Spark

Part 1:

Overview,

Programming with Resilient Distributed Datasets

Dario Colazzo

Credits: Amir H. Payberah

Motivation

- MapReduce greatly simplified big data analysis on large, unreliable clusters.
- But as soon as it got popular, users wanted more:
  - Iterative jobs, e.g., machine learning algorithms
  - Interactive analytics

Motivation

- Both iterative and interactive queries need one thing that MapReduce lacks

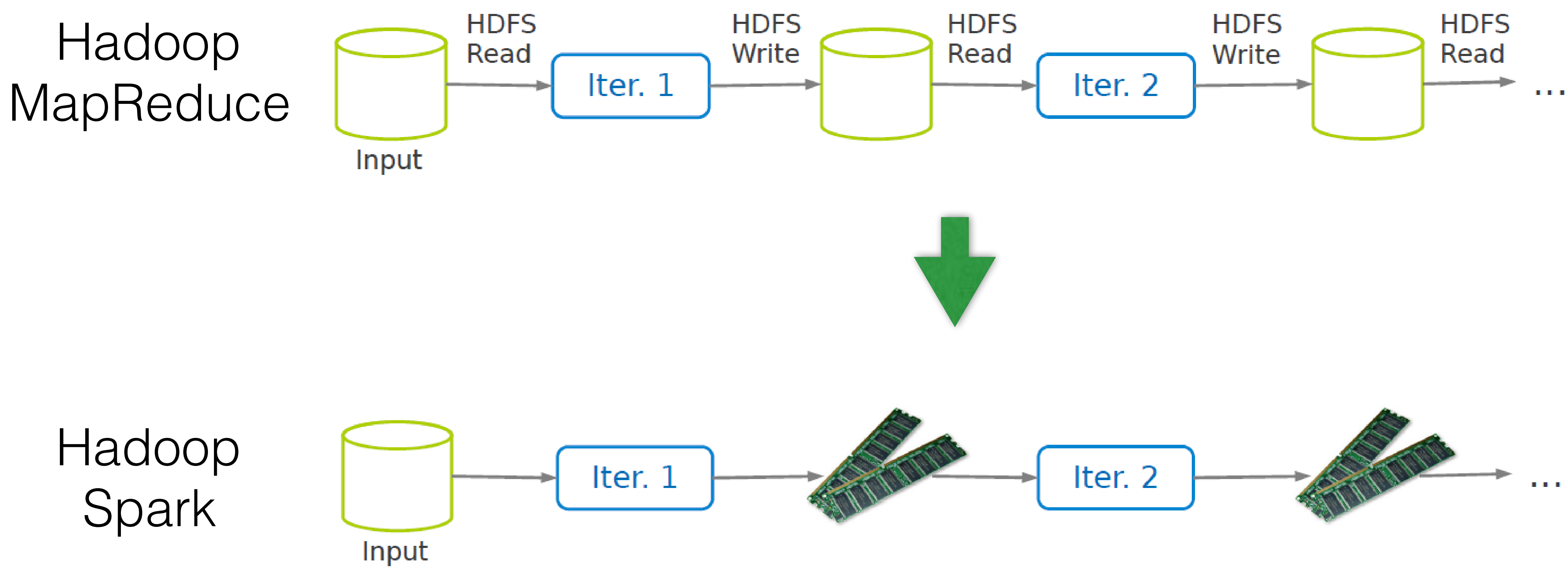
Efficient primitives for data sharing.

- In MapReduce, the only way to share data across processing step is stable storage (disk)

- Replication also makes the system slow, but it is necessary for fault tolerance.

