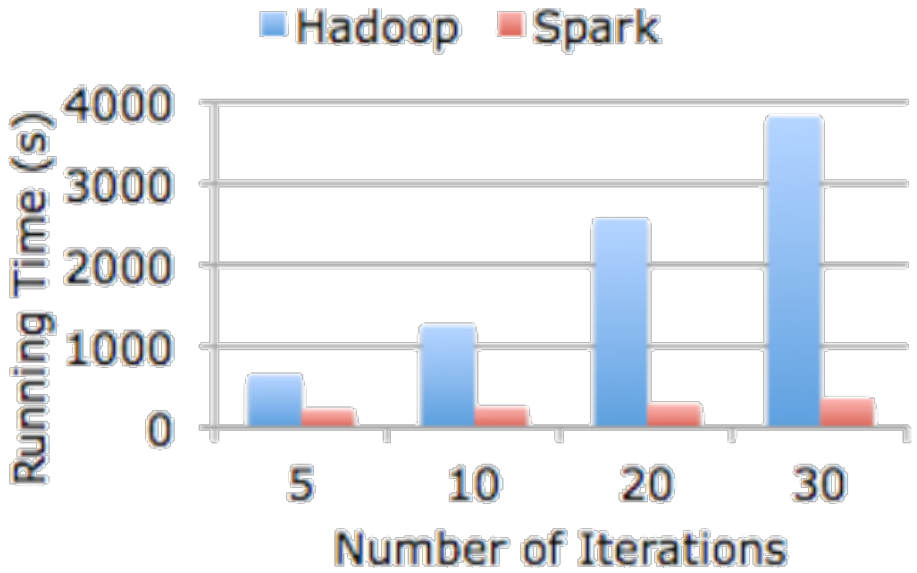
Spark

Beyond Map Reduce: Algebraic Operations

- Cluster computing frameworks let users write parallel code using high-level operators
  - No need to worry about work distribution and fault tolerance.
  - E.g., Hadoop
- Current generation execution engines
  - natively support algebraic operations such as joins, aggregation, etc.
  - allow users to create their own algebraic operators
    - E.g., machine learning as an algebraic operation
  - support trees of algebraic operators that can be executed on multiple nodes in parallel
  - E.g., Spark

Introduction

- What is **Spark**?
  - Unified computation engine for clusters
  - Includes libraries and tools to perform many tasks in parallel
  - At AMPLab (same people who created Hadoop), an API in *Scala* (functional language) to express, briefly and concisely, multiple-step applications
    - Multiple steps carried out **in main memory**, avoiding accesses to disk



- What for?
  - To query our data, and to transform large datasets efficiently using multiple machines
  - Use high-level operators without worrying about work distribution, fault tolerance…

Spark

- Spark is designed to run in a cluster
- A **cluster** should offer:
  - *Persistence*: data is not lost in case of node failures
  - *Robustness* against failures: if a node fails, the task it was running is assigned to another node
  - *Scalability*: if we add new machines, the cluster incorporates them on-the-fly (without rebooting) to enlarge system capabilities

- In systems such as Hadoop, the only way to reuse data between computations was to write it to stable storage
  - Note the overheads!
    Data replication, disk I/O, serialization, …

E.g., between two MapReduce jobs

Resilient Distributed Datasets (RDDs)

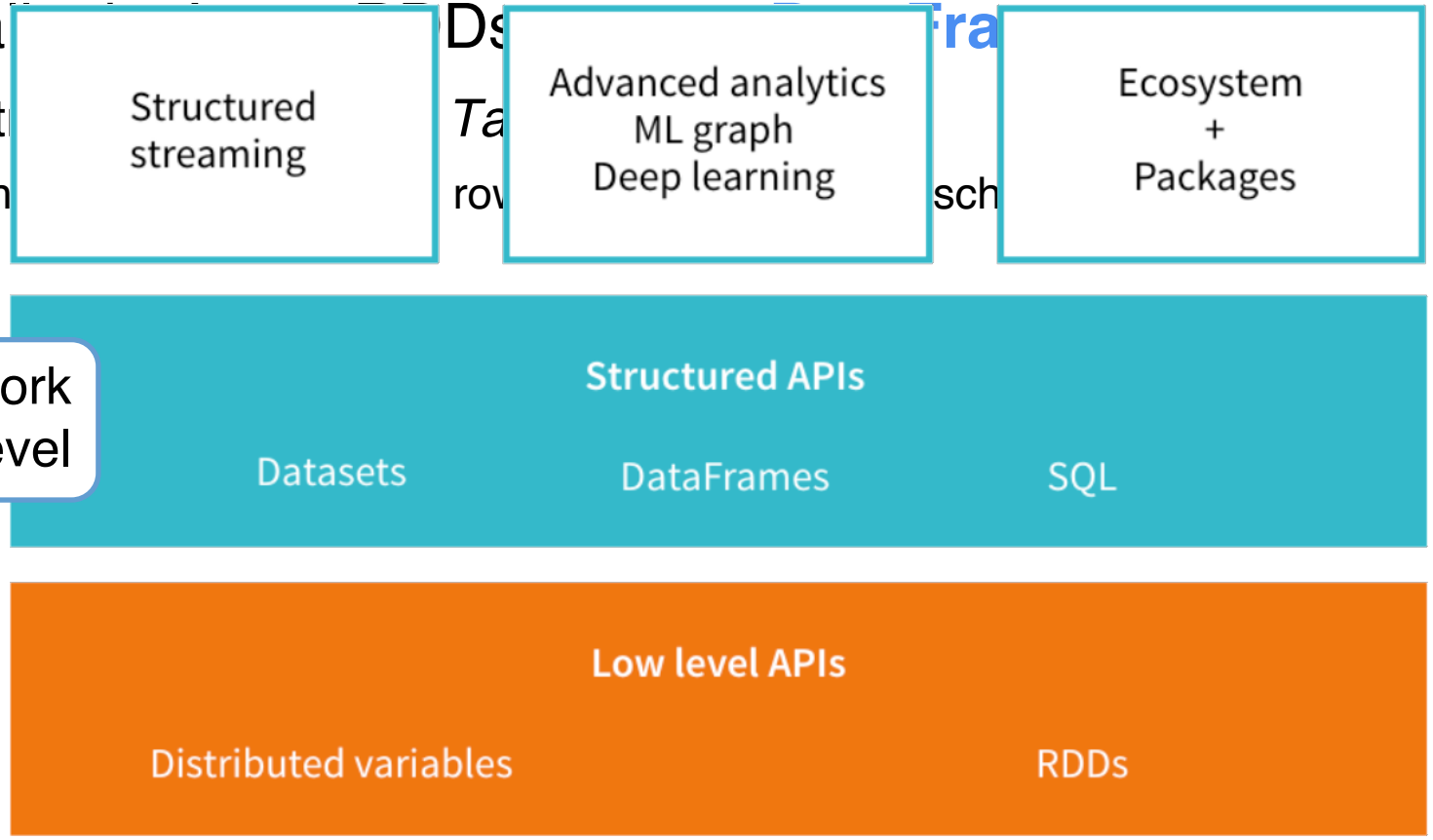
- **Spark** uses **Resilient Distributed Datasets** (RDDs), a data structure that abstracts all the distributed-storage details from the final user
- Enable efficient in-parallel operability
- Enable efficient data reuse
  - Persist intermediate results in memory, optimize data partitioning placement...
- Partitioned collection of records
  - Individual records are just raw Java/Scala/Python objects
- Efficient fault tolerance: log the *transformations* used to build the dataset (*lineage*) rather than the actual data.
  - *Lazy computation*: RDDs are not always materialized
  - Same operation on multiple data points
- Read-only

