Big Data Technologies

Spark

Lionel Fillatre
Polytech Nice Sophia
lionel.fillatre@univ-cotedazur.fr

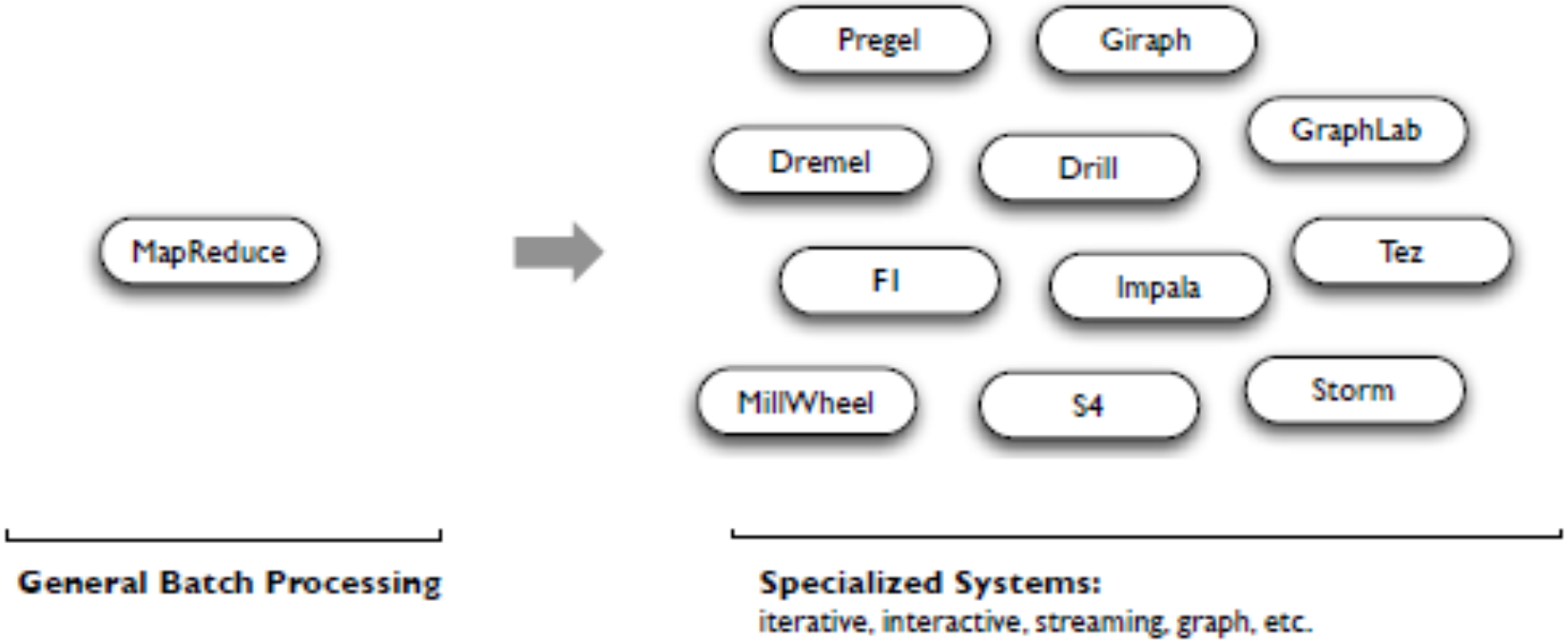
Outlines

- What is Spark?
- Spark Core
- Resilient Distributed Dataset (RDD)
- Programming Model
- Transformations
- Actions
- Task Scheduling
- Persistence
- Shared Variables
- Conclusion

What is Spark

MapReduce: some drawbacks

- MapReduce greatly simplified “big data” analysis on large, unreliable clusters
- But as soon as it got popular, users wanted more:
  - More complex, multi-stage applications (e.g. iterative machine learning & graph processing)
  - More interactive ad-hoc queries



Spark

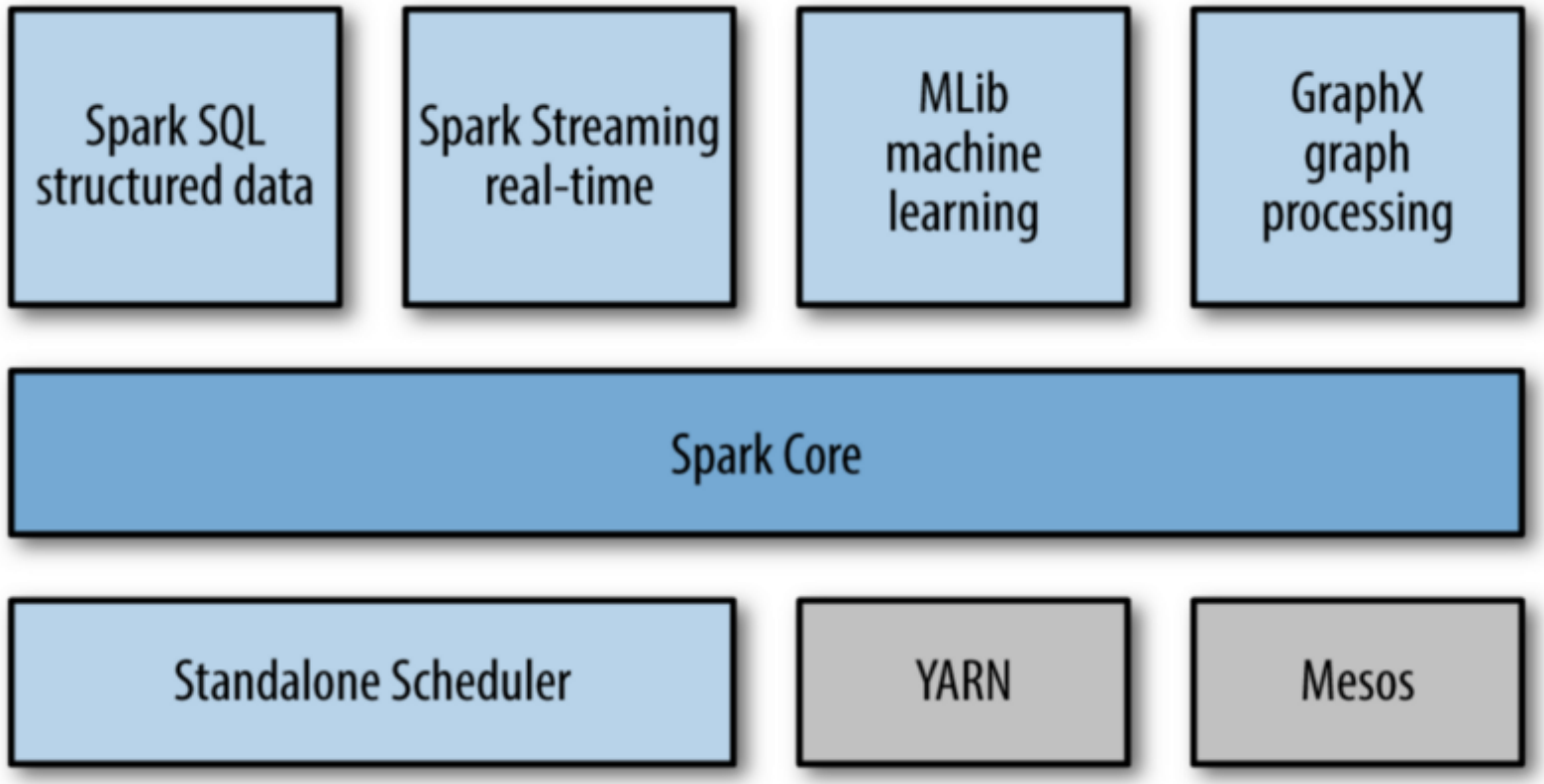
- Unlike the various specialized systems, Spark's goal was to generalize MapReduce to support new apps within same engine
    - Map/reduce is just one set of supported constructs

