Building

solid data pipelines

with

PySpark

by Oliver Willekens,
senior data engineer and instructor
at Data Minded

dataminded

Building

solid data pipelines

with

PySpark

by Oliver Willekens,
senior data engineer and instructor
at Data Minded

dataminded

Structure of this course

18 theory modules

→ videos

5 hands-on workshops

→ interactive sessions, with an instructor

Course outline

1. Context - situating Spark in the big data ecosystem
2. Spark offering - Spark libraries and language bindings
3. Spark's main processes - interprocess communication amongst distributed actors
4. Core concepts of Spark - what one typically interacts with in code
5. Introduction to Pytest - releasing with confidence through automated testing

Hands-on exercises

6. The Spark DataFrame API - understanding the most commonly encountered operations on DataFrames
7. Spark's lazy execution model - why code doesn't do what you think it does
8. Slowly moving to the DataFrame API - when you already have single-node functionality
9. Testing PySpark pipelines using Pytest - what utilities to have, what to test for and what not to do

Hands-on exercises

10. Reading and writing in a distributed setting - advanced serialization formats, impact on the file system
11. Understanding Spark's query plan optimizer - because you don't have to be an expert
12. Monitoring performance on a high-level - the Spark web interface
13. Modifying data partitions for performance - when writing pure Spark SQL, this is the next easy boost
14. Common notions and operations in ETL - Data Catalogs and cleaning data only a single time

Hands-on exercises

15. Reusing pipeline parts - when to use `cache`?
16. Intro to windowing functions - a commonly used analytical tool in reports
17. Applying the performance techniques - a real-world case study

Hands-on exercises

18. More common ETL operations - when reporting matters

Hands-on exercises

Objectives of this course

Objectives of this course

- understand Apache Spark's execution model,

Objectives of this course

- understand Apache Spark's execution model,
- apply common operations,

Objectives of this course

- understand Apache Spark's execution model,
- apply common operations,
- work through the two most common business scenarios where Spark is used

Objectives of this course

- understand Apache Spark's execution model,
- apply common operations,
- work through the two most common business scenarios where Spark is used
- learn to improve the performance of Spark programs

Objectives of this course

- understand Apache Spark's execution model,
- apply common operations,
- work through the two most common business scenarios where Spark is used
- learn to improve the performance of Spark programs
- improve the readability of Spark programs

Objectives of this course

- understand Apache Spark's execution model,
- apply common operations,
- work through the two most common business scenarios where Spark is used
- learn to improve the performance of Spark programs
- improve the readability of Spark programs
- reduce the chance of mistakes

My assumptions about why you're taking this class…

My assumptions about why you're taking this class…

- You're **using Spark** or PySpark or have to use it soon

My assumptions about why you're taking this class…

- You're **using Spark** or PySpark or have to use it soon
- You want to be ***effective*** and ***efficient***

My assumptions about why you're taking this class…

- You're **using Spark** or PySpark or have to use it soon
- You want to be ***effective*** and ***efficient***
- You want to **create applications** that do what you think they should

My assumptions about why you're taking this class…

- You're **using Spark** or PySpark or have to use it soon
- You want to be ***effective*** and ***efficient***
- You want to **create applications** that do what you think they should
- You want to learn more about the **lower level details**

My assumptions about why you're taking this class…

- You're **using Spark** or PySpark or have to use it soon
- You want to be ***effective*** and ***efficient***
- You want to **create applications** that do what you think they should
- You want to learn more about the **lower level details**
- You're not an expert programmer

Context

situating Spark in the big data ecosystem

Hadoop is an ecosystem designed to deal with data across cluster nodes. It is built on top of 4 components.

Hadoop is an ecosystem designed to deal with data across cluster nodes. It is built on top of 4 components.

- Hadoop Distributed File System (HDFS)

Hadoop is an ecosystem designed to deal with data across cluster nodes. It is built on top of 4 components.

- Hadoop Distributed File System (HDFS)
- Hadoop YARN

Hadoop is an ecosystem designed to deal with data across cluster nodes. It is built on top of 4 components.

- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- Hadoop MapReduce

Hadoop is an ecosystem designed to deal with data across cluster nodes. It is built on top of 4 components.

- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- Hadoop MapReduce
- Hadoop Common

Hadoop is an ecosystem designed to deal with data across cluster nodes. It is built on top of 4 components.

- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- Hadoop MapReduce
- Hadoop Common

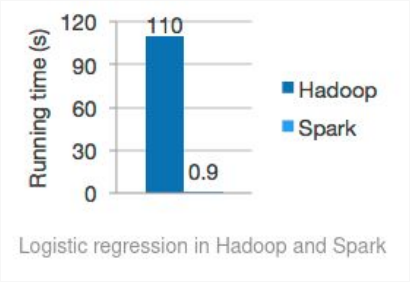
"Ecosystem" is pretty apt:

Spark offering

Spark libraries and language bindings

Apache Spark is mostly a replacement for Hadoop MapReduce. It integrates well with YARN and HDFS and offers an API in 5 languages.

- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- ~~Hadoop MapReduce~~ → Spark
- Hadoop Common



Screenshot from the Apache Spark website

Apache Spark is mostly a replacement for Hadoop MapReduce. It integrates well with YARN and HDFS and offers an API in 5 languages.

- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- ~~Hadoop MapReduce~~ → Spark
- Hadoop Common

Spark

Java

Scala

python™

R

SQL

Apache Spark is mostly a replacement for Hadoop MapReduce. It integrates well with YARN and HDFS and offers an API in 5 languages.

- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- ~~Hadoop MapReduce~~ → Spark
- Hadoop Common

Spark

Java

Scala

python™

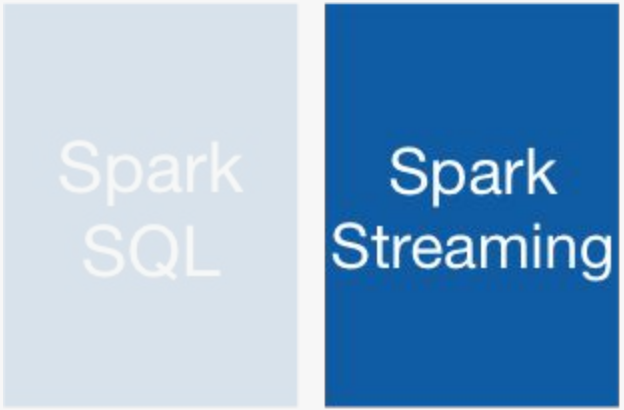
R

SQL

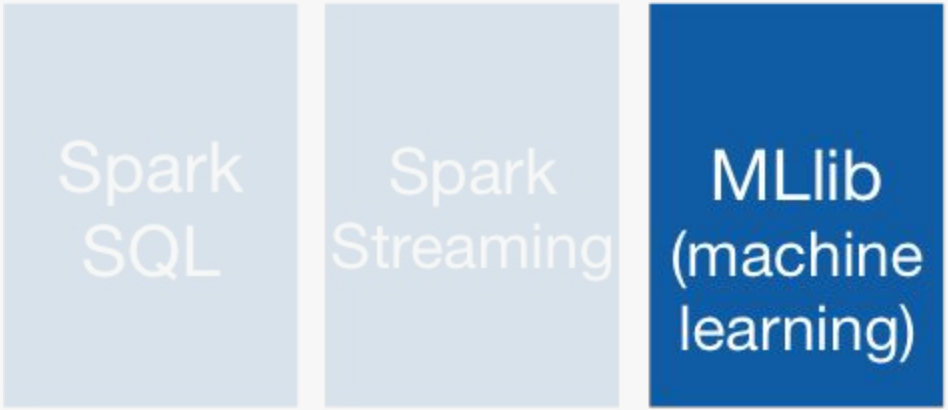
The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

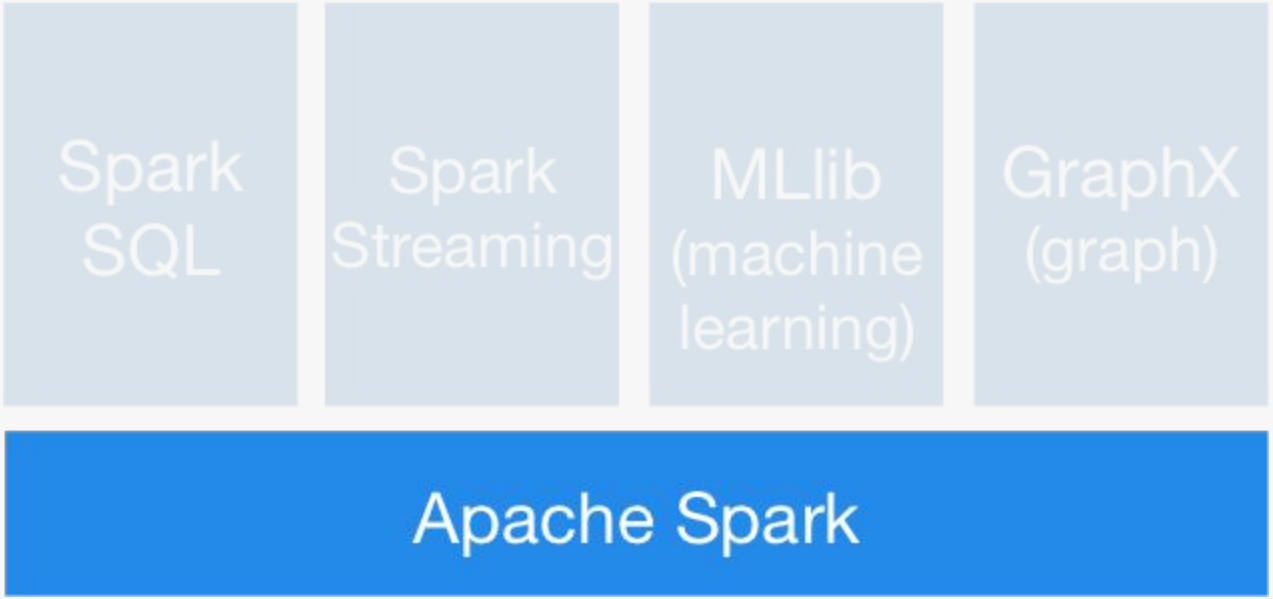


The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

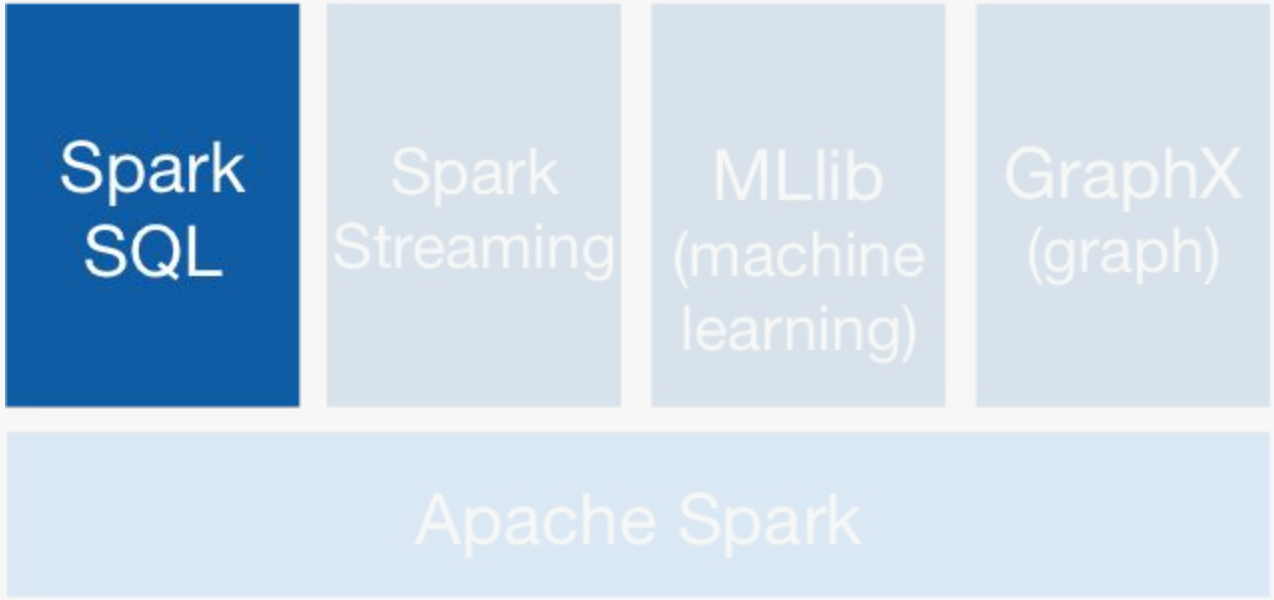


The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

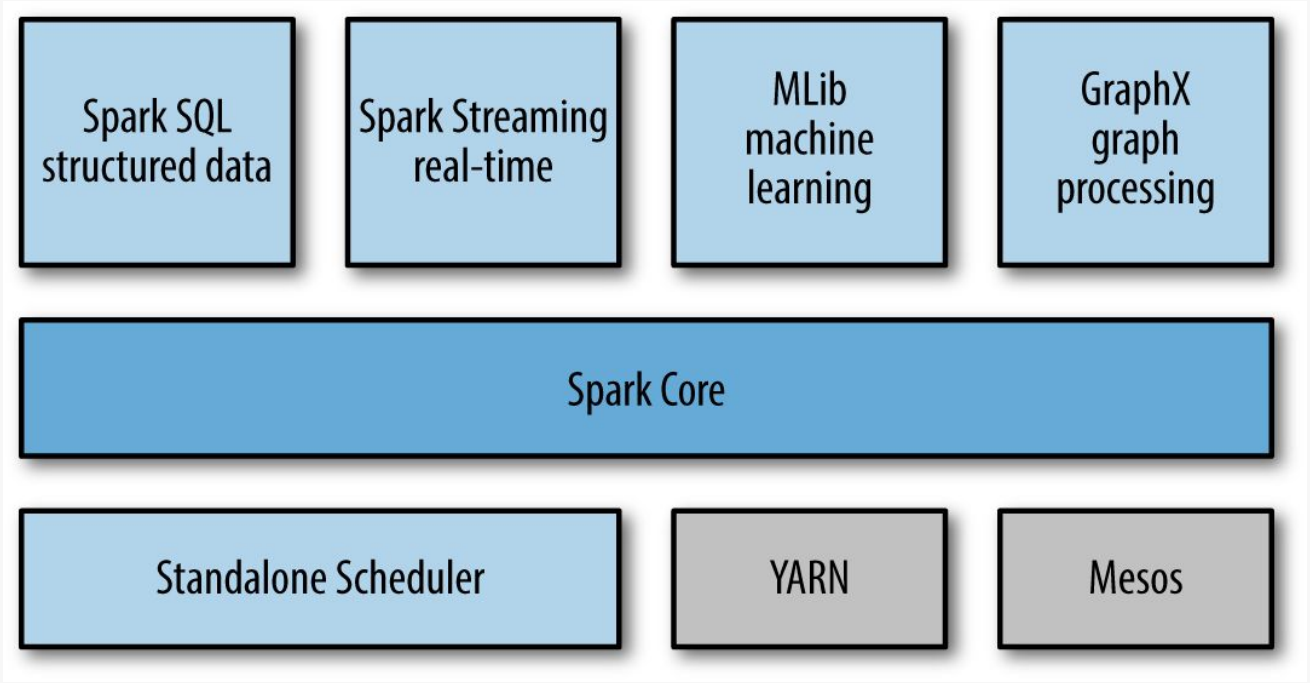
The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers



