Spark Starter

by Nicola Orecchini

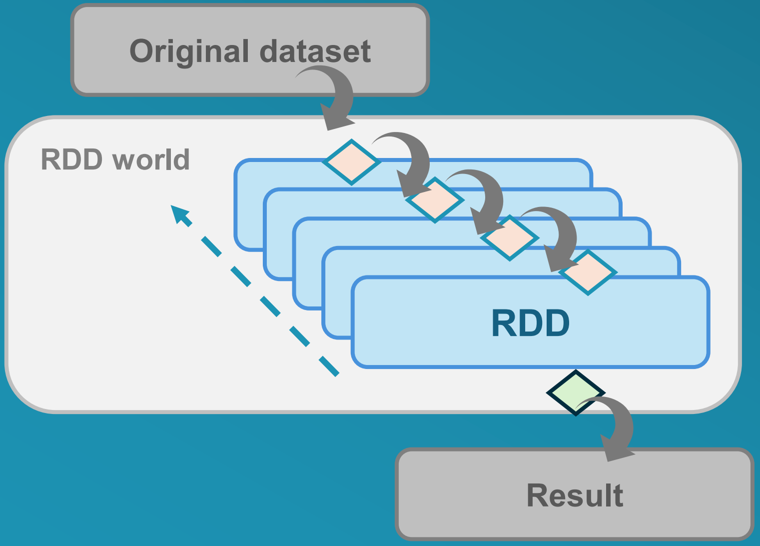
June 2025

Agenda

1. What is Spark
2. Shuffle in depth

1. What is Spark

Spark is a framework designed to process big data leveraging **4 key components**



RDD (Resilient Distributed Dataset)

RDDs are **collections of elements partitioned** across the nodes of the cluster that can be **operated** on in **parallel**

- They contain **immutable**, **lazily evaluated** plans that specify what operations to apply to data residing at a specific location to generate some output

Transformations

Transformations are steps where the user describes (by coding) what operations to perform on data

- The **output** of a transformation is an **RDD**
- The execution of transformations will **not start until an action is triggered**

Actions

Actions represent plans of how to manipulate rows and columns to compute the user's desired result

- Actions bring data **out of the RDD world** into some other storage system
- When we perform an action on an RDD, we instruct Spark to trigger the **evaluation of partitions** via the **DAG**

Lineage (DAG)

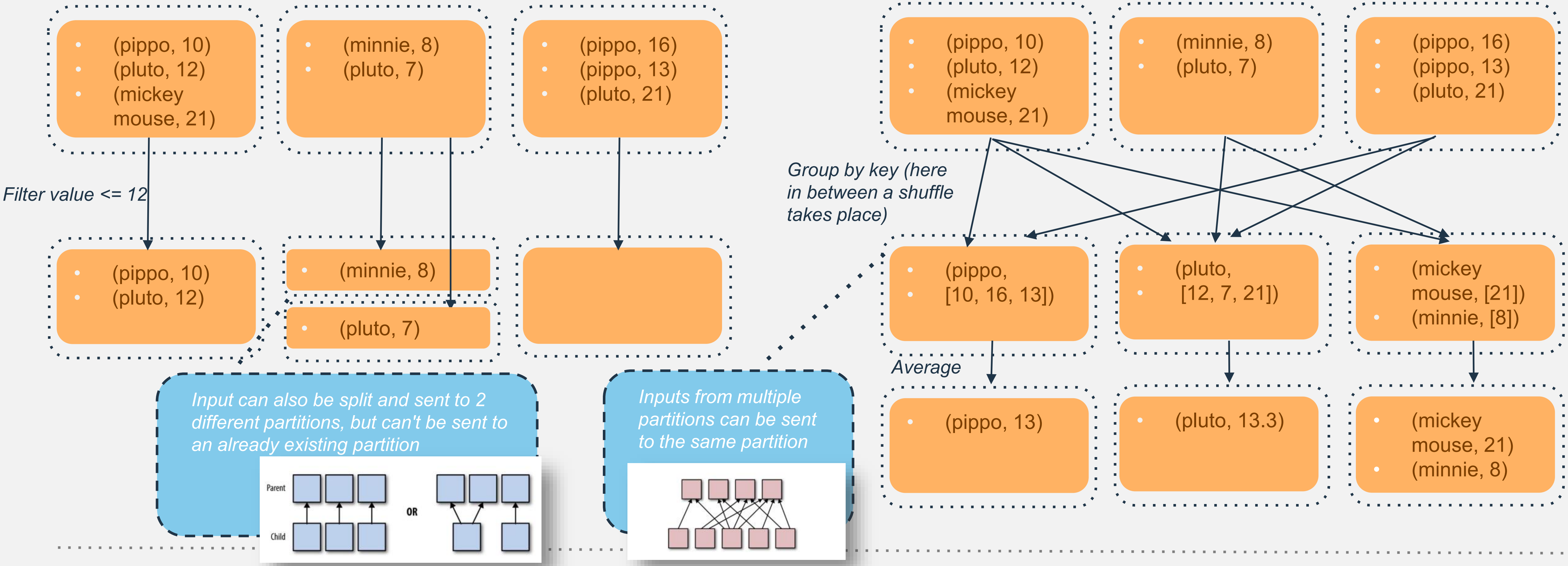
The Lineage represents, by means of a directed acyclic graph (DAG), a sequence of transformations and actions created automatically by Spark, based on the dependencies between RDD transformations. When an action is run:

- The action triggers the **scheduler**, which **builds a DAG**
- Spark then evaluates the action by working backward to define the series of steps it must take to produce each object in the final RDD

Thanks to the DAG, Spark's **fault tolerance** is achieved, because each partition of the data contains the dependency information needed to recalculate the partition

Transformations can be **narrow** and **wide,** based on whether the executors in different partitions need to communicate with each other

Narrow transformations / **Wide** transformations



- Executors **don't need to communicate** with each other
- Examples: map, filter

- Executors **need to communicate** with each other
- Examples: groupByKey, Join

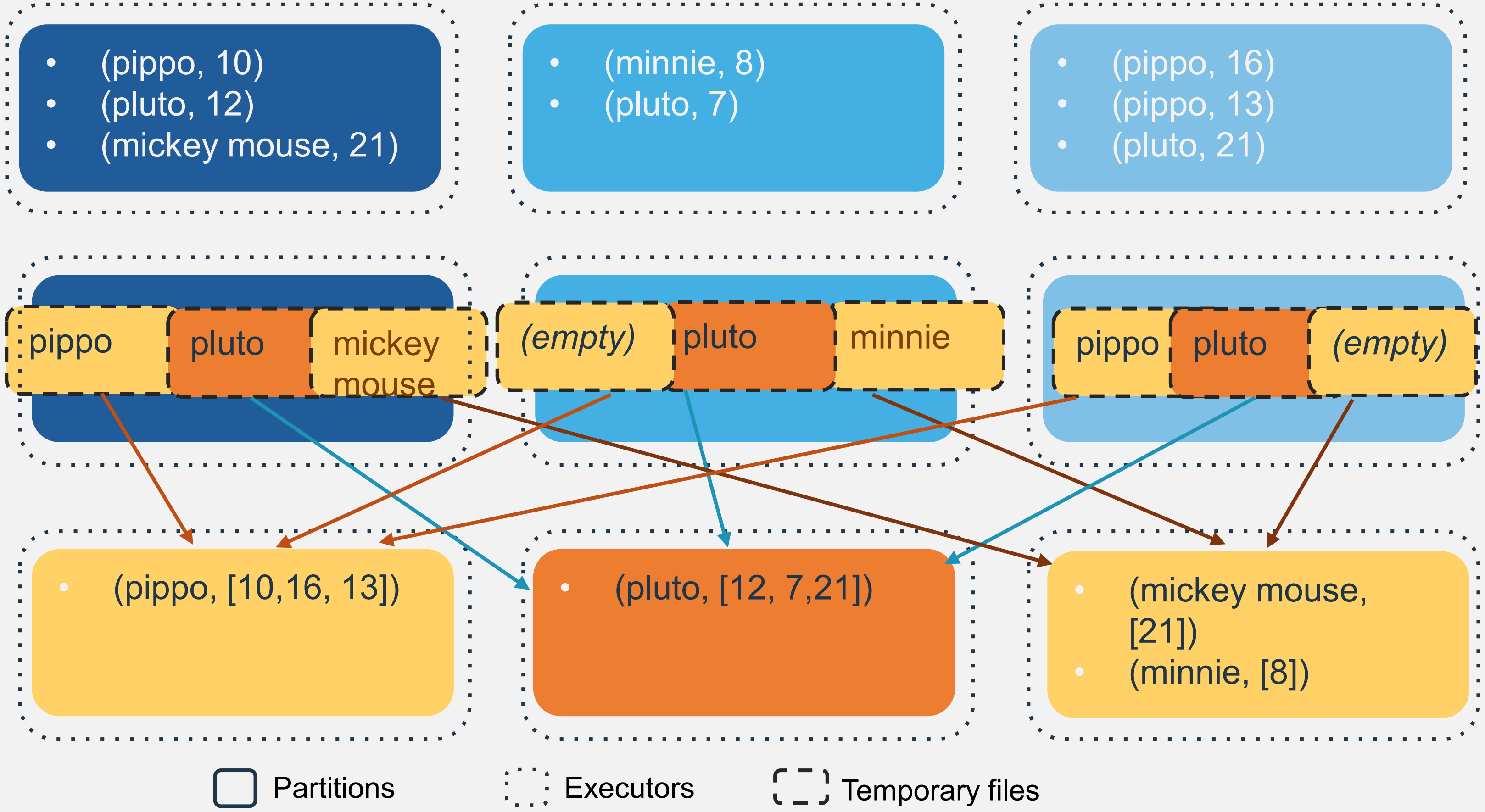
2. Shuffle in depth

When a **transformation** requires data from **partitions other than itself** (e.g., sum all values in a column), **data** is **rearranged** between partitions through a shuffle

Partitioning: the original dataset is split into **partitions** to distribute work across the cluster and reduce memory requirements of each node

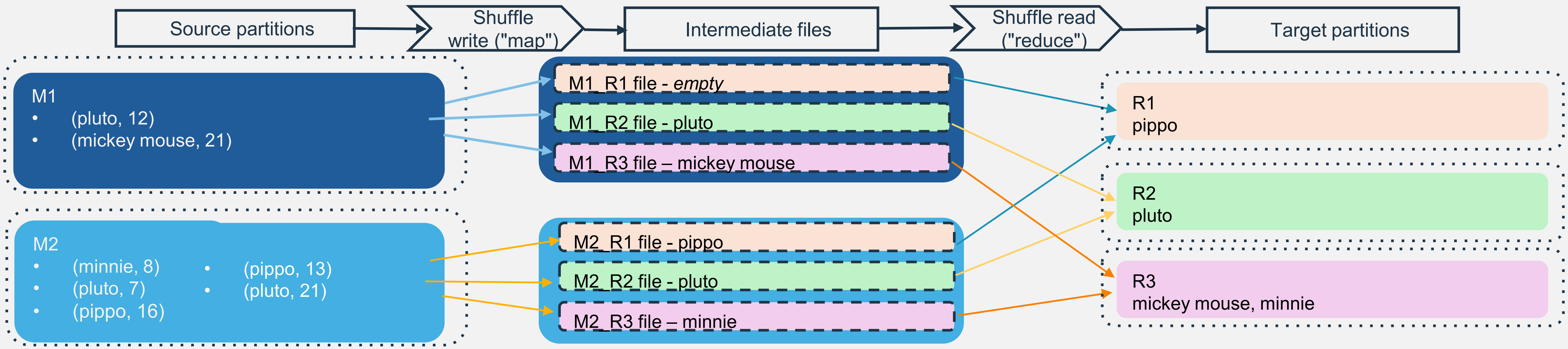
Shuffle Write: each exec takes its partition and **splits its data by key hash** into multiple blocks, **writing** them to **temporary intermediate blocks** (files on disk), one per dest. partition

Shuffle Read: each new partition reads data from all shuffle blocks, across all executors, via **disk I/O** (reading the temporary shuffle files), and **network I/O** (pulling them from all the machines), and assembles bew partitions



- Shuffle is costly: it involves copying data across executors and machines
- Shuffle can be a bottleneck: if key distribution is skewed, then more data will be placed in one partition than another, taking longer to be processed. As the next stage of processing cannot begin until all partitions are evaluated, overall results will be delayed

Hash-based Shuffle



Each key is assigned to a specific reducer by means of a hash function and a *MOD # reducers* operation

Mapper (source partition)	Key	**Assigned Reducer** = Hash(Key) MOD 3
M1	pluto	1
	mickey mouse	2
M2	minnie	2
	pluto	1
	pippo	0
	pippo	0
	pluto	1

M=2 mappers

Each mapper creates a # of new files equal to the # of reducers (=1 file per reducer), e.g. *M1_R1, M1_R2,* etc.

Then, the mapper writes to the files according to the hash calculated in the previous step.

If in the initial partition there are no keys that hash to a specific number, the corresponding file will be empty, but will still be created. Why?

