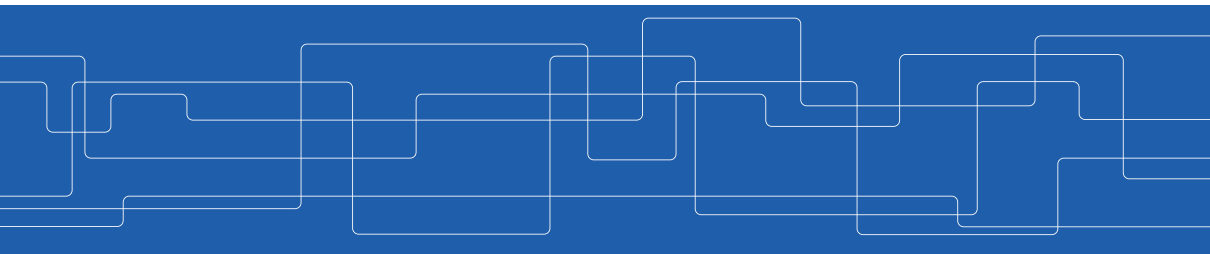




# Parallel Processing - Spark

Amir H. Payberah  
payberah@kth.se  
2021-09-14



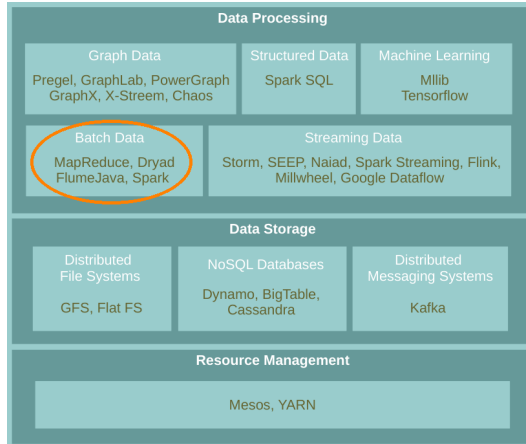


## The Course Web Page

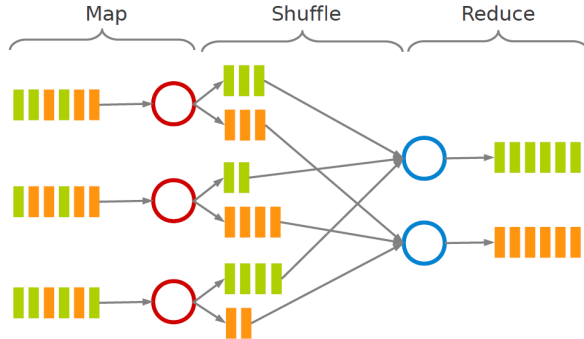
`https://id2221kth.github.io`

`https://tinyurl.com/f6x544h`

# Where Are We?

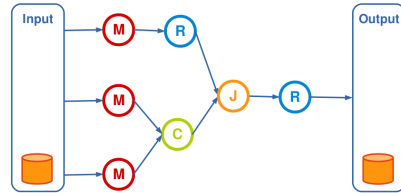
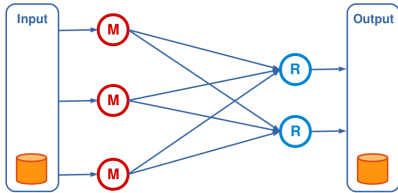


# MapReduce Reminder



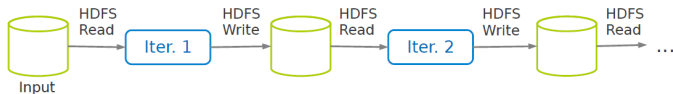
# Motivation (1/2)

- **Acyclic data flow** from stable storage to stable storage.