The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers



The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

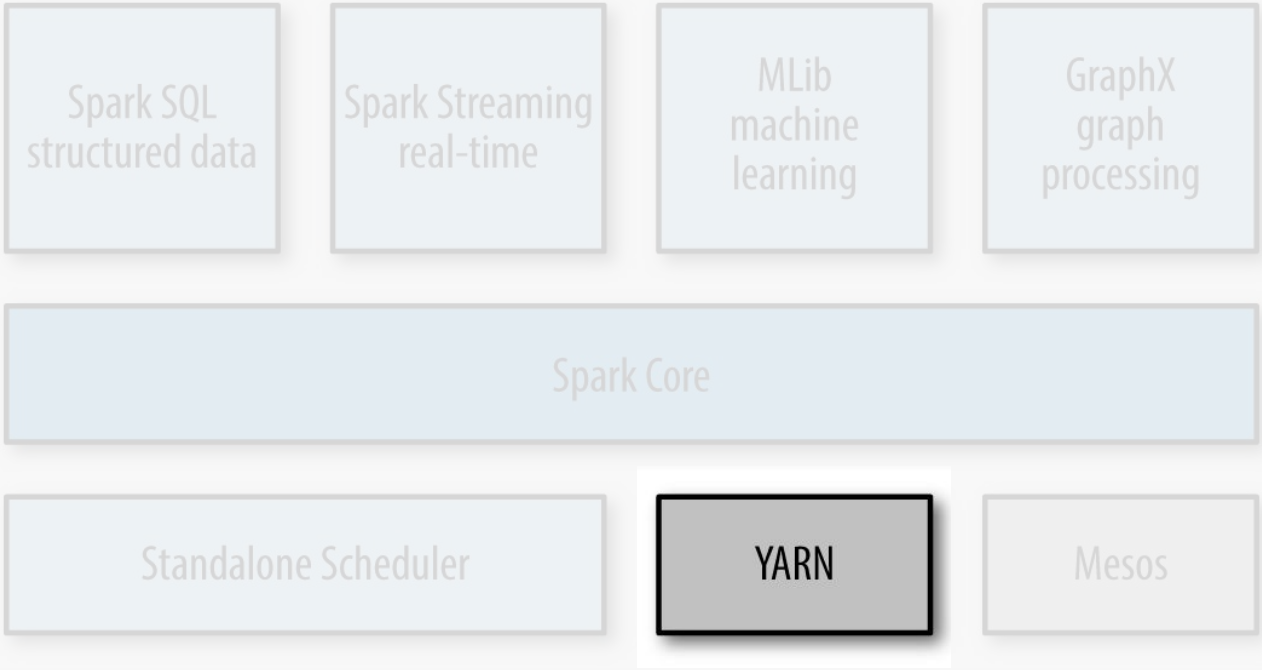


Cluster modes →

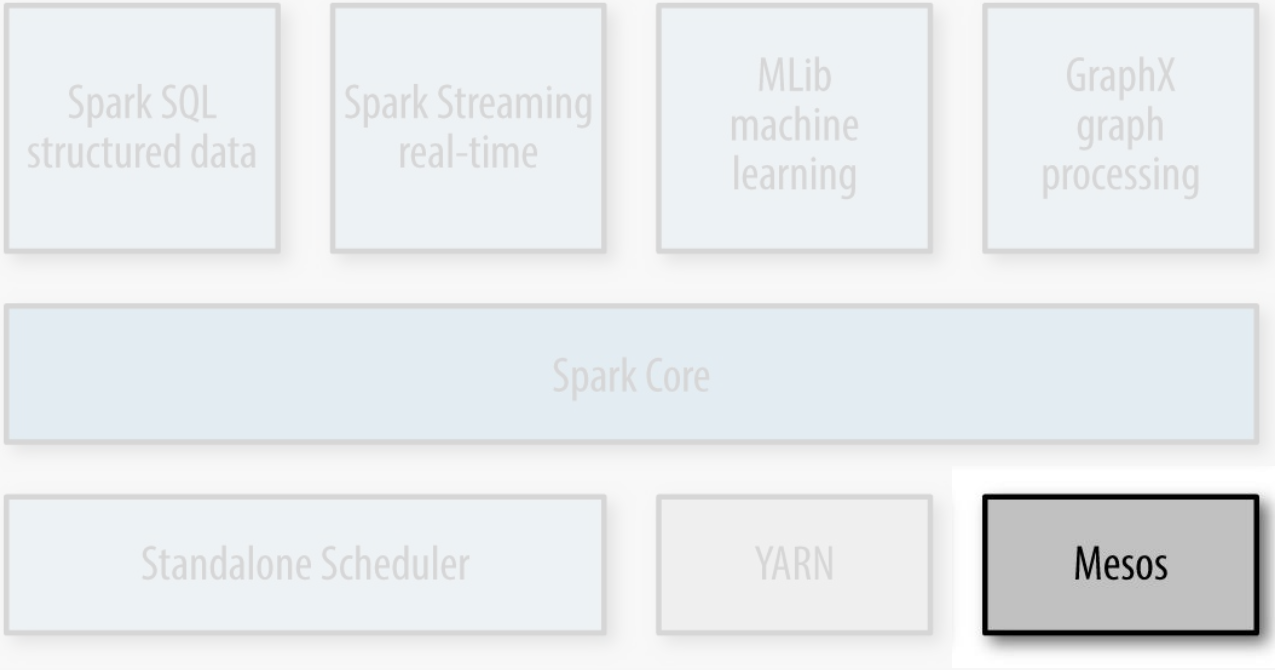
+

kubernetes

The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

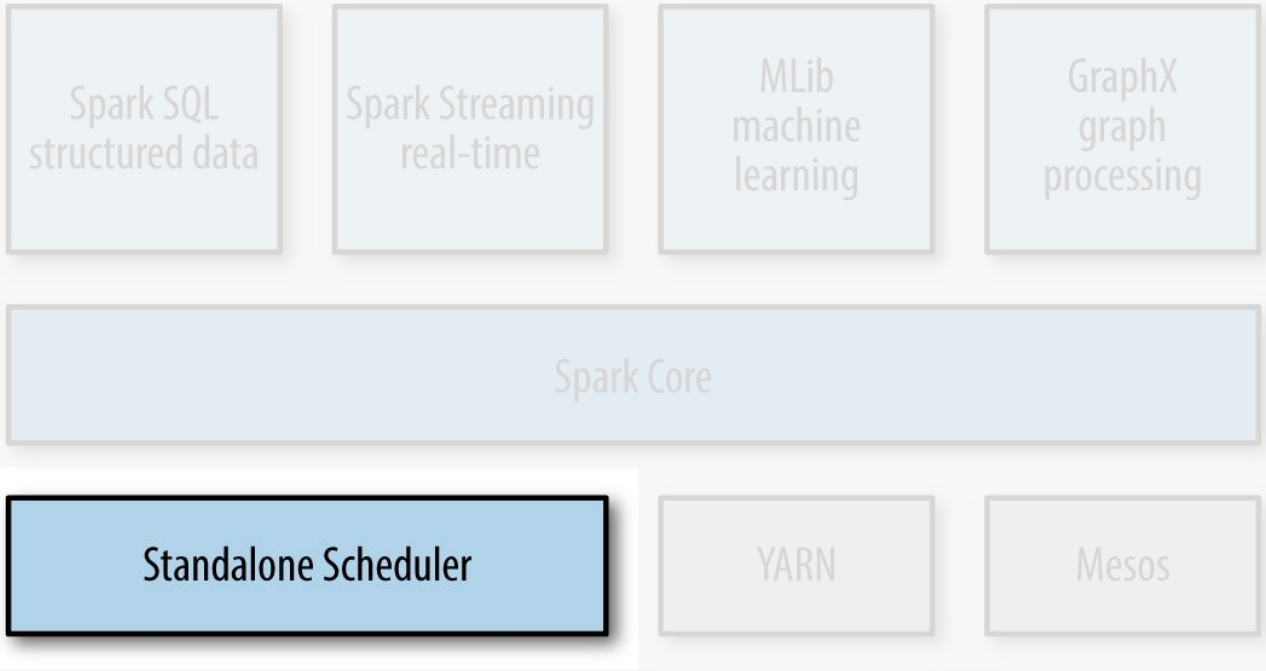


The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

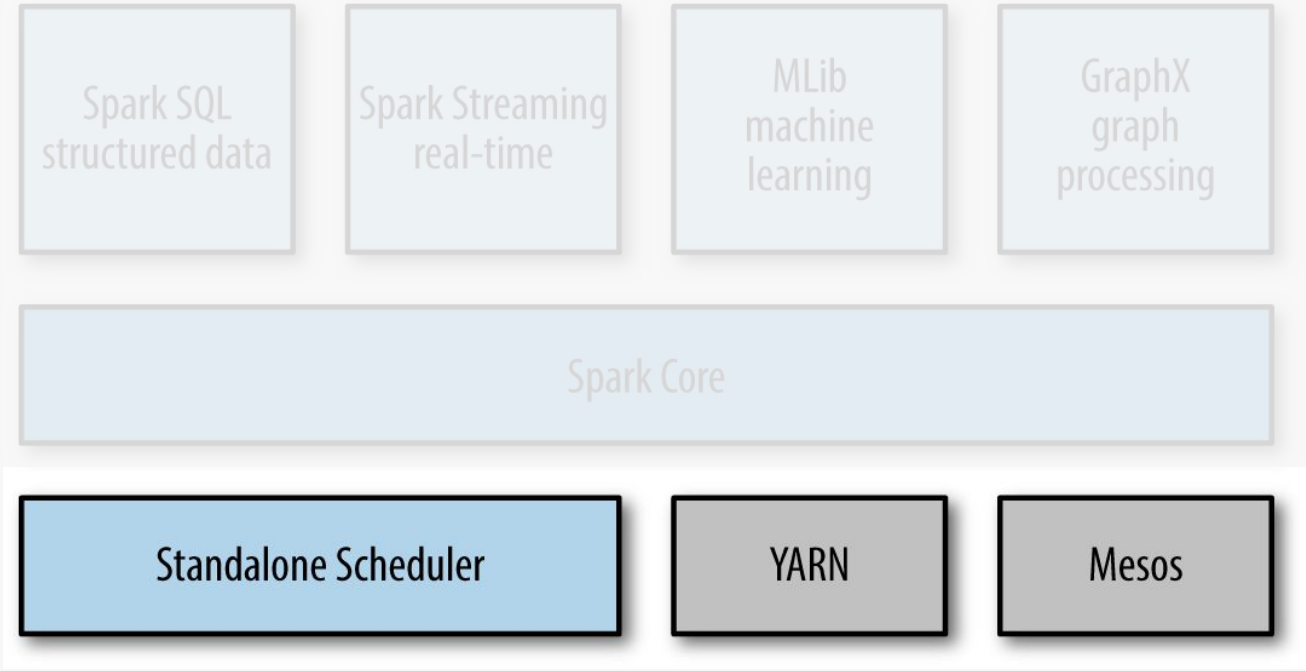


The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers



The Spark stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers



The main concepts behind Hadoop MapReduce can be explained with a deck of cards

Classroom Experiment: need 2 volunteers and a shuffled deck of cards.

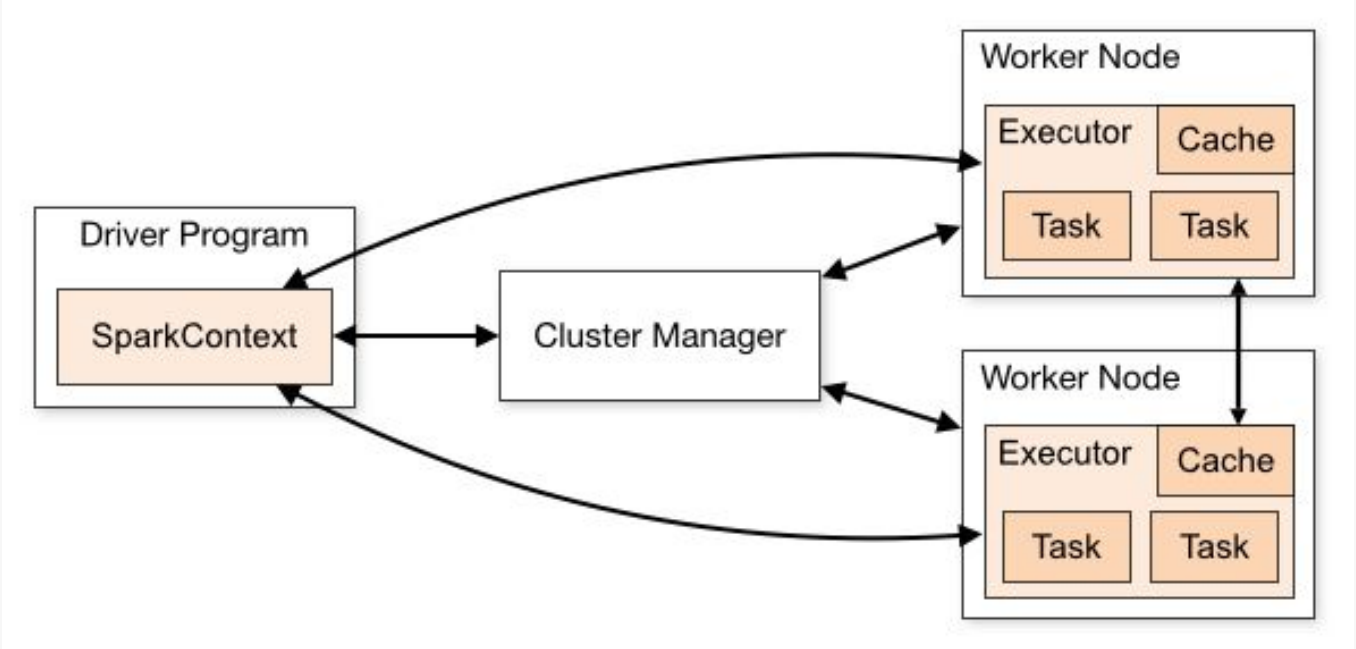
Simulate the computation of finding the largest card value per suit, assuming that non-numbered cards are “bad”.

Explain terms like node, process, shuffle, map and reduce. Master/worker.

Spark's main processes

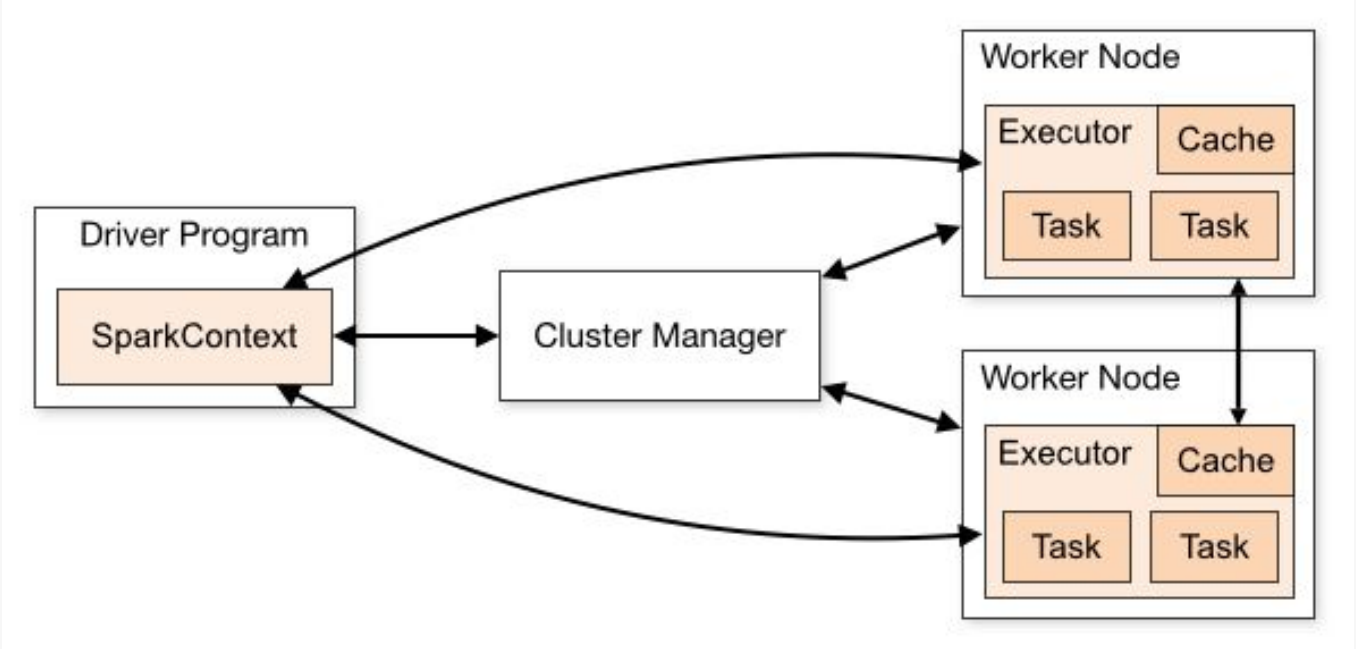
interprocess communication
among distributed actors

Communication between components in a Spark application happens by all actors



Communication between components in a Spark application happens by all actors

```
spark-submit
 SPARK_HOME?
 JAVA_HOME?
```



Which edge in this diagram has not been discussed? Can you come up with a reason for its existence?

Core concepts of Spark

what one typically interacts with in code

Core concepts of the Spark API

- RDDs
- Datasets & Dataframes
- Row
- Column
- SparkSession

Core concepts of the Spark API

- RDDs
- Datasets & Dataframes
- Row
- Column
- SparkSession

Demo with a PySpark application

Introduction to pytest

releasing with confidence
through automated testing

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.

- Improves chance of code still being **correct in the *future***

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.

- Improves chance of code still being **correct in the *future***
  - People tend to test their code *manually* on a small problem.

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.

- Improves chance of code still being **correct in the *future***
  - People tend to test their code *manually* on a small problem.
  - Code will change, as requirements and environments change.

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.

- Improves chance of code still being **correct in the *future***
  - People tend to test their code *manually* on a small problem.
  - Code will change, as requirements and environments change.
  - To prevent introducing breaking changes: write tests and ship these with the code.

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.
- Improves chance of code still being **correct in the *future***
  - People tend to test their code *manually* on a small problem.
  - Code will change, as requirements and environments change.
  - To prevent introducing breaking changes: write tests and ship these with the code.
- Most up-to-date form of **documentation**

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.
- Improves chance of code still being **correct in the *future***
  - People tend to test their code *manually* on a small problem.
  - Code will change, as requirements and environments change.
  - To prevent introducing breaking changes: write tests and ship these with the code.
- Most up-to-date form of **documentation**
  - Documents and wikis will grow out of sync with the code.

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.
- Improves chance of code still being **correct in the *future***
  - People tend to test their code *manually* on a small problem.
  - Code will change, as requirements and environments change.
  - To prevent introducing breaking changes: write tests and ship these with the code.
- Most up-to-date form of **documentation**
  - Documents and wikis will grow out of sync with the code.
  - Help you reason about pieces of code in the bigger picture.

