Resilient Distributed Datasets

Spark Core

Lecture 26

Quiz 6-7

Thursday 2nd June, 2022
Lecture 23-27

Spark Core

Spark Core is the base engine for large-scale parallel and distributed data processing

It is responsible for:

- memory management
- fault recovery
- scheduling, distributing and monitoring jobs on a cluster
- interacting with storage systems

Spark Core

simplilearn

Spark Core

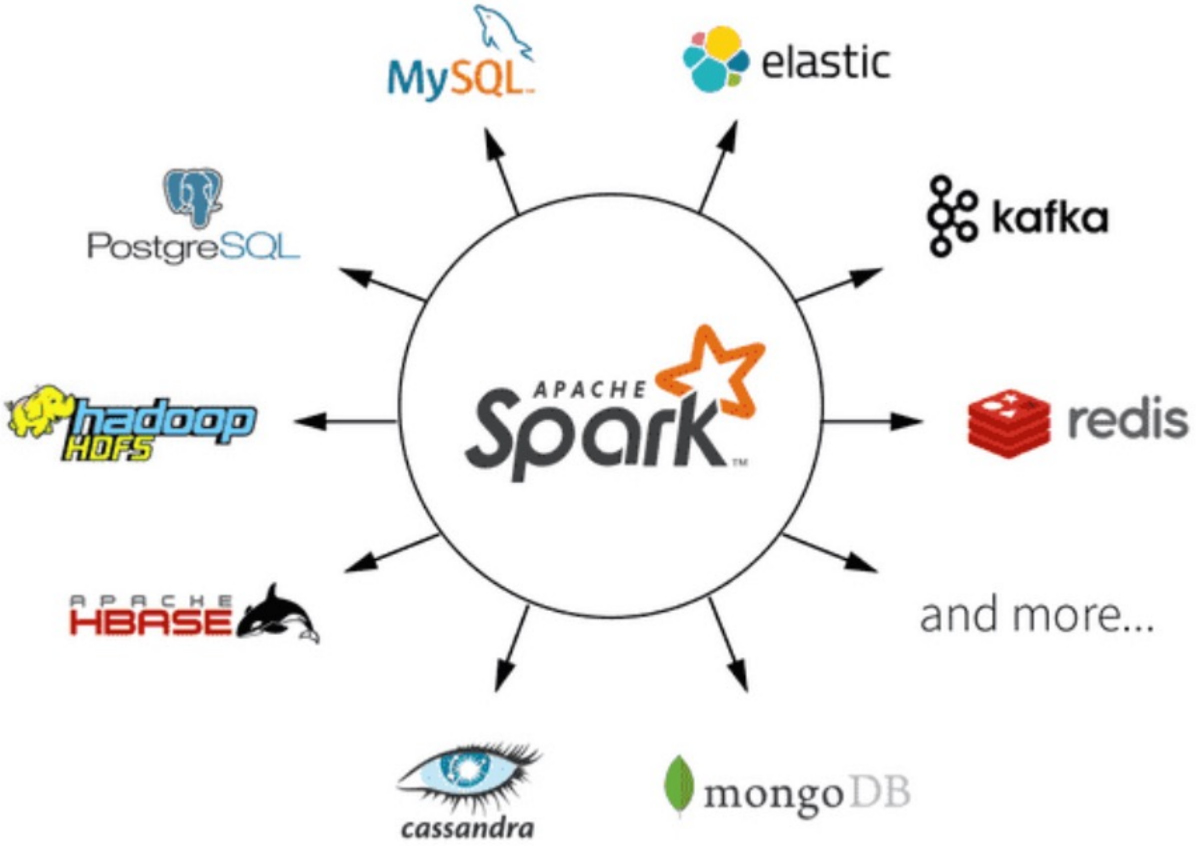
Spark Core is the base engine for large-scale parallel and distributed data processing

Spark doesnot have **its own storage**. It relies on other storage that storage could be your **HDFS, NoSql, RDBMS etc.** to which you could connect your spark to fetch and analyze the data

It is responsible for:

- memory management
- fault recovery
- scheduling, distributing and monitoring jobs on a cluster
- interacting with storage systems

Spark Core


APACHE Spark
MySQL
elastic
kafka
redis
and more...
mongoDB
cassandra
APACHE HBASE
hadoop HDFS
PostgreSQL

Resilient Distributed Dataset

Dataset

Data storage created from: HDFS, S3, HBase, JSON, text, Local hierarchy of folders

Or created transforming another RDD

Resilient Distributed Dataset

Distributed

Distributed across the cluster of machines

Divided in partitions, atomic chunks of data

Resilient Distributed Dataset

Resilient

Recover from errors, e.g. node failure, slow processes

Track history of each partition, re-run

Resilient Distributed Dataset

Spark Core is embedded with RDDs (Resilient Distributed Datasets), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel

Spark Core

RDD

Transformation

These are operations (such as map, filter, join, union) that are performed on an RDD that yields a new RDD containing the result

Action

These are operations (such as reduce, first, count) that return a value after running a computation on an RDD

Resilient Distributed Dataset

HDFS

Spark Core is embedded with RDDs (Resilient Distributed Datasets), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel

RDD

RDD

RDD

Spark Core

RDD

Transformation

These are operations (such as map, filter, join, union) that are performed on an RDD that yields a new RDD containing the result

Action

These are operations (such as reduce, first, count) that return a value after running a computation on an RDD

val x = sc.textfile

val y = x.map

val z = y.filter

z.count

z.count

Directed Acyclic Graphs

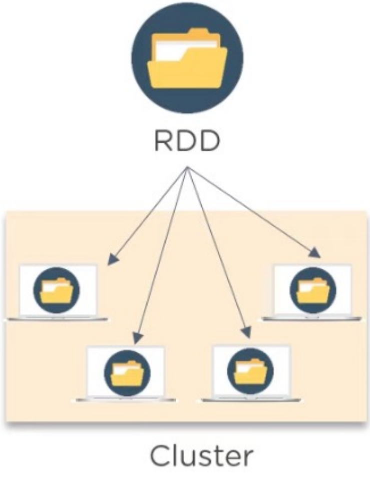
Track **dependencies**!
(also known as lineage or provenance)