Solution

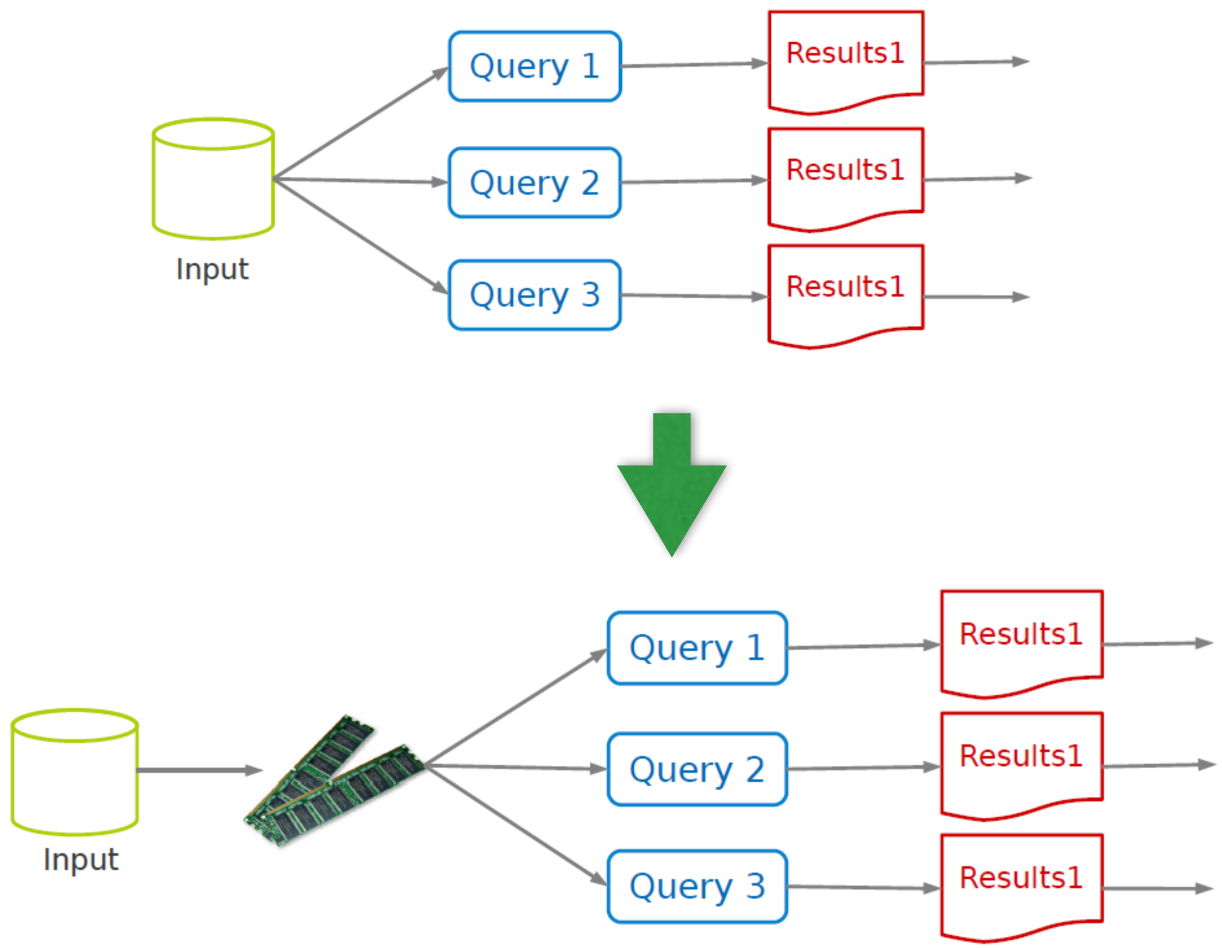
In memory data processing and sharing



Sharing

Hadoop
MapReduce

Hadoop
Spark

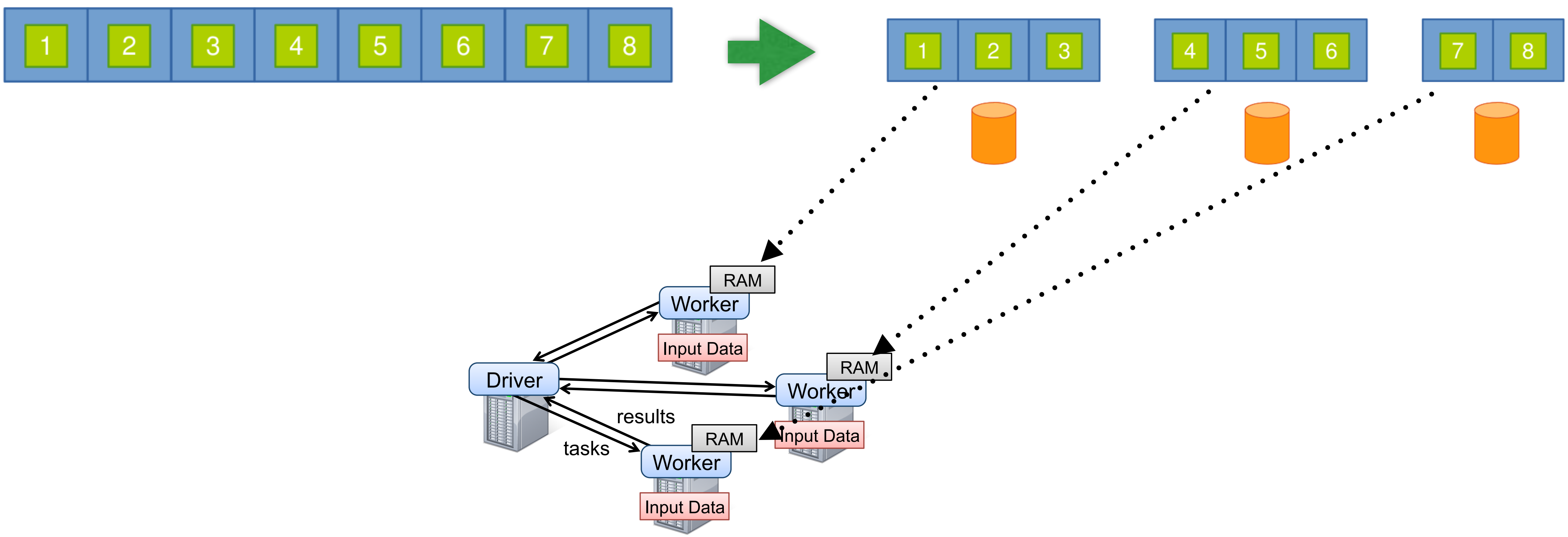


Challenge

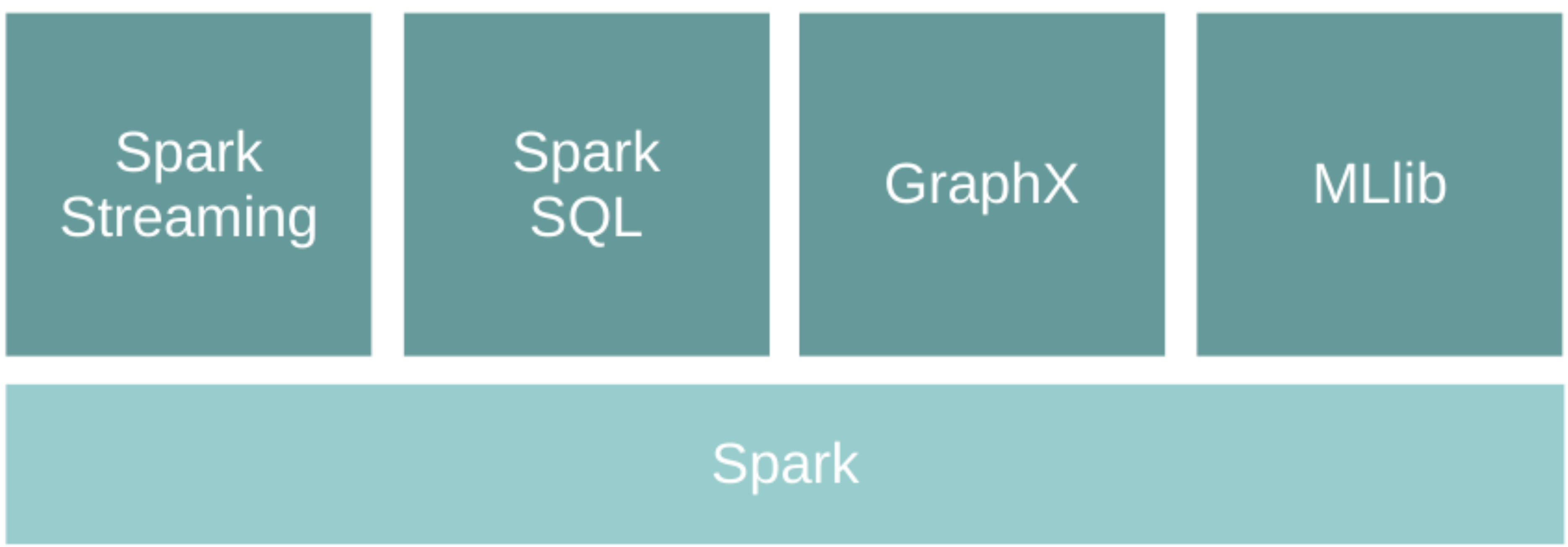
- How to design a distributed memory abstraction that is both fault tolerant and efficient?
- Solution: Resilient Distributed Datasets (RDD)
  - A distributed main-memory abstraction.
  - Immutable collections of objects spread across a cluster.
  - Lineage among RDDs to enable their re-evaluation in case of cluster node failures