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Raises confidence that code is **correct *now***
  - Assert that the results match expectations.
  - Handle edge cases.

- Improves chance of code still being **correct in the *future***
  - People tend to test their code *manually* on a small problem.
  - Code will change, as requirements and environments change.
  - To prevent introducing breaking changes: write tests and ship these with the code.

- Most up-to-date form of **documentation**
  - Documents and wikis will grow out of sync with the code.
  - Help you reason about pieces of code in the bigger picture.

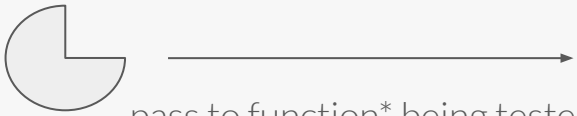
The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

prepare input data

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

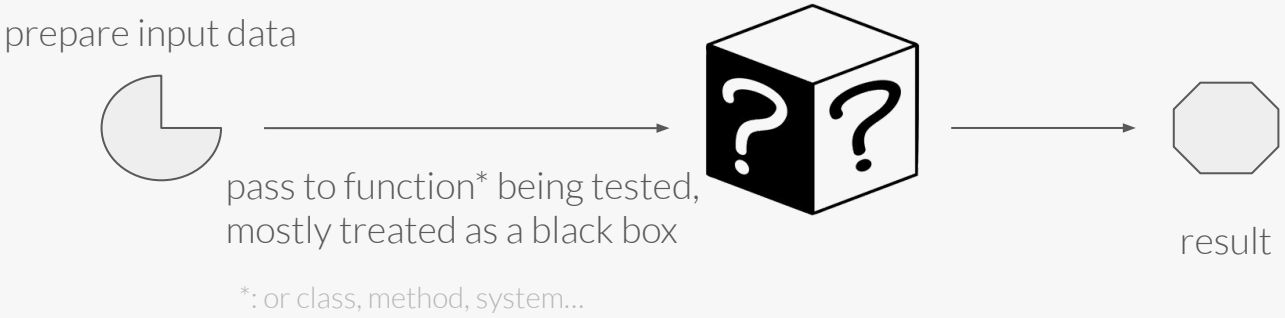
prepare input data



pass to function* being tested,
mostly treated as a black box

*: or class, method, system…

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations



The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

*: or class, method, system...

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

*: or class, method, system...

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

statement evaluating to a boolean

```
assert   2 != 3
```

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

	statement evaluating to a boolean	meaning
`assert`	`2 != 3`	the numerical value of 2 is not that of 3

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

	statement evaluating to a boolean	meaning
`assert`	`2 != 3`	the numerical value of 2 is not that of 3
`assert`	`len("hello") == len("world")`	

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

	statement evaluating to a boolean	meaning
`assert`	`2 != 3`	the numerical value of 2 is not that of 3
`assert`	`len("hello") == len("world")`	the strings “hello” and “world” have the same number of characters

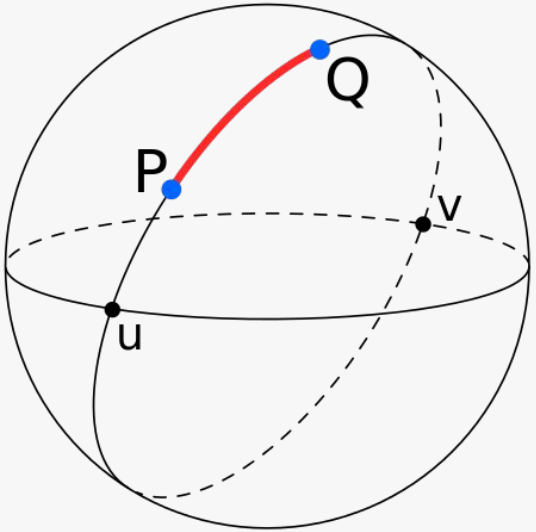
The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

	statement evaluating to a boolean	meaning
`assert`	`2 != 3`	the numerical value of 2 is not that of 3
`assert`	`len("hello") == len("world")`	the strings “hello” and “world” have the same number of characters
`assert`	`{1, 2, 3}.issubset(range(5))`	

The basic pattern in testing has three stages: setting up, creating a result and verifying expectations

	statement evaluating to a boolean	meaning
`assert`	`2 != 3`	the numerical value of 2 is not that of 3
`assert`	`len("hello") == len("world")`	the strings “hello” and “world” have the same number of characters
`assert`	`{1, 2, 3}.issubset(range(5))`	the former set is a subset of the latter collection

A warm-up to testing PySpark code: writing a unit test for the great-circle-distance function



The great-circle-distance (gcd) or Haversine distance gives the shortest distance *along* the surface of a sphere between any two points.

It is a commonly encountered problem in any project dealing with geographical data.

ASSIGNMENT:
exercises/b_unit_test_demo contains a faulty implementation of the Haversine formula. Through means of automated tests, show that the implementation is faulty.

Recap of useful methods

Because repetition is
a good way to learn

Little recap of the PySpark functionality you should be familiar with

```
spark = SparkSession.builder.getOrCreate()

df = spark.createDataFrame(
   data=[row1, row2, …, rowN],
   schema=sequence_of_column_names_or_a_structtype
)

df.select("id").withColumn("foo", psf.upper(psf.col("id"))).orderBy("foo").show()

spark.read.csv(some_path, **options).write.repartition(N).parquet(some_other_path)
```

Little recap of the PySpark functionality you should be familiar with

```
spark = SparkSession.builder.getOrCreate()

df = spark.createDataFrame(
   data=[row1, row2, …, rowN],
   schema=sequence_of_column_names_or_a_structtype
)

df.select("id").withColumn("foo", psf.upper(psf.col("id"))).orderBy("foo").show()

spark.read.csv(some_path, **options).write.repartition(N).parquet(some_other_path)
```

```
# for better readability:
df2 = (
    df
    .select("id")
    .withColumn(
        "foo",
        psf.upper(psf.col("id"))
    )
    .orderBy("foo")
)

df2.show()
```

Little recap of the PySpark functionality you should be familiar with

```
spark = SparkSession.builder.getOrCreate()

df = spark.createDataFrame(
   data=[row1, row2, …, rowN],
   schema=sequence_of_column_names_or_a_structtype
)

df.select("id").withColumn("foo", psf.upper(psf.col("id"))).orderBy("foo").show()

spark.read.csv(some_path, **options).write.repartition(N).parquet(some_other_path)
```

```
 # for better readability:
 (
     spark
     .read
     .csv(some_path, **options)
     .write
     .repartition(N)
     .parquet(some_other_path)
 )
```

When we read to understand the global picture, we scan text, which we do quicker from top to bottom.

Let a linter, which is a type of program, reformat your code automatically.

More DataFrame methods

Just enough for most use cases

Two of the more common methods you will find in code implementing business logic are groupby and join.

```
DataFrame.groupby(columns)
```

Allows you to perform “an analytical function” on groups.

Common analytical functions: sum, mean, max

Common example: sum the sales numbers per region

Two of the more common methods you will find in code implementing business logic are groupby and join.

```
DataFrame.groupby(columns)
```

Allows you to perform “an analytical function” on groups.

Common analytical functions: sum, mean, max

Common example: sum the sales numbers per region

```
DataFrame.join(other_dataframe, on=..., how=...)
```

Merge two dataframes “horizontally”

join types (the “`how`” keyword) correspond to those found in any relational database system.

Two of the more common methods you will find in code implementing business logic are groupby and join.

```
DataFrame.groupby(columns)
```

Allows you to perform “an analytical function” on groups.

Common analytical functions: sum, mean, max

Common example: sum the sales numbers per region

```
DataFrame.join(other_dataframe, on=..., how=...)
```

Merge two dataframes “horizontally”

join types (the “`how`” keyword) correspond to those found in any relational database system.

DEMO.

Adding moderately complex columns

going beyond simple math

Many datasets involve a time series column. Often we'd like to enrich those time indications with other information, which only depends on the time value inside the same row.

The holidays module offers a good playground for functionality that isn't built in to Spark.

It's also a common "feature" in machine learning applications.

ASSIGNMENT:
In *exercises/c_labellers*, extend the function
`is_belgian_holiday`
so that it can be used to validate whether a date instance is also a Belgian holiday.

Validate your logic using a test.

Testing the logic of functions that involve PySpark DataFrames isn't as easy as a regular unit test, because of the interprocess communication overhead.

Observation	Remedy or suggestion

Testing the logic of functions that involve PySpark DataFrames isn't as easy as a regular unit test, because of the interprocess communication overhead.

Observation	Remedy or suggestion
Spark tests are slow	• Understand when to cache results • Create only a single SparkSession for the entire test suite

Testing the logic of functions that involve PySpark DataFrames isn't as easy as a regular unit test, because of the interprocess communication overhead.

Observation	Remedy or suggestion
Spark tests are slow	• Understand when to cache results • Create only a single SparkSession for the entire test suite
Tests refer to data in files	Create DataFrames from tuples of well-chosen examples, close to the function you're testing, preferably in the same module.

Testing the logic of functions that involve PySpark DataFrames isn't as easy as a regular unit test, because of the interprocess communication overhead.

Observation	Remedy or suggestion
Spark tests are slow	• Understand when to cache results • Create only a single SparkSession for the entire test suite
Tests refer to data in files	Create DataFrames from tuples of well-chosen examples, close to the function you're testing, preferably in the same module.
Only checking the number of rows (and columns) in DataFrames	This is typically not what you're after. The test should likely go deeper and inspect contents. That's a more involved process, because of the distributed nature of the data. □ assignment

Testing the logic of functions that involve PySpark DataFrames isn't as easy as a regular unit test, because of the interprocess communication overhead.

Observation	Remedy or suggestion
Spark tests are slow	• Understand when to cache results • Create only a single SparkSession for the entire test suite
Tests refer to data in files	Create DataFrames from tuples of well-chosen examples, close to the function you're testing, preferably in the same module.
Only checking the number of rows (and columns) in DataFrames	This is typically not what you're after. The test should likely go deeper and inspect contents. That's a more involved process, because of the distributed nature of the data. □ assignment

ASSIGNMENT:

Write a helper function for tests that allows you to compare two DataFrames for "functional equivalence": column and row order shouldn't matter, as long as the values can be found in both frames in the same columns and associated with identical rows. Validate your logic using tests/test_comparers.py

User defined functions (UDFs) are a wrapper over pure Python functions. Their typed nature reduces the general character of the original function.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

def square(x):
    return x**2

square_udf_int = udf(lambda z: square(z), IntegerType())

(
    df
    .select(
        "integers",
        "floats",
        square_udf_int("integers").alias("int_squared"),
        square_udf_int("floats").alias("float_squared")
    )
    .show()
)
```

nameless function

The intended return type

"This function returns a Python object, which can be converted to this specified PySpark datatype"

User defined functions suffer from serialization overhead and a large number of invocation calls, making them slow.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

def square(x):
    return x**2

square_udf_int = udf(lambda z: square(z), IntegerType())

(
    df
    .select(
        "integers",
        "floats",
        square_udf_int("integers").alias("int_squared"),
        square_udf_int("floats").alias("float_squared")
    )
    .show()
)
```

The intended return type

"This function returns a Python object, which can be converted to this specified PySpark datatype"

nameless function

A lambda (nameless function) for a function that takes the lambda's argument when called, is silly, and another layer of overhead.

- high serialization overhead
- large number of invocation calls

User defined functions are not the only way to implement functionality. An incredible amount of work can be done with what Spark provides you.

- The functionality is likely already in pyspark.sql.functions
- You could also look into redesigning the algorithm
- Or go low-level with `DataFrame.rdd.mapPartitions`

We'll see examples of the first two in the exercises.

User defined functions are not the only way to implement functionality. An incredible amount of work can be done with what Spark provides you.

- The functionality is likely already in pyspark.sql.functions
- You could also look into redesigning the algorithm
- Or go low-level with `DataFrame.rdd.mapPartitions`

We'll see examples of the first two in the exercises.

ASSIGNMENT:
In *exercises/c_labellers/dates.py*, implement the desired functionality described in the docstrings of the functions `label_weekend` and `label_holidays`

Validate your logic using tests.

Lazy evaluation

postponing work to be more efficient

Pop quiz: action or transformation?

df = spark.range(0)

other_df = spark.range(0)

- df.withColumn("foo", lit(5))
- df.join(other_df, on=["id"], how="left")
- df.count
- df.count()
- df.rdd
- df.collect()
- df.describe()
- df.groupBy("id").count()
- df.groupBy("id").agg(sum(lit(1)).alias("count"))
- df.take(4)
- df.schema
- df.cache()

- spark.read.parquet(some_file)

- transformation
- transformation
- neither: simply a reference to a bound method
- action
- transformation
- action
- transformation
- transformation! Note: this count is a convenience method. It replaces an aggregator.
- transformation: same result as above
- action
- neither: simply an attribute of a DataFrame
- transformation, and just like all others: a lazy one! The DataFrame won't be cached until the next action.
- tricky one: some work is done, because the file's metadata is read (e.g. the schema), but the entire file isn't processed. It is about DataFrame creation, so your DataFrame isn't there yet. You're creating an instance of the class, and some minor work is being done

Generating errors in light of lazily evaluated functions

Check out the *exercises/d_laziness/date_helper.py* module.

1. What is the intention of the author of this function?
2. Will it work?

Generating errors in light of lazily evaluated functions

Check out the *exercises/d_laziness/date_helper.py* module.

1. What is the intention of the author of this function?
2. Will it work?

ASSIGNMENT:
Check out *exercises/d_laziness/date_helper.py*.

Prove that it does/does not work using a test (no need to be rigorous, and it's the last one, I promise!)

Lazily executed functionality emphasizes the need for tests to speed up development.

Spark's **execution model is lazy:** transformations won't be triggered until an action is called for. So **errors won't be thrown until runtime.** This puts even more emphasis on the need for tests to speed up development…

… unless you love waiting until your code is deployed to a dev environment, only to find out silly things, like typos in column names.

A DataFrame isn't really a distributed collection of data. It's an **execution plan**, with references to the source data. That execution plan is heavily optimized by Spark's Catalyst process.

`DataFrame.explain()`

Action or transformation? Neither: the workers don't do anything, so not an action, but it also doesn't create a new DataFrame (queryplan isn't modified).

Condensed form (default): shows you the execution plan, which is optimized. Learn how to read and reason on this, as it may indicate performance bottlenecks.

Non-condensed form:

- *Parsed Logical Plan*
  Spark just reads your description of a pipeline (in sequence) and parses it to its own language and datatypes
- *Analyzed Logical Plan*
  Spark tries to identify unbound columns: column references made later in the pipeline refer to columns made earlier. Spark links these, through internal identification numbers (the number suffixes, prepended with #, that you see in a plan)
- *Optimized Logical Plan*
  Spark's Catalyst process has optimized the analyzed plan with several SQL "tricks" and knowledge about the dataset sizes: filters get pushed "down", to the source & some joins get executed more efficiently (e.g. broadcast joins)
- *Physical Plan*
  The actual implementation of the optimized plan, with references to the file system and the pushed down filters.

Exercises: observe Catalyst's query optimization in action

ASSIGNMENT:

Find the query_plan_demo module and answer the questions therein.

Exercises: observe Catalyst's query optimization in action

ASSIGNMENT:
Find the query_plan_demo module and answer the questions therein.

Execute (once only!) the predicate_pushdown script. Observe how the filter gets moved into the reader. If the source format supports it, the filtering will be done using the tooling available from that format (think about database connections here using `spark.read.jdbc` or indeed, Parquet and ORC, which support this). Pushing filters closer to the data source is known as **predicate pushdown.**

Data serialization formats

The big three

Avro, Parquet and ORC are three **self-describing** serialization formats used commonly in big data pipelines.

A 2016 study done at Hortonworks compares the file sizes, full scan throughput and partial scan throughput of these 3 formats using various compression techniques and for differently sized, but realistic datasets (NYC taxi data, Github actions & sales data).

The key takeaways are these:

Avro is a **row-oriented** file format, so typically better in streaming cases (e.g. Kafka), as the full record needs to be processed

Apache Parquet is a **columnar storage** format, making it great for analytics. Data is split in larger chunks (row groups) which allow it to be easily splittable, for parallel processing.

ORC, short for Optimized Row Column format, is **similar to Parquet** in its uses.

Understand the usage patterns of your data and run benchmarks.
The formats get new features over time (ORC supports column encryption e.g.).

Partitions and shuffles

🎵 *everyday I'm shufflin'*

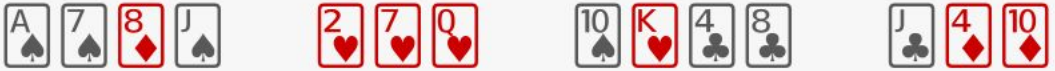
Partitions are simply chunks of data that are processed by executors. How the data is partitioned affects the cluster's performance.

Partitions are simply chunks of data that are processed by executors. How the data is partitioned affects the cluster's performance.

one possible partitioning

Partitions are simply chunks of data that are processed by executors. How the data is partitioned affects the cluster's performance.