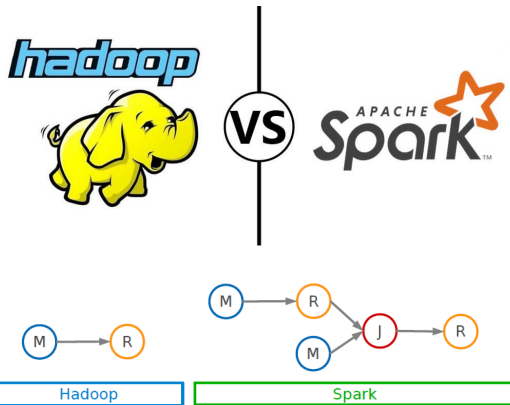


## Motivation (2/2)

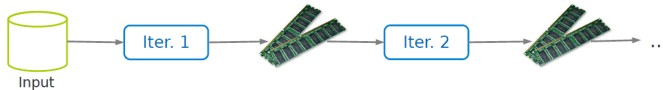
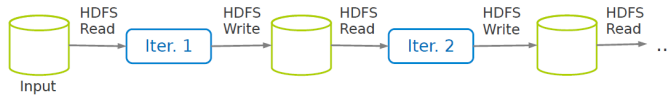
- ▶ MapReduce is **expensive** (**slow**), i.e., always goes to disk and **HDFS**.



# So, Let's Use Spark

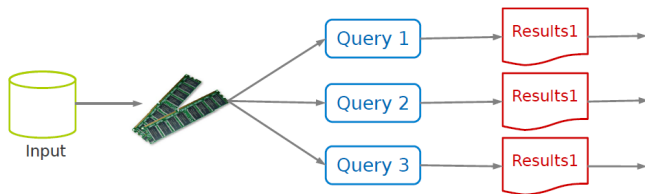
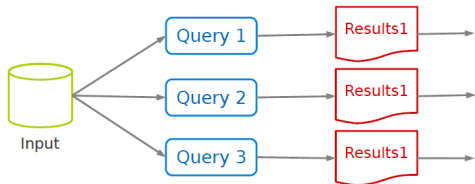


# Spark vs. MapReduce (1/2)





## Spark vs. MapReduce (2/2)

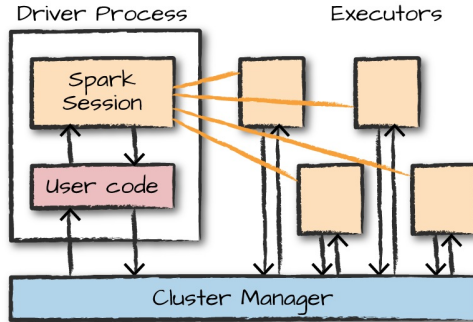




# Spark Application

# Spark Applications Architecture

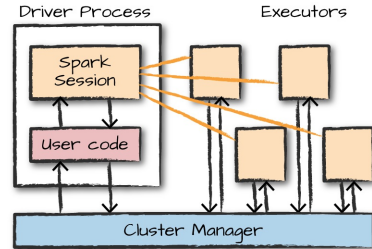
- ▶ Spark applications consist of
  - A driver process
  - A set of executor processes



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

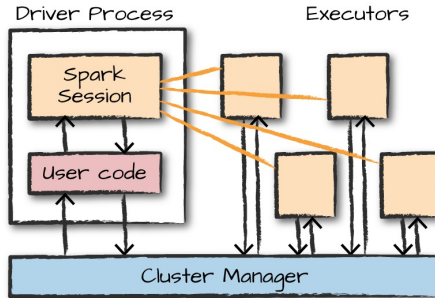
# Driver Process

- ▶ The **heart** of a **Spark application**
- ▶ Sits on a **node** in the cluster
- ▶ Runs the **main()** function
- ▶ Responsible for **three** things:
  - **Maintaining information** about the Spark application
  - **Responding to a user's program or input**
  - **Analyzing, distributing, and scheduling** work across the **executors**



# Executors

- ▶ Responsible for **two** things:
  - **Executing code** assigned to it by the **driver**
  - **Reporting the state** of the computation on that executor back to the **driver**





# SparkSession

- ▶ A **driver process** that controls a **Spark application**.
- ▶ **Main entry** point to Spark functionality.
- ▶ A **one-to-one correspondence** between a **SparkSession** and a **Spark application**.
- ▶ Available in **console** shell as **spark**.

```
SparkSession.builder.master(master).appName(appName).getOrCreate()
```



