More
(mostly non-streaming)
Spark

Advanced Spark / leftovers

- Strings
- RDD transformations
- Pandas functions
- User-defined functions (UDFs)
- Unions and joins
- Aggregations (GroupBy)

Strings

- from pyspark.sql.functions import *
- Standard string functions:
  - initcap, lower, upper, lit, ltrim, rtrim, rpad, lpad, trim
- Character translations:
  - translate(*col, from_chars, to_chars*)
- Regular expressions:
  - regexp_replace(*df_column, from_regex, to_string*)
  - regexp_extract(*df:column, extr_regex, pos*)
- JSON:
  - parse from JSON or extract JSON objects
  - JSON operations directly on strings

Resilient Distributed Datasets (RDDs)

- From exercise 1:

```
word_lists = texts.select(split(texts.text, ' ').alias('word_list'))

from pyspark.sql.functions import explode
words = word_lists.select(explode(word_lists.word_list).alias('word'))
```

- RDD solution:

```
text_rdd = texts.rdd
word_rdd = text_rdd.flatMap(lambda row: row.text.split(' '))

from pyspark.sql.types import StructType, StructField, StringType
str_schema = StructType([StructField('word', StringType(), True)])

from pyspark.sql import Row
words = word_rdd.map(lambda word: Row(word)).toDF(str_schema)
```

Pandas functions

- From exercise 1:

```
from pyspark.sql.functions import split, explode
word_lists = texts.select(split(texts.text, ' ').alias('word_list'))
words = word_lists.select(explode(word_lists.word_list).alias('word'))
```

- Pandas solution:

```
import itertools
import pandas as pd  # pip install pandas pyarrow

def word_map(dfs):
    for df in dfs:
        yield pd.DataFrame(
            itertools.chain(*df.text.apply(lambda t: t.split(' '))))

words = texts.mapInPandas(word_map, word_schema)
```

Pandas functions

- Easier to read Pandas solution:

```
import itertools
import pandas as pd

def word_map_df(df):
    word_list_df = df.text.apply(lambda t: t.split(' '))
    chained_list = itertools.chain(*word_list_df)
    return pd.DataFrame(chained_list)

def word_map(dfs):
    for df in dfs:
        yield word_map_df

words = texts.mapInPandas(word_map, word_schema)
```