- Two reasonably small additions are enough to express the previous models:
    - Fast data sharing
    - General DAGs (Directed Acyclic Graph)

- This allows for an approach which is more efficient for the engine, and much simpler for the end users

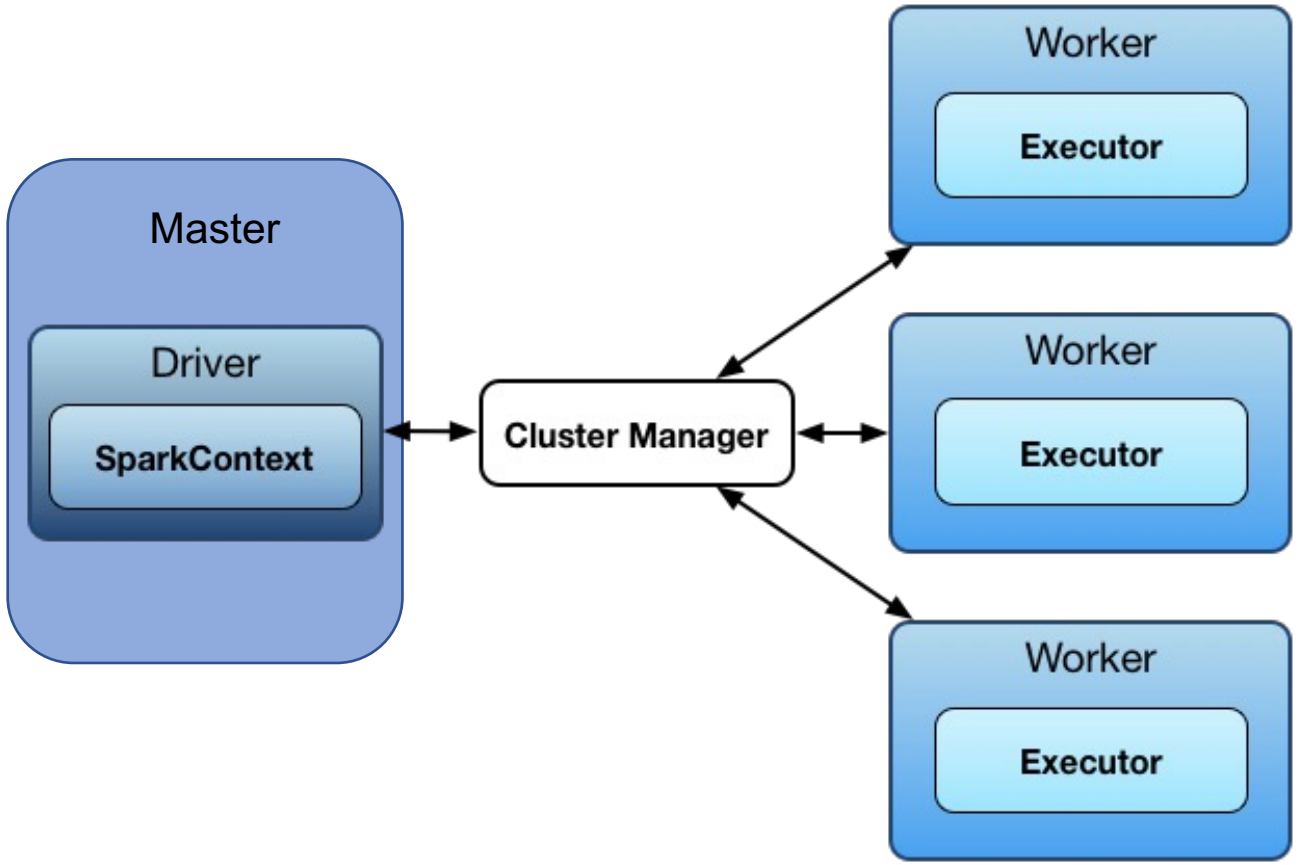
- Programming at a higher level of abstraction

A general view of Spark



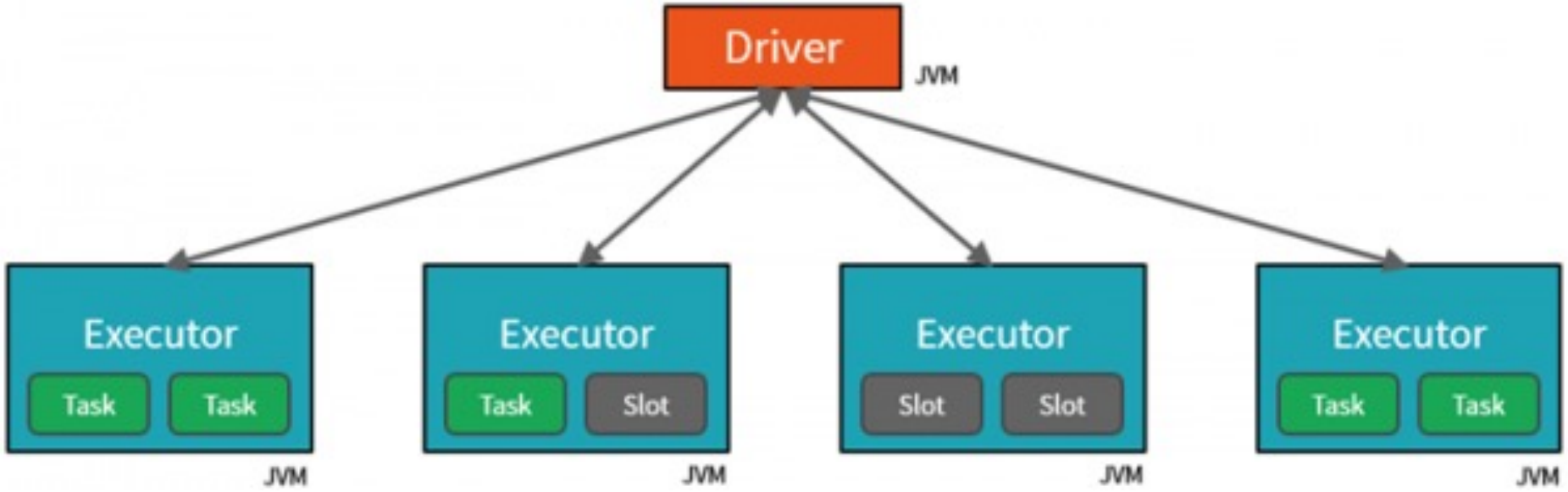
Master/worker architecture

- **Master:** running instance of spark (daemon) that manages the ressource to run driver and executors.
- **Workers:** running spark instance (daemon) where executor will be created to run tasks.
- **Driver:** JVM process that hosts SparkContext for a Spark application. It's responsible to running executor process.
- **Executors:** JVM process that executes multiple tasks and stores RDD.



Driver/Executor architecture

- **Driver:** It is a JVM process that hosts SparkContext for a Spark application. It's responsible to running executor process where task scheduler schedule tasks. It coordinates to workers and overall execution of tasks.
- **Executors:** It's JVM process that execute multiple tasks and provide in-memory storage for RDD. An executor have multiple slots to run multiple tasks.

