Spark Technology

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. Basic programming examples
6. Basic examples on pair RDDs
7. PageRank with Spark

1 - Spark main objectives

Spark has been designed:
• To efficiently run iterative and interactive applications
  ➔ keeping data in-memory between operations
• To provide a low-cost fault tolerance mechanism
  ➔ low overhead during safe executions
  ➔ fast recovery after failure
• To be easy and fast to use in interactive environment
  ➔ Using compact Scala programming language
• To be « scalable »
  ➔ able to efficiently process bigger data on larger computing clusters

Spark is based on a distributed data storage abstraction:
− the « RDD » (Resilient Distributed Datasets)
− compatible with many distributed storage solutions

Spark Technology

2 - RDD concepts and operations

A RDD (Resilient Distributed Dataset) is:
• an immutable (read only) dataset
• a partitioned dataset
• usually stored in a distributed file system (like HDFS)

When stored in HDFS:
− One RDD ➔ One HDFS file
− One RDD partition block ➔ One HDFS file block
− Each RDD partition block is replicated by HDFS
2 - RDD concepts and operations

Example of a 4 partition blocks stored on 2 data nodes (no replication)

Source: http://images.backtobazics.com/

Example of Transformations and Actions

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Initial input RDDs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(x)</td>
<td>• are usually created from distributed files (like HDFS files),</td>
</tr>
<tr>
<td></td>
<td>• Spark processes read the file blocks that become in-memory RDD</td>
</tr>
<tr>
<td></td>
<td>Operations on RDDs:</td>
</tr>
<tr>
<td></td>
<td>• Transformations: read RDDs, compute, and generate a new RDD</td>
</tr>
<tr>
<td></td>
<td>• Actions: read RDDs and generate results out of the RDD world</td>
</tr>
<tr>
<td></td>
<td>Map and Reduce are parts of the operations</td>
</tr>
</tbody>
</table>

Source: Stack Overflow

Fault tolerance:
- Transformation are coarse grained op: they apply on all data of the source RDD
- RDD are read-only, input RDD are not modified
- A sequence of transformations (a lineage) can be easily stored
  - In case of failure: Spark has just to re-apply the lineage of the missing RDD partition blocks.

Source: Stack Overflow

5 main internal properties of a RDD:
- A list of partition blocks
- A function for computing each partition block
- A list of dependencies on other RDDs: parent RDDs and transformations to apply

Optionally:
- A Partitioner for key-value RDDs: metadata specifying the RDD partitioning
- A list of nodes where each partition block can be accessed faster due to data locality

To compute and re-compute the RDD when failure happens

To control the RDD partitioning, to achieve co-partitioning...

To improve data locality with HDFS & YARN...

Narrow transformations
- Local computations applied to each partition block
  - no communication between processes/nodes
  - only local dependencies (between parent & son RDDs)

- In case of sequence of Narrow transformations:
  - possible pipelining inside one step

2 - RDD concepts and operations

Narrow transformations
- Local computations applied to each partition block
  → no communication between processes/nodes
  → only local dependencies (between parent & son RDDs)
  *Map()*
  *Filter()*
  *Union()*
- In case of failure:
  → recompute only the damaged partition blocks
  → recompute/reload only its parent blocks

Wide transformations
- Computations requiring data from all parent RDD blocks
  → many communication between processes/nodes (shuffle & sort)
  → non-local dependencies (between parent & son RDDs)
  *groupByKey()*
  *reduceByKey()*
- In case of sequence of transformations:
  → no pipelining of transformations
  → wide transformation must be totally achieved before to enter
  next transformation

Avoiding wide transformations with co-partitioning
- With identical partitioning of inputs:
  → wide transformation → narrow transformation
- less expensive communications
- possible pipelining
- less expensive fault tolerance

Persistence of the RDD
- RDD are stored:
  - in the memory space of the Spark Executors
  - or on disk (of the node) when memory space of the Executor is full
- By default: an old RDD is removed when memory space is required
  (Least Recently Used policy)
  → An old RDD has to be re-computed (using its lineage) when needed again
  → Spark allows to make a « persistent » RDD to avoid to recompute it