14 What is the significance of Resilient Distributed Datasets in Spark?

Spark Core is embedded with Resilient Distributed Datasets, which is a fundamental data structure of Apache Spark

RDDs are immutable, fault-tolerant, distributed collection of objects that can be operated on in parallel

RDD's are split into partitions and can be executed on different nodes of a cluster



5 What is lazy evaluation in Spark?

When Spark operates on any dataset, it remembers the instructions, so that it does not forget

When a transformation such as map() is called on an RDD, the operation is not performed instantly

Transformations in Spark are not evaluated until you perform an action, which aids in optimizing the overall data processing workflow. This is called lazy evaluation

simplilearn

18

What is a lineage graph?

Directed Acyclic Graphs

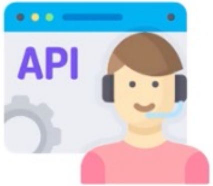
Track **dependencies**!
(also known as lineage or provenance)

Spark does not support data replication in the memory. So, if any data is lost, it can be rebuilt using RDD lineage

It is also called an RDD operator graph or RDD dependency graph

What is PySpark?

PySpark



PySpark is the Python API to support Apache Spark

python + APACHE Spark = PySpark

RDD in PySpark

From the PySpark console:

```
integer_RDD = sc.parallelize(range(10), 3)
```

https://spark.apache.org/docs/3.1.1/api/python/reference/pyspark.html

Check partitions

Gather all data on the driver:

```
integer_RDD.collect()
```

```
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Check partitions

Maintain splitting in partitions:

```
integer_RDD.glom().collect()
```

```
Out: [[0, 1, 2], [3, 4, 5], [6, 7, 8, 9]]
```

Read text into Spark

from local filesystem:

```
text_RDD =
sc.textFile("file:///home/cloudera/testfile1")
```

from HDFS:

```
text_RDD =
sc.textFile("/user/cloudera/input/testfile1")
text_RDD.take(1) #outputs the first line
```

Wordcount in Spark: map

```
def split_words(line):
    return line.split()

def create_pair(word):
    return (word, 1)

pairs_RDD=text_RDD.flatMap(split_words).map(create_pair)
```

```
pairs_RDD.collect()
Out[]: [(u'A', 1),
 (u'long', 1),
 (u'time', 1),
 (u'ago', 1),
 (u'in', 1),
 (u'a', 1),
 (u'galaxy', 1),
 (u'far', 1),
 (u'far', 1),
 (u'away', 1)]
```

Wordcount in Spark: reduce

```
def sum_counts(a, b):
    return a + b

wordcounts_RDD = pairs_RDD.reduceByKey(sum_counts)
```

```
Out[]:
[(u'A', 1),
 (u'ago', 1),
 (u'far', 2),
 (u'away', 1),
 (u'in', 1),
 (u'long', 1),
 (u'a', 1),
 (u'time', 1),
 (u'galaxy', 1)]
```

27 How would you compute the total count of the unique words in Spark?

1 Load the text file as RDD:

```
sc.textFile("hdfs://Hadoop/user/test_file.txt");
```

2 Function that breaks each line into words:

```
def toWords(line):
return line.split();
```

3 Run the toWords function on each element of RDD on Spark as flatMap transformation:

```
words = line.flatMap(toWords);
```

4 Convert each word into (key,value) pair:

```
def toTuple(word):
return (word, 1);
wordTuple = words.map(toTuple);
```

5 Perform reduceByKey() action:

```
def sum(x, y):
return x+y:
counts = wordsTuple.reduceByKey(sum)
```

6 Print:

