Large-Scale Data Management and Distributed Systems

II. MapReduce and Hadoop

Vania Marangozova

Vania.Marangozova@imag.fr

2023-2024

References

- Lecture notes of V.Leroy
- Lecture notes of Y.Vernaz
- Lecture notes of T.Ropars

Agenda

- The Basics of Apache Spark
- Spark API
- Programming with PySpark

Introduction to Spark

- Originally developed at Univ. of California
- Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, M. Zaharia et al. NSDI, 2012.
- One of the most popular Big Data projects today.

https://spark.apache.org/faq.html

How large a cluster can Spark scale to?

Many organizations run Spark on clusters of thousands of nodes. The largest cluster we know has 8000 of them. In terms of data size, Spark has been shown to work well up to petabytes. It has been used to sort 100 TB of data 3X faster than Hadoop MapReduce on 1/10th of the machines, winning the 2014 Daytona GraySort Benchmark, as well as to sort 1 PB. Several production workloads use Spark to do ETL and data analysis on PBs of data.

Motivations behind Spark

- Limitations of Hadoop MapReduce
  - Limited performance for iterative algorithms
    - Data are flushed to disk after each iteration
    - More generally, low performance for complex algorithms

- Main novelties of Spark
  - Computing in **memory**
  - A new computing abstraction: **Resilient Distributed Datasets (RDD)**

Spark vs Hadoop

- Better performances
- Interactives queries
- Supports more operations on data
- A full ecosystem (high-level libraries)
- Running on your machine or at scale

Programmer with Spark

- Core API
  - Scala
  - Python
  - Java
- Storage: any supported by Hadoop
  - Local FS
  - HDFS
  - Cassandra
  - Amazon S3

Many resources to get started

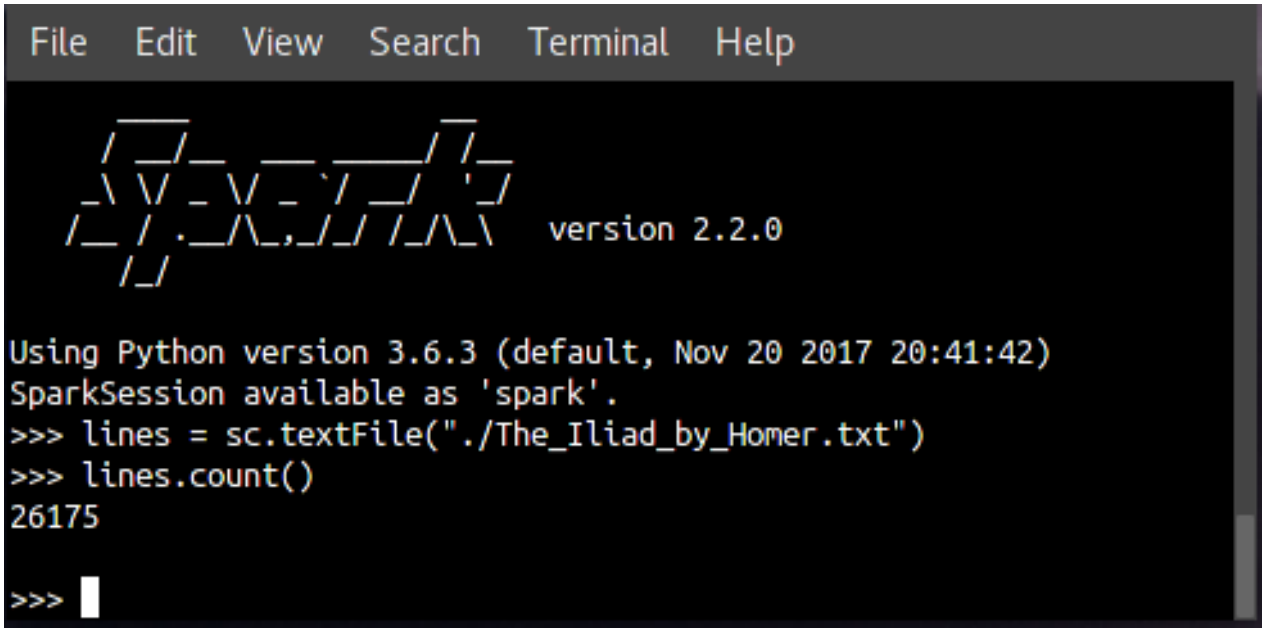
- https://spark.apache.org/
- https://sparkhub.databricks.com/
- Tutorials, blogs, ...

