Introduction to Distributed Computing with Spark

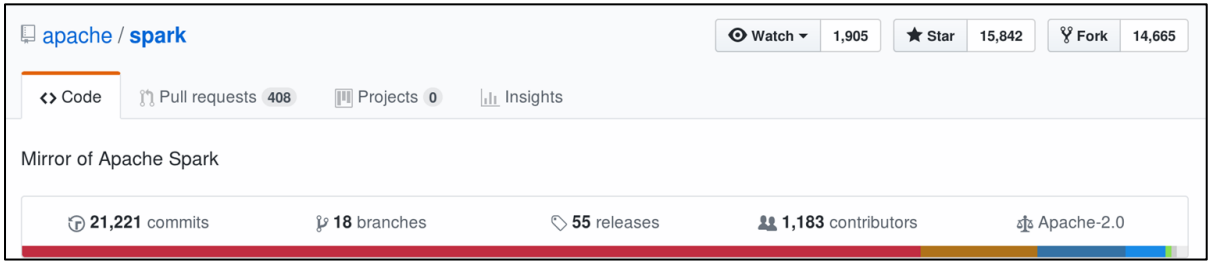
Gina Cody School of Engineering and Computer Science

Department of Computer Science and Software Engineering

Credit: Tristan Glatard - Big Data Analytics

Scope

- Most active open-source project in Big Data Analytics



- Used in thousands organizations from diverse sectors:
  - Technology, banking, retail, biotech, astronomy
- Deployments with up to 8,000 nodes have been announced

Credit: Tristan Glatard - Big Data Analytics

Introduction

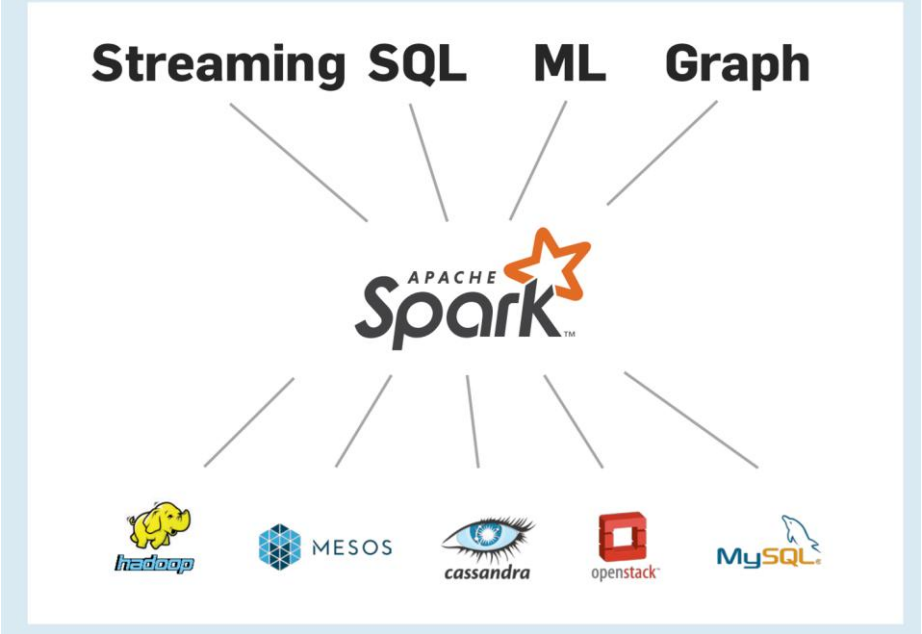
- **We need cluster computing to analyse Big Data**
- **There has been an explosion of specialized computing models**
  - MapReduce
  - Google's Dremel (SQL) and Pregel (graphs)
  - Hadoop's Storm, Impala, etc
- **Big Data applications need to combine many types of processing**
  - To handle data variety
  - Users need to stitch systems together

Credit: Tristan Glatard - Big Data Analytics

Introduction (2)

- The profusion of systems is detrimental to
  - Usability
  - Performance
- Spark provides a uniform framework for Big Data Analytics

Figure 1. Apache Spark software stack, with specialized processing libraries implemented over the core engine.