# SparkContext

- ▶ The entry point for **low-level API** functionality.
- ▶ You **access it** through the **SparkSession**.
- ▶ You can access a **SparkContext** via `spark.sparkContext`.
- ▶ Available in **console** shell as `sc`.

```
val conf = new SparkConf().setMaster(master).setAppName(appName)
new SparkContext(conf)
```



## SparkSession vs. SparkContext

- ▶ Prior to `Spark 2.0.0`, a the `spark driver` program uses `SparkContext` to connect to the cluster.
- ▶ In order to use APIs of `SQL`, `Hive` and `streaming`, `separate SparkContexts` should to be created.
- ▶ `SparkSession` provides access to all the spark functionalities that `SparkContext` does, e.g., `SQL`, `Hive` and `streaming`.
- ▶ `SparkSession` internally has a `SparkContext` for actual computation.

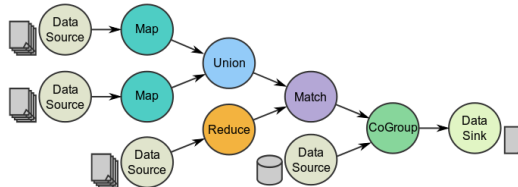




# Programming Model

# Spark Programming Model

- ▶ **Job** is described based on **directed acyclic graphs (DAG)** **data flow**.
- ▶ A **data flow** is composed of any number of **data sources**, **operators**, and **data sinks** by connecting their inputs and outputs.
- ▶ **Parallelizable operators**





## Resilient Distributed Datasets (RDD) (1/3)

- ▶ A distributed memory abstraction.
- ▶ Immutable collections of objects spread across a cluster.
  - Like a `LinkedList <MyObjects>`



## Resilient Distributed Datasets (RDD) (2/3)

- ▶ An **RDD** is divided into a number of **partitions**, which are **atomic** pieces of information.
- ▶ **Partitions** of an RDD can be stored on **different nodes** of a cluster.





## Resilient Distributed Datasets (RDD) (3/3)

- ▶ RDDs were the primary API in the [Spark 1.x](#) series.
- ▶ They are **not commonly used** in the [Spark 2.x](#) series.
- ▶ Virtually all Spark code you run, [compiles down to an RDD](#).



## Types of RDDs

- ▶ Two types of RDDs:
  - Generic RDD
  - Key-value RDD
  
- ▶ Both represent a collection of objects.
  
- ▶ Key-value RDDs have special operations, such as aggregation, and a concept of custom partitioning by key.



## When To Use RDDs?

- ▶ **Short answer:** you **should not manually** create RDDs unless you have a very **specific reason**.
- ▶ They are a much **lower-level API** that provides a lot of power.
- ▶ But, **lack of the optimizations** that are available in the Structured APIs.
- ▶ The **most likely reason to use RDDs:** **custom partitioning of data**.
  - **Fine-grained control** over the physical distribution of data.

# Creating RDDs





## Creating RDDs - Parallelized Collections

- ▶ Use the `parallelize` method on a `SparkContext`.
- ▶ This turns a `single node` collection into a `parallel` collection.
- ▶ You can also explicitly state the `number of partitions`.
- ▶ In the console shell, you can either use `sc` or `spark.sparkContext`

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

```
val wordsCollection = "take it easy, this is a test".split(" ")
val words = spark.sparkContext.parallelize(wordsCollection, 2)
```



## Creating RDDs - External Datasets

- ▶ Create RDD from an **external storage**.
  - E.g., **local file system**, **HDFS**, **Cassandra**, **HBase**, **Amazon S3**, etc.
- ▶ Text file RDDs can be created using **textFile** method.

```
val myFile1 = sc.textFile("file.txt")  
val myFile2 = sc.textFile("hdfs://namenode:9000/path/file")
```



# RDD Operations



# RDD Operations

- ▶ RDDs support **two** types of operations:
  - **Transformations**: allow us to **build the logical plan**
  - **Actions**: allow us to **trigger the computation**

# Transformations



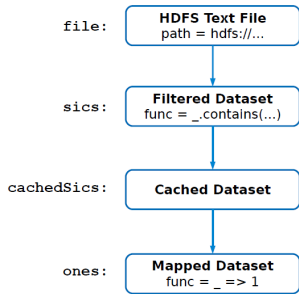
# Transformations

- ▶ 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**.



