CPSC 436C
Cloud Computing for Data Science

Apache Spark

Maryam R.Aliabadi

mraiyata@cs.ubc.ca

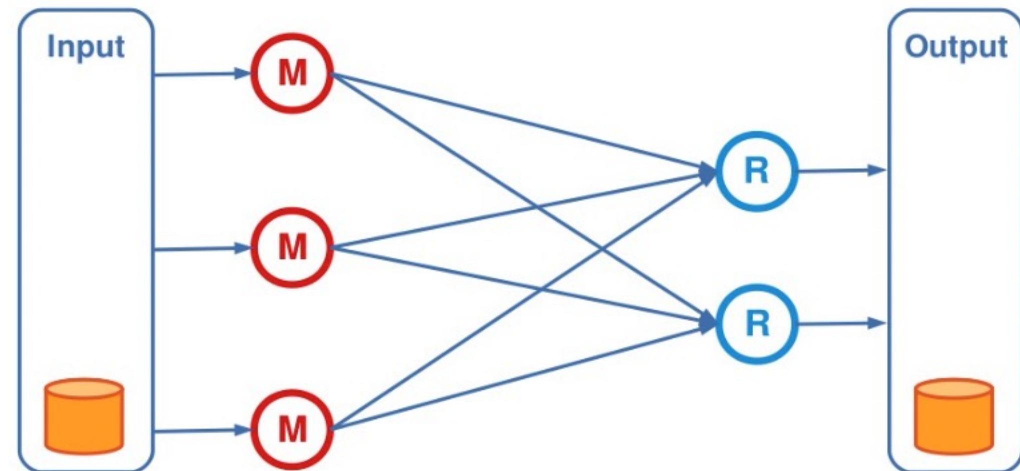
Spring 2024

Last Week's Review

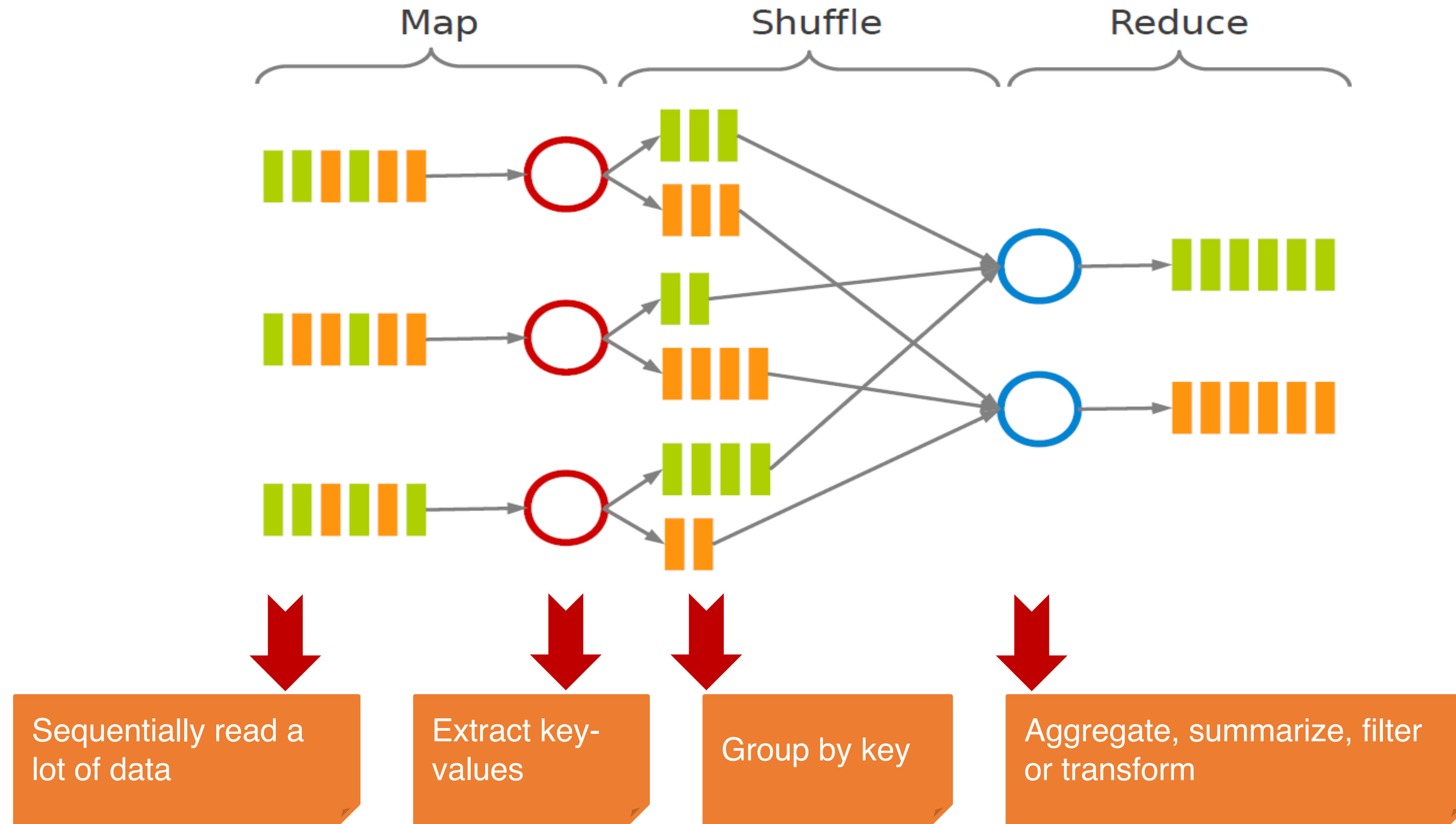
- Data Processing by MapReduce
  - Programming model: Map and Reduce
  - Execution framework
  - Batch processing
  - Shared nothing architecture

MapReduce

- MapReduce processes large data sets with a parallel/distributed algorithm on clusters of commodity hardware.
- It Provides a cluster programming language that is based on acyclic data flow from stable storage to stable storage.
  - Data distribution
  - Fault tolerance
  - Load balancing
  - Locality

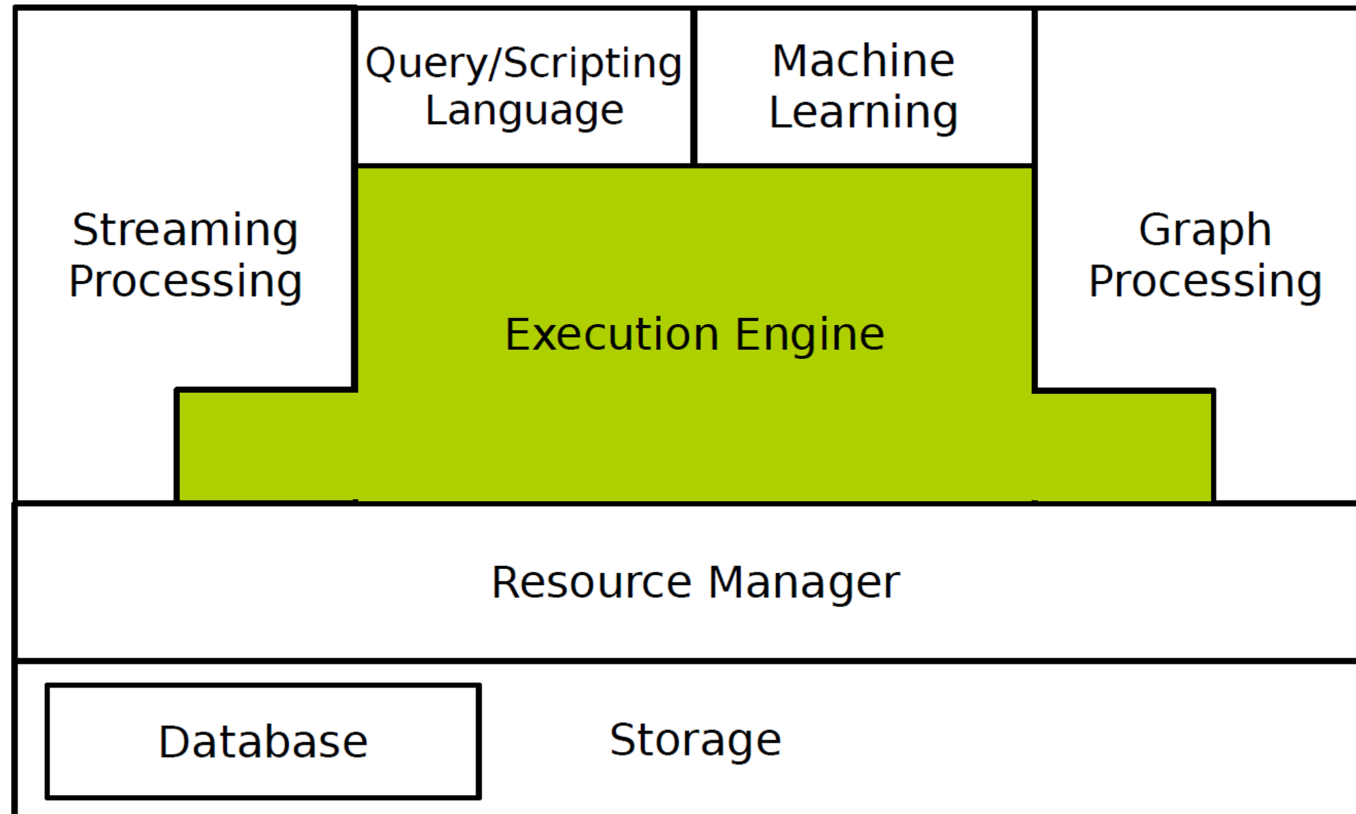


MapReduce Example – Word Count



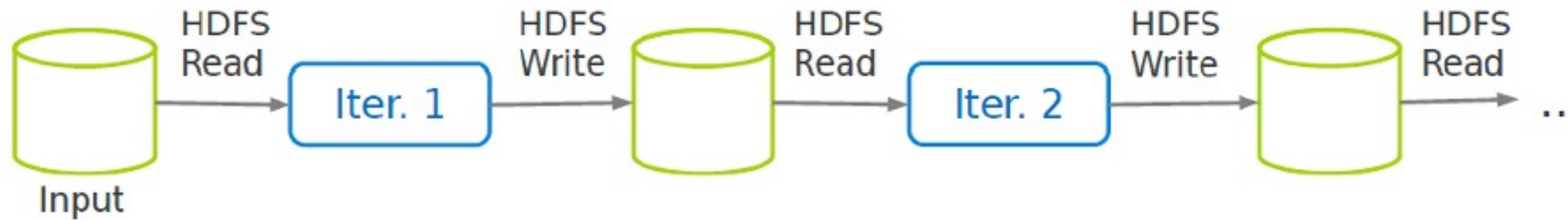
Today's Topics

- Data Processing by Apache Spark

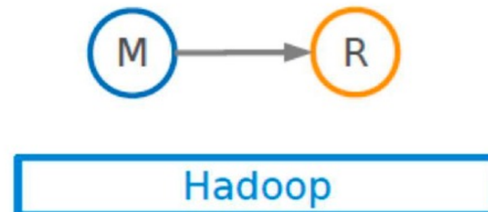


MapReduce Limitations

- Very slow for iterative and interactive jobs, i.e., Commonly spends 90% of time doing I/O.