```
counts.collect()
```

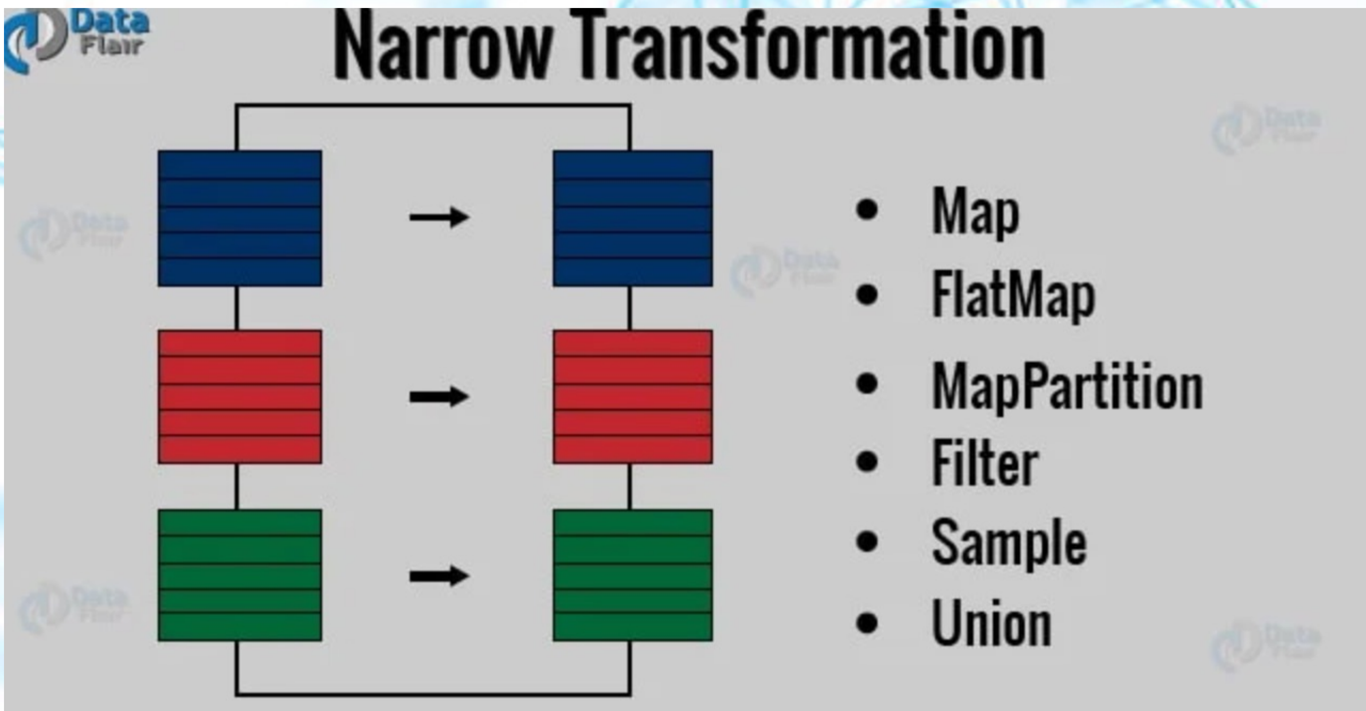
Transformations

Transformations

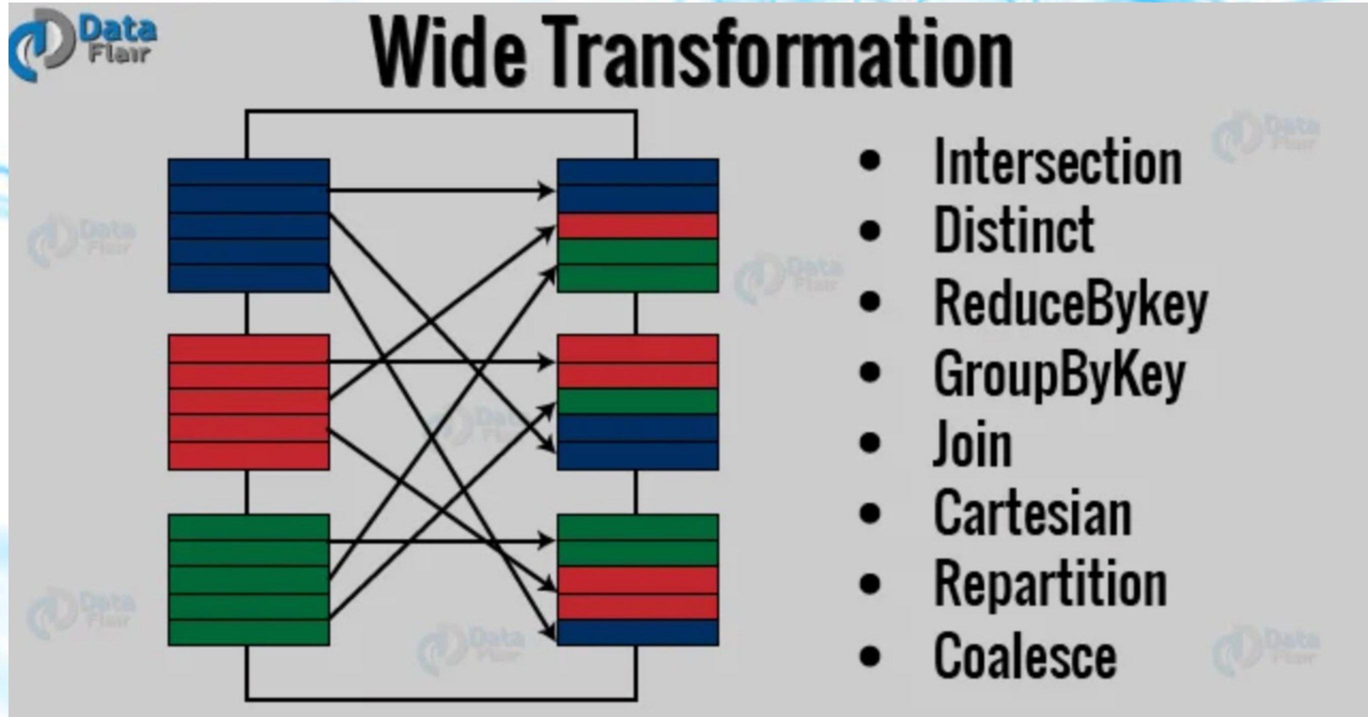
- RDD are immutable
- Never modify RDD in place
- Transform RDD to another RDD
- Lazy

https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

In ***Narrow transformation***, all the elements that are required to compute the records in single partition live in the **single partition of parent RDD**. A limited subset of partition is used to calculate the result. *Narrow transformations* are the result of *map(), filter()*.



In **wide transformation**, all the elements that are required to compute the records in the **single partition may live in many partitions of parent RDD**. The partition may live in many partitions of parent RDD. *Wide transformations* are the result of *groupbyKey()* and *reducebyKey()*.



Create RDD

from local filesystem:

text_RDD =

sc.textFile("file:///home/cloudera/testfile1")

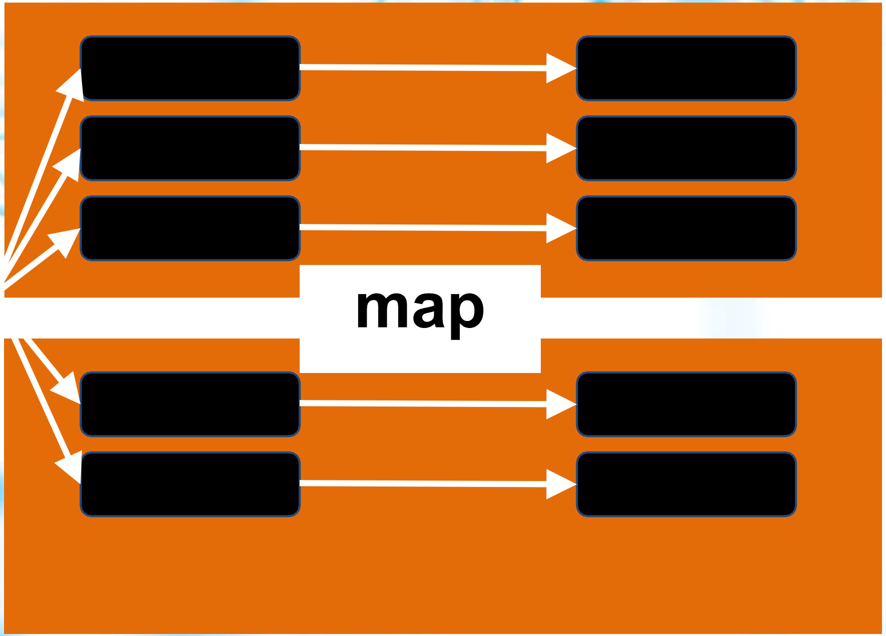
Apply a transformation: map

map : apply function to each element of RDD

