




Big Data

Spark optimizations

Stéphane Vialle & Gianluca Quercini



1

Spark optimizations

1. **Wide and Narrow transformations**
2. Optimizations
3. *Page Rank* example

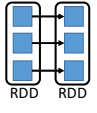
2

Wide and Narrow transformations

Narrow transformations


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

• Map()
• Filter()

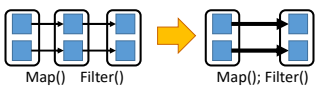


RDD RDD

• Union()



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



Map() Filter()

→

Map(); Filter()

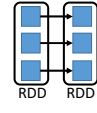
3

Wide and Narrow transformations

Narrow transformations


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

• Map()
• Filter()

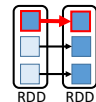


RDD RDD

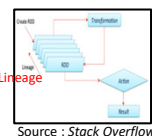
• Union()



- In case of failure:
→ recompute only the damaged partition blocks
→ recompute/reload only its parent blocks



RDD RDD



Source : Stack Overflow

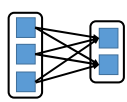
4

Wide and Narrow transformations

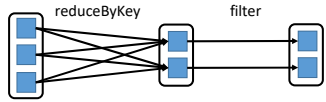
Wide transformations

- Computations requiring data from all parent RDD blocks
→ many comms between processes (and 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



reduceByKey filter

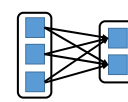
5

Wide and Narrow transformations

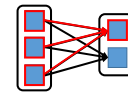
Wide transformations

- Computations requiring data from all parent RDD blocks
→ many comms between processes (and nodes) (*shuffle & sort*)
→ non-local dependencies (between parent & son RDDs)

• groupByKey()
• reduceByKey()



- In case of sequence of failure:
→ recompute the damaged partition blocks
→ recompute/reload all blocks of the parent RDDs



6

Wide and Narrow transformations

Avoiding wide transformations with **co-partitioning**

- With identical partitioning of inputs:
 - wide transformation → narrow transformation

Join with inputs not co-partitioned

Join with inputs co-partitioned

- less expensive communications
- possible pipelining
- less expensive fault tolerance

Control RDD partitioning
 Force co-partitioning
 (using the same partition map)

7

Spark optimizations

- Wide and Narrow transformations
- Optimizations**
 - RDD Persistence**
 - RDD Co-partitioning
 - RDD controlled distribution
 - Traffic minimization
 - Maintaining parallelism
- Page Rank example

8

Optimizations: persistence

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

Source : Stack Overflow

9

Optimizations: persistence

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:

```
myRDD.persist(StorageLevel) // or myRDD.cache()
... // Transformations and Actions
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

10

Optimizations: persistence

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**

```
myRDD.persist(storageLevel.MEMORY_AND_DISK_SER_2)
... // Transformations and Actions
myRDD.unpersist()
```

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

```
myRDD.sparkContext.setCheckpointDir(directory)
myRDD.checkpoint()
... // Transformations and Actions
```

→ Longer, but secure!

11

Spark optimizations

- Wide and Narrow transformations
- Optimizations**
 - RDD Persistence
 - RDD Co-partitioning**
 - RDD controlled distribution
 - Traffic minimization
 - Maintaining parallelism
- Page Rank example

12

Optimizations: RDD co-partitioning

5 main internal properties of a RDD:

- A list of partition blocks
`getPartitions()`
- A function for computing each partition block
`compute(...)`
- A list of dependencies on other RDDs: parent RDDs and transformations to apply
`getDependencies()`

Optionally:

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

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...

13

Optimizations: RDD co-partitioning

Specify a « partitioner »

```
val rdd2 = rdd1
    .partitionBy(new HashPartitioner(100))
    .persist()
```

Creates a new RDD (rdd2):

- Partitionned according to hash partitionner strategy
- On 100 Spark Executors
- Redistribute the RDD (rdd1 → rdd2)
- WIDE (expensive) transformation
- Do not keep the original partition (rdd1) in memory / on disk
- Keep the new partition (rdd2) in memory / on disk
- to avoid to repeat a WIDE transformation when rdd2 is re-used

14

Optimizations: RDD co-partitioning

Specify a « partitioner »