Highlights

- **Spark concepts**
  - Built around a *data structure*: Resilient Distributed Datasets (RDDs)
  - Programming model is MapReduce + extensions
- **Added value compared to MapReduce**
  - Easier to program because API is unified
  - Interactive explorations of Big Data are possible (pyspark)
  - More efficient to combine multiple tasks (workflows) due to lazy evaluation.

In-memory processing

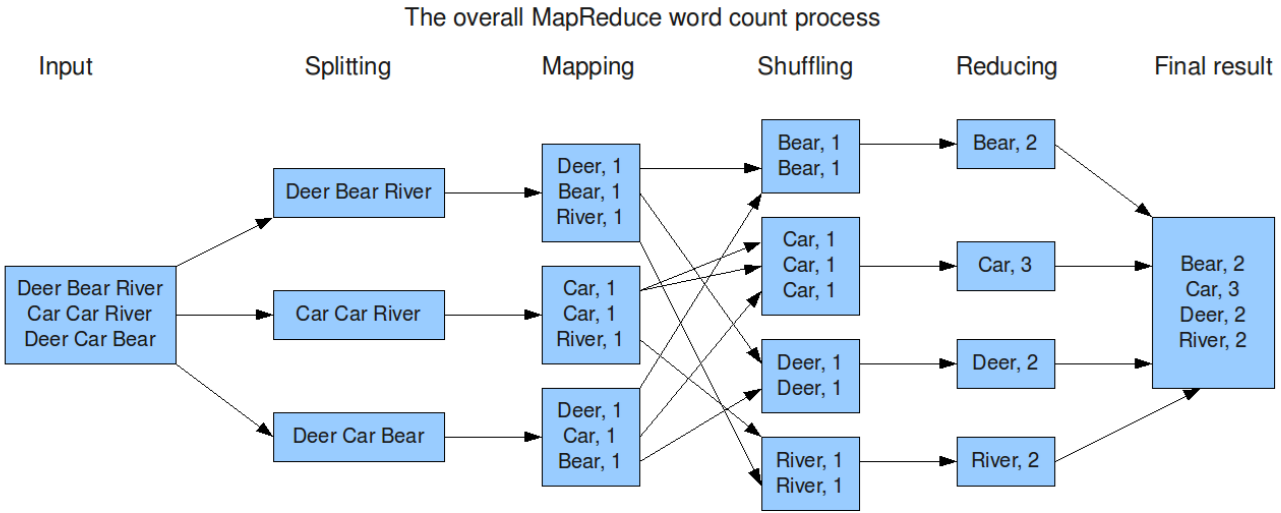
- **Spark keeps data in memory**
  - Reduces disk I/O
  - Spills to disk if not enough space

- **MapReduce lacks abstraction to leverage distributed memory**
  - Inefficient for applications that *reuse* intermediate results
  - For instance *iterative* algorithms: kmeans, PageRank

Credit: Tristan Glatard - Big Data Analytics

MapReduce

- MapReduce is a programming model for processing large data sets in a distributed fashion over a several machines.



Credit: https://nyu-cds.github.io/python-mapreduce/ Credit: Tristan Glatard, Big Data Analytics

Programming model: RDDs

RDDs are fault-tolerant collections of objects partitioned across a cluster that can be manipulated in parallel.

Credit: Tristan Glatard - Big Data Analytics

RDD abstraction

- **An RDD is a read-only, partitioned collection of records.**
  - RDD is fault-tolerant, hence it is read-only.
- **Can be created through operations on:**
  - Data in stable storage (e.g., sc.textFile)
  - Other RDDs: transformations
  - Existing collections (sc.parallelize)

Credit: Tristan Glatard - Big Data Analytics

Functional programming API

- **Transformations:**
  - Return RDDs
  - Are parametrized by user-provided functions