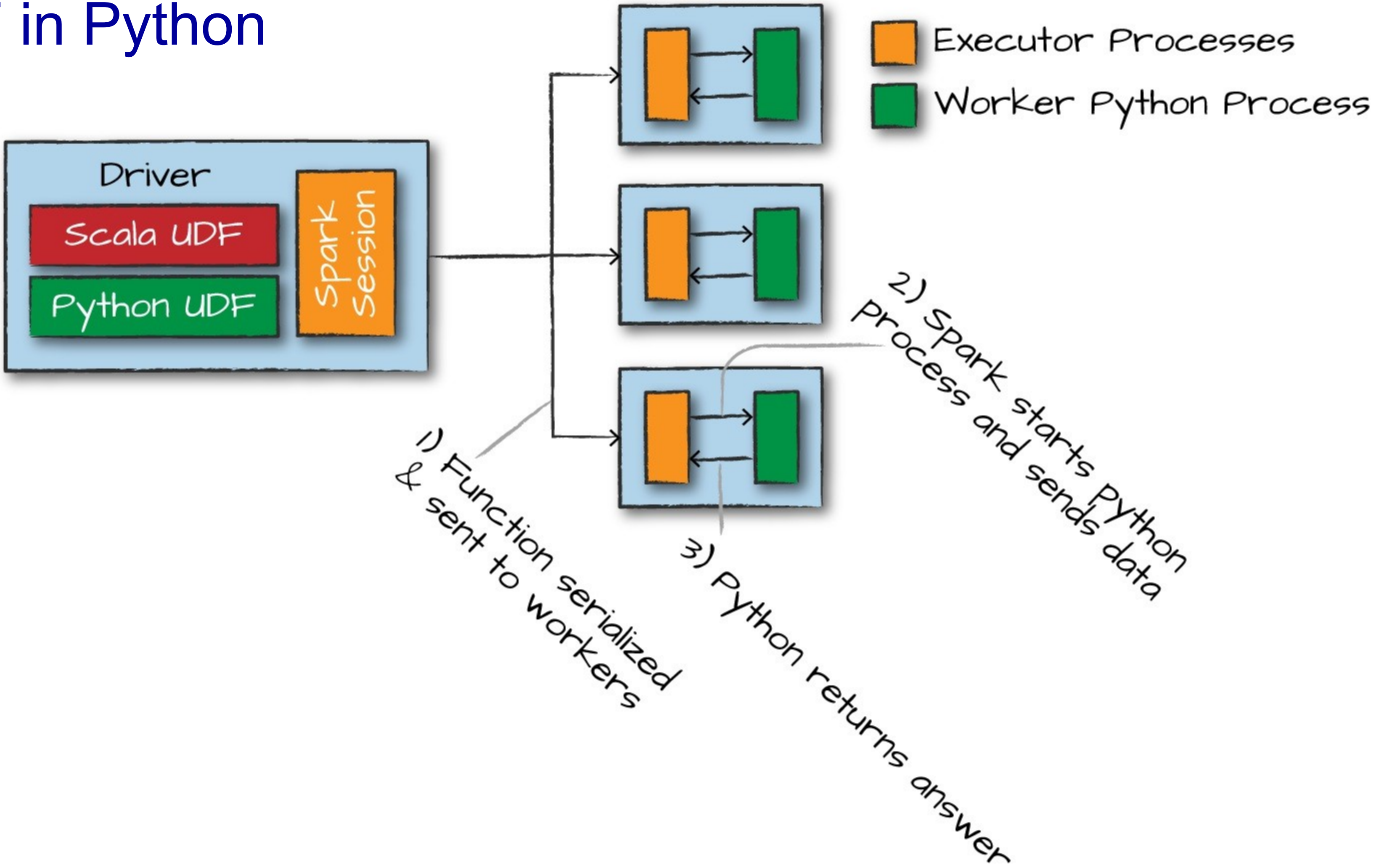
User-Defined Functions

- User-defined functions (UDFs):
  - express custom transformations in Java, Scala, Python...
  - can use external libraries
  - take and return one or more columns
  - can be written in several different programming languages
  - operate on the data, row-by-row or frame-by-frame
  - need to be registered as temporary functions in a specific SparkSession
    - from pyspark.sql.functions import udf
      udf_*func* = udf(lambda x: *func*(x), *SparkType()*)
      - or @udf decorator
    - serialised and distributed to worker machines (executors)
  - to use also in SQL statements
      - spark.udf.register("*function_name*", *udf_func*, *SparkType()*)

UDF registration

- Spark:
  - serialises the function on the driver (e.g., to Java bytecode)
  - transfers it over the network to all executor processes
  - regardless of language
- Scala or Java functions:
  - run by the Java Virtual Machine (JVM) on the worker
- Python functions:
  - Spark starts a Python process on the worker
  - serializes the JVM-data to a Python-readable format
  - executes the function row by row on that data in the Python process
  - returns the results of the row operations to the JVM and Spark.