Main key novelty

There is no concept of “row” in RDDs

Spark

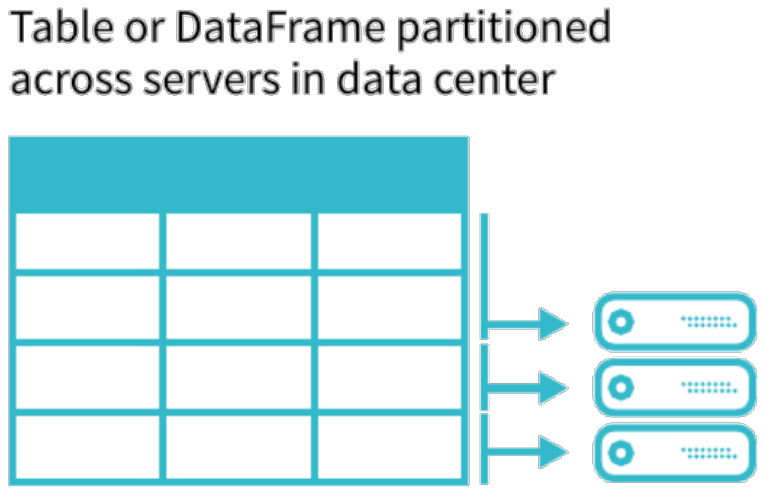
- Components and basic libraries:
- We usua
  - Data st
    - Each



We usually work at this level

DataFrames

- We usually work with **DataFrames**
- Intuitively, *spreadsheet partitioned and distributed*
  - Each **partition** is a set of registers of the spreadsheet stored in a specific node
  - Represent how data is physically distributed across the cluster during execution
  - Spark divides data into partitions automatically and manages them
    - The cluster manager stores partitions in the nodes using, e.g., HDFS
  - Similar to R and Python's DataFrames, but distributed



Using partitions might be beneficial even in your local machine!
Each processor deals with a different partition

DataFrames

- We usually work with **DataFrames**
  - They have a **schema**: definition of columns and their types
  - When running a task, a process deals locally with the partition stored in each node
    - Allow to work in parallel
    - If you use one partition, no-parallelism (even if you have many executors –processes –)
    - If you have many partitions but only one executor (process), no-parallelism

- Users don't deal with any of this (abstracted)
  - You don't manipulate partitions individually
  - You simply specify high-level transformations of data, and Spark schedules the work on the cluster

Transformations

- In Spark, the core data structures are immutable
  i.e., they cannot be changed after they're created
  - To make changes to a DataFrame, you *instruct* Spark how you would like to modify it
  - These *instructions* are called transformations
    - **Algebraic operations**!
      E.g., projection, scan, filter, join, etc.
- Transformations take 1+ DFs, and output a DF
  - Operators are stacked into an *operator tree*
  - Algebraic operations, executed in parallel on many nodes
    - With data partitioned across the machines
  - Algebraic operations, executed lazily (not immediately)
    - A tree is only executed on specific functions (actions)
      E.g., `saveAsTextFile()` or `collect()`

Transformations

- We usually perform different operations
  - E.g., ordering the elements of the *Dataframe*, filtering, grouping, etc.

```
myRange = spark.range(1000).toDF("number")
divisionBy2 = myRange.where("number % 2 = 0")
```

  - A more complex example:

```
# Create a dataframe and read data
flightData = spark \
       .read \
       .option("inferSchema", "true") \
       .option("header", "true") \
       .csv("flight-data/2015-summary.csv")
# transform dataframe by ordering on count
flightDataOrdered = flightData.sort("count")
```

Spark's data structures are immutable. We create a new DF after each transformation!