```
def lower(line):
    return line.lower()
lower_text_RDD = text_RDD.map(lower)
```

map

map : apply function to each element of RDD



Other transformations

flatMap(func) - map then flatten output

filter(func) - keep only elements where func is true

sample(withReplacement, fraction, seed) - get a random data fraction

coalesce(numPartitions) - merge partitions to reduce them to numPartitions

11 What is the use of coalesce in Spark?

Spark uses coalesce method to reduce the number of partitions in a DataFrame

Filter operation to remove all multiples of 10

Final result

Partition A: 11, 12	Partition A: 11, 12	Partition A: 11, 12
Partition B: 30, 40, 50	Partition B: -	Partition C: 6, 7, 9
Partition C: 6, 7	Partition C: 6, 7	
Partition D: 9, 10	Partition D: 9	

Data read from a CSV file into an RDD having four partitions

The RDD has some empty partitions. It makes sense to reduce the number of partitions. Using coalesce we can achieve the same

Resultant RDD when coalesce(2) has been applied will look like:

simplilearn

Other transformations

flatMap(func) - map then flatten output
filter(func) - keep only elements where func is true
sample(withReplacement, fraction, seed) - get a random data fraction
coalesce(numPartitions) - merge partitions to reduce them to numPartitions

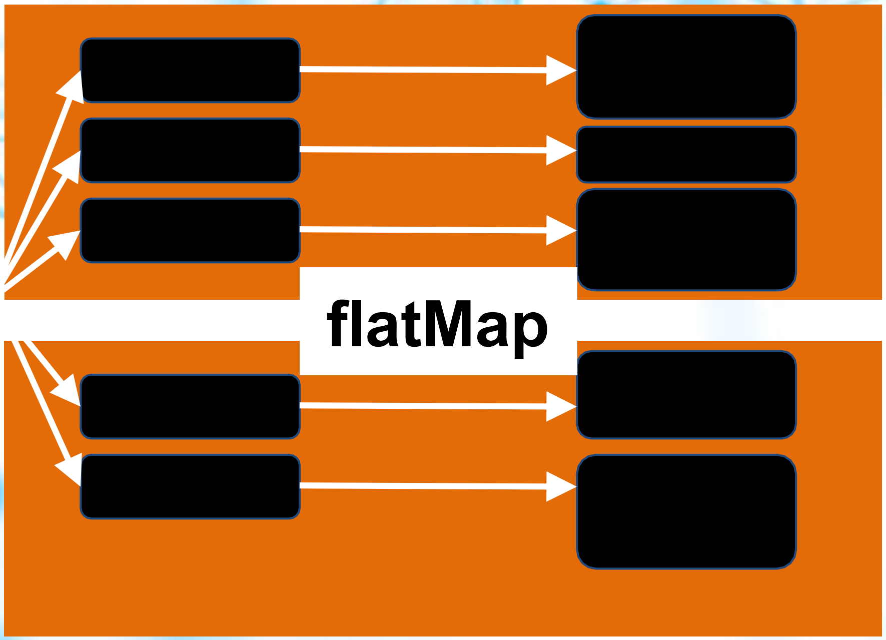
```python
def split_words(line):
    return line.split()

words_RDD =
text_RDD.flatMap(split_words)
words_RDD.collect()
```

```
Out[]:
[u'A',
 u'long',
 u'time',
 u'ago',
 u'in',
 u'a',
 u'galaxy',
 u'far',
 u'far',
 u'away']
```

flatMap

flatMap : map then flatten output

RDD Partitions



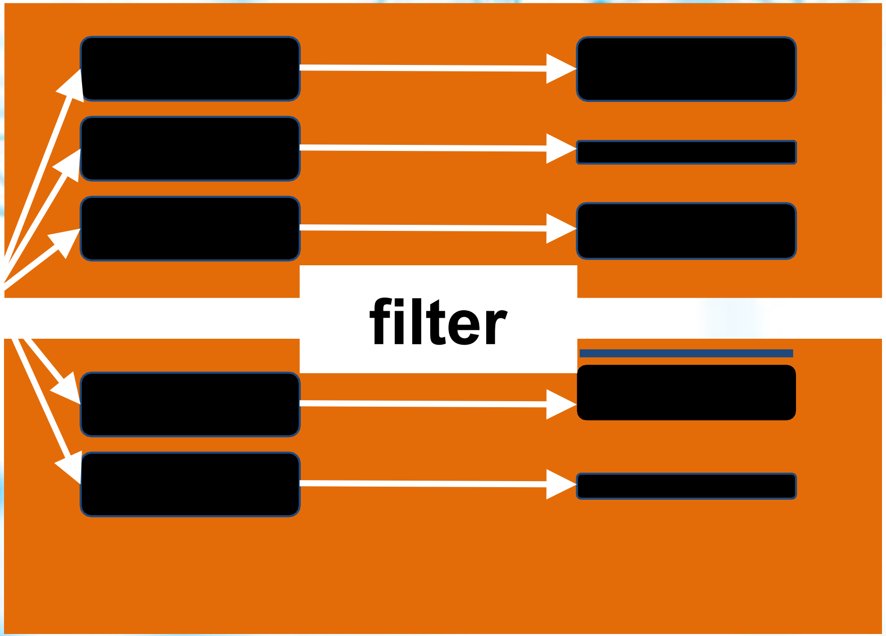
filter

