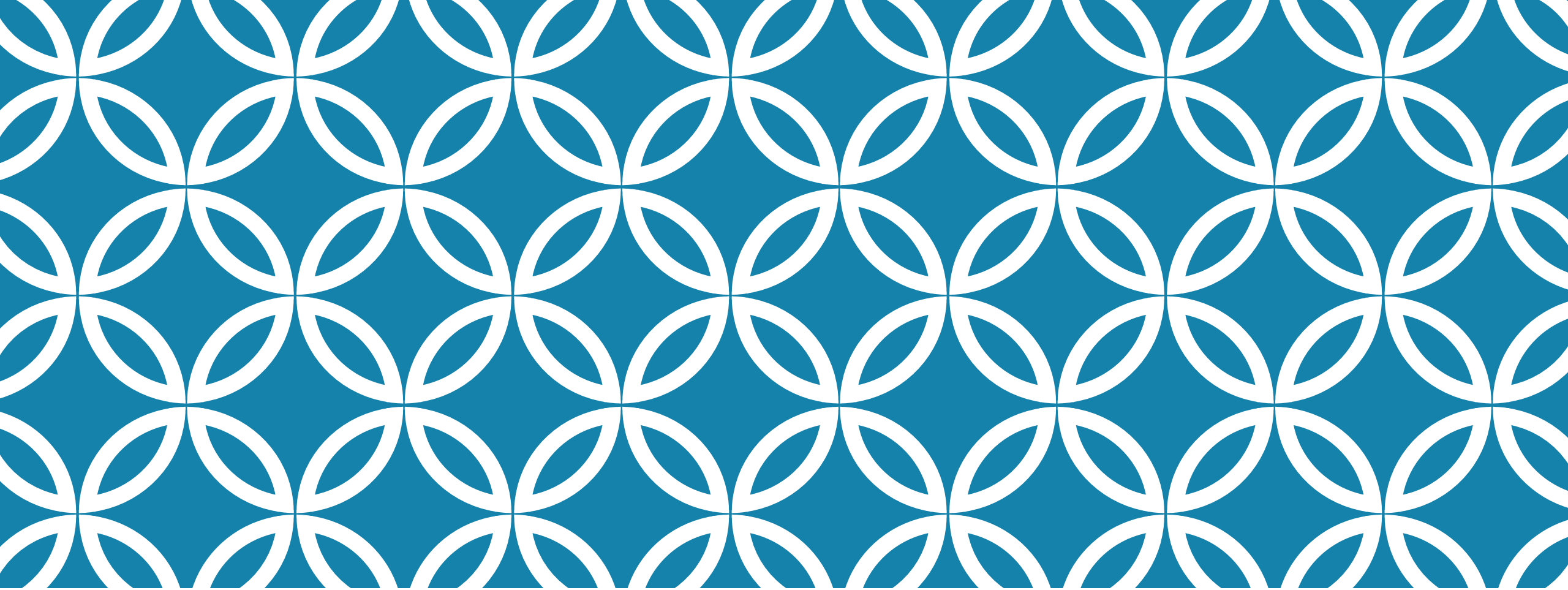
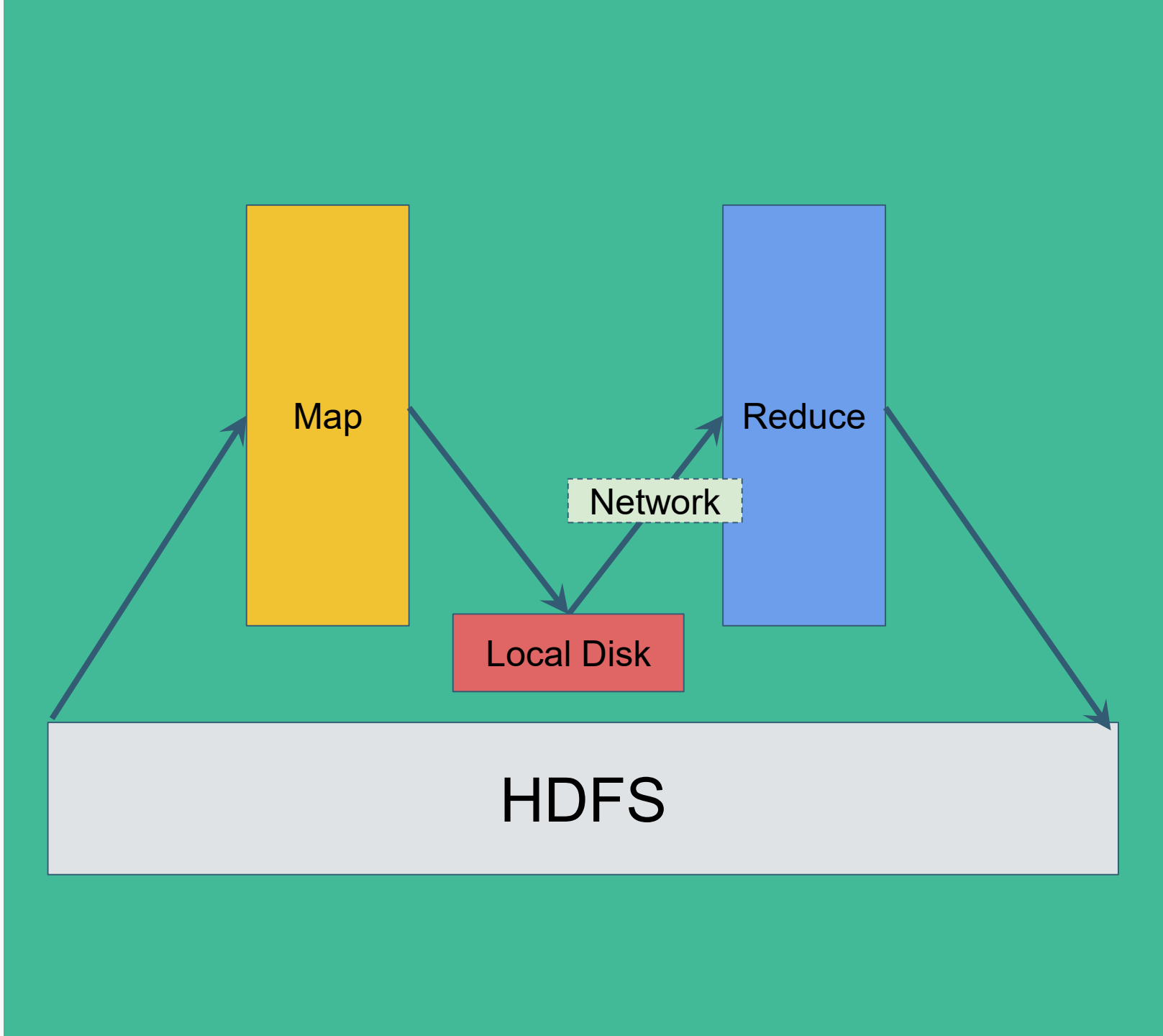