# Lineage

- ▶ **Lineage:** transformations used to build an RDD.
- ▶ **RDDs** are stored as a chain of objects capturing the **lineage** of each RDD.



```
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```



## Generic RDD Transformations (1/3)

- ▶ `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
```

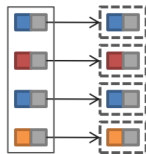
```
val words = sc.parallelize("this it easy, this is a test".split(" "))
val distinctWords = words.distinct()
// a, this, is, easy,, test, it
```

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



## Generic RDD Transformations (2/3)

- ▶ `map` and `flatMap` apply a given function on each RDD record **independently**.



```
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)
```



## Generic RDD Transformations (3/3)

- ▶ `sortBy` sorts an RDD records.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))  
  
val sortedWords = words.sortBy(word => word.length())  
// a, it, is, take, this, test, easy,
```



## Key-Value RDD Transformations - Basics (1/2)

- ▶ In a  $(k, v)$  pairs,  $k$  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 - Basics (2/2)

- ▶ `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)
```

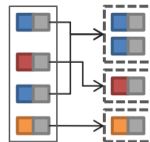
```
val k = keyword.keys
val v = keyword.values
```

```
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 (1/2)

- ▶ Aggregate the values associated with each key.



```
val kvChars = ...
// (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,1), ...

val grpChar = kvChars.groupByKey().map(row => (row._1, row._2.reduce(addFunc)))
// (t,5), (h,1), (,1), (e,3), (a,3), (i,3), (y,1), (s,4), (k,1))
```

```
def addFunc(left:Int, right:Int) = left + right
val redChar = kvChars.reduceByKey(addFunc)
// (t,5), (h,1), (,1), (e,3), (a,3), (i,3), (y,1), (s,4), (k,1))
```

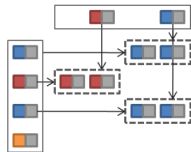


## Key-Value RDD Transformations - Aggregation (2/2)

- ▶ `groupByKey` or `reduceByKey`?
- ▶ In `groupByKey`, each `executor` must hold **all values for a given key** in `memory` before applying the function to them.
  - This is problematic in massive `skewed key`.
- ▶ In `reduceByKey`, the reduce happens **within each partition**, and does not need to put everything in memory.

## Key-Value RDD Transformations - Join

- ▶ `join` performs an **inner-join** on the key.
- ▶ `fullOuterJoin`, `leftOuterJoin`, `rightOuterJoin`, and `cartesian`.



```

val keyedChars = ...
// (t,4), (h,6), (,9), (e,8), (a,3), (i,5), (y,2), (s,7), (k,0)

val kvChars = ...
// (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,,1), ...

val joinedChars = kvChars.join(keyedChars)
// (t,(1,4)), (t,(1,4)), (t,(1,4)), (t,(1,4)), (t,(1,4)), (h,(1,6)), (,(1,9)), (e,(1,8)), ...
    
```

# 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



## RDD Actions (1/6)

- ▶ `collect` returns all the elements of the RDD as an array at the driver.
- ▶ `first` returns the first value in the RDD.

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



## RDD Actions (2/6)

- ▶ `take` returns an **array** with the **first n elements** of the RDD.
- ▶ Variations on this function: `takeOrdered` and `takeSample`.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))  
  
words.take(5)  
// Array(take, it, easy,, this, is)
```

```
words.takeOrdered(5)  
// Array(a, easy,, is, it, take)  
  
val withReplacement = true  
val numberToTake = 6  
val randomSeed = 100L  
words.takeSample(withReplacement, numberToTake, randomSeed)  
// Array(take, it, test, this, test, take)
```



## RDD Actions (3/6)

- ▶ `count` returns the **number of elements** in the dataset.
- ▶ `countByValue` counts the **number of values** in a given RDD.
- ▶ `countByKey` returns a **hashmap of (K, Int)** pairs with the count of each key.
  - Only available on key-value RDDs, i.e., (K, V)

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

words.count()
// 10

words.countByValue()
// Map(this -> 1, is -> 1, it -> 2, a -> 1, easy, -> 1, test, -> 1, take -> 2, easy -> 1)
```



## RDD Actions (4/6)

- ▶ `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
```



