

Spark Architecture

Basics

Spark!

- A *cluster*, or group, of computers, pools the resources of many machines together, giving us the ability to use all the cumulative resources as if they were a single computer.
- A group of machines alone is not powerful, you need a framework to coordinate work across them.
- Spark does just that, managing and coordinating the execution of tasks on data across a cluster of computers.

Spark!

- The cluster of machines that Spark will use to execute tasks is managed by a cluster manager like...
 1. Spark's standalone cluster manager,
 2. YARN,
 3. Mesos.
- We then submit Spark Applications to these cluster managers, which will grant resources to our application so that we can complete our work.

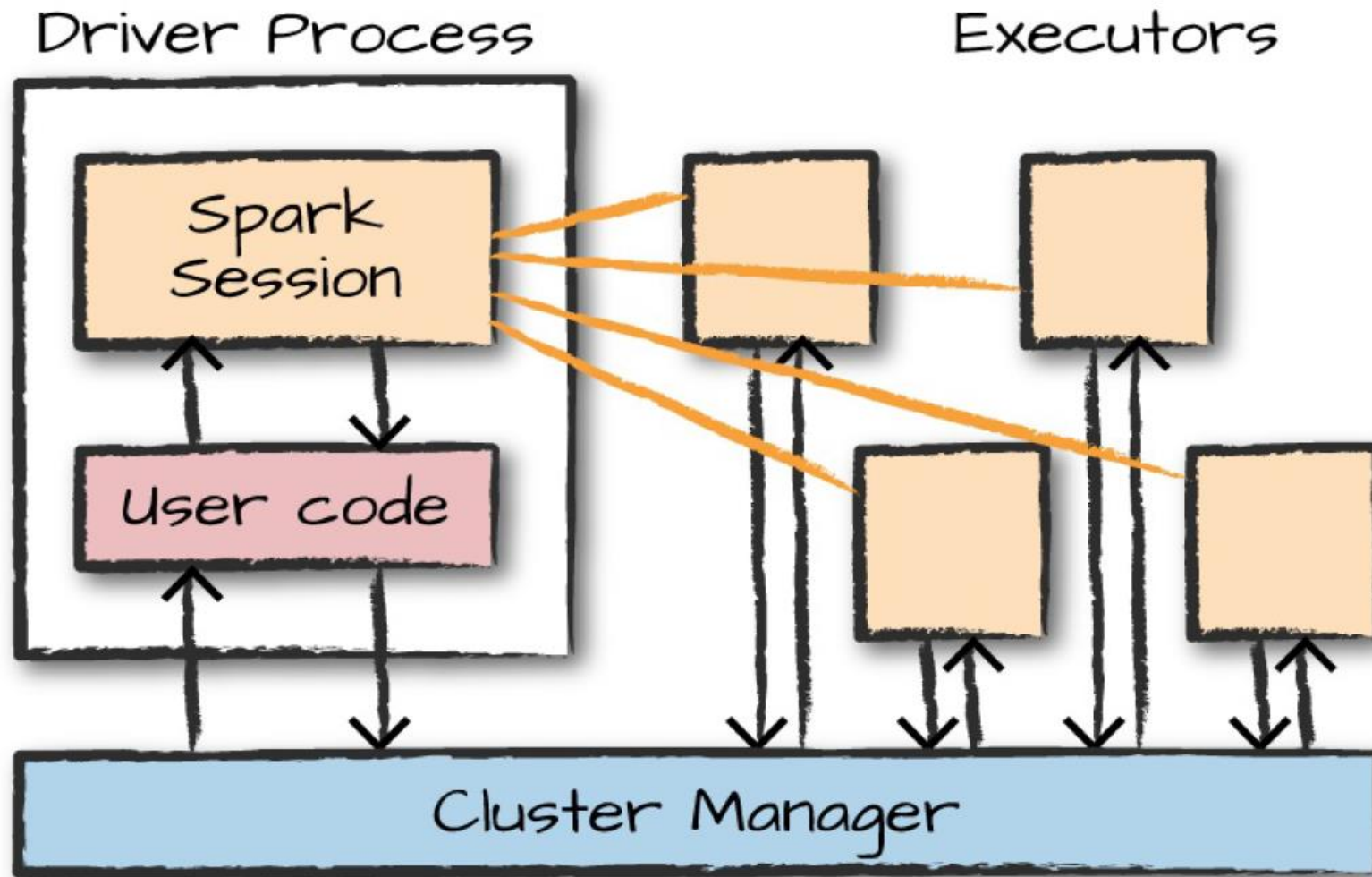
Spark Applications

- Spark Applications consist of a *driver* process and a set of *executor* processes.
- The driver process runs your `main()` function, sits on a node in the cluster, and is responsible for three things:
 1. maintaining information about the Spark Application;
 2. responding to a user's program or input;
 3. analyzing, distributing, and scheduling work across the executors

Spark Applications

- The *executors* are responsible for actually carrying out the work that the driver assigns them.
- This means that each executor is responsible for only two things:
 1. executing code assigned to it by the driver,
 2. reporting the state of the computation on that executor back to the driver node.