- MapReduce programming model has not been designed for complex operations, e.g., graph analysis.



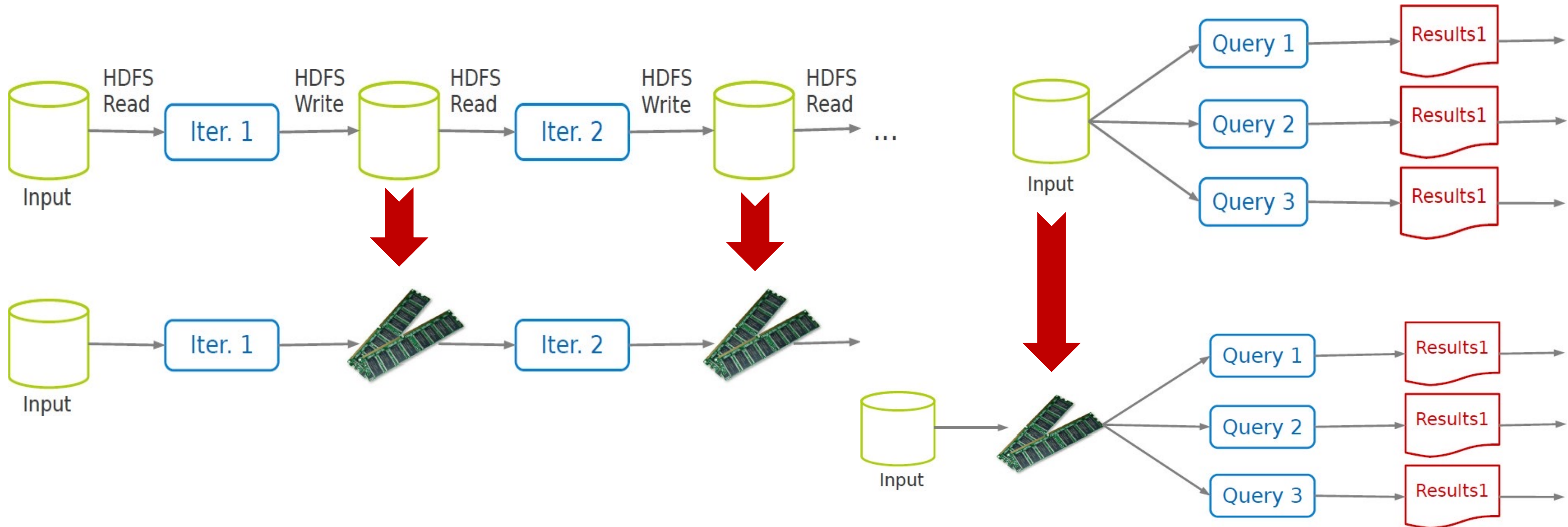
"Spark, you're on fire!"

Spark: A programming framework for large-scale distributed processing

- In-Memory Data Processing and Sharing, leading to high speed.
- Extends MapReduce with more operators.
- Support for advanced data flow graphs.

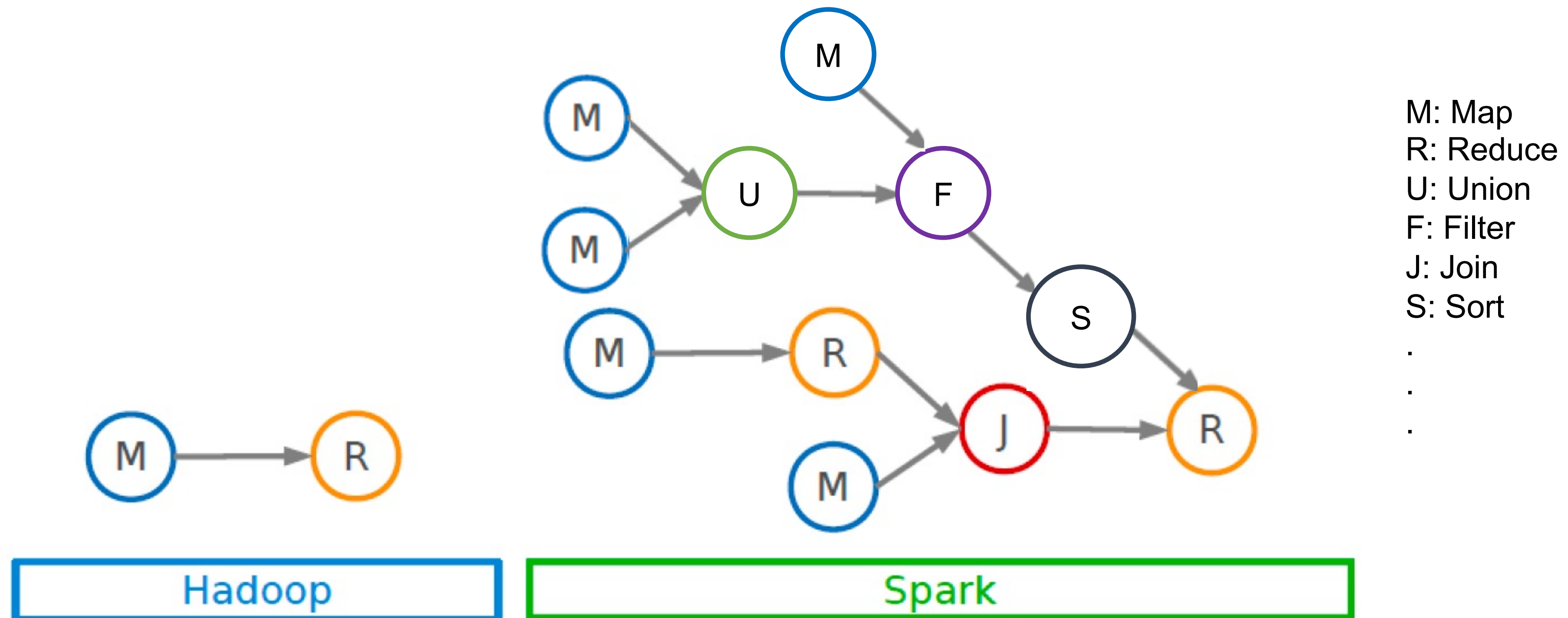
Spark Vs. MapReduce

- **In-Memory** Data Processing and Sharing.



Spark Vs. MapReduce

- Extends MapReduce with more (over 80) operators.



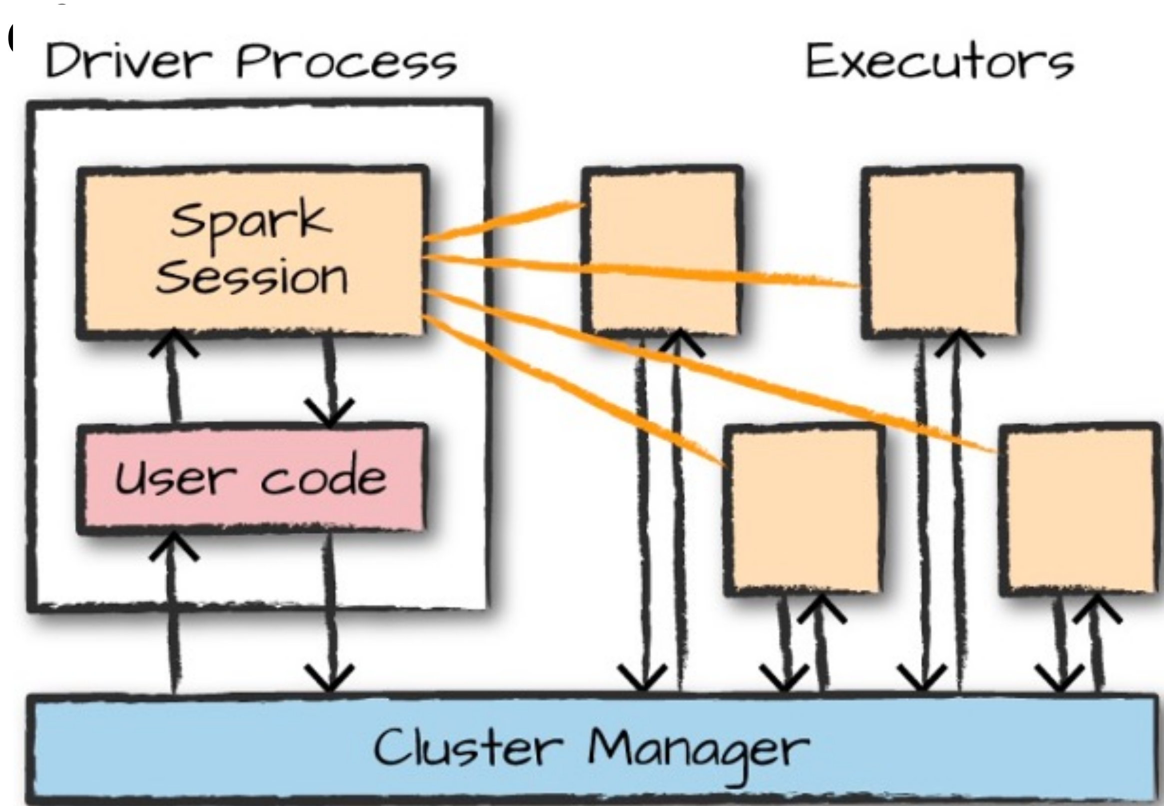
M: Map
R: Reduce
U: Union
F: Filter
J: Join
S: Sort
.
.
.