SPARK

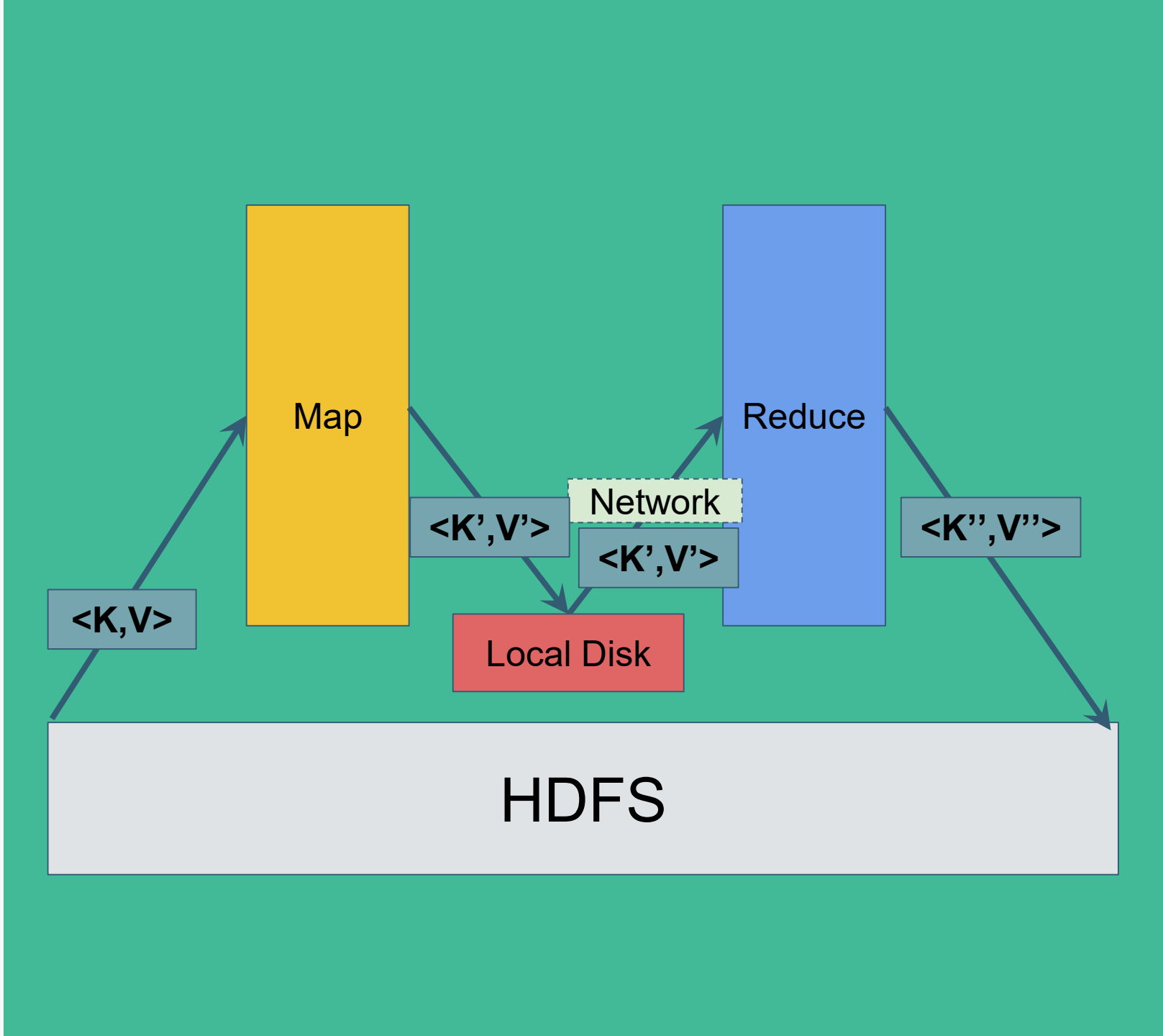
Map-Reduce Limits

Disk-based computation
Key-Value data
Limited to Map & Reduce
Not designed for iterative jobs