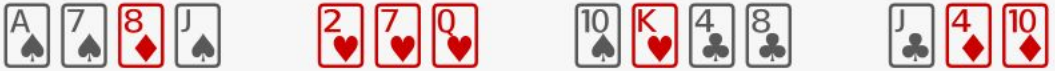
one possible partitioning



another partitioning

Partitions are simply chunks of data that are processed by executors. How the data is partitioned affects the cluster's performance.

one possible partitioning

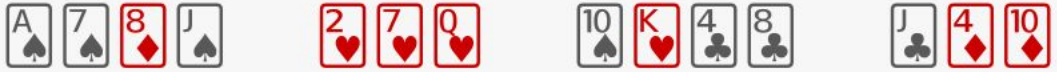


another partitioning

an improbable partitioning

Partitions are simply chunks of data that are processed by executors. How the data is partitioned affects the cluster's performance.

one possible partitioning



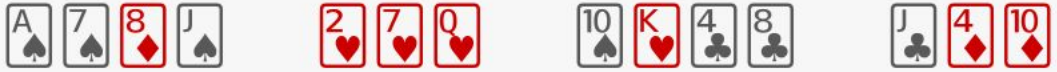
another partitioning

an improbable partitioning

an unbalanced partitioning

Partitions are simply chunks of data that are processed by executors. How the data is partitioned affects the cluster's performance.

one possible partitioning



another partitioning

an improbable partitioning

an unbalanced partitioning

another partitioning

Some DataFrame actions cause shuffles: partitions being redistributed over the network. Network traffic is typically detrimental to the performance of a Spark job.

A **stage boundary** marks the start of a shuffle operation.

Repeat the exercise related to the query plan analysis. This time, ensure the SparkSession isn't terminated early (e.g. add a `time.sleep(2*60)`). Go to the Spark UI while the session is active.

How many stages are there involved?

Sometimes, using a different algorithm, the number of stages can be decreased. This requires ingenuity. The programmer's time is also costly, so make an educated assumption about whether it will be worth the cost.

There's an optional exercise related to this stage reduction, to get the message across. Two data transformation pipelines give the exact same result, but one has a stage less and should be preferred, though a benchmark should provide an objective evaluation..

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause
One partition takes much longer to process than others	

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause
One partition takes much longer to process than others	**Data skew**: one or a few partitions are much larger than the others.

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause
One partition takes much longer to process than others	**Data skew**: one or a few partitions are much larger than the others.
Some executors are idling at the end of a stage.	

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause
One partition takes much longer to process than others	**Data skew**: one or a few partitions are much larger than the others.
Some executors are idling at the end of a stage.	Number of partitions is not a multiple of the number of cores assigned to the job.

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause
One partition takes much longer to process than others	**Data skew**: one or a few partitions are much larger than the others.
Some executors are idling at the end of a stage.	Number of partitions is not a multiple of the number of cores assigned to the job.

Solutions?

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause
One partition takes much longer to process than others	**Data skew**: one or a few partitions are much larger than the others.
Some executors are idling at the end of a stage.	Number of partitions is not a multiple of the number of cores assigned to the job.

Solutions?

- DataFrame.repartition(numPartitions, *cols)

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause
One partition takes much longer to process than others	**Data skew**: one or a few partitions are much larger than the others.
Some executors are idling at the end of a stage.	Number of partitions is not a multiple of the number of cores assigned to the job.

Solutions?

- DataFrame.repartition(numPartitions, *cols)
- DataFrame.coalesce(lowerNumberOfPartitions)

Sometimes forcing a shuffle can improve the performance of a job.

Observation	Probable root cause
One partition takes much longer to process than others	**Data skew**: one or a few partitions are much larger than the others.
Some executors are idling at the end of a stage.	Number of partitions is not a multiple of the number of cores assigned to the job.

Solutions?

- DataFrame.repartition(numPartitions, *cols)
- DataFrame.coalesce(lowerNumberOfPartitions)
- config option: `spark.sql.shuffle.partitions`

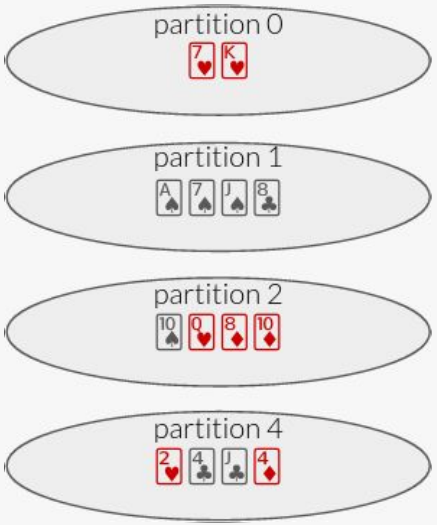
Example of starving executors and the magnificence of cloud computing

`set` **`spark.sql.shuffle.partitions`** to 3000
and cluster change (number of cores x2, memory/2, so same cost of cloud cluster)

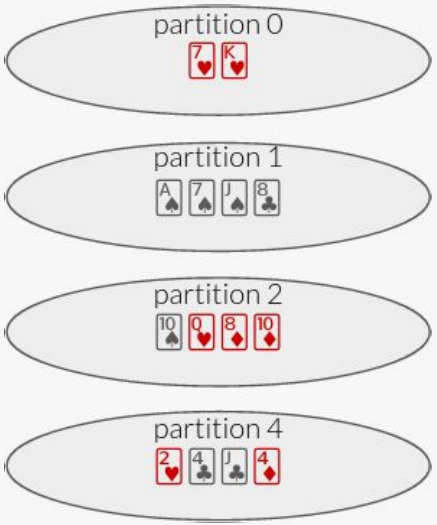
`spark.sql.shuffle.partitions`
→ default = 200
increasing means more, but smaller partitions. It *can* improve a cluster's efficiency, but is not a guarantee: it depends on the situation and you should back it up with research, like these Ganglia charts and Spark UI graphs.

`repartition` shuffles all the data over the network and should be used mainly to create balanced partitions

`repartition` shuffles all the data over the network and should be used mainly to create balanced partitions

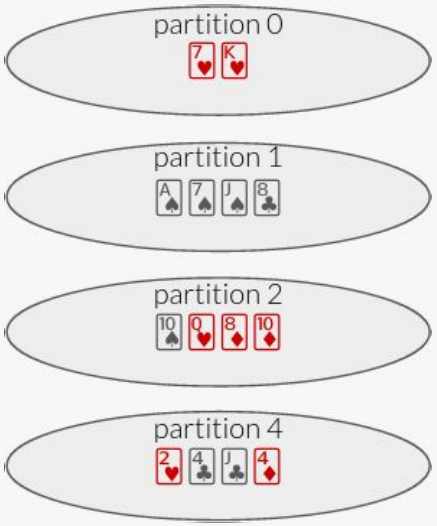


`repartition` shuffles all the data over the network and should be used mainly to create balanced partitions




```
repartition(3, "suit")
```

`repartition` shuffles all the data over the network and should be used mainly to create balanced partitions




```
repartition(3, "suit")
```

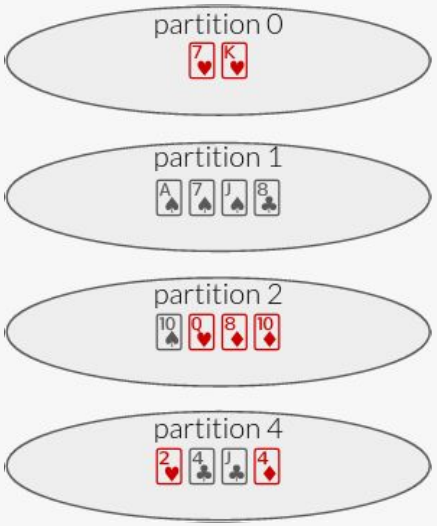
custom_hash(♠) % 3 = 1 -> partition 1 will contain spades

custom_hash(♥) % 3 = 2 -> partition 2 will contain hearts

custom_hash(♣) % 3 = 0 -> partition 0 will contain clubs

custom_hash(♦) % 3 = 2 -> partition 2 will contain diamonds

`repartition` shuffles all the data over the network and should be used mainly to create balanced partitions



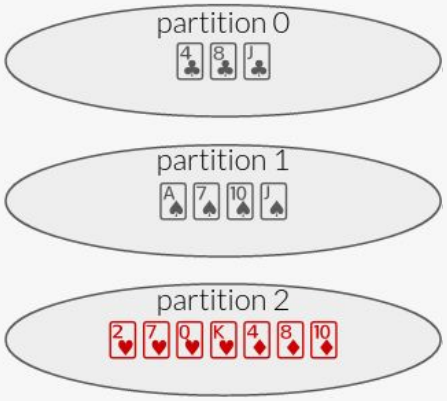
`repartition(3, "suit")`

custom_hash(♠) % 3 = 1 -> partition 1 will contain spades

custom_hash(♥) % 3 = 2 -> partition 2 will contain hearts

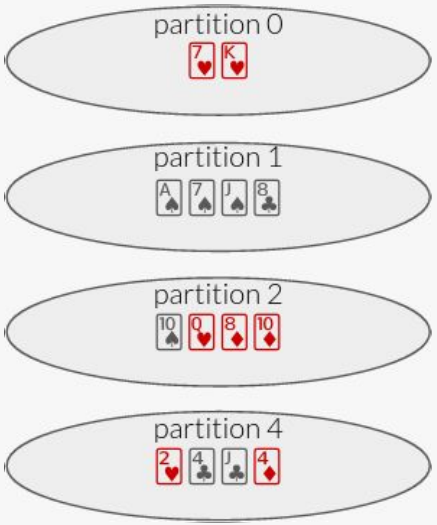
custom_hash(♣) % 3 = 0 -> partition 0 will contain clubs

custom_hash(♦) % 3 = 2 -> partition 2 will contain diamonds



*unbalanced, due to parameter choice

`repartition` shuffles all the data over the network and should be used mainly to create balanced partitions



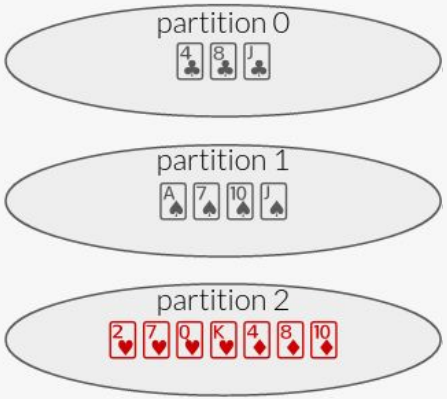
`repartition(3, "suit")`

custom_hash(♠) % 3 = 1 -> partition 1 will contain spades

custom_hash(♥) % 3 = 2 -> partition 2 will contain hearts

custom_hash(♣) % 3 = 0 -> partition 0 will contain clubs

custom_hash(♦) % 3 = 2 -> partition 2 will contain diamonds

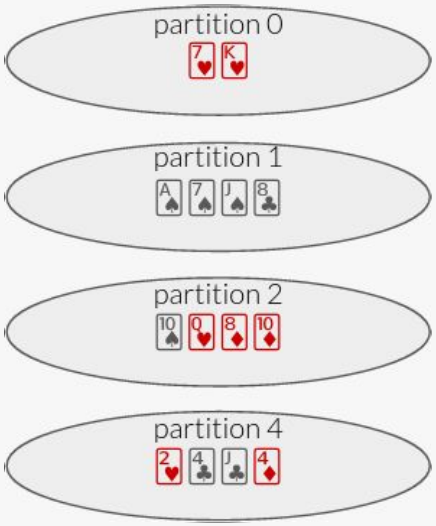


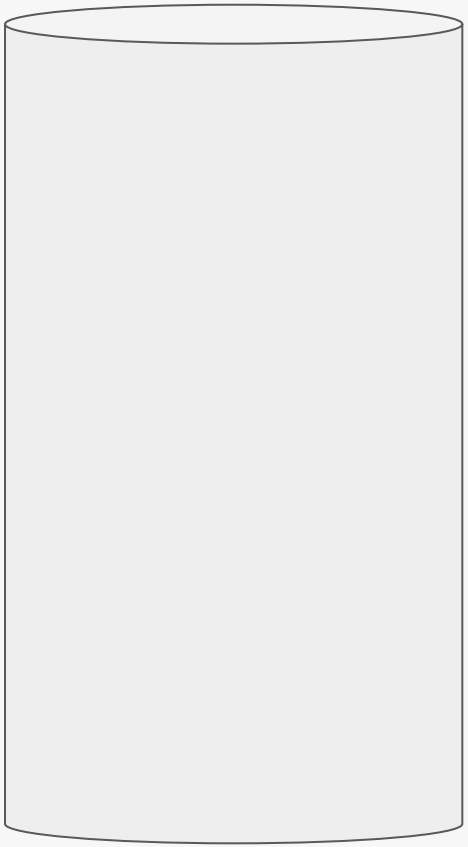
*unbalanced, due to parameter choice

If you don't specify `numPartition` (3) then Spark will use the default configuration value (200).
If you don't specify a **column** (or sequence of columns), Spark will feed the entire row to the hashing function.

`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

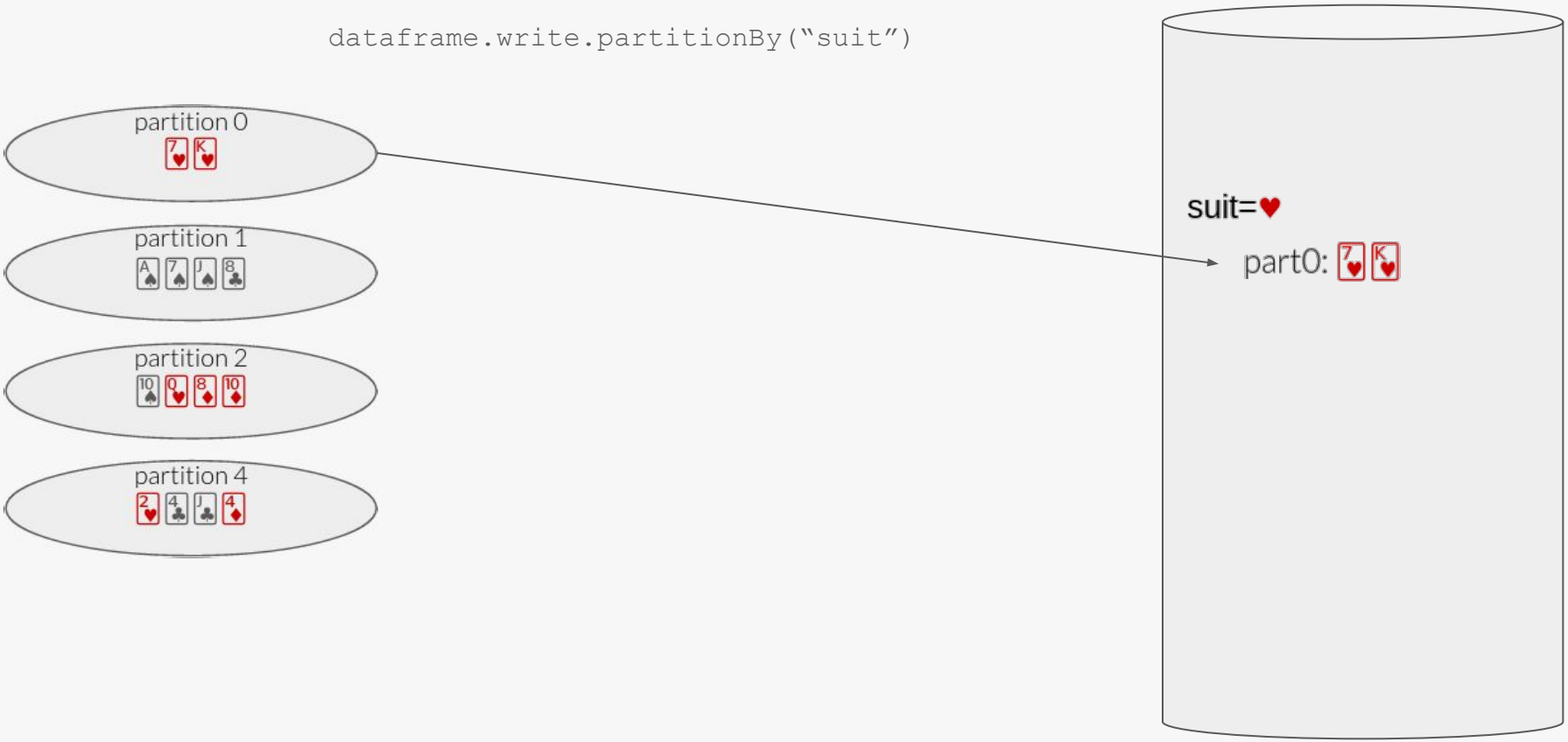
```
dataframe.write.partitionBy("suit")
```





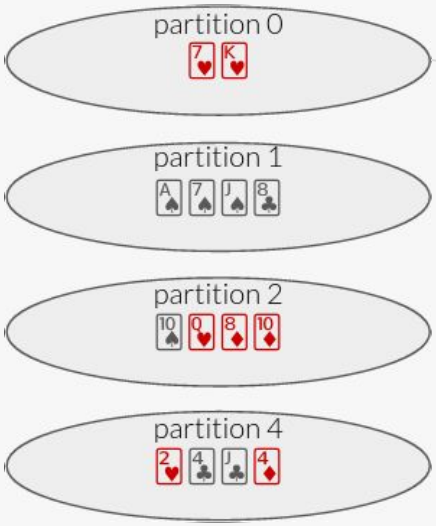
`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