Strating with Spark

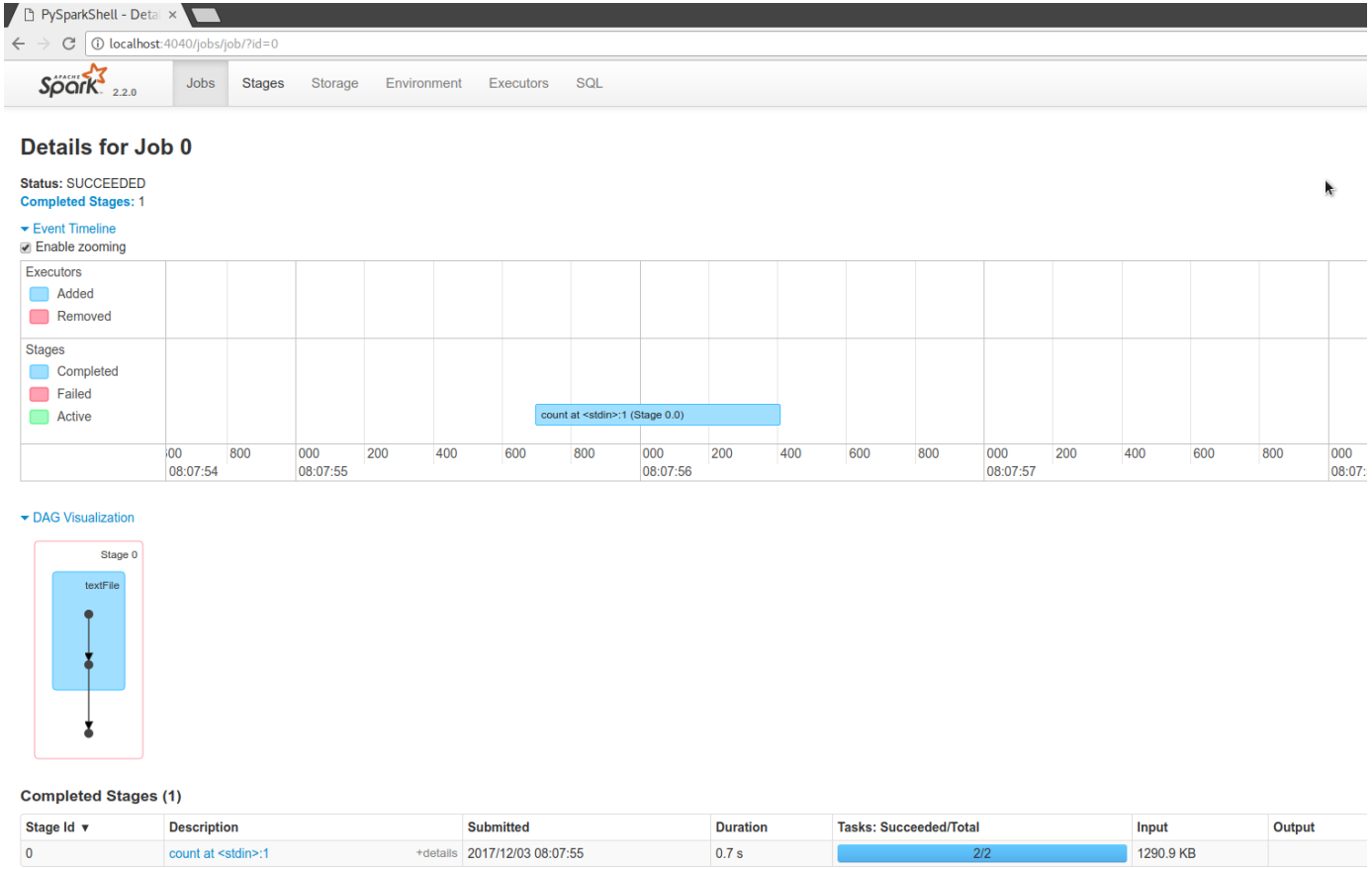
- Running in local mode
  - Spark runs in a JVM
  - Spark is coded in Scala
  - Read data from your local file system
- Use interactive shell
  - Scala (spark-shell)
  - Python (pyspark)
  - Run locally or distributed at scale