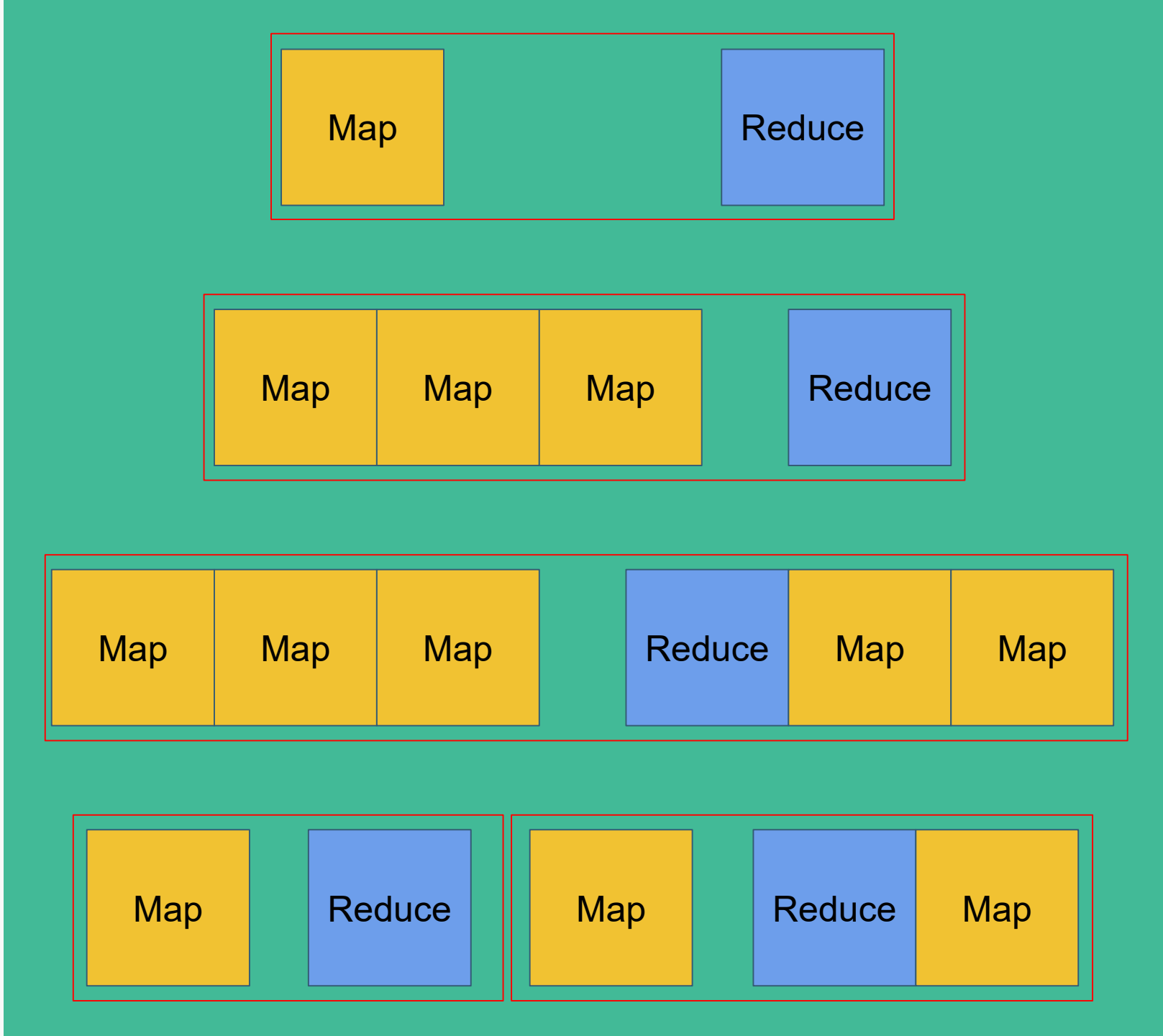
Map-Reduce Limits

Disk-based computation
Key-Value data
Limited to Map & Reduce
Not designed for iterative jobs



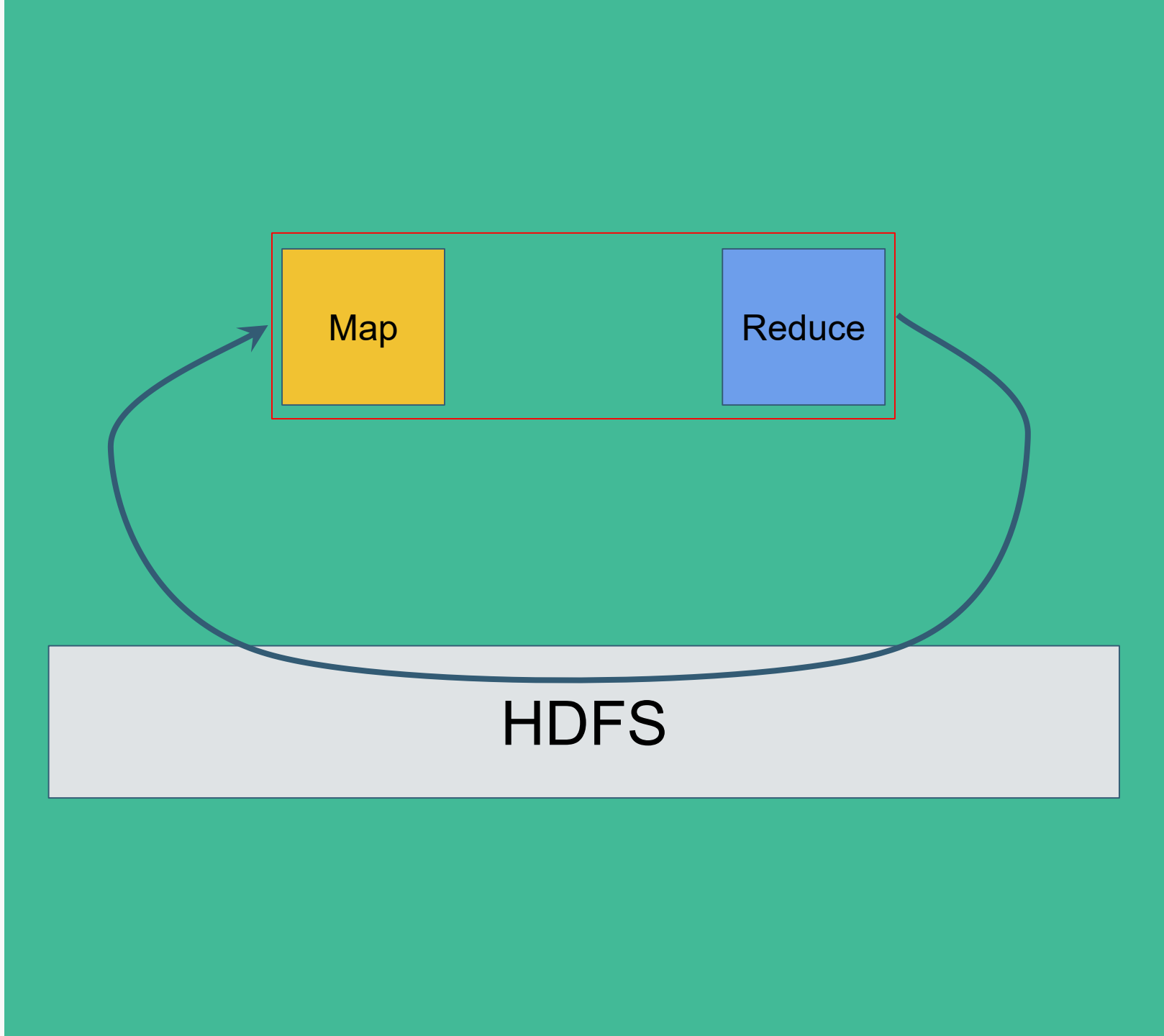
Map-Reduce Limits

Disk-based computation
Key-Value data
Limited to Map & Reduce
Not designed for iterative jobs



Map-Reduce Limits

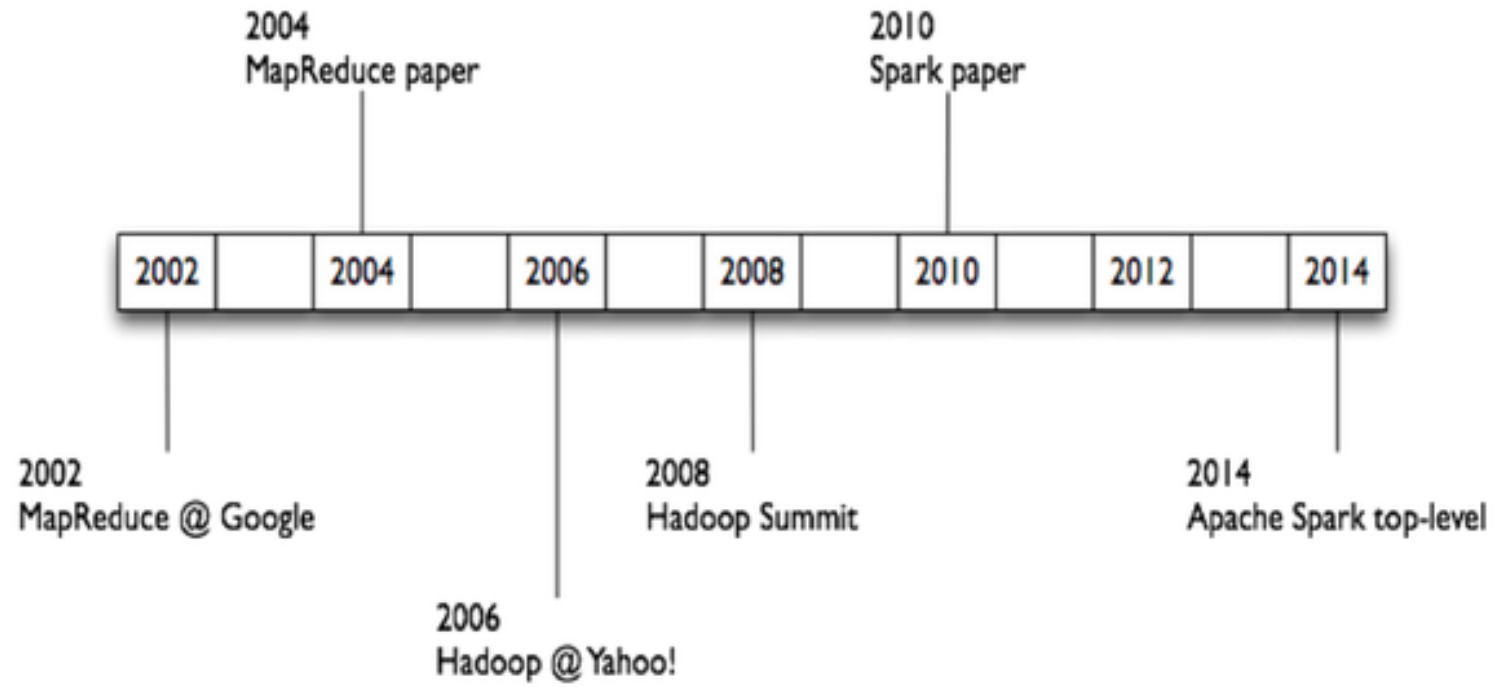
Disk-based computation
Key-Value data
Limited to Map & Reduce
Not designed for iterative jobs