Actions

- Transformations allow us to build up our logical transformation plan
  - Operator tree
- To **trigger** the computation, we run an **action**.
  - Spark computes the result from a series of transformations

```
myRange = spark.range(1000).toDF("number")
divisionBy2 = myRange.where("number % 2 = 0")
divisionBy2.count()
```

- There exist 3 types of actions:
  - Visualize data/results
  - Write data to disk
  - Transfer data to objects in the base programming language (*Java*, *Python*, …)

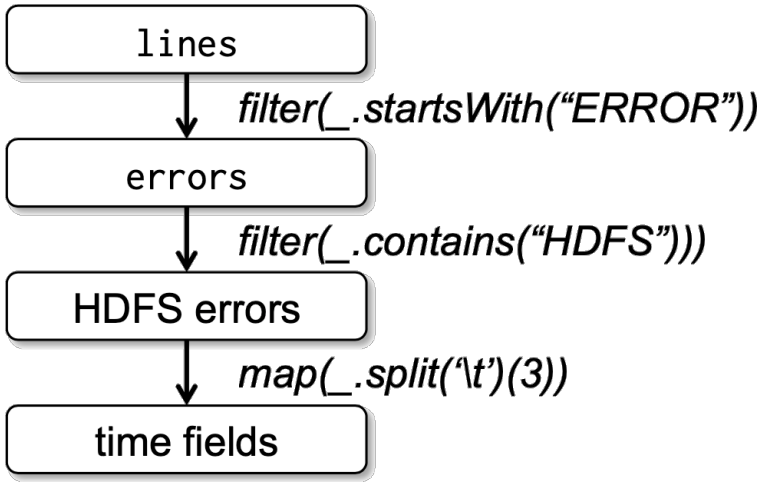
Running jobs in Spark

Lineage graph: thus, our model achieves fault tolerance

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()

errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

Tells Spark to store the partitions of `errors` in memory. `lines`, is *not!*



Running jobs in Spark

- `explain()` function at any DataFrame allows us to see the DataFrame's lineage:

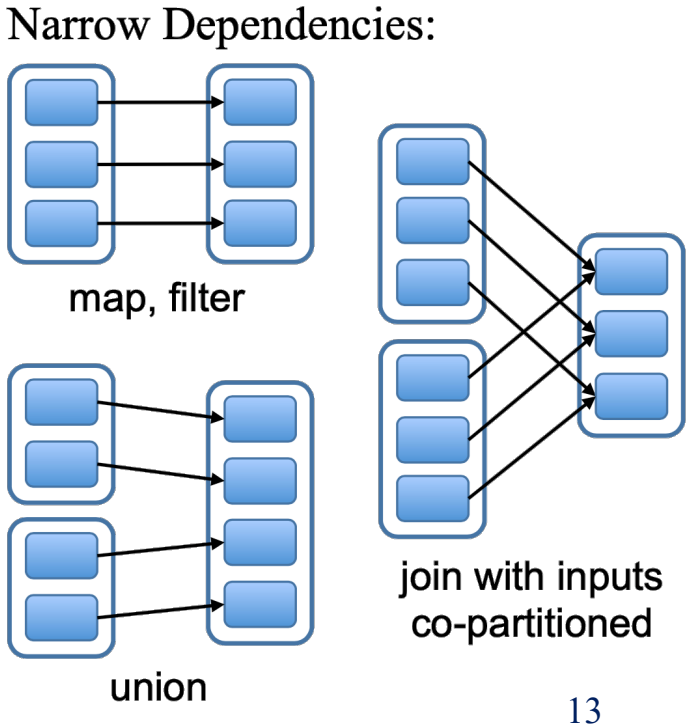
```
flightData2015 = spark\
.read\
.option("inferSchema", "true")\
.option("header", "true")\
.csv("../2015-summary.csv")

flightData2015.sort("count").explain()
```

```
== Physical Plan ==
*(1) Sort [count#39 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#39 ASC NULLS FIRST, 200), true, [id=#98]
   +- FileScan csv [DEST_COUNTRY_NAME#37,ORIGIN_COUNTRY_NAME#38,count#39] Batched: false, Dat
aFilters: [], Format: CSV, Location: InMemoryFileIndex[file:/home/jovyan/data/summary_flight
s/2015-summary.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<DEST_COUNTRY
_NAME:string,ORIGIN_COUNTRY_NAME:string,count:int>
```

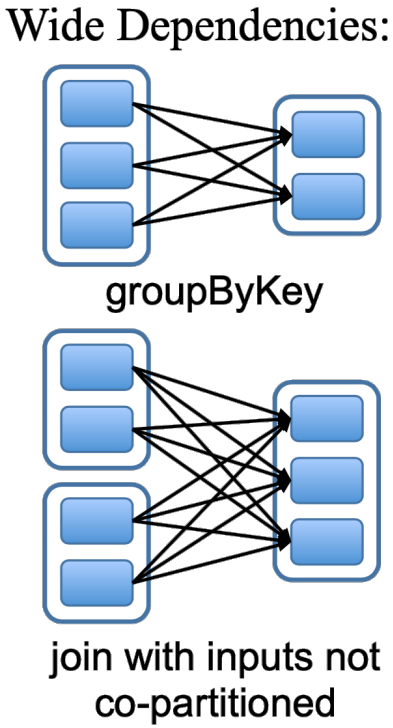
Spark: how it works

- *Lazy evaluation* allows for **pipelining**
  - Spark waits until the very last moment to run the transformation plan
  - By waiting until the last minute, consecutive operations might be *pipelined* if compatible.
- Transformations can involve two types of dependencies:
  - **Narrow dependency**: each partition produces another partition
    - Allow for pipelined execution on one node
    - Recovery after node failure, simple
    - E.g., sum two columns, add a new column, filtering out, remove a column, …
  - **Wide dependency**