```
val rdd2 = rdd1
    .partitionBy(new HashPartitioner(100))
    .persist()
```

Partitioners:

- Hash partitioner :**
Key0, Key0+100, Key0+200... on one Spark Executor
- Range partitioner :**
[Key-min ; Key-max] on one Spark Executor
- Custom partitioner (develop your own partitioner) :**
Ex : Key = URL, hash partitioned
BUT : hash only the domain name of the URL
→ all pages of the same domain on the same Spark Executor because they are frequently linked

15

Optimizations: RDD co-partitioning

Avoid repetitive WIDE transformations on large data sets

→ Control partitioning to avoid many Wide ops

- Explicit & structured re-partitioning of A → A' Will propagate to the join result
- Create B with A' partitioning

16

Optimizations: RDD co-partitioning

Avoid repetitive WIDE transformations on large data sets

→ Control partitioning to avoid many Wide ops

- Explicit & structured re-partitioning of A → A' Will propagate to the join result
- Create B with A' partitioning

17

Optimizations: RDD co-partitioning

PageRank with partitioner (see further)

```
val links = ..... // previous code
val links1 = links.partitionBy(new HashPartitioner(100)).persist()
var ranks = links1.mapValues(v => 1.0)
for (i <- 1 to iters) {
    val contribs =
        links1.join(ranks)
            .flatMap{ case (url (urlLinks, rank)) =>
                urlLinks.map(dest => (dest, rank/urlLinks.size)) }
            ranks = contribs.reduceByKey(_ + _)
                .mapValues(0.15 + 0.85 * _)
}
```

- Initial links and ranks are co-partitioned
- Repeated join is Narrow-Wide
- Repeated mapValues is Narrow: respects the reduceByKey partitioning
- Pb: flatMap{...urlinks.map(...)} can change the partitioning ?!

18

Spark optimizations

1. Wide and Narrow transformations
2. **Optimizations**
 - RDD Persistence
 - RDD Co-partitioning
 - **RDD controlled distribution**
 - Traffic minimization
 - Maintaining parallelism
3. *Page Rank* example

19

Optimization: RDD distribution

Create and distribute a RDD

- 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:


```
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

20

Optimization: RDD distribution

Control of the RDD distribution

- Most of transformations support an **extra parameter** to control the distribution (and the parallelism)
- **Example:**

Default parallelism:

```
val theData = List(("a",1), ("b",2), ("c",3), ...)
sc.parallelize(theData).reduceByKey((x,y) => x+y)
```

Tuned parallelism:

```
val theData = List(("a",1), ("b",2), ("c",3), ...)
sc.parallelize(theData).reduceByKey((x,y) => x+y, 8)
```

But better to use PARTITIONERS... → 8 partition blocks imposed for the result of the reduceByKey

21

Spark optimizations

1. Wide and Narrow transformations
2. **Optimizations**
 - RDD Persistence
 - RDD Co-partitioning
 - RDD controlled distribution
 - **Traffic minimization**
 - Maintaining parallelism
3. *Page Rank* example

22

Optimization: traffic minimization

RDD redistribution:

Scala : `rdd.groupByKey()` → rdd: {(1, [2]), (3, [3, 4])}

Group values associated to the same key

→ Move almost all input data
→ Huge traffic in the shuffle step !!

`groupByKey` will be time consuming:

- no computation time...
- ... but huge traffic on the network of the cluster/cloud

→ Optimize computations **and** communications in a Spark program

23

Optimization: traffic minimization

RDD reduction:

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

Reduce values associated to the same key

`((x,y) => x+y)`:
1 int + 1 int → 1 int
→ Limited traffic in the shuffle step

But: `((x,y) => x+y)`:
1 list + 1 list → 1 longer list → TD-1

24

Optimization: traffic minimization

RDD reduction with different input and reduced datatypes:

```
Scala : rdd.aggregateByKey(init_acc) (
  ..., // mergeValueAccumulator fct
  ..., // mergeAccumulators fct
)
```

```
Scala : rdd.combineByKey(
  ..., // createAccumulator fct
  ..., // mergeValueAccumulator fct
  ..., // mergeAccumulators fct shuffle
)
```

See further

25

Spark optimizations

1. Wide and Narrow transformations
2. **Optimizations**
 - RDD Persistence
 - RDD Co-partitioning
 - RDD controlled distribution
 - Traffic minimization
 - **Maintaining parallelism**
3. Page Rank example

26

Optimization: maintaining parallelism

Computing an average value per key in parallel

```
theMarks: {"julie", 12}, {"marc", 10}, {"albert", 19}, {"julie", 15}, {"albert", 15}, ...}
```

- **Solution 1: mapValues + reduceByKey + collectAsMap + foreach**

```
val theSums = theMarks
  .mapValues(v => (v, 1))
  .reduceByKey((vc1, vc2) => (vc1._1 + vc2._1,
    vc1._2 + vc2._2))
  .collectAsMap() // Return a 'Map' datastructure
  // ACTION -> Break parallelism! Bad performances!
```

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

Sequential computing!