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(
  s => s.startsWith("ERROR"))
println("Total errors: "+errors.count())
```

- **Actions return results to application or storage system**

Credit: Tristan Glatard - Big Data Analytics

RDD transformations and actions

Transformations Return an RDD	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	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)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions Return something else	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	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.

(K, V) is a key-value pair
MapReduce = flatMap + reduceByKey

Lazy evaluation

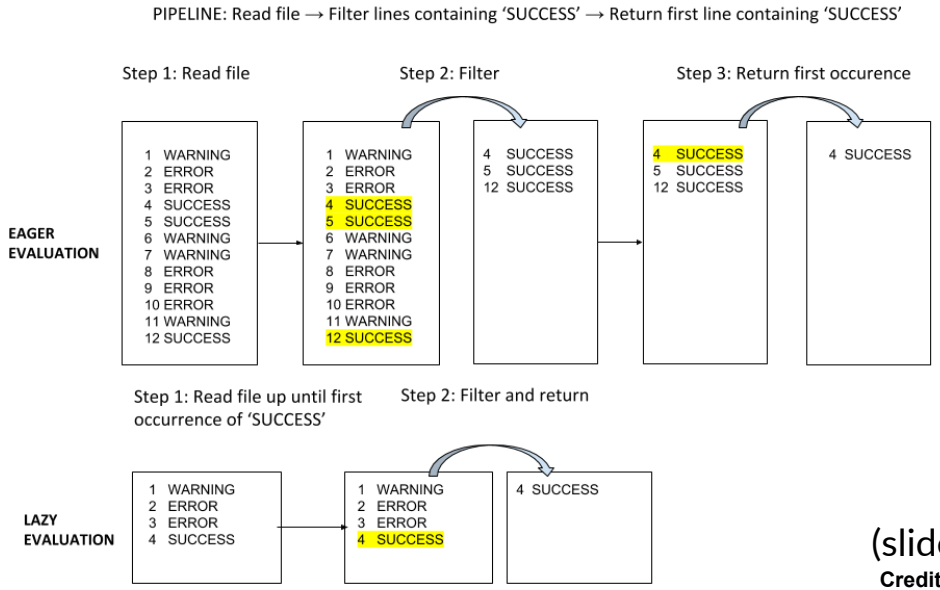
- **Transformations do not compute RDDs immediately**
  - The returned RDD object is a representation of the result
- **Actions trigger computations**
- **This allows optimization in the execution plan**
  - Multiple map operations are fused in one pass
  - While preserving modularity

Credit: Tristan Glatard - Big Data Analytics

Lazy evaluation (2)

Computations are costly and memory is limited → Compute only what is necessary

Note:
With *eager evaluation*, the entire file must be read and filtered.
Lazy evaluation limits the amount of data that needs to be processed



(slide from V. Hayot-Sasson)

Credit: Tristan Glatard - Big Data Analytics

Persistence (or caching)

- By default RDDs are ephemeral
- They might be evicted from memory
- Users can persist RDDs in memory or on disk, by calling .persist()
- persist() accepts priorities for spilling to disk

Credit: Tristan Glatard - Big Data Analytics

Example: Log Mining

Reads from HDFS

Nothing has executed so far (lazy)

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

Triggers execution

```
errors.count()
```

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()
```

Other uses of errors won't reload data from the text file.

```
// 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()
```

Lines is never stored in memory!

Lineage graph

An RDD maintains information on how it was derived from other datasets.

From the lineage, the RDD can be completely (re-)computed.

This is used for fault-tolerance.