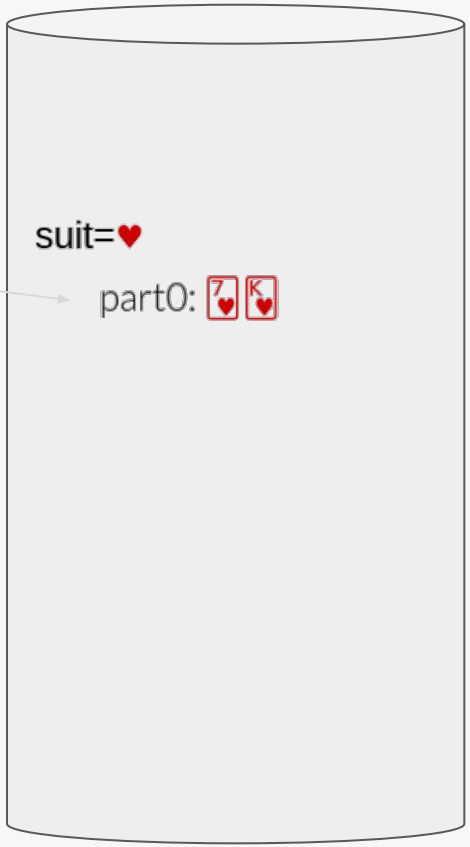
```
dataframe.write.partitionBy("suit")
```



`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

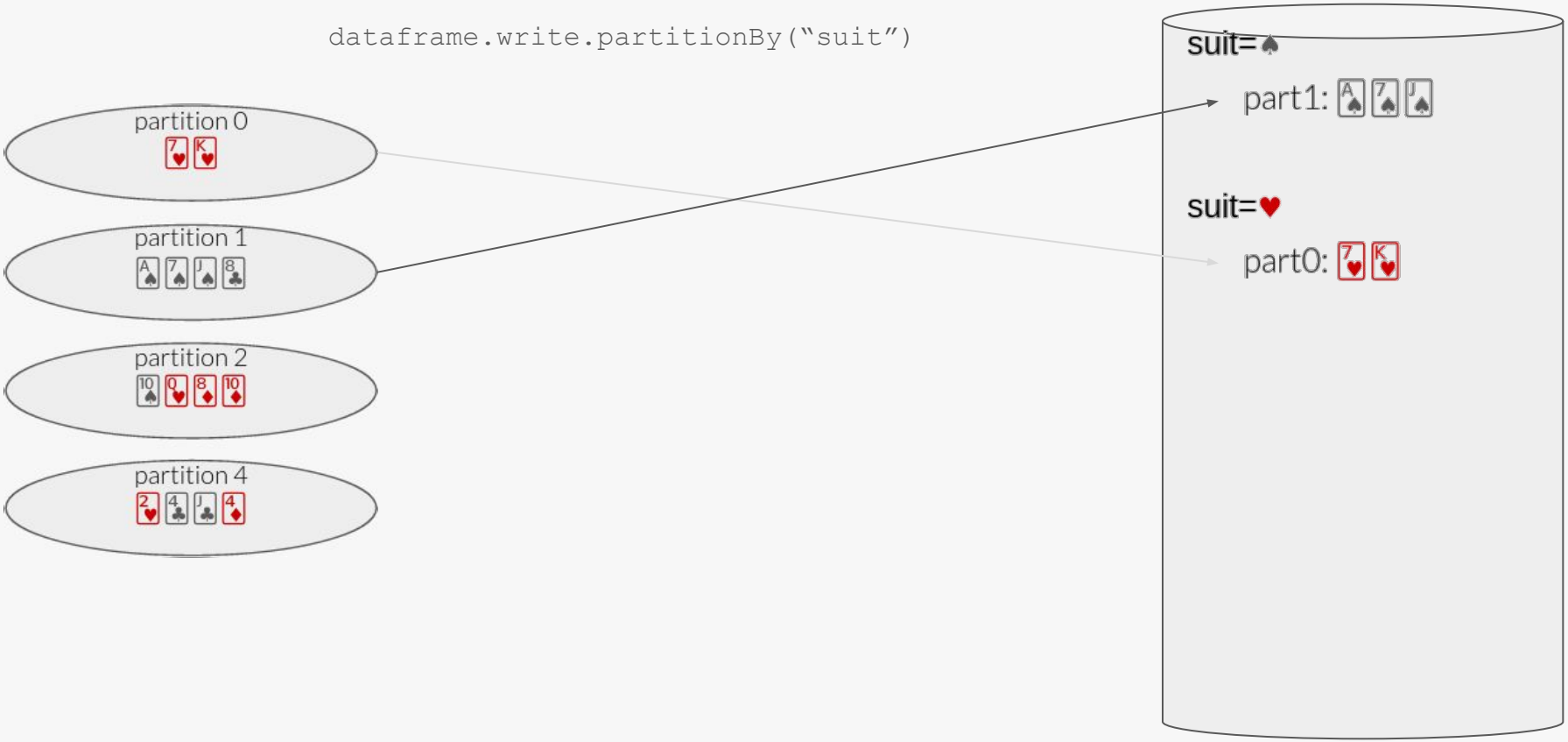
```
dataframe.write.partitionBy("suit")
```





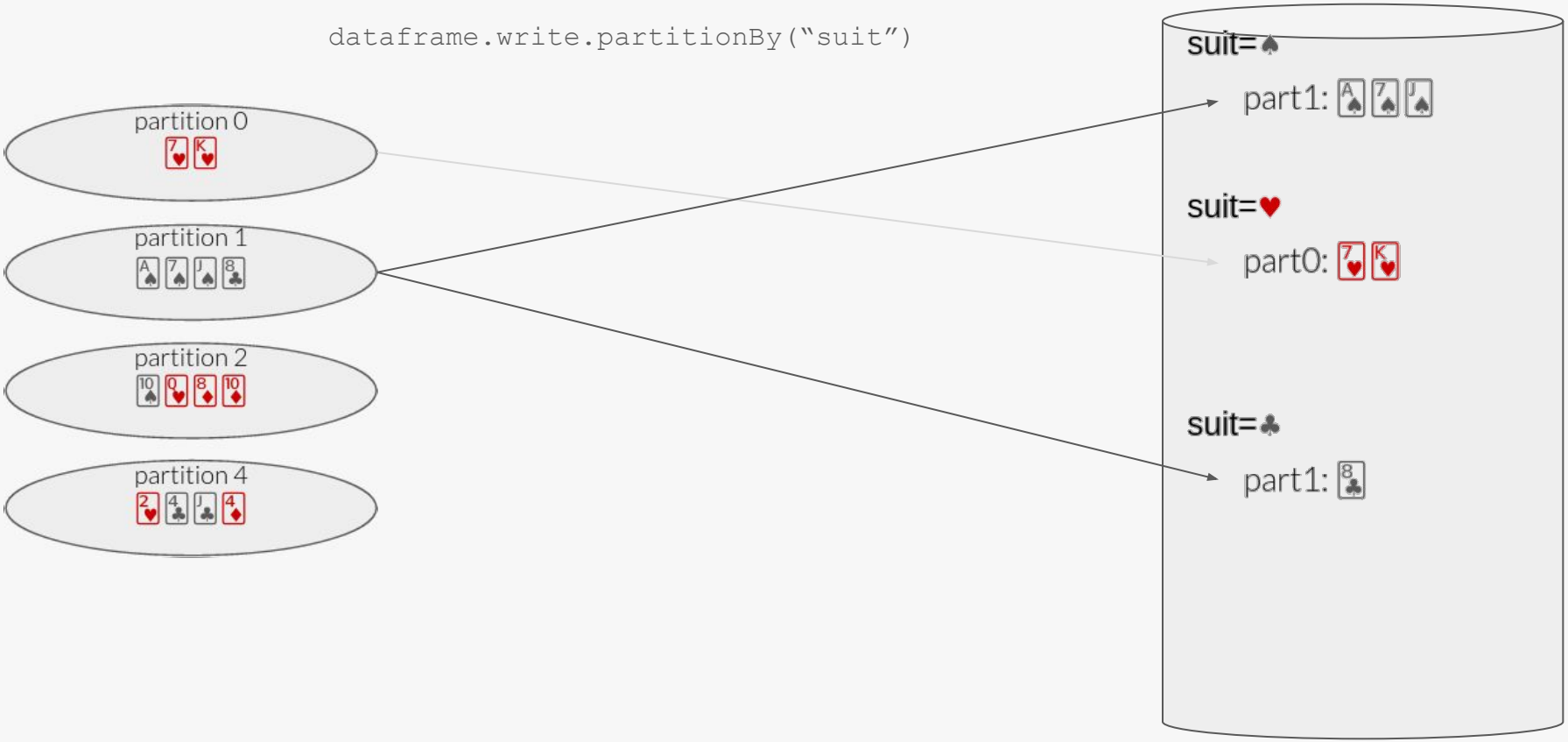
`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```



`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```

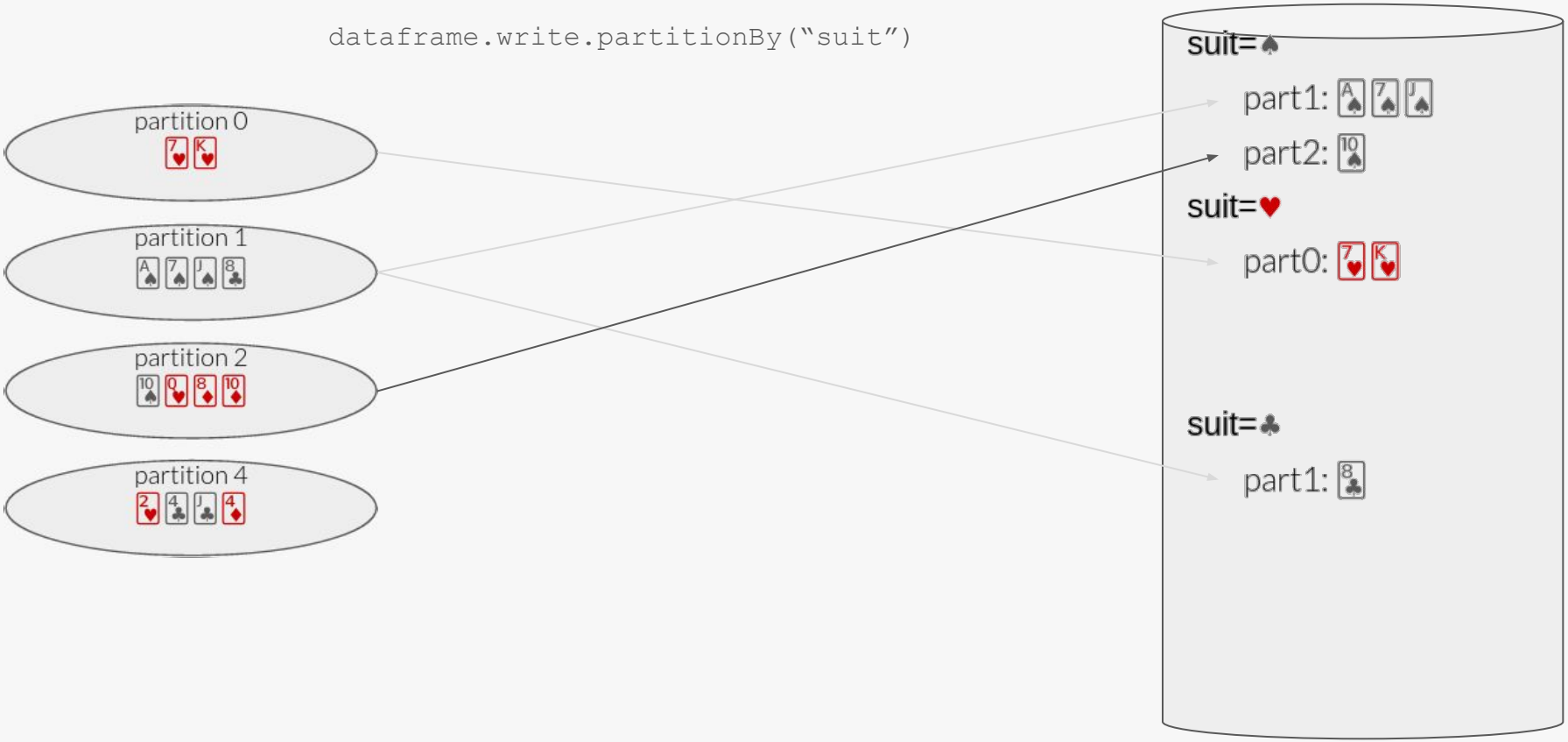


`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```

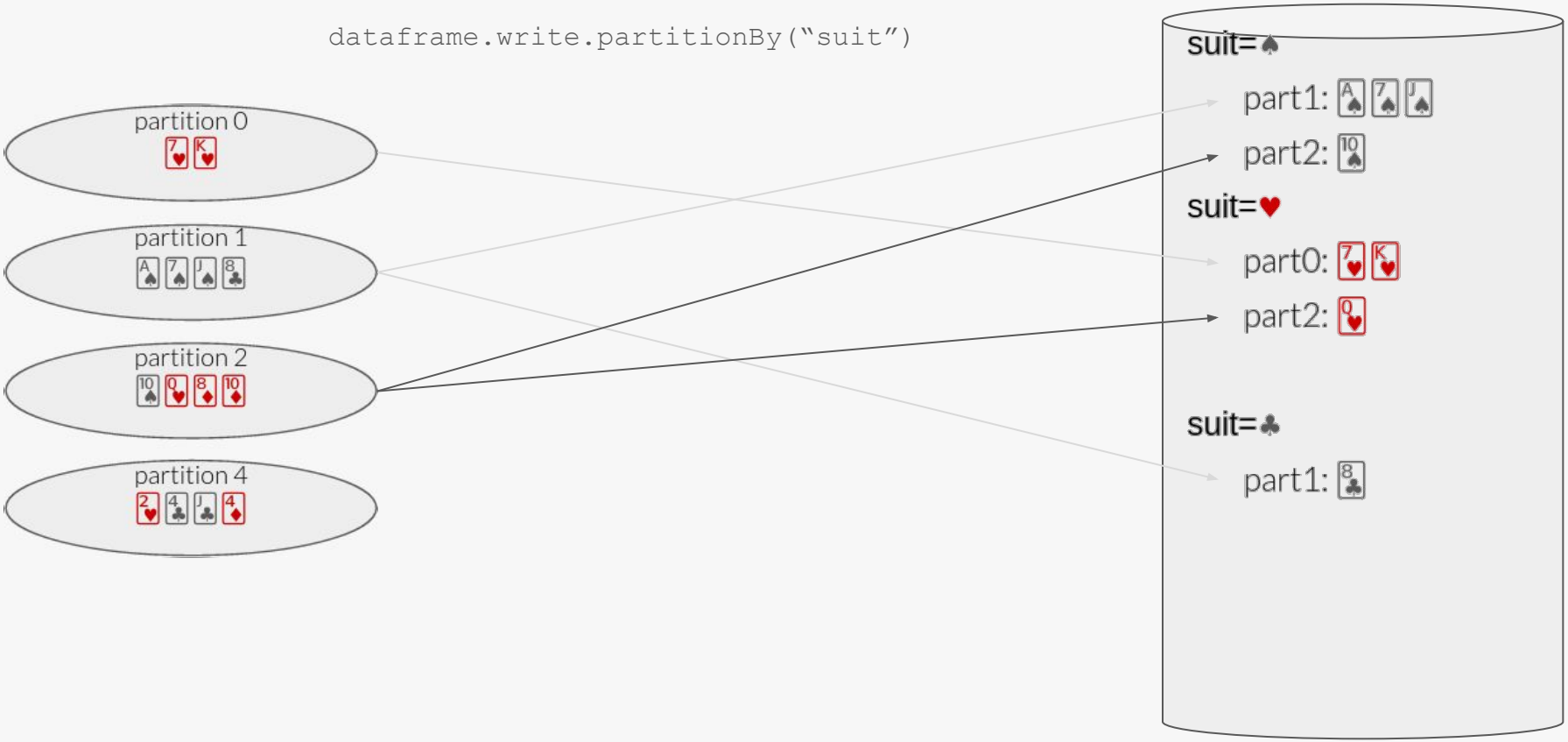
`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```