The Force Behind Spark

Execution Engine

- Spark execution engine consists of

  - A driver process
  - A set of executor processes

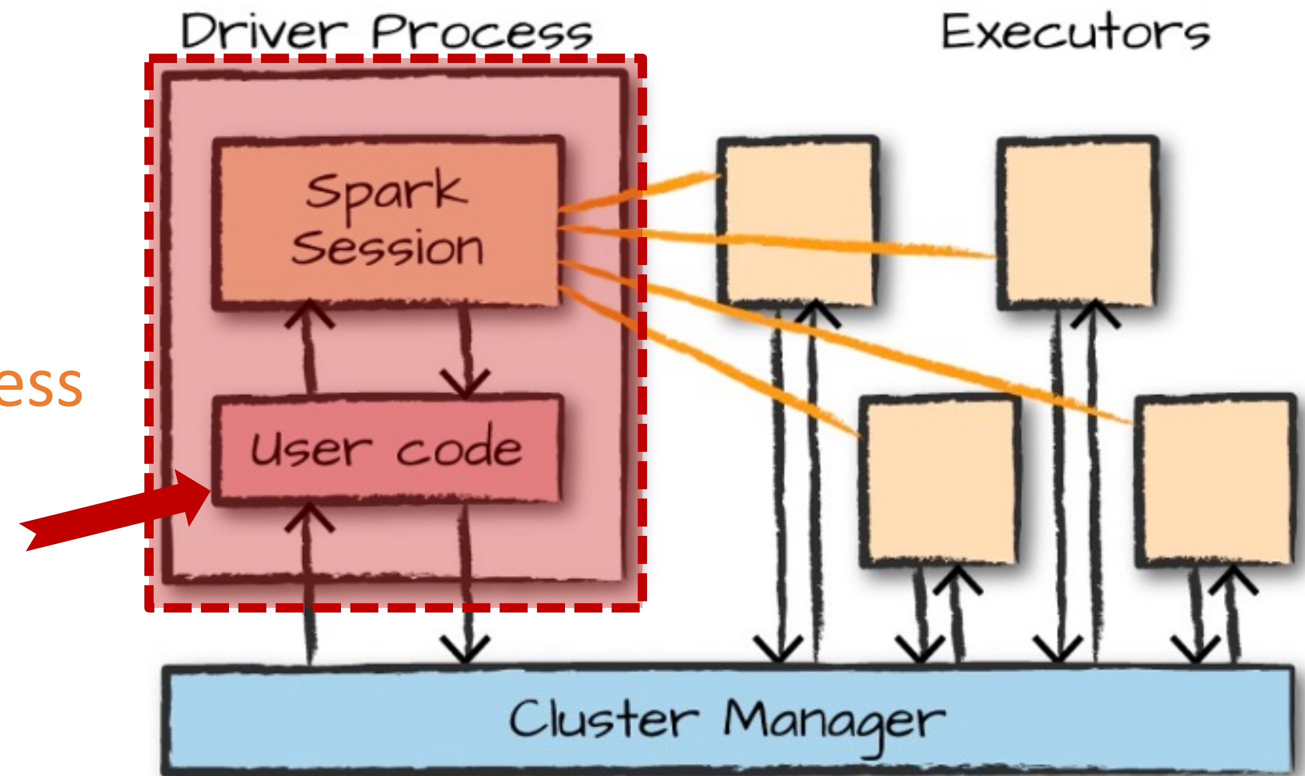


[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Driver Process

- The heart of a Spark architecture
- Runs the main() function
- SparkSession is a driver process that controls a Spark application.

Your code runs here

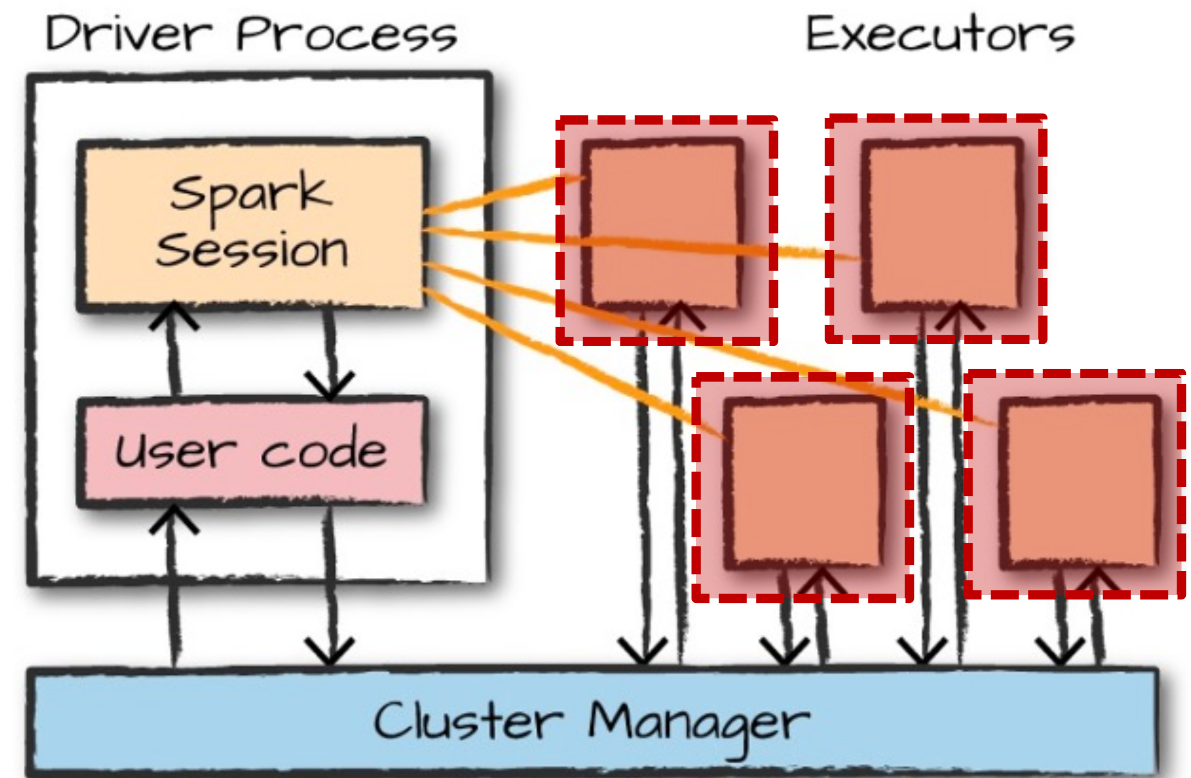


[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Executors

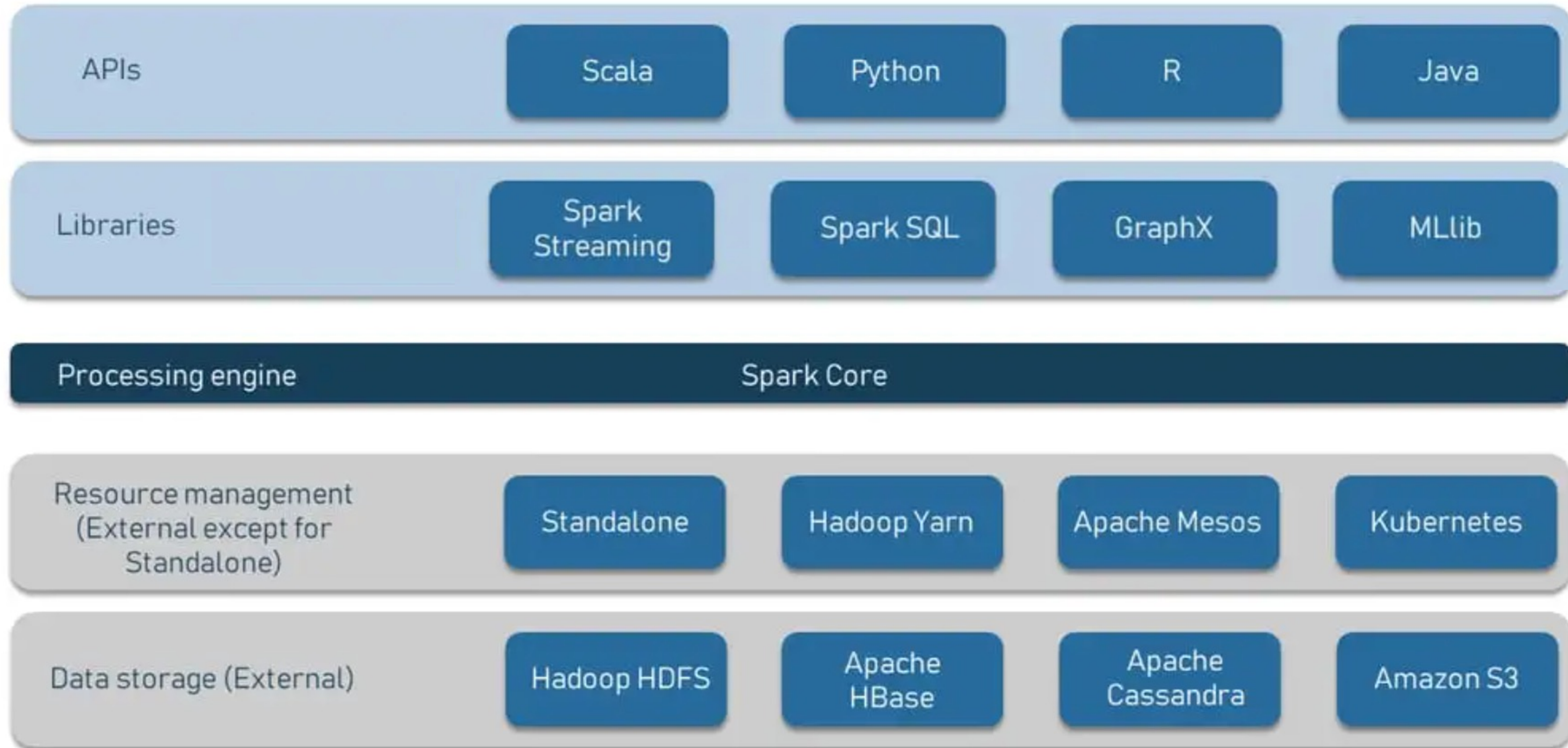
- Responsible for executing code assigned to it by the driver

- Reporting the state of the computation on that executor back to the driver



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Spark Ecosystem

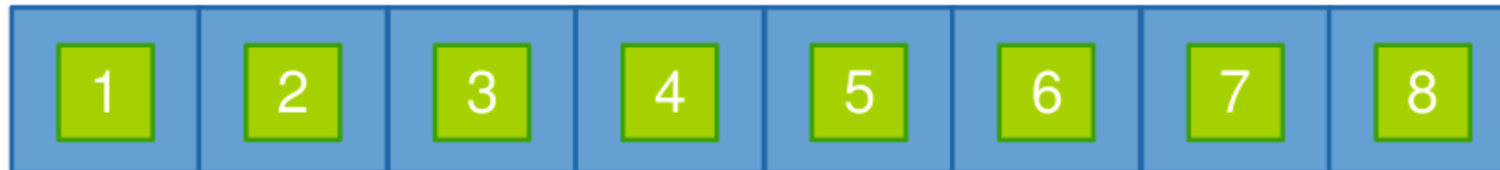


[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Spark Programming Model

Spark Data Model

- Resilient Distributed Dataset (RDD)
- Immutable collection of objects spread across a cluster.
- Like a LinkedList <MyObjects>