A very first example with pyspark

- Counting lines

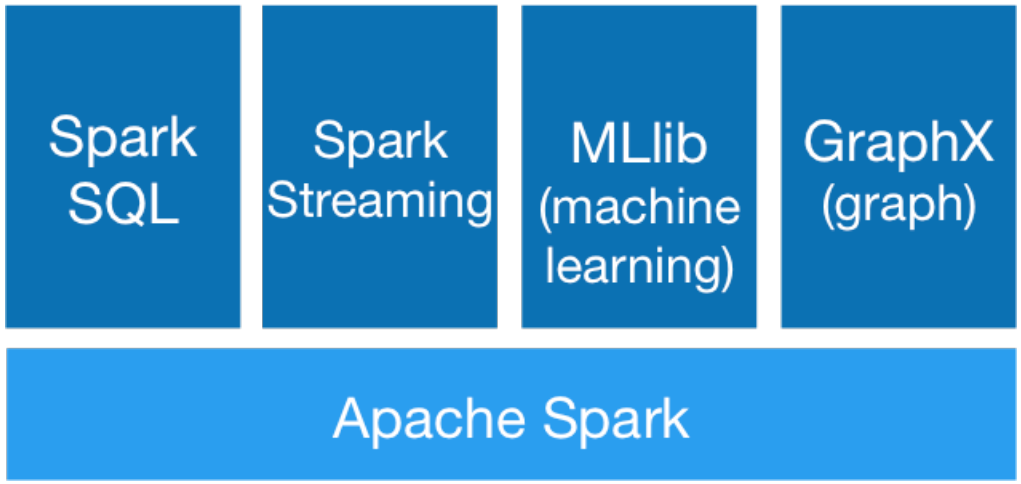


The Spark Web UI



The Spark Built-in Libraries

- Spark SQL: For structured data (Dataframes)
- Spark Streaming: Stream processing (micro-batching)
- MLlib: Machine learning
- GraphX: Graph processing



In-memory computing: Insights

See Latency Numbers Every Programmer Should Know
https://gist.github.com/jboner/2841832

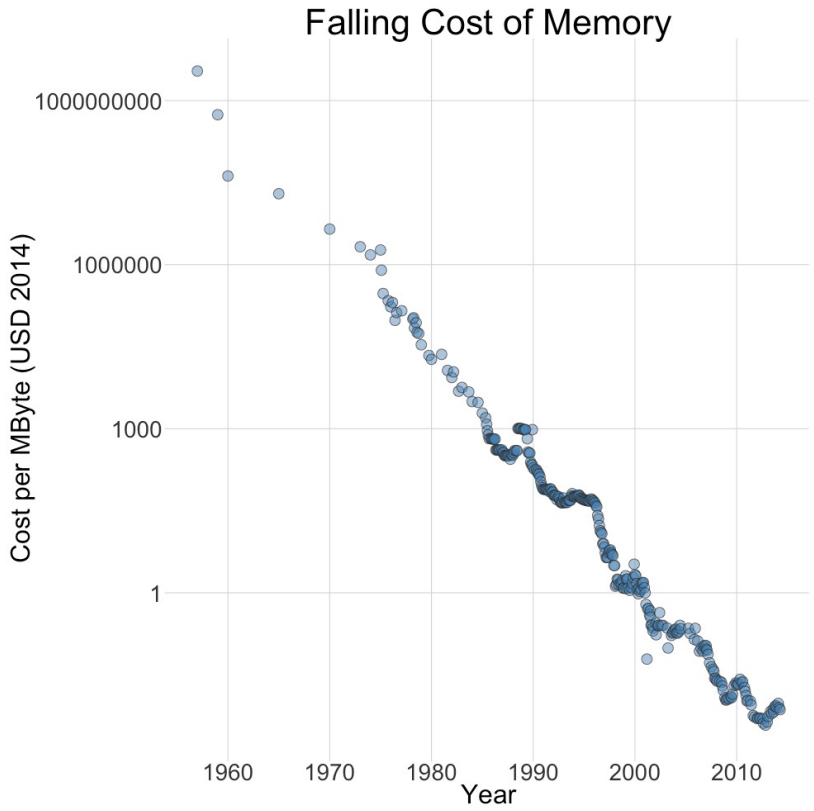
Memory is way faster than disks

Read latency

- HDD: a few milliseconds
- SDD: 10s of microseconds (100X faster than HDD)
- DRAM: 100 nanoseconds (100X faster than SDD)