Persistence of the RDD to improve Spark application performances
Spark application developer has to add instructions to force RDD
storage, and to force RDD forgetting:

```scala
myRDD.persist(StorageLevel) // or myRDD.cache()
myRDD.unpersist()
```

Available storage levels:
- MEMORY_ONLY : in Spark Executor memory space
- MEMORY_ONLY_SER : + serializing the RDD data
- MEMORY_AND_DISK : on local disk when no memory space
- MEMORY_AND_DISK_SER : + serializing the RDD data in memory
- DISK_ONLY : always on disk (and serialized)

RDD is saved in the Spark executor memory/disk space
→ limited to the Spark session
2 - RDD concepts and operations

Persistence of the RDD to improve fault tolerance

To face short term failures: Spark application developer can force RDD storage with replication in the local memory/disk of several Spark Executors

```scala
myRDD.sparkContext.setCheckpointDir(directory)
myRDD.checkpoint()
```

To face serious failures: Spark application developer can checkpoint the RDD outside of the Spark data space, on HDFS or S3 or...

```scala
myRDD.sparkContext.setCheckpointDir(directory)
myRDD.checkpoint()
```

- Longer, but secure!

### Spark Technology

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. Basic programming examples
6. Basic examples on pair RDDs
7. PageRank with Spark

3 – SPARK application scheme and execution

**Transformations** are lazy operations: saved and executed further

**Actions** trigger the execution of the sequence of transformations

A **job** is a sequence of RDD transformations, ended by an action

A **Spark application** is a set of jobs to run sequentially or in parallel

The **Spark application driver** controls the application run

- It creates the Spark context
- It analyses the Spark program
- It creates a DAG of tasks for each job
- It optimizes the DAG
  - pipelining narrow transformations
  - identifying the tasks that can be run in parallel
- It schedules the DAG of tasks on the available worker nodes (the Spark Executors) in order to maximize parallelism (and to reduce the execution time)

3 – SPARK application scheme and execution

The Spark application driver controls the application run

- It attempts to keep in-memory the intermediate RDDs in order the input RDDs of a transformation are already in-memory (ready to be used)
- A RDD obtained at the end of a transformation can be explicitly kept in memory, when calling the persist() method of this RDD (interesting if it is re-used further).
Spark cluster configuration:
- Add the list of cluster worker nodes in the Spark Master config.
- Specify the maximum amount of memory per Spark Executor:
  `spark-submit --executor-memory XX ...`
- Specify the total amount of CPU cores used to process one Spark application (through all its Spark executors):
  `spark-submit --total-executor-cores YY ...`

Client deployment mode:
- Interactive control of the application: development mode

Cluster deployment mode:
- Laptop connection can be turned off: production mode

The Cluster Worker nodes should be the Data nodes, storing initial RDD values or new generated (and saved) RDD:
- Will improve the global data-computations locality
- When using HDFS: the Hadoop data nodes should be re-used as worker nodes for Spark Executors.

When using the Spark Master as Cluster Manager:
...there is no way to localize the Spark Executors on the data nodes hosting the right RDD blocks!
4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)

```
spark-submit --master spark://node:port ... myApp
```

Cluster deployment mode:

- Spark Master ➔ Cluster Manager
- HDFS Name Node
- Spark master node & Hadoop Data Node
- Cluster worker node & Hadoop Data Node
- Cluster worker node & Hadoop Data Node

Strength and weakness of standalone mode:

- Nothing more to install (included in Spark)
- Easy to configure
- Can run different jobs concurrently
- Can not share the cluster with non-Spark applications
- Can not launch Executors on data node hosting input data
- Limited scheduling mechanism (unique queue)

2 - with YARN cluster manager

```
export HADOOP_CONF_DIR = ${HADOOP_HOME}/conf
spark-submit --master yarn ... myApp
```

Spark cluster configuration:

- By default:
  - (only) 1GB/Spark Executor
  - (only) 1 CPU core per Spark Executor
  - (only) 2 Spark Executors per job
- Usually better with few large Executors (RAM & nb of cores)...

