Apache Spark (introduction)

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

Slides are derived from online materials:
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf
https://spark.apache.org/examples.html

Learning objectives

Introduction to Apache Spark:

- RDDs
- operator pipelining
- lineage
- transformations and actions
- examples: word count, PageRank

Motivation

From Spark paper (NSDI'12):

"Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance. Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that reuse intermediate results across multiple computations. Data reuse is common in many iterative machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is interactive data mining, where a user runs multiple adhoc queries on the same subset of the data."

Resilient Distributed Datasets

From Spark paper (NSDI'12):

"In this paper, we propose a new abstraction called resilient distributed datasets (RDDs) that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators."

From API docs:

A Resilient Distributed Data Set (RDD) is the basic abstraction in Spark. It represents an immutable, partitioned collection of elements that can be operated on in parallel.

RDDs in action

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

errors.count()

// 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()
```

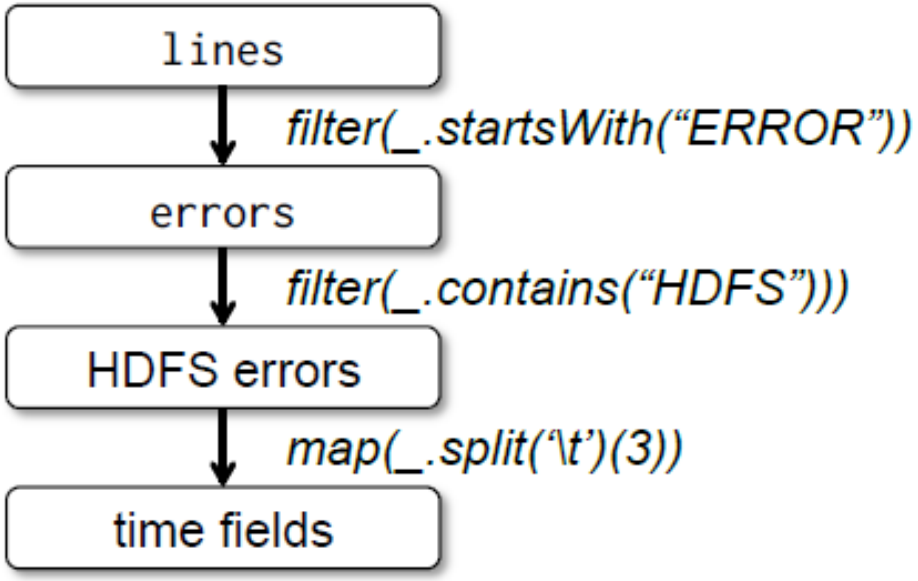
fast

← hint

slow

Lineage

Lineage: how an RDD was derived from other RDDs, used to rebuild an RDD in the event of a failure.



Lineage graph for example from earlier slide.

Transformations and actions

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