Spark Applications



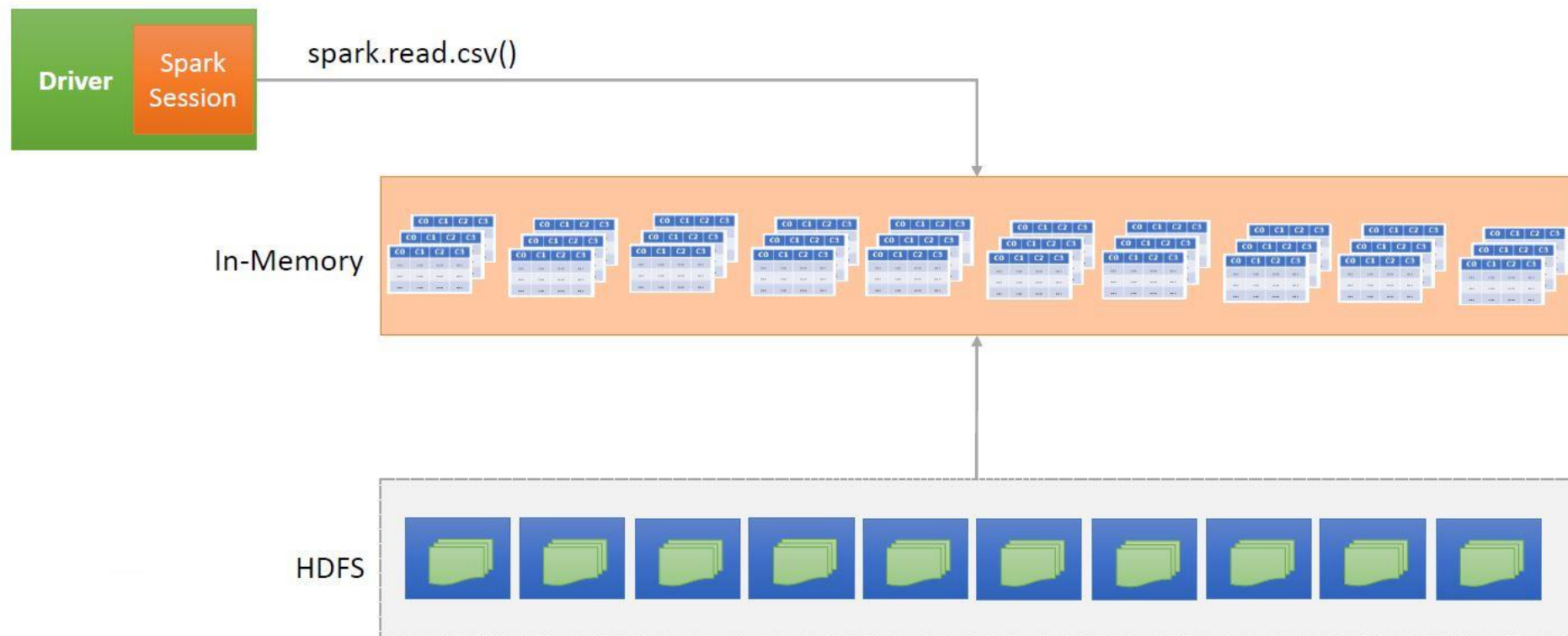
Spark Applications

- Spark, in addition to its cluster mode, also has a *local mode*.
- The driver and executors are simply processes, which means that they can live on the same machine or different machines.
- In local mode, the driver and executors run (as threads) on your individual computer instead of a cluster.

Spark Applications (Key Points)

- Spark employs a cluster manager that keeps track of the resources available.
- The driver process is responsible for executing the driver program's commands across the executors to complete a given task.
- The executors, for the most part, will always be running Spark code. However, the driver can be “driven” from a number of different languages through Spark's language APIs.

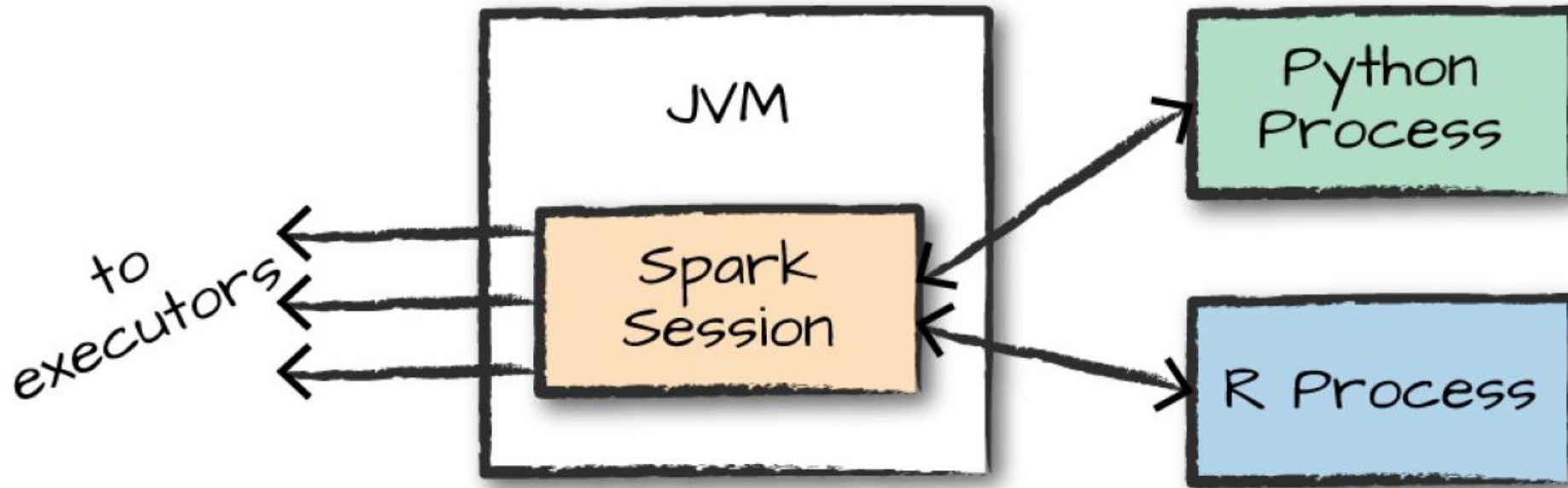
Simple Visualization



Spark's Language APIs

- Spark's language APIs make it possible for you to run Spark code using various programming languages.
- For the most part, Spark presents some core “concepts” in every language; these concepts are then translated into Spark code that runs on the cluster of machines.
- If you use just the Structured APIs, you can expect all languages to have similar performance characteristics

Spark's Language APIs



The relationship between the SparkSession and Spark's Language API

Spark's APIs

- We can drive Spark from a variety of languages. (Python, R...)

What it makes available in those languages ?.

- Spark has two fundamental sets of APIs:
 1. The low-level “unstructured” APIs
 2. The higher-level structured APIs.

SparkSession

- You control your Spark Application through a driver process called the SparkSession.
- The SparkSession instance is the way Spark executes user-defined manipulations across the cluster.