Spark Context construction:

```
val sc = new SparkContext(sparkConf, 
inputformatInfo.computePreferredLocations(), 
Seq(new InputFormatInfo(conf, classOf[org.apache.hadoop.mapred.TextInputFormat], hdfspath))...) 
```

Client deployment mode:

- Spark Driver
- DAG scheduler-optimizer
- Task scheduler
- App. Master
- Executor launcher

Spark cluster configuration:

- Link Spark RDD meta-data «preferred locations» to HDFS meta-data about «localization of the input file blocks»
Export HADOOP_CONF_DIR = ${HADOOP_HOME}/conf

spark-submit --master yarn ...

– Application execution on clusters and clouds
2 - with YARN cluster manager

Client deployment mode:

Spark Driver
- (DJ builder
- DAG scheduler- optimizer
- Task scheduler

App. Master
- Executor + launcher

HDFS Name Node

YARN Resource Manager

Cluster worker node & Hadoop Data Node

YARN vs standalone Spark Master:
- Usually available on HADOOP/HDFS clusters
- Allows to run Spark and other kinds of applications on HDFS
- Advanced application scheduling mechanisms
  (multiple queues, managing priorities…)

YARN Resource Manager

Cluster worker node & Hadoop Data Node

HDFS Name Node

App. Master / Spark Driver
- DAG scheduler-optimizer
- Task scheduler

HDFS Name Node

Spark executor

App. Master

Spark executor

HDFS Name Node

Spark executor

HDFS Name Node

Spark executor

HDFS Name Node

Spark executor

HDFS Name Node

Spark executor

YARN vs standalone Spark Master:
- Improvement of the data-computation locality… but is it critical?
  - Spark reads/writes only input/output RDD from Disk/HDFS
  - Spark keeps intermediate RDD in-memory
  - With cheap disks: disk-I/O time > network time
  - Better to deploy many Executors on unloaded nodes?

spark-submit --master mesos://node:port ...

– Application execution on clusters and clouds
3 - with MESOS cluster manager

Mesos is a generic cluster manager
- Supporting to run both:
  - short term distributed computations
  - long term services (like web services)
- Compatible with HDFS

spark-submit --executor-memory XX ...

spark-submit --total-executor-cores YY ...

• Specify the maximum amount of memory per Spark Executor
  spark-submit --executor-memory XX...

• Specify the total amount of CPU cores used to process one Spark application (through all its Spark executors)
  spark-submit --total-executor-cores YY ...

• Default config:
  - create few Executors with max nb of cores (≠ standalone…)
  - use all available cores to process each job (like standalone…)
4 – Application execution on clusters and clouds

3 - with MESOS cluster manager

spark-submit --master mesos://node:port ... myApp

- Client deployment mode:
  - Spark Driver
  - DAG builder
  - Task scheduler

- With just Mesos:
  - No Application Master
  - No Input Data – Executor locality

- Cluster deployment mode:
  - Spark Driver
  - DAG builder
  - Task scheduler

Coarse grained mode: number of cores allocated to each Spark Executor are set at launching time, and cannot be changed

Fine grained mode: number of cores associated to an Executor can dynamically change, function of the number of concurrent jobs and function of the load of each executor

Better solution/mechanism to support many shell interpreters
But latency can increase (Spark Streaming lib can be disturbed)

4 – on Amazon Elastic Compute Cloud «EC2»

spark-ec2 ... -s <#nb of slave nodes> -t <type of slave nodes> launch MyCluster-1

- Spark app. Driver
- DAG builder
- Task scheduler

Spark Streaming

Better solution/mechanism to support many shell interpreters
But latency can increase (Spark Streaming lib can be disturbed)
4 – Application execution on clusters and clouds

4 – on Amazon Elastic Compute Cloud «EC2»

spark-ec2 destroy MyCluster-2

Standalone Spark Master

HDFS Name Node

Spark Master

Task scheduler

DAG scheduler-optimizer

DAG builder

Spark app. Driver

Spark executor

spark-ec2 launch MyCluster-1

spark-ec2 get-master MyCluster-1 ➔ MasterNode

cp - myApp.jar root@MasterNode

spark-ec2 login MyCluster-1

spark-submit --master spark://node:port ... myApp

spark-ec2 stop MyCluster-1

spark-ec2 start MyCluster-1 ➔ Restart billing

spark-ec2 destroy MyCluster-1

Machines in a HPC cloud

(standard)

(5x less expensive!)

require to support to loose some tasks, or to checkpoint...