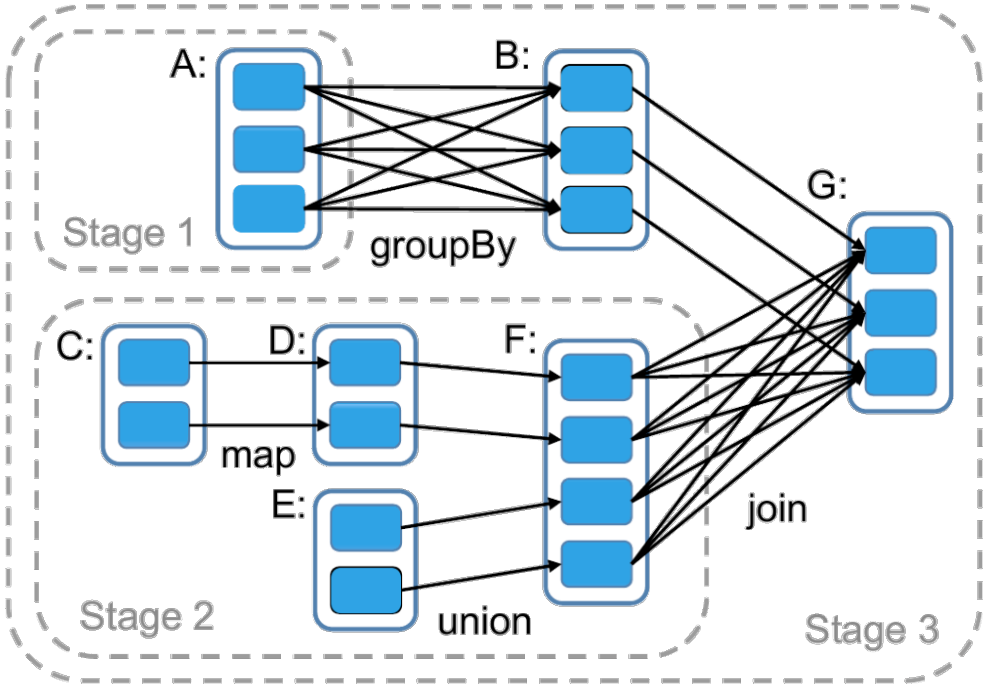
Spark: how it works

- *Lazy evaluation* allows for **pipelining**
  - Spark waits until the very last moment to run the transformation plan
  - By waiting until the last minute, consecutive operations might be *pipelined* if compatible.
- Transformations can involve two types of dependencies:
  - **Narrow dependency**
  - **Wide dependency**: each partition contributes to produce several new partitions
    - *Shuffle*
    - Data needs to be stored to disk
    - E.g., ordering, grouping, …



Spark: how it works

- *Lazy evaluation* allows for **pipelining**
- Scheduler separates the computation in stages (*pipelined work*)
  - For wide dependencies, intermediate results are *materialized*



- Tasks are assigned to machines based on data locality
  - If a task needs to process a partition that is available in memory in a node, Spark sends it there
- If a node fails, Spark can reconstruct the lost partitions from the original data

Spark: how it works

- *Lazy evaluation* allows for **query optimization**
  - Spark waits until the very last moment to run the transformation plan
  - By waiting until the last minute, Spark can compile this plan into an *optimized* physical plan
    - Spark can optimize the entire data flow from end to end
      E.g., predicate pushdown: If we write a large job but, at the end, a filter fetches only one row, the most efficient execution might be to access that single record directly.

Spark: how it works

```
# Create a dataframe and read data
flightData = spark \
       .read \
       .option("inferSchema", "true") \
       .option("header", "true") \
       .csv("flight-data/2015-summary.csv")
# transform dataframe by ordering on count
flightDataOrdered = flightData.sort("count")
# show first 5 elements
flightDataOrdered.show(5)
```

Action

We need to read the whole file, but not to order it completely!

```
# Create a dataframe and read data
flightData = spark \
       .read \
       .option("inferSchema", "true") \
       .option("header", "true") \
       .csv("flight-data/2015-summary.csv")
# transform dataframe by filtering on count
flightDataOrdered = flightData.where("count % 10 = 0")
# show first 5 elements
flightDataOrdered.show(5)
```

