Spark internal

Spark SQL vs Dataframe API

Performance

Whether you write the code using DataFrame API or Spark Sql API , there is no significant difference in terms of performance because both the dataframe api and spark sql api are abstractions on top of RDD (Resilient Distributed Dataset).

Error at compile vs runtime

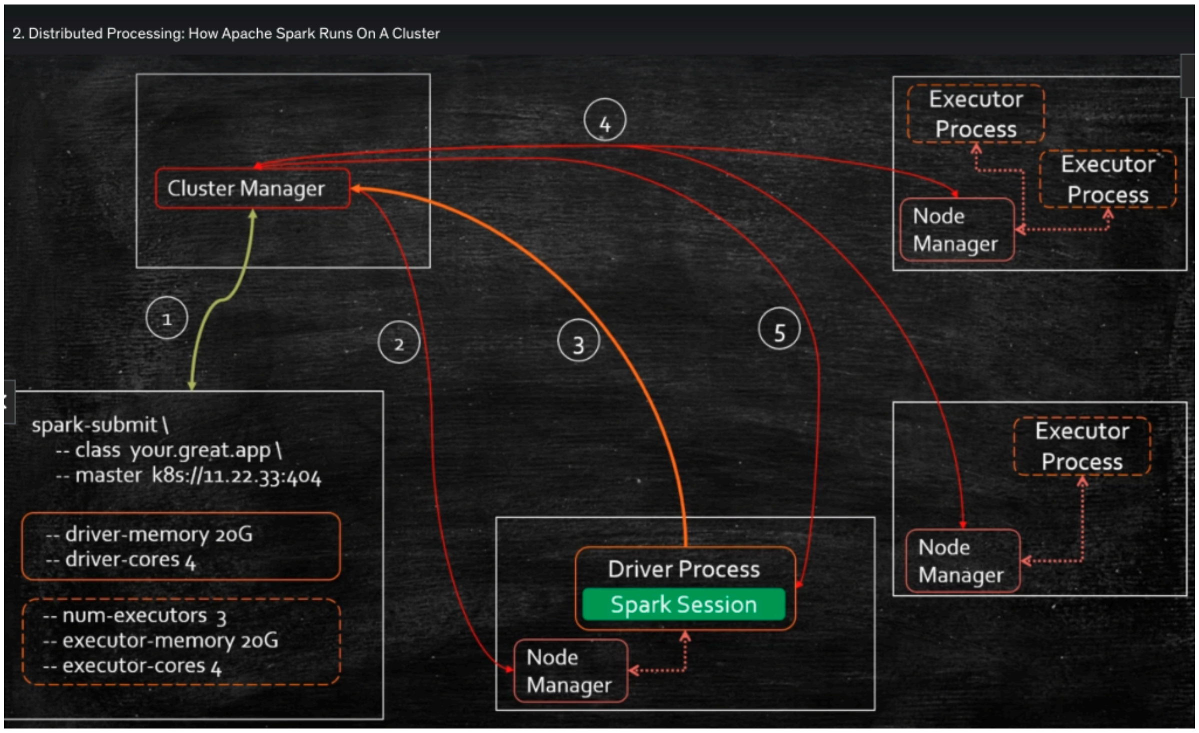
In your Spark SQL string queries, you won't know a syntax error until runtime (which could be costly), whereas in DataFrames syntax errors can be caught at compile time.

Spark API has more control of memory and CPU usage than Spark SQL

The main advantage of dataframe api is that you can use dataframe optimized functions, for example : cache() , in general you will have more control of the execution plan.

