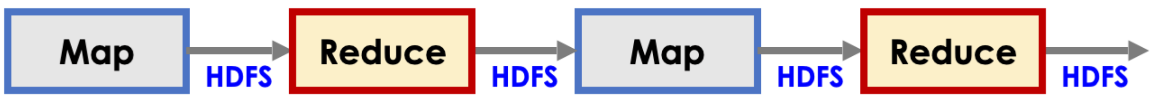
Limitations of Hadoop MapReduce

- good for one-shot queries when analyzing data (word count, table join, log search) convert to one MR job
- inefficient for iterative queries
  - must have multiple map reduce
  - shows up in many ml task (gradient descent)
  - applications that reuse intermediate results across multiple computations



An *iterative query* includes multiple mr jobs

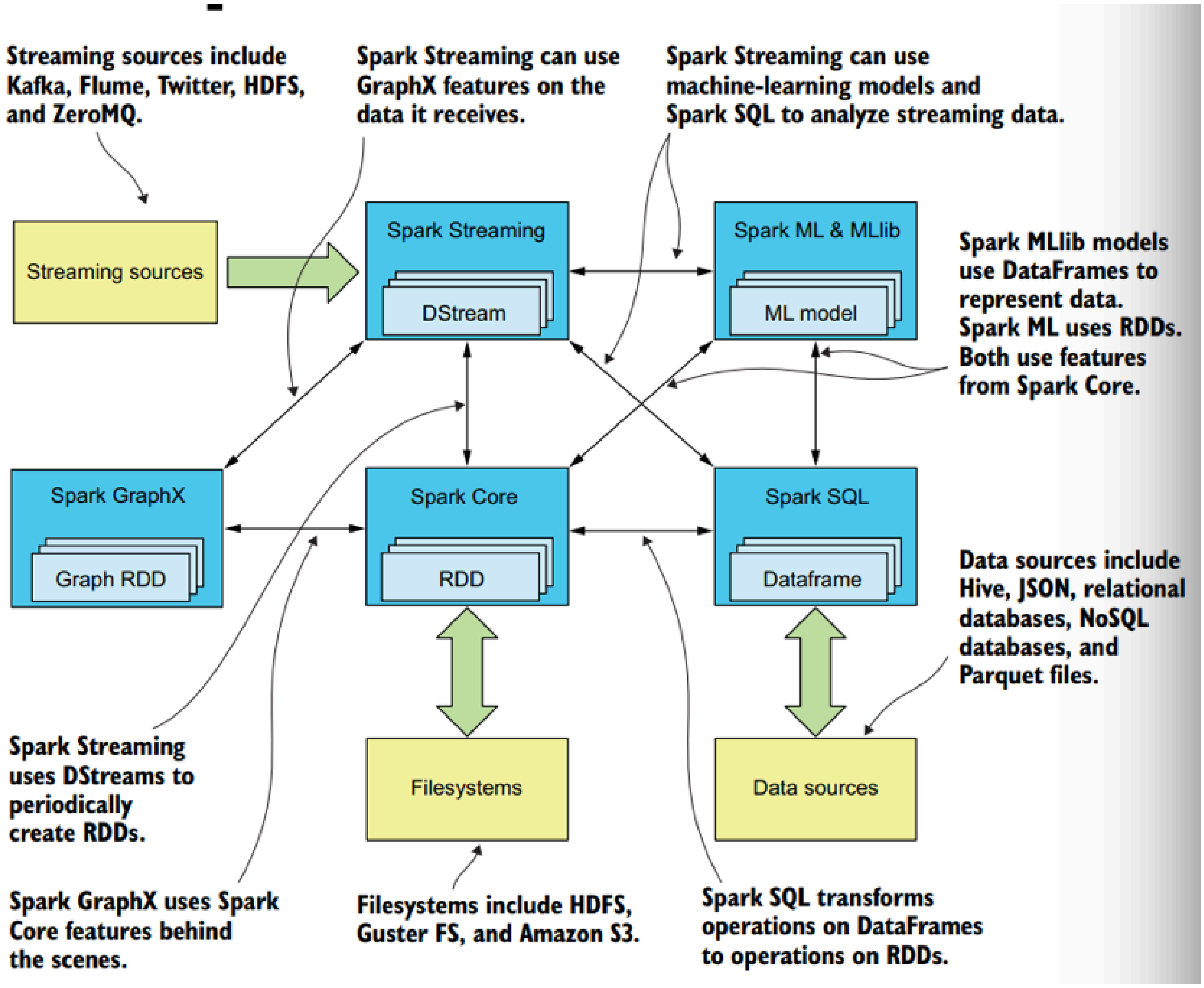
- output of the 1st mr job is the output of the 2nd mr job .....
  Each phase outputs intermediate results in HDFS(on disk), very slow