UDF in Python



Performance warnings

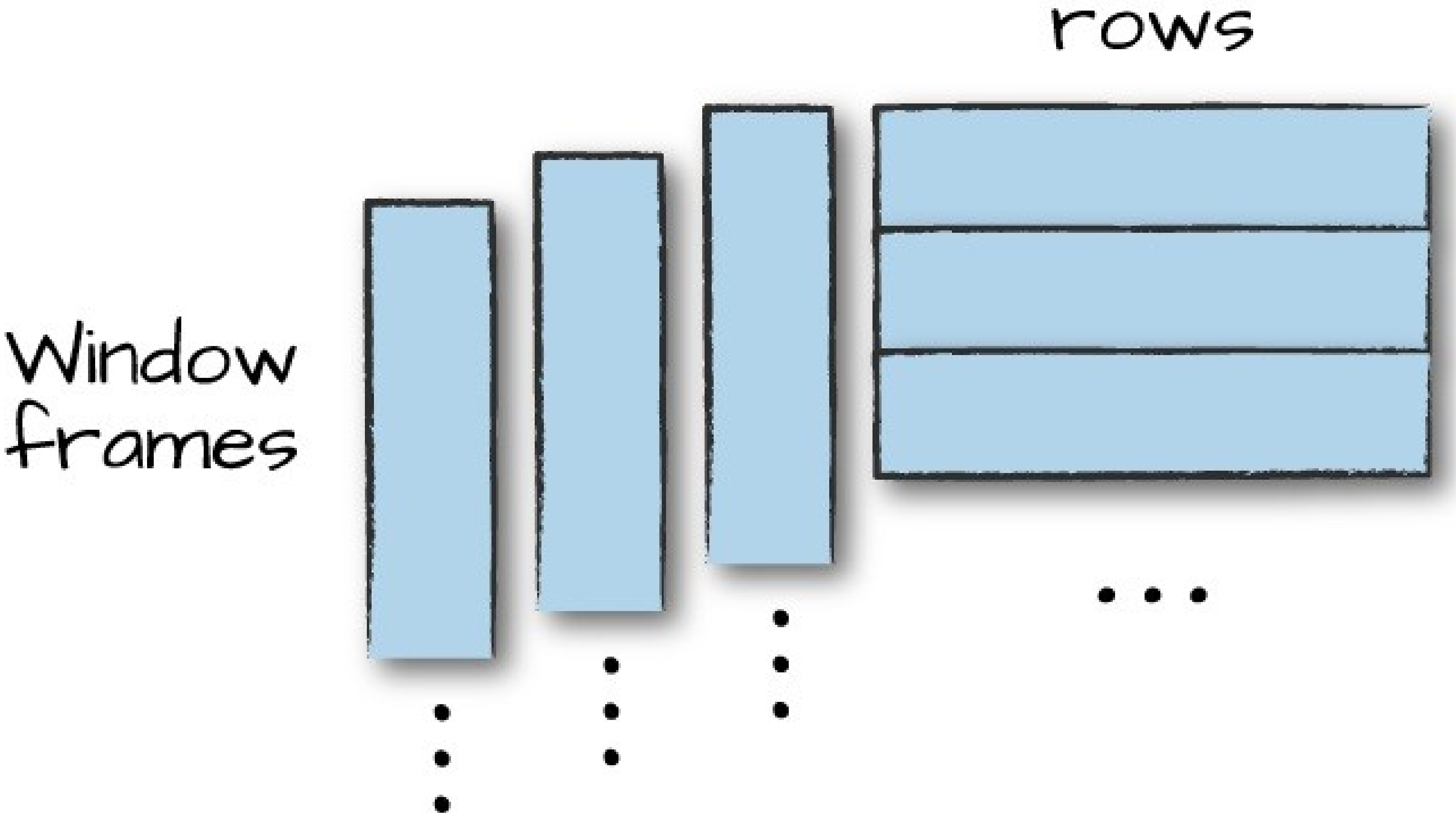
- Scala/Java functions:
  - runs on JVM - little performance penalty
  - careful about memory
  - a black-box to Spark
  - but misses some Spark optimisations
- Python:
  - starting a Python process
  - serializing data to Python – and back to JVM:
    - expensive computationa
  - *Spark cannot manage the memory of the worker*
  - potentially the worker can fail
  - JVM and Python are competing for memory on the same machine
- *Write UDFs in Scala or Java even if you use Python overall!*