Choose to allocate reliable or preemptible machines:

• Reliable machines during all the session

• Preemptibles machines (5x less expensive!)

…or you can use a “Spark Cluster service” ready to use in a CLOUD!

Lear to minimize the cost (€) of a Spark cluster

• Allocate the right number of nodes

• Stop when you do not use, and re-start further

5 – Basic programming examples

Ex. of transformations on one RDD:

\[
\text{rdd : \{1, 2, 3, 3\}}
\]

Python: rdd.map(lambda x: x+1) ➔ rdd : \{2, 3, 4, 4\}
Scala : rdd.map(x => x+1) ➔ rdd : \{2, 3, 4, 4\}

Scala : rdd.map(x => x.to(3)) ➔ rdd : \{1,2,3,2,3,3,3\}
Scala : rdd.flatMap(x => x.to(3)) ➔ rdd : \{1, 2, 3, 2, 3, 3, 3\}
Scala : rdd.filter(x => x != 1) ➔ rdd : \{2, 3, 3\}
Scala : rdd.distinct() ➔ rdd : \{1, 2, 3\}
Scala : rdd.sample(false,0.5) ➔ rdd : \{1\} or \{2,3\} or ...
Scala : rdd.filter(x => x != 1).map(x => x+1) ➔ rdd : \{3, 4, 4\}

Some sampling functions exist:

Scala : rdd.sample(true,0.5) ➔ rdd : \{1,2\} or \{3\} or ...

with replacement + true

Sequence of transformations:

Scala : rdd.filter(x => x != 1).map(x => x+1) ➔ rdd : \{3, 4, 4\}

Spark Technology

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. Basic programming examples
6. Basic examples on pair RDDs
7. PageRank with Spark
### 5 – Basic programming examples

**Ex. of transformations on two RDDs:**

- Scala: `rdd.union(rdd2)` → `rdd: {1, 2, 3, 3, 4, 5}
- Scala: `rdd.intersection(rdd2)` → `rdd: {3}
- Scala: `rdd.subtract(rdd2)` → `rdd: {1, 2}
- Scala: `rdd.cartesian(rdd2)` → `rdd: {(1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,3), (3,4), (3,5)}

**Ex. of actions on a RDD:**

- Examples of « aggregations »: computing a sum
  - Scala: `rdd.reduce((x,y) => x+y)` → `9`
  - Scala: `rdd.foldLeft(0)((accu,value) => accu+value)` → `9`
- Specifying the initial value of the accumulator:
  - Scala: `rdd.foldRight(0)((accu,value) => accu+value)` → `9`

**Ex. of actions on a RDD:**

- Scala: `rdd.collect()` → `{1, 2, 3, 3}
- Scala: `rdd.count()` → `4`
- Scala: `rdd.countByValue()` → `{(1,1), (2,1), (3,2))
- Scala: `rdd.take(2)` → `{1, 2}
- Scala: `rdd.top(2)` → `{3, 3}
- Scala: `rdd.takeOrdered(3,Ordering[Int].reverse)` → `{3,3,2}
- Scala: `rdd.takeSample(false,2)` → `{?, ?}`

### 6 – Basic examples on pair RDDs

**Ex. of transformations on one RDD:**

- Scala: `rdd.reduceByKey((x,y) => x+y)` → `rdd: {(1,2), (3,7)}

**Ex. of actions on a RDD:**

- Scala: `rdd.reduceByKey((x,y) => x+y)` → `rdd: {1, 2, 3, 3, 4, 5}

**Spark Technology**

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. Basic programming examples
6. Basic examples on pair RDDs
7. PageRank with Spark
### Ex. of transformations on one RDD

**Scala**:

- `rdd.keys()`: Returns an RDD of just the keys
  - `rdd: {1, 3, 3}`
- `rdd.values()`: Returns an RDD of just the values
  - `rdd: {2, 3, 4}`
- `rdd.sortByKey()`: Returns a pair RDD sorted by the keys
  - `rdd: {(1, 2), (3, 3), (3, 4)}`

**Scala**:

- `rdd.combineByKey`:
  - `reduceCombiner = Hadoop Combiner`
  - `mergeValue = Hadoop Reduce` (Voir plus loin...)

