Spark

Fast, Interactive, Language-Integrated Cluster Computing

Project Goals

Extend the MapReduce model to better support two common classes of analytics apps:

>> Iterative algorithms (machine learning, graph)

>> Interactive data mining

Enhance programmability:

>> Integrate into Scala programming language

>> Allow interactive use from Scala interpreter

Motivation

- 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

Response: *specialized* frameworks for some of these apps (e.g. Pregel for graph processing)

Motivation

- Complex apps and interactive queries both need one thing that MapReduce lacks:
    - Efficient primitives for **data sharing**

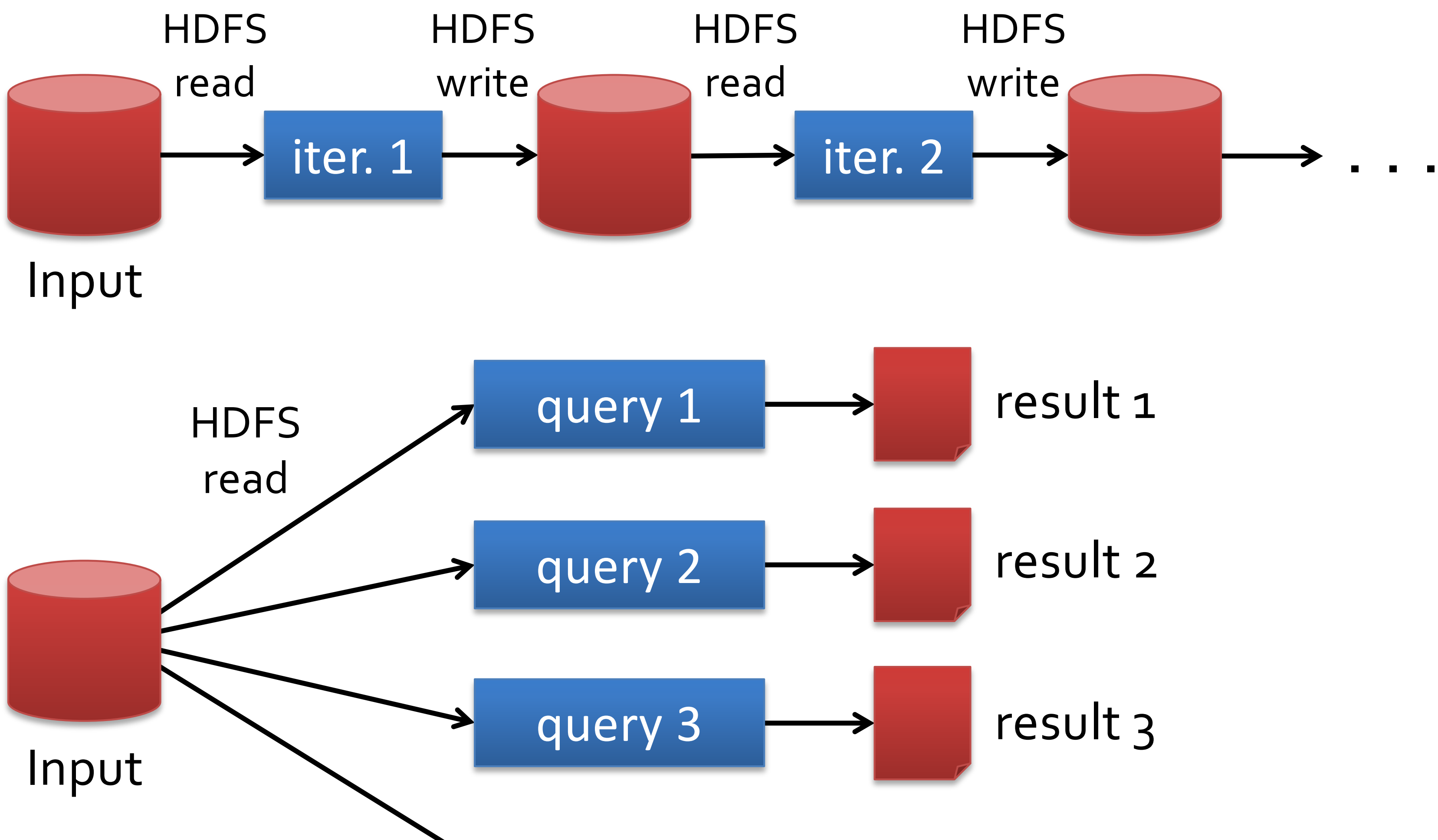
In MapReduce, the only way to share data across jobs is stable storage → slow!

Memory vs Disk

If Memory = Minute
Network = Weeks
Flash = Months
Disk = Decades

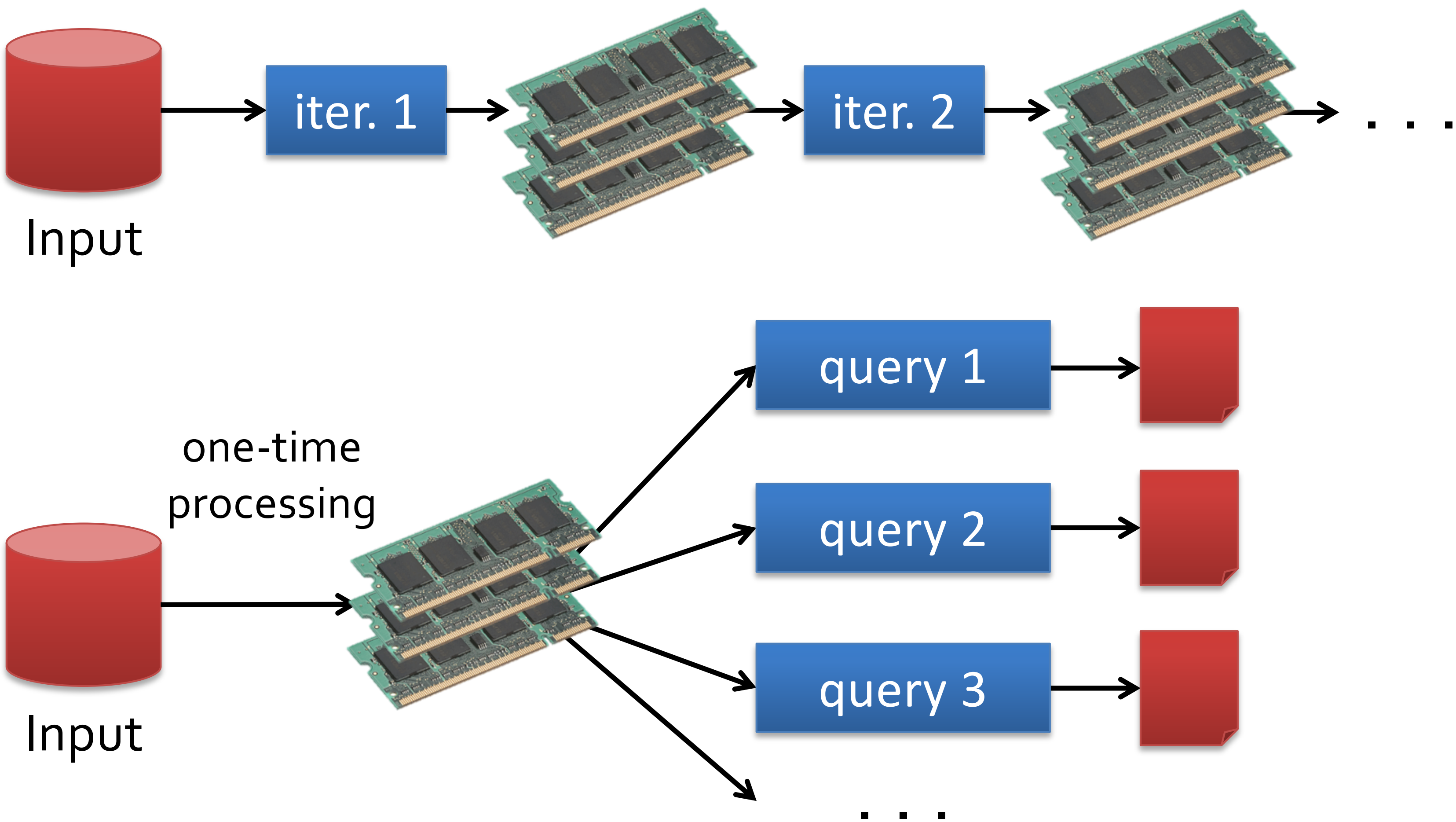
Source: http://blog.infinio.com/relative-speeds-from-ram-to-flash-to-d
http://blog.scoutapp.com/articles/2011/02/08/how-much-slower-is-di