What is Spark ?

In-memory computing
Data-centric through Resilient Distributed Datasets (RDDs)
More operations
More flexibility

History Review



What is Spark ?

In-memory computing
Data-centric through Resilient Distributed Datasets (RDDs)
More operations
More flexibility

Languages:

- Java
- Scala
- Python

Libs:

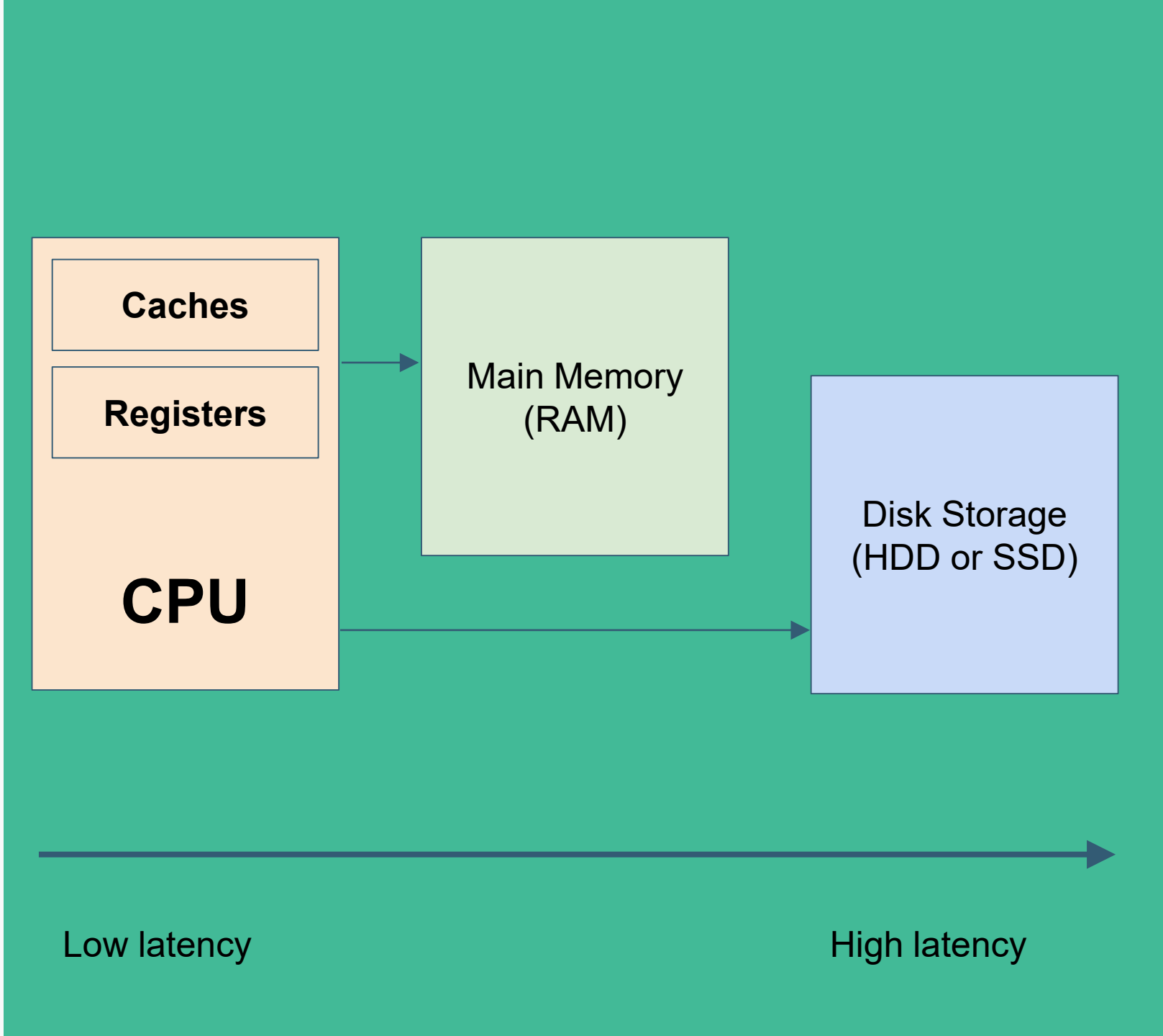
- MLlib
- GraphX
- Spark Streaming
- Spark SQL

Deployment:

- Spark standalone
- YARN
- Mesos

What is Spark ?