Resilient Distributed Datasets (RDDs)

- An RDD is a collection which divided into a number of partitions, which can be independently processed.

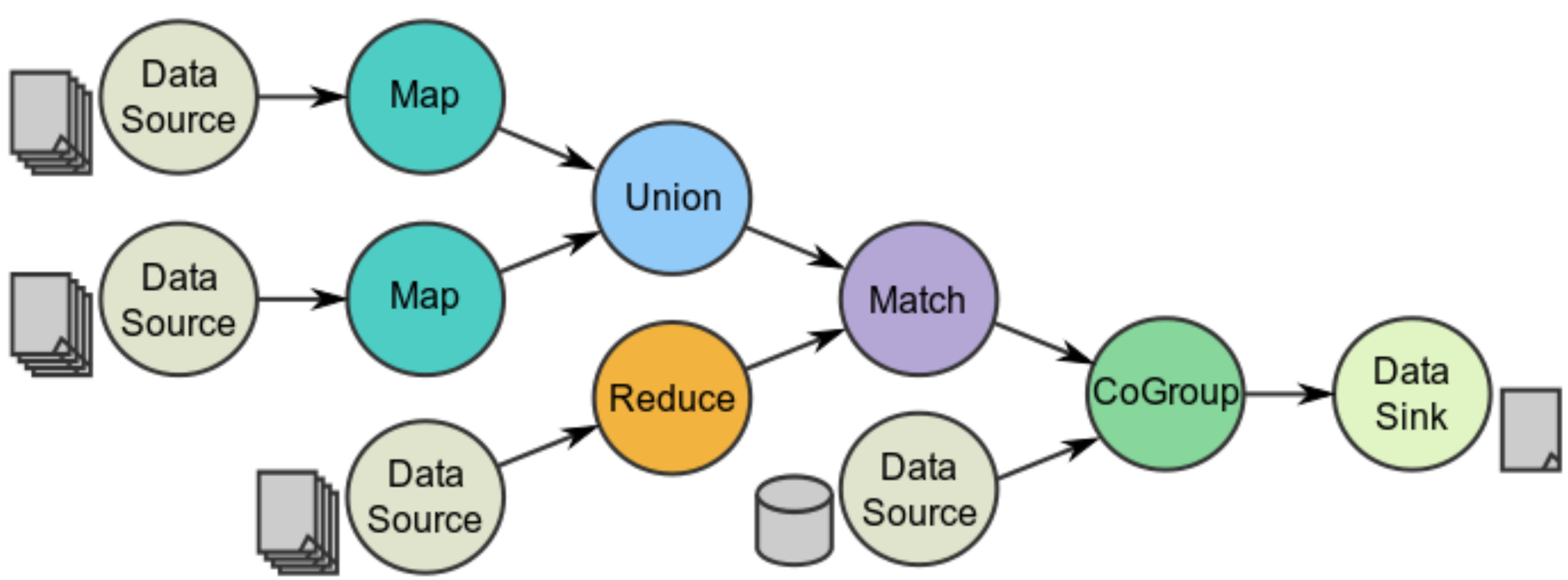


Spark Processing engine



Programming model

- A data flow is composed of any number of data sources and data sinks by connecting their inputs and outputs by means of data operators.