## RDD Actions (5/6)

- ▶ **reduce** aggregates the elements of the dataset using a **given function**.
- ▶ The given function should be **commutative and associative** so that it can be computed correctly in **parallel**.

```
sc.parallelize(1 to 20).reduce(_ + _)
// 210

def wordLengthReducer(leftWord:String, rightWord:String): String = {
  if (leftWord.length > rightWord.length)
    return leftWord
  else
    return rightWord
}

words.reduce(wordLengthReducer)
// easy,
```



## RDD Actions (6/6)

- ▶ `saveAsTextFile` writes the elements of an RDD as a **text file**.
  - Local filesystem, HDFS or any other Hadoop-supported file system.
- ▶ `saveAsObjectFile` explicitly writes **key-value pairs**.

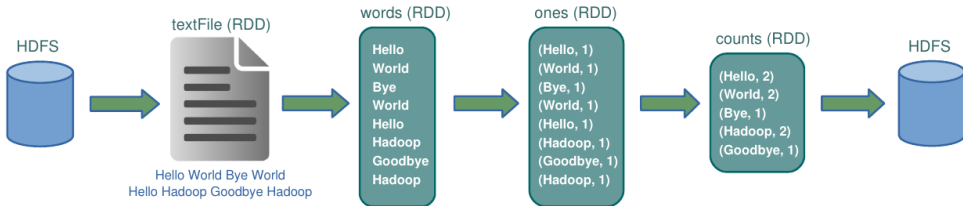
```
val words = sc.parallelize("take it easy, this is a test".split(" "))  
words.saveAsTextFile("file:/tmp/words")
```

# Example

```
val textFile = sc.textFile("hdfs://...")

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```





# Cache and Checkpoints



# 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`.
- ▶ Checkpointed data is `not removed` after `SparkContext` is destroyed.
- ▶ 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()
```



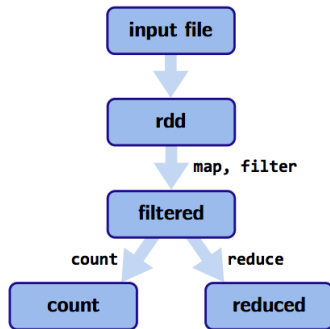
# Execution Engine



## More About Lineage

- ▶ A DAG representing the **computations** done on the RDD is called **lineage graph**.

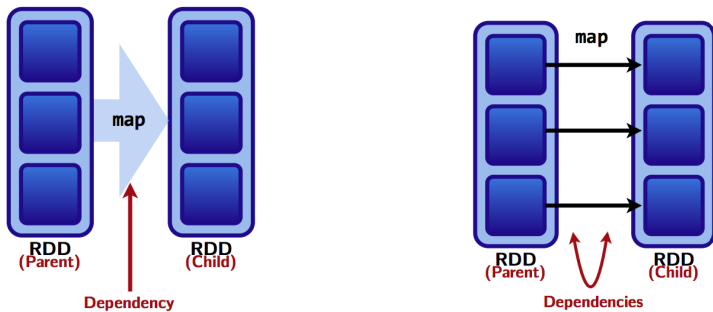
```
val rdd = sc.textFile(...)
val filtered = rdd.map(...).filter(...).persist()
val count = filtered.count()
val reduced = filtered.reduce()
```



[<https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies>]

# Dependencies

- ▶ RDD dependencies encode when data must **move across network**.



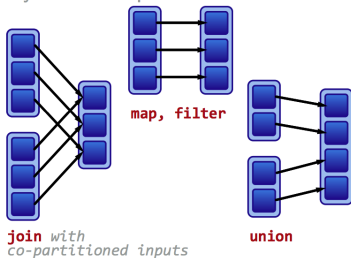
[<https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies>]

## Two Types of Dependencies (1/2)

- ▶ **Narrow** transformations (dependencies)
  - Each **input partition** will contribute to **only one output partition**.
  - With narrow transformations, Spark can perform a **pipelining**

### Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.