```
def starts_with_a(word):
    return word.lower().startswith("a")

words_RDD.filter(starts_with_a).collect()

Out[]: [u'A', u'ago', u'a', u'away']
```

filter

`filter` : keep only elements where func is True

RDD Partitions

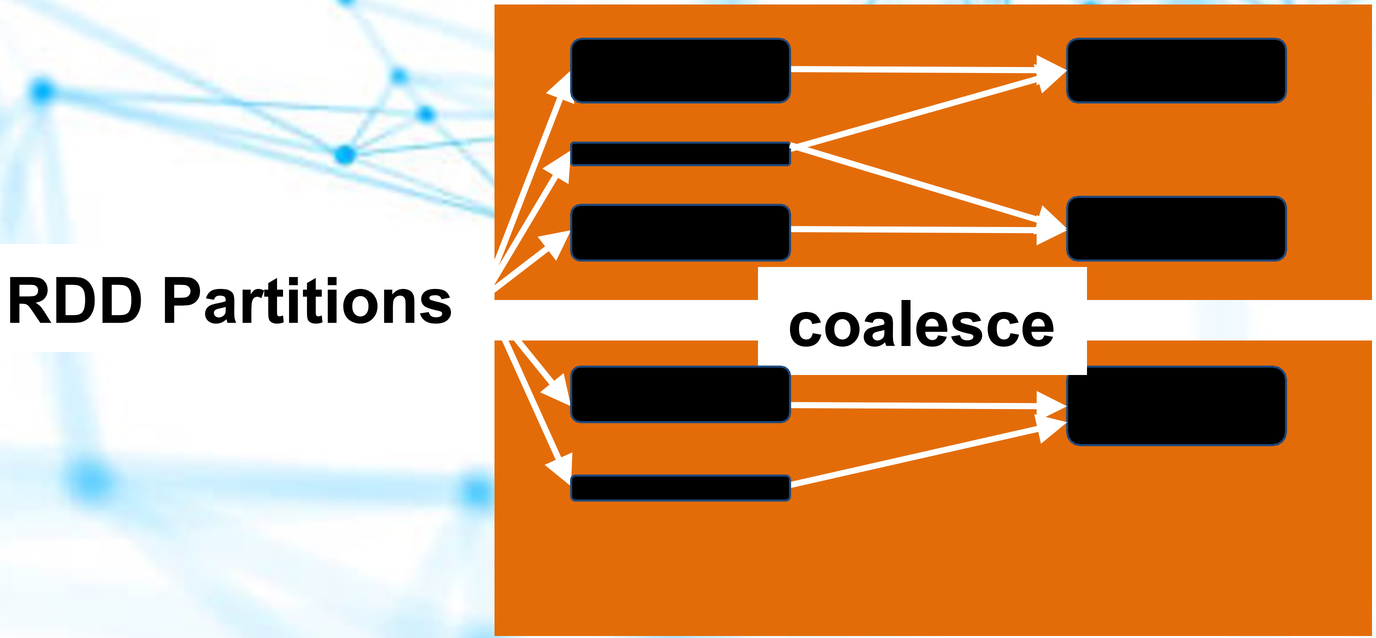


coalesce

```
sc.parallelize(range(10), 4).glom().collect()
Out[]: [[0, 1], [2, 3], [4, 5], [6, 7, 8, 9]]