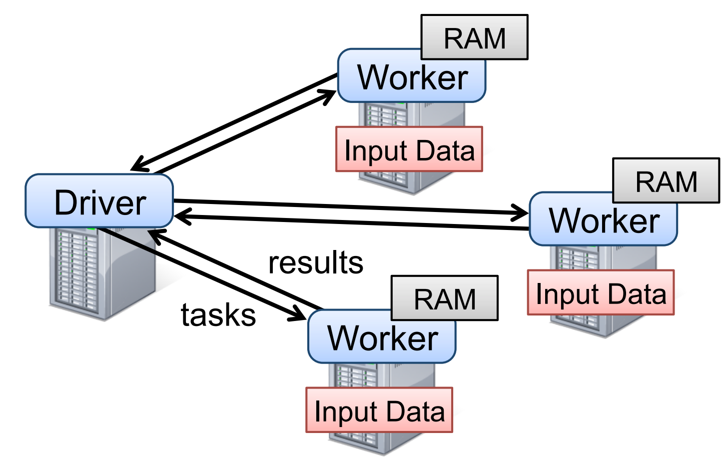


Programming model

- Based on parallelizable operators.
- Parallelizable operators are higher-order functions that execute user-defined functions in parallel, on each partition of an RDD.
- There are two types of RDD operators : transformations and actions.

Programming model

- Transformations : lazy operators that create new RDDs.
- Actions : lunch a computation and return a value to the program driver or write data to the external storage



- Implemented in Scala:
    - a strongly and statically typed functional-OO language
    - compiled and run over the JVM
    - designed at EPFL (Switzerland).
- Java and [Python] can be used too for Spark programming.

Example (1/2)

- Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause.

- Here is Spark code in Scala (but we will switch soon to Python)

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

- Actions can be used to count errors:

```
errors.count()
```

- Or counting errors mentioning MySQL:

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()
```

Borrowed from *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia et al, NSDI 2012.*

Example (1/2)

- Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause.

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

lines is not loaded in memory only **errors** is (simple static analysis)

- Actions can be used to count errors:

```
errors.count()
```

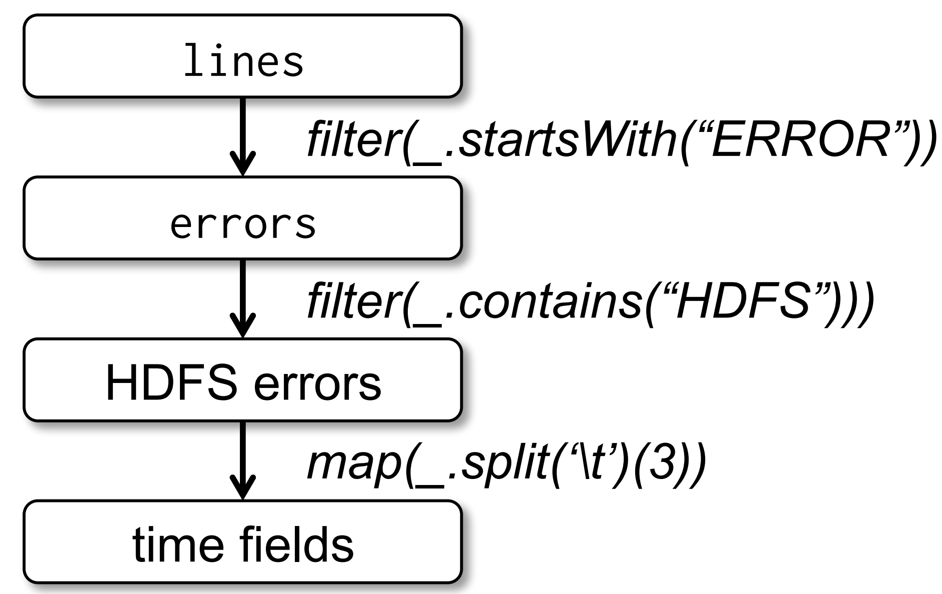
lazy evaluation: **errors** is actually calculated and put in memory when the **count()** action is evaluated

- Or counting errors mentioning MySQL:

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()
```