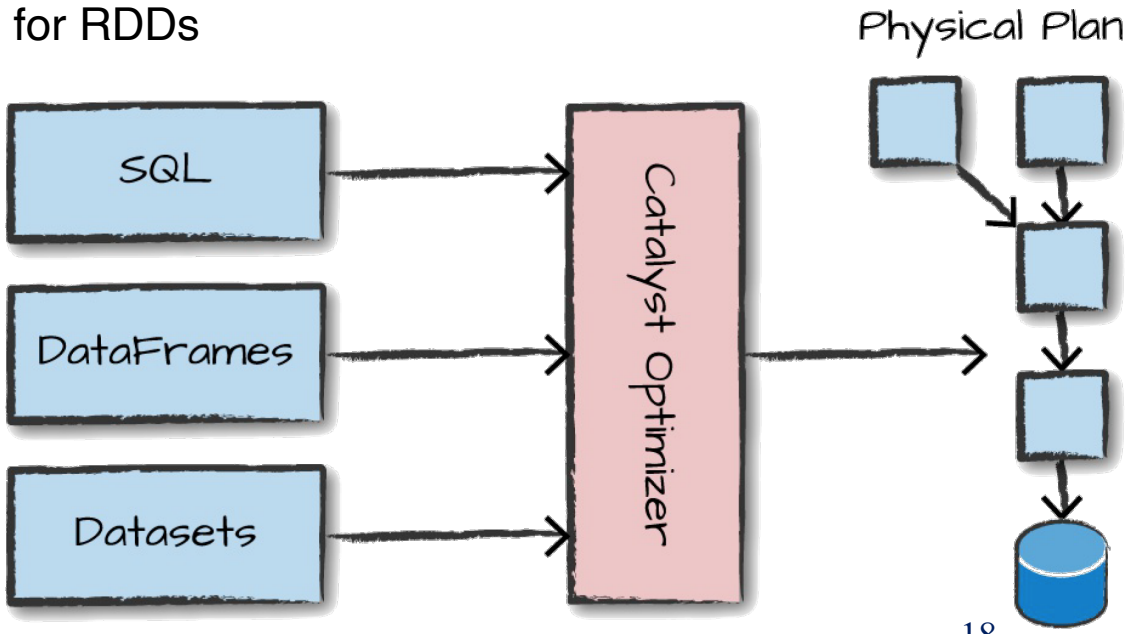
Action

We do not need to read the whole file:

1. Read and filter out not-matching elements
2. Stop when 5 elements are collected

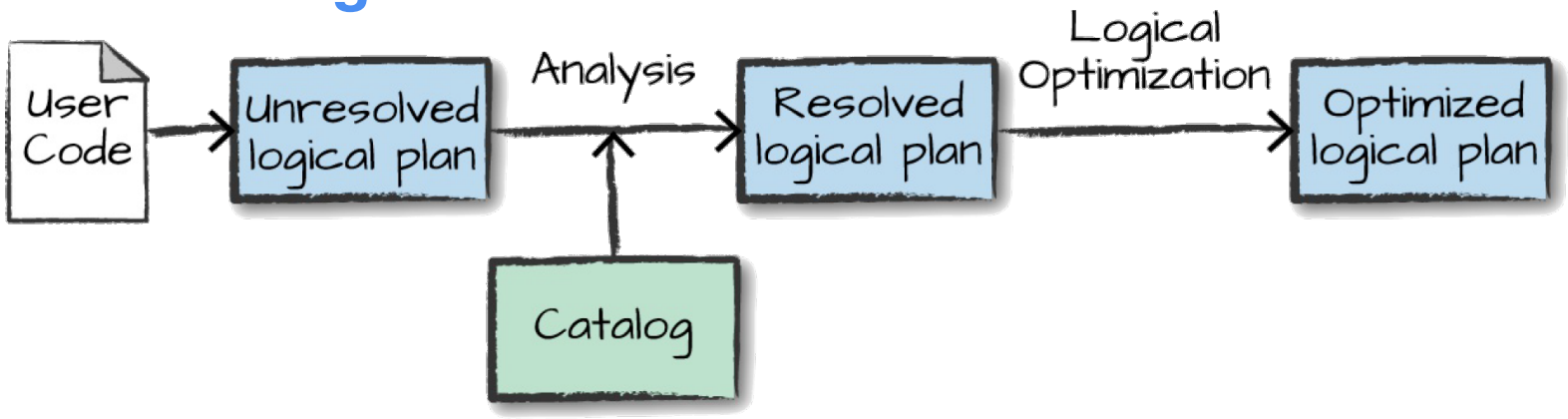
Spark: how it works

- It is interesting to have a look to Spark's workflow:
  1. User writes its code (transformations + actions)
     - With DF functions, SQL, or both
  2. If code is valid, Spark generates a **logical execution plan**
     - A logical plan is an abstraction of the required operations
  3. The logical plan is translated into a **physical plan** by the *Catalyst Optimizer*
     - Aka, *Spark plan*: set of operations optimized for RDDs
  4. The physical plan is run on the cluster