sc.parallelize(range(10), 4).coalesce(2).glom().collect()
Out[]: [[0, 1, 2, 3], [4, 5, 6, 7, 8, 9]]
```

coalesce

coalesce : reduce the number of partitions



11

What is the use of coalesce in Spark?

Spark uses coalesce method to **reduce the number of partitions** in a DataFrame

Filter operation to remove all multiples of 10

Final result

Partition A: 11, 12

Partition B: 30, 40, 50

Partition C: 6, 7

Partition D: 9, 10

Data read from a CSV file into an RDD having four partitions

Partition A: 11, 12

Partition B: -

Partition C: 6, 7

Partition D: 9

The RDD has some empty partitions. It makes sense to reduce the number of partitions. Using coalesce we can achieve the same

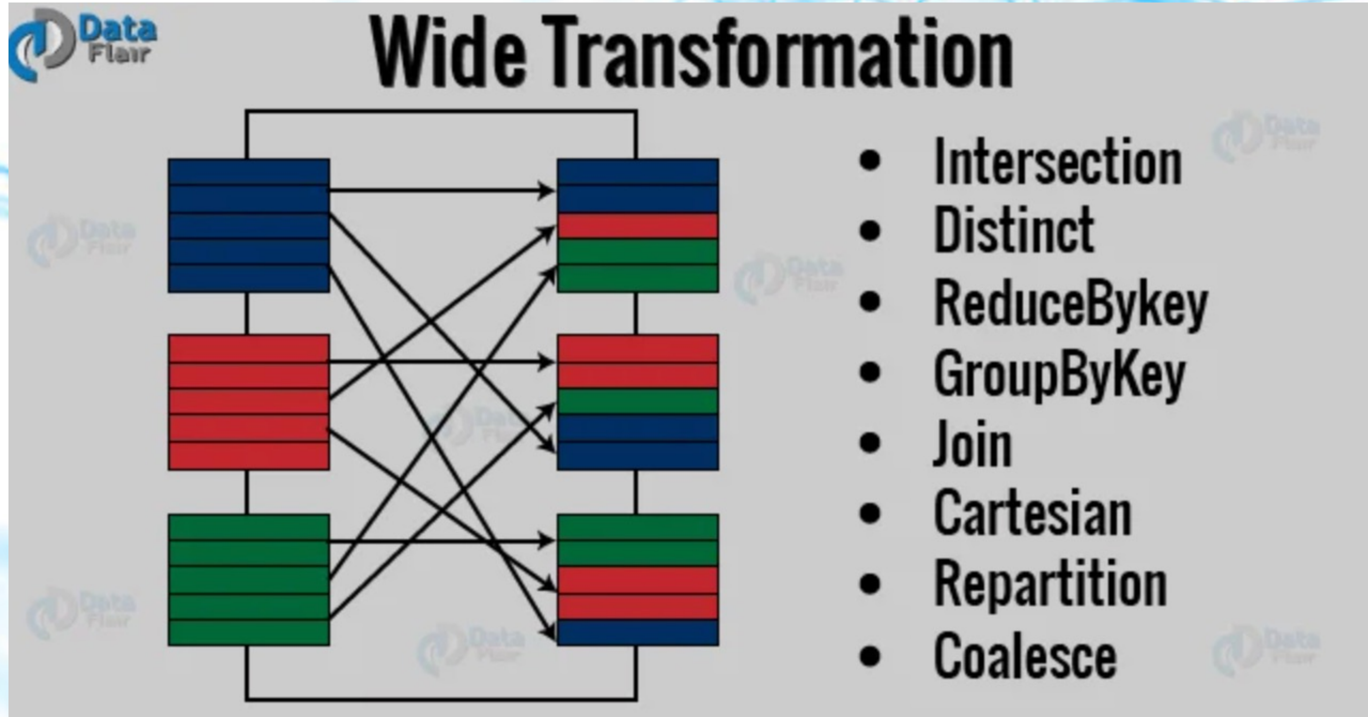
Partition A: 11, 12

Partition C: 6, 7, 9

Resultant RDD when coalesce(2) has been applied will look like:

Wide Transformations

In **wide transformation**, all the elements that are required to compute the records in the **single partition may live in many partitions of parent RDD**. The partition may live in many partitions of parent RDD. *Wide transformations* are the result of *groupbyKey()* and *reducebyKey()*.



Transformations of (K,V) pairs

```
def create_pair(word):
    return (word, 1)
```

```
pairs_RDD=text_RDD.flatMap(split_words).map(create_pair)
```

```
pairs_RDD.collect()
```

```
Out[]: [(u'A', 1),
(u'long', 1),
(u'time', 1),
(u'ago', 1),
(u'in', 1),
(u'a', 1),
(u'galaxy', 1),
(u'far', 1),
(u'far', 1),
(u'away', 1)]
```

groupByKey

[redacted]: (K, V) pairs => (K, iterable of all V)

(A, 1)

(B, 8)

(A, [1, 2, 5])

(B, [8])

(A, 2)

(A, 5)

```
pairs_RDD.groupByKey().collect()
```

```
Out[]: [(u'A', <pyspark.resultiterable.ResultIterable at XXX>),
(u'ago', <pyspark.resultiterable.ResultIterable at XXX>),
(u'far', <pyspark.resultiterable.ResultIterable at XXX>),
(u'away', <pyspark.resultiterable.ResultIterable at XXX>),
(u'in', <pyspark.resultiterable.ResultIterable at XXX>),
(u'long', <pyspark.resultiterable.ResultIterable at XXX>),
(u'a', <pyspark.resultiterable.ResultIterable at XXX>),<
```

<MORE output>