Examples



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

Goal: In-Memory Data Sharing



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

Challenge

How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

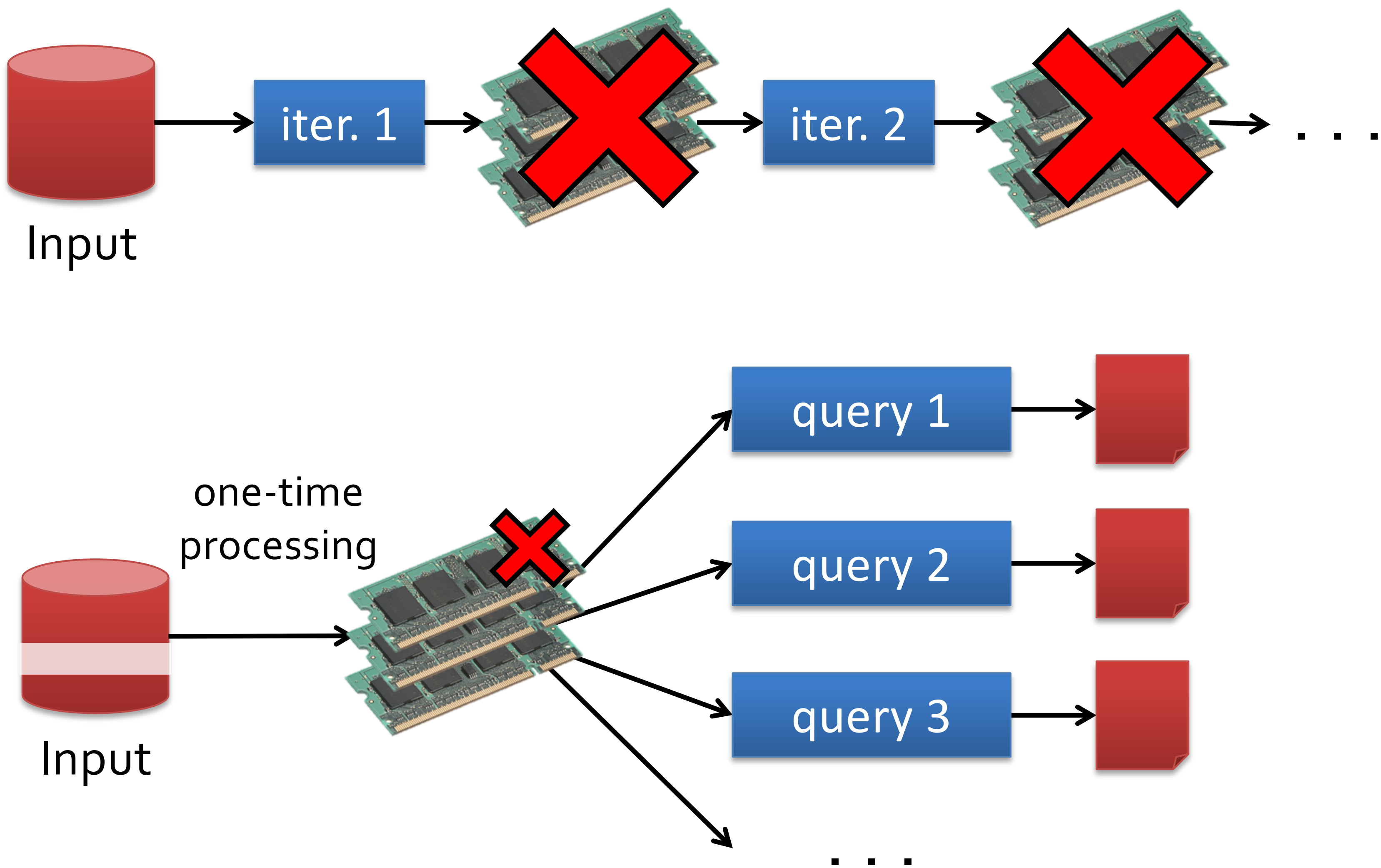
Challenge

- Existing storage abstractions have interfaces based on *fine-grained* updates to mutable state
  - RAMCloud, databases, distributed mem, Piccolo
- Requires replicating data or logs across nodes for fault tolerance
  - Costly for data-intensive apps
  - 10-100x slower than memory write

Solution: Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)
- Efficient fault recovery using *lineage*
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails

RDD Recovery



Generality of RDDs

- Despite their restrictions, RDDs can express surprisingly many parallel algorithms
  - These naturally *apply the same operation to multiple items*
- Unify many current programming models
  - *Data flow models:* MapReduce, Dryad, SQL, …
  - *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …
- Support *new apps* that these models don't

Introduction to Spark programming

Key things to know about Spark

- What is the entry point for using Spark functionality?
- SparkContext (sc) object
- It represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
- Almost every line of code starts with sc object.

RDDs

- What are RDD?
  - fault-tolerant in-memory collection of elements that can be operated on in parallel.
- 2 ways of creating RDDs?
- parallelize an existing collection
- reference an existing dataset in external storage system e.g. HDFS, Hbase, etc

Parallelize RDD

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

Create RDD from HDFS file

```
val distFile = sc.textFile("data.txt")
```

By default, Spark creates one partition for each block of the file (blocks being 64MB by default in HDFS).

What other files are supported by Spark

- wholeTextFiles -> entire directory can be read
- SequenceFiles -> maps to Hadoop's sequenceFiles, which are highly efficient binary representation of Hadoop data.
- sc.hadoopRDD -> takes arbitrary inputformat class object and converts it into RDD

Operations

RDDs support two types of operations:

1. Transformations -> which create a new dataset from existing one.

e.g. map

2. Action -> returns a value to the driver program after running a computation on the dataset.

e.g. reduce

Operations

All transformations in Spark are lazy

⇒ They do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file).

⇒ The transformations are only computed when an action requires a result to be returned to the driver program.

Illustration

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

=> When is the dataset loaded in memory?

Illustration

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

⇒When is the dataset loaded in memory?

⇒First line:
This dataset is not loaded in memory or otherwise acted on: lines is merely a pointer to the file.

Illustration

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

⇒When is the dataset loaded in memory?

⇒Second line:

The second line defines lineLengths as the result of a map transformation. Again, lineLengths is not immediately computed, due to laziness.

Illustration

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

⇒When is the dataset loaded in memory?

⇒Third line:

We run reduce, which is an action. At this point Spark breaks the computation into tasks to run on separate machines, and each machine runs both its part of the map and a local reduction,

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Transformations

Transformations	Meaning
map(func)	Return a new distributed dataset formed by passing each element of the source through a function *func*
flatMap(func)	Return a new datasets formed by selecting those elements of the source on which *func* returns true
union(otherDateset)	Return a new dataset that contains the union of the elements in the source dataset and the argument
…	…

Actions

Actions	Meaning
reduce(func)	Aggregate the elements of the dataset using a function *func*
collect()	Return all the elements of the dataset as an array at the driver program
count()	Return the number of elements in dataset
first()	Return the first element of the dataset
saveAsTextFile(path) *.....*	Write the elements of the dataset as text file (or set of text file) in a given dir in the local file system, HDFS or any other Hadoop-supported file system ......