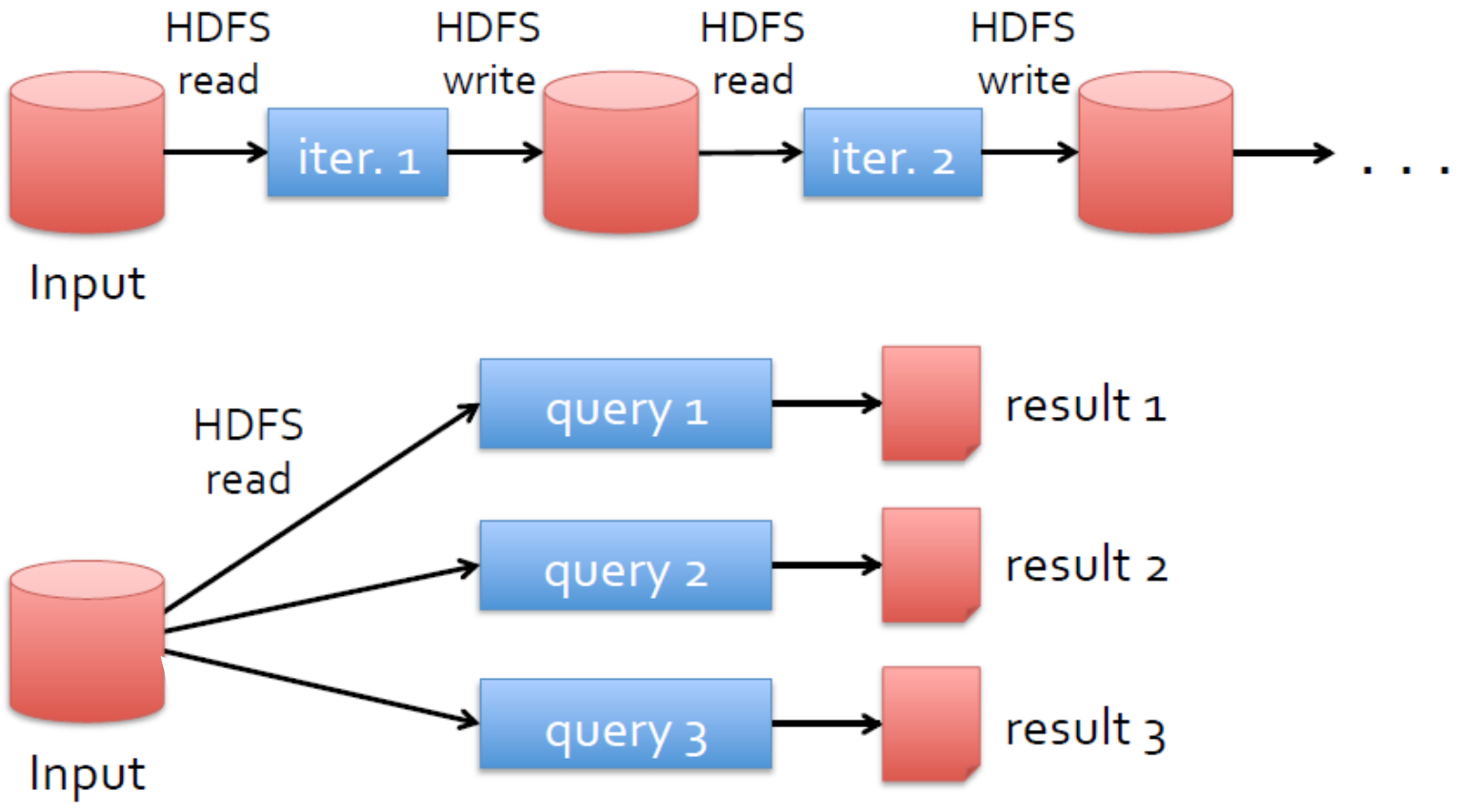


Spark Core

What is Spark Core

- Spark Core is the general execution engine for the Spark platform that other functionality is built atop:
  - In-memory computing capabilities deliver speed
  - Task scheduling
  - General execution model supports wide variety of use cases
  - Ease of development: native APIs in Java, Scala, Python, etc.

MapReduce



Slow due to replication and disk I/O,
but necessary for fault tolerance

MapReduce: The Good

- Built in fault tolerance
- Optimized IO path
- Scalable
- Developer focuses on Map/Reduce, not infrastructure
- Simple API

MapReduce: the Bad

- Op**timized for disk IO**
  - Doesn’t leverage memory well
  - Iterative algorithms go through disk IO path again and again

- **Complex apps and interactive queries** both need one thing that MapReduce lacks:
  - Efficient primitives for data sharing
  - Even basic things like join require extensive code

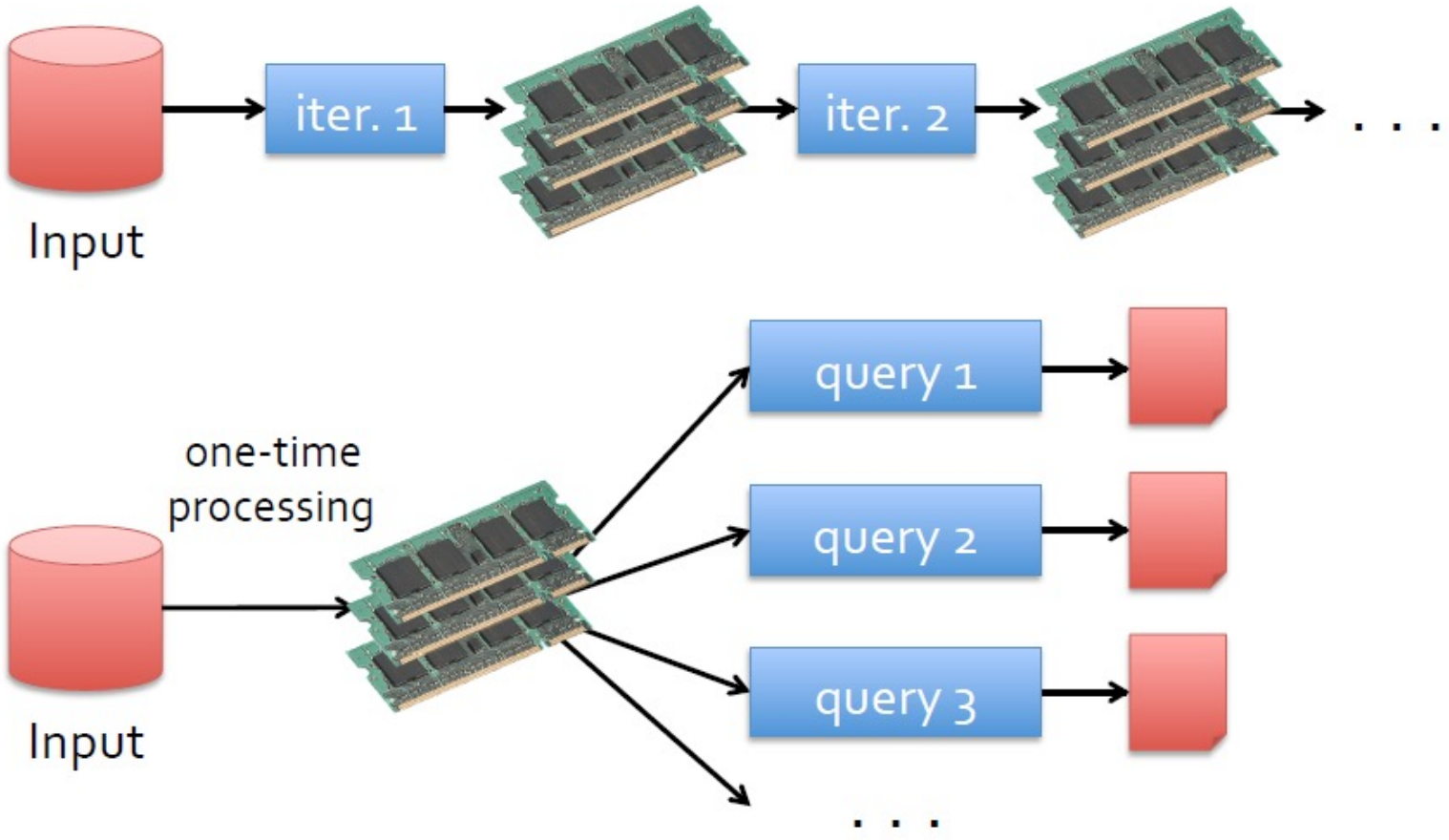
In MapReduce, the only way to share data across jobs is stable storage: slow!