SparkSession

- There is a one-to-one correspondence between a SparkSession and a Spark Application.
- In Scala and Python, the variable is available as a spark when you start the console.
 - Python - *<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>*
 - Scala -
res0:org.apache.spark.sql.SparkSession=org.apache.spark.sql.SparkSession@.
..

DataFrame : Concept


- A DataFrame is the most common Structured API and simply represents a table of data with rows and columns.
- The list that defines the columns and the types within those columns is called the *schema*.
- You can think of a DataFrame as a spreadsheet with named columns.
- A spreadsheet sits on one computer in one specific location, whereas a Spark DataFrame can span thousands of computers.
- The reason for putting the data on more than one computer should be intuitive:
 1. The data is too large to fit on one machine
 2. It would simply take too long to perform that computation on one machine

DataFrame : Concept

- **Example:**

```
myRange = spark.range(1000).toDF("number")
```


DataFrame : Concept

 databricks

vrp02@ganpatuniversity.ac.in

DataFrame Demo Python

File Edit View Run Help Last edit was 2 days ago Provide feedback

Run all Terminated Publish

Cmd 1

```
1 myRange = spark.range(1000).toDF("number")
```

myRange: pyspark.sql.dataframe.DataFrame = [number: long]

Command took 0.36 seconds -- by vrp02@ganpatuniversity.ac.in at 8/12/2023, 11:51:42 AM on My Cluster

Cmd 2

Python

```
1 myRange.show()
```

(1) Spark Jobs

number
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

DataFrame : Concept

1. **spark**: This refers to the SparkSession, which is the entry point for using Spark functionality in PySpark. It's the main interface through which you interact with Spark.
2. **range(1000)**: This is a Python built-in function that generates a sequence of numbers from 0 to 999 (1000-1). It creates a sequence of integers.
3. **spark.range(1000)**: In this context, the SparkSession's range() method is used to create a data frame with a single column named "id" (by default) containing the generated sequence of numbers from 0 to 999.
4. **toDF("number")**: The toDF() method converts the generated DataFrame into a new DataFrame with the specified column name. In this case, "number" is the name of the single column in the resulting DataFrame

DataFrame : Concept

Spreadsheet on
a single machine

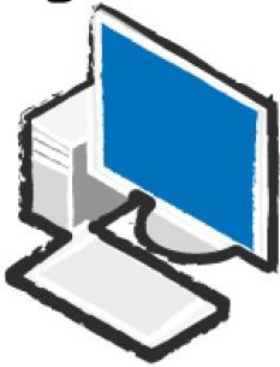
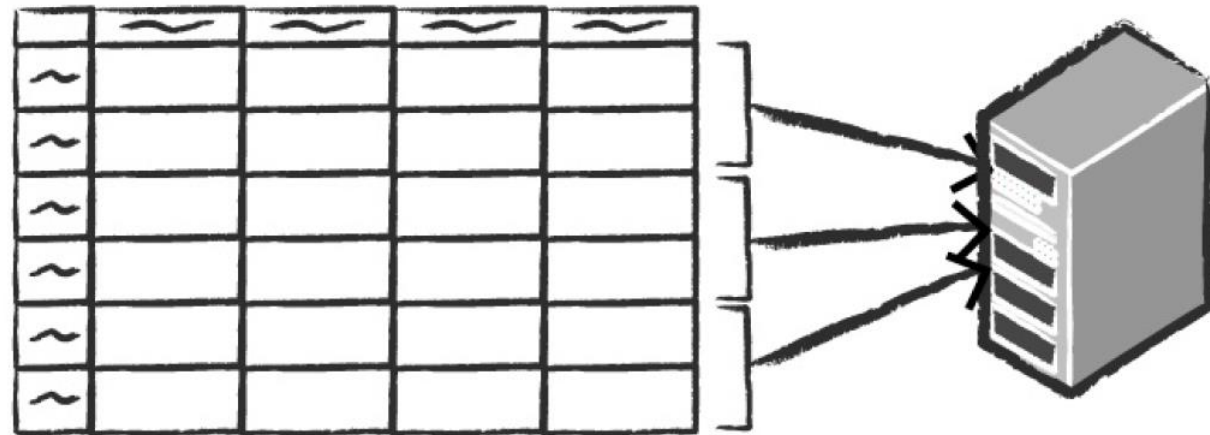


Table or Data Frame
partitioned across servers
in a data center



Distributed versus single-machine analysis

DataFrame : Concept

- The DataFrame concept is not unique to Spark. R and Python both have similar concepts.
- However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines. This limits what you can do with a given DataFrame to the resources that exist on that specific machine.
- However, because Spark has language interfaces for both Python and R, it's quite easy to convert Pandas (Python) DataFrames to Spark DataFrames, and R DataFrames to Spark DataFrames.