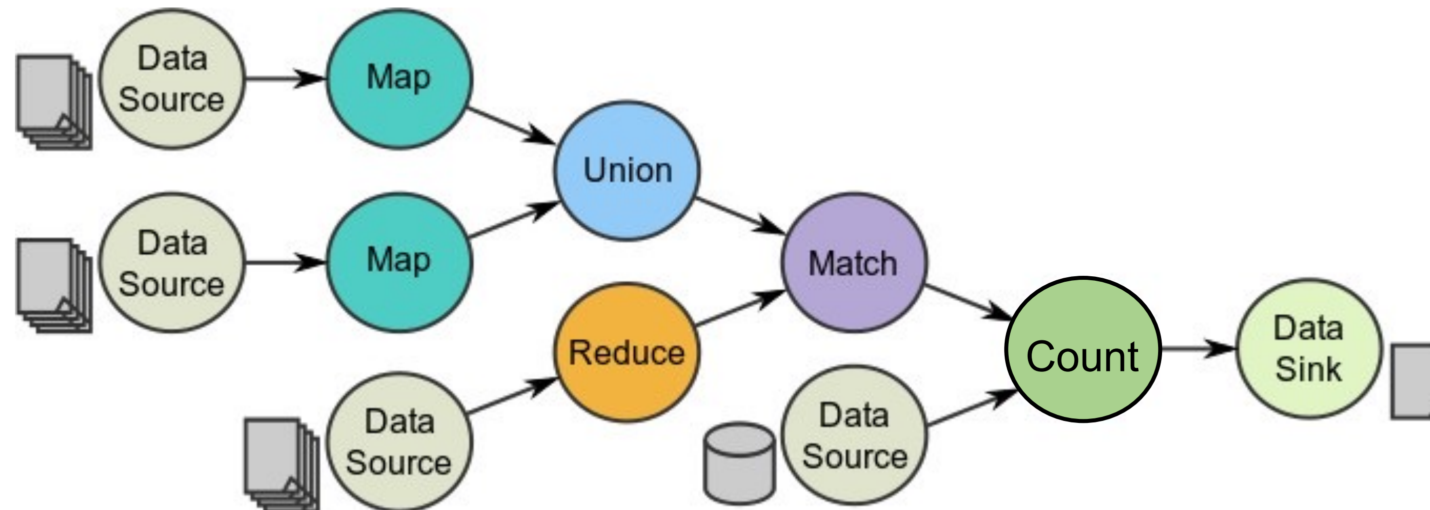


Spark Data Model

- An RDD is divided into a number of partitions.
- Partitions of an RDD can be stored on different nodes of a cluster.

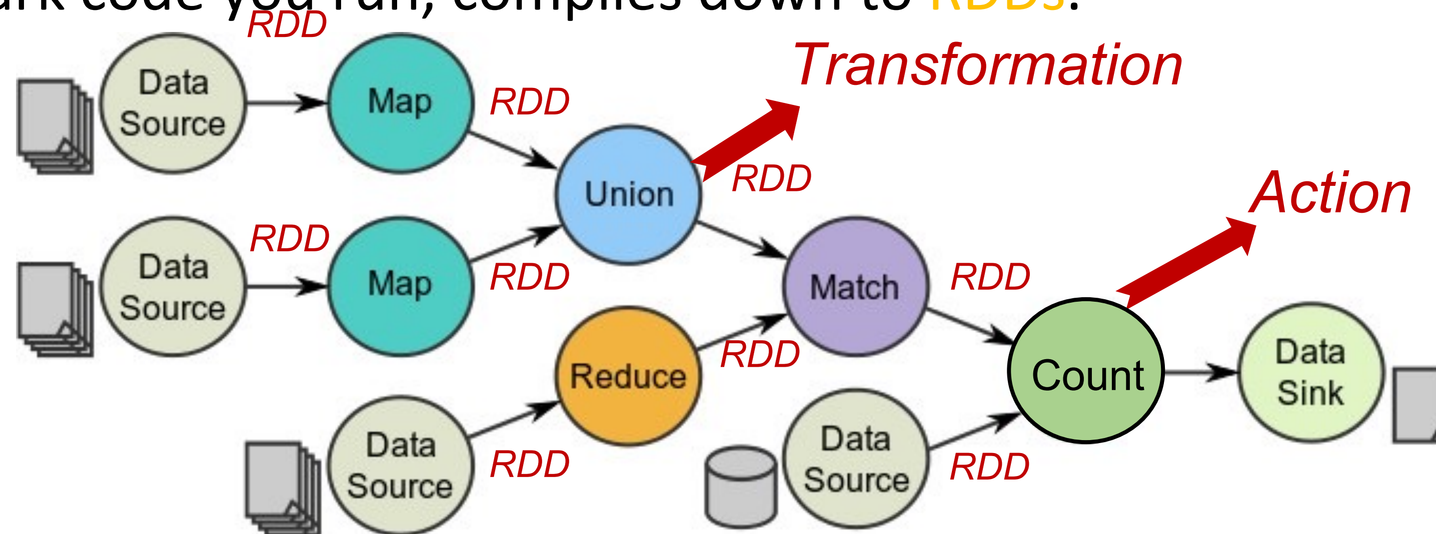
Spark Programming Model

- A Job/program is described based on directed acyclic graphs (DAG) data flow.
- A DAG representing the computations done on the RDD is called lineage graph.
- Parallelizable operators that execute user-defined functions in parallel.



RDD Operations

- There are two types of RDD operators: Transformations and Actions.
  - Transformations: Allow us to build a logical plan.
    - ►Create a new RDD from an existing one.
  - Actions: Allow us to trigger the computation.
    - ►Instructs Spark to compute a result from a series of transformations
  - All Spark code you run, compiles down to RDDs.



High Order Functions

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)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Transformations

- Create a new RDD from an existing one.
- All transformations are **lazy**.
  - Not compute their results right away.
  - Remember the transformations applied to the base dataset.
  - They are only computed when an action requires a result to be returned to the driver program.

Generic RDD Transformation

- distinct removes duplicates from the RDD.
- filter returns the RDD records that match some predicate function.

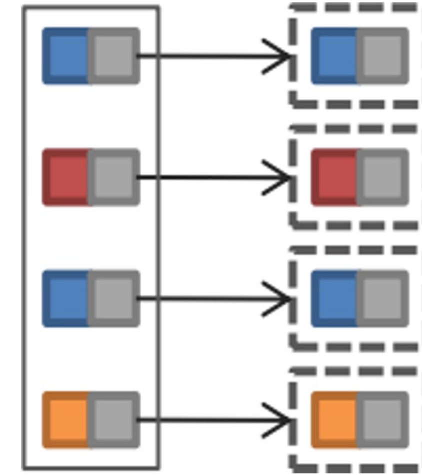
```
val nums = sc.parallelize(Array(1, 2, 3))  val even =
nums.filter(x => x % 2 == 0)
// 2
```

```
val words = sc.parallelize("this it easy, this is a test".split(" "))  val distinctWords =
words.distinct()
// a, this, is, easy,, test, it
```

```
def startsWithT(individual:String) = { individual.startsWith("t") }  val tWordList =
words.filter(word => startsWithT(word))
// this, test
```

Generic RDD Transformations

- Map: All pairs are independently processed.



```
// passing each element through a  function.
val nums =  sc.parallelize(Array(1, 2, 3))
val squares =  nums.map(x  =>  x *  x) // {1, 4, 9}

val words = sc.parallelize("take it easy, this is a test".split(" "))  val tWords = words.map(word => (word,
word.startsWith("t")))
// (take,true), (it,false), (easy,,false), (this,true), (is,false), (a,false), (test,true)
```

Key-Value RDD Transformations

- In a (k, v) pairs, k is is the key, and v is the value.
- To make a key-value RDD:
  - map over your current RDD to a basic key-value structure.
  - Use the keyBy to create a key from the current value.
  - Use the zip to zip together two RDD.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))  val keyword1 =
words.map(word => (word, 1))
// (take,1), (it,1), (easy,,1), (this,1), (is,1), (a,1), (test,1)

val keyword2  =  words.keyBy(word =>  word.toSeq(0).toString)
// (t,take), (i,it), (e,easy,), (t,this), (i,is), (a,a), (t,test)


val numRange = sc.parallelize(0 to 6)  val
keyword3 = words.zip(numRange)
// (take,0), (it,1), (easy,,2), (this,3), (is,4), (a,5), (test,6)\
```

Key-Value RDD Transformations

- keys and values extract keys and values, respectively.
- lookup looks up the values for a particular key with an RDD.
- mapValues maps over values.

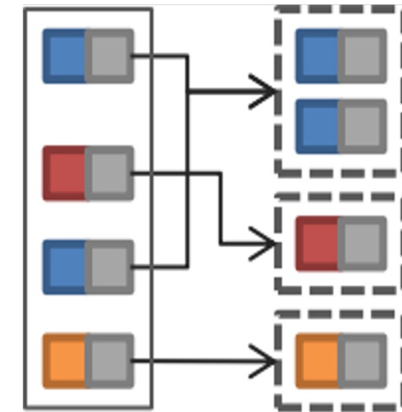
```
val words = sc.parallelize("take it easy, this is a test".split(" "))  val keyword =
words.keyBy(word => word.toLowerCase.toSeq(0).toString)
// (t,take), (i,it), (e,easy,), (t,this), (i,is), (a,a), (t,test)
val k =  keyword.keys
val v = keyword.values
```

```
val tValues =  keyword.lookup("t")
// take, this, test
```

```
val mapV =  keyword.mapValues(word  =>  word.toUpperCase)
// (t,TAKE), (i,IT), (e,EASY,), (t,THIS), (i,IS), (a,A), (t,TEST)
```

Key-Value RDD Transformations - Aggregation

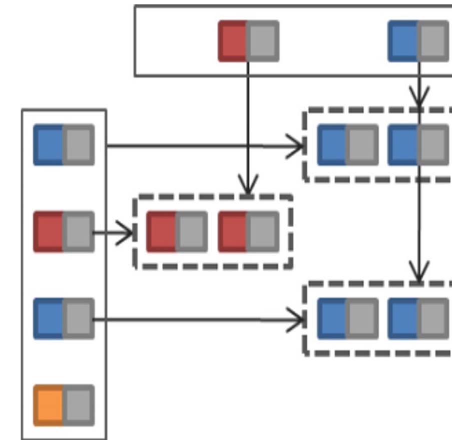
- Pairs with identical key are grouped.
  - GroupByKey
  - ReduceByKey



```
val pets =  sc.parallelize(Seq(("cat", 1), ("dog", 1), ("cat", 2)))

pets.reduceByKey((x, y) =>  x +  y)
// {(cat, 3), (dog, 1)}

pets.groupByKey()
// {(cat, (1, 2)), (dog, (1))}
```

Key-Value RDD Transformations - Join

- Performs an inner-join on the key.
- Join candidates are independently processed.