Differences between map and flatMap – converting String to Int

The following examples show more differences between `map` and `flatMap` for a simple `String` to `Int` conversion example. Given this `toInt` method:

```
def toInt(s: String): Option[Int] = {
    try {
        Some(Integer.parseInt(s.trim))
    } catch {
        // catch Exception to catch null 's'
        case e: Exception => None
    }
}
```

Here are a few examples to show how `map` and `flatMap` work on a simple list of strings that you want to convert to Int:

```
scala> val strings = Seq("1", "2", "foo", "3", "bar")
strings: Seq[java.lang.String] = List(1, 2, foo, 3, bar)

scala> strings.map(toInt)
res0: Seq[Option[Int]] = List(Some(1), Some(2), None, Some(3), None)

scala> strings.flatMap(toInt)
res1: Seq[Int] = List(1, 2, 3)

scala> strings.flatMap(toInt).sum
res2: Int = 6
```

MapReduce with Spark RDD

val textFile = sc.textFile("README.md")

textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)

*NOTE: Input to map is a closure function closure.

Advanced Spark programming

Key Value Pairs

- Many operations involve working on pair RDDs and per key operations.

- Creating pair RDD

```
val pairs = words.map(x => (x, 1))
```

- First part is always the key and second part is the value

Key Value Pair Operations

Consider RDD $\{(1, 2), (3, 4), (3, 6)\}$

`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`

Other operations such as: mapValues, keys, values, sortByKey

Operation on two pair RDDs

$$rdd = \{(1, 2), (3, 4), (3, 6)\}\ other = \{(3, 9)\}$$

`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	`{(1, 2)}`
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.rightOuterJoin(other)`	`{(3, (Some(4),9)), (3, (Some(6),9))}`

Operation on two pair RDDs

$$rdd = \{(1, 2), (3, 4), (3, 6)\}\ other = \{(3, 9)\}$$

`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.leftOuterJoin(other)`	`{(1, (2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,([2], [])), (3,([4, 6],[9]))}`

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

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

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

Accumulators

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 standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator's value, not the tasks

```
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Memory Management

Spark provides three options for persist RDDs:

(1) in-memory storage as deserialized Java Objs

    >> fastest, JVM can access RDD natively

(2) in-memory storage as serialized data

    >> space limited, choose another efficient representation, lower performance cost

(3) on-disk storage

    >> RDD too large to keep in memory, and costly to recompute

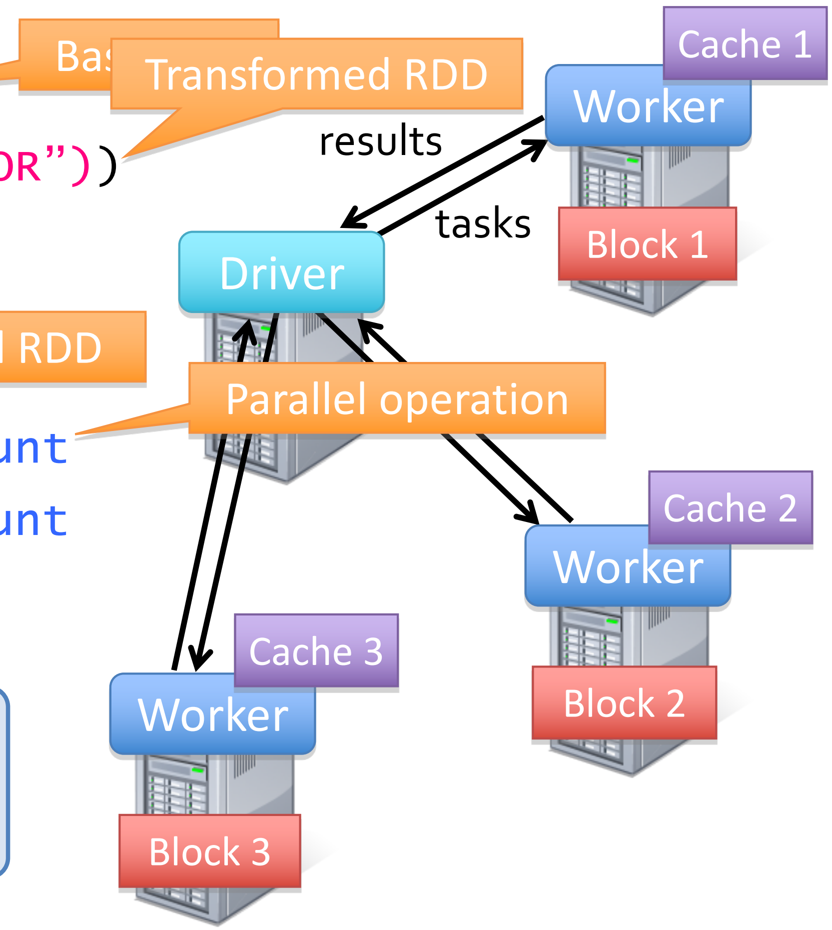
Example: Log Mining

- 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
. . .
```

Cached RDD

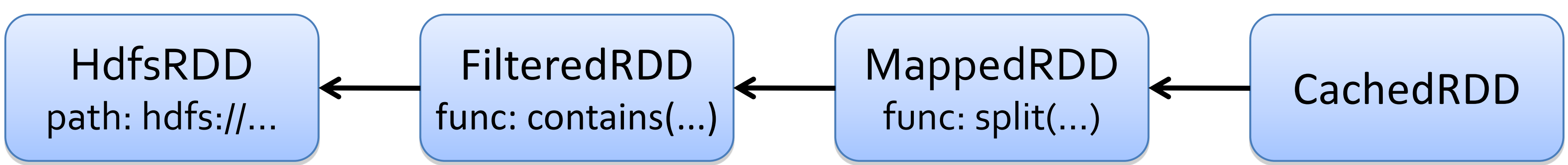


Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

RDD Fault Tolerance

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

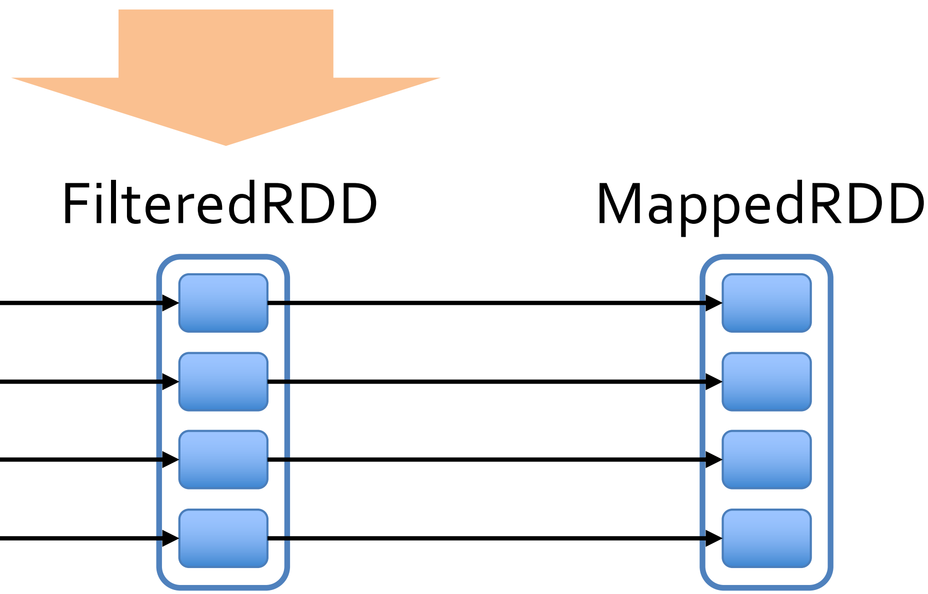
```
cachedMsgs =
textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
                        .cache()
```