```
for k,v in pairs_RDD.groupByKey().collect():
    print "Key:", k, ",Values:", list(v)
```

Out[]: Key: A , Values: [1]

Key: ago , Values: [1]

Key: far , Values: [1, 1]

Key: away , Values: [1]

Key: in , Values: [1]

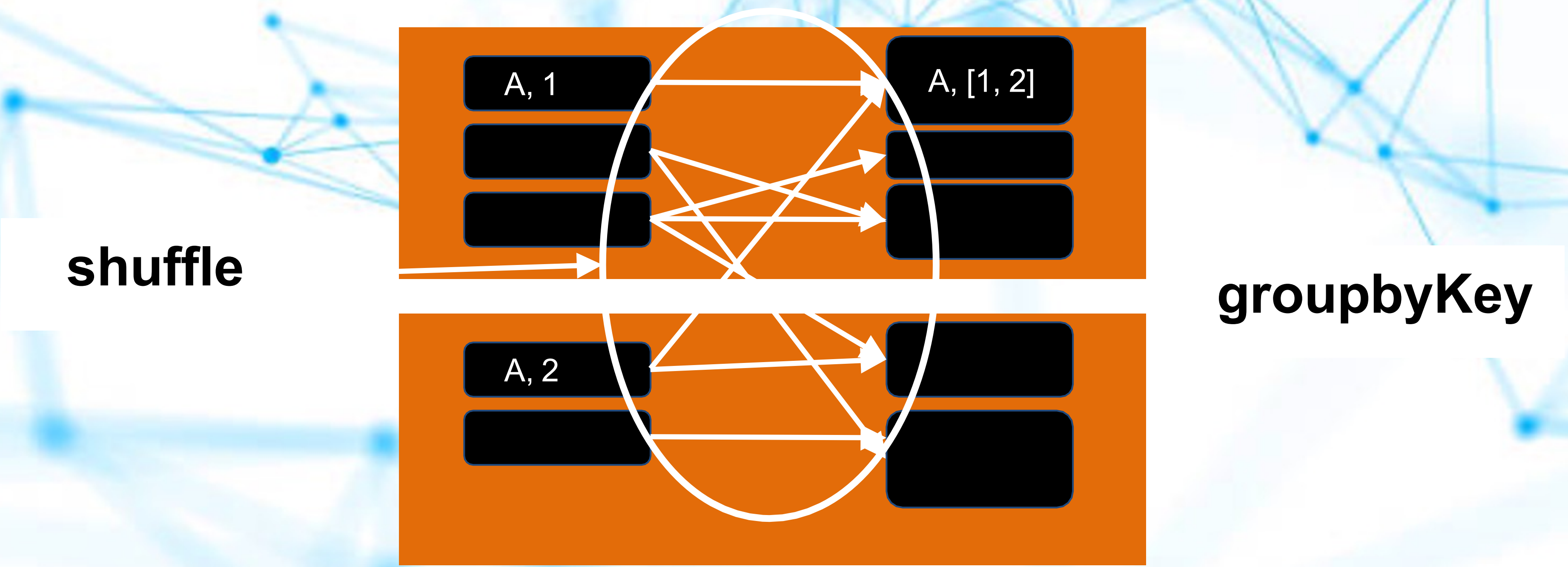
Key: long , Values: [1]

Key: a , Values: [1]

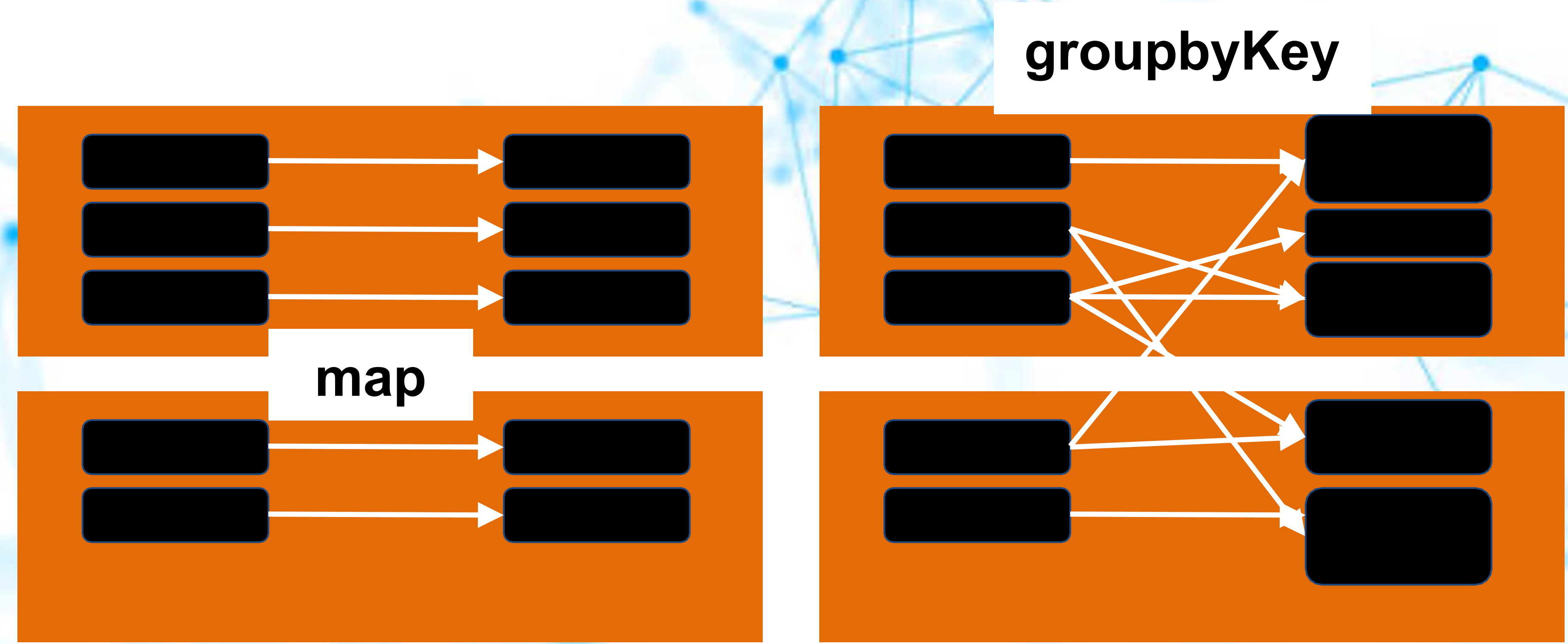
<MORE output>

groupByKey

groupByKey : (K, V) pairs => (K, iterable of all V)



Narrow vs Wide



Wide transformations

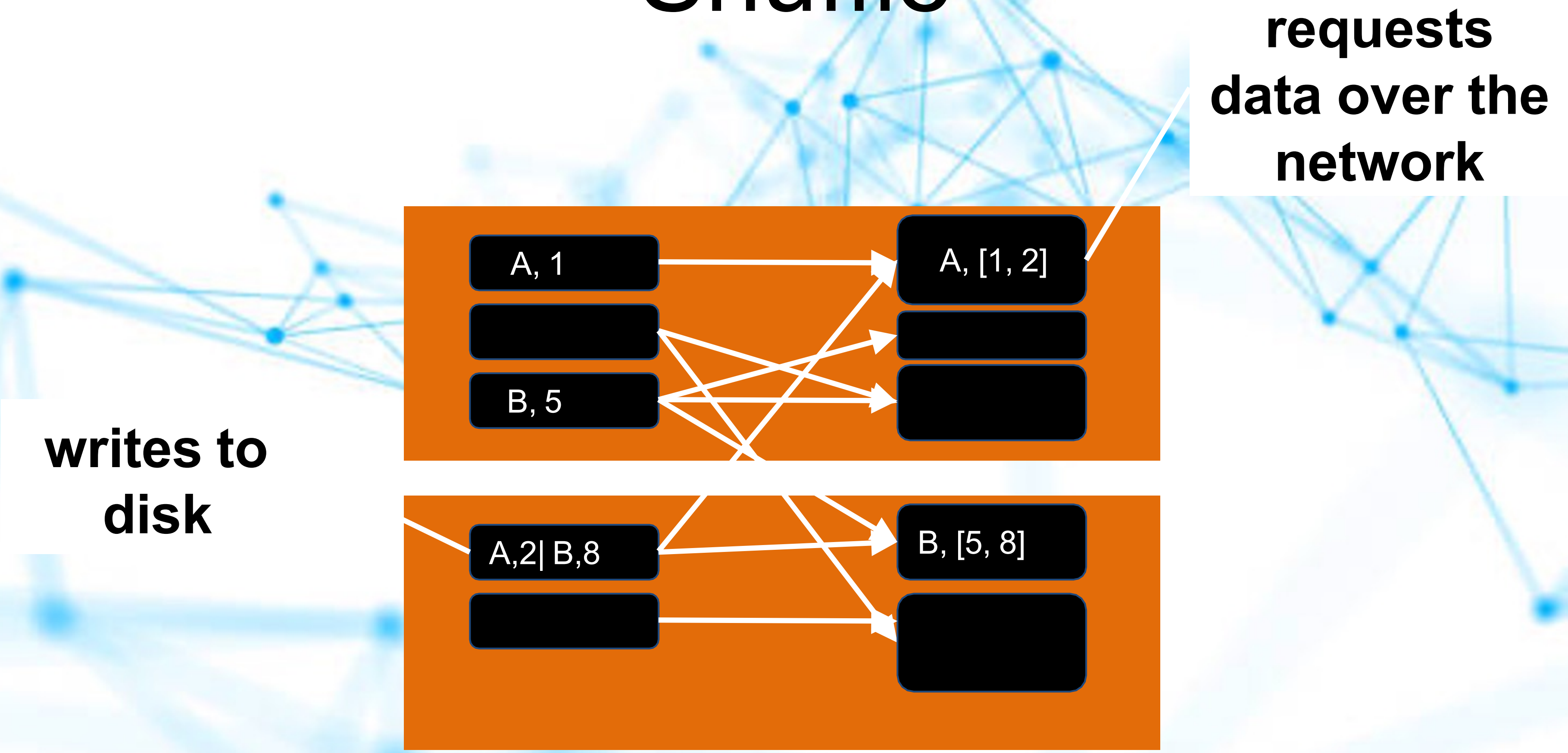
- groupByKey : (K, V) pairs => (K, iterable of all V)
- reduceByKey(func) : (K, V) pairs => (K, result of reduction by func on all V)
