると、pyspark.sql.functions に Average() メソッドがないことがわかります。
静的ノート | 動的ノートブック: テスト 2 を参照

最新問題: **49**
次のコード ブロックは、DataFrame トランザクション Df のすべての列のさまざまな集計統計 (各列の値の標準偏差と最小値を含む) を表示するのはどれですか?
A. transactionDf.summary()
B. transactionsDf.agg("カウント", "平均", "stddev", "25%", "50%", "75%", "min")
C. transactionsDf.summary("count", "mean", "stddev", "25%", "50%", "75%", "max").show()
D. transactionsDf.agg("count", "mean", "stddev", "25%", "50%", "75%", "min").show()
E. transactionsDf.summary().show()
Answer: E (メーッセージを残す)
説明
DataFrame.summary() コマンドは、DataFrame の統計を迅速に計算するのに非常に実用的です。計算結果を表示するには、.show() を呼び出す必要があります。デフォルトでは、コマンドは標準偏差や最小値などのさまざまな統計を計算します (以下にリンクされているドキュメントを参照)。
summary() の括弧内に多くのオプションをリストした回答には、質問で求められている最小値が含まれていないことに注意してください。
DataFrame.agg() は値を集計する方法についてのより複雑な列固有の命令を期待しているため、agg() を含む回答オプションはここでは機能しません。
より詳しい情報：
- pyspark.sql.DataFrame.summary - PySpark 3.1.2 ドキュメント
- pyspark.sql.DataFrame.agg - PySpark 3.1.2 ドキュメント
静的ノート | 動的ノートブック: テスト 3 を参照

最新問題: **50**
RDD に関する次の記述のうち、間違っているものはどれですか?
A. RDD は単一のパーティションで構成されます。

B. 高レベルの DataFrame API は、低レベルの RDD API の上に構築されます。
C. RDD は不変です。
D. RDD は Resilient Distributed Dataset の略です。
E. RDD は、クエリの実行方法を Spark に正確に指示するのに最適です。
Answer: A (メーッセージを残す**)**
説明
RDD は単一のパーティションで構成されます。
まったく逆です。Spark は RDD を分割し、そのパーティションを複数のノードに分散します。

最新問題**: 51**
Spark の DataFrame に関する次の記述のうち、間違っているものはどれですか?
A. Spark の DataFrame は不変です。
B. Spark の DataFrame は Python の DataFrame と同等です。
C. DataFrame 内のデータは名前付き列に編成されます。
D. RDD は DataFrame の中核です。
E. DataFrame 内のデータは複数のチャンクに分割される場合があります。
Answer: (解答を表示する**)**
説明
Spark の DataFrame は、Python または R の DataFrame と同等です。
いいえ、それらは平等ではありません。似ているだけです。Spark と Python の主な違いは、Spark の DataFrame は分散されるのに対し、Python の DataFrame は分散されないことです。

最新問題**: 52**
クライアント実行モードとクラスター実行モードの違いを説明しているものは次のうちどれですか?
A. クラスター モードでは、ドライバーはワーカー ノードで実行されますが、クライアント モードでは、ドライバーはクライアント マシンで実行されます。
B. クラスター モードでは、ドライバーはエッジ ノードで実行されますが、クライアント モードでは、ドライバーはワーカー ノードで実行されます。
C. クラスター モードでは、各ノードが独自のエグゼキューターを起動しますが、クライアント モードでは、エグゼキューターはクライアント マシン上で排他的に実行されます。
D. クライアント モードでは、クラスター マネージャーはドライバーと同じホスト上で実行されますが、クラスター モードでは、クラスター マネージャーは別のノード上で実行されます。
E. クラスター モードでは、ドライバーはマスター ノード上で実行されますが、クライアント モードでは、ドライバーはクラウド内の仮想マシン上で実行されます。
Answer: A (メーッセージを残す**)**