Fault Recovery

- RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.:
```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



Benefits of RDD Model

- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- Despite being restricted, model seems applicable to a broad variety of applications

RDDs vs Distributed Shared Memory

Concern	RDDs	Distr. Shared Mem.
Reads	Fine-grained	Fine-grained
Writes	Bulk transformations	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using speculative execution	Difficult
Work placement	Automatic based on data locality	Up to app (but runtime aims for transparency)

Representing RDDs

Challenge: choosing a representation for RDDs that can track lineage across transformations

Each RDD include:

1) A set of partitions(atomic pieces of datasets)
2) A set of dependencies on parent RDDs
3) A function for computing the dataset based its parents
4) Metadata about its partitioning scheme
5) Data placement

Interface used to represent RDDs

Operation	Meaning
partitons()	Return s list of partition objects
preferredLocations(p)	List nodes where partition p can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(p, parenetIters)	Compute the elements of partition p given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

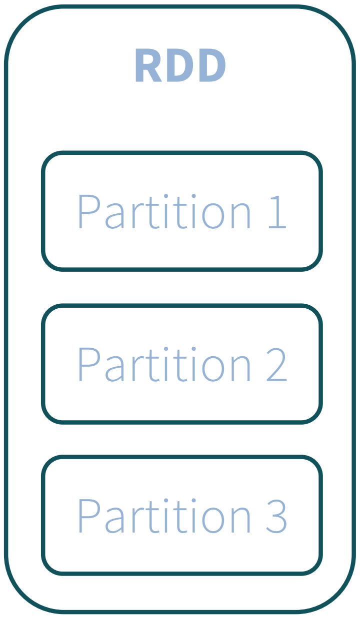
Internals of the RDD Interface

1) List of partitions

2) Set of dependencies on parent RDDs

3) Function to compute a partition, given parents

4) Optional partitioning info for k/v RDDs (Partitioner)



Example: Hadoop RDD