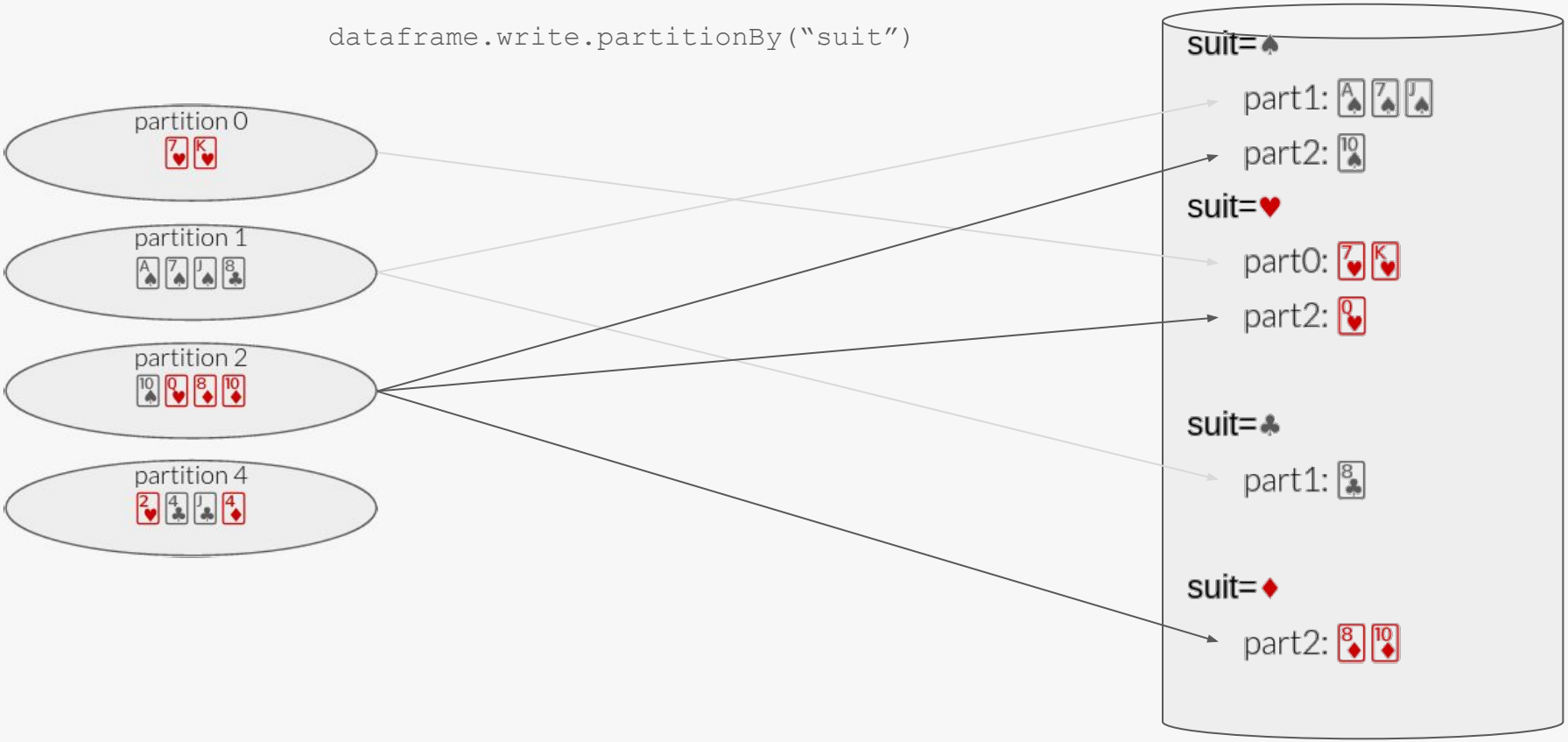
`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```



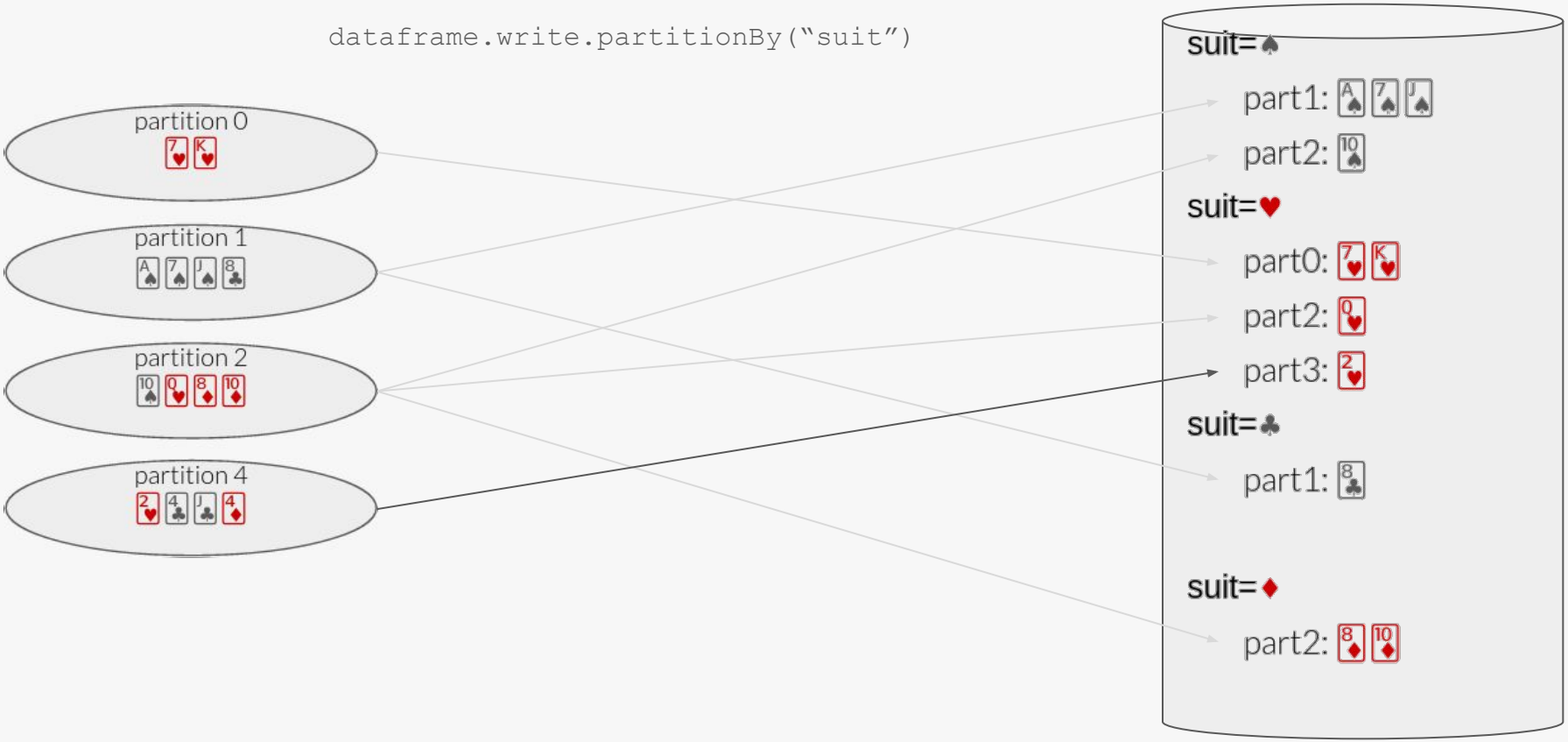
`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```



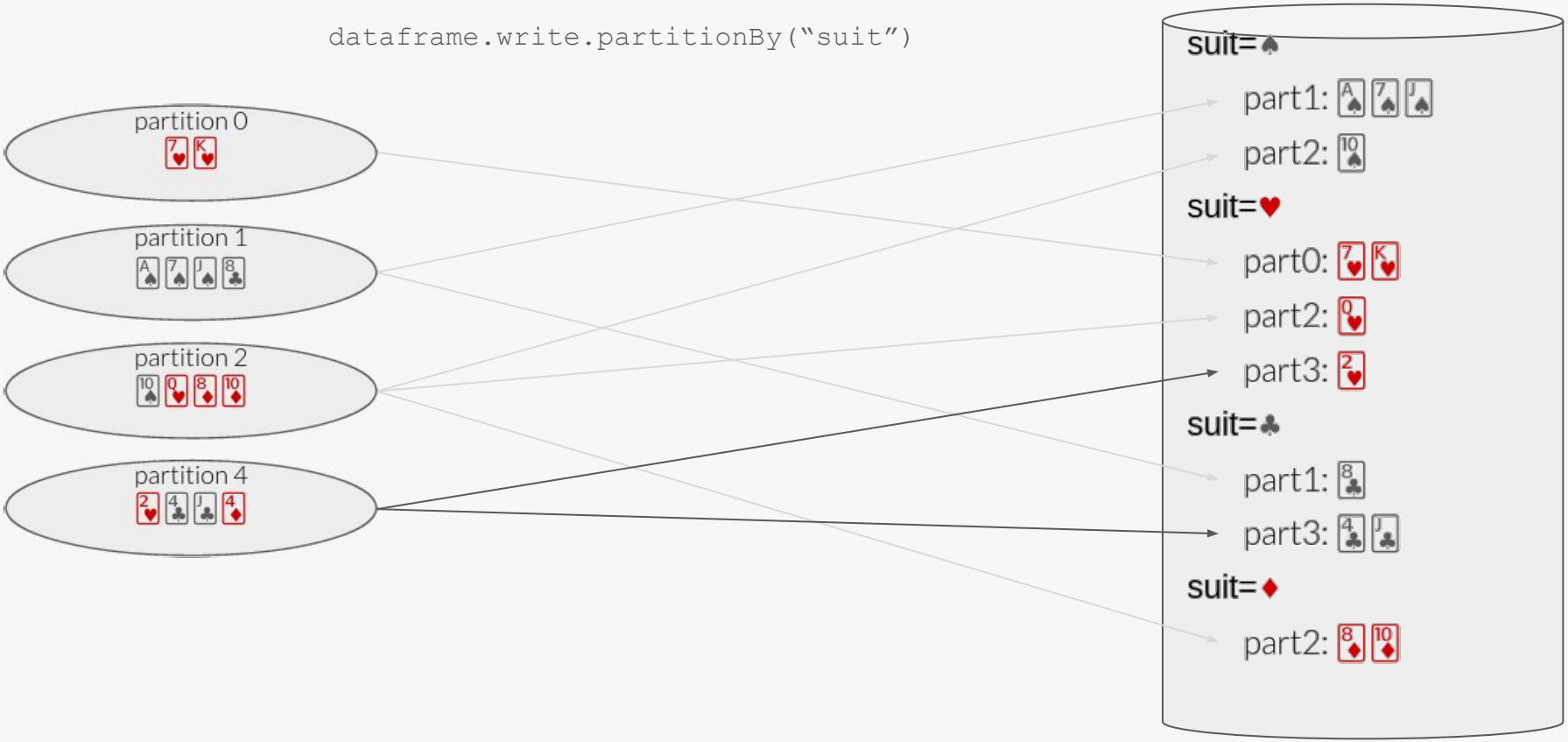
`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```



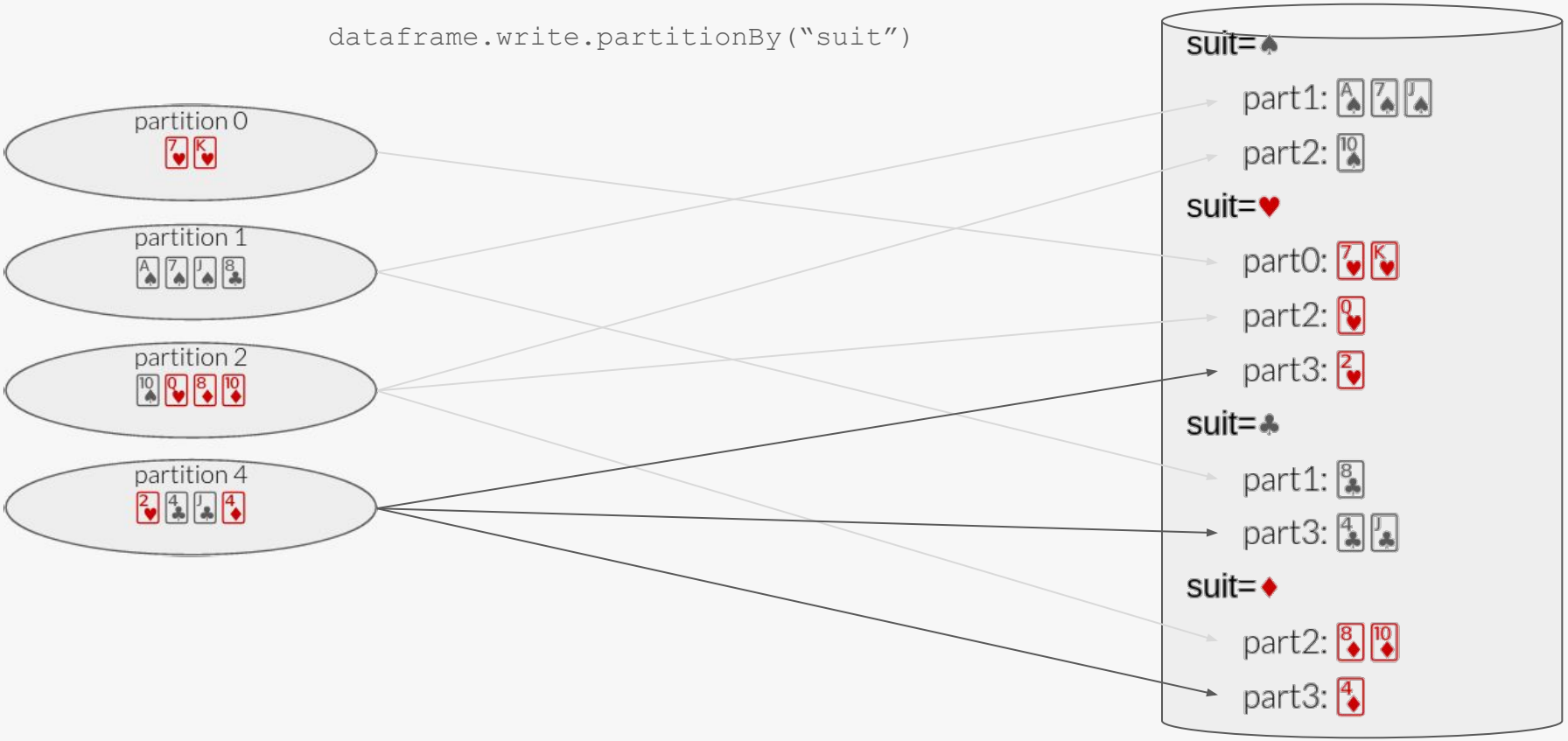
`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```



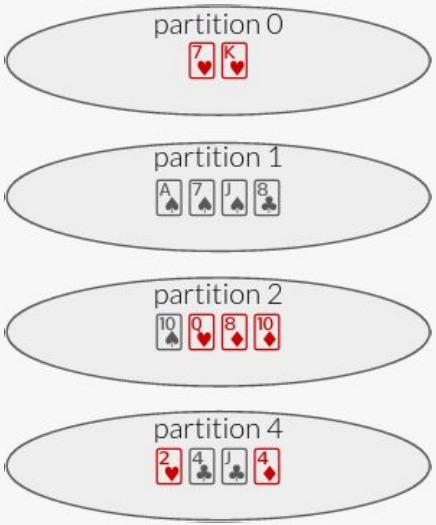
`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

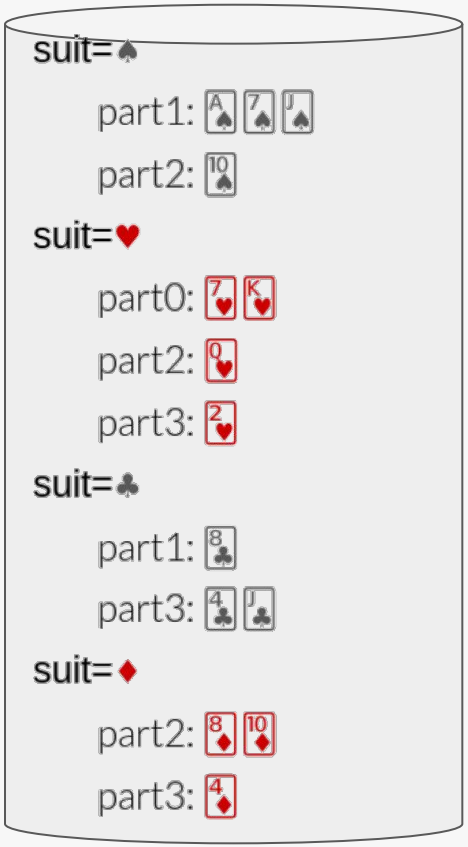
```
dataframe.write.partitionBy("suit")
```



`partitionBy` creates a hierarchy of folders on the filesystem, which could at a later stage be used to load data selectively.

```
dataframe.write.partitionBy("suit")
```





DEMO

key take-away

`Repartition` shuffles data in transit,

`partitionBy` affects data at rest and possibly speeds up subsequent reading queries.

key take-aways of today's session

key take-aways of today's session

1. **Spark's Catalyst will optimize your queries.**
   That doesn't mean you should write bad transformation pipelines, because it can't do everything. For example, it can't automatically discover when to cache or when to drop a shuffle.

key take-aways of today's session

1. **Spark's Catalyst will optimize your queries.**
   That doesn't mean you should write bad transformation pipelines, because it can't do everything. For example, it can't automatically discover when to cache or when to drop a shuffle.

2. **Avro, Parquet and ORC are common data storage formats for big data.**
   Avro for streaming, Parquet and ORC for batch analysis.

key take-aways of today's session

1. **Spark's Catalyst will optimize your queries.**
   That doesn't mean you should write bad transformation pipelines, because it can't do everything. For example, it can't automatically discover when to cache or when to drop a shuffle.