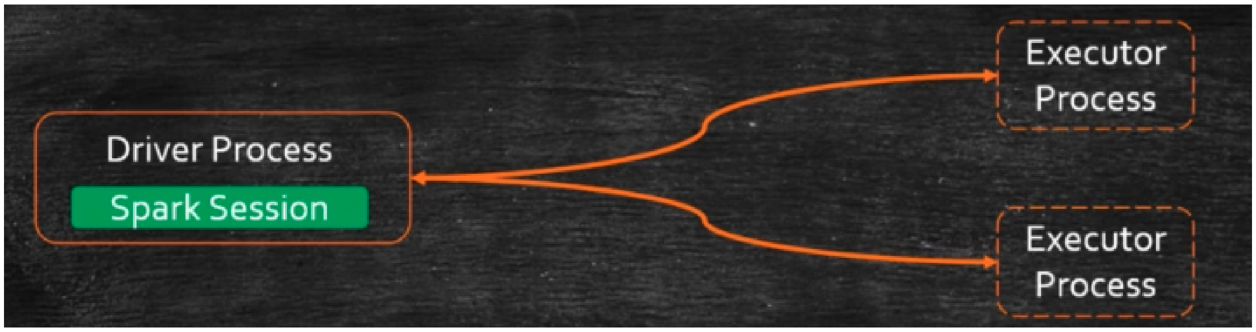
Architecture

- Driver != cluster managerやで
    - Driverも一つのnode managerによってmanageされる。
- Executorってのはprocessのことやから、node managerがいくつものexecutorをrunする。
-
-

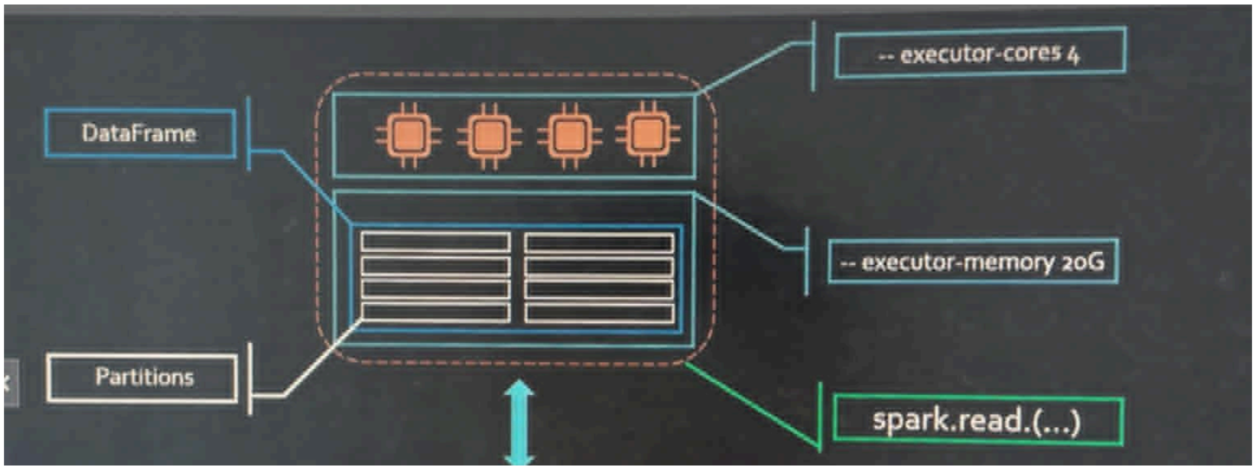


```
spark-submit \
  -- class your.great.app \
  -- master k8s://11.22.33:404
-- driver-memory 20G
-- driver-cores 4
-- num-executors 3
-- executor-memory 20G
-- executor-cores 4
```

- Driver is responsible for
    - 1. the resource management,
    - 2.dynamically increase/decrease resource consumption
    - 3.distribute and schedule work across executor processes.
    - 4.Reacts to failures.
    - 5.Monitors progress
- Executor is responsible for
    - 1.executor tasks
    - 2.report to the driver.