Aggregations

- Aggregating:
  - the act of collecting something together
  - a cornerstone of big-data analytics
  - also used in Pandas, SQL, SPARQL, spreadsheets...
- An aggregation specifies:
  - a *grouping strategy* that groups (splits) the rows in the DataFrame
    - overlapping or not, and completely or not
  - one or more *aggregation functions*
    - transforms one or more columns of each group (split) of input rows
    - must produce one result for each group (split) of input rows
    - the function results for each group (split) of input rows becomes an output row

Aggregations: grouping strategies

- No grouping: aggregation summarises the whole DataFrame
- Key grouping: group by one or more keys (columns)
- Windowing: a group per row, but also including «neighbouring» rows
- Grouping set:
  - aggregate at several levels in one operation
  - a “rollup” makes it possible for you to specify one or more keys, which will be summarized hierarchically
  - a “cube” allows you to specify one or more keys to transform the value columns, which will be summarized across all combinations of columns
- Each grouping returns a RelationalGroupedDataset on which we specify our aggregations


rows
Window
frames

Aggregations: aggregation functions

- Aggregation functions work as in Pandas, SQL, SPARQL, spreadsheets...
  - from pyspark.sql.functions import *
  - counting: count, countDistinct, approx_count_distinct
  - picking rows: first, last
  - statistics:
    - min, max, sum, sumDistinct, avg
    - var_pop, stddev_pop, var_samp, stddev_samp
    - skewness, kurtosis, correlation, covariance
  - aggregating to complex values:
    - collect_set, collect_list
    - for example, the result can be passed on to UDFs...
  - User-Defined Aggregation Functions (UDAFs)

Aggregations: rollups and cubes

- Performs several group-by style calculations in one go
  - rollup(): treats elements hierarchically
  - cube(): does the same thing across all combinations of columns
- Example:
  - two columns: time (a «Date» column) and location (a «Country» column)
  - *rollup(*) calculates aggregations of:
    - all rows
    - all times
    - all time and location *combinations*
  - *cube()* also calculates aggregations of:
    - all locations

Union

- Concatenating and appending rows
- To append to an immutable DataFrame:
  - union the original DataFrame along with the new DataFrame
  - make sure that they have the same schema and number of columns
    - otherwise, the union will fail

Join expressions (CZ, chapter 10)

