



POLITECNICO
MILANO 1863

Apache Spark

Alessandro Margara

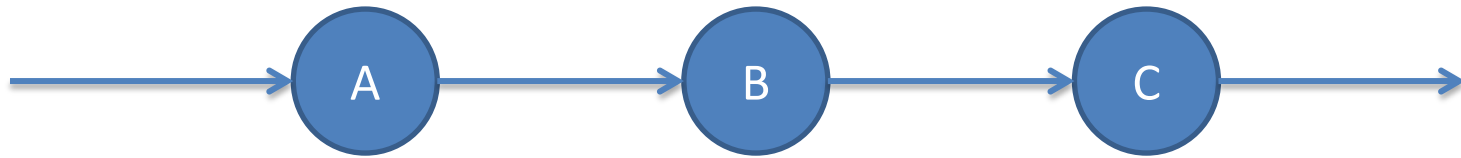
alessandro.margara@polimi.it

<https://margara.faculty.polimi.it>

Big Data processing

- Technologies to enable large-scale data analytics
 - Extract knowledge from large volumes of data
 - Example: study user behaviors to customize advertisements
- Pioneered in the early 2000s by the Google MapReduce framework
 - Computations on cluster computing infrastructures
 - Simple programming API
 - The framework abstracts away most distribution concerns
 - Data distribution
 - Scheduling of computation
 - Fault tolerance
 - Stragglers