Executor architecture

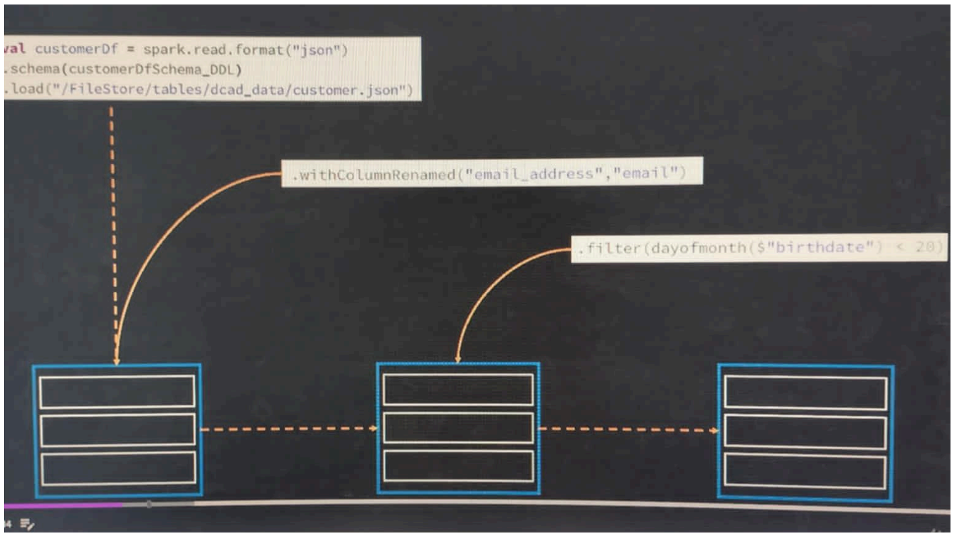


Dataframe architecture in executor

Everytime you run a function/transformation, the executor creates a new dataframe. Why?
This is because

- dataframe is an immutable object.
- Dataframe is distributed across multiple executors.

ちなみに
You can run
df.explain("formatted") to check execution graph.



```
val customerDf = spark.read.format("json")
.schema(customerDfSchema_DDL)
.load("/FileStore/tables/dcad_data/customer.json")
```

```
.withColumnRenamed("email_address","email")
```

```
.filter(dayofmonth($"birthdate") < 20)
```

Actions in dataframe

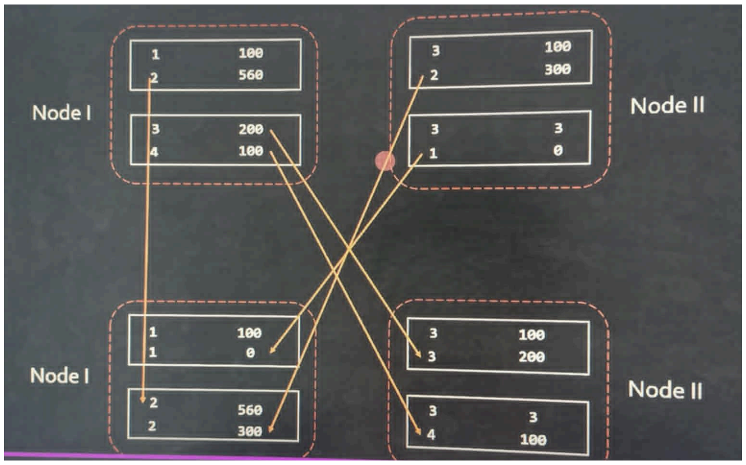
Action triggers the computation of all the transformation applied on a dataframe.
There are 3 types of actions

- 1.view data
- 2.collect data ex) take, collect, takeAsList
- 3.save data ex)save, saveAsTable