2. **Avro, Parquet and ORC are common data storage formats for big data.**
   Avro for streaming, Parquet and ORC for batch analysis.

3. **`repartition` shuffles data in transit, `partitionBy` affects data at rest**
   Use the former to optimize cluster load and the latter to optimize for data access patterns.

Cleaning data

Part of many an ETL, and the absolute time waste of many a data scientist.

Agenda for the 4th session

- Use case 1: “cleaning” a CSV file of flights
  - choosing data types wisely
  - choosing placeholders wisely
  - the zones of a data lake
  - cleaning a csv file

- The data catalog

  - Notions
  - Implementation

Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

unstructured:

semi-structured:

structured:

Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

unstructured:
🎵🖼️

semi-structured:

structured:

Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

unstructured:
🎵🖼️

semi-structured:
JSON/XML/YAML

structured:

Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

unstructured:
🎵🖼️

semi-structured:
JSON/XML/YAML

structured:
CSV/Parquet/...

Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

CLEAN:
reusable, validated

unstructured:
🎵🖼️

semi-structured:
JSON/XML/YAML

structured:
CSV/Parquet/...

Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

CLEAN:
reusable, validated

unstructured:
🎵🖼️

semi-structured:
JSON/XML/YAML

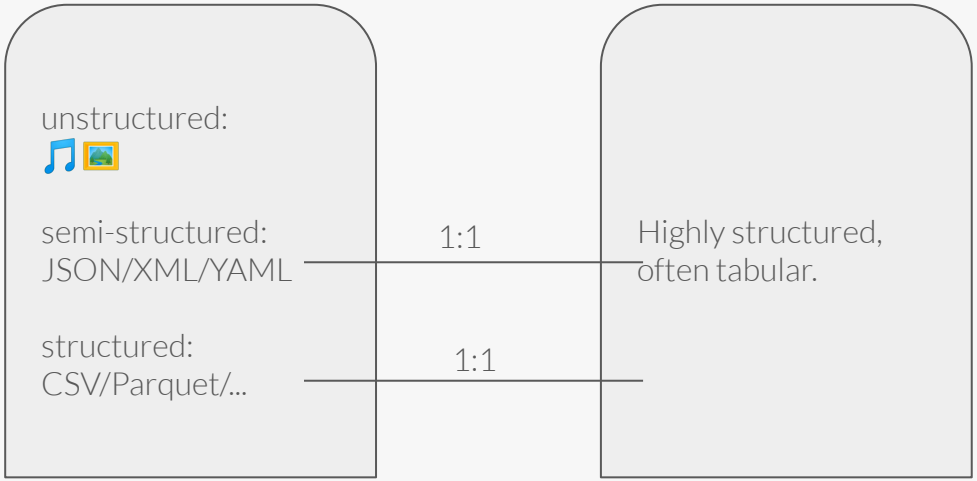
structured:
CSV/Parquet/...

Highly structured,
often tabular.

Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

CLEAN:
reusable, validated

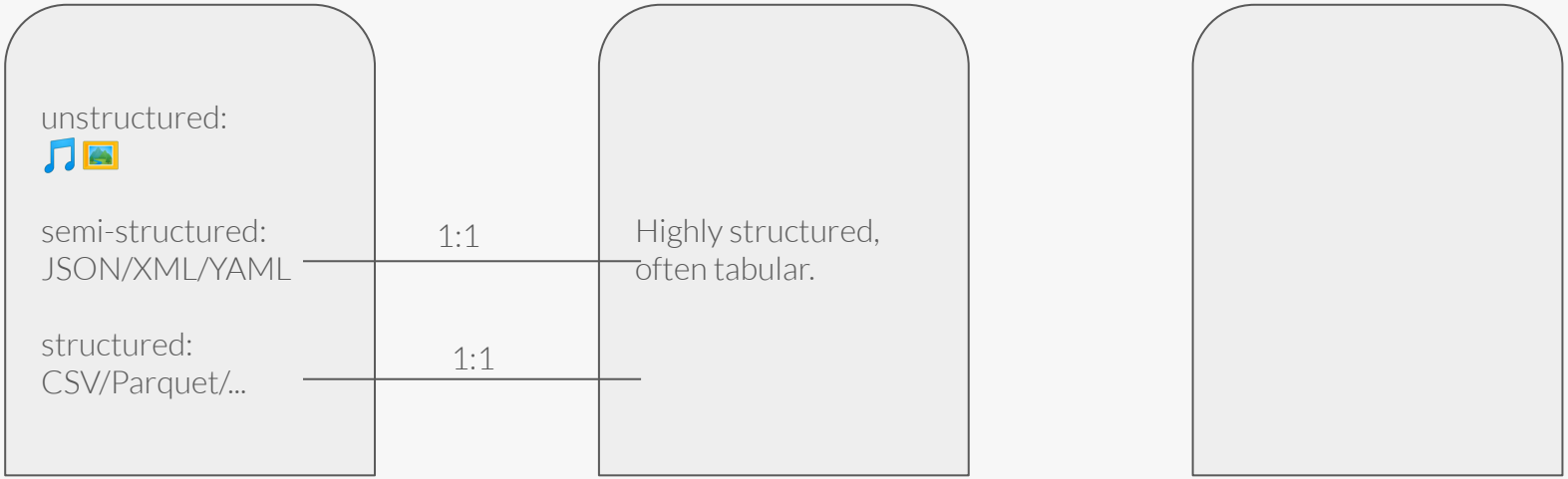


Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

CLEAN:
reusable, validated

BUSINESS:
specific use cases

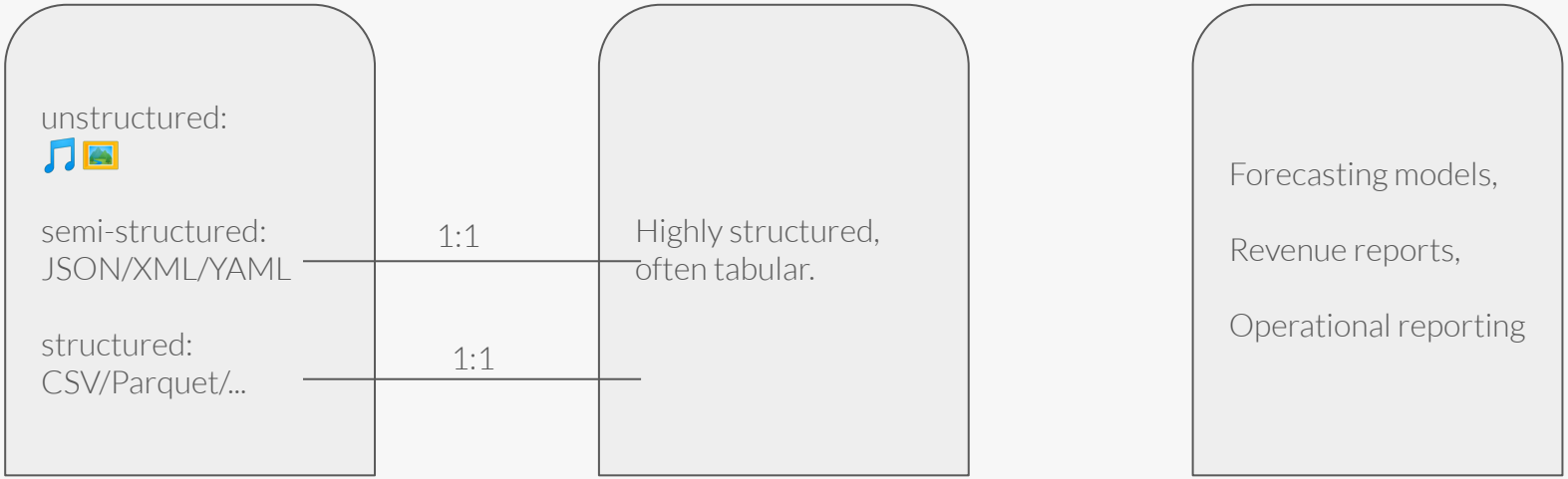


Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

CLEAN:
reusable, validated

BUSINESS:
specific use cases

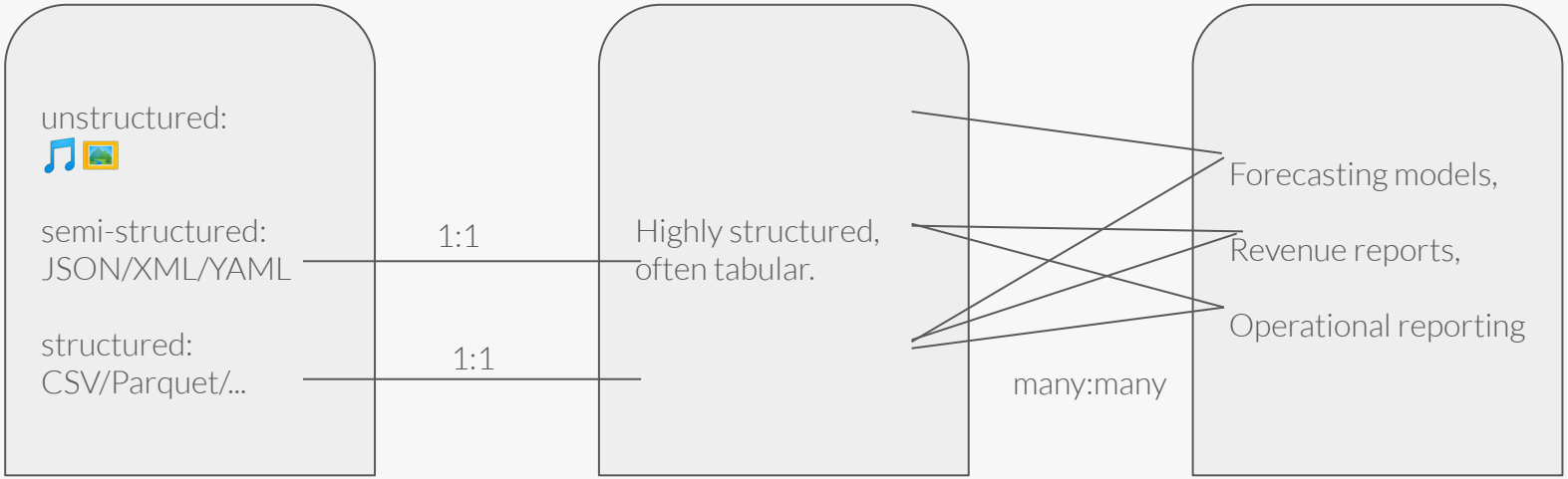


Many data pipelines use 3 zones to store the data: raw, clean and business.

RAW:
archive, buffer

CLEAN:
reusable, validated

BUSINESS:
specific use cases



To move data from raw to clean, one applies a series of common transformations.

Common transformations:

To move data from raw to clean, one applies a series of common transformations.

Common transformations:

- Assign better data types

To move data from raw to clean, one applies a series of common transformations.

Common transformations:

- Assign better data types
- Fix placeholders

To move data from raw to clean, one applies a series of common transformations.

Common transformations:

- Assign better data types
- Fix placeholders
- Rename columns

To move data from raw to clean, one applies a series of common transformations.

Common transformations:

- Assign better data types
- Fix placeholders
- Rename columns

One typically does not drop rows. If you feel the need to do so, consider storing “bad records” elsewhere.

To move data from raw to clean, one applies a series of common transformations.

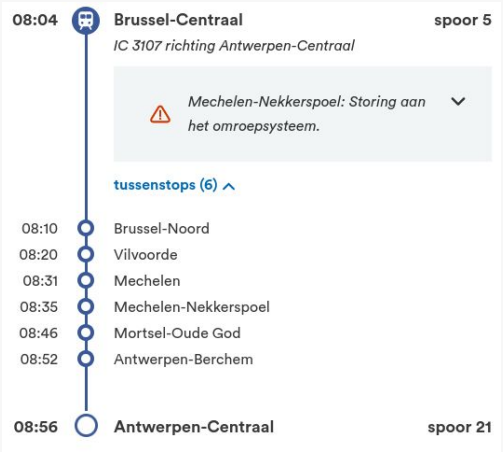
Common transformations:

- Assign better data types
- Fix placeholders
- Rename columns

One typically does not drop rows. If you feel the need to do so, consider storing “bad records” elsewhere.

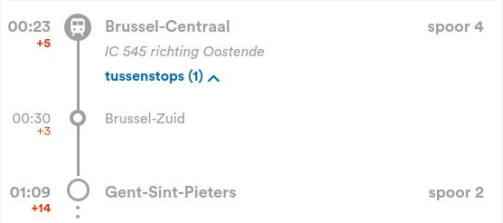
Dropping columns is arguably fine in case of redundancy, which is often more confusing.

Events are facts and come with timestamps. Schedules implicitly have an associated timezone.



Associated timezone: Europe/Brussels.

Merging time information is error-prone: cross-midnight departures, cross-timezone transfers, ...



Departure times become facts. Think about opening hours. Flight plans.

Save yourself from ambiguity: **store events as timestamps (date + time of day + timezone) as one object**. Consider doing this too with schedules if you tend to store their related events (shop opening hours vs departure times).

Avoid relative times, like “age” in data that is meant to be reusable. Those values are relevant now, not in the future anymore.

Bedrijfstakken/branches SBI 2008	Onderwerp: Bedrijfsbestelauto's		
Bedrijfsgrootte / leeftijd bestelauto	2015	2016	201
	aantal		
A-U Alle economische activiteiten	Totaal	814 774	849 655	885 4
	Bedrijfsgrootte: 0 tot 10 werkzame pers.	527 405	558 329	576 2
	Bedrijfsgrootte: 10 tot 100 werkz. pers.	170 797	177 597	187 8
	Bedrijfsgrootte: >= 100 werkzame pers.	116 572	113 729	121 3
	Leeftijd bestelauto: 0 tot 4 jaar	215 888	232 095	258 0
	Leeftijd bestelauto: 4 tot 10 jaar	317 800	323 760	309 7
	Leeftijd bestelauto: 10 jaar en ouder	281 086	293 800	317 7
F Bouwnijverheid	Totaal	236 204	245 736	255 0
	Bedrijfsgrootte: 0 tot 10 werkzame pers.	154 398	164 321	170 6
	Bedrijfsgrootte: 10 tot 100 werkz. pers.	54 688	55 433	58 8
	Bedrijfsgrootte: >= 100 werkzame pers.	27 118	25 982	25 5
	Leeftijd bestelauto: 0 tot 4 jaar	57 287	64 762	74 1
	Leeftijd bestelauto: 4 tot 10 jaar	104 938	103 396	96 5
	Leeftijd bestelauto: 10 jaar en ouder	73 979	77 578	84 3
G Handel	Totaal	194 279	204 623	212 8
	Bedrijfsgrootte: 0 tot 10 werkzame pers.	140 831	147 230	149 6
	Bedrijfsgrootte: 10 tot 100 werkz. pers.	38 619	40 572	44 1
	Bedrijfsgrootte: >= 100 werkzame pers.	14 829	16 821	19 0
	Leeftijd bestelauto: 0 tot 4 jaar	50 102	54 530	59 6
	Leeftijd bestelauto: 4 tot 10 jaar	69 265	71 336	68 5
	Leeftijd bestelauto: 10 jaar en ouder	74 912	78 757	84 5

Bron: CBS

Fine for specific business use cases, like reports. Less so for clean data.

Reference points are often found in the filename or the e-mail in which data was sent.

Spark will allow you to reuse the filename as a column: see `input_file_name`.

Avoid filling in the blanks. If you have to, do not expose this made-up data to anyone else.

```
name;title;is_male;entered_office_on;left_office_on;party;consecutive_terms
Theresa May;;false;2016-07-13;2019-07-24;Conservative;2
Boris Johnson;;true;2019-07-;;nservative;2
```

You typically see placeholders in OLTP and OLAP databases that had to accommodate to changing requirements fast.

Numerical data types come in all shapes and sizes. Consider for a moment their tradeoffs.

Floats vs Doubles:

- For GPS coordinates: there's no benefit in accuracy for most civilian GPS devices to store stuff as doubles, as they're only accurate to about 4m.
- Neither should be used in equality comparisons

Decimals & BigDecimals

- Consider whether using **centimes** is not more useful than using floating point numbers (which come with their own set of problems, especially when used in aggregations of largely varying sizes). Example: €3.14 stored as 314 cents.

Keep the ranges of integral types in mind:

	Byte (1 byte)	Short (2 bytes)	Int (4 bytes)	Long (8 bytes)
min value	-128	-32768	-2 147 483 648	-9223372036854775808
max value	127	32767	2 147 483 647	9223372036854775807

Strings are typically the biggest data types to sling around. If possible, defer using until the presentation layer.

Strings often come back in lookup tables (not facts but *dimension* tables). Join them just before drawing a report, if possible.

Do not store dates or timestamps as strings

- the **methods available** to dates keep track of various features you wouldn't typically think of when manipulating strings (think leap days here and even leap seconds here, time zones if you're looking at timestamps).

- dates **take up less space than strings**:
  storing 2M yyyy-mm-dd formatted strings: 26.8MB in memory,
  storing the same as dates: <2MB.