### Ex. of classic transformations applied on a pair RDD

A pair RDD remains a RDD of tuples (key, values)

- `rdd.filter{case (k,v) => v < 5}`
  - `rdd: {(1, 2), (3, 4)}`
- `rdd.map{case (k,v) => (k,v*10)}`
  - `rdd: {(1, 20), (3, 40), (3, 60)}`

### Ex. of transformation: Computing an average value per key

#### Solution 1: `mapValues + reduceByKey + collectAsMap + foreach`

```scala
def mapValues = theMarks.mapValues(v => (v, 1)).reduceByKey((vc1, vc2) => (vc1._1 + vc2._1, vc1._2 + vc2._2)).collectAsMap
```

```
theSums.foreach(
  kvc => println(kvc._1 + " has average:" + kvc._2._1/kvc._2._2.toDouble))
```

### Ex. of transformation: Computing an average value per key

#### Solution 2: `combineByKey + collectAsMap + foreach`

```scala
def combineByKey1(
  createCombiner = (valueWithNewKey) => (valueWithNewKey, 1),
  mergeValue = (acc:(Int, Int), v) => (acc._1 + v, acc._2 + 1),
  mergeCombiners = (acc1:(Int, Int), acc2:(Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
).collectAsMap
```

```
theSums.foreach(
  kvc => println(kvc._1 + " has average:" + kvc._2._1/kvc._2._2.toDouble))
```

### Ex. of transformations on two pair RDDs

**Ex. of subtractByKey**

- `rdd.subtractByKey(rdd2)`
  - `rdd: {(1,2)}`
  - Remove pairs with key present in the 2nd pair RDD

**Ex. of join**

- `rdd.join(rdd2)`
  - `rdd: {(3, (4, 9)), (3, (6, 9))}`
  - Inner Join between the two pair RDDs

**Ex. of cogroup**

- `rdd.cogroup(rdd2)`
  - Group data from both RDDs sharing the same key
    - `rdd: {(1, ([2], [ ])), (3, ([4, 6], [9]))}`

### Ex. of actions on pair RDDs

- `rdd.countByKey()`
  - `rdd: {(1,1), (3,2)}`
  - Return a tuple of couple, counting the number of pairs per key

- `rdd.collectAsMap()`
  - `Map{(1,2), (3,4), (3,6)}`
  - Return a 'Map' datastructure containing the RDD

- `rdd.lookup(3)`
  - `[4, 6]`
  - Return an array containing all values associated with the provided key

### Ex. of transformations on one RDD

- `rdd: {(1, 2), (3, 3), (3, 4)}`

### Ex. of classic transformations applied on a pair RDD

- `rdd: {(1, 2), (3, 4), (3, 6)}`

**Ex. of classic transformations**

- `rdd.filter{case (k,v) => v < 5}`
  - `rdd: {(1, 2), (3, 4)}`
- `rdd.map{case (k,v) => (k,v*10)}`
  - `rdd: {(1, 20), (3, 40), (3, 60)}`

### Ex. of actions on pair RDDs

- `rdd: {(1, 2), (3, 4), (3, 6)}`

**Ex. of actions**

- `rdd.countByKey()`
  - `rdd: {(1,1), (3,2)}`
  - Return a tuple of couple, counting the number of pairs per key

- `rdd.collectAsMap()`
  - `Map{(1,2), (3,4), (3,6)}`
  - Return a 'Map' datastructure containing the RDD

- `rdd.lookup(3)`
  - `[4, 6]`
  - Return an array containing all values associated with the provided key

### Ex. of transformation: Computing an average value per key

#### Solution 1: `mapValues + reduceByKey + collectAsMap + foreach`

```scala
def mapValues = theMarks.mapValues(v => (v, 1)).reduceByKey((vc1, vc2) => (vc1._1 + vc2._1, vc1._2 + vc2._2)).collectAsMap
```

```
theSums.foreach(
  kvc => println(kvc._1 + " has average:" + kvc._2._1/kvc._2._2.toDouble))
```