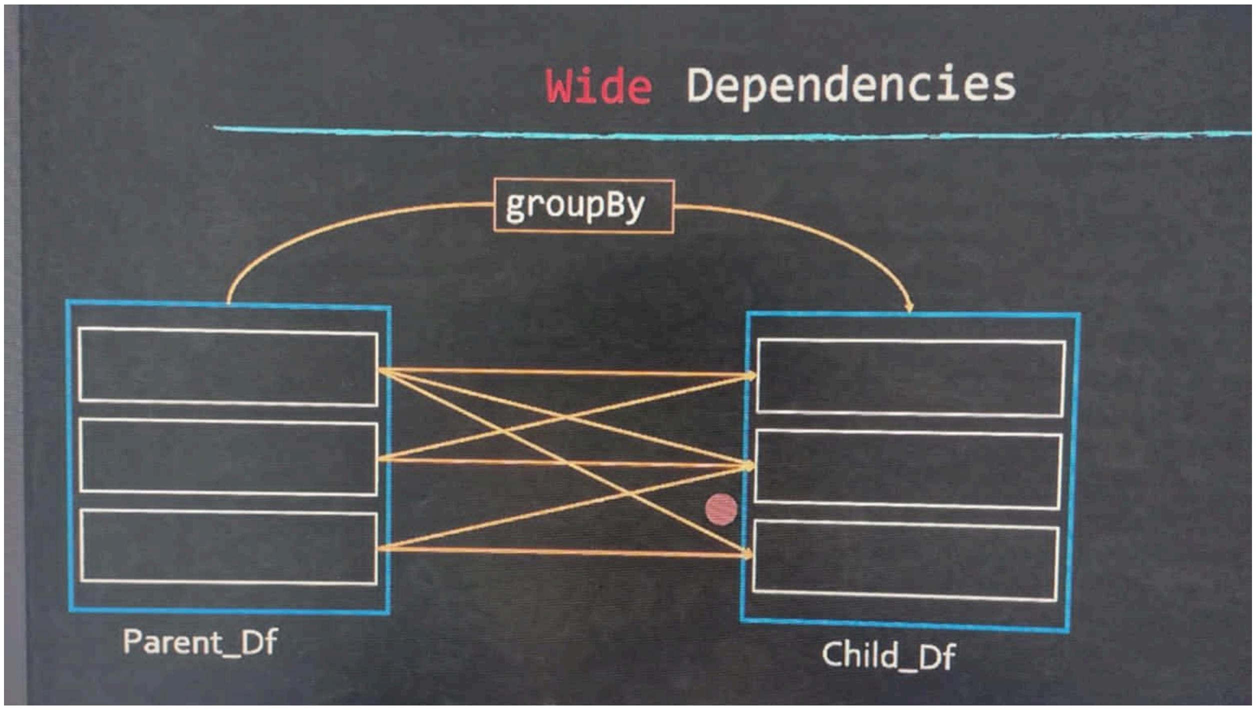
Shuffling

Shuffling should be avoided whenever possible. But shuffling happens at the group by. This is because node needs to gather same key in the same node to calculate aggregation.
Shuffle can happen when you run

- Group by, order by
- Join
- Distinct
- Repartition/coalesce to resize a dataframe partition numbers.



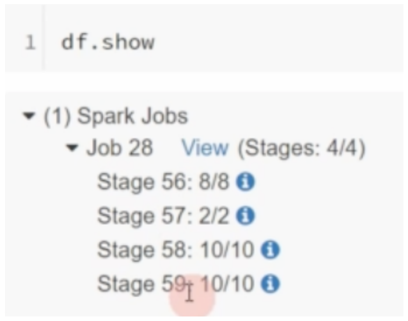
When keys in a partition in a dataframe are used in multiple partitions in the next dataframe, it's called Wide Dependencies.



How to avoid shuffle

- Avoidではないんだけど、partitionを小さくするのもoptimizationの一つ
    - ex) spark.conf.set("spark.sql.shuffle.partitions", 50)
    - Default is 50.
- Broadcast joinするとshuffleが避けられるのはなぜか。
    - Small dimension is assigned to every partition in a dataframe, so you don't need to group the same key together anymore.
    -

DataFrame repartition vs partitionBy() vs coalesce()

- Repartitionでdataframe partitionはresizeされる。
    - If you want to change the file format you can:
        - ex) df.repartititon(8).write().format(“json”)
    - Note: you need to specify the mode() always.
        - ex) df.repartititon(8).write().mode(SaveMode.Overwrite)
- partitionBy(column_name: string)
    - repartitionがresizeが目的なのに対して、partitionBy()はpartition keyを変えることができる。
    - ex) df.repartititon(8).write().partitionBy(col(“example”)).format(“json”).mode(SaveMode.Overwrite).save()
- coalesce() changes the number of partitions in a single node. It will NOT trigger a shuffle.
- Repartition vs coalesce()
    - Both of them returns a new DF.
    - Repartition requires SHUFFLE, but coalesce() does not.
    - This is the result of df.show() after repartition. It has 4 stages.
    - 