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Note: Recent releases of Apache Spark support additional operators. For details refer to the online documentation: http://spark.apache.org/docs/latest/programming-guide.html.

Transformations and actions

- **Transformations** are data operations that convert one RDD or a pair of RDDs into another RDD.
- **Actions** are data operations that convert an RDD into an output, for example a number or a sequence of values.
- When an action is invoked on an RDD, the Spark scheduler examines the lineage graph of the RDD and builds a directed acyclic graph (DAG) of transformations.
- The transformations in the DAG are further grouped into **stages**. A stage is a collection of transformations with **narrow dependencies**, meaning that one partition of the output depends on only one partition of each input (e.g., filter). These transformations can be pipelined together effectively. On the other hand, the boundaries between stages correspond to **wide dependencies**, meaning that one partition of the output depends on multiple partitions of some input (e.g., groupByKey). These transformations require a shuffle.

Narrow vs. wide dependencies

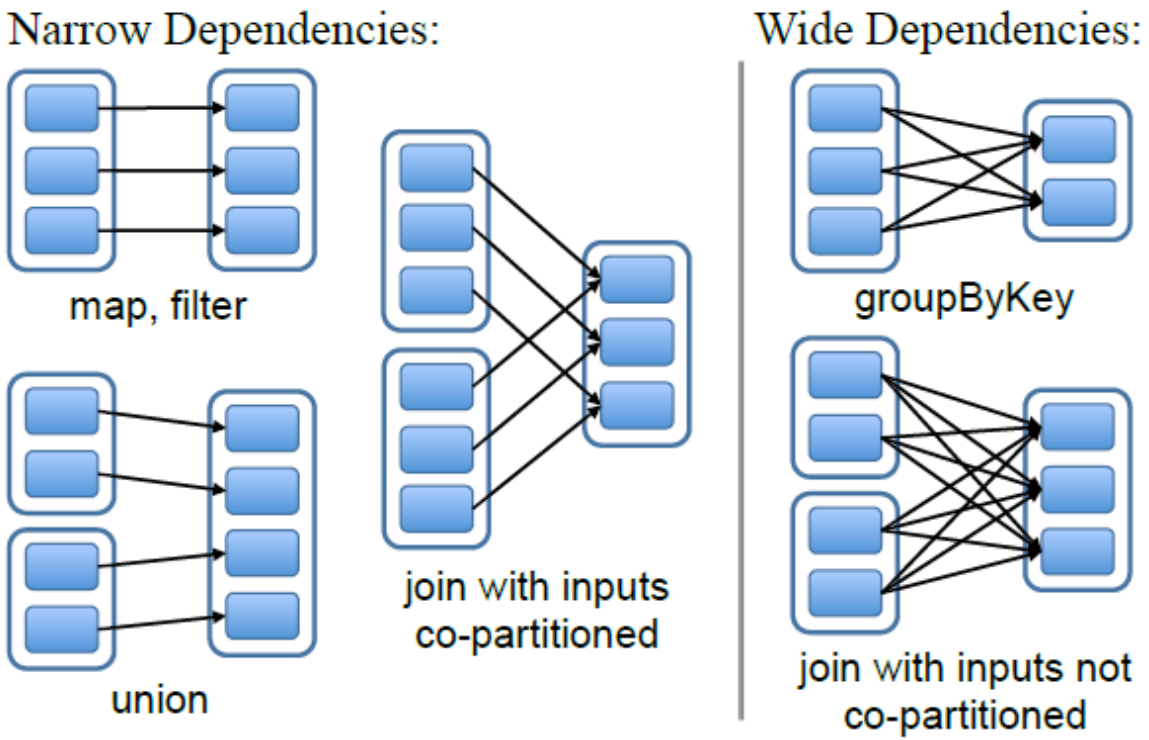


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

Execution of a job in stages

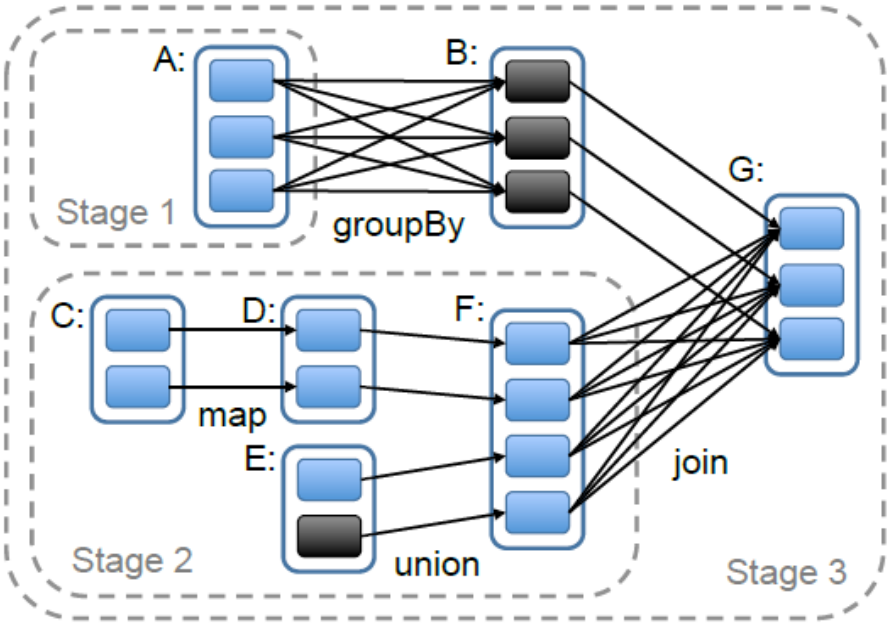


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

Example 1: word count (in Scala)

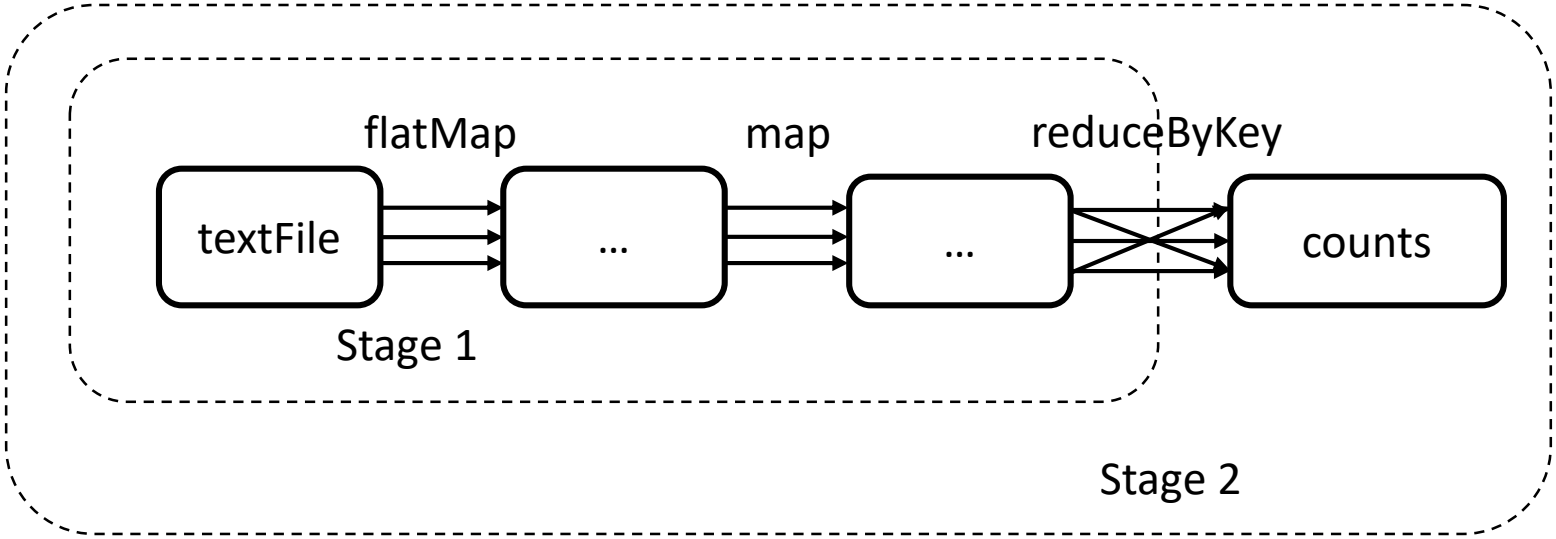
Explanation:

- first build an RDD called textFile using data in HDFS
- then transform textFile to counts by applying a sequence of three operators:
  1. tokenize each line of input
  2. map each word to the tuple (word, 1)
  3. reduce the word counts
- dump the output back to HDFS