In-memory computing: Insights

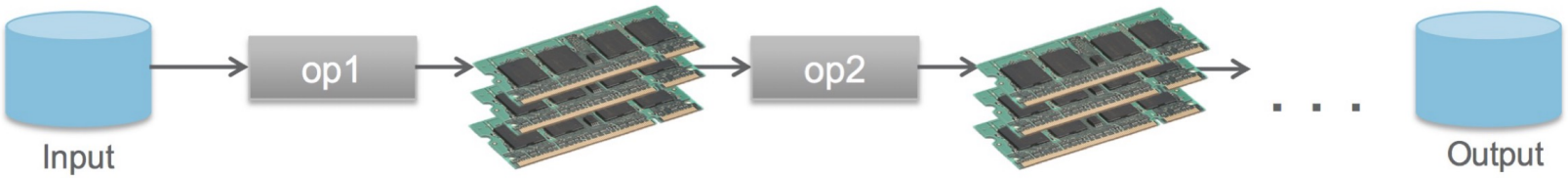
Graph by P. Johnson



Efficient iterative computation

- Hadoop, at each step, data go through the disk

- Spark, data remain in memory (if possible)



Main Challenge: Fault Tolerance

- Failure is the norm rather than the exception
- If there is a failure, data in memory is lost

$\Longrightarrow$

- The RDD Proposal = Resilient Distributed Dataset
  - Read-only partitioned collection of records
  - Creation of a RDD through deterministic operations (transformations) on
    - Data stored on disk
    - an existing RDD

Programming with RDDs

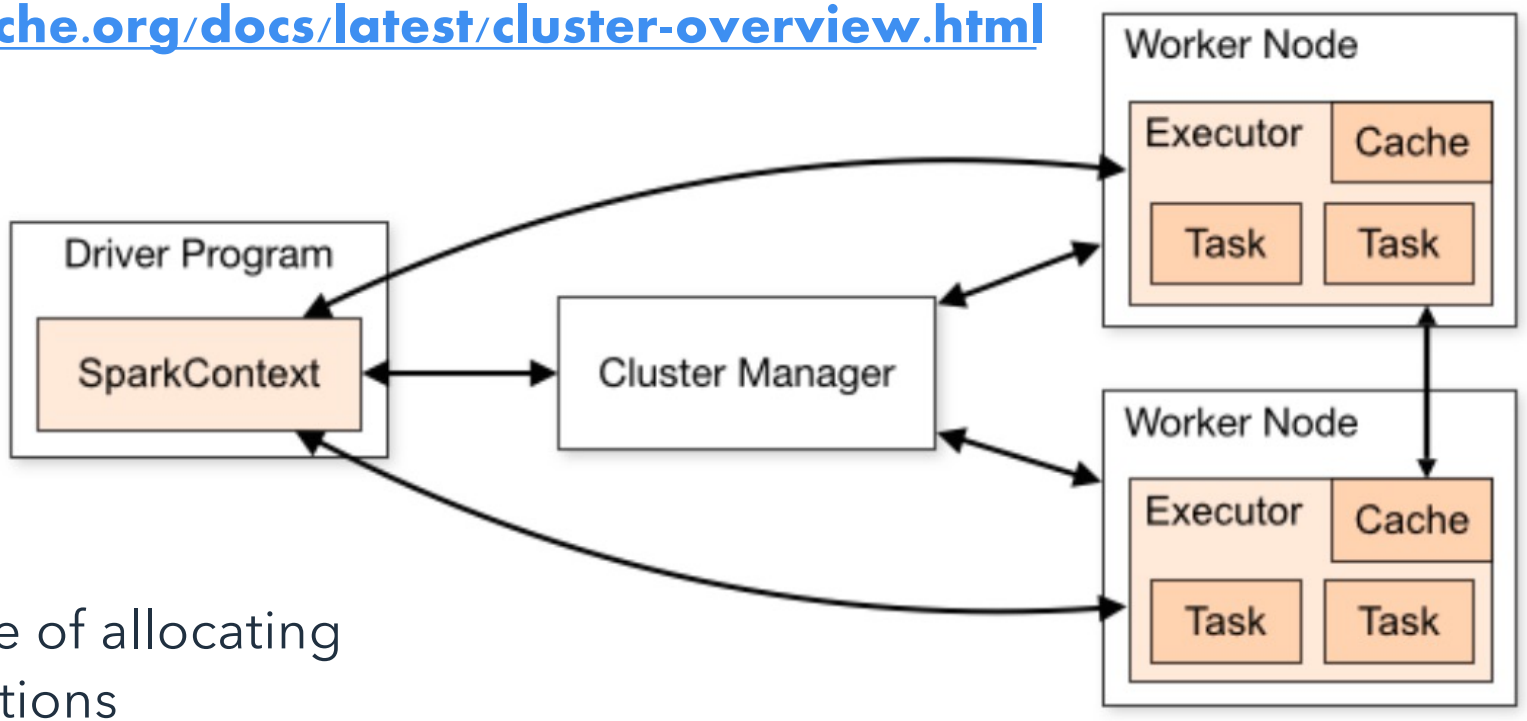
- RDD = objects
- Transformations
  - Programmer defines RDDs using Transformations
  - Applied to data on disk or to existing RDDs
  - Examples of transformations: *map, filter, join*
- Actions
  - Operations that return a value or export data to the file system
  - Examples of actions: *count, reduce*