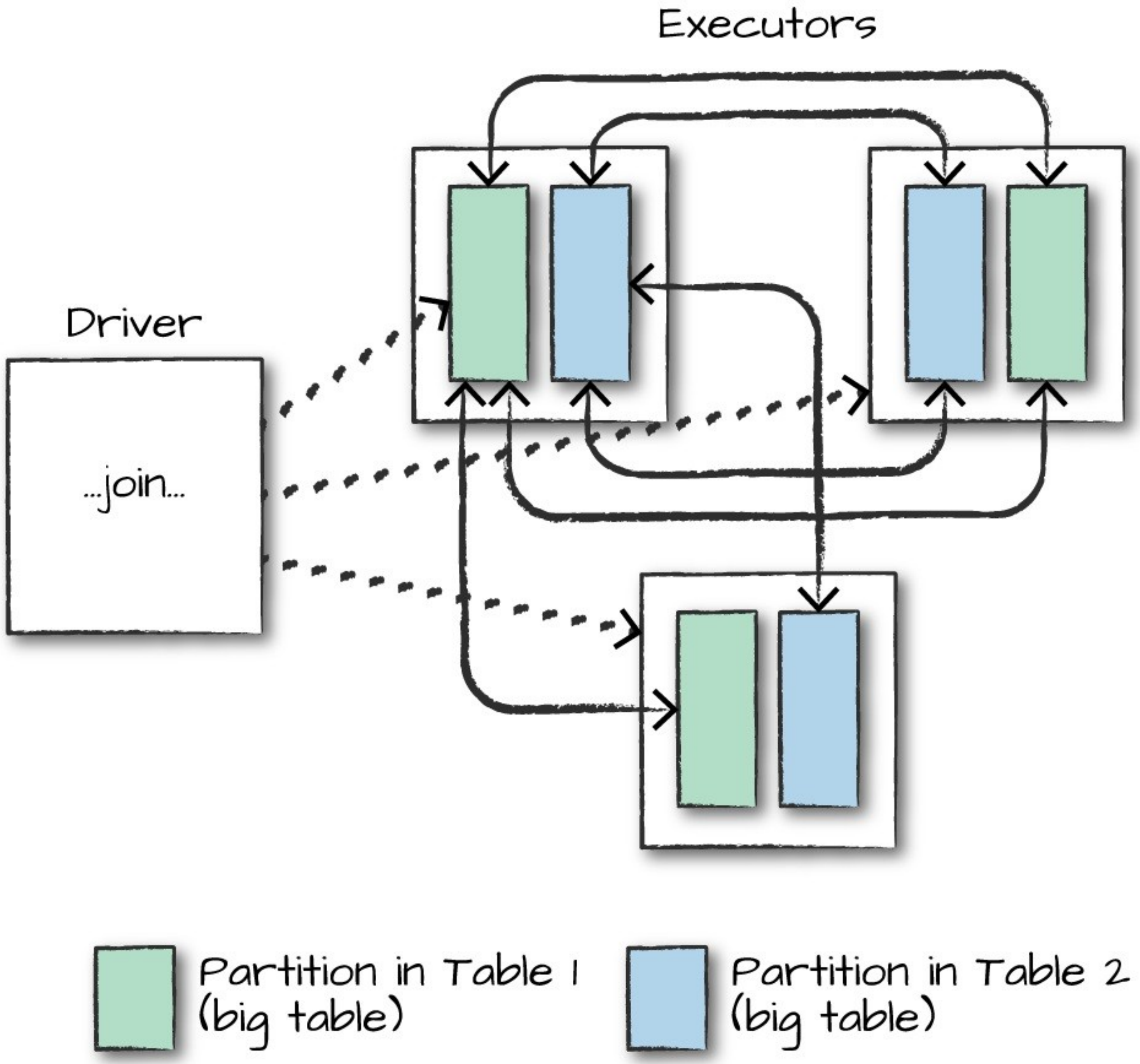
- A join:
  - brings together two sets of data, the left and the right
  - compares the value of one or more keys of the left and right
  - evaluates the result of a join expression
- Join expression:
  - determines whether Spark should bring together the left set of data with the right set of data
  - most common is equi-join:
    - if the specified keys in one row from the left and one row from the right datasets are equal, the results contains two to rows combined
  - many other join expressions
    - similar to Pandas and SQL