Dataflow programming model

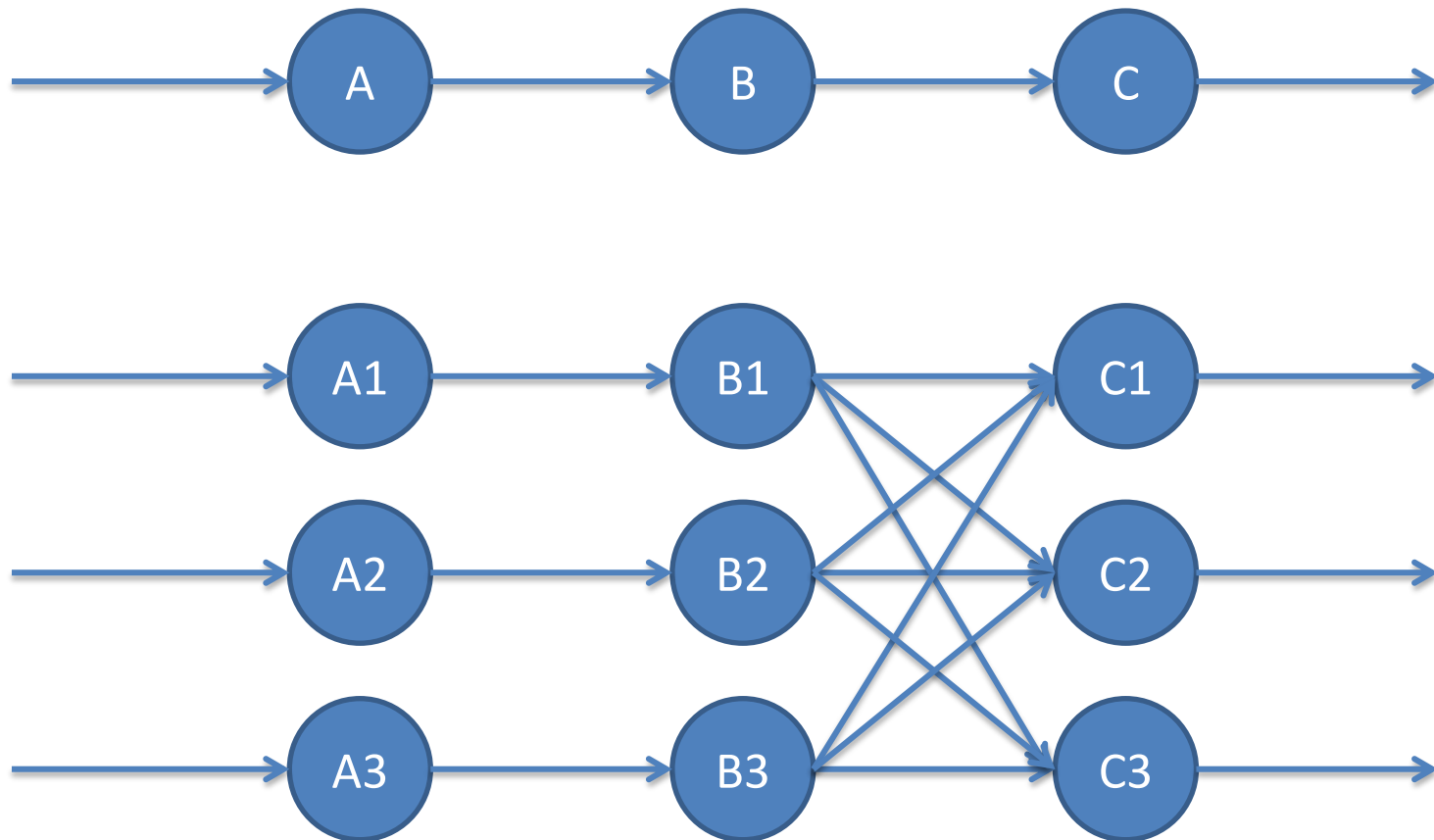


Edges: data

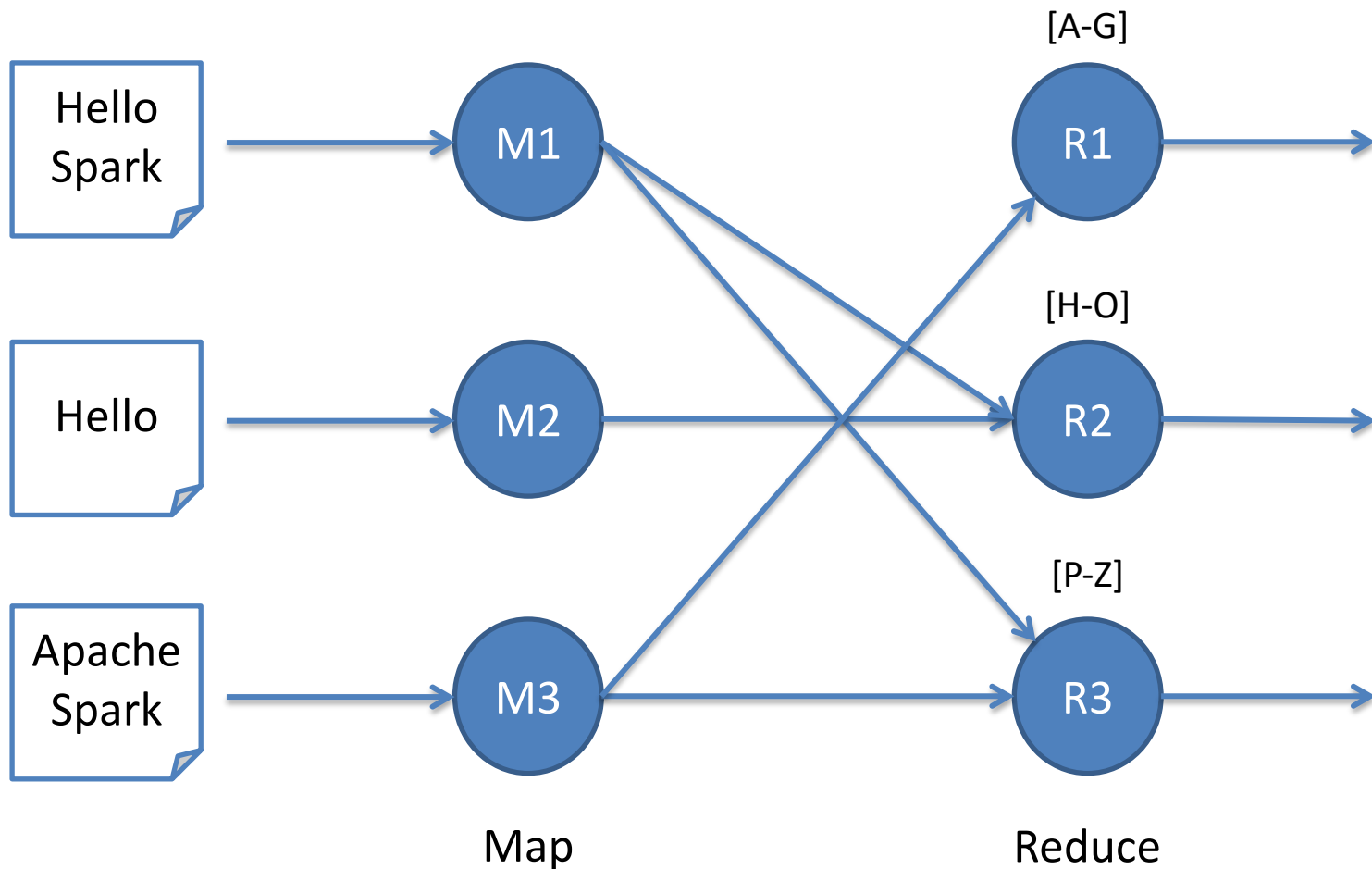
Vertex: Operators

Independent within each toher, they do not share memory

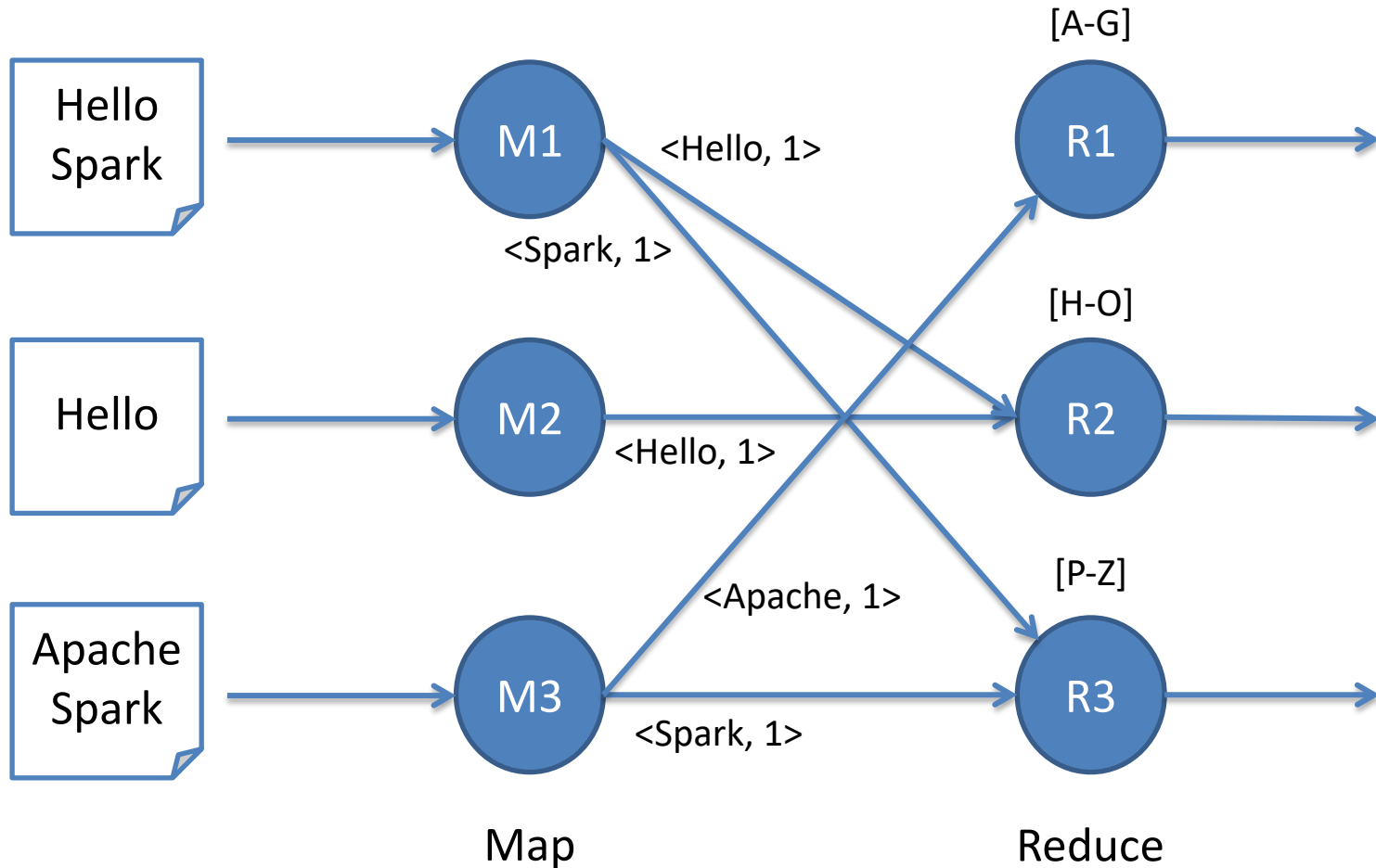
Dataflow programming model



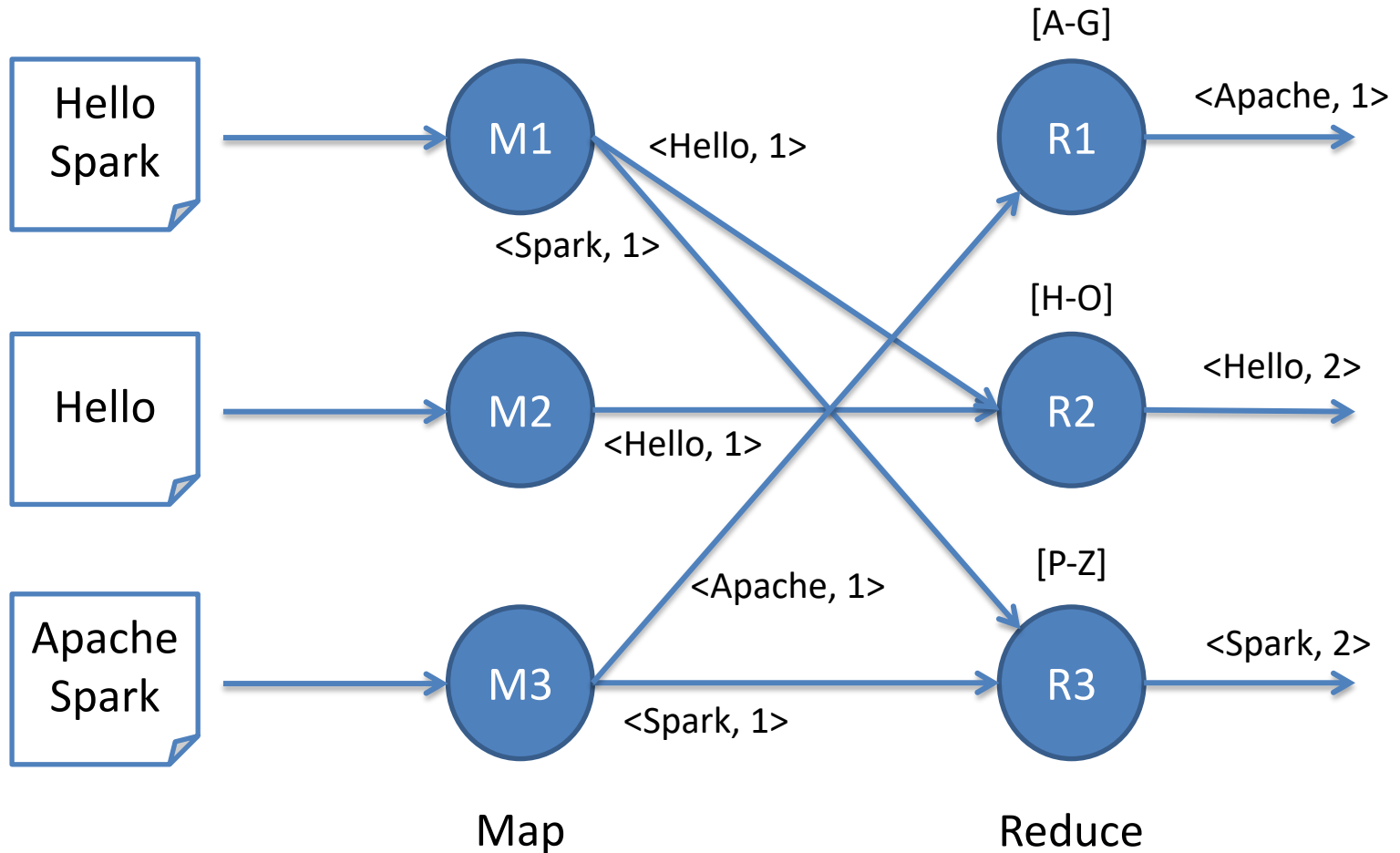
Example: word count



Example: word count



Example: word count



MapReduce: programming model

- The original MapReduce framework was restricted to these two stages
 - Map and Reduce
- Any computation requiring more stages needed to be translated into multiple invocations of the framework
 - Input and output results were written on disk
 - Distributed filesystem: data partitioning and replication

MapReduce: programming model

- Developers write programs from the perspective of individual elements
- Map: given an input document, how to output the occurrences of its words?
- Reducers: given a word and a list of occurrences, how to sum up all occurrences?