Fault tolerance with Lineage

- Lineage = a description of an RDD
  - The data source on disk
  - The sequence of applied transformations
    - Same transformation applied to all elements
    - Low footprint for storing a lineage
- Fault tolerance
  - RDD partition lost:
    replay all transformations on the subset of input data or
    the most recent RDD available
  - Deal with stragglers:
    generate a new copy of a partition on another node

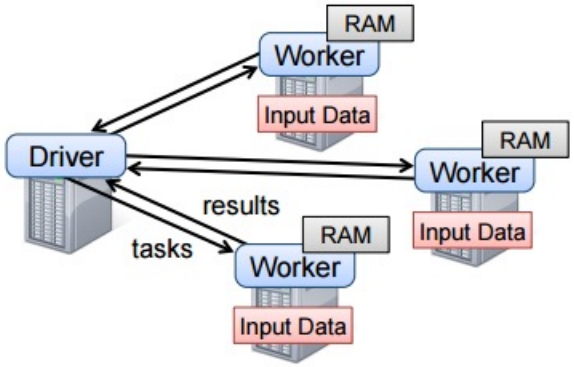
Spark Runtime

https://spark.apache.org/docs/latest/cluster-overview.html



- Cluster Manager:
  The system in charge of allocating resources to applications
- Worker nodes:
  Nodes of the cluster on which the Spark applications are run

Spark Runtime



- Driver (= Master)
  Main program of a spark application
  - Created when an application is submitted
  - Translates the user's program into a graph of tasks
  - Assigns tasks to executors

- Executor: A dedicated process (a new JVM) created on a worker to execute an application
  - Created when an application is submitted
    - By default a Spark apps tries to use all resources of the cluster
    - One executor per worker – An executor uses all cores of the worker
  - Can include multiple executor threads
  - Execute tasks on partitions

Partitioning

See https://spark.apache.org/docs/latest/rdd-programming-guide.html# parallelized-collections

- Partitions are the unit of parallelism in Spark
  - RDDs are divided into partitions
  - To execute an operation on a RDD, a task per partition is created
  - Tasks can be executed in parallel
- Partitions and executors
  - All items of one partition are on the same executor
  - An executor can process multiple partitions