説明
クラスター モードではドライバーはマスター ノードで実行され、クライアント モードではドライバーはクラウド内の仮想マシンで実行されます。
実行モードではワークロードがクラウドで実行されるかオンプレミスで実行されるかが指定されていないため、これは誤りです。
クラスター モードでは、各ノードが独自のエグゼキューターを起動しますが、クライアント モードでは、エグゼキューターはクライアント マシン上で排他的に実行されます。
どちらの場合もエグゼキュータはワーカー ノードで実行されるため、間違いです。
クラスター モードでは、ドライバーはエッジ ノードで実行されますが、クライアント モードでは、ドライバーはワーカー ノードで実行されます。
間違いです。クラスター モードでは、ドライバーはワーカー ノード上で実行されます。クライアント 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ます。
クライアント モードでは、クラスター マネージャーはドライバーと同じホスト上で実行されますが、クラスター モードでは、クラスター マネージャーは別のノード上で実行されます。
いいえ。どちらのモードでも、クラスター マネージャーは通常、ドライバーと同じホスト上ではなく、別のノード上にあります。ローカル実行モードではドライバーと同じホスト上でのみ実行されます。
詳細情報: Learning Spark、第 2 版、第 1 章、および Spark: 決定版ガイド、第 15 章 ()

最新問題: **53**
ファイルがその場所にまだ存在しない場合に、DataFrame itemsDf を avro 形式でその場所にサイレントに書き込むコード ブロックは次のどれですか?
A. itemsDf.write.avro(fileLocation)
B. itemsDf.write.format("avro").mode("ignore").save(fileLocation)
C. itemsDf.write.format("avro").mode("errorifexists").save(fileLocation)
D. itemsDf.save.format("avro").mode("ignore").write(fileLocation)
E. spark.DataFrameWriter(itemsDf).format("avro").write(fileLocation)
Answer: A (メーッセージを残す**)**
説明
この質問のコツは、DataFrameWriter の 『モード」を知ることです。モード無視は、ファイルがすでに存在する場合に無視し、そのファイルを置き換えませんが、エラーもスローしません。モード errorifexists はエラーをスローします。これは DataFrameWriter のデフォルト モードです。質問は 『いいえ」です。
DataFrame が存在しない場合は、DataFrame が 『サイレント」に書き込まれることを明示的に呼び出します。そのため、ファイルがすでに存在する場合に Spark からエラーが通知されるのを避けるために、ここで mode("ignore") を指定する必要があります。
『上書き」モードは、サイレントではありますが、既存のファイルを上書きするため、ここでは適切ではありません。これは質問が求めていることではありません。

Spark は SparkSession オブジェクトを参照しますが、そのオブジェクトは DataFrameWriter を提供しないため、spark.DataFrameWriter(itemsDf) で始まるオプションは機能しないことに注意してください。
ドキュメント (以下) でわかるように、DataFrameWriter は PySpark の SQL API の一部ですが、SparkSession API の一部ではありません。
より詳しい情報：
DataFrameWriter: pyspark.sql.DataFrameWriter.save - PySpark 3.1.1 ドキュメント
SparkSession API: Spark SQL - PySpark 3.1.1 ドキュメント 静的ノートブック | 動的ノートブック: テスト 1 を参照

最新問題**: 54**
次のコード ブロックのうち、タイムスタンプ型の列日付を持つ新しい 1 列 2 行の DataFrame dfDates を作成するものはどれですか?
A. 1.dfDates = dark.createDataFrame(["23/01/2022 11:28:12","24/01/2022 10:58:34"], ["日付"])
2.dfDates = dfDates.withColumn("日付", to_timestamp("dd/MM/yyyy HH:mm:ss", "日付"))
B. 1.dfDates = spark.createDataFrame([("23/01/2022 11:28:12",),("24/01/2022 10:58:34",)], ["日付"])
2.dfDates = dfDates.withColumnRenamed("日付", to_timestamp("日付", "yyyy-MM-dd HH:mm:ss"))
C. 1.dfDates = spark.createDataFrame([("23/01/2022 11:28:12",),("24/01/2022 10:58:34",)], ["日付"])
2.dfDates = dfDates.withColumn("日付", to_timestamp("日付", "dd/MM/yyyy HH:mm:ss"))
D. 1.dfDates = dark.createDataFrame(["23/01/2022 11:28:12","24/01/2022 10:58:34"], ["日付"])
2.dfDates = dfDates.withColumnRenamed("日付", to_datetime("日付", "yyyy-MM-dd HH:mm:ss"))
E. 1.dfDates = spark.createDataFrame([("23/01/2022 11:28:12",),("24/01/2022 10:58:34",)], ["日付"])
Answer: C (メーッセージを残す**)**
説明
この質問は難しいです。ここで知っておくべき重要な点が 2 つあります。
まず、createDataFrame の構文です。ここでは、[(1,), (2,)] のようなタプルのリストが必要です。Python でタプルを定義するには、その中に項目が 1 つしかない場合、Python がそれを単なる通常のかっこではなくタプルとして解釈できるように、項目の後にカンマを置くことが重要です。
次に、to_timestamp 構文を理解する必要があります。詳細については、以下のリンク先のドキュメントを参照してください。
適切な対策として、間違ったオプションが間違っている理由を詳しく調べてみましょう。