MapReduce: execution

- A scheduler instantiates Map workers
 - They save their (intermediate) results on disk
 - Distributed filesystem designed to efficiently handle sequential read of large data blocks
- The scheduler instantiates Reduce workers
 - Data locality: Reduce workers instantiated close to their input data
 - Possibly, same machine: read from local disk

MapReduce: execution

- Benefit of the programming model
 - Operators are independent and stateless
 - Their output only depends on their input
- This simplifies scheduling and fault-tolerance
 - Operators do not need to synchronize with each other
 - In the case of failures, individual operators can simply be re-executed
 - They will produce the same output

MapReduce: execution

- Parallelism
 - Run the same computation in parallel on multiple input elements
- Communication
 - The framework takes care of writing intermediate results
- Scheduling
 - Based on data locality to speed-up computation
 - Tasks are scheduled as close as possible to the input data they consume
- Fault-tolerance / stragglers
 - In the case of failure or slow nodes ...
 - ... reschedule failed or slow nodes and recompute their results

Apache Spark

Programming model overview

Apache Spark

- Unified analytics engine for large-scale data processing
 - Batch, streaming, structured, machine learning, graph processing, ...
- Based on the dataflow programming model
- Same execution strategy as MapReduce ...
 - Stages are scheduled
- ... but more expressive ...
 - Not limited to two stages
 - Support for iterative computations
- ... and more efficient
 - E.g., by caching data in main memory

Apache Spark

- A Spark cluster consists of a *master* and one or more workers
 - Typical configuration: one worker per host, using as many cores as available in the host
- The master
 - Accepts jobs from *driver* programs, and
 - Splits jobs into stages and then into elementary tasks
 - Schedules tasks on available workers

Apache Spark

- We will write *driver programs* that submit jobs to the cluster
 - Each job consists of various *parallel operations*
- The main abstraction that Spark provides is the RDD (resilient distributed dataset)
 - Collection of elements
 - Partitioned across the nodes of the cluster
 - Can be processed in parallel
 - Fault tolerant

Apache Spark

- A Spark program accesses the Spark cluster through a `SparkContext` object
- Contains relevant parameters
 - Name of the Spark application / job
 - Address of the master
 - ...
- Only one context can be active per JVM
 - `close()` closes a context and enables starting a new one

Initializing Spark

```
SparkConf conf = new SparkConf()  
    .setAppName(appName)  
    .setMaster(master);  
JavaSparkContext sc = new JavaSparkContext(conf);  
  
...  
  
sc.close();
```

Initializing Spark

- When running in a real cluster
 - The application is packaged in a jar file
 - The jar is submitted to the cluster with a provided script
- Local mode
 - Run the driver and the workers in the same JVM
 - For testing and debugging
 - Setting the master to local[n] requests Spark to use n cores to run the workers

RDDs: initialization

- RDDs can be created from
 - Existing collections in the driver program
 - External datasets
 - Local filesystem
 - HDFS
 - Kafka topics
 - Several DBMSs
 - ...

RDDs: initialization

- RDD from local collection

```
List<String> data = Lists.newArrayList("A", "B", "C", "D");  
JavaRDD<String> dataRDD = sc.parallelize(data);
```