More on Partitioning

- Number of partitions
  - RDDs are automatically partitioned based on the configuration of the target platform (nodes, CPUs)
  - As many partitions as the number of available cores
- If the input data are already partitioned:
  - Same number of partitions as in the input data
  - Example: RDD from HDFS file – 1 partition per HDFS block
- The number of partitions in a RDD can be changed by the programmer
  - repartition(): change the number of partitions
  - coalesce(): merge partitions

Data Distribution in Partitions

Two default partitioners

- Range partitioner
  - Default partitioner for raw data
  - Consecutive items are put in the same partition
- Hash partitioner
  - Applied after "ByKey" operations
  - *partition = key.hashCode() mod numPartitions*
- The user can define its own partitioning function

RDD Dependencies

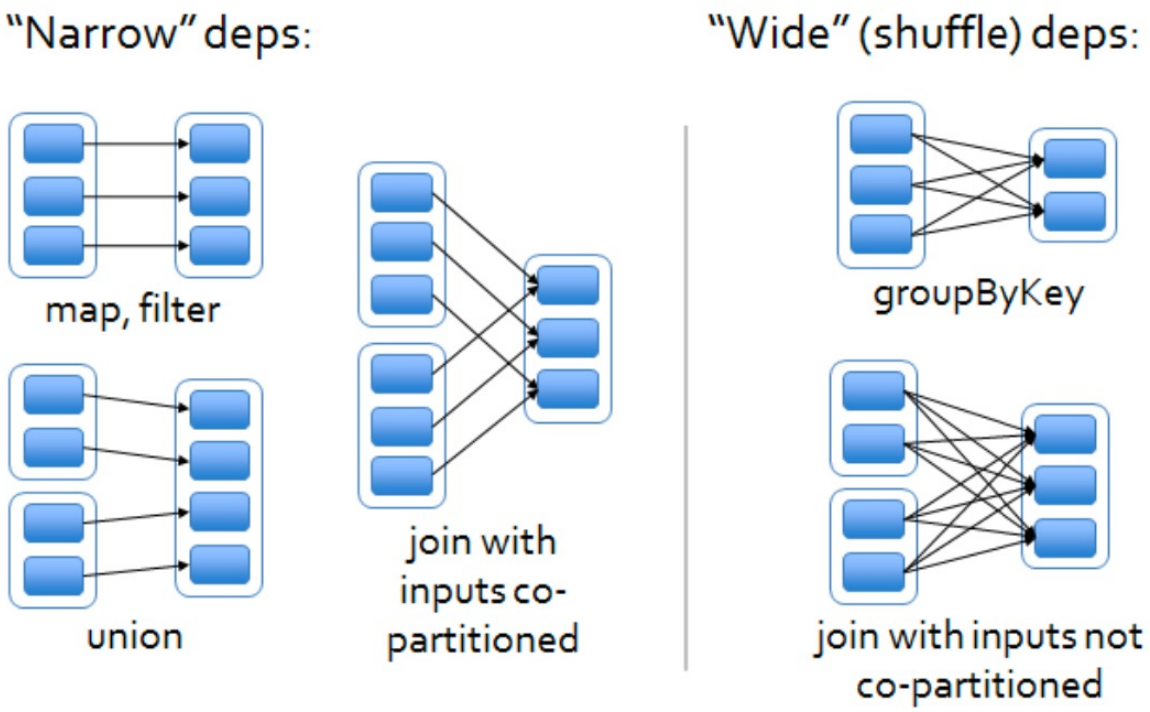
Transformations create dependencies between RDDs : 2 kinds

- Narrow dependencies: Each partition in the parent is used by at most one partition in the child
- Wide (shuffle) dependencies: Each partition in the parent is used by multiple partitions in the child

- Impact of dependencies
  - Scheduling: Which tasks can be run independently
  - Fault tolerance: Which partitions are needed to recreate a lost partition

RDD Dependencies

Figure by M. Zaharia et al



Lazy Evaluation

- Transformations are executed only when an action is called on the corresponding RDD
- Examples of optimizations allowed by lazy evaluation
  - Read file from disk + action first(): no need to read the whole file
  - Read file from disk + transformation filter(): No need to create an intermediate object that contains all lines

About Shuffle Operations

Costly operations

- Triggered by:
  - ByKey operations
  - repartition operations
  - etc.
- May involves significant communication over the network
- Involves disk I/O operations
  - In each source partition, data split by destination partitions are saved to disk.
  - Purpose: limit the number of operations to re-execute in case of crash