In-memory computing
Data-centric through Resilient Distributed Datasets (RDDs)
More operations
More flexibility



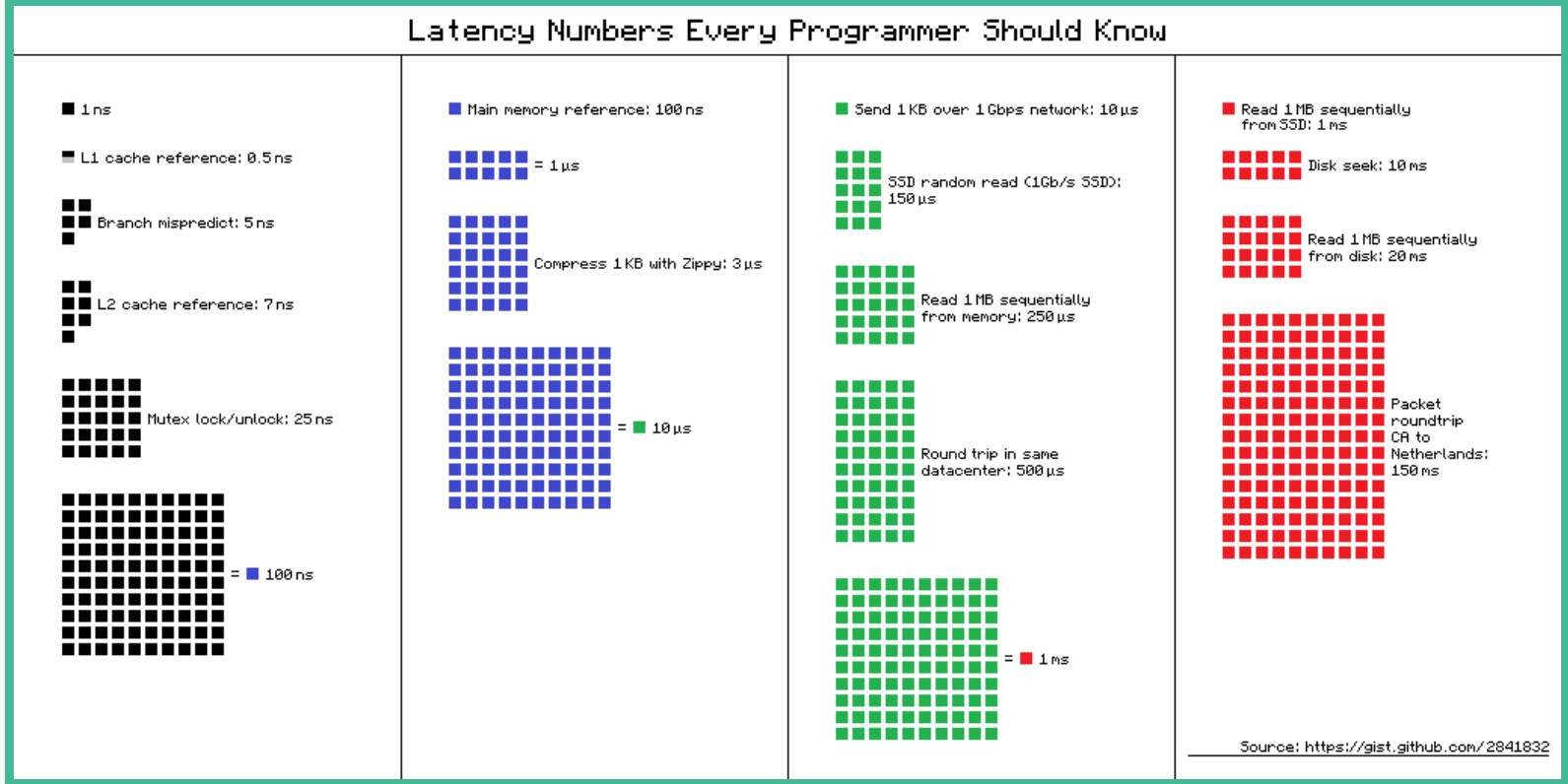
What is Spark ?

In-memory computing

Data-centric through Resilient Distributed Datasets (RDDs)

More operations

More flexibility



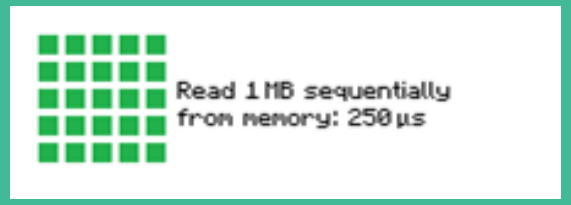
What is Spark ?

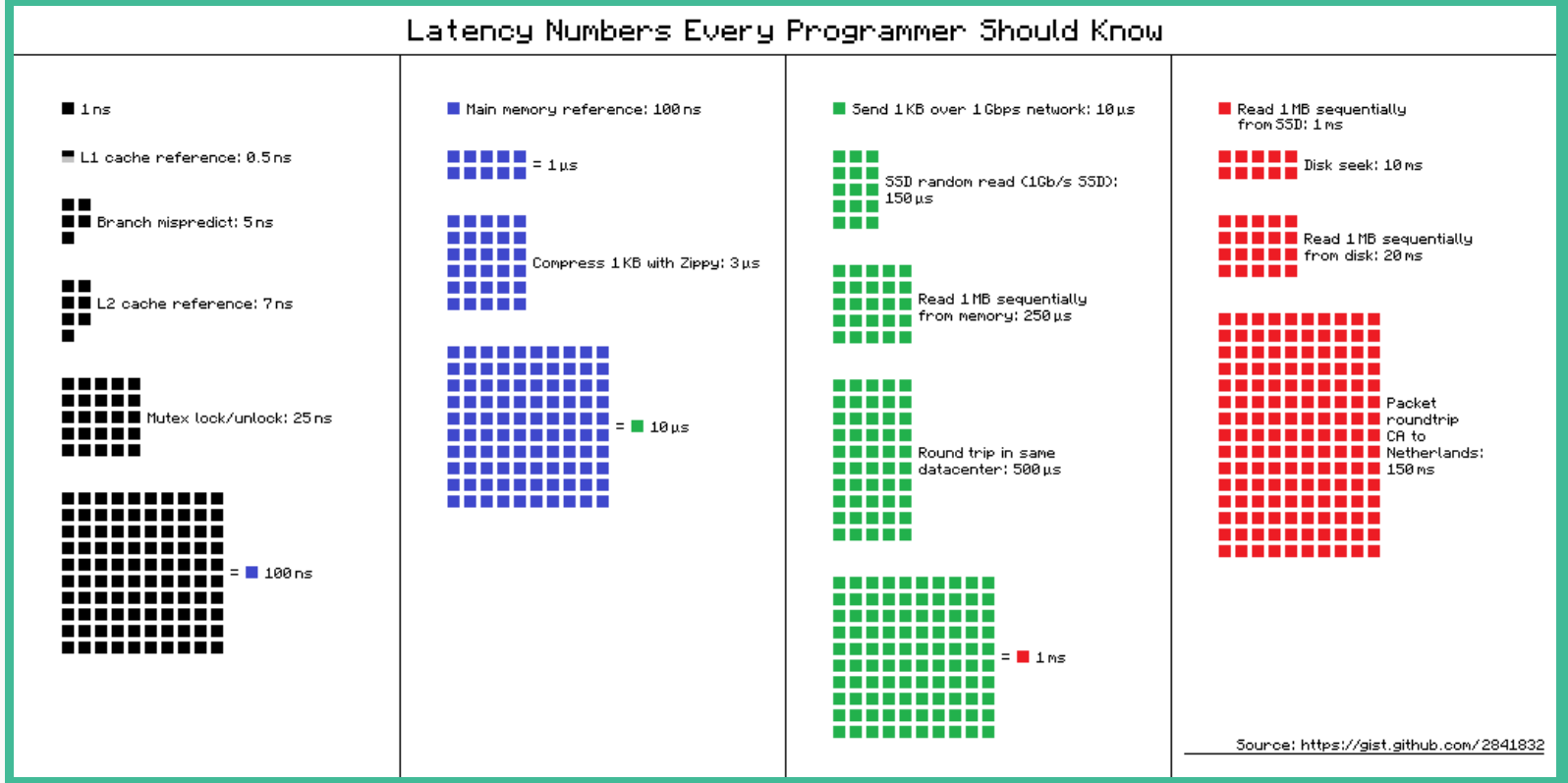
In-memory computing

Data-centric through Resilient Distributed Datasets (RDDs)

More operations

More flexibility





What is Spark ?