Spark: Two Keypoints

- **Fast data sharing:**
  - In in-memory computation, the data is kept in random access memory (RAM) instead of some slow disk drives and is processed in parallel.
  - This has become popular because it reduces the cost of memory, as the cost of RAM has fallen over a period of time.

- **General DAGs (Directed Acyclic Graph):**
  - It is a strict generalization of MapReduce model.
  - DAG operations can do better global optimization than other systems like MapReduce.
  - The picture of DAG becomes clear in more complex jobs.

Spark: Sharing at Memory Speed

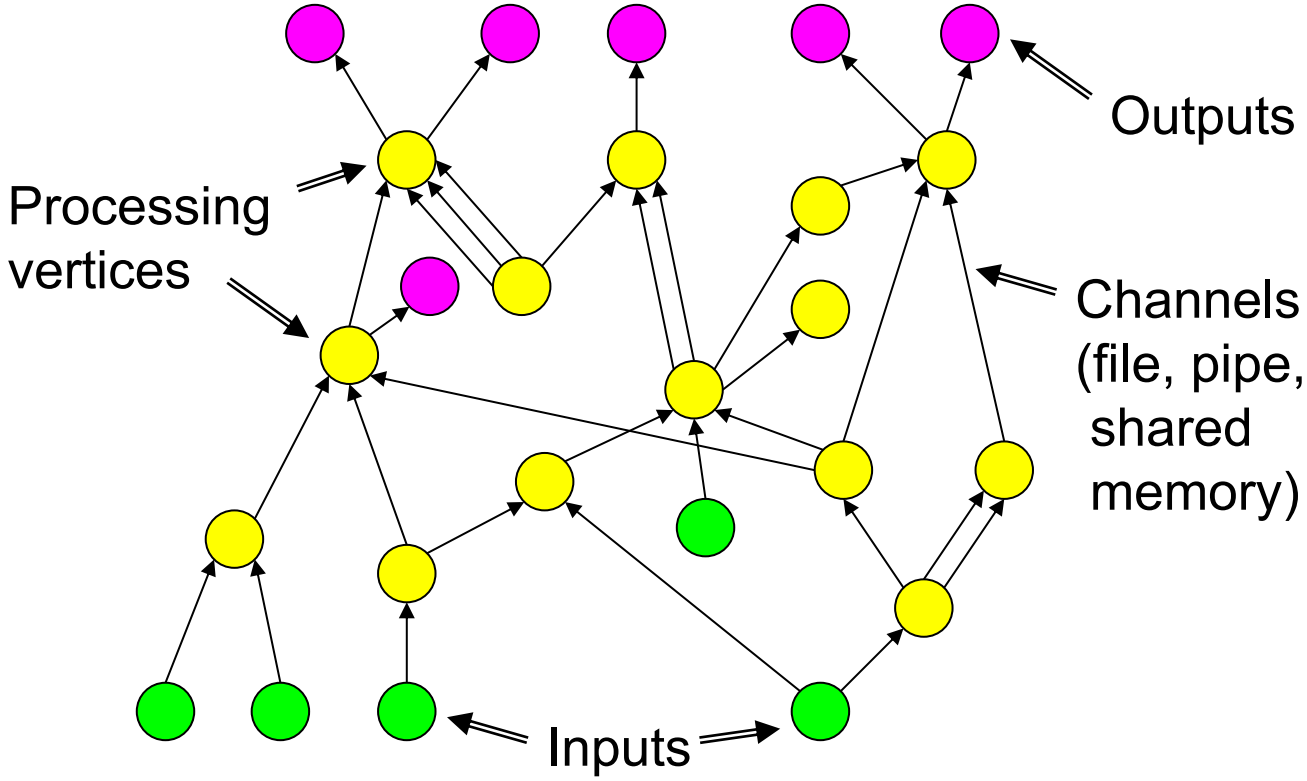


10-100× faster than network/disk, but how to get FT?

FT : Fault Tolerance

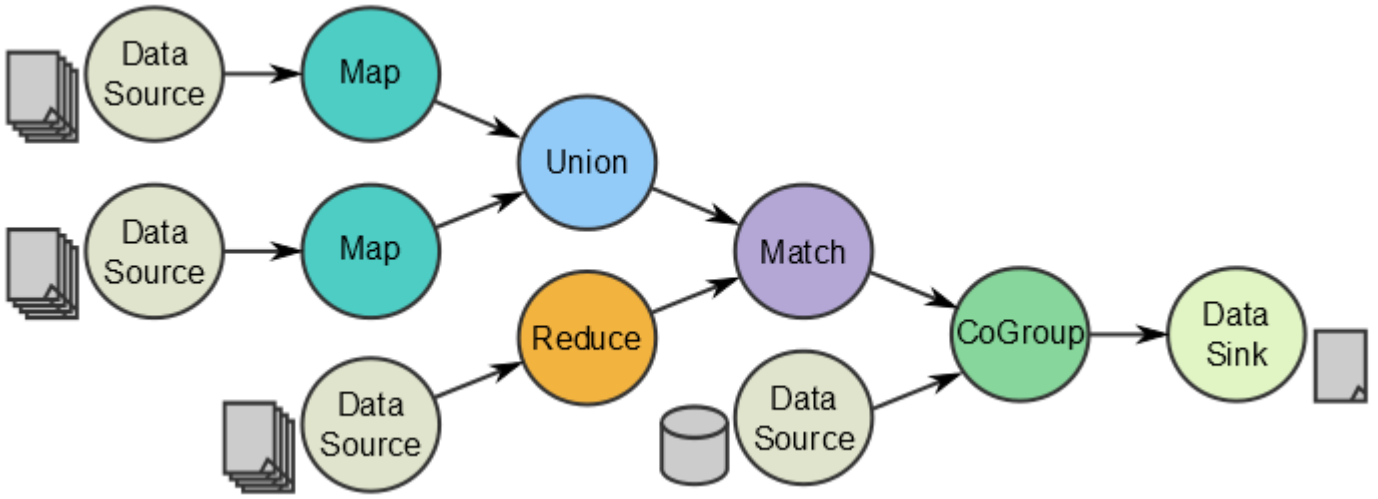
Directed Acyclic Graph (DAG)

- Many programs can be represented as a distributed dataflow graph
- The programmer may not have to know this



Spark: DAG Flow

- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a working set of data:
  - **Iterative** algorithms (many in machine learning)
  - **Interactive** data mining tools (R, Excel, Python)
- Spark makes working sets a first-class concept to efficiently support these apps



Spark Goal

- Provide distributed memory abstractions for clusters to support apps with working sets (fast data and general DAG)

- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability

Solution: augment data flow model with “resilient distributed datasets” (RDDs)

Resilient Distributed Dataset (RDD)

Programming Model

- **Resilient distributed datasets (RDDs)**
  - Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  - Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
  - Provide an interface based on coarse-grained operations (map, group-by, join, …)
  - Can be cached across parallel operations
- **Parallel operations on RDDs**
  - Reduce, collect, count, save, …
- Restricted shared variables
  - Accumulators, broadcast variables

Two main types of RDD

- **Parallelized collections**: take an existing Scala collection and run functions on it in parallel!

```
val count = sc.parallelize(1 to NUM_SAMPLES).filter { _ =>
    val x = math.random
    val y = math.random
    x*x + y*y < 1
}.count()
println(s"Pi is roughly ${4.0 * count / NUM_SAMPLES}")
```