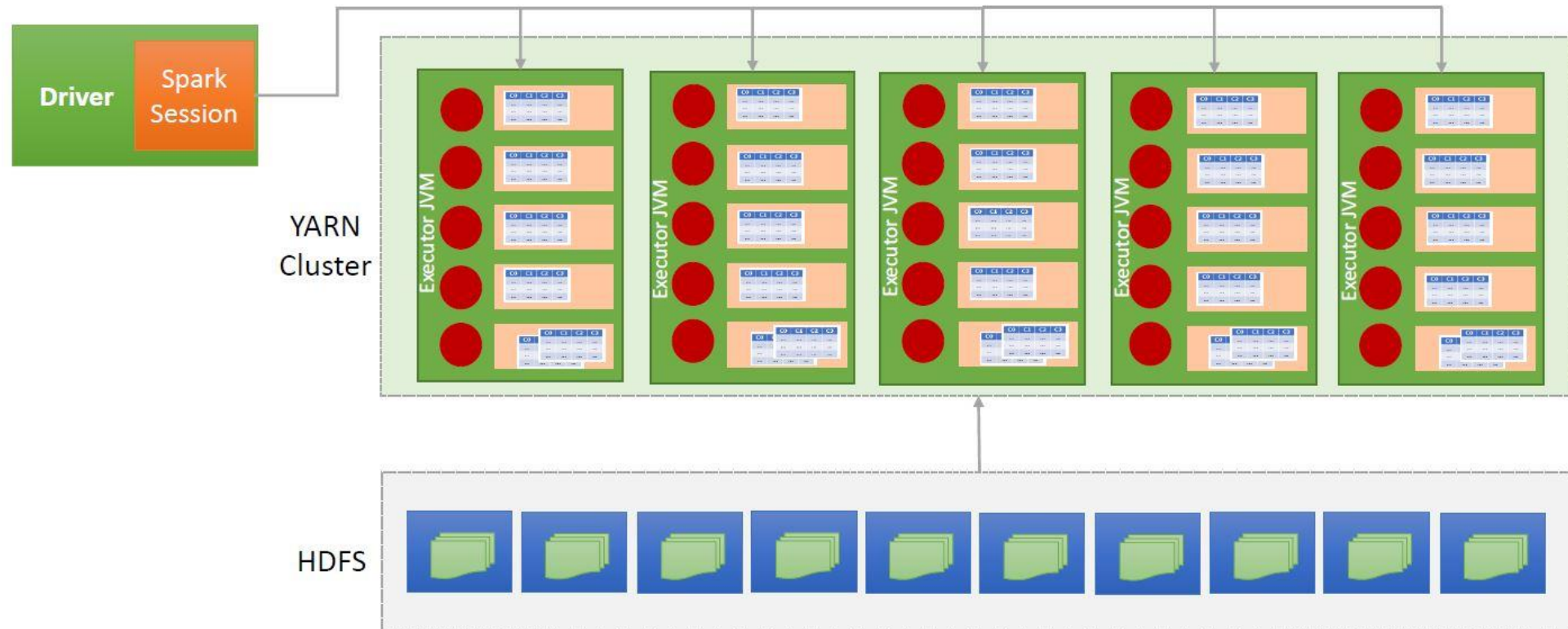
Core Abstractions

- Spark has several core abstractions:
 1. Datasets
 2. DataFrames
 3. SQL Tables
 4. RDDs
- These different abstractions all represent distributed collections of data. The easiest and most efficient are DataFrames, which are available in all languages.

Partitions

- To allow every executor to perform work in parallel, Spark breaks up the data into chunks called *partitions*.
- A partition is a collection of rows that sit on one physical machine in your cluster.
- A DataFrame's partitions represent how the data is physically distributed across the cluster of machines during execution.
- If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors.
- If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource

Simple Visualization



Transformations

- In Spark, the core data structures are immutable, meaning they cannot be changed after they're created.
- To “change” a data frame, you need to instruct Spark how you would like to modify it to do what you want.
- These instructions are called **transformations**.

Transformations : Example

- Our created Dataframe

```
myRange = spark.range(1000).toDF("number")
```

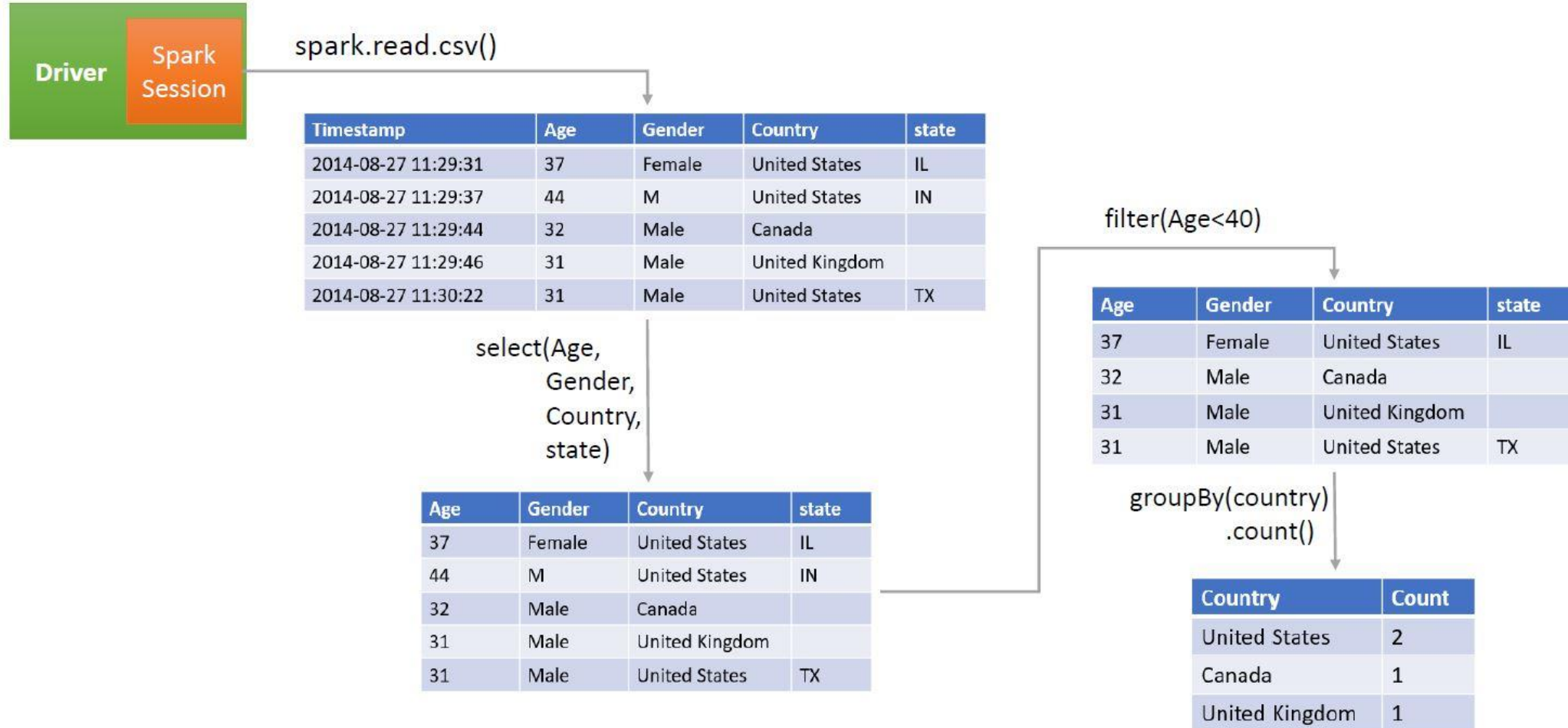
- Now let's apply transformation to it

```
divisBy2 = myRange.where("number % 2 = 0")
```