Partitions = 1 per HDFS block

Dependencies = None

compute(partition) = read corresponding HDFS block

Partitioner = None

> rdd = spark.hadoopFile("hdfs://click_logs/")

Example: Filtered RDD

Partitions = parent partitions

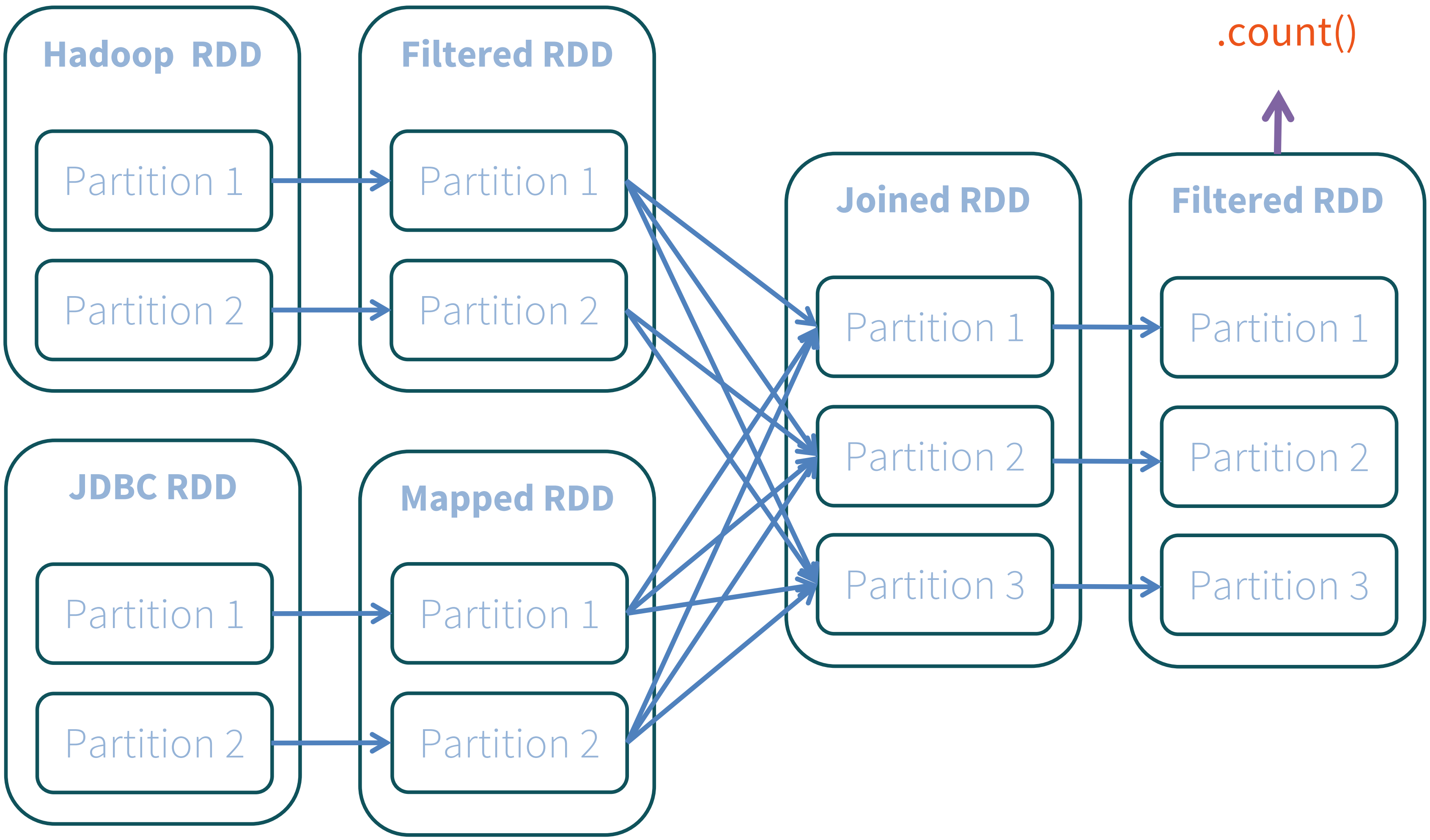
Dependencies = a single parent

compute(partition) = call parent.compute(partition) and filter

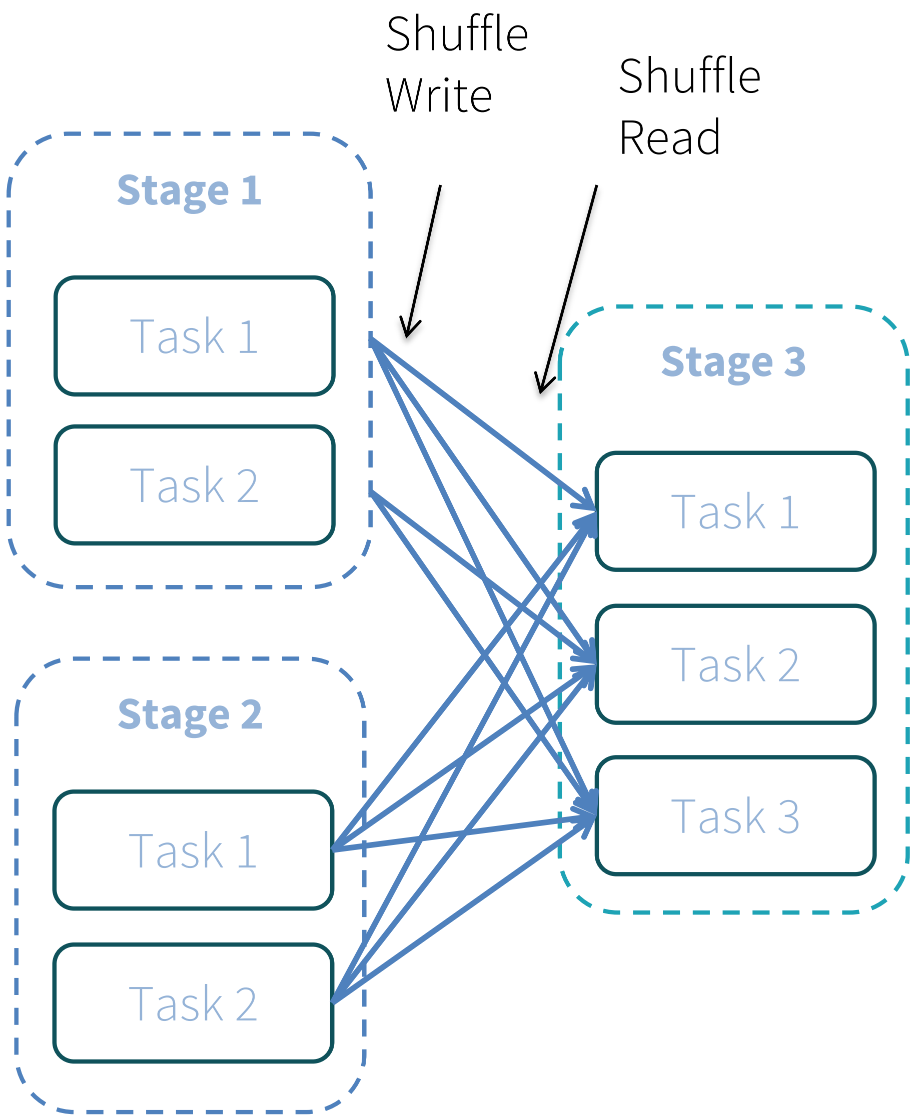
Partitioner = parent partitioner

> filtered = rdd.filter(lambda x: x contains “ERROR”)

A More Complex DAG

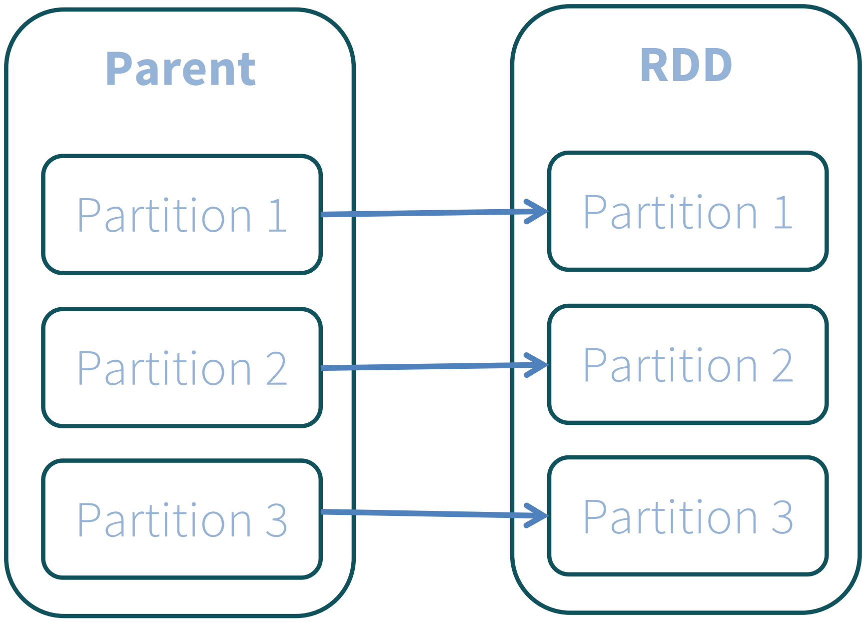


A More Complex DAG

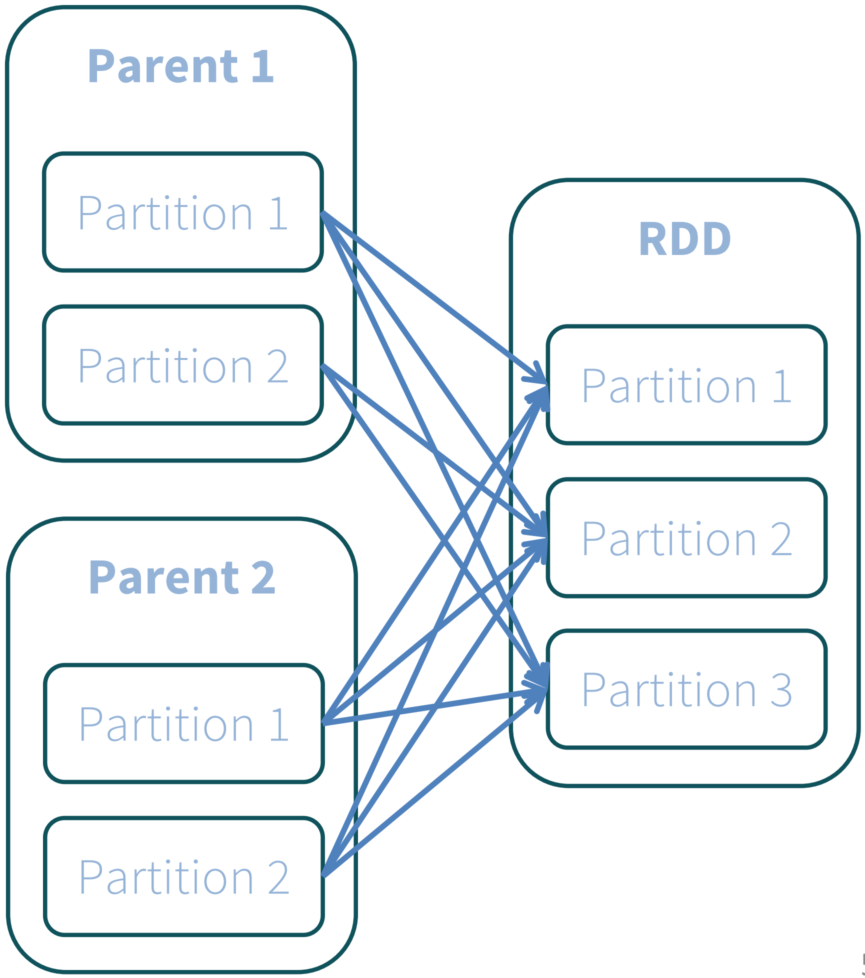


Narrow and Wide Transformations

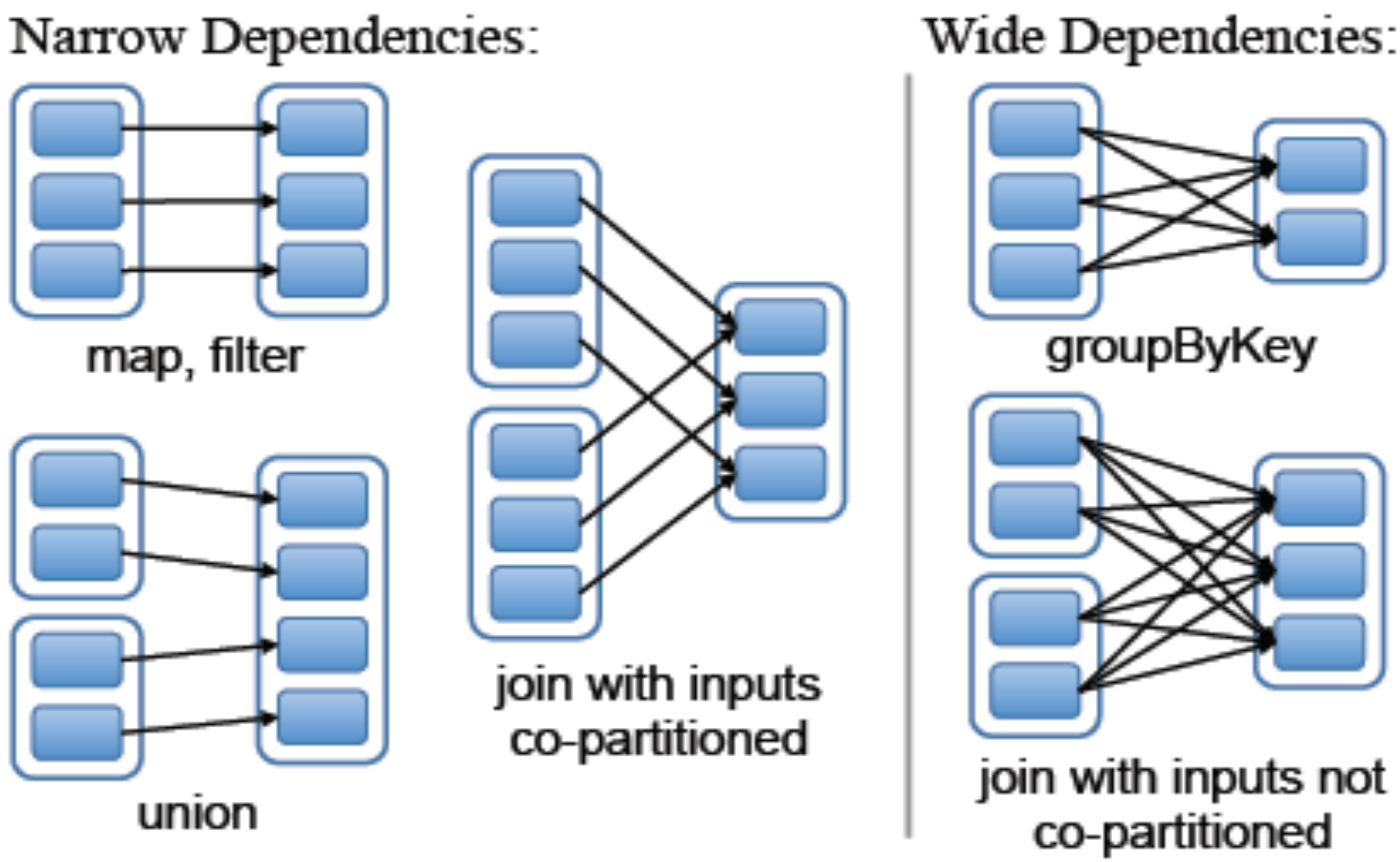
FilteredRDD



JoinedRDD

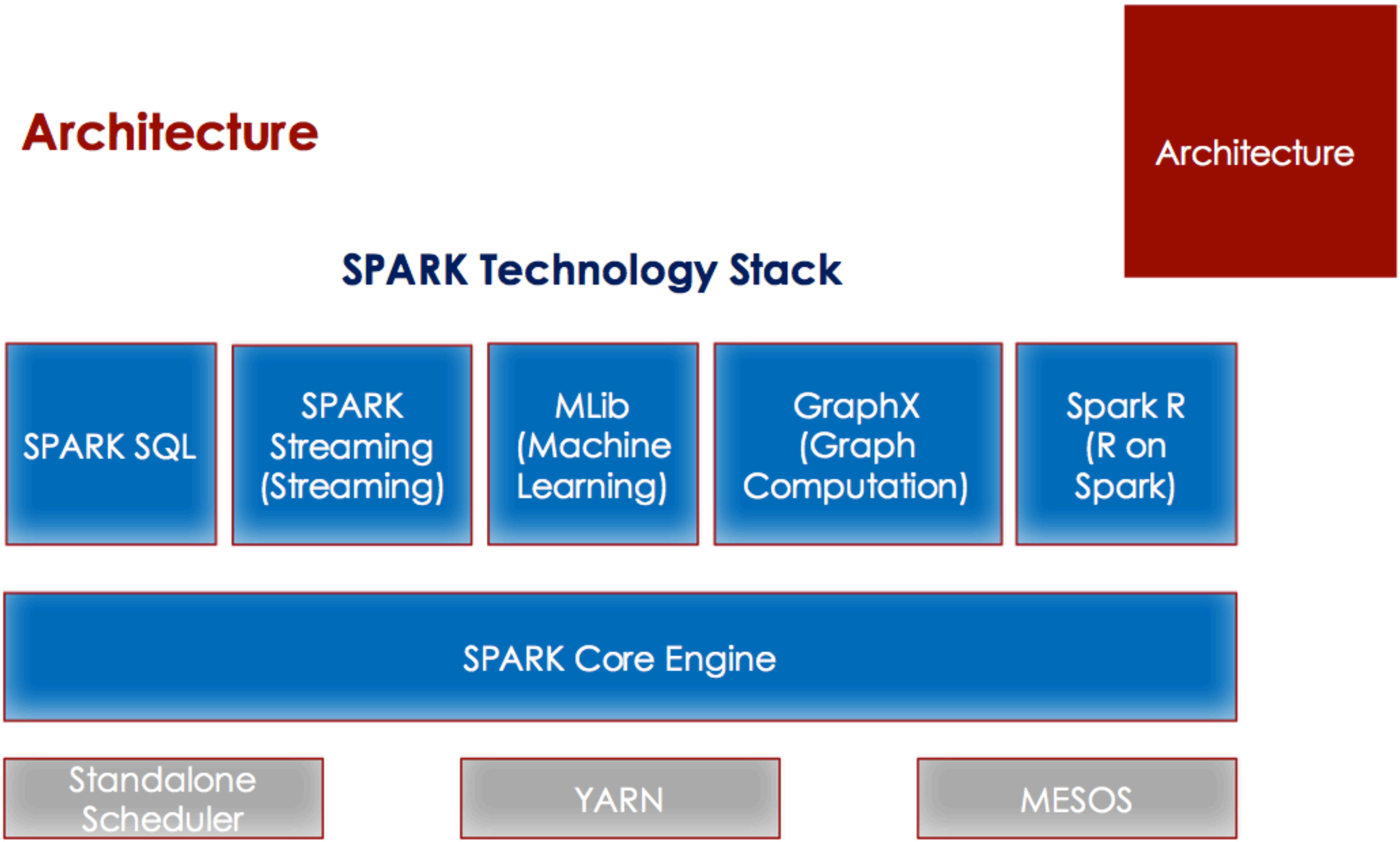


RDD Dependencies