dfDates = dark.createDataFrame([("23/01/2022 11:28:12",),("24/01/2022 10:58:34",)], ["date"]) このコード スニペットは次のことを行います。日付列のデータ型がタイムスタンプではなく文字列であることを除いて、質問で求められるものはすべて含まれています。スキーマが指定されていない場合、Spark は文字列データ型をデフォルトとして設定します。
dfDates = スパーク.createDataFrame(["23/01/2022 11:28:12","24/01/2022 10:58:34"], ["日付"]) dfDates = dfDates.withColumn("日付", to_timestamp("dd/MM/yyyy HH:mm:ss", "date")) このコマンドの最初の行で、Spark は次のエラーをスローします: TypeError: タイプのスキーマを推論できません:
<クラス 'str'>。これは、Spark が行情報の検索を期待しているのではなく、文字列を検索するためです。データをタプルとして指定する必要があるのはこのためです。幸いなことに、Spark ドキュメント (以下にリンク) には、正しい軌道に乗るのに役立つ DataFrame の作成例が多数示されています。
dfDates = スパーク.createDataFrame([("23/01/2022 11:28:12",),("24/01/2022 10:58:34",)], ["日付"]) dfDates = dfDates。withColumnRenamed("date", to_timestamp("date", "yyyy-MM-dd HH:mm:ss")) この回答の問題は、演算子 withColumnRenamed が使用されていることです。この演算子は単に列の名前を変更するだけですが、実際の内容を変更する権限はありません。このため、代わりに withColumn を使用する必要があります。さらに、日付形式 yyyy-MM-dd HH:mm:ss は、実際のタイムスタンプの形式 「23/01/2022 11:28:12」を反映しません。
dfDates = スパーク.createDataFrame(["23/01/2022 11:28:12","24/01/2022 10:58:34"], ["日付"]) dfDates = dfDates.withColumnRenamed("日付", to_datetime("date", "yyyy-MM-dd HH:mm:ss")) ここでは、withColumn の代わりに withColumnRenamed が使用されます (上記を参照)。さらに、行は正しく表現されていません。括弧を使用してタプルとして記述する必要があります。最後に、ここでは日付形式さえオフになっています (上記を参照)。
詳細: pyspark.sql.functions.to_timestamp - PySpark 3.1.2 ドキュメントおよび pyspark.sql.SparkSession.createDataFrame - PySpark 3.1.1 ドキュメント 静的ノートブック | 動的ノートブック: テスト 2 を参照

最新問題**: 55**
次のコード ブロックのうち、DataFrame itemsDf をエグゼキュータ メモリに保存し、メモリが不足している場合はシリアライズしてディスクに保存するものはどれですか?
A. itemsDf.persist(StorageLevel.MEMORY_ONLY)
B. itemsDf.cache(StorageLevel.MEMORY_AND_DISK)
C. itemsDf.store()
D. itemsDf.cache()
E. itemsDf.write.option('宛先', 'メモリ').save()
Answer: D (メーッセージを残す**)**
説明

この疑問を解決する鍵は、デフォルトで、cache() が値をメモリに保存し、メモリが不足しているパーティションをディスクに書き込むことを知る (またはドキュメントを読む) ことです。persist() はまったく同じ動作を実現できますが、ここにリストされている StorageLevel.MEMORY_ONLY オプションでは実現できません。また、cache() には引数がないことにも注意してください。
ドキュメントでストレージ レベルの情報を見つけるのが難しい場合は、ここでいくつかの点を明らかにしているこの学生 Q&A スレッドも参照してください。
静的ノート | 動的ノートブック: テスト 2 を参照