- RDD from file

```
JavaRDD<String> dataRDD = sc.textFile("filename.txt");
```

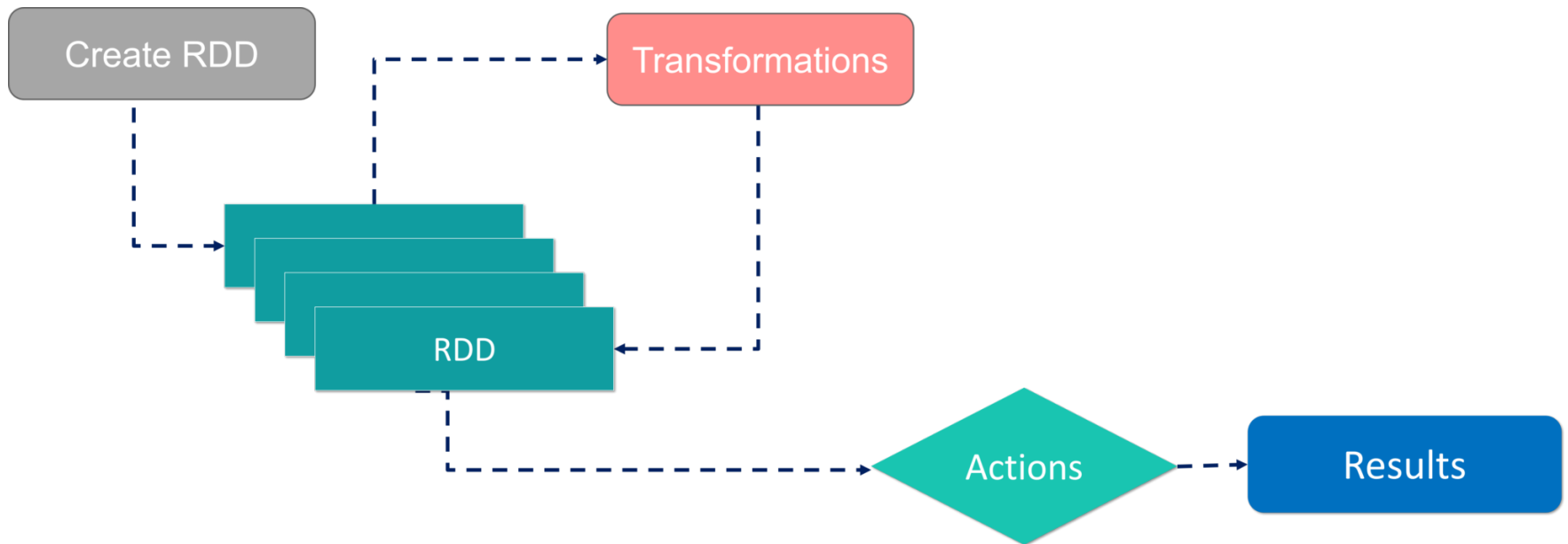
RDDs: operations

- RDDs support two types of operations
 - *Transformations* create a new RDD from an existing one
 - *Actions* return a value to the driver program after running a computation on the dataset
- All transformations are lazy
 - They do not compute the results when invoked
 - They remember the computation to be implemented, ...
 - ... and execute the computation when an action requires a result to be returned to the driver program

RDDs: operations

- By default, each transformed RDD is recomputed each time the driver runs (directly or indirectly) an action on it
- The programmer can also persist/cache an RDD in memory (in the workers) for faster access in subsequent queries
- Various storage levels available
 - In-memory objects
 - In-memory serialized objects
 - In-memory + on disk serialized objects
 - On disk serialized objects
 - ...

RDD: operations



RDDs: simple example

```
/* Creates an RDD. The number of partitions is decided  
based on the available workers */
```

```
JavaRDD<String> lines = sc.textFile("data.txt");
```

```
/* Transformation. Map applies a function to each and every  
element of the original RDD. In this case, it transforms  
each string (line) into a number (the length of the line)  
*/
```

```
JavaRDD<Integer> linesLen = lines.map(s -> s.length());
```

```
/* Action. Reduce aggregates all the values in a single  
element and returns the result to the driver. In this case,  
it returns the sum of all the length of all the lines) */  
int totLen = linesLen.reduce((a, b) -> a + b);
```