```
val textFile = spark.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Example 1: lineage graph

The saveAsTextFile action applied to the "counts" RDD triggers execution of the transformations. Both flatMap and map have narrow dependencies, whereas reduceByKey has a wide dependency because the output of the maps is not partitioned by the key. The job executes in two stages, one for the flatMap/map and another for reduceByKey.



Scala placeholder syntax

Transformations are often specified using anonymous functions, which often use Scala's convenient placeholder syntax (i.e., "_" notation):

_ + 1	x => x + 1
_ * _	(x1, x2) => x1 * x2
(_: Int) * 2	(x: Int) => (x: Int) * 2
if (_) x else y	z => if (z) x else y
_.map(f)	x => x.map(f)
.map(+ 1)	x => x.map(y => y + 1)

Source: http://www.scala-lang.org/files/archive/spec/2.11/06-expressions.html#placeholder-syntax-for-anonymous-functions

Note: The "map" keyword in this context is an ordinary Scala construct, and not a Spark transformation. Use of this construct is demonstrated in the PageRank example shown later on.

Scala arrays and tuples

- An element of an RDD can be a single item, an array of items (e.g., from tokenizing a string), or a tuple (e.g., key-vale pair).
- Use underscore notation to select different elements of a tuple:

```
val t = (4, 3, 2)
val sum = t._1 + t._2 + t._3
```

- Use round brackets to select different elements of an array:

```
val myarr = Array(4, 3, 2)
if ( myarr(0) < myarr(1) ) { … }
```

- Use the drop(n) method to remove the first n elements of an Array or List.

Example 2: PageRank

From Spark paper (NSDI'12):

"The algorithm iteratively updates a rank for each document by adding up contributions from documents that link to it. On each iteration, each document sends a contribution of r/n to its neighbors, where r is its rank and n is its number of neighbors. It then updates its rank to

$$(1 - d)/N + d \sum_i c_i$$

where the sum is over the contributions it received and N is the total number of documents."

Note 1: Initially, we can assign equal ranks of 1/N to all documents. Alternatively, we can assign denormalized ranks of 1 instead of 1/N, in which case the equation changes to $(1 - d) + d \sum_i c_i$.

Note 2: Typical values for the damping factor d are around 0.85.