Each box is an RDD, with partitions shown as shaded rectangles

Spark Architecture



Spark Architecture

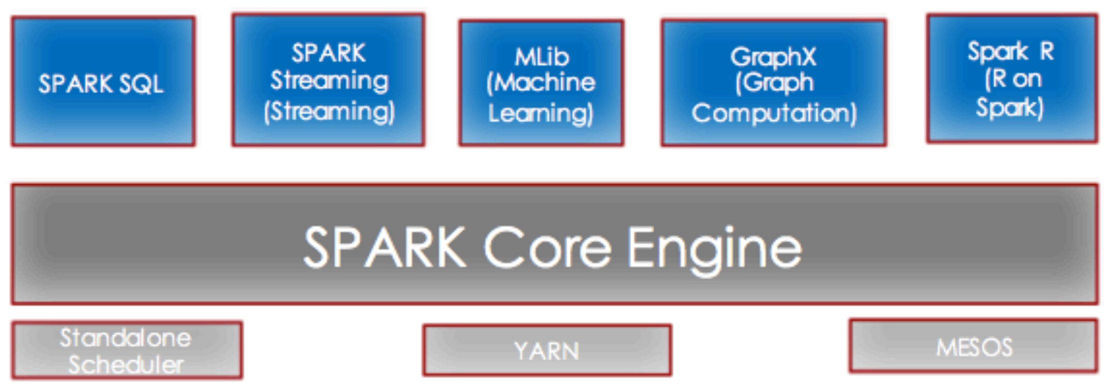
Architecture

Architecture

SPARK Technology Stack

SPARK Core Engine

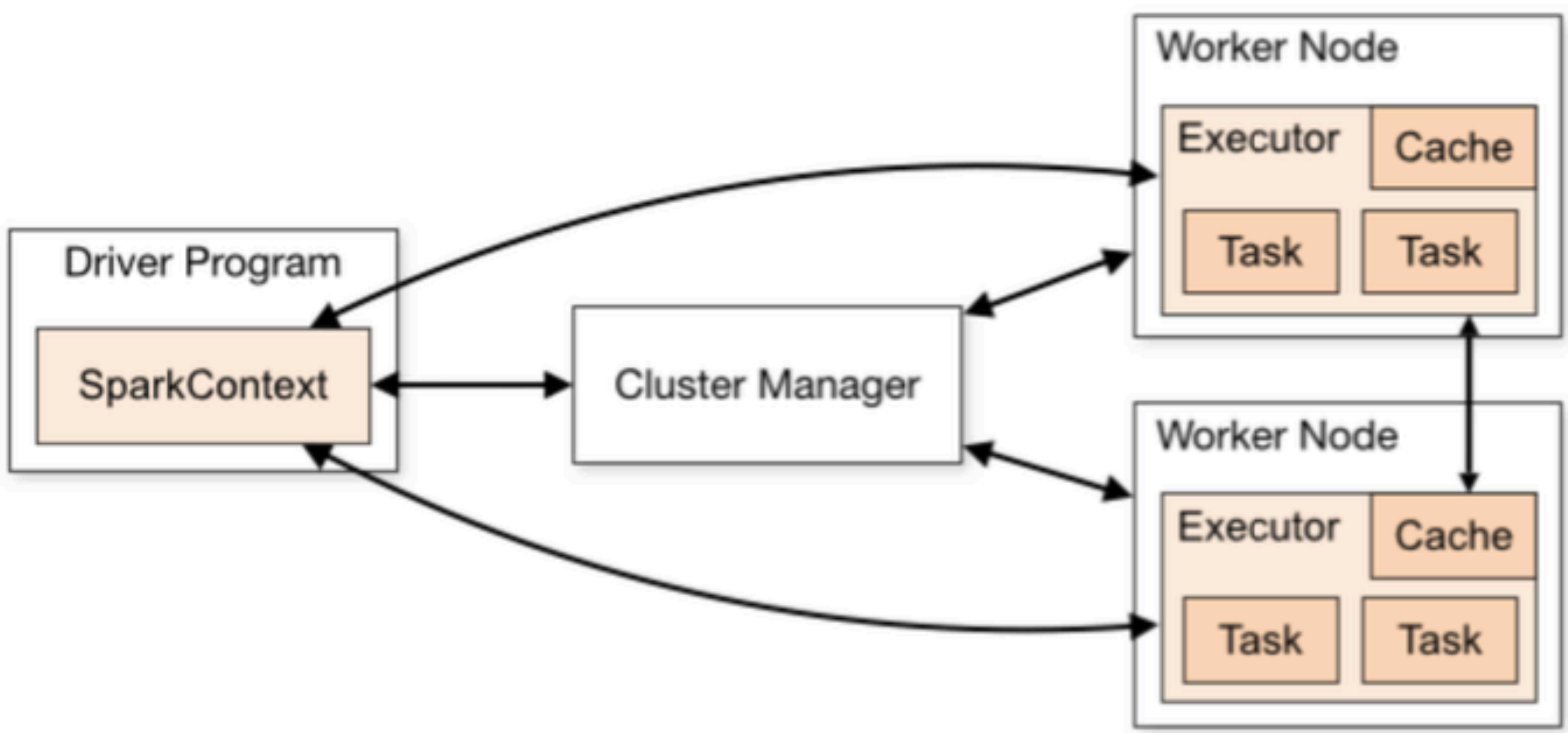
- Basic functionality of Spark
- Uses RDDs (Resilient Distributed Datasets)
- Contains APIs for manipulating RDDs



Spark RDDs are a collection of items distributed across compute nodes.
Spark core APIs allows manipulation of these RDDs in parallel

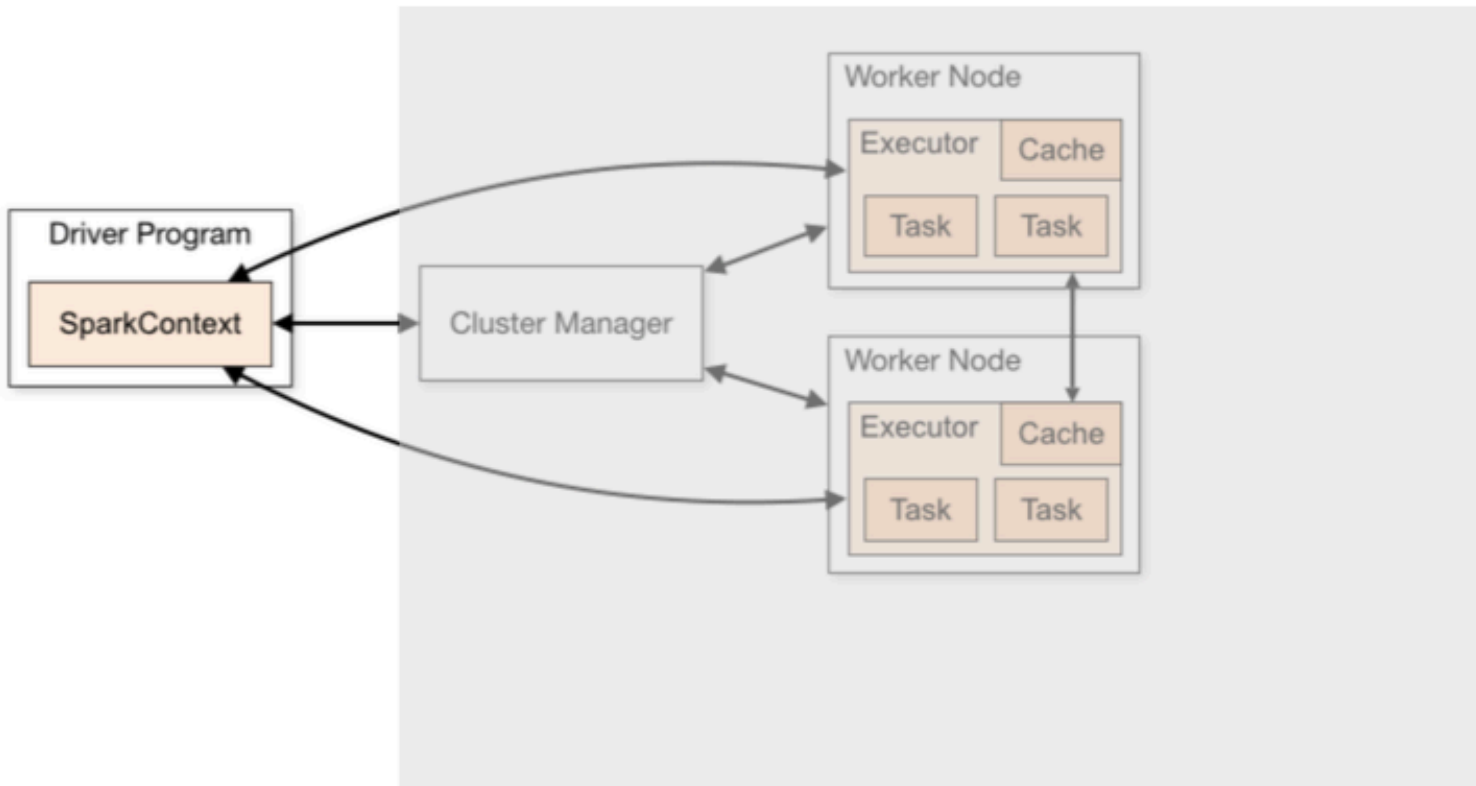
Spark Processing

SPARK Processing



Spark Processing

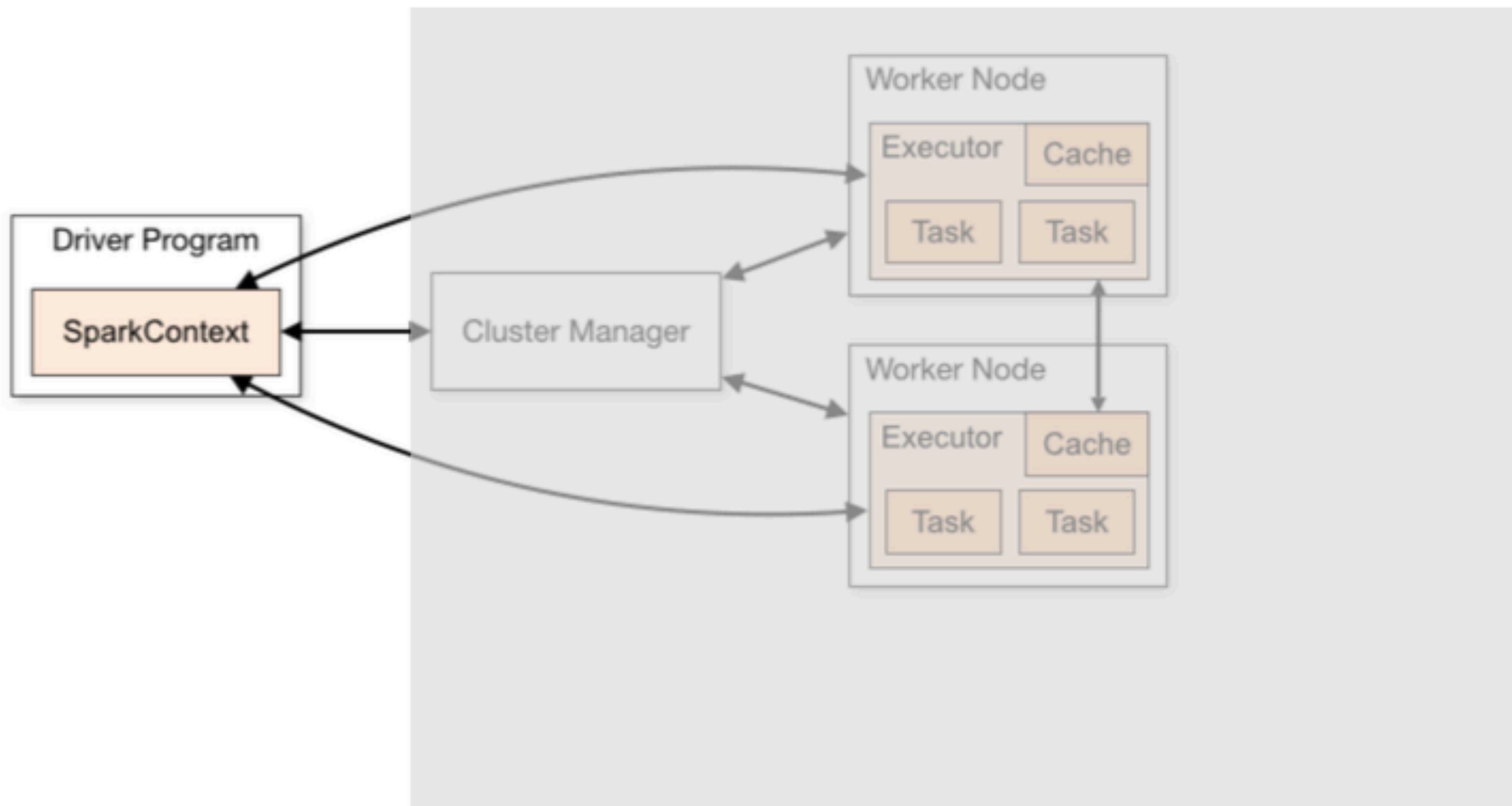
Driver program accesses Spark through a SparkContext object.



Source: https://spark.apache.org/docs/latest/cluster-overview.html