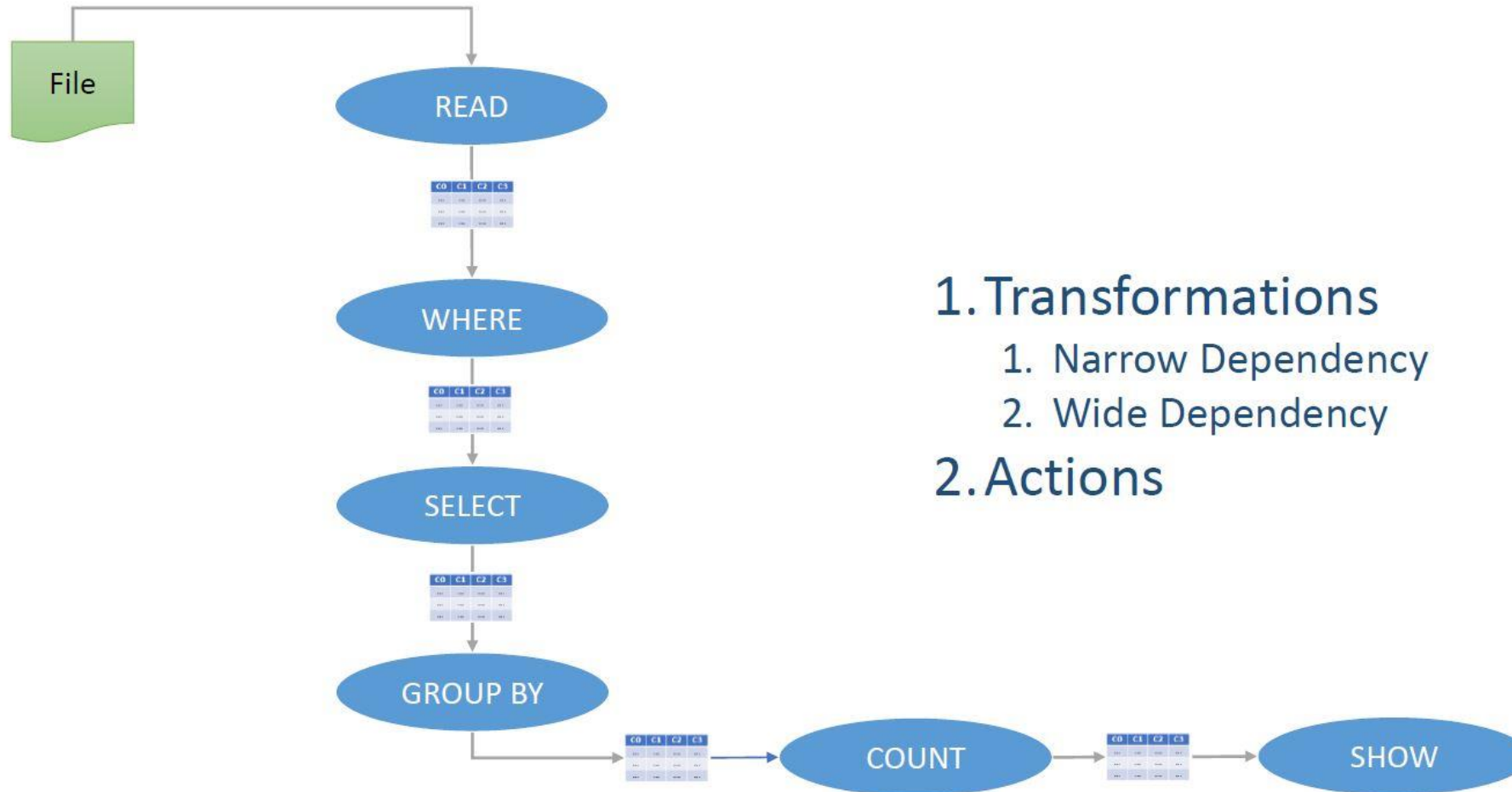
Transformations

- Notice that these return no output. because we specified only an abstract transformation, and Spark will not act on transformations until we call an “*action*”
- Transformations are the core of how you express your business logic using Spark.
- There are two types of transformations:
 1. Those that specify *narrow dependencies*,
 2. Those that specify *wide dependencies*.