- Because reducers expect to receive one chunk of input from every mapper, even if it's empty
- This way, no reducer is left waiting and everything can be parallelized predictably

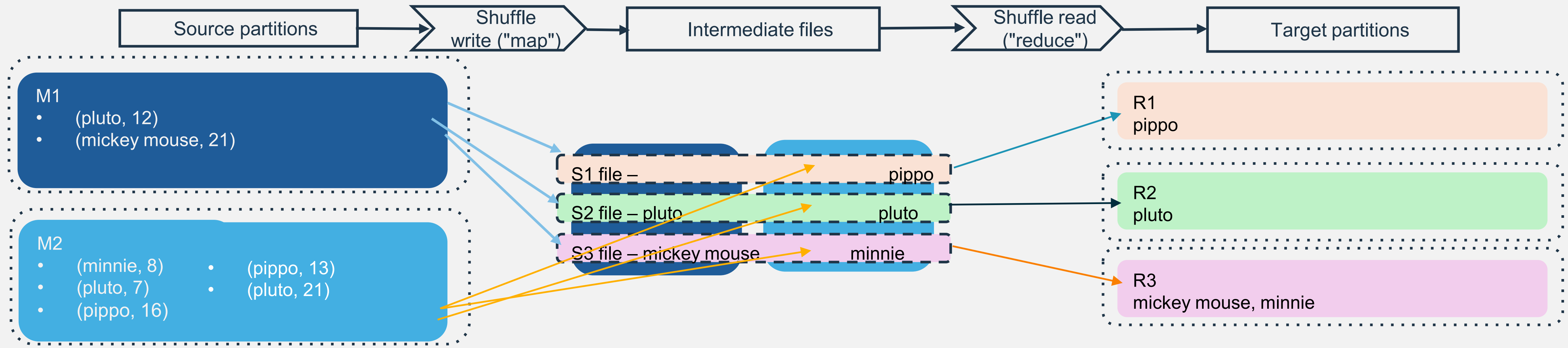
MxR= 6 files written

Each reducer (i.e., target partition) reads data from exactly M files associated to its key

No need to have existence controls on files: each mapper will have emitted a file for each key, even if empty. This speeds up the shuffle read process

R=3 reducers

Hash-based Shuffle with Consolidation



Suppose to have the following shuffling problem:

- 46k partitions as source (46k "mappers"), to be reshuffled into 46k partitions as output (46k "reducers")

With a hash-based shuffle, we would need to write 2B files (46k x 46k). But the cluster can't write all these files in parallel! In fact, if my cluster is made of:

- 100 executors
- each with 10 cores

then we will have 1k cores of computing power.

Now, if 1 task (i.e., 1 "mapper") needs 1 core, then with 1k cores we can run max 1k "mappers" in parallel. And if each of these 1k "mappers" outputs a group of 46k files (the number of "reducers"), then, in a single run, we can write at most 46k * 1k = 46M fi

This means that to write all the 2B files, we will need to write 46 bunches of 46M each.
Idea: after having run the first job and having obtained the first 46M files, why don't we **reuse these files** instead of starting 46M new ones from scratch?

So, we continue sending the data into the same files, keeping the offset to identify which records come from which mapper

Shared File	Keys from M1	Keys from M2
S1	*none*	pippo
S2	pluto	pluto
S3	mickey m.	minnie

With consolidation, Spark creates a pool of shared files, reused across waves of mappers, reducing the total file count to #cores × #reducers