Persist a RDD

Main idea

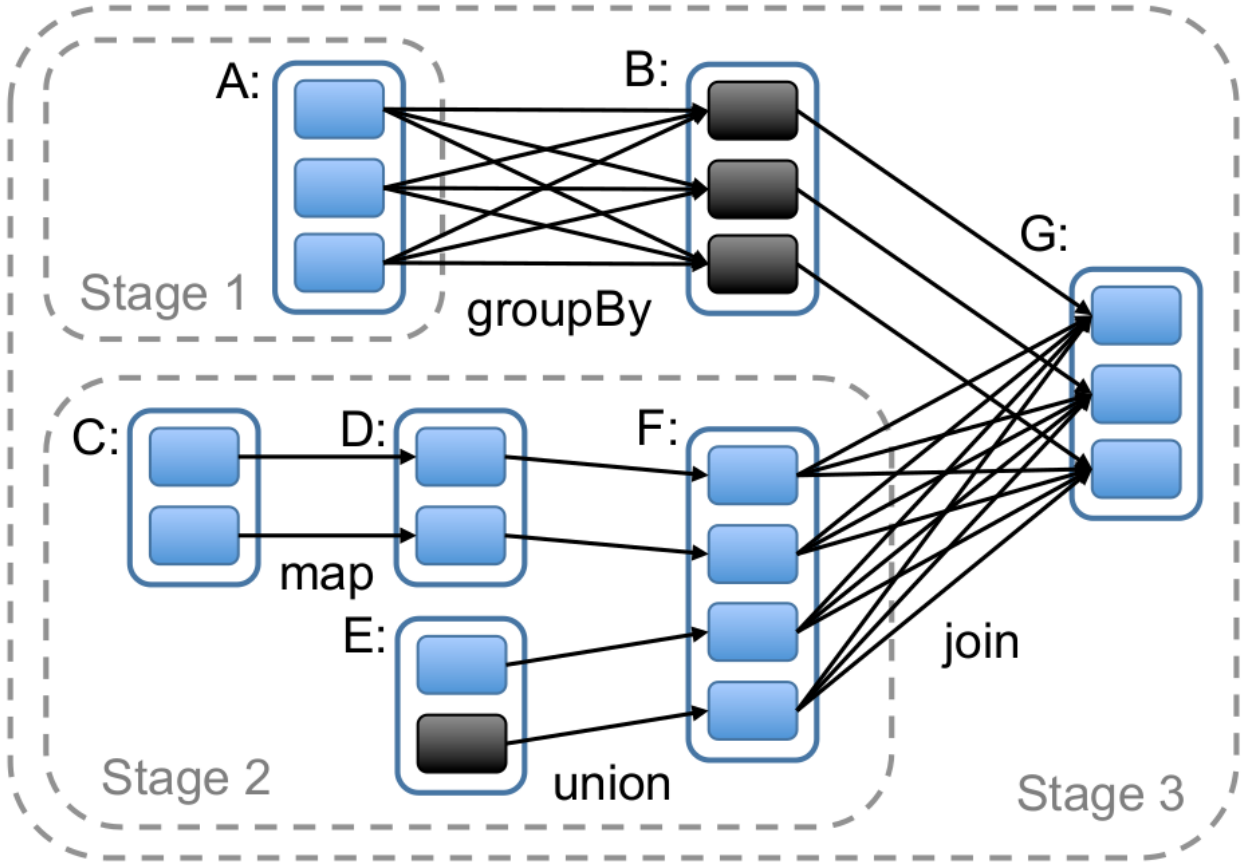
- By default, a RDD is recomputed for each action run on it.
- A RDD can be cached in memory calling persist() or cache()
  - Useful is multiple actions to be run on the same RDD (iterative algorithms)
  - Can lead to 10X speedup
  - Note that a call to persist does not trigger transformations evaluation
  - cache() means that data have to be persisted in memory

Scheduling

- Tasks are run when the user calls an action
- A Directed Acyclic Graph (DAG) of transformations is built based on the RDD's lineage
- The DAG is divided into stages. Boundaries of a stage defined by:
  - Wide dependencies
  - Already computed RDDs
- Tasks are launched to compute missing partitions from each stage until target RDD is computed
  - Data locality is taken into account when assigning tasks to workers

Stages in a RDD's DAG

Figure by M. Zaharia et al



Cached partitions in black

Spark Context

What is it?