Transformations



Transformations



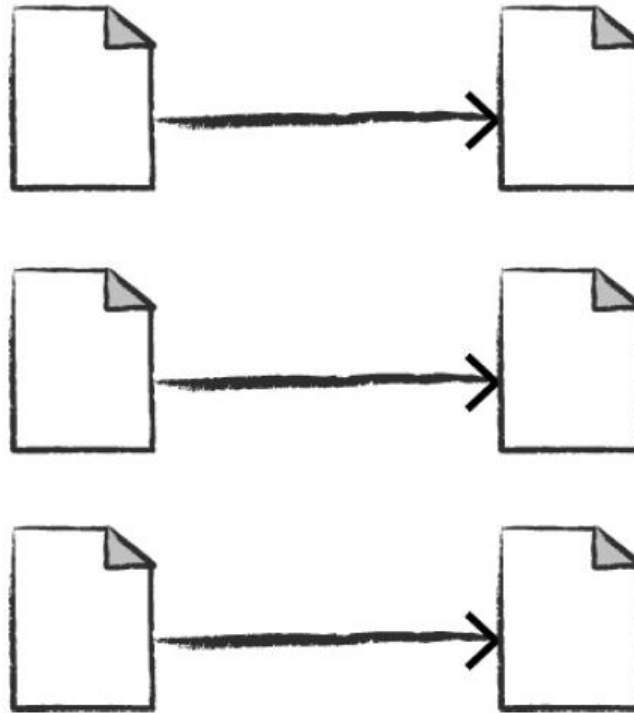
1. Transformations

1. Narrow Dependency
2. Wide Dependency

2. Actions

Transformations

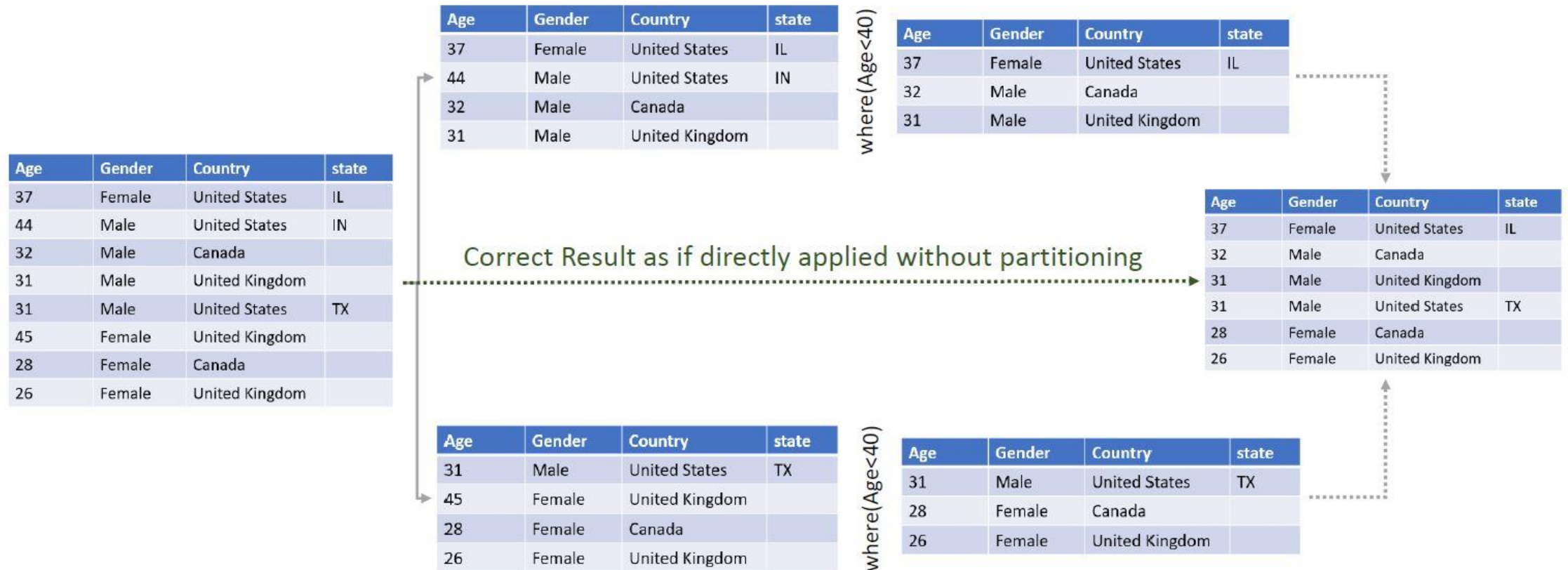
Narrow transformations
1 to 1



A narrow dependency

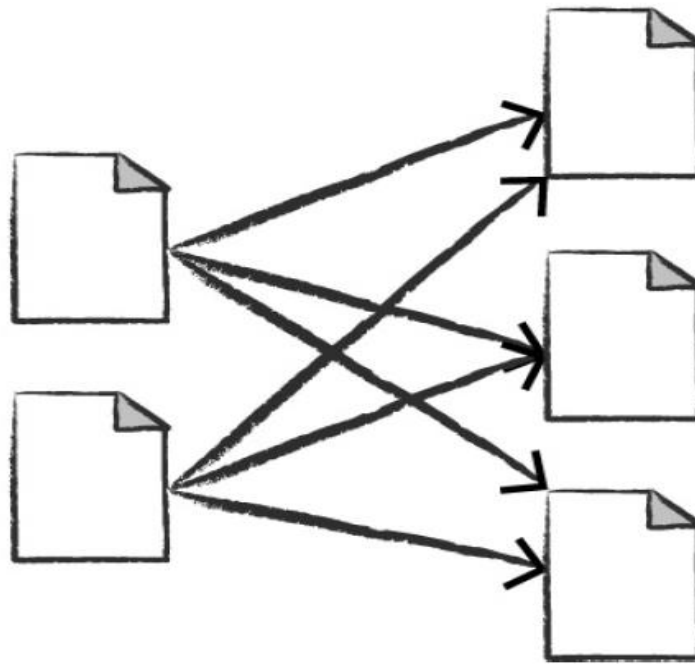
Narrow Transformation

A transformation performed independently on a single partition to produce valid results.