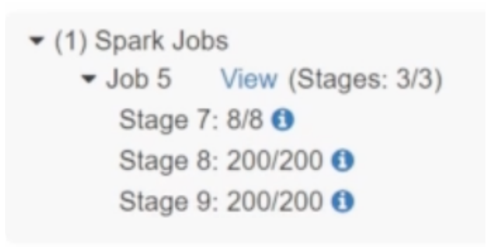
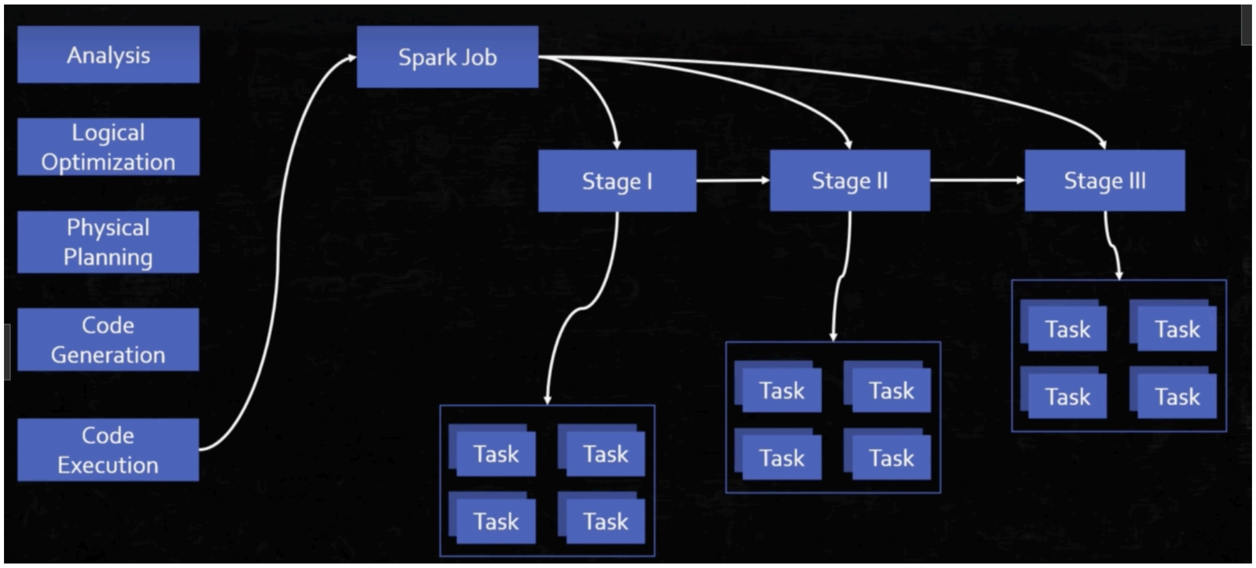
```
df.show
```

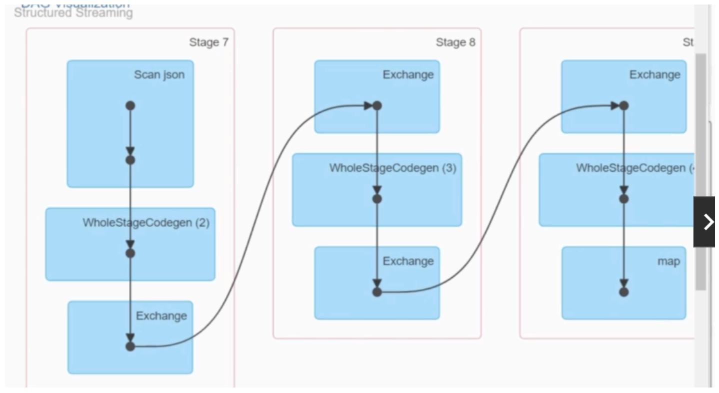
```
▾ (1) Spark Jobs
    ▾ Job 28   View (Stages: 4/4)
        Stage 56: 8/8
        Stage 57: 2/2
        Stage 58: 10/10
        Stage 59: 10/10
```