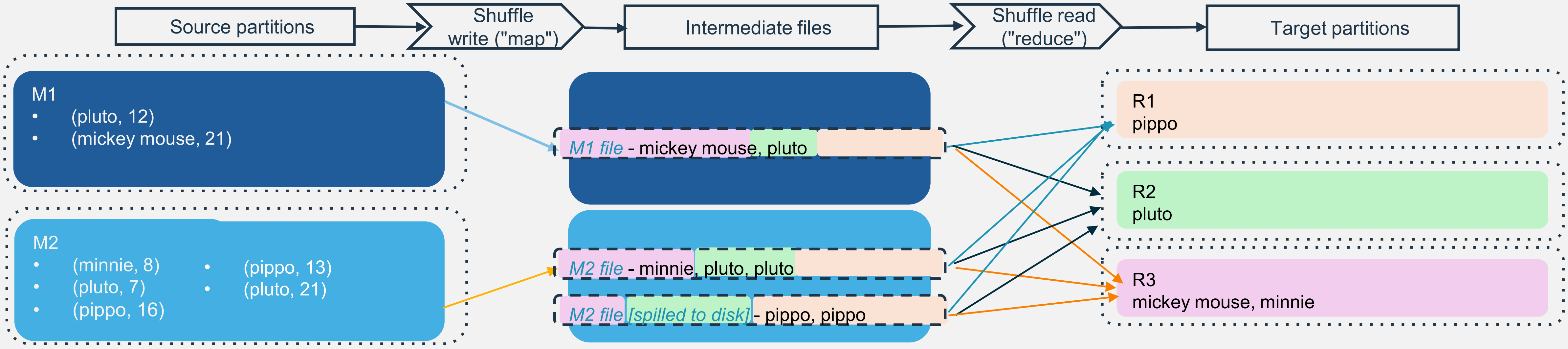
M=2 mappers

C=1 cores

RxC=3 files written

R=3 reducers

Sort-based Shuffle



Each mapper writes 1 file per mapper, **sorting the data** by **reducer ID**, and keeping an **offset index** of where each reducer's chunk is inside it.

As data gets sorted in the initial partitions, this kind of shuffle is useful for **map-side combine operations**

If data doesn't fit in memory, Spark starts spilling to disk. But before doing so, it sorts the spilled data.

Each mapper's spill is written to a separate file (e.g., M1 spill, M2 spill, etc.)

When the reducer comes in, Spark merges those spills in real-time using a priority queue (MinHeap-style)

When a reducer reads data from the output files, it goes through all files (including the spilled ones) and receives the **offset** of that file where needed data is stored

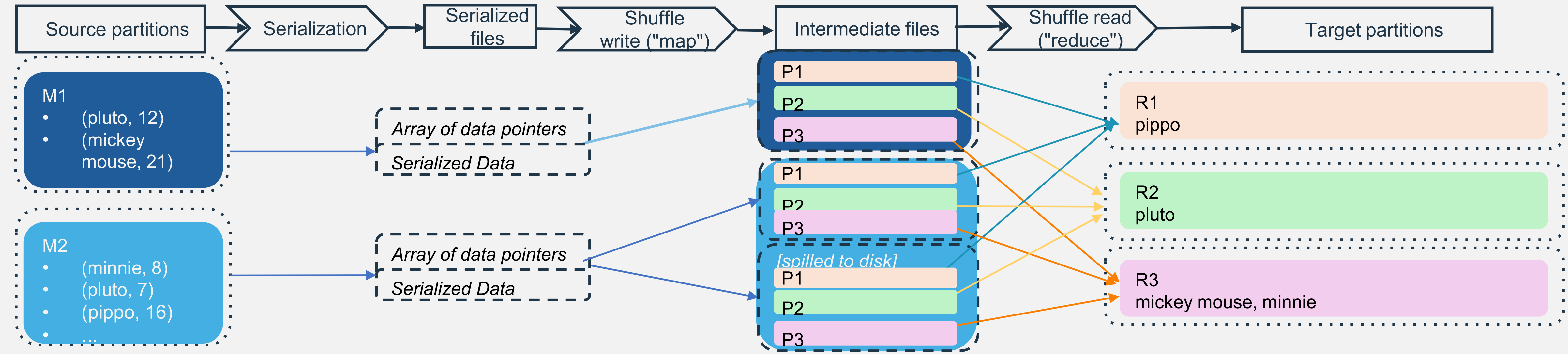
M=2 mappers

M + spilled files written

R=3 reducers

Source:Spark Architecture: Shuffle | Distributed Systems Architecture

Tungsten Sort or Unsafe Shuffle



Serializing means processing the records using their memory reference rather than the records itself, so without information about what the record contains (e.g., the key).

Spark usually works as follows:

- Deserialize data → process (e.g., shuffle) → reserialize → spill to disk

Tungsten shuffle is a technique designed to work with **serialized data** for the whole process:

- Keep the data serialized from start to finish
- Sort by just moving around light pointers, not the full records
- Use raw memory operations like pointer arithmetic
- Spill to d[illegible]hout ever deserializing

After the serialization step, the shuffle follows the same logic of the sort-based shuffle.
Note that this shuffle technique only works if:

- You **don't aggregate data** (no groupBy with sum/count/avg/etc.)
    - Why? There are two broad types of wide transformations:
        - Repartition-like: only care about which partition a record goes to (e.g. repartition, coalesce)
        - Aggregation-like: need to read/compare/merge data inside partitions (e.g. groupBy, reduceBy, aggregateByKey)

    In both cases Spark shuffles data — but the second case needs to know what's inside the boxes, so data must be unserialized
- Your serializer supports **byte relocation** (hello Kryo!)
- You have **<16 million partitions**
- You're **not stuffing 128MB+ per record**

M=2 mappers

M + spilled files written

R=3 reducers