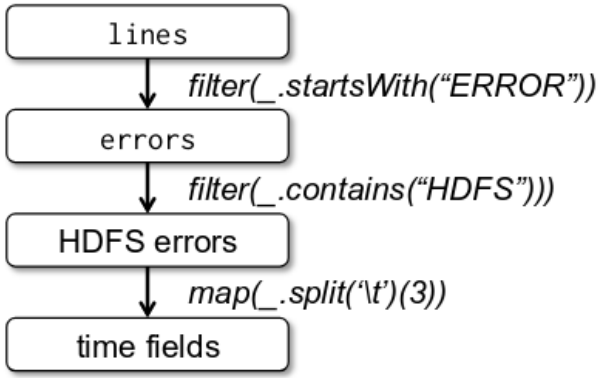


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

Spark programming interface

- **Driver**
  - Runs user program
  - Defines RDDs and invokes actions
  - Tracks the lineage
- **Workers**
  - Long-lived processes
  - Store RDD partitions in RAM across operations

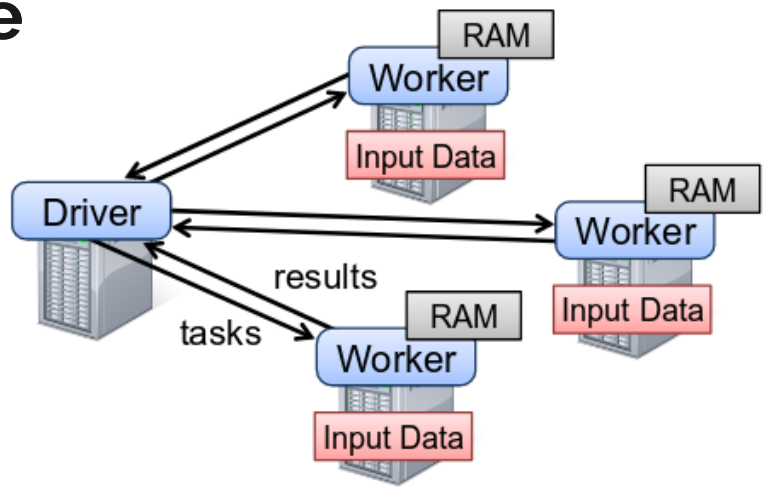


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

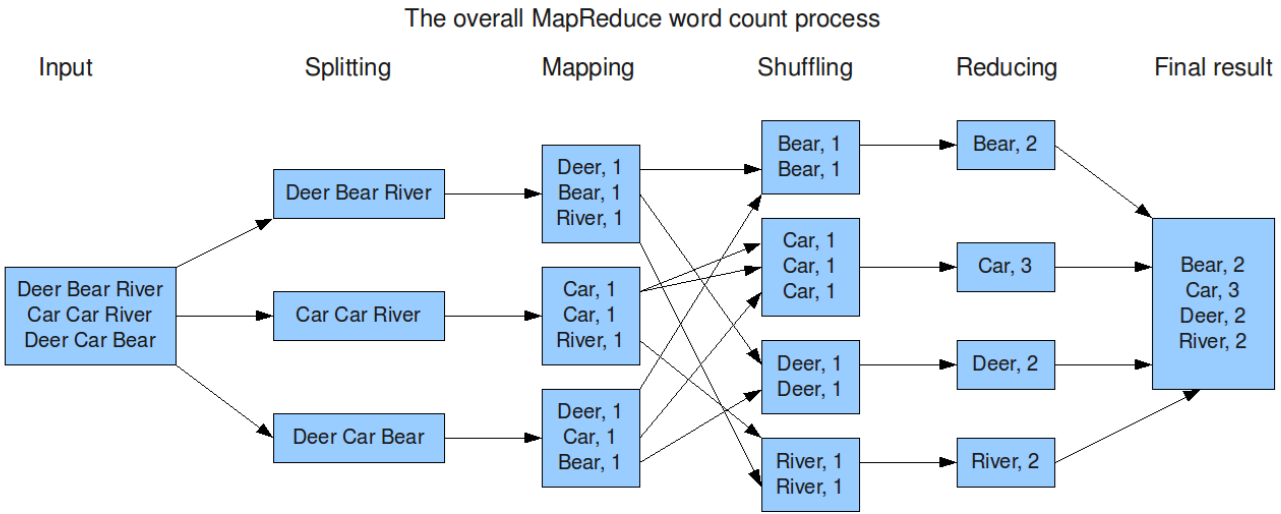
Fault tolerance

- **When a task fail**
  - If parent partitions are available: re-submit to another node
  - Otherwise: resubmit tasks to recompute missing partitions
- **Checkpointing**
  - Lineage-based recovery of RDDs may be time-consuming
  - RDDs can be checkpointed to stable storage, through a flag to .persist()

Credit: Tristan Glatard - Big Data Analytics

MapReduce

- MapReduce is a programming model for processing large data sets in a distributed fashion over a several machines.