    - Coalesce is a narrow dependency, when one parent produces one child node.
        - ex) parent DF had 100 partitions in DF. child DF only has 10 DF, then each child partition came from 10 partitions in the parent DF.
    - Q: when do we prefer to use repartition() ? It seems coalesce() is always preferred as there is no shuffle.

Query Planning

- High level planning flow
    - Stages are kind of partitioned worker. stage1がscanでstage2とかがexchange/shuffleをする。scanは一つのtaskができるので、number of tasksが少ない。逆にshuffleではmany tasksがDFの中でpartitionされるので、number of tasks(=partition)が増える。

- 
  (1) Spark Jobs
  Job 5 View (Stages: 3/3)
  Stage 7: 8/8
  Stage 8: 200/200
  Stage 9: 200/200
    - だとしたら、serializeしないといけないのでは? Dependencyがあるから。動画見るとやっぱりserializeされてるっぽい。でもstageの中でtask distributionできる。
  - Number of stages depend on the number of shuffle necessary.
  - Each stage consists of tasks. Number of tasks == number of cores in node CPU setting.
- 

- To view the planning, run df.explain(True)
  - ここでいうexchangeはshuffleのこと
  -
  - 

- To view the generated java code from python, run df.explain("codegen")
  -
  -

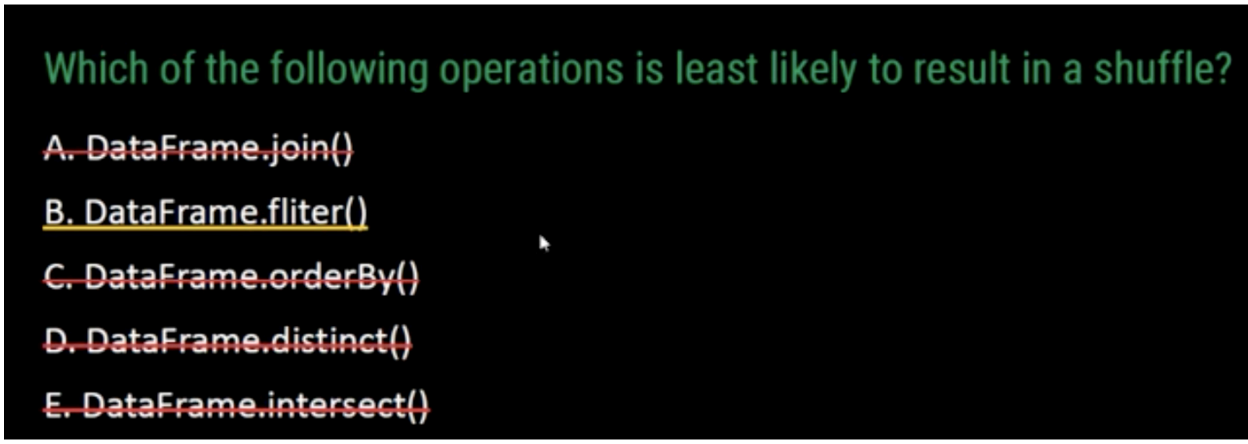
Syntax in dataframe

- df.where(col(“col_A”).isNull) returns rows, but
- df.where(col(“col_A”) == null) returns **nothing**.
- Date/timestamp type
    - col(“unix_seconds”).cast(“Date”)
        - This does not work because you will miss unix_seconds ‘s seconds.
    - Instead, use col(“unix_seconds”).cast(“Timestamp”)
- df.outerJoin() caused an error, why?
    - There is not outerJoin()
    - Instead, run df.join(df_B, col(), “outer”)
- df.join (df_b, col, “cross”) caused an error, why?
    - df.crossjoin() should be used instead.
- What is the root cause of this error? df.write().path(filePath)
    - Error1: There is no path() function.
        - Correct way is df.write.json(filePath)
    - Error2: there is no write(). Use df.write.
-