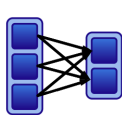
[<https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies>]

## Two Types of Dependencies (2/2)

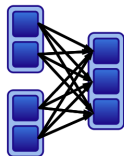
- ▶ **Wide** transformations (dependencies)
  - Each **input partition** will contribute to **many output partition**.
  - Usually referred to as a **shuffle**

### Wide dependencies:

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



**groupByKey**



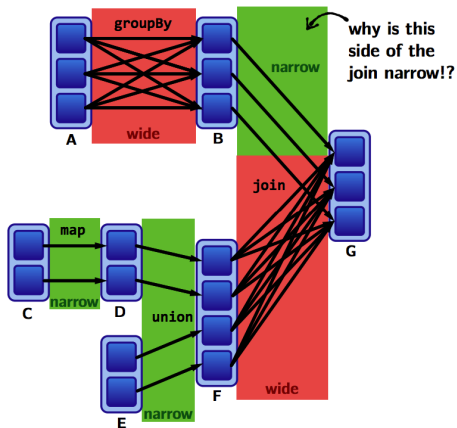
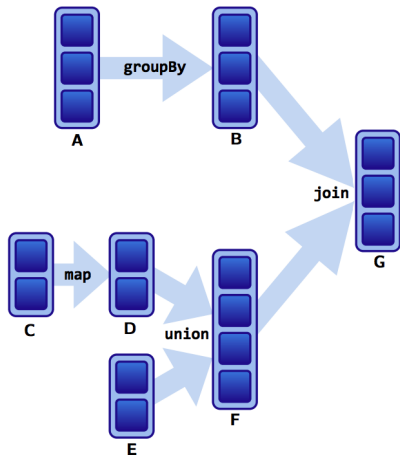
**join**

*with inputs not  
co-partitioned*

[<https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies>]



# Example



[<https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies>]

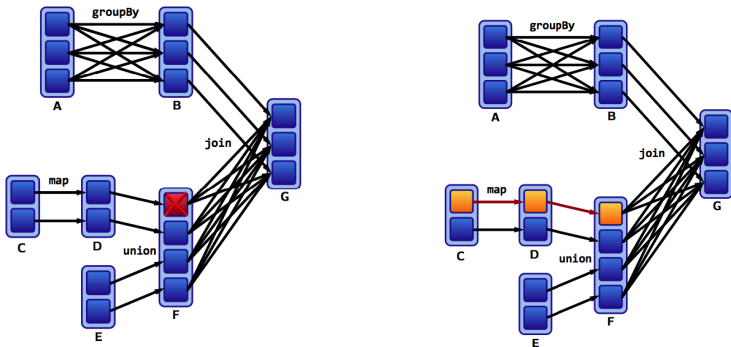


## Lineages and Fault Tolerance (1/2)

- ▶ No replication.
- ▶ Lineages are the key to fault tolerance in Spark.
- ▶ Recompute only the lost partitions of an RDD.

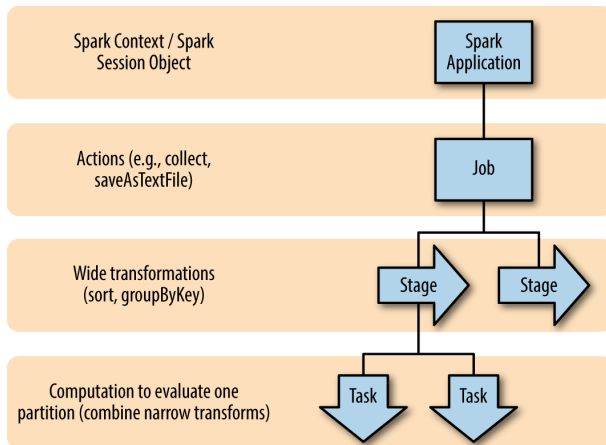
# Lineages and Fault Tolerance (2/2)

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



[<https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies>]

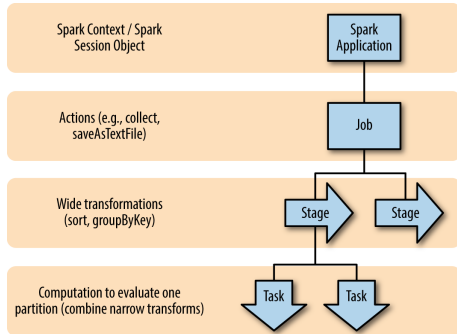
# The Anatomy of a Spark Job



[H. Karau et al., High Performance Spark, O'Reilly Media, 2017]