SPARK

- improved over hadoop
- in memory computing, whenever possible store everything (including intermediate results) in memory instead of disk
- much faster, up to 10 times faster on some iterative workloads, (we care about performance in big data systems)
- has more function, more than simple mr jobs, easier for big data analytics
- written in Scala but can use Java or Python

Spark components

- core
- sql
- graph
- streaming
- ml



SPARK CORE

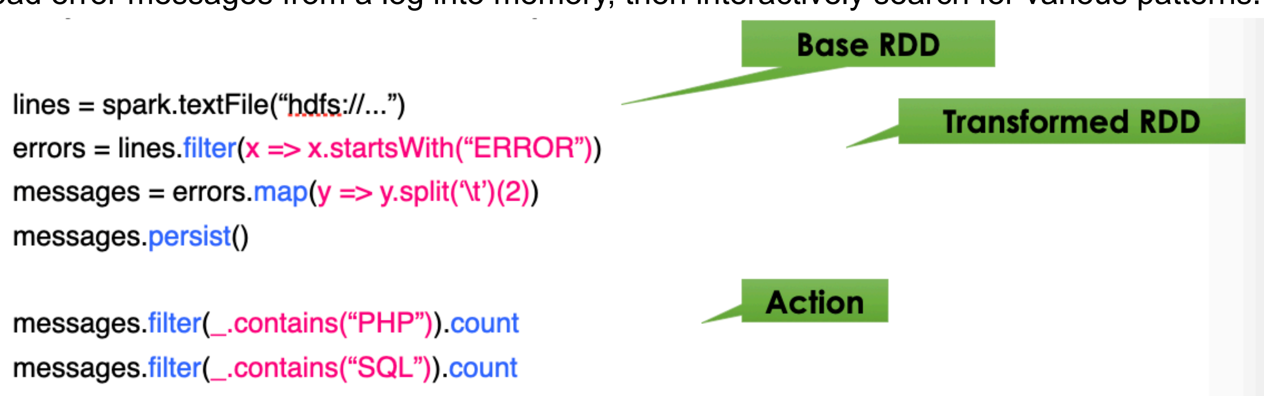
- to keep track of different computation stages, spark defines a new concept called Resilient Distributed Datasets (RDD)
- RDD abstracts the data (or objects) transmitted among different computation stages
- RDD is the basic unit of computation and transformation
- RDD is read-only (immutable), partitioned collection of records (think of it like an array or a list but it is a collection of items , a set??)
- RDD can be created from:
  - data in memory or on storage (base RDD)
  - other RDDs (transformed RDD)

  CREATE RDD

```
# SparkContext sc: Spark environment that stores the configuration
# solution 1: from HDFS
# textFile is a build-in method for parsing various types of data files
> val rdd1 = sc.textFile("hdfs://file-path")
# solution 2: from a local file
> val rdd2 = sc.textFile("input/input1.txt")
# solution 3: convert an in-memory array to an RDD (with 3 partitions)
# by default, it's the num of cores in your server
> val rdd3 = sc.parallelize([1,2,3,4,5],3)
> println(rdd3.getNumPartitions)
# solution 4: from another RDD
> val rdd4=rdd3.map(x=>x+10)
```

- All RDDs of a task can form a graph, called *lineage graph*
  - one RDD can be derived from one or more RDDs
  - overall data flow is a graph
- Fault tolerant: can be reconstructed on failure using lineage graph or checkpointed
  - no need for replication
- RDDs are stored in memory can also persist on disk
  - when possible RDDs are stored in memory for fast performance
  - can be reused for multiple computations efficiently (without disk access)
  - can also persist on disk when necessary (insufficient memory)

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



RDD OPERATIONS

- Transformation: transform one RDD to another one
- Action: take some actions on a particular RDD, like count(),...

RDDs provide more functionalities than Hadoop MR

RDD operations are coarse-grained, applied to all items on RDD

Transformation RDD

- map(f: T➔U)
  - RDD[T] ➔ RDD[U]
  - Convert an old RDD to a new RDD by applying the function f to each item in the old RDD

```
data = [1,2,3,4,5]
rdd = sc.parallelize(data).map(x=>x+1)
rdd.foreach(println) # output is [2,3,4,5,6]
```

- flatMap(f: T➔seq[U])
  - RDD[T] ➔ RDD[U]
  - Similar to map(), but it'll flatten the output

```
data = [2,3,4]
rdd1 = sc.parallelize(data)

# range(1,x) will print out values from 1..x-1
# output: [[1], [1, 2], [1, 2, 3]]
rdd1.map(x => range(1,x))

# output: [1, 1, 2, 1, 2, 3]
rdd1.flatMap(x => range(1,x))
```