### Ex. of transformation: Computing an average value per key

#### Solution 2: `combineByKey + collectAsMap + foreach`

```scala
def combineByKey1(
  createCombiner = (valueWithNewKey) => (valueWithNewKey, 1),
  mergeValue = (acc:(Int, Int), v) => (acc._1 + v, acc._2 + 1),
  mergeCombiners = (acc1:(Int, Int), acc2:(Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
).collectAsMap
```

```
theSums.foreach(
  kvc => println(kvc._1 + " has average:" + kvc._2._1/kvc._2._2.toDouble))
```
6 – Basic examples on pair RDDs

Ex. of transformation: Computing an average value per key

```scala
val theMarks = theMarks
  .combineByKey(
    // createCombiner function
    (valueWithNewKey) => (valueWithNewKey, 1),
    // mergeValue function (inside a partition block)
    (acc:(Int, Int), v) => (acc._1 + v, acc._2 + 1),
    // mergeCombiners function (after shuffle comm.)
    (acc1:(Int, Int), acc2:(Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
  .map{case (k,vc) => (k, vc._1/vc._2.toDouble)}
```

theSums.collectAsMap().foreach(
  kv => println(kv._1 + " has average: " + kv._2))

TheSums: {('julie', 12), ('marc', 10), ('albert', 19), ('julie', 15), ('albert', 15)...}

Solution 2: combineByKey + map + collectAsMap + foreach

```
10/05/2019
```

6 – Basic examples on pair RDDs

Tuning the level of parallelism

- By default: level of parallelism set by the nb of partition blocks of the input RDD
- When the input is a in-memory collection (list, array...), it needs to be parallelized:
  ```scala
  val theData = List(("a",1), ("b",2), ("c",3),...)
  sc.parallelize(theData).theTransformation(...)
  Or:
  val theData = List(1,2,3,...).par
  theData.theTransformation(...)
  ```

Spark adopts a distribution adapted to the cluster... but it can be tuned

6 – Basic examples on pair RDDs

Tuning the level of parallelism

- Most of transformations support an extra parameter to control the distribution (and the parallelism)
- Example:
  ```scala
  val theData = List(("a",1), ("b",2), ("c",3),...)
  sc.parallelize(theData).reduceByKey((x,y) => x+y)
  ```
  ```scala
  def reduceByKey(key=>x+y): RDD[(K1, V1)]
  ```

- Default parallelism: 8 partition blocks imposed for the result of the reduceByKey

Spark Technology

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. Basic programming examples
6. Basic examples on pair RDDs
7. PageRank with Spark

6 – PageRank with Spark

PageRank objectives

Compute the probability to arrive at a web page when randomly clicking on web links...

- If a URL is referenced by many other URLs then its rank increases (because being referenced means that it is important — ex: URL 1)
- If an important URL (like URL 1) references other URLs (like URL 4) this will increase the destination's ranking

PageRank principles

- Simplified algorithm:
  ```latex
  PR(u) = \sum_{v\in S(u)} \frac{PR(v)}{L(v)}
  ```
  ```latex
  \text{PR}(_u): \text{the set containing all pages linking to page } u
  \text{PR}(_v): \text{PageRank of page } v
  L(_v): \text{the number of outbound links of page } v
  ```

- Initialize the PR of each page with an equi-probability
- Iterate k times: compute PR of each page
PageRank principles

- The damping factor:
  the probability a user continues to click is a damping factor: $d$

$$PR(v) = \frac{1-d}{|N_{out}|} + d \sum_{u \in N_{out}} \frac{PR(u)}{L(u)}$$

Sum of all PR is 1

Variant:

$$PR(v) = (1-d) + d \sum_{u \in N_{out}} \frac{PR(u)}{L(u)}$$

Sum of all PR is $N_{pages}$

PageRank second step in Spark (Scala)

```scala
val links = // links RDD -> ranks RDD
val ranks = links.mapValues(v => 1.0)
```

Other strategy:

Initialization with $1/N$ equi-probability:

```scala
links.mapValues(_ => 1.0 / links.mapValues(_ => 1.0).count)
```

Sparc & Scala allow a short/compact implementation of the PageRank algorithm

Each RDD remains in-memory from one iteration to the next one

PageRank third step in Spark (Scala)

```scala
val contribs = links.join(ranks).values.flatMap{ case (urls, rank) =>
  // links RDD & contributions
  urls.map(url => (url, rank / urls.size))
}.reduceByKey(mapValues(0.15 + 0.85 * _))
```

Spark Technology