Credit: https://nyu-cds.github.io/python-bigdata/02-hadoop/

MapReduce Programming model

- Spark supports MapReduce on RDDs.
- **Input**
  - A set of *input* key-value pairs
- **Map function**
  - Provided by user
  - Takes *an* input pair
  - Produces *intermediate* key-value pairs
- Shuffle and sort
  - Done by framework
  - Aggregate intermediate values by key
- Reduce function
  - Provided by user
  - Takes *intermediate* key and list of values
  - Produces a set of *output* key-value pairs

Credit: Tristan Glatard - Big Data Analytics

Example: Word Count (1)

```
from pyspark import SparkContext
sc = SparkContext("local[*]")

# read the text file
words = sc.textFile('grocery.txt')
# map each word with one occurrence
words_mapped = words.map(lambda x: (x, 1))
# reduce each word tuples by summing the occurrences
counts = words_mapped.reduceByKey(lambda x, y: x+y)

print(counts.collect())
```

Input:

```
butter
mango
pizza
croissant
stew
wheat
stew
empanada
butter
nectarine
mousse
pizza
croissant
```

Output:

```
butter, 2
mango, 1
pizza, 2
croissant, 2
stew, 2
wheat, 1
empanada, 1
nectarine, 1
mousse, 1
```

Credit: Tristan Glatard - Big Data Analytics

Example: Word Count (2)

```
from pyspark import SparkContext
sc = SparkContext("local[*]")

# read the text file
lines = sc.textFile('shakespear.txt')
# map each lines to word occurrences
lines_mapped = lines.flatMap(lambda x: [(word, 1) for word in x.split()])
# reduce each word tuples by summing the occurrences
counts = lines_mapped.reduceByKey(lambda x, y: x+y)

print(counts.collect())
```

Input:

```
From fairest creatures we desire increase
That thereby beauty's rose might never die
But as the riper should by time decease
His tender heir might bear his memory
But thou contracted to thine own bright eyes
```

Output:

```
From, 1
fairest, 1
creatures, 1
we, 1
desire, 1
...
```

Credit: Tristan Glatard - Big Data Analytics

Recap

General framework (increases usability and performance)

Rich programming model (transformations, actions, libraries)

Data locality (as in MapReduce)

RDDs remain in-memory (reduces disk I/O)

Lazy RDD evaluations (allow for merging transformations)

Fault-tolerance through lineage (better than data replication)

Installing PySpark

- Make sure Java (JDK) is installed.
- Install PySpark:

```
pip install pyspark
```

- For Windows, make sure environment variables are set correctly and winutils.exe is installed. For a step-by-step guide, see here .

RDDs can map any function - Finding Primes

```
def isprime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if not n & 1:
        return False
    for x in range(3, int(n**0.5)+1, 2):
        if n % x == 0:
            return False
    return True

from pyspark import SparkContext
sc = SparkContext("local[*]")
nums = sc.parallelize(range(10000000))
num_primes = nums.map(lambda x: isprime(x))
primes = num_primes.filter(lambda x: x)
print(primes.count())
```

Credit: Tristan Glatard - Big Data Analytics

PySpark Hello World

```
import pyspark
sc = pyspark.SparkContext('local[*]')

txt = sc.textFile('sample.log')

error_lines = txt.filter(lambda line: 'error' in line)

print(error_lines.count())
```

Credit: https://realpython.com/pyspark-intro/

Exercises

- **Write MapReduce programs in PySpark that:**
  - Given receipt.txt containing items and their prices, calculate the total revenue of the store.
  - Given students.txt containing the courses registered by each student, calculate the total number of students per course.