RDUE

- filter(f: T ➔ Bool)
  - RDD[T] ➔ RDD[T]
  - Convert an old RDD to a new RDD by applying the function f to each item in the old RDD and only showing the qualified items
  - You can think f as a filter

```
data = [1,2,3,4,5]
rdd = sc.parallelize(data).filter(x=>x%2==0)
rdd.foreach(println) # output is [2,4]
```

- reduceByKey(f: (V,V) ➔ V)
  - RDD[(K,V)] ➔ RDD[(K,V)]
  - You can also define a function for more complicated computations

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
rdd.reduceByKey(add) # output is [('a', 2), ('b', 1)]
rdd.reduceByKey((x,y) => x+y) # output is [('a', 2), ('b', 1)]
```

- join()
  - (RDD[(K, V)],RDD[(K, W)]) => RDD[(K, (V, W))]
  - It merges two RDDs based on the same key

```
rdd1 = sc.parallelize([("a", 1), ("b", 4)])
rdd2 = sc.parallelize([("a", 2), ("a", 3)])
rdd1.join(rdd2) # output is [('a', (1, 2)), ('a', (1, 3))]
```

- union()
  - It merges two RDDs (keeps duplicates if any)

```
rdd1 = sc.parallelize([("a", 1), ("b", 4)])
rdd2 = sc.parallelize([("a", 2), ("a", 3)])
rdd1.union(rdd2) # output is [("a", 1), ("b", 4), ("a", 2), ("a", 3)]
```

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$

ACTION RDDS

- action RDD performs actual computation on the input RDD
  - Count(): RDD[T] => Long
  - Collect(): RDD[T] => Seq[T]
  - Reduce(): RDD[T] => T
  - Save(): Outputs RDD to a storage system, e.g., HDFS

- Count()
  - Return the num of items in an RDD

    sc.parallelize([1,2,3,4,5]).count() # output is 5

- Collect()
  - Return the items in an RDD

    sc.parallelize([1,2,3,4,5]).collect() # output is [1,2,3,4,5]

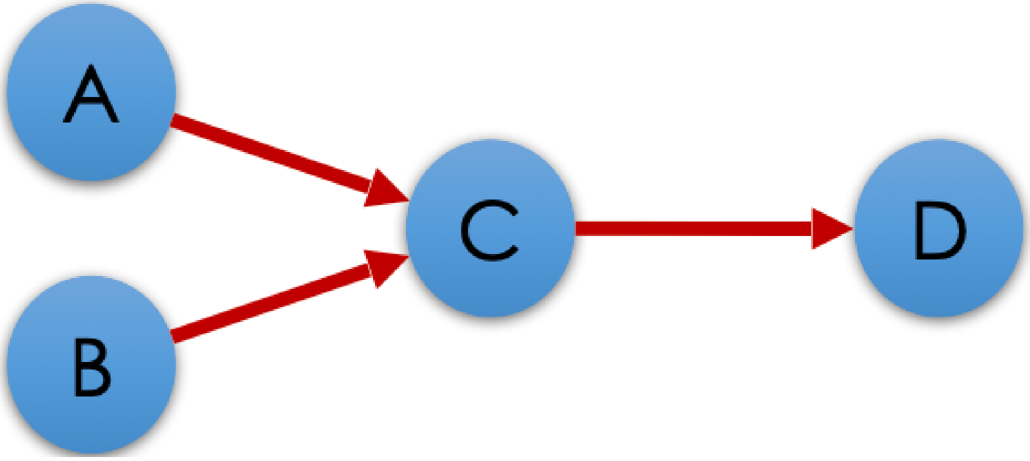
- Reduce(f: (T,T) => T)
  - RDD[T] ➔ T
  - Reduce the items of the input RDD using the function specified
  - Use the function to compute the first two items and produce a new item. Then use the function to compute the new item and the 3rd item and produce another new item…