# Jobs

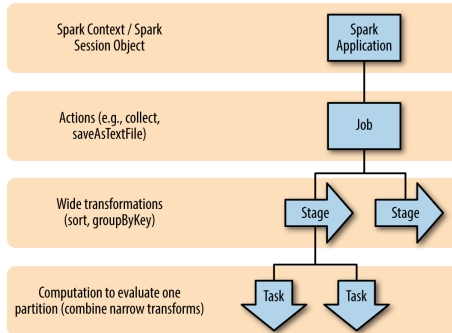
- ▶ A Spark **job** is the **highest** element of Spark's **execution hierarchy**.
  - Each Spark **job** corresponds to one **action**.
  - Each **action** is called by the **driver** program of a Spark application.



[H. Karau et al., High Performance Spark, O'Reilly Media, 2017]

# Stages

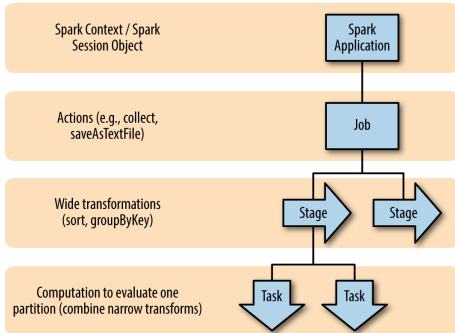
- ▶ Each **job** breaks down into a **series** of **stages**.
  - **Stages** in Spark represent **groups of tasks** that can be **executed together**.
  - **Wide transformations** define the **breakdown of jobs into stages**.



[H. Karau et al., High Performance Spark, O'Reilly Media, 2017]

# Tasks

- ▶ A **stage** consists of **tasks**, which are the **smallest execution unit**.
  - Each task represents one **local computation**.
  - All of the **tasks in one stage** execute the same code on a **different piece of the data**.



[H. Karau et al., High Performance Spark, O'Reilly Media, 2017]



# Partitioning and Shuffle Operations





## Shuffle Operations

- ▶ The **shuffle** is Spark's mechanism for **re-distributing data** so that it's grouped differently across partitions.
- ▶ This typically involves **copying data** across **executors and machines**, making the shuffle a **complex and costly** operation.



## Spark Built-in Partitioners

- ▶ Hash partitioner
- ▶ Range partitioner



## Hash Partitioning (1/2)

- ▶ **Hash partitioning** attempts to spread data **evenly across partitions** based on the **key**.
- ▶ E.g., `groupByKey`
  - First computes the **partition  $p$**  of each tuple  $(k, v)$ :  
 `$p = k.hashCode() \% numPartitions$`
  - Then, **all tuples** in the **same partition  $p$**  are sent to the **machine hosting  $p$** .



## Hash Partitioning (2/2)

- ▶ Assume a key-value RDD, with keys  $k = [8, 96, 240, 400, 401, 800]$ , and a desired number of partitions of  $p = 4$ .
- ▶ Assume, that `hashCode()` is the `identity`, i.e.,  $n.hashCode() = n$ .
- ▶ The hash partitioning distributes the keys as follows among the partitions ( $p = k \% 4$ ):
  - partition 0: `[8, 96, 240, 400, 800]`
  - partition 1: `[401]`
  - partition 2: `[]`
  - partition 3: `[]`
- ▶ The result is a very `unbalanced distribution` which hurts `performance`.



## Range Partitioning (1/2)

- ▶ Key-value RDDs may contain **keys** that have an **ordering** defined, e.g., Int, Char, String, ...
- ▶ For such RDDs, **range partitioning** may be more **efficient**.
- ▶ Using a range partitioner, keys are partitioned according to:
  - An **ordering** for keys
  - A set of **sorted ranges** of keys
- ▶ Tuples with **keys in the same range** appear on the **same machine**.