Spark Processing

Spark Context represents a connection to a computing cluster
Once created, it can be used to build RDDs

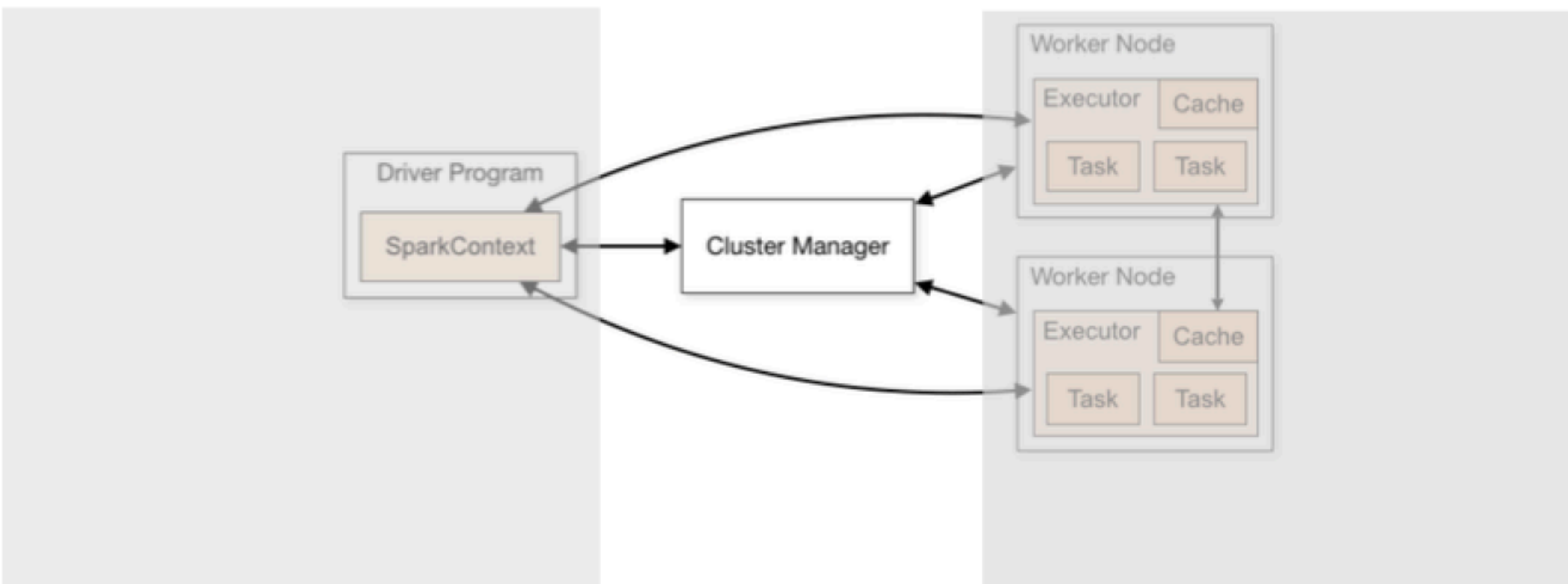


Source: https://spark.apache.org/docs/latest/cluster-overview.html

Spark Processing

Cluster Manager is an external service

- A default built-in cluster manager called Standalone Cluster manager is pre-packaged with Spark
- Hadoop YARN and Apache Mesos are two popular cluster managers
- Driver requests cluster manager to provide resources for launching executors
- Cluster manager launches executors which are then used by driver to run tasks



Spark Processing

Executors are processes that execute tasks

- Executors run the tasks and return results to the driver
- Also provide in-memory storage for RDDs