Transformations

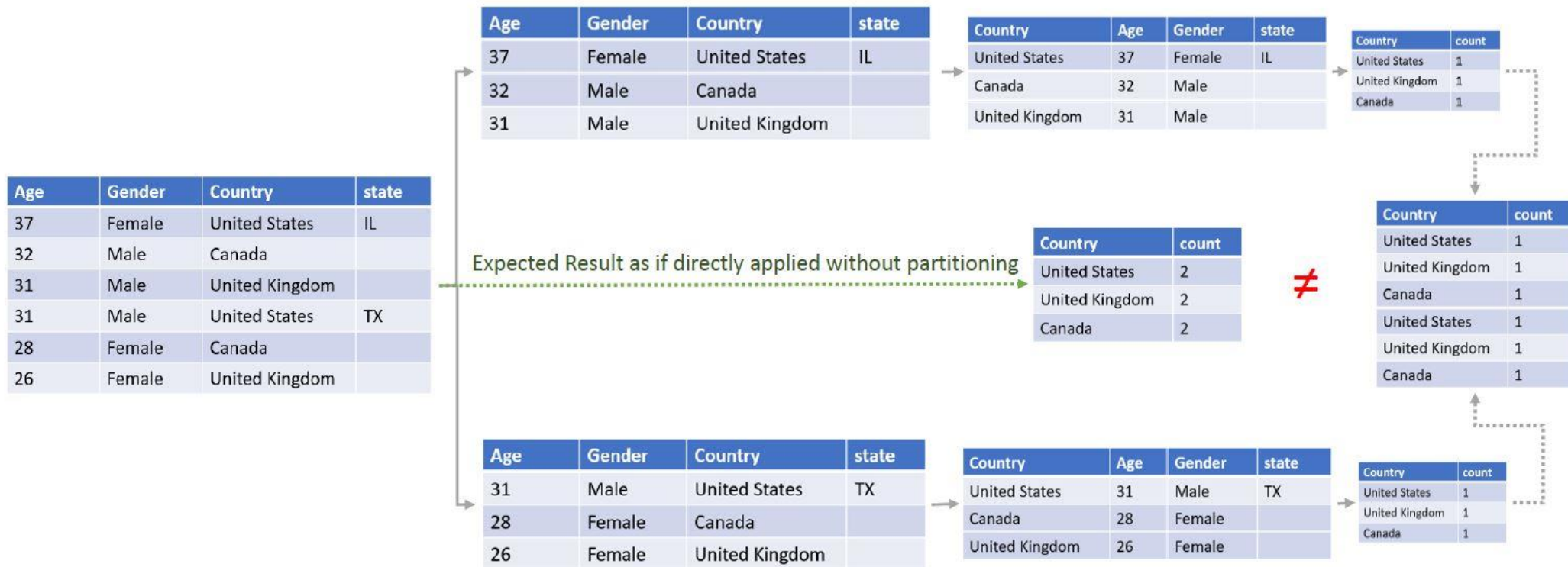
Wide transformations
(shuffles) 1 to N



A wide dependency

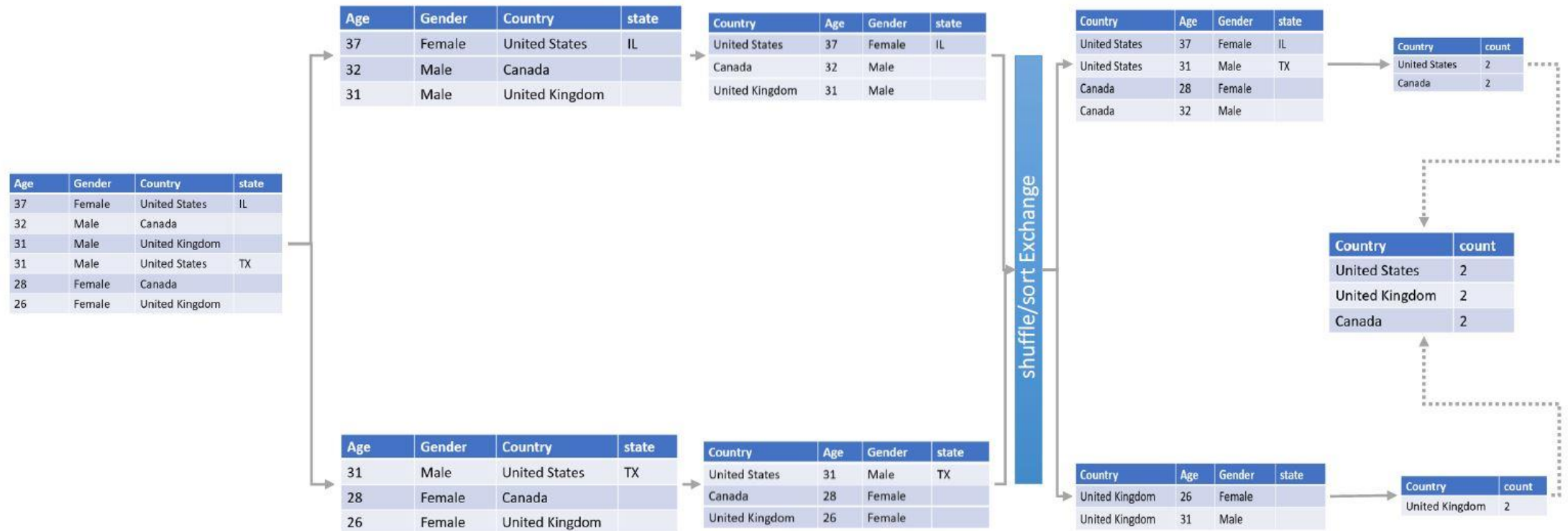
Wide Transformation

A transformation that requires data from other partitions to produce valid results.



Wide Transformation

A transformation that requires data from other partitions to produce valid results.



Wide Transformation

- A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions.
- This is referred to as a *shuffle* whereby Spark will exchange partitions across the cluster.

Narrow vs Wide Transformation Note:

- A narrow dependency, also known as a one-to-one dependency, occurs when each partition of the parent RDD is used by at most one partition of the child RDD.
- In other words, a narrow transformation doesn't require data shuffling across the partitions.
- Narrow dependencies are more efficient as they minimize data movement and increase parallelism.

Narrow vs Wide Transformation Note:

- **Example of Narrow Dependency:** Consider you have an RDD of student scores and you want to filter out students with scores above a certain threshold. This transformation is narrow because each partition of the parent RDD contributes data to only one partition of the resulting RDD. There's no need to shuffle data; each partition filters its own data independently.
- # Original RDD: student_scores_rdd
- # Narrow Transformation
- ***filtered_scores_rdd = student_scores_rdd.filter(lambda score: score >= 60)***

Narrow vs Wide Transformation Note:

- A wide dependency, also known as a shuffle dependency, occurs when multiple child partitions depend on the same parent partitions or involve data shuffling. Wide transformations lead to data movement across partitions, which can be costly in terms of time and resources. It often requires the data to be exchanged between different nodes.
- Suppose you want to count the number of students who scored within specific score ranges. This transformation involves a group by operation where data with different scores must be aggregated together. It's a wide transformation because data needs to be shuffled and reorganized across partitions.

Narrow vs Wide Transformation Note:

- # Original RDD: student_scores_rdd
- # Wide Transformation
- *score_range_counts_rdd = student_scores_rdd.map(lambda score: (get_range(score), 1)).reduceByKey(lambda a, b: a + b)*

Lazy Evaluation

- Spark is known as a "lazy evaluator" because of its approach to processing data transformations.
- This concept is closely related to Spark's strategy for optimizing the execution of data processing operations.