- **Hadoop datasets**: run functions on each record of a file in HDFS or any other storage system supported by Hadoop

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

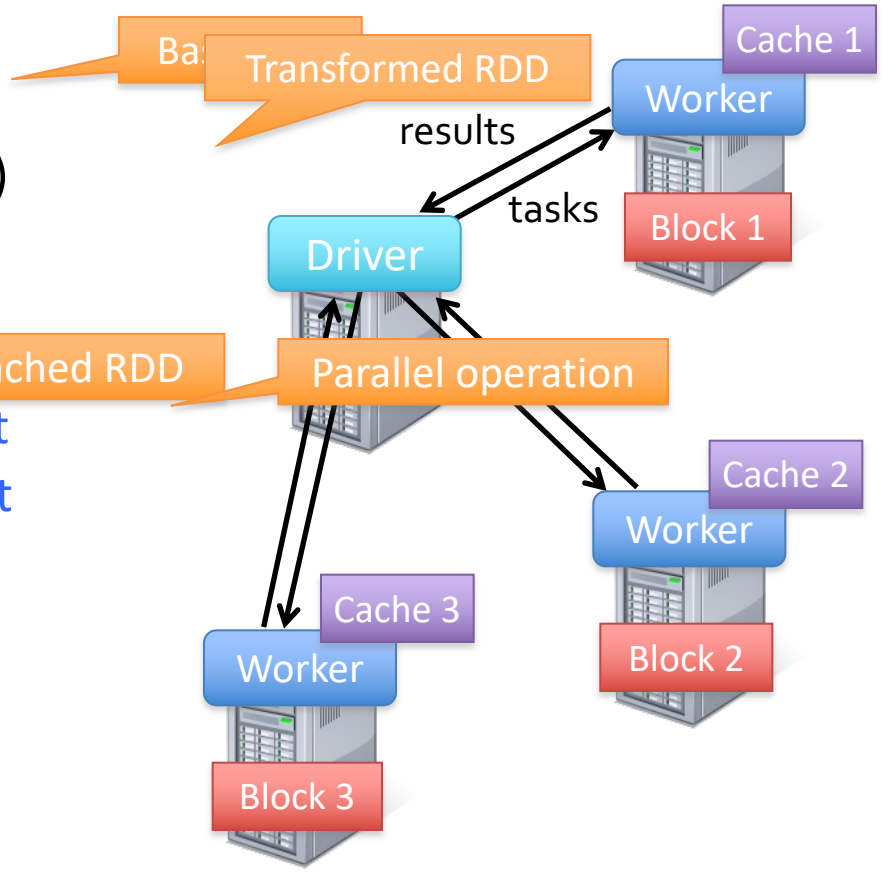
RDD

- Two types of operations on RDDs: transformations and actions
  - Transformations are lazy (not computed immediately)
  - The transformed RDD gets recomputed when an action is run on it

Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

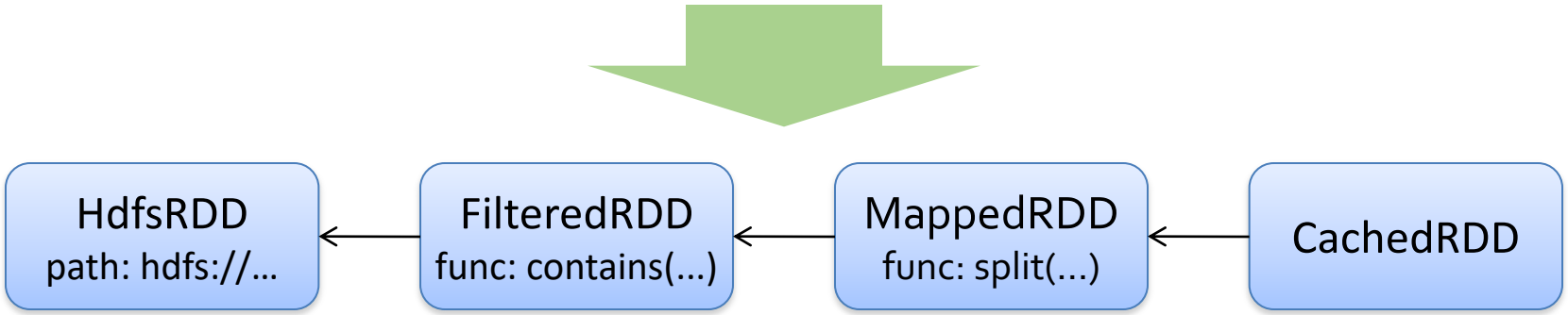
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(_.startsWith(“ERROR”))
messages = errors.map(_.split(‘\t’)(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains(“foo”)).count
cachedMsgs.filter(_.contains(“bar”)).count
. . .
```



RDD Fault Tolerance

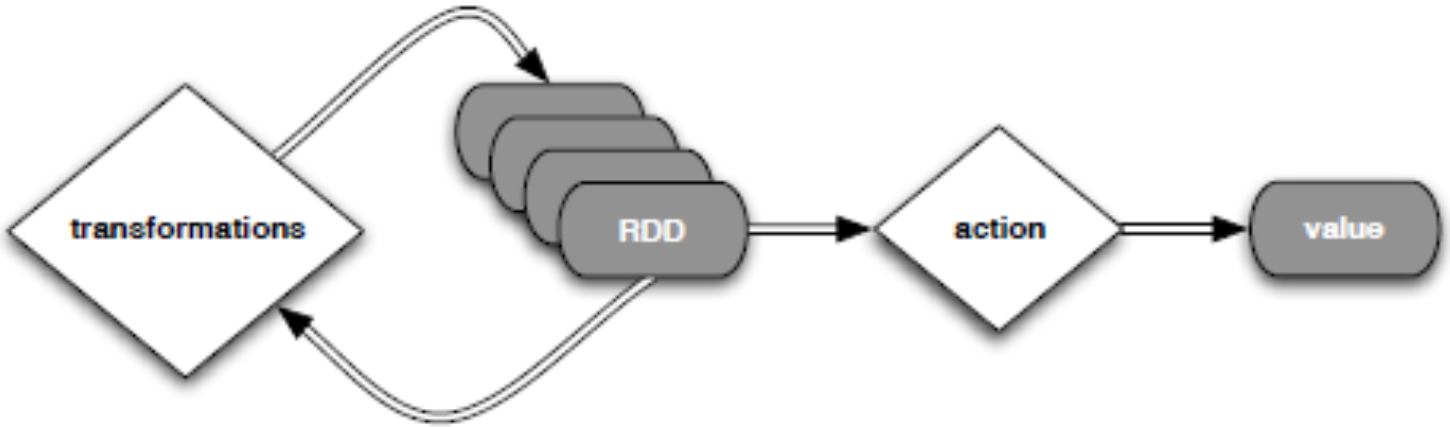
- RDDs maintain lineage information that can be used to reconstruct lost partitions
- Example:

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```



Re-Use Existing HDFS Location

- Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, HBase, etc.
- Spark supports text files, SequenceFiles, and any other Hadoop InputFormat, and can also take a directory or a glob (e.g. /data/201404*)



Programming Model

Spark Programming Interface

- Spark provides a programming interface in Scala (spark-shell)

- Provides:
  - Resilient distributed datasets (RDDs): RDD is an object in Scala
  - Operations on RDDs: transformations (build new RDDs), actions (compute and output results)
  - Control of each RDD's partitioning (layout across nodes) and persistence (storage in RAM, on disk, etc)

- The basic concepts are in: **spark.apache.org/**

SparkContext

- First thing that a Spark program does is create a SparkContext object, which tells Spark how to access a cluster
- In the shell for either Scala or Python, this is the “sc” variable, which is created automatically
- Other programs must use a constructor to instantiate a new SparkContext
- Then in turn SparkContext gets used to create other variables
- Scala:
  - scala> sc!
  - res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@6064cd08

SparkContext

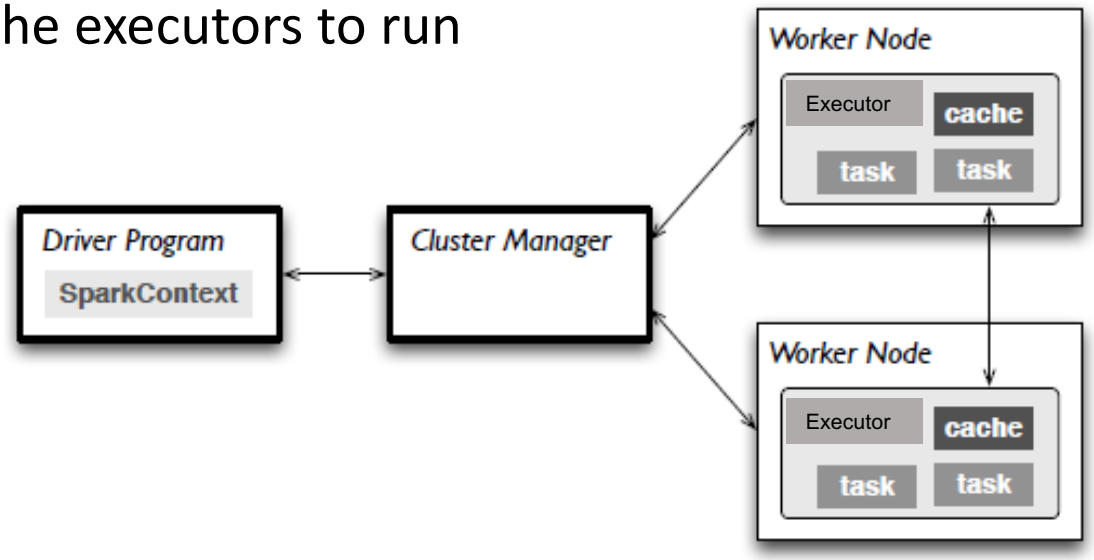
- The master parameter for a SparkContext determines which cluster to use

Master	Description
local	Run Spark locally with one worker thread (no parallelism)
local[K]	Run Spark locally with K worker threads
spark://HOST:PORT	Connect to a Spark standalone cluster; PORT depends on the configuration of the Spark cluster
yarn://HOST:PORT	Connect to a YARN cluster; PORT depends on the configuration of the YARN cluster

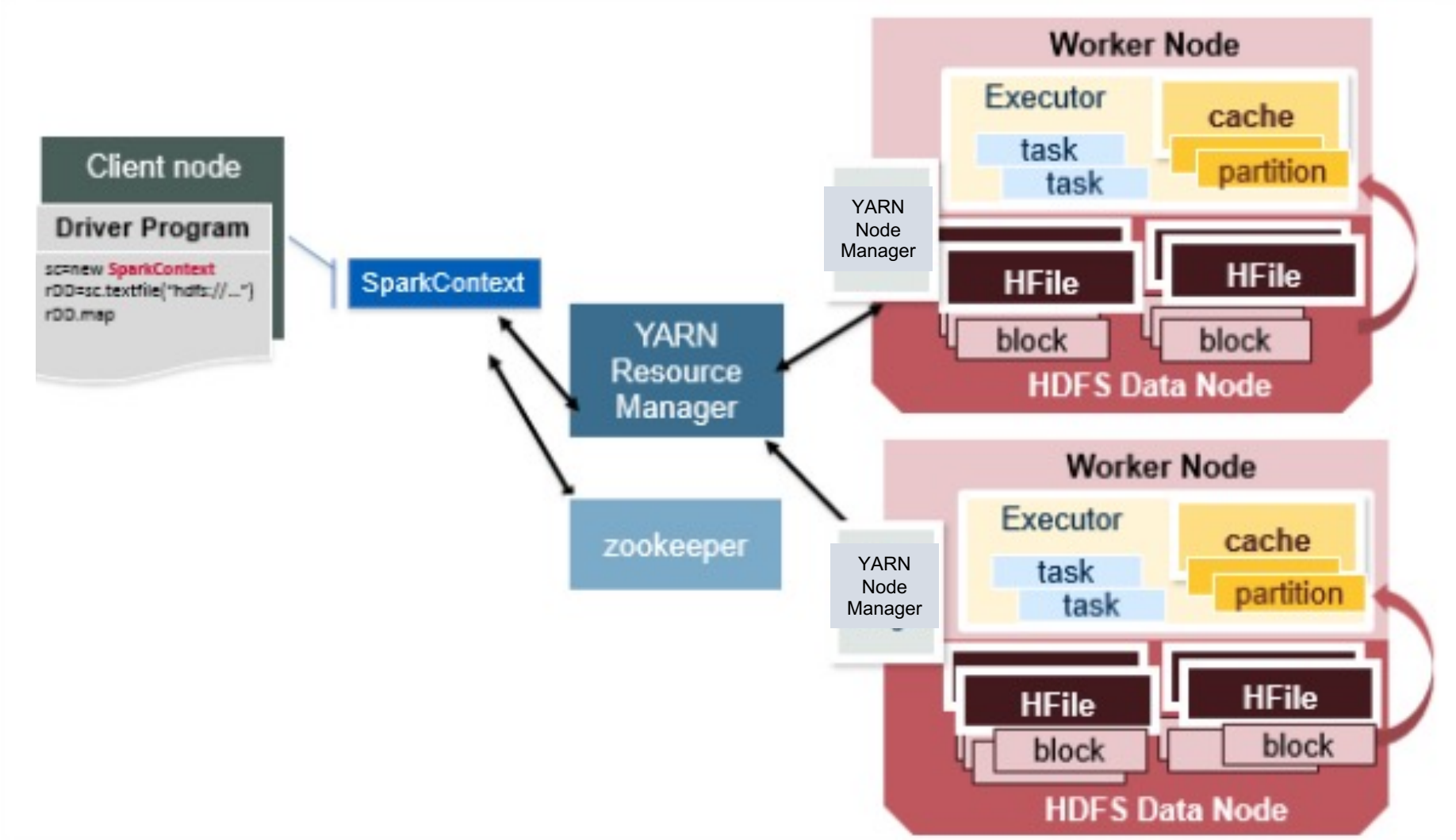
- NB: YARN can be replaced by Mesos. Apache Mesos is an open-source project to manage computer clusters (it is an alternative to YARN).

SparkContext

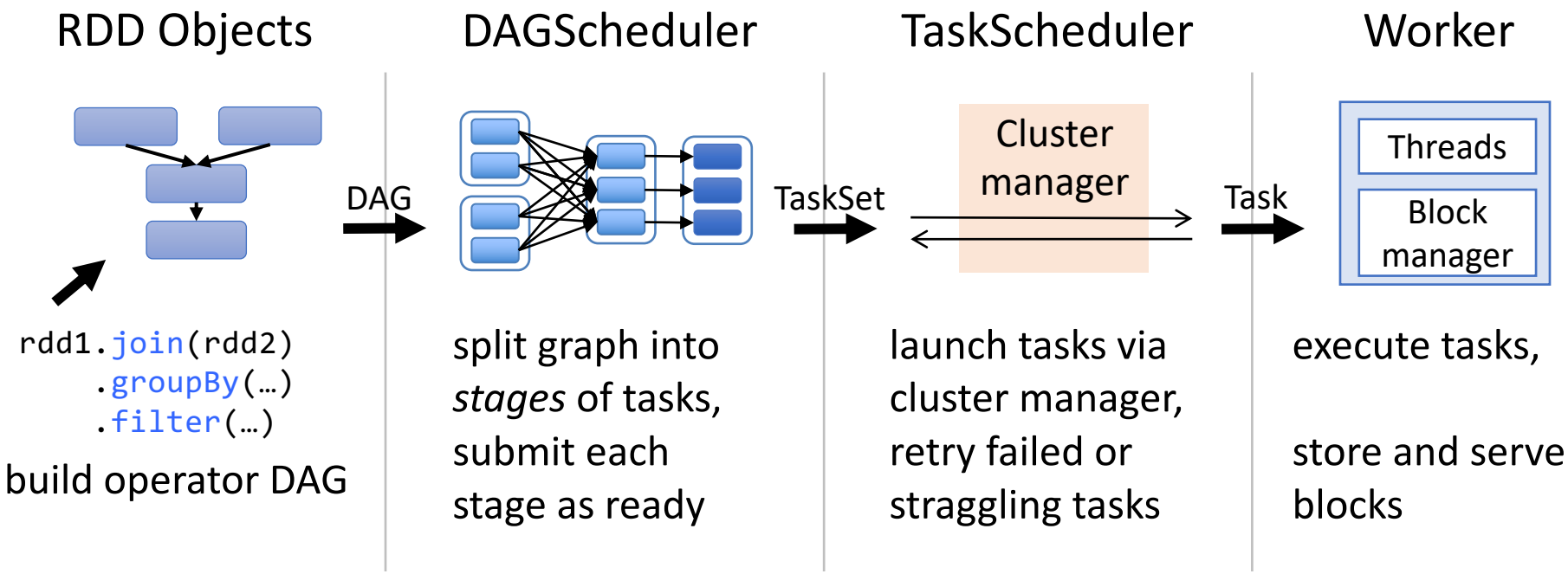
1. Driver program: the process running the main() function of the application and creating the SparkContext
2. Connects to a cluster manager which allocate resources across applications
3. Acquires executors on cluster nodes to run computations and store data
4. Sends app code to the executors
5. Sends tasks for the executors to run



How Spark Works - SparkContext



Job scheduling



Transformations

Transformations

- Transformations create a new dataset from an existing one