```
sc.parallelize([1,2,3]).reduce((a,b) => a + b) # output is 6
sc.parallelize([1,2,3]).reduce((a,b) => a min b) # output is 1
sc.parallelize([1,2,3]).reduce((a,b) => a max b) # output is 3
```

SPARK DAG (directed acyclic graph)

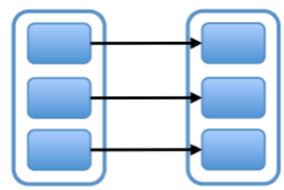
- workflow is represented as a DAG
- DAG tracks dependencies (lineage)
  - nodes are RDDs

- arrows are transformations

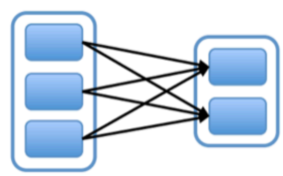


SPARK DEPENDENCY

- Narrow dependency: Parent partition is used by only one child partition
  - Examples: map, filter

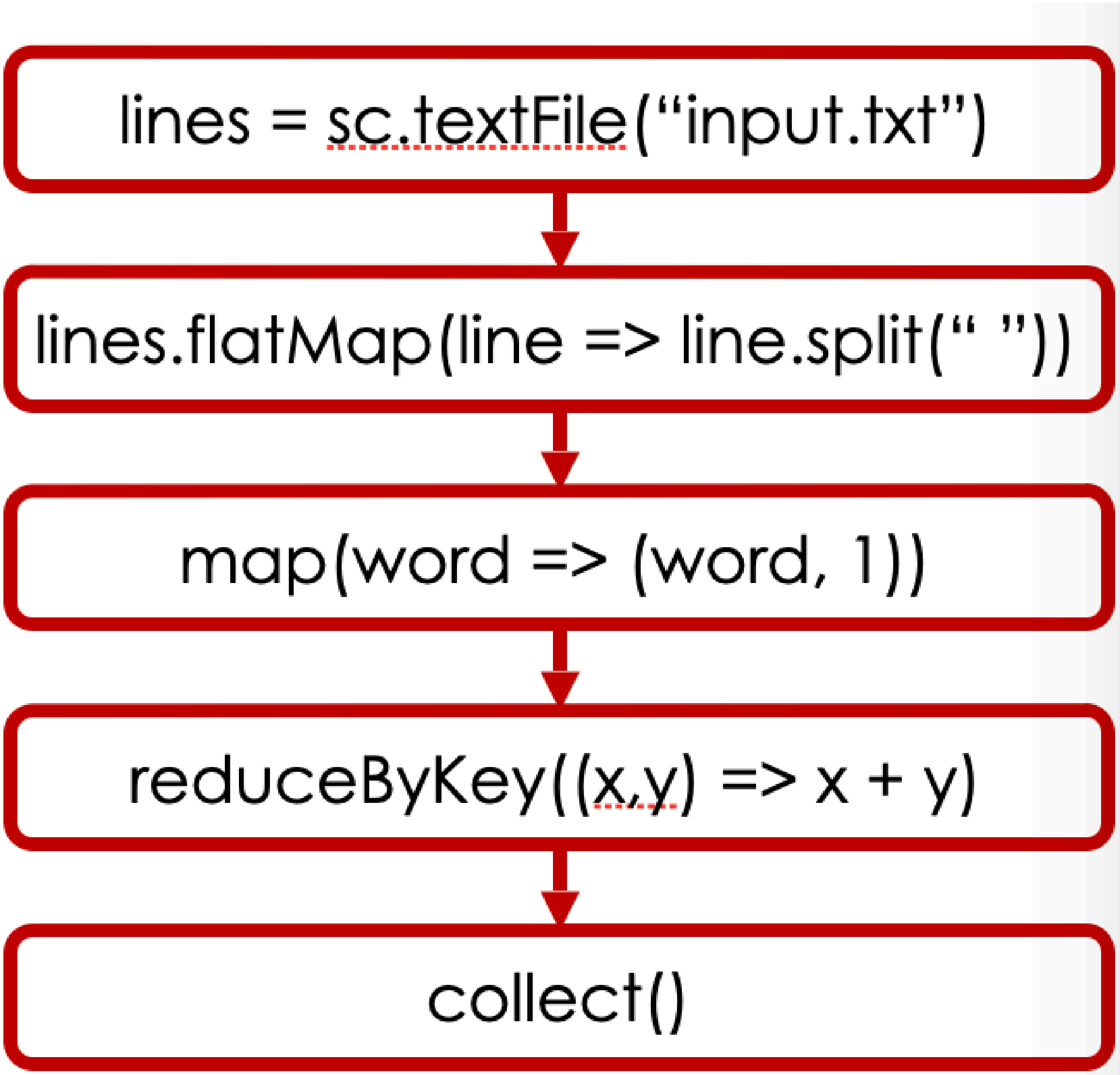


- Wide dependency: Parent partition is used by many child partitions
  - Example: reduceBy



SPARK EXECUTION

- Lazy evaluation
  - data in RDDs is not processed until an action is performed
  - do actual evaluation only when we see action RDDs (only in collect() will trigger actual evolution & computation)



FAULT TOLERANCE

- if a server executing RDD is crashed, we simply reconstruct the RDD from the lineage graph
- For fast recovery, you can persist some intermediate RDDs so that you don't have to rebuild from beginning (checkpointing)