最新問題: **56**
次のコード ブロックのうち、列 productId の値が一意である行のみを DataFrame トランザクション Df から返しますか?
A. transactionsDf.distinct("productId")
B. transactionsDf.dropDuplicates(subset=["productId"])
C. transactionsDf.drop_duplicates(subset="productId")
D. transactionsDf.unique("productId")
E. transactionsDf.dropDuplicates(subset="productId")
Answer: B (メーッセージを残す)
説明
質問ではここで unique() というメソッドを使用することが示唆されていますが、そのメソッドは実際には PySpark に存在しません。PySparkでは、これはdistinct()と呼ばれます。ただし、distinct() を使用すると、特定の列の一意の値を除外できるため、このメソッドはここで使用するのには適切ではありません。
ただし、ここでは行全体を返したいと考えています。したがって、重要なのは、dropDuplicates を subset キーワード パラメータとともに使用することです。DropDuplicates のドキュメントの例では、リストとともにサブセットを使用する必要があることが示されています。そして、これがまさにこの質問を解決する鍵です。productId 列は、たとえ単一の列であっても、リスト内のサブセット引数に入力する必要があります。
詳細: pyspark.sql.DataFrame.dropDuplicates - PySpark 3.1.1 ドキュメント 静的ノートブック | 動的ノートブック: テスト 1 を参照

最新問題: **57**
次のコード ブロックのうち、列 itemId とtransactionId をそれぞれ結合キーとして使用して、DataFrame itemsDf と DataFrametransactionDf の間の内部結合を実行するのはどれですか?
A. itemsDf.join(transactionsDf, "inner", itemsDf.itemId ==transactionsDf.transactionId)
B. itemsDf.join(transactionsDf, itemId ==transactionId)
C. itemsDf.join(transactionsDf, itemsDf.itemId ==transactionsDf.transactionId, "inner")
D. itemsDf.join(transactionsDf, "itemsDf.itemId ==transactionDf.transactionId", "inner")
E. itemsDf.join(transactionsDf,col(itemsDf.itemId) ==col(transactionsDf.transactionId))

Answer: C (メーッセージを残す)
説明
詳細: pyspark.sql.DataFrame.join - PySpark 3.1.2 ドキュメント
静的ノート | 動的ノートブック: テスト 2 を参照

最新問題**: 58**
エグゼキュータとして動作する各 JVM をタスク実行スロットのプールとみなすことができると仮定した場合、エグゼキュータに関する次の記述のうち、正しいものはどれですか?
A. スロットはエグゼキュータの別名です。
B. タスクよりも実行者の数が少なくなければなりません。
C. エグゼキュータは単一のコア上で実行されます。
D. タスクよりも多くのスロットが必要です。
E. タスクはスロットを介して並行して実行されます。
Answer: E (メーッセージを残す)
説明
タスクはスロットを介して並行して実行されます。
正しい。この仮定を考慮すると、エグゼキュータには、spark.executor.cores / spak.task.cpus という式で定義される 1 つ以上の 「スロット」があります。エグゼキュータのリソースがスロットに分割されているため、各タスクがスロットを占有し、複数のタスクを並行して実行できます。
スロットはエグゼキューターの別名です。
いいえ、スロットはエグゼキューターの一部です。
エグゼキューターは単一のコア上で実行されます。
いいえ、エグゼキュータは複数のコアを占有することができます。これは、spark.executor.cores オプションによって設定されます。
タスクよりも多くのスロットが必要です。
いいえ、スロットはタスクを処理するだけです。複数のタスクに対して 1 つのスロットのみがあり、一度に 1 つのタスクを処理するシナリオを想像することもできます。確かに、これは Spark が本来使用すべき用途、つまり複数のコアとマシンにわたる分散データ処理で多くのタスクを並行して実行することとは逆です。
タスクよりも実行者の数を少なくする必要があります。
いいえ、そのような要件はありません。
詳細情報: Spark アーキテクチャ | 分散システム アーキテクチャ (https://bit.ly/3x4MZZt)

最新問題**: 59**
遺言執行者に関する次の記述のうち、正しいものはどれですか?
A. エグゼキュータはドライバによって起動されます。
B. デフォルトでは、アプリケーションの完了時にエグゼキュータが停止します。
C. 各ノードは単一のエグゼキュータをホストします。