Why Spark is called a "lazy evaluator":

Deferred Execution:

- When you perform a transformation (e.g., filtering, mapping, aggregating) on a Spark DataFrame, the actual computation doesn't happen immediately.
- Instead, Spark records the transformation operation as a logical plan, creating a sequence of operations that need to be performed on the data.

Why Spark is called a "lazy evaluator":

Transformation Chain:

- Spark allows you to chain multiple transformations together to build a more complex processing pipeline.
- These transformations are not executed as you apply them but are remembered in a directed acyclic graph (DAG) of transformations.

Why Spark is called a 'lazy evaluator':

Action Triggers Execution:

- The actual execution of these transformations is deferred until an action is performed on the data.
- Actions are operations that trigger the computation and result in data being produced or collected, such as `show()`, `count()`, `collect()`, or saving the data to a file.

Actions

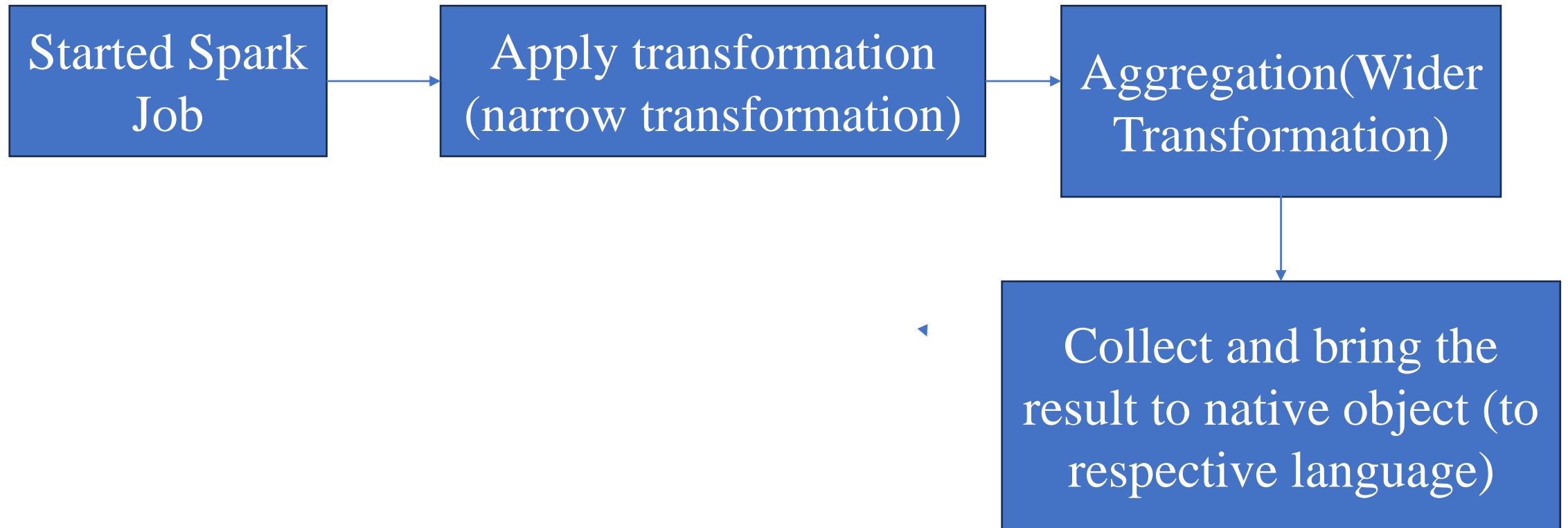
- Transformations allow us to build up our logical transformation plan.
- To trigger the computation, we run an *action*.
- An action instructs Spark to compute a result from a series of transformations.
- The simplest action is `count`, which gives us the total number of records in the DataFrame for the previous example.

`divisBy2.count()`

Actions

- There are three kinds of actions:
 1. Actions to view data in the console
 2. Actions to collect data to native objects in the respective language
 3. Actions to write to output data sources

Simple Flow :



Dataframe Examples

- Follow this official document QuickStart and understand the process of creation with only a simple example.
- **Link:**
https://spark.apache.org/docs/3.1.1/api/python/getting_started/quickstart.html#DataFrame-Creation
- **Note:** You can check the output running these on Databricks (If requires)
- **Note:** In the exam, you don't need to initialize packages. Instead, focus on the main code. (methods and their meaning according to questions)

Dataframe Example :

- *from pyspark.sql import SparkSession*
- *from pyspark.sql import Row*
- *from pyspark.sql import functions as func*
- *spark=SparkSession.builder.appName("FriendsByAge").getOrCreate)*
- **Note : You are not expected to write above configuration in exam**
- `lines = spark.read.option("header", "true").option("inferSchema", "true").csv("path")`

Dataframe Example :

- # Select only age and numFriends columns

friendsByAge = lines.select("age", "friends")

- # From friendsByAge we group by "age" and then compute average

friendsByAge.groupBy("age").avg("friends").show()

- # Sorted

friendsByAge.groupBy("age").avg("friends").sort("age").show()

Dataframe Example :

- # Formatted differently :

```
friendsByAge.groupBy("age").agg(func.round(func.avg("friends"),2)).sort("age").show()
```

- # With a custom column name

```
friendsByAge.groupBy("age").agg(func.round(func.avg("friends"), 2).alias("friends_avg")).sort("age").show()
```

- spark.stop()

Dataframe Example :

- `from pyspark.sql import SparkSession`

Create a Spark session

- `spark = SparkSession.builder.appName("WhereClauseExample").getOrCreate()`
- **`# df = spark.read.csv("data.csv")`**

Create a DataFrame

```
data = [("Alice", 25, "Engineer"),  
        ("Bob", 30, "Data Scientist"),  
        ("Charlie", 22, "Analyst")]  
columns = ["Name", "Age", "Occupation"]  
df = spark.createDataFrame(data, columns)
```

Dataframe Example :

- **# Use the where clause to filter rows where Age is greater than 23**
- `filtered_df = df.where(df.Age > 23)`
- `selected_df = df.select("Name", "Occupation")`
- **# Show the result**
- `filtered_df.show()`
- **# Stop the Spark session**
- `spark.stop()`