Exercise: cleaning a CSV file

```
1 spark = (
2     SparkSession.builder
3     .config("spark.jars.packages", "org.apache.hadoop:hadoop-aws:3.1.2")
4     .config("fs.s3a.aws.credentials.provider", "org.apache.hadoop.fs.s3a.AnonymousAWSCredentialsProvider")
5     .getOrCreate()
6 )
7
8 frame = spark.read.csv("s3a://somebucket/some_key.tgz")
```

🐌 - downloads every time

💡 - place jars in specific directory on host

ASSIGNMENT:
Work out the cleaning steps for the Flights dataset. See clean_flights_starter for details.

The data catalog

Because shopping for data is only fun when you know where you can get it.

The essence of a data catalog is to have *one* single “place” where you can look up the essential details about data you want to do something with.

The essence of a data catalog is to have *one* single “place” where you can look up the essential details about data you want to do something with.

To load and store data, one needs:

- storage location (e.g “s3://bucket/obj”, “wasbs://container/blob”)
- format
  - Parquet
  - SQL → JDBC/ODBC
  - JSON
  - ...
- format-specific options

Some commercial catalogs offer some useful extras, like meaningful aggregation statistics:

- size/number of records
- distribution of the data (min, max, nullcount, median)

Exercise: implement a simple data catalog

ASSIGNMENT:

Implement a data catalog using a **dictionary**: `catalog = {"clean_flights": …}`

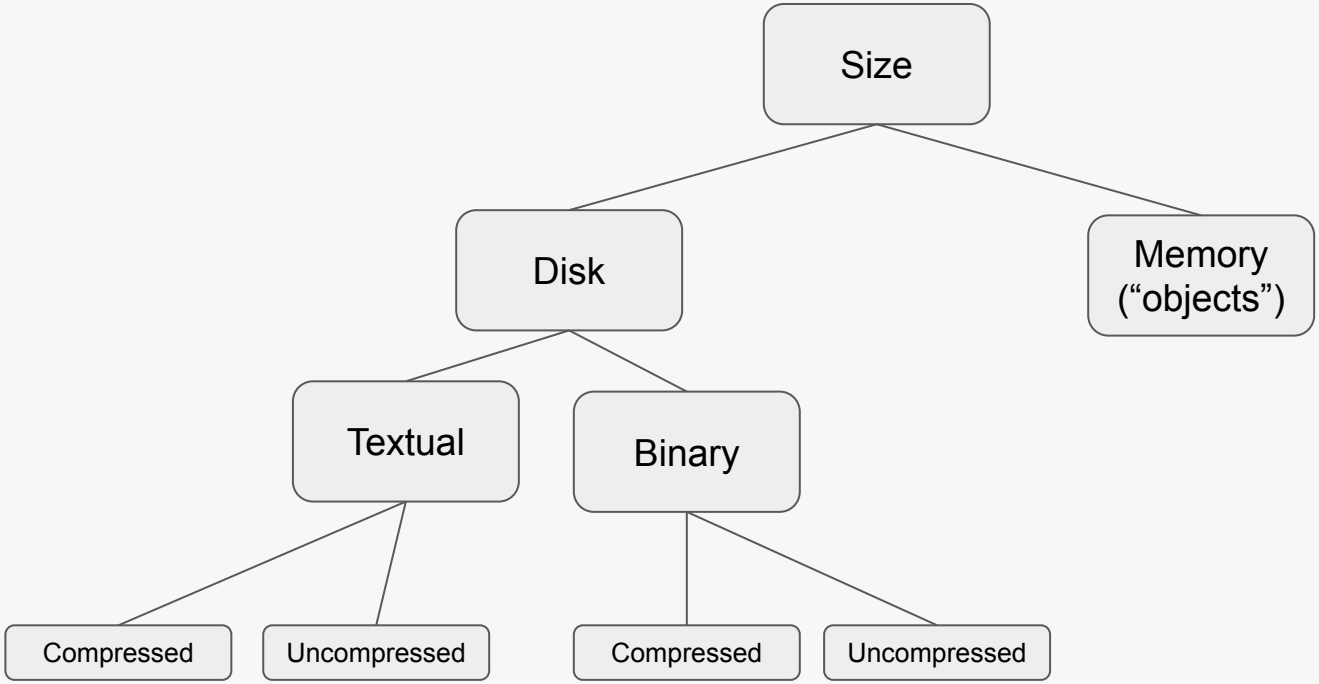
Reference at least the following files:

- The flights data: its raw and cleaned forms
- The lookup table that maps IANA airport codes to their full names
- The lookup table that maps IATA carrier codes to airline carriers over a specific time range.

Build up the catalog gradually:

1. Link a dataset name to a location (bonus: make it independent of the location of your project on the laptop).
2. Add sufficient information about the format so that it can be used by an automated process.
3. Add format-specific options. Again, see that it can be used in an automated way.
4. Create a reader function that, given a key of the catalog, will return the corresponding dataframe.

The size of data on a disk is not the same as that in memory.



When writing, avoid creating many small files, as this makes subsequent reading slower. Aim for about 250MB/file.

Reasons for slowdown:

- getting directory listings
- overhead on file metadata

Techniques to reduce file sizes:

- repartition (similar items are colocated)
- compression ("uncompressed", "snappy", "gzip", ...)
- orderBy (run-length encoding at work)

Keep in mind that the amount of data records doesn't change. Merely the amount of meta-data or its encoding.

With the exception of compression, don't expect your data volume to magically decrease by a factor of 10.

ASSIGNMENT:
Revisit the module where you clean the flights dataset. Use various techniques to alter the size of the resulting files on disk: format, compression, data colocation.

Reusing lazily computed data

Knowing when to cache.

Consider caching a dataset after a series of heavy transformations (e.g. a HashJoin), when you will reuse the same data

```
df2 = transform1(df1)

df3 = transform2(df2).cache()

df4 = transform3(df3)
df4.write(…)

df5 = transform4(df3)
df6 = transform5(df5)
df6.write(...)
```

df1

df2

df3

df4

df5

df6

In **production**: remove or disable the actions that were meant only for debugging. And possibly clean up any calls to `cache()` that became superfluous because of that too.

DEMO.

Advanced analytical functions

Doing better than average

Some advanced functionality is provided by windowing functions, which define a range of records (the window) per row *over* which values are computed.

The size of this group is determined by the partitioning elements defining the groups and the row or range specification supplied to the windowing specification:

```
window_spec = (pyspark.sql.Window
.partitionBy("person")
.orderBy("check-in date")
.rowsBetween(-2, pyspark.sql.Window.currentRow))
```

When observing this row in the "Sarah" partition/group, ...

person	check-in date	minutes online
Sarah	2020-01-01	10
Sarah	2020-01-02	20
Sarah	2020-01-03	20
Sarah	2020-01-04	1
Marcus	2020-01-06	10
Marcus	2020-02-09	20
Marcus	2020-02-10	15

Some advanced functionality is provided by windowing functions, which define a range of records (the window) per row *over* which values are computed.

The size of this group is determined by the partitioning elements defining the groups and the row or range specification supplied to the windowing specification:

```
window_spec = (pyspark.sql.Window
.partitionBy("person")
.orderBy("check-in date")
.rowsBetween(-2, pyspark.sql.Window.currentRow))
```

The records in this window are considered

When observing this row in the "Sarah" partition/group, ...

person	check-in date	minutes online
Sarah	2020-01-01	10
Sarah	2020-01-02	20
Sarah	2020-01-03	20
Sarah	2020-01-04	1
Marcus	2020-01-06	10
Marcus	2020-02-09	20
Marcus	2020-02-10	15

A Window Specification can be (re)used for several functions of the aggregation, analytics or ranking kind. The function is computed using only the limited window, per row.

```
window_spec = (pyspark.sql.Window
  .partitionBy("person")
  .orderBy("check-in date")
  .rowsBetween(-2, pyspark.sql.Window.currentRow))

df.withColumn(
  "running mean",
  mean("minutes online").over(window_spec))
```

standard aggregation function

There's a handful of functions that can be used with WindowSpecifications. Look for the **Window functions heading** and **Aggregate functions headings** on the spark.sql.functions doc.

person	check-in date	minutes online	running mean
Sarah	2020-01-01	10	
Sarah	2020-01-02	20	
Sarah	2020-01-03	30	
Sarah	2020-01-04	1	
Marcus	2020-01-06	10	
Marcus	2020-02-09	20	
Marcus	2020-02-10	15	

A Window Specification can be (re)used for several functions of the aggregation, analytics or ranking kind. The function is computed using only the limited window, per row.

```
window_spec = (pyspark.sql.Window
  .partitionBy("person")
  .orderBy("check-in date")
  .rowsBetween(-2, pyspark.sql.Window.currentRow))

df.withColumn(
  "running mean",
  mean("minutes online").over(window_spec))
```

standard aggregation function

There's a handful of functions that can be used with WindowSpecifications. Look for the **Window functions heading** and **Aggregate functions headings** on the spark.sql.functions doc.

person	check-in date	minutes online	running mean
Sarah	2020-01-01	10	10
Sarah	2020-01-02	20	
Sarah	2020-01-03	30	
Sarah	2020-01-04	1	
Marcus	2020-01-06	10	
Marcus	2020-02-09	20	
Marcus	2020-02-10	15	

A Window Specification can be (re)used for several functions of the aggregation, analytics or ranking kind. The function is computed using only the limited window, per row.

```
window_spec = (pyspark.sql.Window
  .partitionBy("person")
  .orderBy("check-in date")
  .rowsBetween(-2, pyspark.sql.Window.currentRow))

df.withColumn(
  "running mean",
  mean("minutes online").over(window_spec))
```

standard aggregation function

There's a handful of functions that can be used with WindowSpecifications. Look for the **Window functions heading** and **Aggregate functions headings** on the spark.sql.functions doc.

person	check-in date	minutes online	running mean
Sarah	2020-01-01	10	10
Sarah	2020-01-02	20	(10+20)/2
Sarah	2020-01-03	30	
Sarah	2020-01-04	1	
Marcus	2020-01-06	10	
Marcus	2020-02-09	20	
Marcus	2020-02-10	15	

A Window Specification can be (re)used for several functions of the aggregation, analytics or ranking kind. The function is computed using only the limited window, per row.

```
window_spec = (pyspark.sql.Window
  .partitionBy("person")
  .orderBy("check-in date")
  .rowsBetween(-2, pyspark.sql.Window.currentRow))

df.withColumn(
  "running mean",
  mean("minutes online").over(window_spec))
```

standard aggregation function

There's a handful of functions that can be used with WindowSpecifications. Look for the **Window functions heading** and **Aggregate functions headings** on the spark.sql.functions doc.

person	check-in date	minutes online	running mean
Sarah	2020-01-01	10	10
Sarah	2020-01-02	20	(10+20)/2
Sarah	2020-01-03	30	(10+20+30)/3
Sarah	2020-01-04	1	
Marcus	2020-01-06	10	
Marcus	2020-02-09	20	
Marcus	2020-02-10	15	

A Window Specification can be (re)used for several functions of the aggregation, analytics or ranking kind. The function is computed using only the limited window, per row.

```
window_spec = (pyspark.sql.Window
  .partitionBy("person")
  .orderBy("check-in date")
  .rowsBetween(-2, pyspark.sql.Window.currentRow))

df.withColumn(
  "running mean",
  mean("minutes online").over(window_spec))
```

standard aggregation function

There's a handful of functions that can be used with WindowSpecifications. Look for the **Window functions heading** and **Aggregate functions headings** on the spark.sql.functions doc.

person	check-in date	minutes online	running mean
Sarah	2020-01-01	10	10
Sarah	2020-01-02	20	(10+20)/2
Sarah	2020-01-03	30	(10+20+30)/3
Sarah	2020-01-04	1	(20+30+1)/3
Marcus	2020-01-06	10	
Marcus	2020-02-09	20	
Marcus	2020-02-10	15	

A Window Specification can be (re)used for several functions of the aggregation, analytics or ranking kind. The function is computed using only the limited window, per row.

```
window_spec = (pyspark.sql.Window
  .partitionBy("person")
  .orderBy("check-in date")
  .rowsBetween(-2, pyspark.sql.Window.currentRow))

df.withColumn(
  "running mean",
  mean("minutes online").over(window_spec))
```

standard aggregation function

There's a handful of functions that can be used with WindowSpecifications. Look for the **Window functions heading** and **Aggregate functions headings** on the spark.sql.functions doc.

person	check-in date	minutes online	running mean
Sarah	2020-01-01	10	10
Sarah	2020-01-02	20	(10+20)/2
Sarah	2020-01-03	30	(10+20+30)/3
Sarah	2020-01-04	1	(20+30+1)/3
Marcus	2020-01-06	10	10
Marcus	2020-02-09	20	
Marcus	2020-02-10	15	

A Window Specification can be (re)used for several functions of the aggregation, analytics or ranking kind. The function is computed using only the limited window, per row.

```
window_spec = (pyspark.sql.Window
  .partitionBy("person")
  .orderBy("check-in date")
  .rowsBetween(-2, pyspark.sql.Window.currentRow))

df.withColumn(
  "running mean",
  mean("minutes online").over(window_spec))
```

standard aggregation function

There's a handful of functions that can be used with WindowSpecifications. Look for the **Window functions heading** and **Aggregate functions headings** on the spark.sql.functions doc.

person	check-in date	minutes online	running mean
Sarah	2020-01-01	10	10
Sarah	2020-01-02	20	(10+20)/2
Sarah	2020-01-03	30	(10+20+30)/3
Sarah	2020-01-04	1	(20+30+1)/3
Marcus	2020-01-06	10	10
Marcus	2020-02-09	20	15
Marcus	2020-02-10	15	

A Window Specification can be (re)used for several functions of the aggregation, analytics or ranking kind. The function is computed using only the limited window, per row.

```
window_spec = (pyspark.sql.Window
  .partitionBy("person")
  .orderBy("check-in date")
  .rowsBetween(-2, pyspark.sql.Window.currentRow))

df.withColumn(
  "running mean",
  mean("minutes online").over(window_spec))
```

standard aggregation function

There's a handful of functions that can be used with WindowSpecifications. Look for the **Window functions heading** and **Aggregate functions headings** on the spark.sql.functions doc.

person	check-in date	minutes online	running mean
Sarah	2020-01-01	10	10
Sarah	2020-01-02	20	(10+20)/2
Sarah	2020-01-03	30	(10+20+30)/3
Sarah	2020-01-04	1	(20+30+1)/3
Marcus	2020-01-06	10	10
Marcus	2020-02-09	20	15
Marcus	2020-02-10	15	15

`monotonically_increasing_id` is just that: a numerical identifier which (over the partitions and their records) is strictly increasing, but can have non-consecutive jumps

- It finds its uses in some highly specialized self-joins

- is more efficient than using `row_number().over(Window.partitionBy(...))`
  since there is no shuffle involved:
  each executor can process the partitions without needing to get data from different processes.

- Does not suffer from hash-collisions, unlike "`hash`".
  Note that good hashing functions will have very few collisions, even on large datasets.

Reviewing code

To improve not only code,
but yourself as well.

Code review is an essential process to improve knowledge and skill within a team. It may also lead to, more efficient and robust programs, and possibly remove issues early on.

With everything you've learned so far, you should be able to read, understand and improve someone else's work.

We will step through an actual use case of a different business: finding out which people are "binge watchers". Note that we do not need to understand the criterion to define when someone is flagged as a binge watcher. But we can still improve this code.

Applying business logic

The stuff that really answers questions

Exercise: implementing a business use case. Pretend you work for American Airlines

- Create a 360°-view of the flights data in which you combine the airline carriers (a dimension table), the airport names (another dimension table) and the flights tables (a facts table).

Sounds simple, right? Participants tend to need well over an hour for this.

- Your manager wants to know how many flights we operated in 2011.
- How many of those flights arrived with less than (or equal to) 10 minutes of delay?
- A data scientist is looking for correlations between the departure delays and the dates. In particular, he/she thinks that on Fridays there are relatively speaking more flights departing with a delay than on any other day of the week. Verify his/her claim.
- Out of the 5 categories of sources for delays, which one appeared most often in 2011? In other words, in which category should we invest more time to improve?

Use the right tool for the right job

All of this has focussed on PySpark. In practice, many problems can be solved elegantly, and sometimes faster with single-node programs. Pandas has ways to read csv files and process them. Parquet files too, they're not just a byproduct of Spark tooling. Standard Python also offers ways to process files (there's a groupby function in the standard library!).

Use the right tool for the job. This requires experience. Or sometimes just a moment of thought before someone blindly starts assuming everything is a nail. *If a dataset fits comfortably in the memory of one single node, using Spark is usually overkill, and will typically prove to be slower than a solution with e.g. Pandas.*

Shameless plug: if you want to further your understanding of PySpark and writing good data engineering pipelines, check out our online course.

"Building data engineering pipelines in Python" is a course on DataCamp that we've built to aid our clients in their journey to help employees get their job done more efficiently, and for managers to get a high level understanding of current data practices (that's in chapter 1, which is for free even).

It is part of a larger set of data engineering related courses at DataCamp, but was the first to implement new technologies offered by the platform, and is unique in that it touches several important but disparate topics in one ~4h course (note: exercises are considered challenging compared to most other courses at DataCamp).

dataminded

Data Minded bvba
Tiensevest 102 bus 0101
3000 Leuven, Belgium

BE.0667.976.246

dataminded.be