D. エグゼキュータはデータをメモリのみに保存します。
E. エグゼキュータは複数のアプリケーションにサービスを提供できます。
Answer: B (メーッセージを残す**)**
説明
デフォルトでは、エグゼキュータはアプリケーションの完了時に停止します。
正しい。エグゼキュータは、アプリケーションの存続期間中のみ存続します。
注目すべき例外は、動的リソース割り当てが有効になっている場合です (デフォルトでは有効ではありません)。動的リソース割り当てを有効にすると、アプリケーションが完了したかどうかに関係なく、エグゼキュータはアイドル状態になると終了されます。
エグゼキュータは複数のアプリケーションにサービスを提供できます。
間違い。エグゼキューターは常にアプリケーションに固有です。アプリケーションが完了すると終了します (例外は上記を参照)。
各ノードは単一のエグゼキュータをホストします。
いいえ。各ノードは 1 つ以上のエグゼキュータをホストできます。
エグゼキュータはデータをメモリにのみ保存します。
いいえ。Executor はデータをメモリまたはディスクに保存できます。
エグゼキュータはドライバによって起動されます。
正しくない。エグゼキュータは、ドライバに代わってクラスタ マネージャによって起動されます。
詳細情報: ジョブ スケジューリング - Spark 3.1.2 ドキュメント、Spark クラスターでのアプリケーションの実行方法 | Spark アプリケーションの構造 | InformIT と初心者向けの Spark 用語。このブログは、いくつかの問題を解決することを目的としています。マゲスワラン D 著 | 中くらい

最新問題**: 60**
以下に表示されているコード ブロックにはエラーが含まれています。コード ブロックは、列 predError、productId、および value を除く DataFrametransactionDf のすべての列を返すことを目的としています。エラーを見つけてください。
DataFrame トランザクション Df の抜粋:
transactionsDf.select(~col("predError"), ~col("productId"), ~col("value"))
A. 選択演算子はドロップ演算子に置き換える必要があり、ドロップ演算子の引数は列名 predError、productId、および値を Col 演算子でラップする必要があるため、drop(col(predError),col(のように表現する必要があります) productId)、col(値))。
B. 選択演算子を選択解除演算子に置き換える必要があります。
C. select 演算子の列名は文字列であってはならず、col 演算子で囲む必要があるため、select(~col(predError), ~col(productId), ~col(value)) のように表現する必要があります。
D. 選択演算子はドロップ演算子に置き換える必要があります。

E. 選択演算子はドロップ演算子で置き換える必要があり、ドロップ演算子の引数は文字列としての列名 predError、productId、および value である必要があります。
（正しい）
Answer: E (メーッセージを残す**)**
説明
正しいコードブロック:
transactionsDf.drop("predError", "productId", "value")
静的ノート | 動的ノートブック: テスト 1 を参照

最新問題**: 61**
以下に表示されているコード ブロックにはエラーが含まれています。コード ブロックは、DataFrames itemsDf とtransactionsDf のデータを結合し、列 itemId の値が DataFrametransactionsDf の列transactionsId の値と一致する DataFrame itemsDf のすべての行を表示する必要があります。エラーを見つけてください。
コードブロック:
itemsDf.join(itemsDf.itemId==transactionsDf.transactionId)
A. 結合ステートメントが不完全です。
B. join の代わりに Union メソッドを使用する必要があります。
C. 結合方法が不適切です。
D. 結合の代わりにマージ方法を使用する必要があります。
E. 結合式の形式が不正です。
Answer: A (メーッセージを残す**)**
説明
正しいコードブロック:
itemsDf.join(transactionsDf, itemsDf.itemId==transactionsDf.transactionId) join ステートメントが不完全です。
正しい！DataFrame.join() のドキュメント (以下にリンク) を見ると、join の最初の引数が結合される DataFrame であることがわかります。この最初の引数がコード ブロックにありません。
結合方法が不適切です。
いいえ。デフォルトでは、DataFrame.join() は内部結合を使用します。この方法は、質問で説明されているシナリオに適しています。
結合式の形式が正しくありません。
正しくない。結合式 itemsDf.itemId==transactionsDf.transactionId は正しい構文です。
結合の代わりにマージ方法を使用する必要があります。
間違い。PySpark には DataFrame.merge() メソッドはありません。
結合の代わりに結合メソッドを使用する必要があります。
間違い。DataFrame.union() は行をマージしますが、質問で要求されているように列はマージしません。

詳細: pyspark.sql.DataFrame.join - PySpark 3.1.2 ドキュメント、pyspark.sql.DataFrame.union - PySpark 3.1.2 ドキュメント 静的ノートブック | 動的ノートブック: テスト 3 を参照