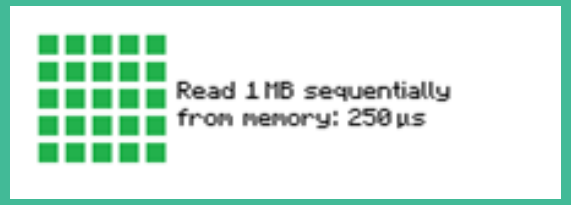
In-memory computing

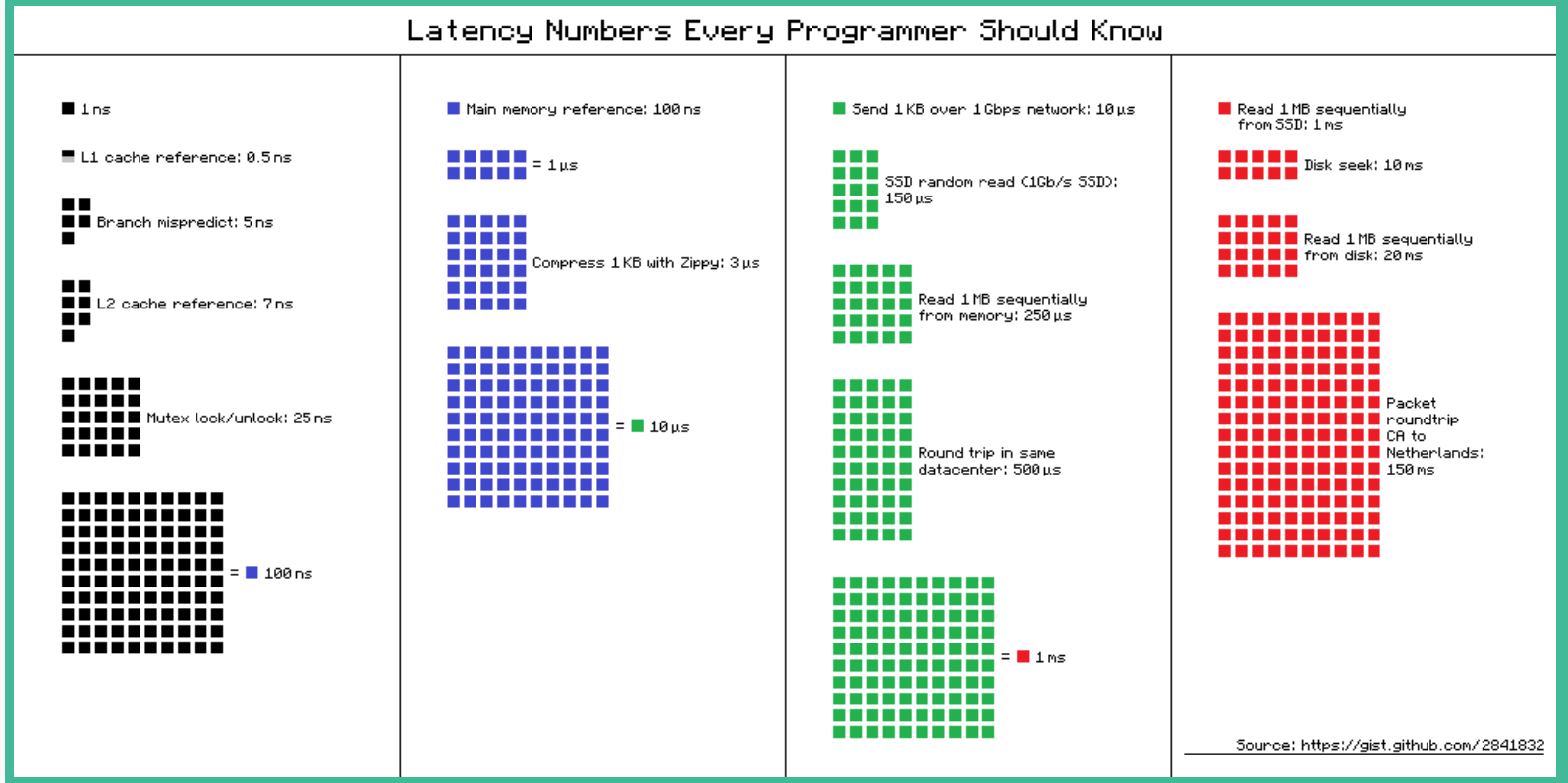
Data-centric through Resilient Distributed Datasets (RDDs)