- Object representing a connection to an execution cluster
- We need a SparkContext to build RDDs

Creation

- Automatically created when running in shell (variable sc)
- To be initialized when writing a standalone application

Initialization

- Run in local mode with nb threads = nb cores: `local[*]`
- Run in local mode with 2 threads: `local[2]`
- Run on a spark cluster: `spark://HOST:PORT`

SparkContext

- Python shell
- `$ pyspark --master local[*]`

- Python program
- ```
  import pyspark
  sc = pyspark.SparkContext("local[*]")
  ```
  ```

The first RDDs

Create RDD from existing iterator

- Use `SparkContext.parallelize()`
  - Optional second argument to define the number of partitions

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Create RDD from a file

- Utilisation de `SparkContext.textFile()`

```
data = sc.textFile("myfile.txt")
hdfsData = sc.textFile("hdfs://myhdfsfile.txt")
```

Some Transformations

https: //spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

- **map(f):** Applies f to all elements of the RDD. f generates a single item
- **flatMap(f):** Same as map but f can generate 0 or several items
- **filter(f):** New RDD with the elements for which f returns true
- **union(other)/intersection(other):** New RDD being the union/intersection of the initial RDD and other.
- **cartesian(other):** When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)
- **distinct**(): New RDD with the distinct elements
- **repartition(n):** Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them

Some transformations with <K,V> pairs

- **groupByKey():** When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
- **reduceByKey(f):** When called on a dataset of (K, V) pairs, Merge the values for each key using an associative and commutative reduce function.
- **aggregateByKey**(): see documentation
- **join**(other): Called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.

Some actions

https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

- **reduce(f):** Aggregate the elements of the dataset using f (takes two arguments and returns one).
- **collect()**: Return all the elements of the dataset as an array.
- **count()**: Return the number of elements in the dataset.
- **take(n)**: Return an array with the first n elements of the dataset.
- **countByKey():** Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

Example

```
from pyspark.context import SparkContext
sc = SparkContext("local")

# define a first RDD
lines = sc.textFile("data.txt")
# define a second RDD
lineLengths = lines.map(lambda s: len(s))
# Make the RDD persist in memory
lineLengths.persist()
# At this point no transformation has been run
# Launch the evaluation of all transformations
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

Example with tuples <K,V>

```
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)

# Warning: sortByKey implies shuffle
result = counts.sortByKey().collect()
```

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
# mapValues applies f to each value
#       without changing the key
sorted(rdd.groupByKey().mapValues(len).collect())
# [('a', 2), ('b', 1)]
sorted(rdd.groupByKey().mapValues(list).collect())
# [('a', [1, 1]), ('b', [1])]
```

On Distributed Execution

```
1   counter = 0
2   rdd = sc.parallelize(data)
3
4   def increment_counter(x):
5       global counter
6       counter += x
7
8   rdd.foreach(increment_counter)
9
10  print("Counter value: ", counter) # displays 0
```

What is the problem?

On Distributed Execution

```
1   counter = 0
2   rdd = sc.parallelize(data)
3
4   def increment_counter(x):
5       global counter
6       counter += x
7
8   rdd.foreach(increment_counter)
9
10  print("Counter value: ", counter) # displays 0
```

What is the problem?

- We have multiple JVMs, and so, multiple *counter* variables
  - *counter* in lines 1 and 10 is in the JVM of the driver
  - In lines 5 and 8, we create one counter per executor JVM

Shared Variables

- Accumulator
  - Use-case: Accumulate values over all tasks
  - Declare an Accumulator on the driver
  - Updates by the tasks are automatically propagated to the driver.
- Default accumulator: operator '+=' on int and float.
  - User can define custom accumulator functions

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def splitLine(line):
    # Make the global variable accessible
    global blankLines
    if not line:
        blankLines += 1
    return line.split("␣")

words = file.flatMap(splitLine)
print(blankLines.value)
```