27

Optimization: maintaining parallelism

Computing an average value per key in parallel

```
theMarks: {"julie", 12}, {"marc", 10}, {"albert", 19}, {"julie", 15}, {"albert", 15}, ...}
```

- **Solution 2: combineByKey + collectAsMap + foreach**

```
val theSums = 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))
  .collectAsMap() // Still bad performances! (Break parallelism)
```

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

Still sequential!

Type inference needs some help!

28

Optimization: maintaining parallelism

Computing an average value per key in parallel

```
theMarks: {"julie", 12}, {"marc", 10}, {"albert", 19}, {"julie", 15}, {"albert", 15}, ...}
```

- **Solution 3: combineByKey + map + collectAsMap + foreach**

```
val theSums = 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))
```

Transformation: compute in parallel and return a RDD

```
theSums.collectAsMap().foreach( Action: at the end (just to print)
  kv => println(kv._1 + " has average:" + kv._2))
```

29

Spark optimizations

1. Wide and Narrow transformations
2. Optimizations
3. **Page Rank example**

30

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

31

PageRank with Spark

PageRank principles

- Simplified algorithm:

$$PR(u) = \sum_{v \in B(u)} \frac{PR(v)}{L(v)}$$
 - $B(u)$: the set containing all pages linking to page u
 - $PR(x)$: PageRank of page x
 - $L(v)$: the number of outbound links of page v

Contribution of page v to the rank of page u
- Initialize the PR of each page with an equi-probability
- Iterate k times: compute PR of each page

32

PageRank with Spark

PageRank principles

- The *damping* factor: the probability a user continues to click is a *damping* factor: d the probability a user *jumps* to a random page is: $1-d$

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

Sum of all PR is 1

Variant:

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

Sum of all PR is N_{pages}

Usually: $d = 0.85$

33

PageRank with Spark

PageRank first step in Spark (Scala)

```

// read text file into Dataset[String] -> RDD1
val lines = spark.read.textFile(args(0)).rdd

val pairs = lines.map{ s =>
  // Splits a line into an array of
  // 2 elements according space(s)
  val parts = s.split("\\s+")
  // create the parts<url, url>
  // for each line in the file
  (parts(0), parts(1))
}

// RDD1 <string, string> -> RDD2<string, iterable>
val links = pairs.distinct().groupByKey().cache()
    
```

34

PageRank with Spark

PageRank second step in Spark (Scala)

Initialization with $1/N$ equi-probability:

```

// links <key, Iter> RDD -> ranks <key, 1.0/N_{pages}> RDD
var ranks = links.mapValues(v => 1.0/4.0)
    
```

`links.mapValues(...)` is an immutable RDD
`var ranks` is a mutable variable

```

var ranks = RDD1
ranks = RDD2
    
```

« ranks » is re-associated to a new RDD
 RDD1 is forgotten ...
 ...and will be removed from memory

Other strategy:

```

// links <key, Iter> RDD -> ranks <key, one> RDD
var ranks = links.mapValues(v => 1, 0)
    
```

35

PageRank with Spark

PageRank third step in Spark (Scala)

```

for (i <- 1 to iters) {
  val contribs =
    links.join(ranks)
    .flatMap{ case (url, (urlLinks, rank)) =>
      urlLinks.map(dest => (dest, rank/urlLinks.size))
    }
    ranks = contribs.reduceByKey(_ + _)
    .mapValues(0.15 + 0.85 * _)
}
    
```

36

PageRank with Spark

PageRank third step in Spark (Scala)

- Spark & Scala allow a **short/compact implementation** of the PageRank algorithm
- Each RDD remains **in-memory** from one iteration to the next one

```
val lines = spark.read.textFile(args(0)).rdd
val pairs = lines.map{ s =>
  val parts = s.split("\\s+")
  (parts(0), parts(1)) }
val links = pairs.distinct().groupByKey().cache()

var ranks = links.mapValues(v => 1.0)

for (i <- 1 to iters) {
  val contribs =
    links.join(ranks)
      .flatMap{ case (url (urlLinks, rank)) =>
        urlLinks.map(dest => (dest,rank/urlLinks.size))}
    ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
```

37

PageRank with Spark

PageRank third step in Spark (Scala): optimized with partitioner

```
val links = ..... // previous code
val links1 = links.partitionBy(new HashPartitioner(100)).persist()

var ranks = links1.mapValues(v => 1.0)

for (i <- 1 to iters) {
  val contribs =
    links1.join(ranks)
      .flatMap{ case (url (urlLinks, rank)) =>
        urlLinks.map(dest => (dest,rank/urlLinks.size))}
    ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
```

- Initial links and ranks are co-partitioned
- Repeated join is Narrow-Wide
- Repeated mapValues is Narrow: respects the reduceByKey partitioning
- Pb: flatMap{...urlinks.map(...)} can change the partitioning ?!

38

Spark optimizations



39