- All transformations in Spark are lazy: they do not compute their results right away – instead they remember the transformations applied to some base dataset
  - optimize the required calculations
  - recover from lost data partitions

Transformations

transformation	*description*
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

Transformations

transformation	*description*
groupByKey([*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset*)	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

Actions

Actions

- Return a value after running a computation
- All actions in Spark are computed immediately

Actions

action	description
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n*)	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed*)	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Actions

action	description
saveAsTextFile(*path*)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
saveAsSequenceFile(*path*)	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
countByKey()	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
foreach(*func*)	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Task Scheduling

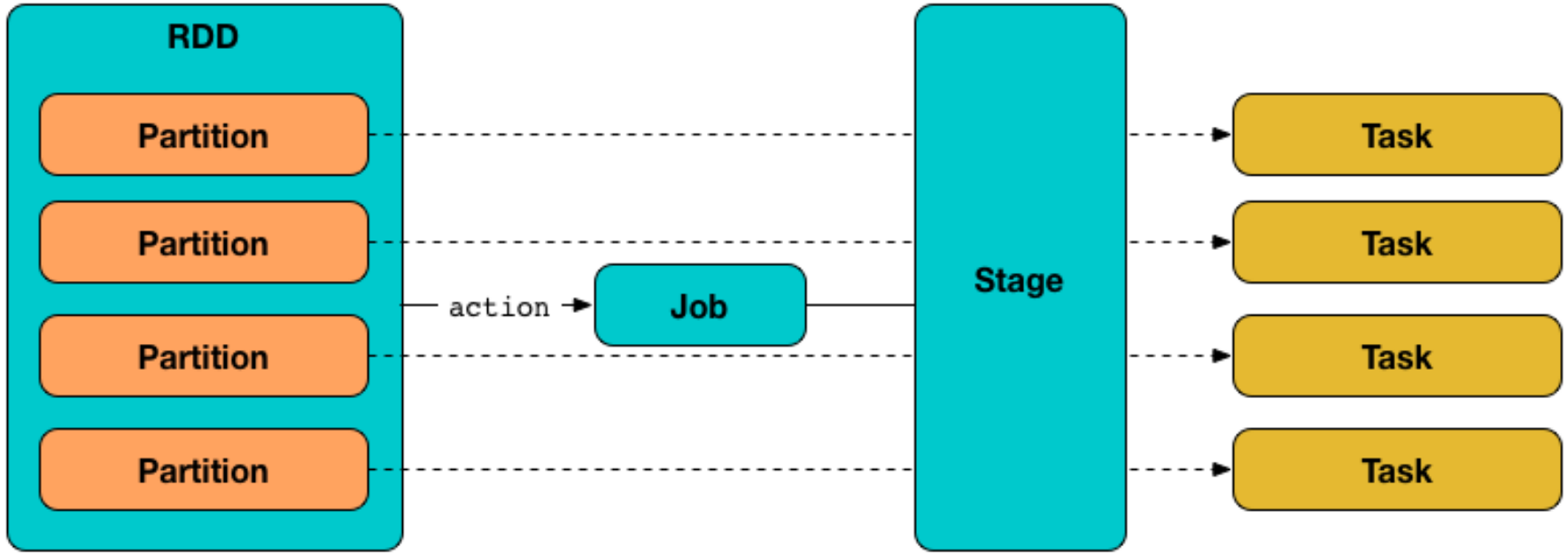
Logical DAG

- User submits a spark application to the Apache Spark.
- Driver is the module that takes in the application from Spark side.
- Driver identifies transformations and actions present in the spark application. These identifications are the **tasks**.
- Based on the flow of program, these tasks are arranged in a graph like structure with directed flow of execution from task to task forming no loops in the graph (also called DAG).
- DAG is pure logical.

https://books.japila.pl/apache-spark-internals/

Task and Stage within the Logical DAG

- Task is an abstraction of the smallest individual units of execution that can be executed (to compute an RDD partition).
- This logical DAG is converted to Physical Execution Plan. Physical Execution Plan contains stages.



Physical Execution Plan (PEP)

- Some of the subsequent tasks like map, filter in DAG could be combined together in a single stage.
- Based on the nature of transformations, Driver sets **stage boundaries**.
- There are two transformations, namely narrow transformations and wide transformations, that can be applied on RDD.
  - **Narrow transformations**: Transformations like Map and Filter that does not require the data to be shuffled across the partitions. This leads to a **pipeline of tasks** performed in-memory.
  - **Wide transformations**: Transformations like ReduceByKey that does require the data to be shuffled across the partitions. Transformation that requires data shuffling between partitions, i.e., **a wide transformation results in stage boundary (new stage is required)**.
- DAG Scheduler creates a PEP from the logical DAG.
- PEP contains tasks and are bundled to be sent to the cluster.

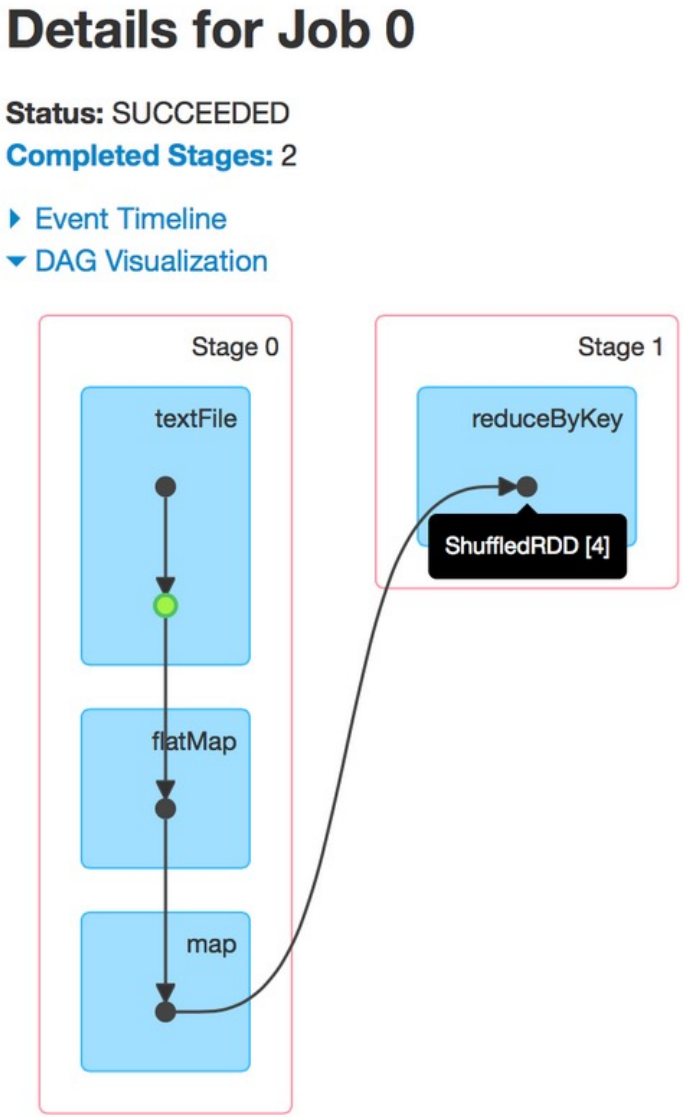
Code Example: Word Count

- Scala version:

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
```