Fault tolerance via *lineage*

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```



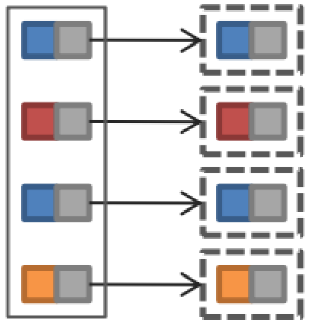
the lineage graph enables RDD re-evaluation in case of failure

RDD transformations and actions

Transformations	$map(f: \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f: \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f: \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction: \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f: (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f: \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c: \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p: \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f: (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k: \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path: \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

RDD transformations : Map

- All pairs are independently processed

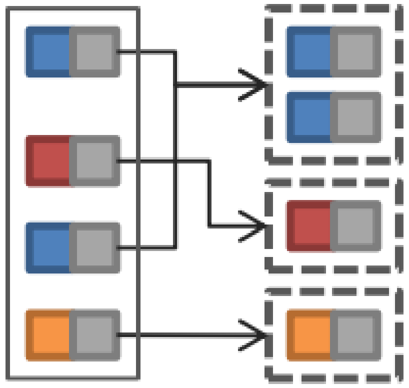


```
# passing each RDD element trough a function
nums = sc.parallelize([1,2,3])
squares = nums.map(lambda x: x * x)

# selecting elements making a boolenba function returning true
even = squares.filter(lambda x : x % 2 ==0)

# map + flattening
m = nums.map(lambda x: range(x))
# [[0], [0, 1], [0, 1, 2]]
fm = nums.flatMap(lambda x: range(x))
# [0, 0, 1, 0, 1, 2]
```

RDD transformations : Reduce

- Pairs with identical key are grouped

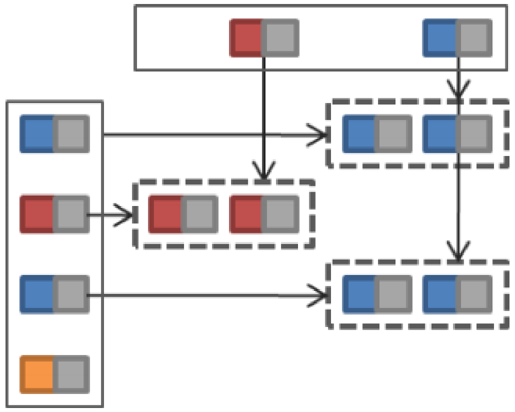
- Each group is independently processed for aggregation



```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2),  ("dog", 3) ])

pets.reduceByKey(lambda x, y : x +y)
# [(‘dog', 4), ('cat', 3)]

pets.groupByKey()
pets.groupByKey().map(lambda x : (x[0], list(x[1])))
# [('dog', [1,3]), ('cat', [1, 2])]
```

RDD transformations : Join

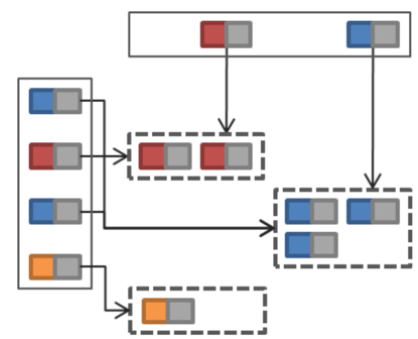
- Equi-join on the key



```
visits = sc.parallelize( [("h", “1.2.3.4"), ("a", "3.4.5.6"), ("h", “1.3.3.1")] )

pageNames = sc.parallelize( [("h", “Home"), ("a", “About")] )

visits.join(pageNames)

# [('a', ('3.4.5.6', 'About')), ('h', ('1.2.3.4', Home')),
    ('h', ('1.3.3.1', 'Home'))]
```