Apache Spark

Architecture and execution model

Spark execution model

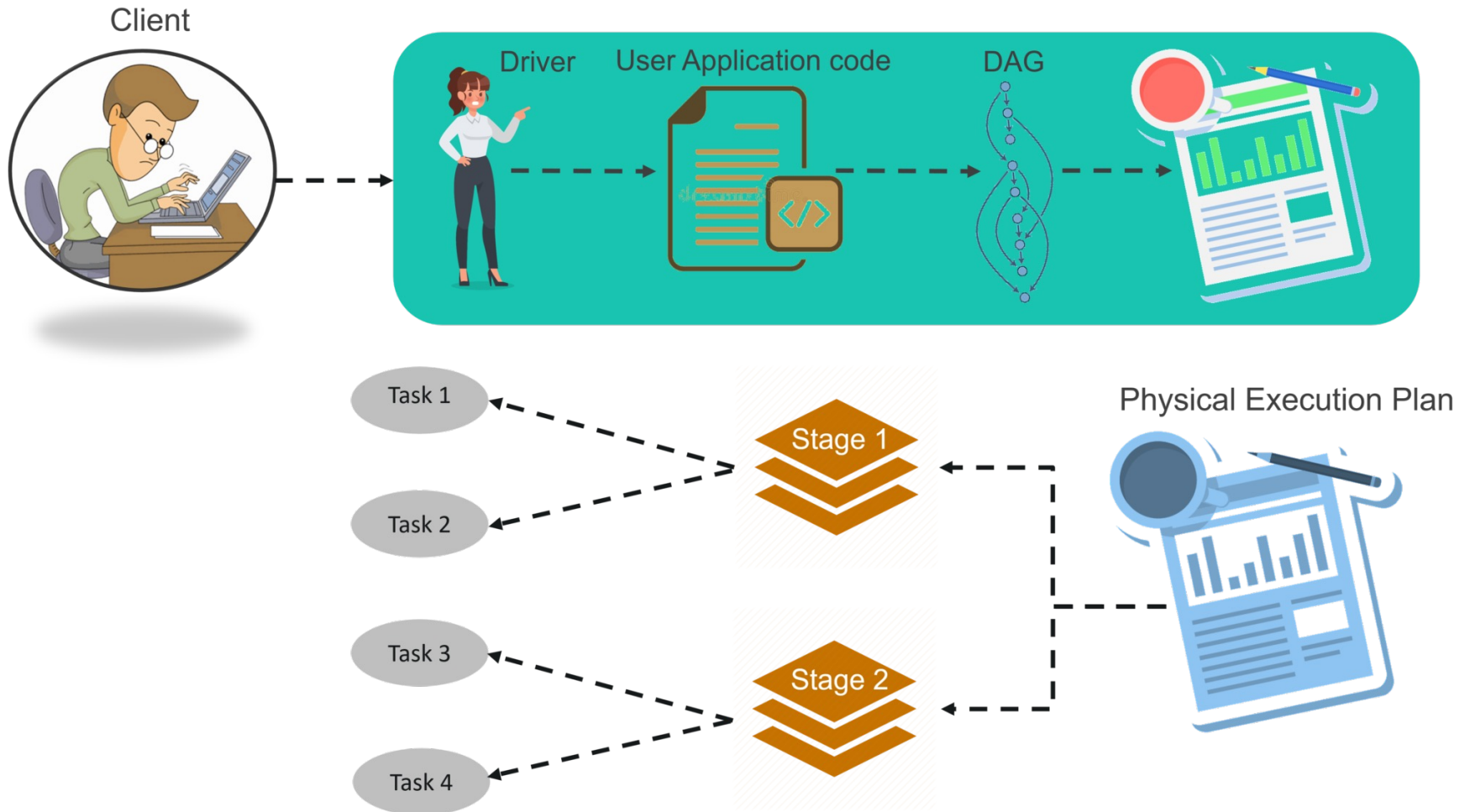
- A driver program can run on any machine
 - In cluster mode, it is typically executed in the master
- Whenever encountering an action in the driver program, Spark creates a new data processing job
- The job consists of a directed acyclic graph (DAG) of operators
 - Dataflow model
 - Intermediate nodes are transformations, and the final node is the action