More operations

More flexibility

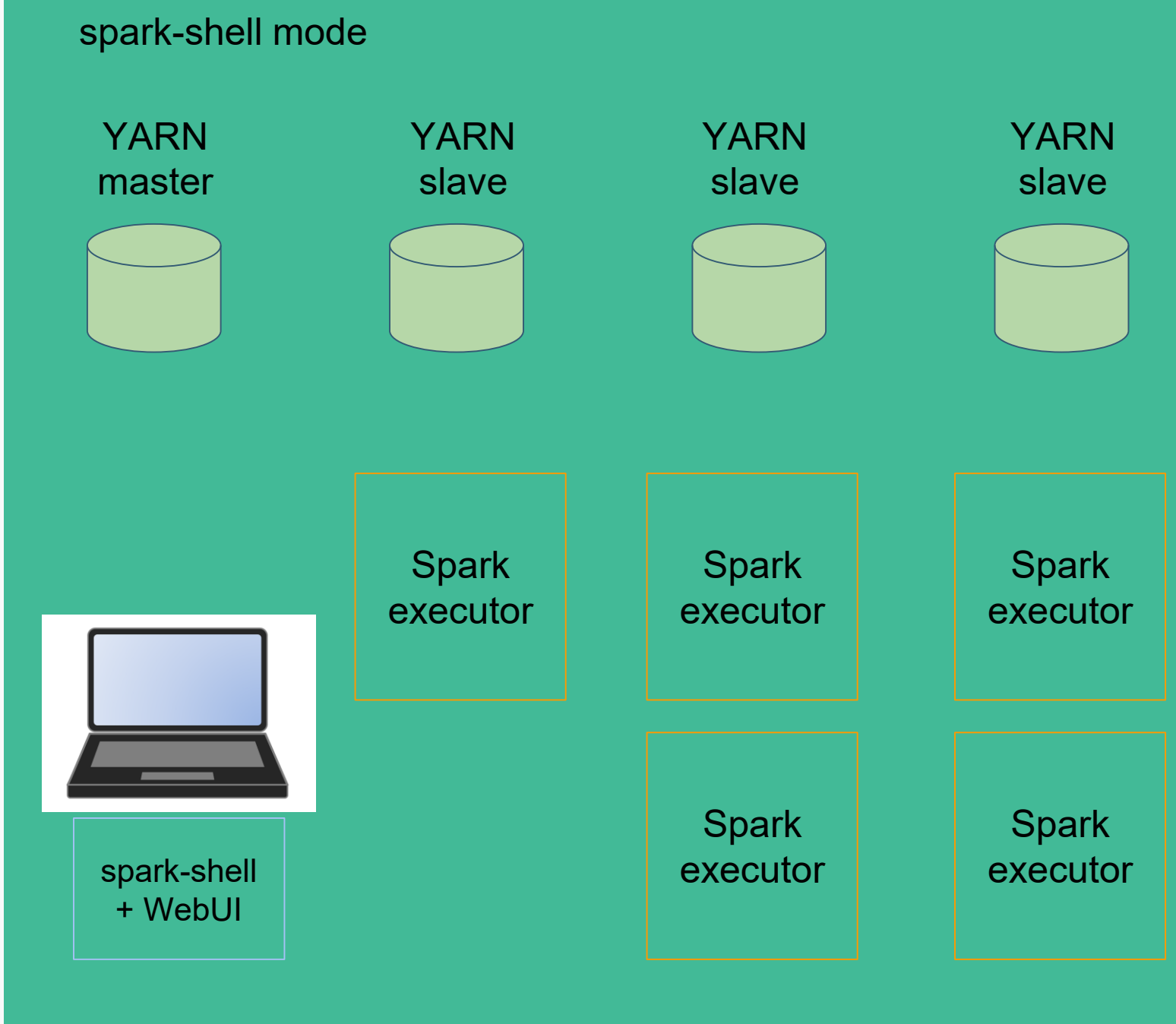
X 100





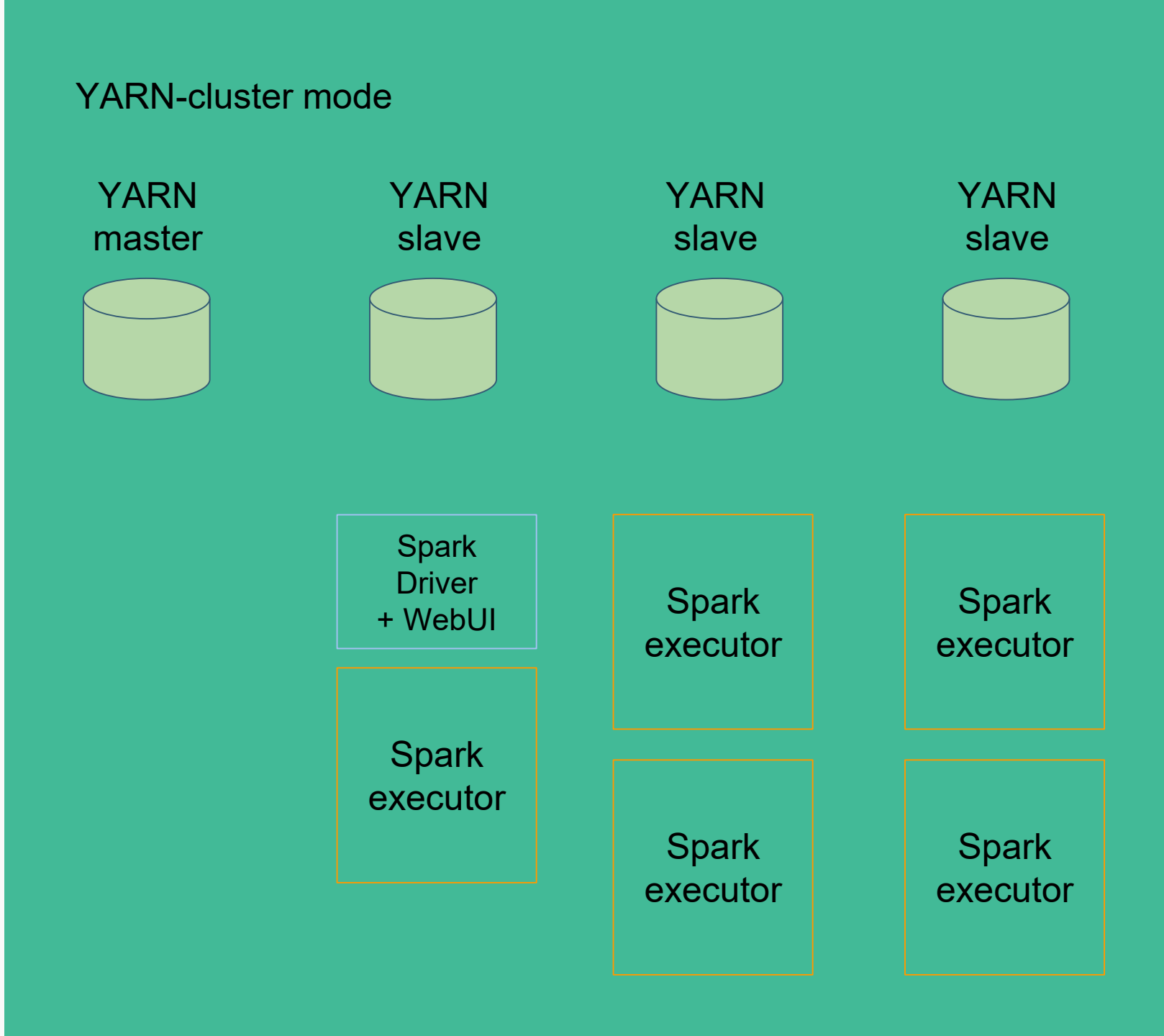
What is Spark ?