```
val visits =  sc.parallelize(Seq(("index.html", "1.2.3.4"),
                                 ("about.html", "3.4.5.6"),
                                 ("index.html", "1.3.3.1")))

val pageNames =  sc.parallelize(Seq(("index.html", "Home"), ("about.html", "About")))

visits.join(pageNames)
  // ("index.html", ("1.2.3.4", "Home"))
  // ("index.html", ("1.3.3.1", "Home"))
  // ("about.html", ("3.4.5.6", "About"))
```

Actions

Actions

- Transformations allow us to build up our logical transformation plan.

- We run an action to trigger the computation.
  - Instructs Spark to compute a result from a series of transformations

- There are three kinds of actions:
  - Actions to view data in the console
  - Actions to collect data to native objects in the respective language
  - Actions to write to output data sources

Basic RDD Actions (1/2)

- Collect: Returns all the elements of the RDD as an array.

```
val nums =  sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Take: Returns an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- Count: Returns the number of elements in the RDD.

```
nums.count() // 3
```

Basic RDD Actions (2/2)

- reduce: Aggregates the elements of the RDD using the given function.

```
val nums =  sc.parallelize(Array(1, 2, 3))
nums.reduce((x, y) => x + y)  or nums.reduce(_ +  _) // 6
```

- saveAsTextFile Writes the elements of the RDD as a text file.

```
nums.saveAsTextFile("hdfs://file.txt")
```

- max and min return the maximum and minimum values, respectively.

```
val nums =  sc.parallelize(1 to 20)
val maxValue  =  nums.max() // 20
val minValue =  nums.min()// 1
```

Caching

- When you cache an RDD, each node stores any partitions of it that it computes in memory.
- An RDD that is not cached is re-evaluated each time an action is invoked on that RDD.
- A node reuses the cached RDD in other actions on that dataset.
- There are two functions for caching an RDD:
  - cache caches the RDD into memory
  - persist(level) can cache in memory, on disk, or off-heap memory

```
val words = sc.parallelize("take it easy, this is a test".split(" "))

words.cache()
```

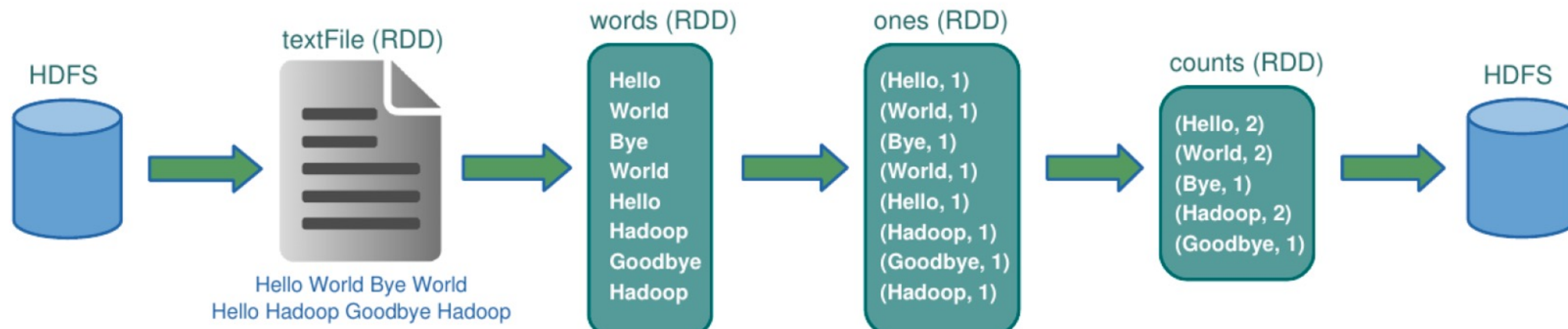
Checkpointing

- checkpoint saves an RDD to disk.
- When we reference a checkpointed RDD, it will derive from the checkpoint instead of the source data.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))

sc.setCheckpointDir("/path/checkpointing")
words.checkpoint()
```

Example 1: Word Count

```
sc = SparkContext( "local", "WordCount")

text_file = sc.textFile(”hdfs://…")
words = text_file.flatMap(lambda line: line.split())
key_values= words.map(lambda word: (word, 1))
reduced= key_values.reduceByKey(lambda x, y: x + y)
reduced.saveAsTextFile (”hdfs://…")
sc.stop()
```



Connect to the cluster

- Create an instance of SparkContext class to connect to the cluster.

```
sc= SparkContext( “local”,”Word_Count” )
```

The type of cluster manager. It can be Mesos, Yarn, Kubernetes or local.

Creating RDD

- Load text file from local HDFS, or any other external source.
- Turn the collection into an RDD

```
text_file = sc.textFile(”hdfs://inputfile.txt")
```

Transformations

- flatmMap splits all words in an RDD

```
words = text_file.flatMap(lambda line: line.split())
```

- map ceates a key-value pair for each word.

```
word_counts = words.map(lambda word: (word, 1))
```

- reduceByKey reduces by key within each partition.

```
word_counts = word_counts.reduceByKey(lambda x, y: x + y)
```

Action

- saveAsTextFile: Writes the elements of the RDD as a text file.

```
word_counts.saveAsTextFile (”hdfs://…")
```

Stop the cluster

- sc.stop: Terminates the SparkContext and releases all associated resources

```
sc.stop()
```

Example 1: Word Count

```
sc = SparkContext( "local", "WordCount")

text_file = sc.textFile(”hdfs://…")
words = text_file.flatMap(lambda line: line.split())
key_values= words.map(lambda word: (word, 1))
reduced= key_values.reduceByKey(lambda x, y: x + y)
reduced.saveAsTextFile (”hdfs://…")
sc.stop()
```

Spark runs this program on the same workload 170 time faster than MapReduce.

Example2: K-mean Clustering

- Training a model on a local machine.

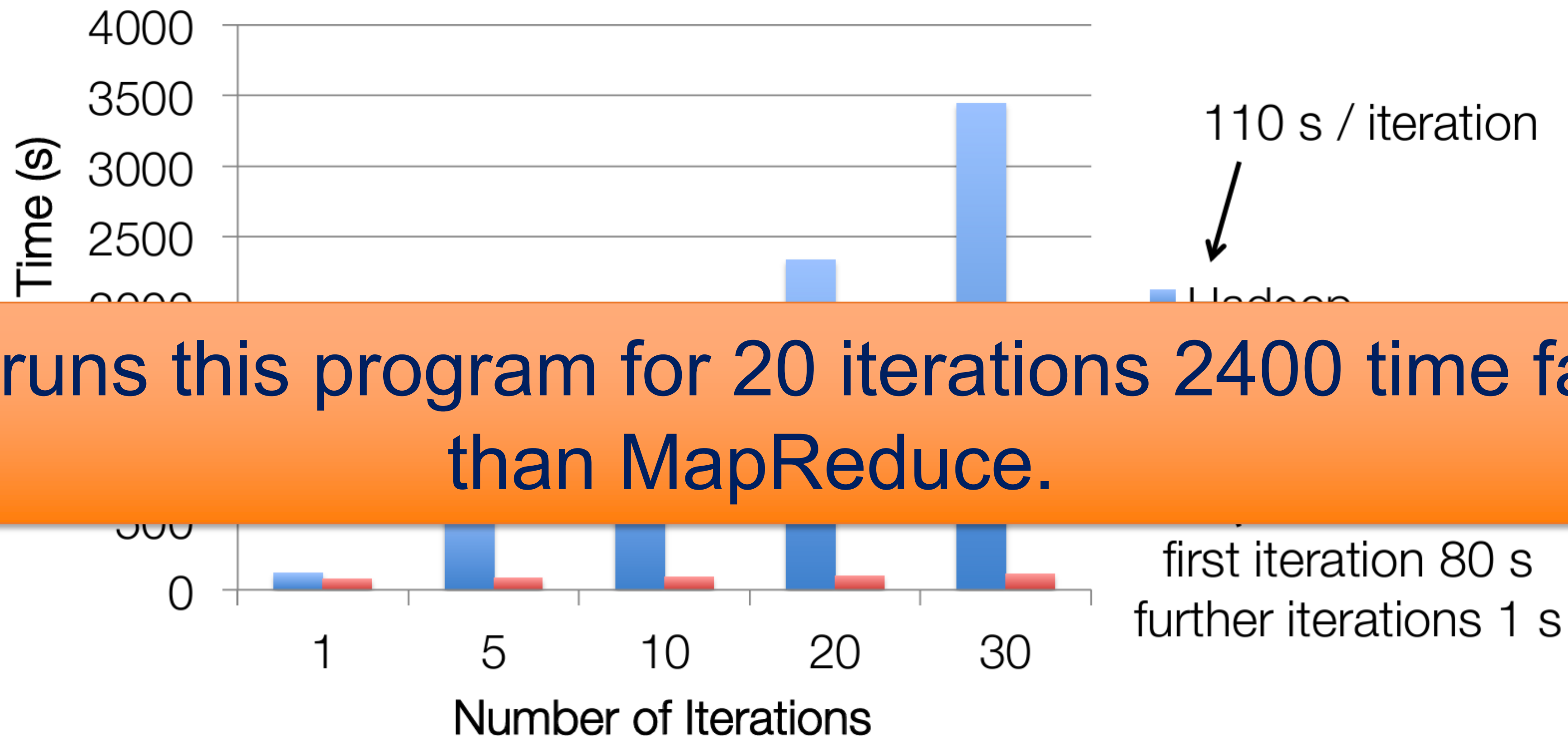
```
from sklearn.cluster import KMeans
import numpy as np

# Load data from file
data = np.loadtxt( "kmean_data.txt” )

# Train KMeans model
kmeans = KMeans(n_clusters=2,
max_iter=20)
clusters = kmeans.fit(data)

# Compute cost
cost = clusters.inertia_
print("Sum of squared errors =", cost)
```

K-mean Clustering – Local Vs Distributed

```
from sklearn.cluster import KMeans
import numpy as np


# Load data from file
data = np.loadtxt( "kmean_data.txt” )




# Train KMeans model
kmeans = KMeans(n_clusters=2,
max_iter=20)
clusters = kmeans.fit(data)

# Compute cost
cost = clusters.inertia_
print("Sum of squared errors =", cost)
```

```
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans, KMeansModel
# Initialize SparkContext
 sc = SparkContext( "local", "KMeans Inference” )

# Load data from file
data = sc.textFile( "kmean_data.txt” )

# Parse the data and convert it to RDD of vectors
parsedData = data.map(lambda line: [float(x) for x in line.split(' ')]).cache()