Spark execution model

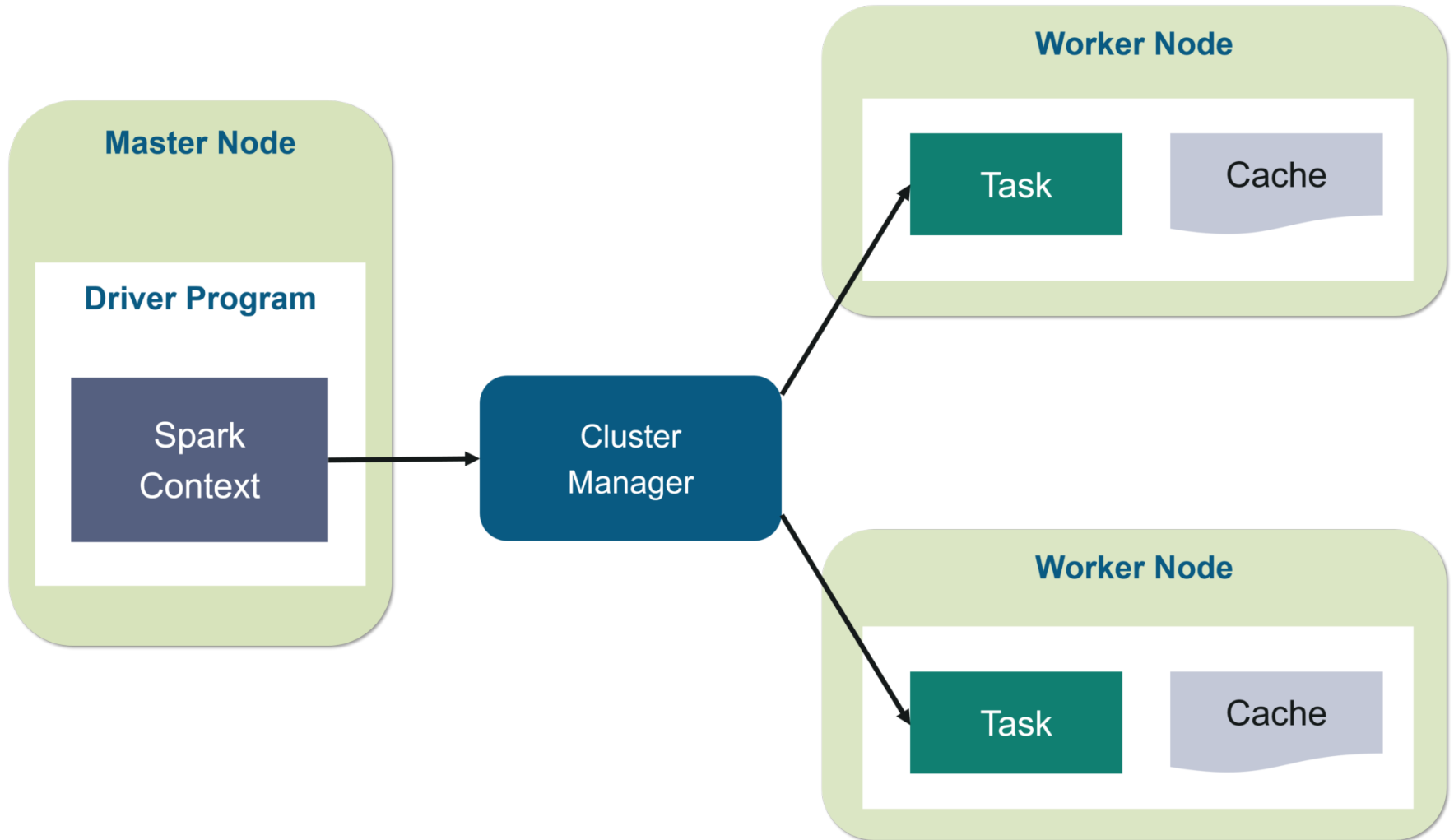
Creating a job

1. The Spark context determines the topology of the DAG
2. The logical DAG is transformed into a physical execution plan
 - Multiple *stages*: each stage contains a sequence of operations with no intermediate data shuffle
3. Each stage is split into tasks (one for each partition)
4. Tasks are scheduled on the cluster
 - Where / close to the data they consume
 - Considering the dependencies between tasks

Spark execution model



Spark architecture



Spark architecture

- When persisted, RDDs are stored in the cache of worker nodes
 - Memory or disk
- RDDs are partitioned across workers according to the specified key
- Tasks are scheduled where the data they consume is located

Fault tolerance

- Fault tolerance based on lineage
 - If an RDD is lost due to a failure just recompute it starting from its input RDDs
 - If any of its input RDDs is lost just recompute it starting from its input RDDs
 - ...
 - Apply recursively until some available RDD is found
- In the worst case, recompute the RDD starting from the source
- Important: since RDDs are partitioned, only lost partitions are recomputed when needed, not the entire dataset