## Range Partitioning (2/2)

- ▶ Using range partitioning the **distribution can be improved** significantly:
  - Assumptions: (a) keys **non-negative**, and (b) 800 is **biggest key** in the RDD
  - Set of ranges: [1, 200], [201, 400], [401, 600], [601, 800]
- ▶ The **range partitioning** distributes the keys as follows among the partitions:
  - **partition 0**: [8, 96]
  - **partition 1**: [240, 400]
  - **partition 2**: [401]
  - **partition 3**: [800]
- ▶ The resulting partitioning is much more **balanced**.



## Partition Operations (1/2)

- ▶ `mapPartitions` is similar to `map`, but runs **separately on each partition** of the RDD.
- ▶ `mapPartitionsWithIndex` applies the function on **specific partitions**.
- ▶ The given functions must be of type `Iterator<T> => Iterator<U>` when running on an RDD of type `T`.

```
val words = sc.parallelize("take it easy, this is a test".split(" "), 2)

def func(partIndex: Int, withinPartIter: Iterator[String]) = {
  withinPartIter.toList.map(value => s"Partition: $partIndex => $value").iterator
}

words.mapPartitionsWithIndex(func).collect()
// Array(Partition: 0 => take, Partition: 0 => it, Partition: 0 => easy,,
// Partition: 1 => this, Partition: 1 => is, Partition: 1 => a, Partition: 1 => test)
```



## Partition Operations (2/2)

- ▶ `foreachPartitions` is similar to `mapPartition`, but does not return a return a value.

```
val words = sc.parallelize("take it easy, this is a test".split(" "), 2)

words.foreachPartition { iter =>
  import java.io._
  import scala.util.Random
  val rndName = new Random().nextInt()
  val pw = new PrintWriter(new File(s"/tmp/file-${rndName}.txt"))
  while (iter.hasNext) {
    pw.write(iter.next())
  }
  pw.close()
}
```





## Partitioning Data

- ▶ How do we **set a partitioning** for our data?
- ▶ There are **two ways** to create RDDs with specific partitionings:
  1. Call **partitionBy** on an RDD, providing an **explicit partitioner**.
  2. Using **transformations** that return RDDs with specific partitioners.



## Partitioning Data Using `partitionBy`

- ▶ Invoking `partitionBy` creates an RDD with a **specified partitioner**.

```
val keyword = ...  
// (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,1), ...  
  
val tunedPartitioner = new RangePartitioner(3, keyword)  
  
val partitioned = keyword.partitionBy(tunedPartitioner).persist()  
  
partitioned.getNumPartitions
```

- ▶ The result of `partitionBy` should be **persisted**, otherwise the partitioning is **repeatedly applied each time** the partitioned RDD is used.



# Partitioning Data Using Transformations

- ▶ Some operations on RDDs automatically result in an RDD with a `known partitioner` - for when it makes sense.
- ▶ For example
  - When using `sortByKey`, a `RangePartitioner` is used.
  - When using `groupByKey`, a `HashPartitioner` is used.



## Controlling Partitions (1/2)

- ▶ The challenge is that **not all values for a single key** necessarily **reside on the same partition**, or even the **same worker**, but they must be co-located to compute the result.
- ▶ For example, the **reduceByKey** generates a tuple of a key and the result of executing a **reduce function** against all values associated with that key.



## Controlling Partitions (2/2)

- ▶ With RDDs, you have control over **how data is exactly physically distributed** across the cluster.
- ▶ **coalesce** effectively **collapses partitions** on the **same worker** in order to avoid a shuffle.
- ▶ **repartition** operation allows you to **repartition your data up or down**.

```
val words = sc.parallelize("take it easy, this is a test".split(" "), 2)

words.coalesce(1).getNumPartitions

words.repartition(10)
```

# Summary



## Summary

- ▶ RDD: a distributed memory abstraction
- ▶ Two types of operations: transformations and actions
- ▶ Lineage graph
- ▶ Caching
- ▶ Wide vs. narrow dependencies
- ▶ Partitioning and shuffle



## References

- ▶ M. Zaharia et al., “Spark: The Definitive Guide”, O’Reilly Media, 2018 - Chapters 2, 12, 13, and 14
- ▶ M. Zaharia et al., “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”, USENIX NSDI, 2012.
- ▶ Some slides were derived from Heather Miller’s slides:  
<http://heather.miller.am/teaching/cs4240/spring2018>



Questions?