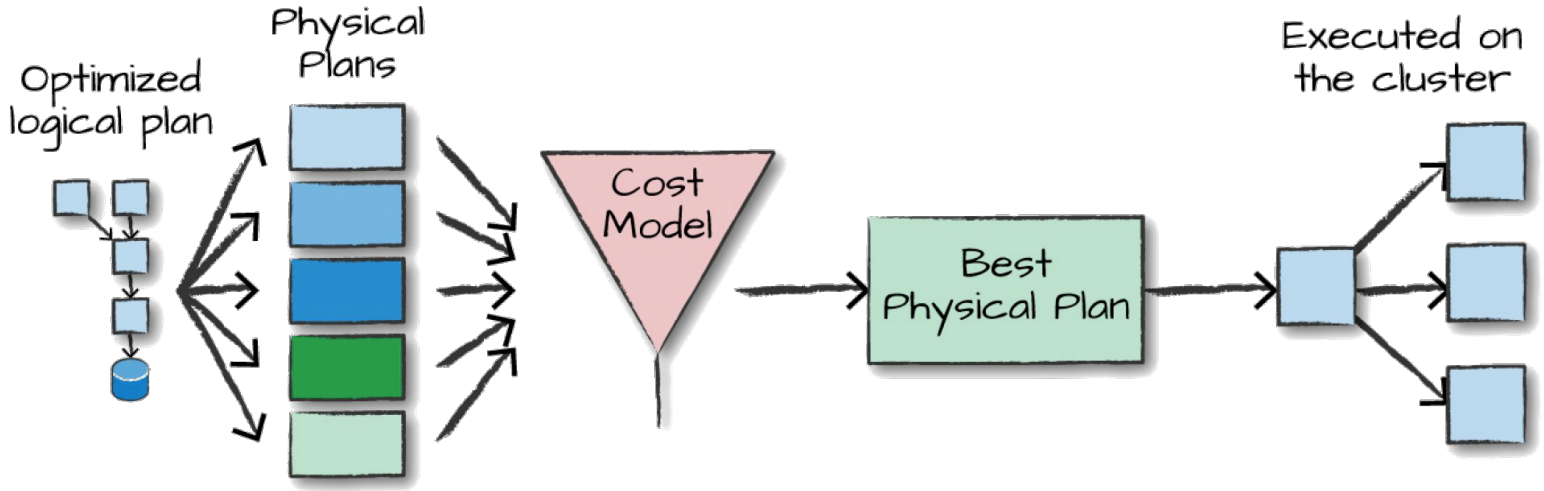


Spark: how it works

1. From **Code** to **Logical Plan**



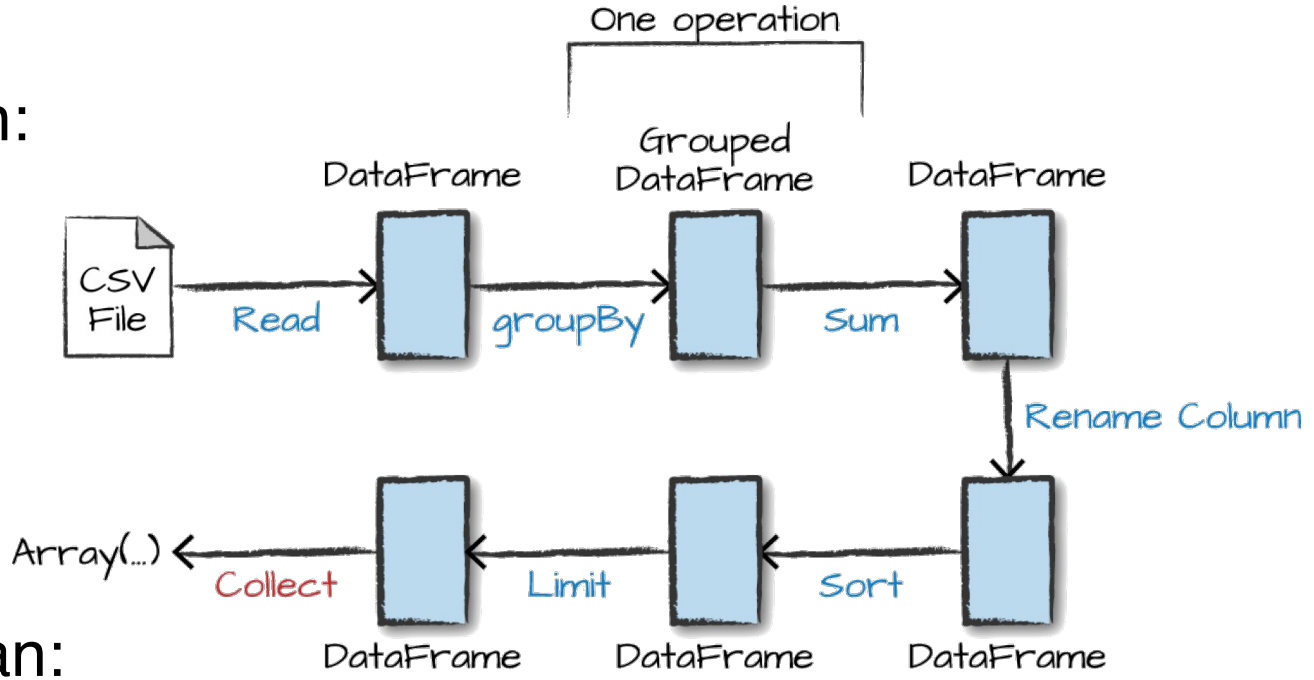
2. From **Logical Plan** to **Physical Plan**



3. Execution (evaluation) of **Physical Plan**

Spark: how it works

- Logical plan:



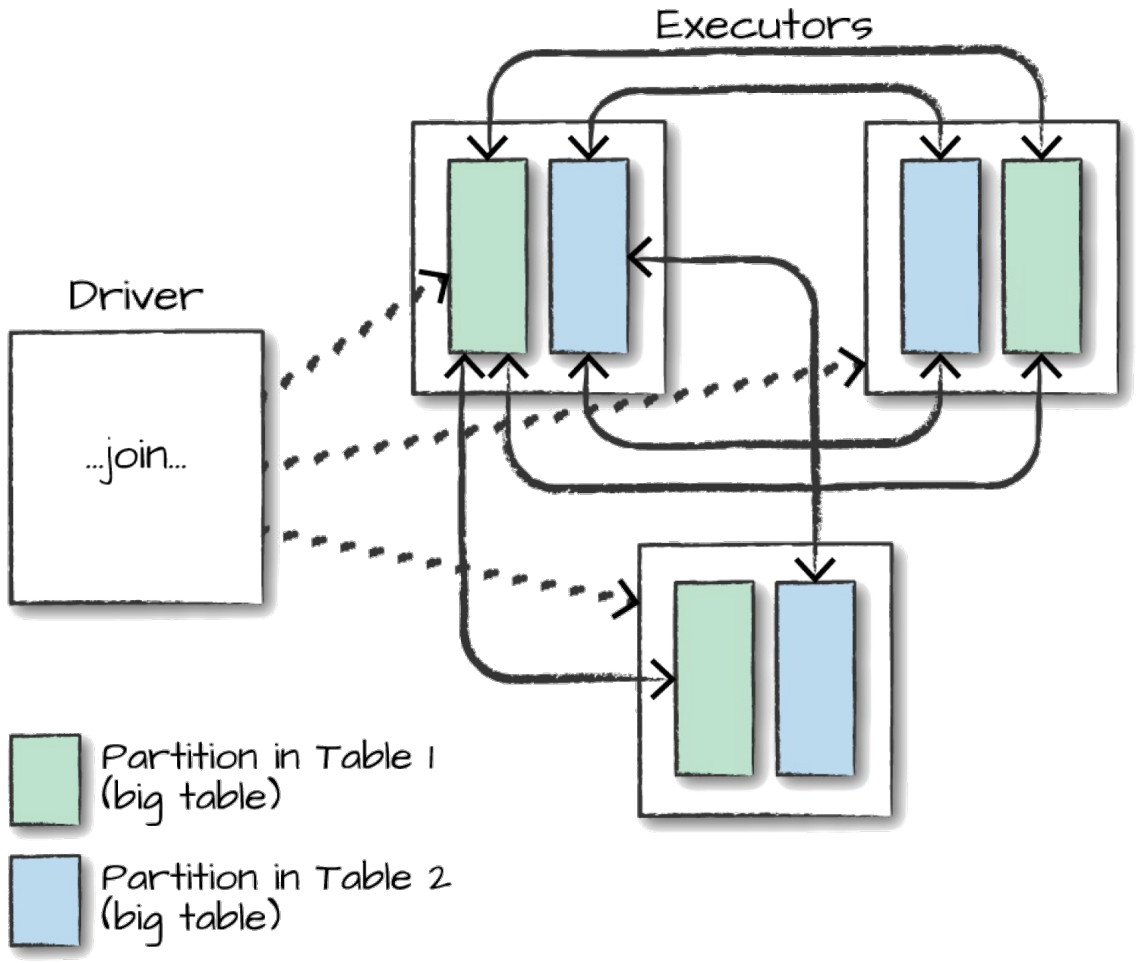
- Physical plan:

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#241L DESC NULLS LAST], output=[DEST
_COUNTRY_NAME#154,destination_total#241L])
+- *(2) HashAggregate(keys=[DEST_COUNTRY_NAME#154], functions=[sum(cast(count#156 as bigin
t))])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#154, 5), true, [id=#536]
      +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#154], functions=[partial_sum(cast(count#1
56 as bigint))])
         +- FileScan csv [DEST_COUNTRY_NAME#154,count#156] Batched: false, DataFilters: [], F
ormat: CSV, Location: InMemoryFileIndex[file:/home/jovyan/data/summary_flights/2015-summary.c
sv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<DEST_COUNTRY_NAME:string,cou
nt:int>
```

Spark: how it works

- When making **joins** *(wide dependency)* in Spark, there are **two core resources** at play:
  - Node-to-node communication strategy
    - Shuffle join (all-to-all communication)
    - Broadcast join
  - Per-node computation strategy

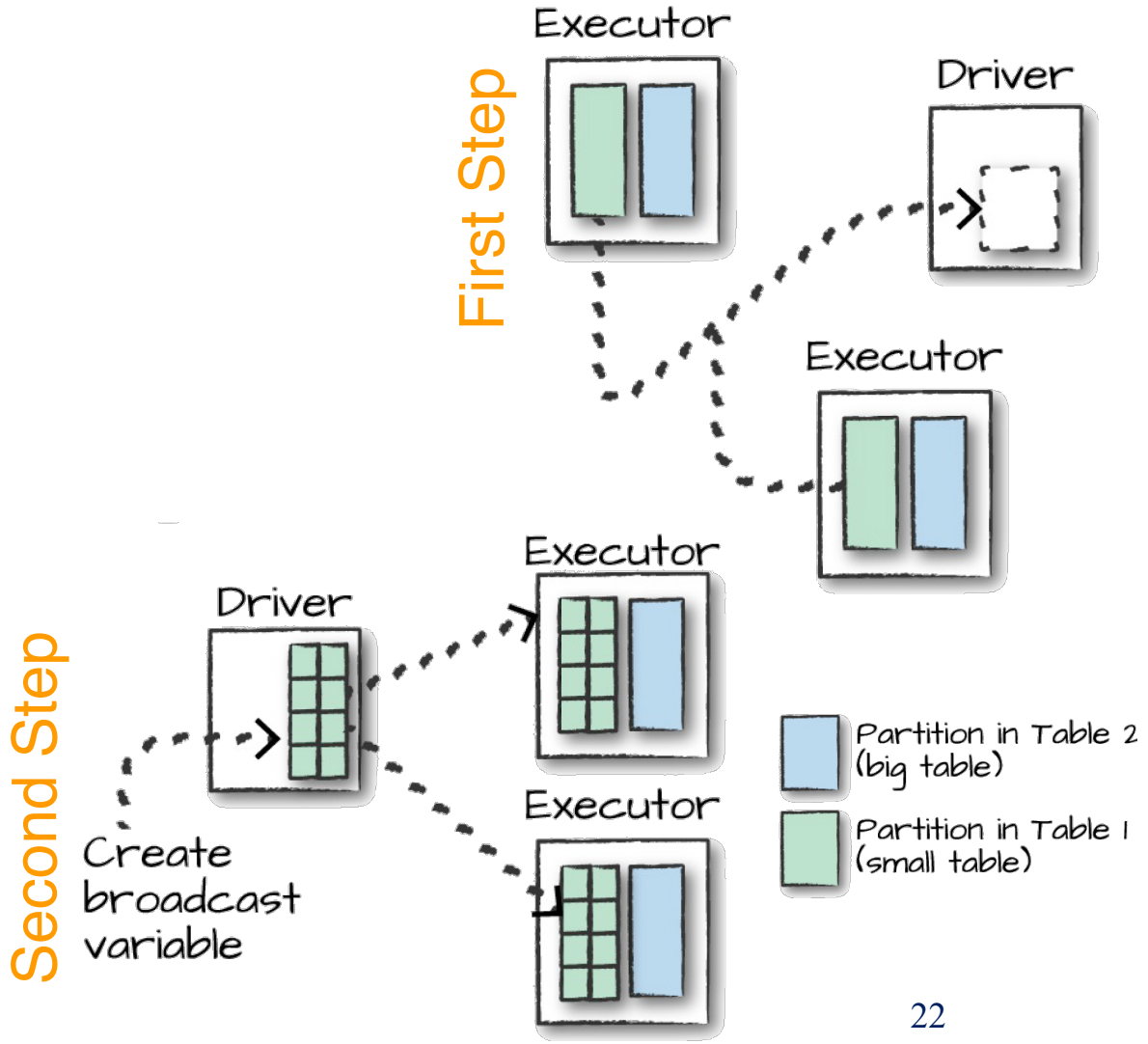
- Two types of joins:
  - **Shuffle join** (two large tables)
    - Expensive
    - Large network traffic
  - **Broadcast join**



Spark: how it works

- When making **joins** *(wide dependency)* in Spark, there are **two core resources** at play:
  - Node-to-node communication strategy
    - Shuffle join (all-to-all communication)
    - Broadcast join
  - Per-node computation strategy

- Two types of joins:
  - **Shuffle join**
  - **Broadcast join** (one not-so-large table fits into main memory)
    - Copy the small table to all the workers
    - Only-once (initial) communication



DataFrames and SQL

- DataFrames can be queried using pure SQL!
  - DF needs to be registered as a table or view before
- Spark runs the same transformations, regardless of the language in which they were expressed, in the exact same way.
  - You can express your query logic in SQL or DataFrames (or both!)
  - Spark will compile that logic down to an underlying plan