最新問題: **62**
次のコード ブロックのうち、列 predError によって降順に並べ替えられた DataFrame トランザクション Df を返し、欠損値が最後に表示されるのはどれですか?
A. transactionsDf.sort(asc_nulls_last("predError"))
B. transactionsDf.orderBy("predError").desc_nulls_last()
C. transactionsDf.sort("predError", ascending=False)
D. transactionsDf.desc_nulls_last("predError")
E. transactionsDf.orderBy("predError").asc_nulls_last()
Answer: C (メーッセージを残す)
説明
transactionsDf.sort("predError", ascending=False)
正しい！DataFrame.sort() を使用し、ascending=False に設定すると、DataFrame は指定された列によって降順に並べ替えられ、すべての欠落値が最後に配置されます。ここでは答えとしてリストされていませんが、代わりの方法
は、transactionsDf.sort(desc_nulls_last("predError")) です。
transactionsDf.sort(asc_nulls_last("predError"))
正しくない。これは有効な構文ですが、DataFrame は列 predError で降順ではなく昇順に並べ替えられ、欠損値が最後に配置されます。
transactionsDf.desc_nulls_last("predError")
違います。これは無効な構文です。Spark API には DataFrame.desc_nulls_last() メソッドはありません。ただし、Spark 関数 desc_nulls_last() があります (リンクは以下を参照)。
transactionsDf.orderBy("predError").desc_nulls_last()
いいえ。transactionsDf.orderBy("predError") は正しい構文で (ただし、DataFrame を列 predError で昇順にソートします)、DataFrame を返しますが、Spark API にはメソッド

DataFrame.desc_nulls_last() がありません。ただし、Spark 関数 desc_nulls_last() があります (リンクは以下を参照)。
transactionsDf.orderBy("predError").asc_nulls_last()
正しくない。Spark API にはメソッド DataFrame.asc_nulls_last() がありません (上記を参照)。
詳細: pyspark.sql.functions.desc_nulls_last - PySpark 3.1.2 ドキュメントおよび pyspark.sql.DataFrame.sort - PySpark 3.1.2 ドキュメント
(https://bit.ly/3g1JtbI 、https://bit.ly/2R90NCS)) 静的ノートブック | 動的ノートブック: テスト 1 (https://flrs.github.io/spark_practice_tests_code/#1/32.html) を参照してください。https://bit.ly/sparkpracticeexams_import_instructions)

最新問題: **63**
Spark のメモリ管理方法を説明しているものは次のうちどれですか?
A. Spark は予約されたシステム メモリのサブセットを使用します。
B. ストレージ メモリは、DataFrame から派生したパーティションのキャッシュに使用されます。
C. ガベージ コレクションの一般的なルールとして、Spark は、少数の大きなオブジェクトよりも多数の小さなオブジェクトに対してパフォーマンスが向上します。
D. シリアル化を無効にすると、Spark アプリケーションのメモリ フットプリントが大幅に削減される可能性があります。
E. Spark のメモリ使用量は、実行、トランザクション、ストレージの 3 つのカテゴリに分類できます。
Answer: B (メーッセージを残す**)**
説明
Spark のメモリ使用量は、実行、トランザクション、ストレージの 3 つのカテゴリに分類できます。
いいえ、実行か保管のどちらかです。
ガベージ コレクションの一般的なルールとして、Spark は、少数の大きなオブジェクトよりも多数の小さなオブジェクトに対してパフォーマンスが向上します。
いいえ、Spark のガベージ コレクションは、多数の小さなオブジェクトよりも少数の大きなオブジェクトの方が高速に実行されます。
シリアル化を無効にすると、Spark アプリケーションのメモリ フットプリントが大幅に削減される可能性があります。
逆も当てはまります。シリアル化するとメモリ フットプリントは削減されますが、パフォーマンスに悪影響を及ぼす可能性があります。
Spark は、予約されたシステム メモリのサブセットを使用します。
いいえ、予約されたシステム メモリは Spark メモリとは別のものです。予約されたメモリには、Spark の内部オブジェクトが保存されます。

詳細情報: チューニング - Spark 3.1.2 ドキュメント、Spark メモリ管理 | 分散システムアーキテクチャ、Learning Spark、第 2 版、第 7 章