Note 3: Sometimes the damping factor d is written as (1 – a), for example in the Spark Scala code on the next slide.

Example 2: PageRank (in Scala)

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
               .mapValues(sum => a/N + (1-a)*sum)
}
```

For detailed source code refer to
https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/SparkPageRank.scala

Example 2: lineage graph

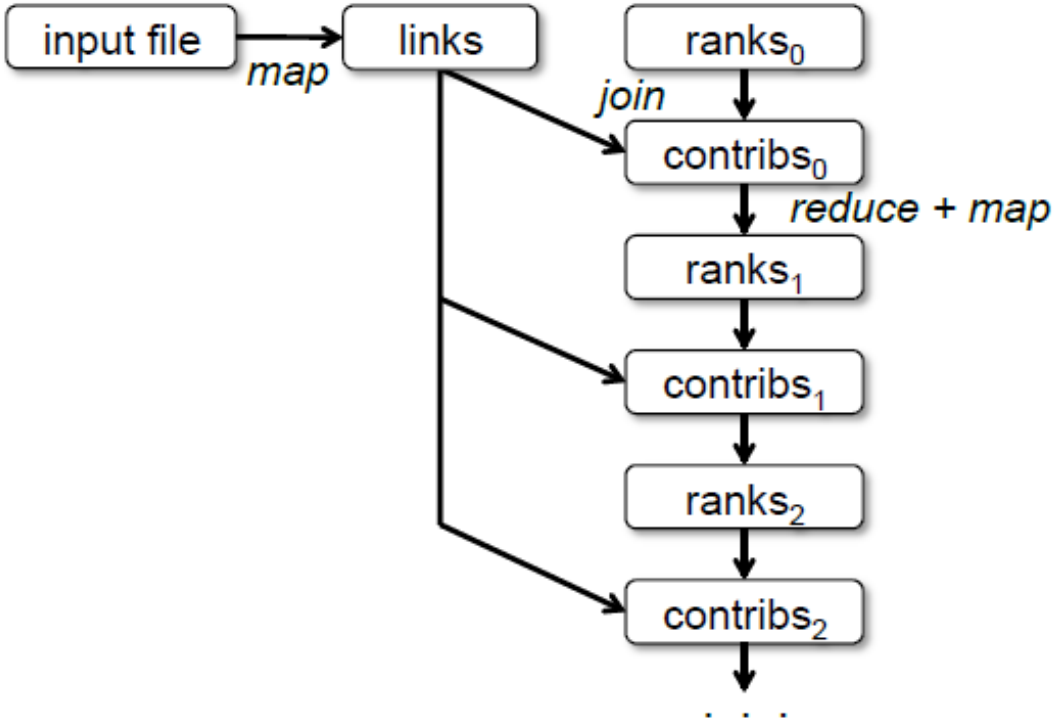


Figure 3: Lineage graph for datasets in PageRank.

Example 2: input data set and intermediate outputs

links RDD:

url1 url2
url2 url1
url3 url1 url4
url4 url3

ranks RDD: after 1 iteration

url1 1.425
url2 1.0
url3 1.0
url4 0.575

ranks RDD: after 2 iterations

url1 1.425
url2 1.36125
url3 0.63875
url4 0.575

ranks RDD: after 10 iterations

url1 1.661…
url2 1.562…
url3 0.437…
url4 0.338…

Note: The code at the URL at the bottom of the earlier slide assumes the input file contains one edge per line.

Example 2: first iteration in detail

initial ranks RDD:

url1 1
url2 1
url3 1
url4 1

links.join(ranks) RDD:

url1 ({url2}, 1)
url2 ({url1}, 1)
url3 ({url1, url4}, 1)
url4 ({url3}, 1)

links.join(ranks).flatMap(…) RDD:

url2 1
url1 1
url1 0.5
url4 0.5
url3 1

RDD after reduceByKey:

url1 1.5
url2 1
url3 1
url4 0.5

Next step: **mapValues** with
d = (1 – a) = 0.85

Spark vs. Hadoop MapReduce

PageRank can be implemented using Hadoop as well, but such a solution has two weaknesses:

1. The intermediate output is dumped to HDFS after each iteration because each iteration requires a separate Hadoop MapReduce job. This leads to unnecessary I/O.
2. If a convergence test is used to terminate the loop, as opposed to computing for a fixed number of iterations, then a second MapReduce job is required in each iteration to evaluate the convergence condition.

In comparison, Spark is able to cache intermediate data (i.e., links and ranks) in main memory provided that it is given sufficient resources.

Note: The "persist" keyword in the Scala code requests that Spark hold the "links" RDD in main memory. Outputs of transformations are cached automatically, subject to resource limits.