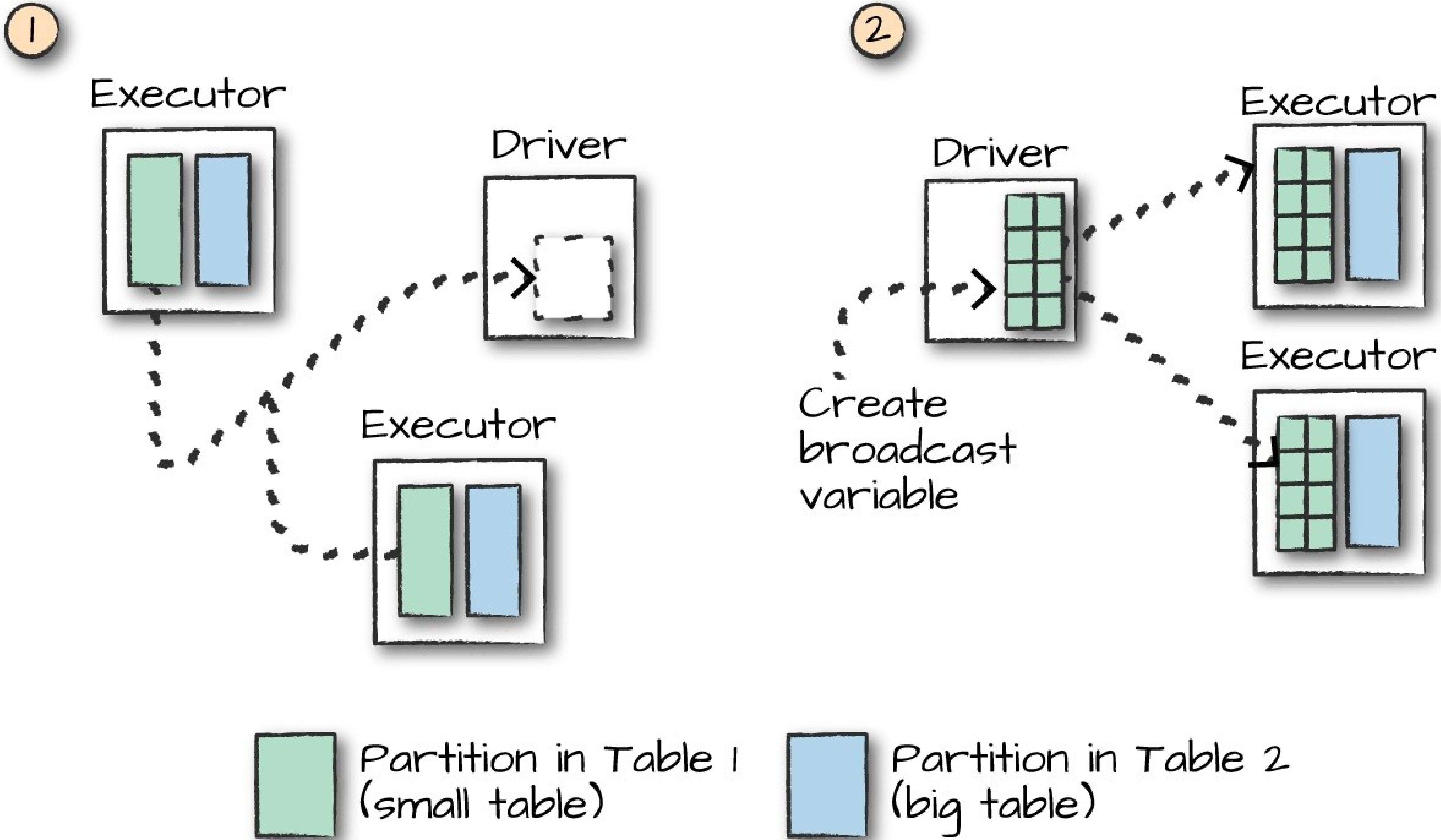
Join types

- Determines what should be in the result set:
  - inner (keep rows with keys that exist in the left and right datasets)
  - outer (keep rows with keys in either the left or right datasets)
  - left outer (keep rows with keys in the left dataset)
  - right outer (keep rows with keys in the right dataset)
  - left semi (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)
  - left anti (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)
  - natural (perform a join by implicitly matching the columns between the two datasets with the same names)
  - cross (or Cartesian) (match every row in the left dataset with every row in the right dataset)

How Spark performs joins

- Depends on:
  - per node computation strategy
  - node-to-node communication strategy:
    - shuffle join: rows from both tables are reshuffled by join keys
    - broadcast join: the smallest table is copied to all workers
- Big table–to–big table:
  - uses shuffle join
  - expensive because the network can become congested with traffic
  - best if the data are suitably partitioned already...


Executors
Driver
...join...
Partition in Table 1
(big table)
Partition in Table 2
(big table)


1
Executor
Driver
Executor
2
Driver
Executor
Executor
Create broadcast variable
Partition in Table 1 (small table)
Partition in Table 2 (big table)

How Spark performs joins

- Big table–to–small table:
  - uses broadcast join
  - replicates the smallest DataFrame onto every worker node
  - prevents all-to-all communication during the entire join process
  - joins will be performed on every single node individually
  - CPU is the biggest bottleneck.
- Little table–to–little table:
  - let Spark decide
  - can also force a broadcast join