# Train KMeans model
clusters = KMeans.train(parsedData, 2, maxIterations=20)


# Compute cost
cost = clusters.computeCost(parsedData)
print("Sum of squared errors =", cost)
sc.stop()
```

Example 2: K-Mean Clustering



Spark runs this program for 20 iterations 2400 time faster than MapReduce.

100 GB of data on 50 m1.xlarge EC2 machines

Example 3: Earthquake Detection

Earthquake Shakes Twitter Users: Real-time Event Detection by Social Sensors

Takeshi Sakaki
The University of Tokyo
Yayoi 2-11-16, Bunkyo-ku
Tokyo, Japan
sakaki@biz-model.t.u-tokyo.ac.jp

Makoto Okazaki
The University of Tokyo
Yayoi 2-11-16, Bunkyo-ku
Tokyo, Japan
m_okazaki@biz-model.t.u-tokyo.ac.jp

Yutaka Matsuo
The University of Tokyo
Yayoi 2-11-16, Bunkyo-ku
Tokyo, Japan
matsuo@biz-model.t.u-tokyo.ac.jp

ABSTRACT

Twitter, a popular microblogging service, has received much attention recently. An important characteristic of Twitter is its real-time nature. For example, when an earthquake occurs, people make many Twitter posts (*tweets*) related to the earthquake, which enables detection of earthquake occurrence promptly, simply by observing the tweets. As described in this paper, we investigate the real-time interaction of events such as earthquakes, in Twitter, and pro-

struct an earthquake reporting system in Japan. Because of the numerous earthquakes and the large number of Twitter users throughout the country, we can detect an earthquake by monitoring tweets with high probability (96% of earthquakes of Japan Meteorological Agency (JMA) seismic intensity scale 3 or more are detected). Our system detects earthquakes promptly and sends e-mails to registered users. Notification is delivered much faster than the announcements that are broadcast by the JMA.

currently estimated as 44.5 million worldwide[1]. Monthly growth of users has been 1382% year-on-year, which makes Twitter one of the fastest-growing sites in the world[2].

Some studies have investigated Twitter: Java et al. analyzed Twitter as early as 2007. They described the social network of Twitter users and investigated the motivation of Twitter users [13]. B. Huberman et al. analyzed more than 300 thousand users. They discovered that the relation between friends (defined as a person to whom a user has

microblogging, and Plurk has a timeline view integrating video and picture sharing. Although our study is applicable to other microblogging services, in this study, we specifically examine Twitter because of its popularity and data volume.