- Python version:

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
```



Persistence

Persistence

- Spark can persist (or cache) a dataset in memory across operations.

- Caching RDDs in Spark:
  - It is one mechanism to speed up applications that access the same RDD multiple times.
  - An RDD that is not cached, nor checkpointed, is re-evaluated again each time an action is invoked on that RDD.

- There are two function calls for caching an RDD: cache() and persist (level: StorageLevel).
  - cache() will cache the RDD into memory,
  - persist(level) can cache in memory, on disk, or off-heap memory according to the caching strategy specified by level.
    - Writing to disk is called checkpoint.
    - persist() without an argument is equivalent with cache().
    - Freeing up space from the Storage memory is performed by unpersist().

- The cache is fault-tolerant: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

Persistence

transformation	*description*
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

Persistence

```
val f = sc.textFile("README.md")
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).collect.foreach(println)
```

Shared Variables

Broadcast variables

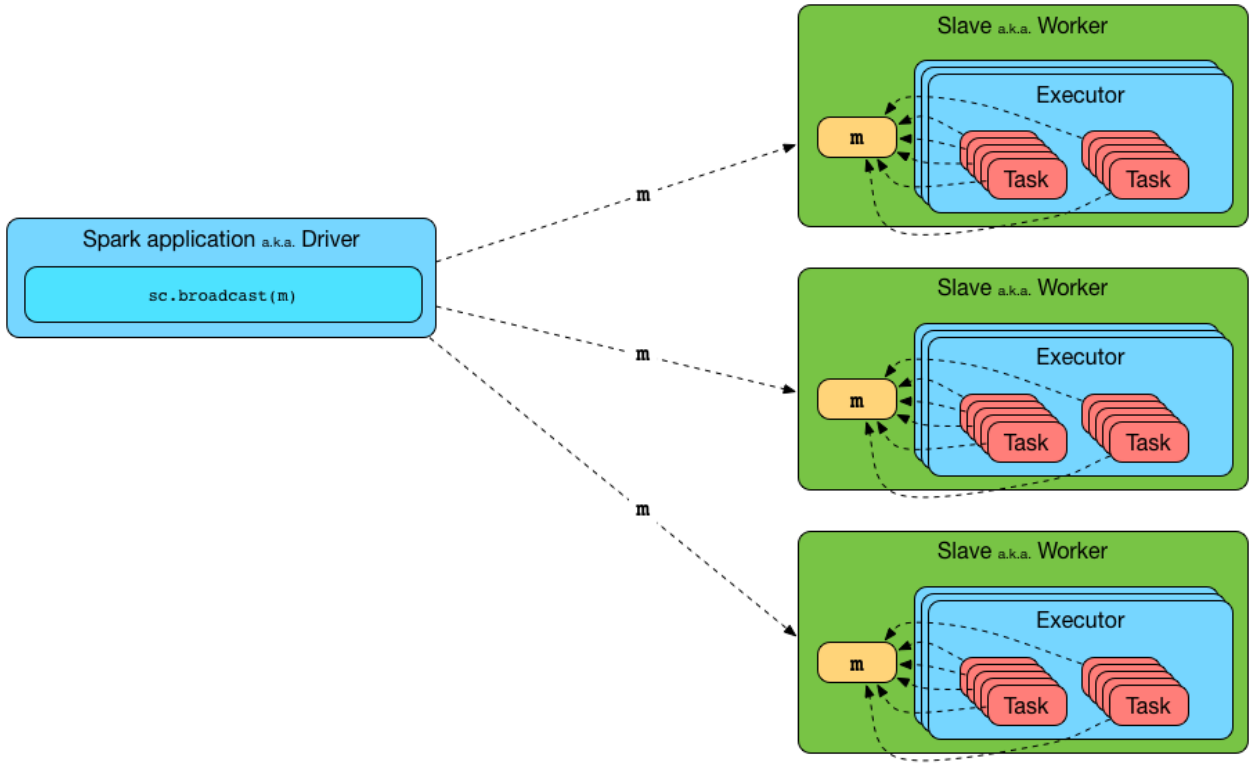
- Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

- For example, to give every node a copy of a large input dataset efficiently
  - Imagine you need to lookup a large table of zip codes.
  - It is neither feasible to send the large lookup table every time to the executors, nor can we query the database every time.
  - The solution should be to convert this lookup table to a broadcast variables and Spark will cache it in every executor for future reference.

- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Broadcast Variables

```
val m = Array(1, 2, 3)
val broadcastVar = sc.broadcast(m)
broadcastVar.value
```



Accumulator

- Accumulators are variables that can only be “added” to through an associative operation
- Used to implement counters and sums, efficiently in parallel
- Spark natively supports accumulators of numeric value types and programmers can extend for new types
- **Only the driver program** can read an accumulator’s value, not the tasks

Why use Spark Accumulators?

```
var blankLines: Int = 0
sc.textFile(“logfile” ). foreach {
        line => if (line.length() == 0) blankLines +=1
}
println(s“Blank Lines=$blankLines”)
```

- The problem with the above code is that when the driver prints the variable blankLines its value will be zero.
- When Spark ships this code to every executor the variables become local to that executor and its updated value is not relayed back to the driver.
- To avoid this problem, we need to make blankLines an accumulator such that all the updates to this variable in every executor is relayed back to the driver.

```
var blankLines  = sc.doubleAccumulator("0")
sc.textFile(“logfile”). foreach {
        line => if (line.length() == 0) blankLines.add(1)
}
println(s“Blank Lines=$blankLines”) // driver side
```

Conclusion

Conclusion

- “Big data” is moving beyond one-passbatch jobs, to low-latency apps that need datasharing
- RDDs offer a simple and efficient programming model for a broad range of applications
- Many “big data” apps need to work in real time :
    - Site statistics, spam filtering, intrusion detection, etc.
- For more details about the Spark API:

https://spark.apache.org

- Also challenging in streaming systems.