Exam questions

- Is Spark driver fault tolerant ? What happens if it fails?
    - No, the Spark driver is not fault-tolerant. If the Spark driver fails, the entire application will crash. Here's what happens if the Spark driver fails
    - Spark does not have built-in mechanisms for automatic driver recovery. While executors can fail and be retried, a driver failure typically requires manual intervention to restart the application unless external systems are involved.
- What is spark driver?
    - Ans: Spark driver is the program space where the Spark’s main method runs. ✅coordinating the Spark entire application.
- What is the relationship between node and executor?
    - An executor is a process running on a node.
- explain the difference between slots and tasks in Spark, and explain why spark job will not run efficiently if the number of slots is higher than the one of tasks.
    - **Slots (=physical hardware work unit)** represent the available capacity for parallel execution on the executors. Each executor has a fixed number of slots, which is typically determined by how many cores or threads it has.
    - **Tasks (=logical work unit)** represent the smallest units of work that are distributed across executors for parallel processing.

    - Relationship: A **task is executed in a slot**. A Spark job is divided into stages, and within each stage, there are multiple tasks corresponding to partitions of data.
    - Therefore, if number of slots > number of tasks, that means there exists some idle slots.
    - なので、Node >> Executor >> Slot (hardware limit, CPU) >> Task (logical work unit)
- In your words, what is lazy evaluation?
    - A process is lazily evaluated if its execution does NOT start until it is put into action by some trigger.
- How do we define data frame partition in Spark? What is the default setting?
    - Default Partitioning (Based on Input Source)
        - When you read data from a file or a data source like HDFS, S3, or a database, Spark automatically partitions the data based on the input source.
        - For example, when reading from HDFS or S3, Spark partitions the DataFrame based on the block size of the file (usually 128 MB or 256 MB). Each block is loaded as a partition.
    - Using repartition()
    -
    - You can manually repartition a DataFrame using the repartition() function. This function allows you to increase or decrease the number of partitions.
        - val df = spark.read.csv("data.csv")
        - val repartitionedDF = df.repartition(10) // Repartition into 10 partitions
    - If I want to partition DF into 12 where DF is already partitioned by 8, do you use df.repartition(12) or df.coalesce(12)?
        - repartition(12) because coalesce can only reduce the number.
    - Suppose you have a function assess() and let’s say you want to run it on each row. Which option is correct?
        - Option1: [assess(row) for row in df.collect()]
        - Option2: [assess(row) for row in df]
    - If I want to store data in a dataframe into Spark's memory only, then should I use cache() or persist() ? If I understand correctly, cache() cached data into memory and disk. But I want to store it only in memory.
        - If you want to store your DataFrame only in memory, you need to use: persist(StorageLevel.MEMORY_ONLY).
    - Does union() shuffle data ?
        - No.s
- What’s wrong with this?
    - df.wrte.partitionBy(col(“col_A”)).path(file_path)
        - partitionBy does not require col()
    - df.write.option(“parquet”).partitionBy(“col_A”).path(file_path)
        - Option never has parquet format.
            - まず、.path()なんてないぞ。

      - df.write.partitionBy(“col_A”).parquet(file_path)が正解
  - df.duplicates().drop()
    - Duplicatesなんてないぞ。
    - 正しくはdf.dropDuplicates()
  - ds.withColumn(“col_A”, substr(col(“col_B”) , 0 , 2))
    - substr() なんてないぞ。col(“col_B”).substr(0, 2)が正解。
- めちゃいい問題。
  - 

Which of the following operations is least likely to result in a shuffle?

~~A. DataFrame.join()~~

B. DataFrame.fliter()

~~C. DataFrame.orderBy()~~

~~D. DataFrame.distinct()~~

~~E. DataFrame.intersect()~~