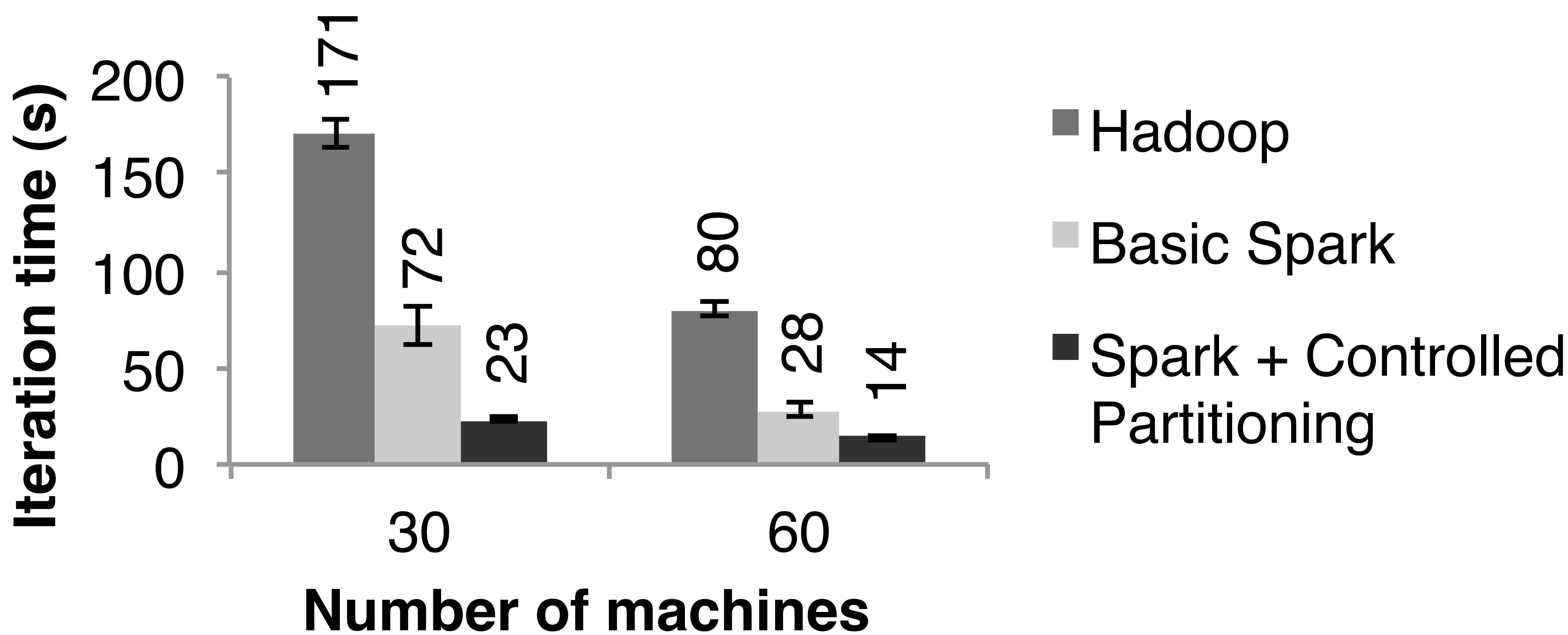
RDD transformations : CoGroup

- Groups each input on key
- Groups with identical keys are processed together



```
visits = sc.parallelize([("h", “1.2.3.4"), ("a", "3.4.5.6"), ("h", "1.3.3.1")] )

pageNames = sc.parallelize([("h", "Home"), ("a", "About"), ("o", "Other")])

visits.cogroup(pageNames)

visits.cogroup(pageNames).map(lambda x :(x[0], ( list(x[1][0]), list(x[1][1]))))

# [('a', (['3.4.5.6'], ['About'])), ('h', (['1.2.3.4', '1.3.3.1'], ['Home'])),
('o', ([], ['Other']))]
```

Some experiments on PageRank



Borrowed from *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia et al, NSDI 2012.*

Remarks

- MapReduce makes important abstraction step that greatly helps rapid development of efficient and robust Big Data data flows.
- But:
  - we still need some ‘hacking’ to ensure good performances
  - problems with iterative analyses
  - MapReduce programming is not easy
- Spark overcomes these limitations in a large extent, at the cost of more RAM needed.
- Makes a one more step towards ‘The data center is the computer’ scenario.