```
sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")
```

```
dataFrameWay = flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.count()
```

  - No performance difference as they both “compile” to the same underlying plan:

```
sqlWay.explain() == dataFrameWay.explain()
```

DataFrames and SQL

- Equivalent queries:

```
spark.sql("""
SELECT max(count)
FROM flight_data_2015
""")\
.take(1)
```

```
from pyspark.sql.functions import max
flightData2015.select(max("count")).take(1)
```

```
spark.sql("""
SELECT DEST_COUNTRY_NAME,
sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")\
.show()
```

```
from pyspark.sql.functions import desc
flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)",
                   "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.show()
```

DataFrames and SQL

- Interoperate SQL and DF functionalities

```
spark.sql("SELECT DEST_COUNTRY, sum(count) "+
                  "FROM some_sql_view GROUP BY DEST_COUNTRY")\
  .where("DEST_COUNTRY like 'S%'")\
  .where("`sum(count)` > 10")\
  .explain()
```

```
== Physical Plan ==
*(2) Filter (isnotnull(sum(count)#141L) AND (sum(count)#141L > 10))
+- *(2) HashAggregate(keys=[DEST_COUNTRY#26], functions=[sum(cast(count#28 as bigint))])
   +- Exchange hashpartitioning(DEST_COUNTRY#26, 200), true, [id=#336]
      +- *(1) HashAggregate(keys=[DEST_COUNTRY#26], functions=[partial_sum(cast(count#28 as bigint))])
         +- *(1) Project [DEST_COUNTRY#26, count#28]
            +- *(1) Filter (isnotnull(DEST_COUNTRY#26) AND StartsWith(DEST_COUNTRY#26, S))
               +- FileScan csv [DEST_COUNTRY#26,count#28] Batched: false, DataFilters:
                  [isnotnull(DEST_COUNTRY#26), StartsWith(DEST_COUNTRY#26, S)], Format: CSV, Location:
                  InMemoryFileIndex[file:/home/jovyan/data/summary_flights/2015-summary.csv],
                  PartitionFilters: [], PushedFilters: [IsNotNull(DEST_COUNTRY),
                  StringStartsWith(DEST_COUNTRY,S)], ReadSchema: struct<DEST_COUNTRY:string,count:int>
```

Summary

- Spark is a programming model that uses distributed optimized data structures
- The user specifies transformations
  - Multiple transformations build up a tree of instructions
  - The logical data structures that we manipulate with transformations and actions are DataFrames
  - With each transformation, you create a new DataFrame
- To trigger (start) computation, you call an action
  - An action begins the process of executing that the tree of instructions, as a single job, by breaking it down into stages and tasks to execute across the cluster
  - Types of actions: show results/data, convert to native language types, …

Spark