An important common characteristic among microblogging services is its real-time nature. Although blog users typically update their blogs once every several days, Twitter users write tweets several times in a single day. Users can know how other users are doing and often what they are thinking about *now*, users repeatedly return to the site and

Spark informs you of an earthquake in Japan quicker than the Japan Meteorological Agency

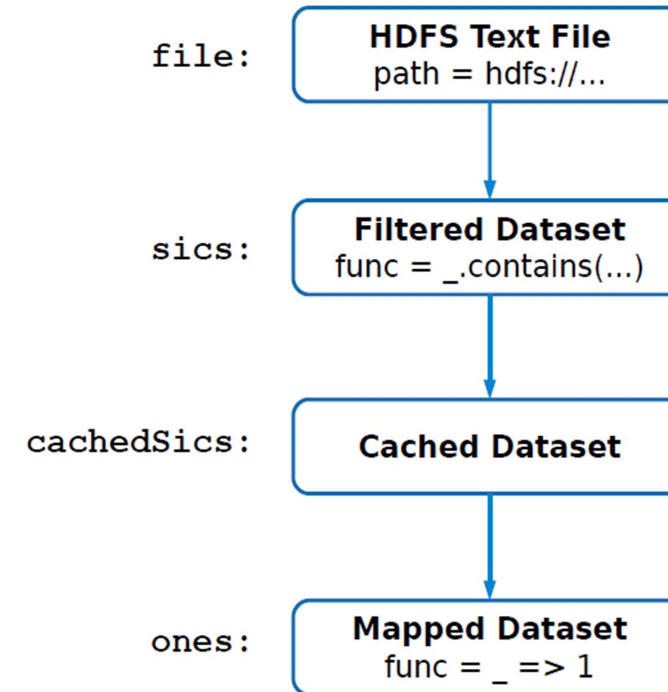
Spark Execution Engine

Job Execution Workflow

1. Lineage Creation
2. Dependence and Stage Identification
3. Task Scheduling
4. Shuffle
5. Execution

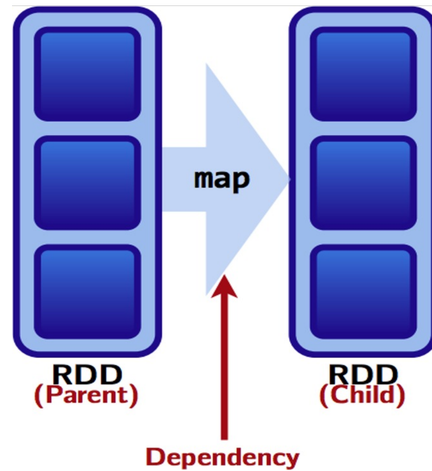
Lineage

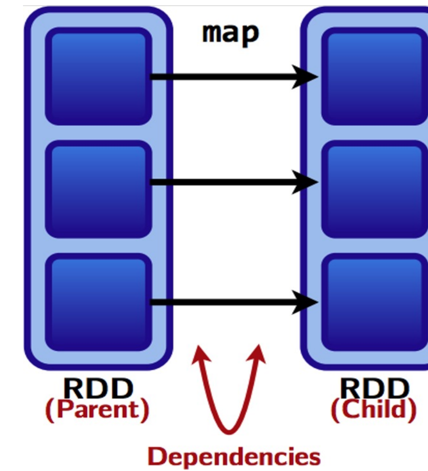
- A DAG representing the computations done on the RDD is called lineage graph.
- RDDs are stored as a chain of objects capturing the lineage of each RDD.



RDD Dependencies

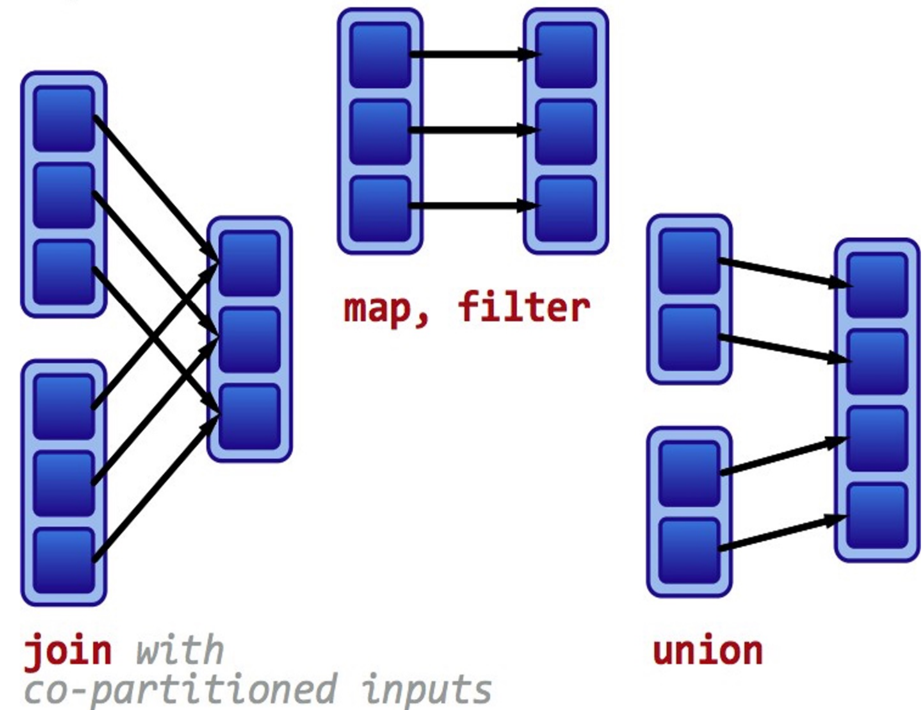
- RDD dependencies encode when data must move across network.
  - Narrow dependencies
  - Wide dependencies





RDD Dependencies - Narrow

- Narrow: each partition of a parent RDD is used by at most one partition of the child RDD.
- Narrow dependencies allow pipelined execution on one cluster node: a map followed by a filter.

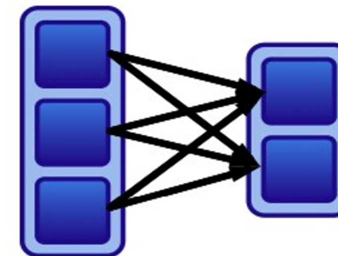


RDD Dependencies - Wide

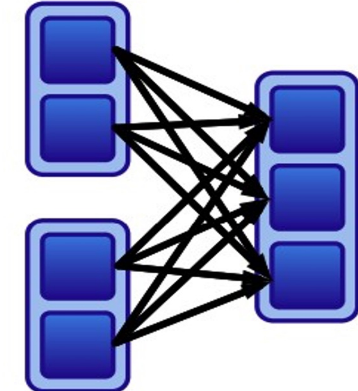
- Wide: each partition of a parent RDD is used by multiple partitions of the child RDDs.

Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.



groupByKey



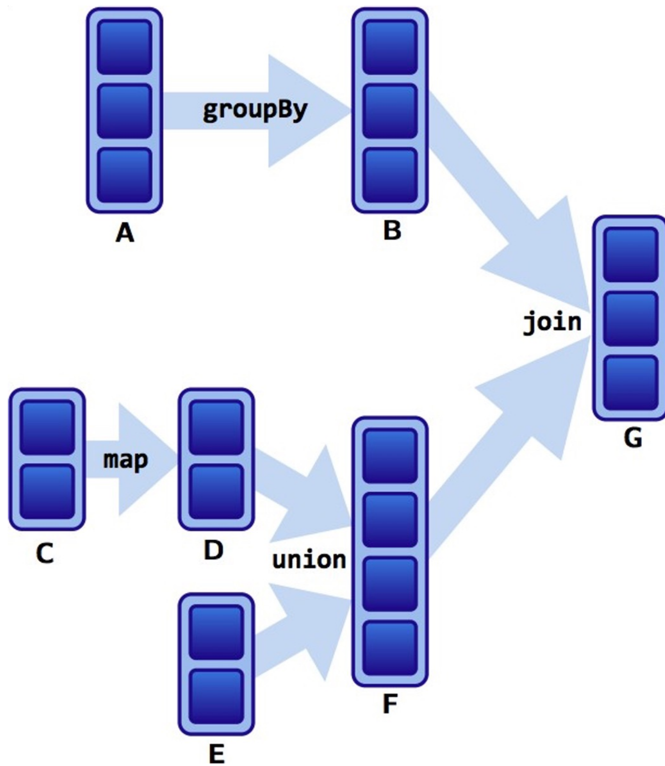
join
with inputs not co-partitioned

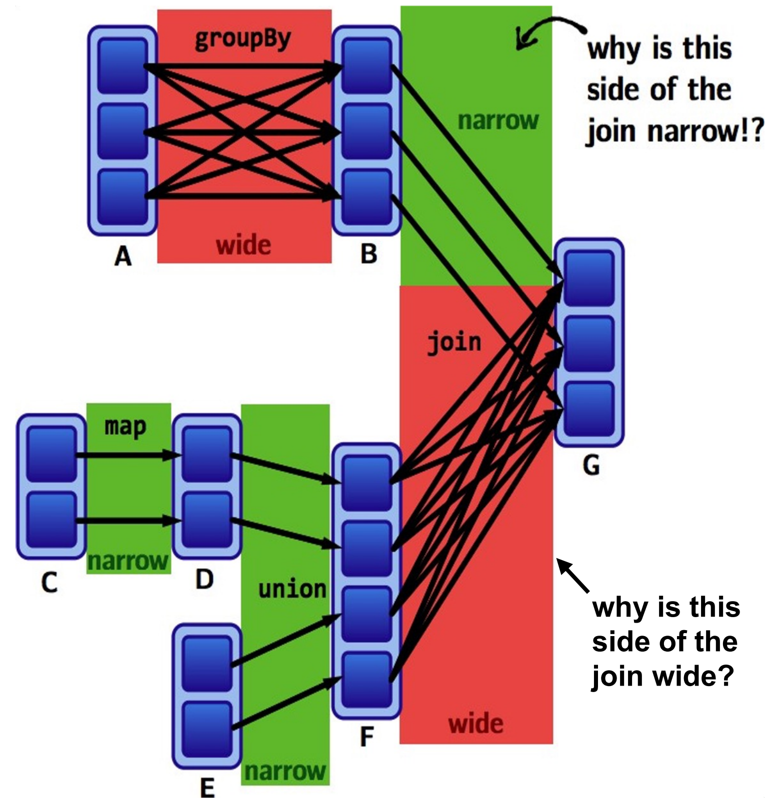
Stage Identification

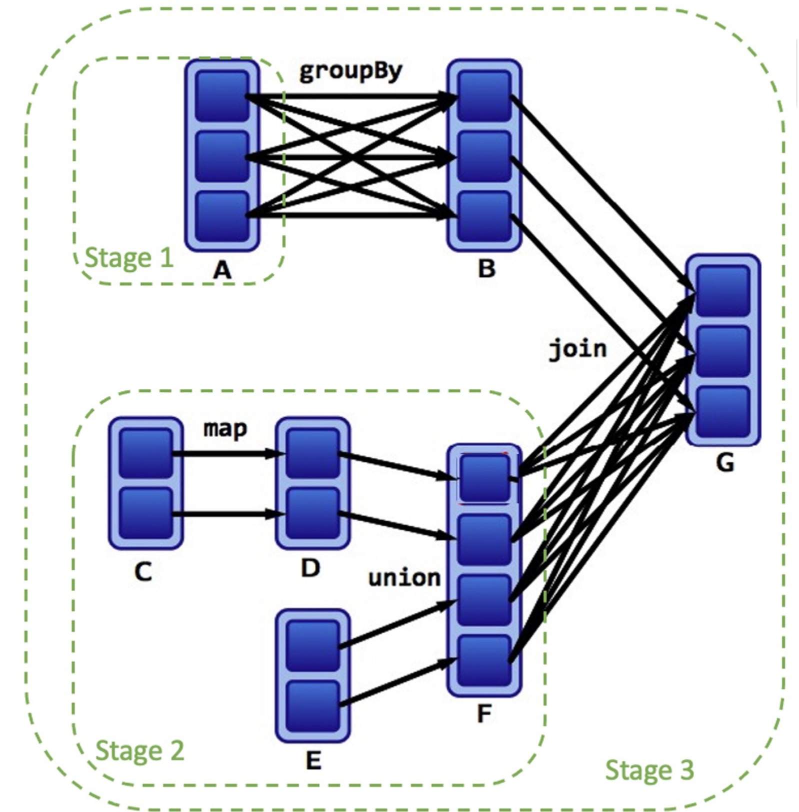
- A stage in Spark represents a set of transformations that can be executed together, given the dependencies between RDDs.

- Stages are the units of execution in a Spark job, and they play a crucial role in the overall job execution process.

Stage Identification - Example







“Join” Dependence type

- Data Shuffling typically happens in wide dependencies.
- The join operation typically involves data shuffling, which is necessary to align and combine data from different partitions.
- In specific scenarios, join has narrow dependence.

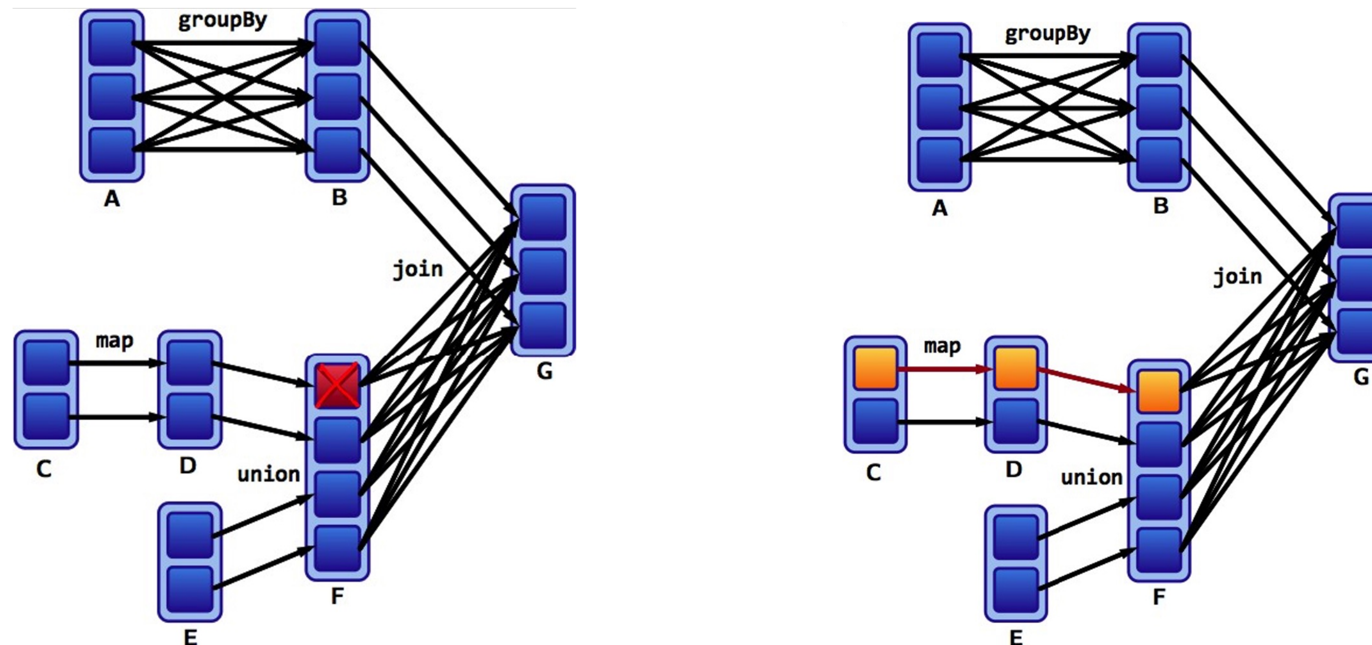
```
val rdd1 = sc.parallelize(Array((1, "apple"), (2, "banana"), (3, "cherry")))
val rdd2 = sc.parallelize(Array((1, "red"), (2, "yellow"), (3, "purple")))