- Repartition(numPartitions) : similar to coalesce, shuffles all data to increase or decrease number of partitions to numPartitions

Shuffle

Shuffle

- Global redistribution of data
- High impact on performance

Shuffle


requests data over the network
writes to disk
A, 1
B, 5
A,2| B,8
A, [1, 2]
B, [5, 8]

Know shuffle, avoid it

- Which operations cause it?
- Is it necessary?

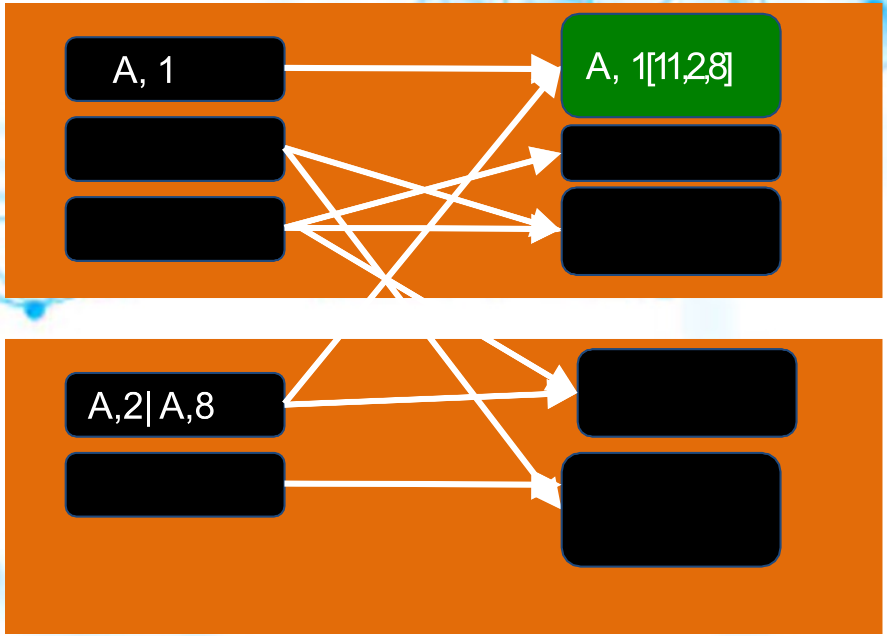
Really need groupByKey?

groupByKey: (K, V) pairs => (K, iterable of all V)

if you plan to call reduce later in the pipeline,

reduceByKey

groupByKey + reduce



reduceByKey