// Assume rdd1 and rdd2 have the same partitioning and ordering of keys.

// Perform a "join" without shuffling, which essentially combines the values of the same-key elements.
val joinedRDD = rdd1.join(rdd2)

joinedRDD.collect().foreach(println)
```

Fault Tolerance

- No replication – Automatically rebuilt on failures.
- Lineages are the key to fault tolerance in Spark.
- Recompute only the lost RDD.

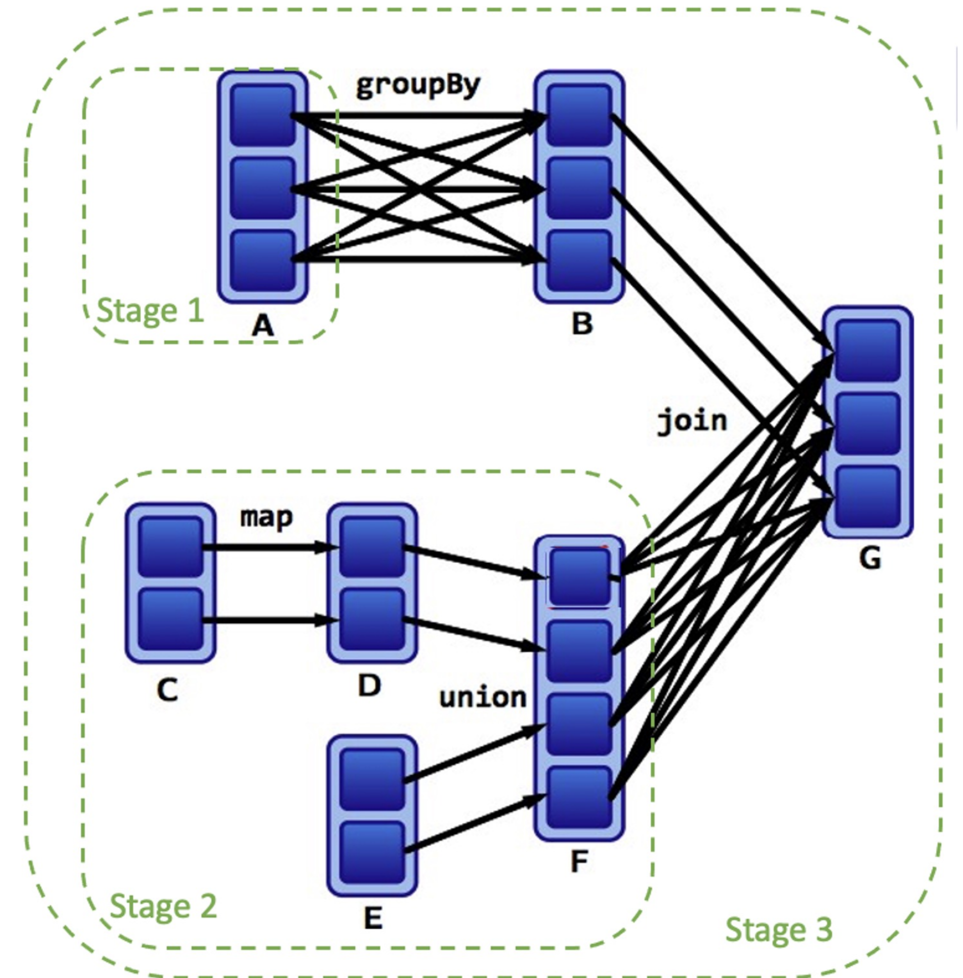
Fault Tolerance

- Assume one of the partitions fails.
- We only have to recompute the data shown below to get back on track.



Job Scheduling

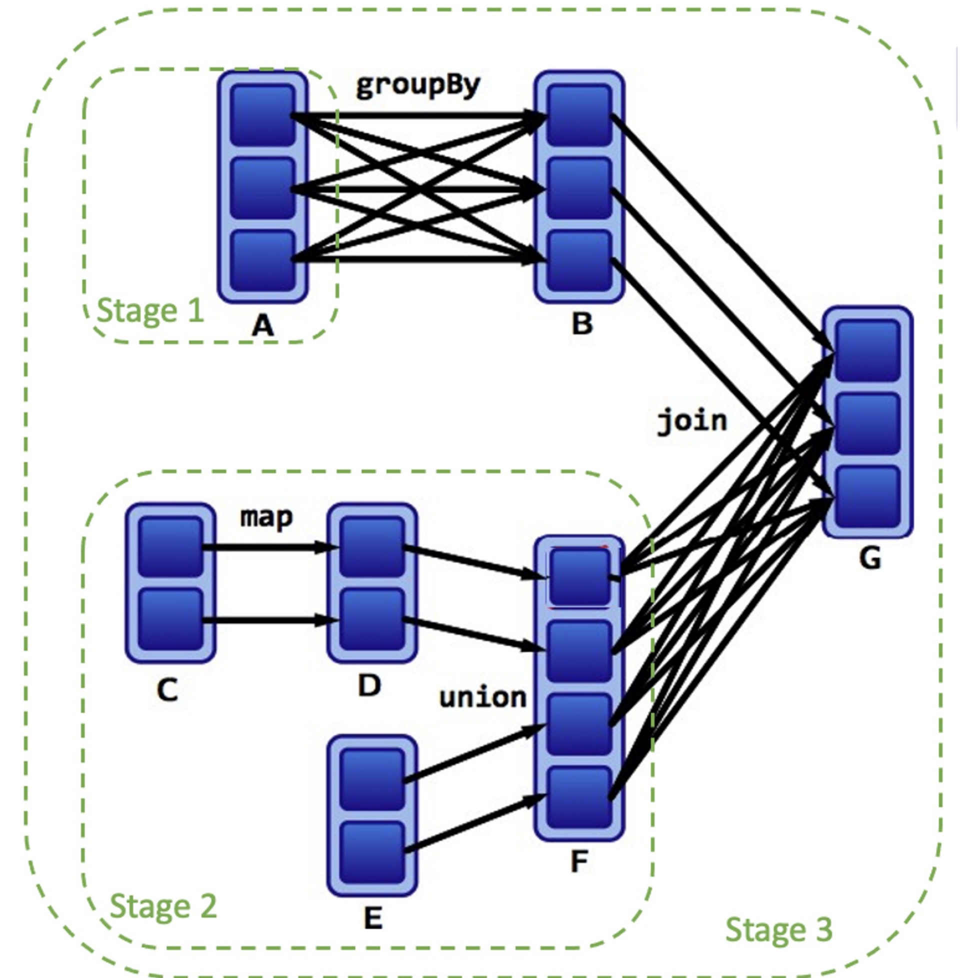
- When a user runs an action on an RDD: the scheduler builds a DAG of stages from the RDD lineage graph.
- A stage contains as many pipelined transformations with narrow dependencies.
- The boundary of a stage:
  - Shuffles for wide dependencies.
  - Already computed partitions.



Job Scheduling

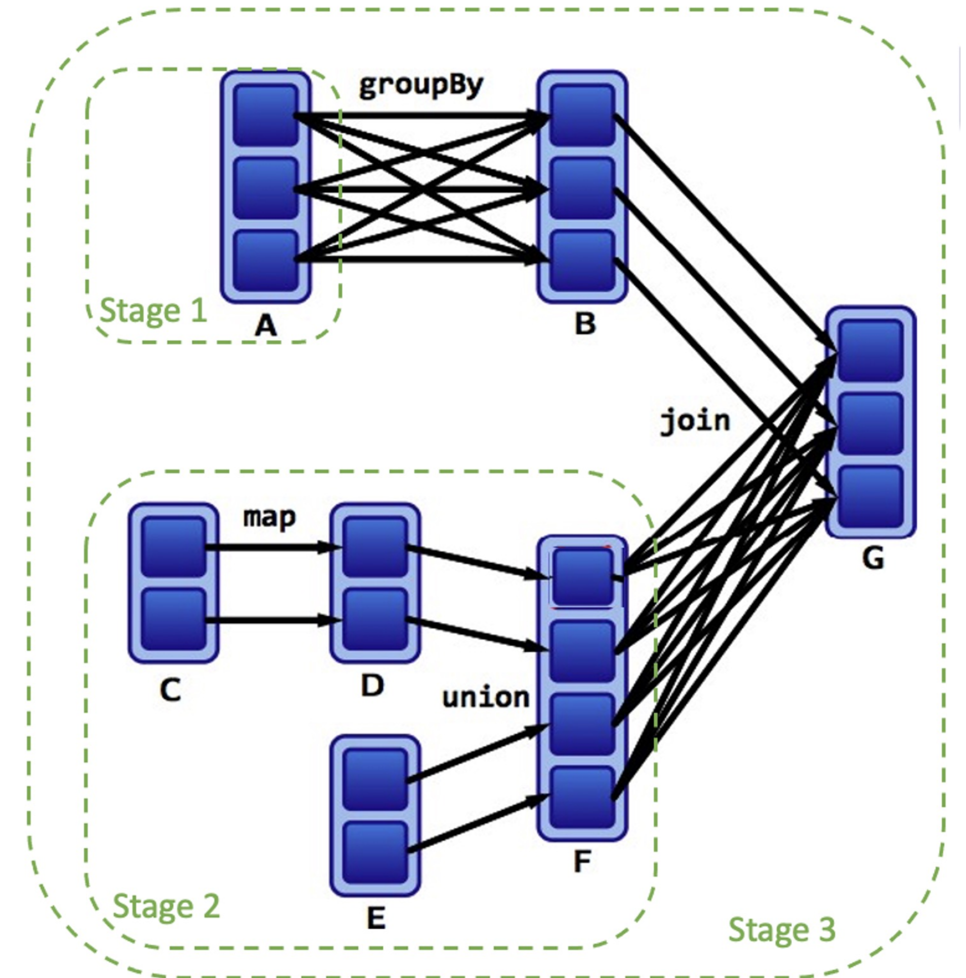
- The scheduler launches tasks to compute missing partitions from each stage until it computes the target RDD.

- Tasks are assigned to machines based on data locality.
  - If a task needs a partition, which is available in the memory of a node, the task is sent to that node.



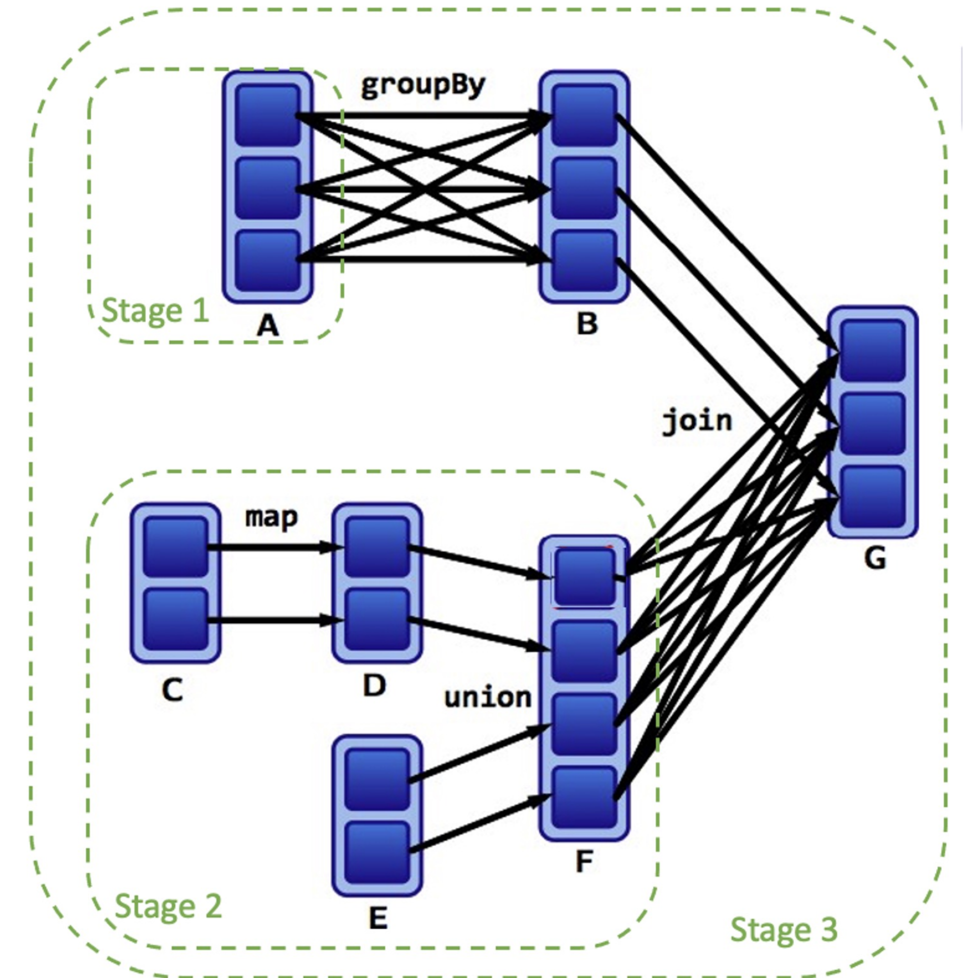
Shuffling

- The boundary of a stage shuffles for wide dependencies.



Task Execution

- Tasks within a stage are executed.
- Results are passed to subsequent stages.



Spark Use Cases

- Applications suitable for Spark
  - Batch applications that apply the same operation to all elements of a dataset.
    - Uber: Real-time Data Processing for Dynamic Pricing
    - Netflix: Personalized Content Recommendations

Applications not suitable for Spark

- Applications that make asynchronous fine-grained updates to shared state, e.g., storage system for a web application.
- Applications that work on small scale datasets

Recap

- RDD: a distributed memory abstraction that is both fault tolerant and efficient
- Two types of operations: Transformations and Actions.
- RDD fault tolerance: Lineage
- Spark execution flow

Assignment 3

- Assignment 3 is out now!
- Due date: Mar 11th, at 11:59 pm

Next class:

Structured Data Processing