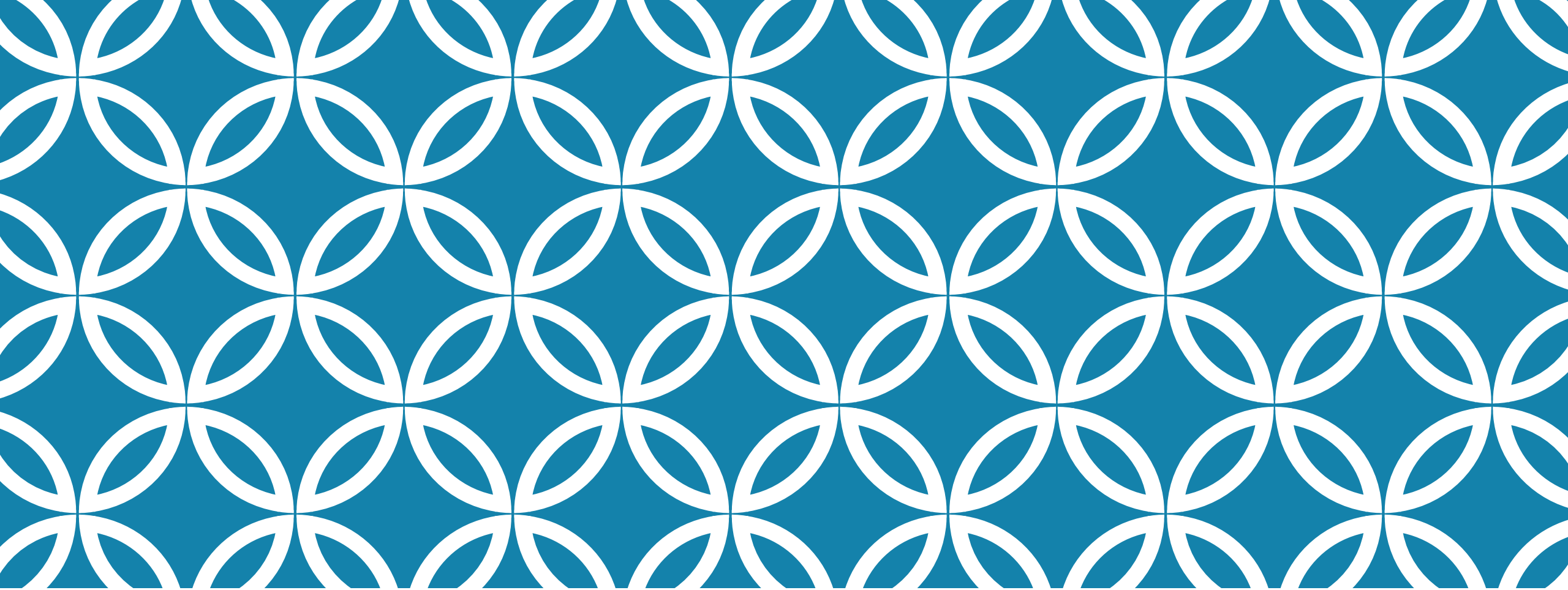
In-memory computing
Data-centric through Resilient Distributed Datasets (RDDs)
More operations
More flexibility



What is Spark ?

In-memory computing
Data-centric through Resilient Distributed Datasets (RDDs)
More operations
More flexibility





RDD:

Resilient Distributed Datasets

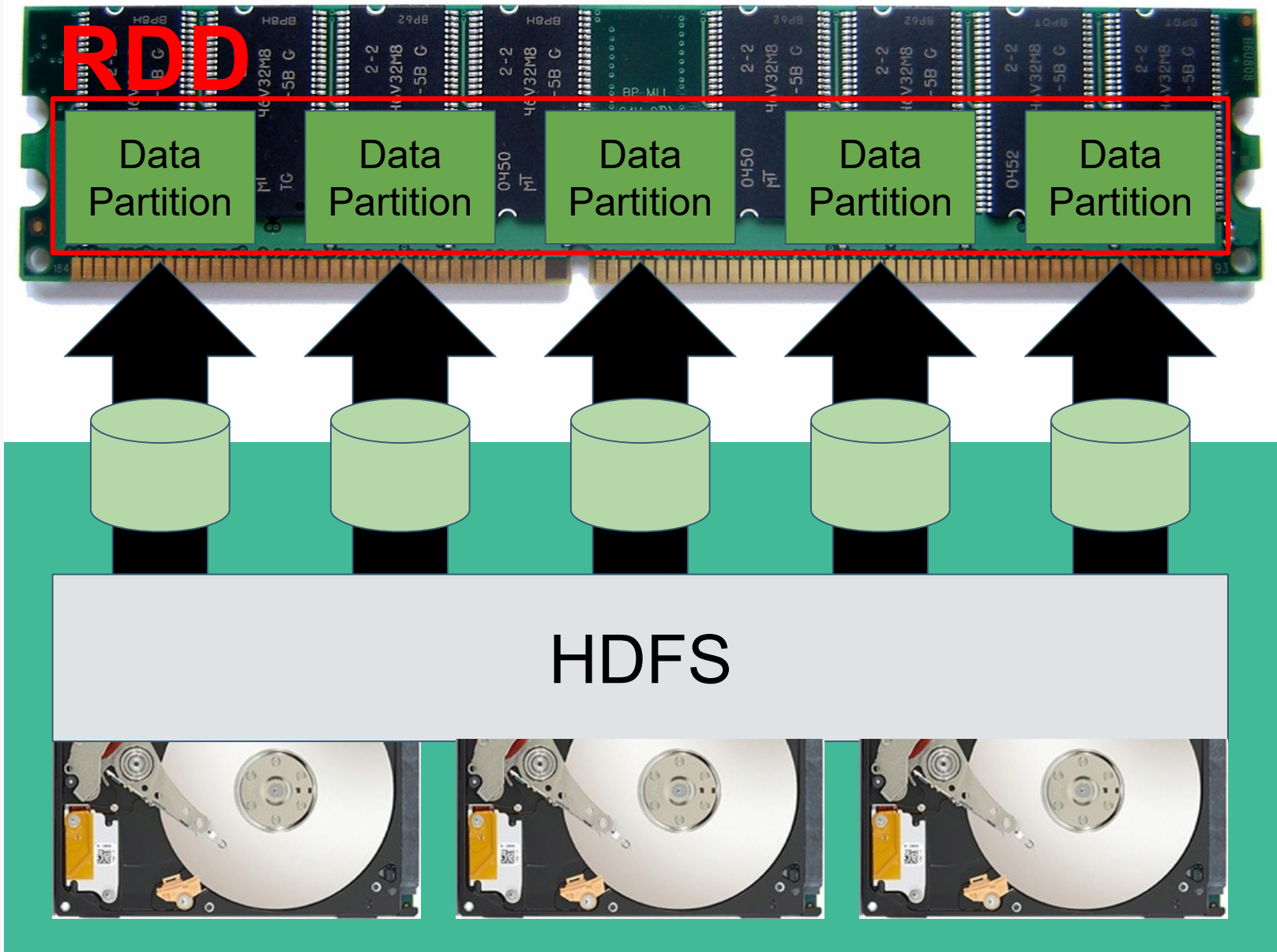
RDD

Spark est basé sur le concept des *resilient distributed dataset* (RDD).

Les RDD sont des collection d'élément tolérantes aux pannes sur lesquelles on peut faire des opérations en parallèle.

Il y deux manières de créer un RDD:

- En distribuant (parallelizing) une collection qui se trouve dans votre driver (programme lance sur le master).
- En référençant un dataset qui se trouve sur un stockage externe. Par exemple : HDFS, HBase, or any data source offering a Hadoop InputFormat.



RDD

Spark est basé sur le concept des *resilient distributed dataset* (RDD).

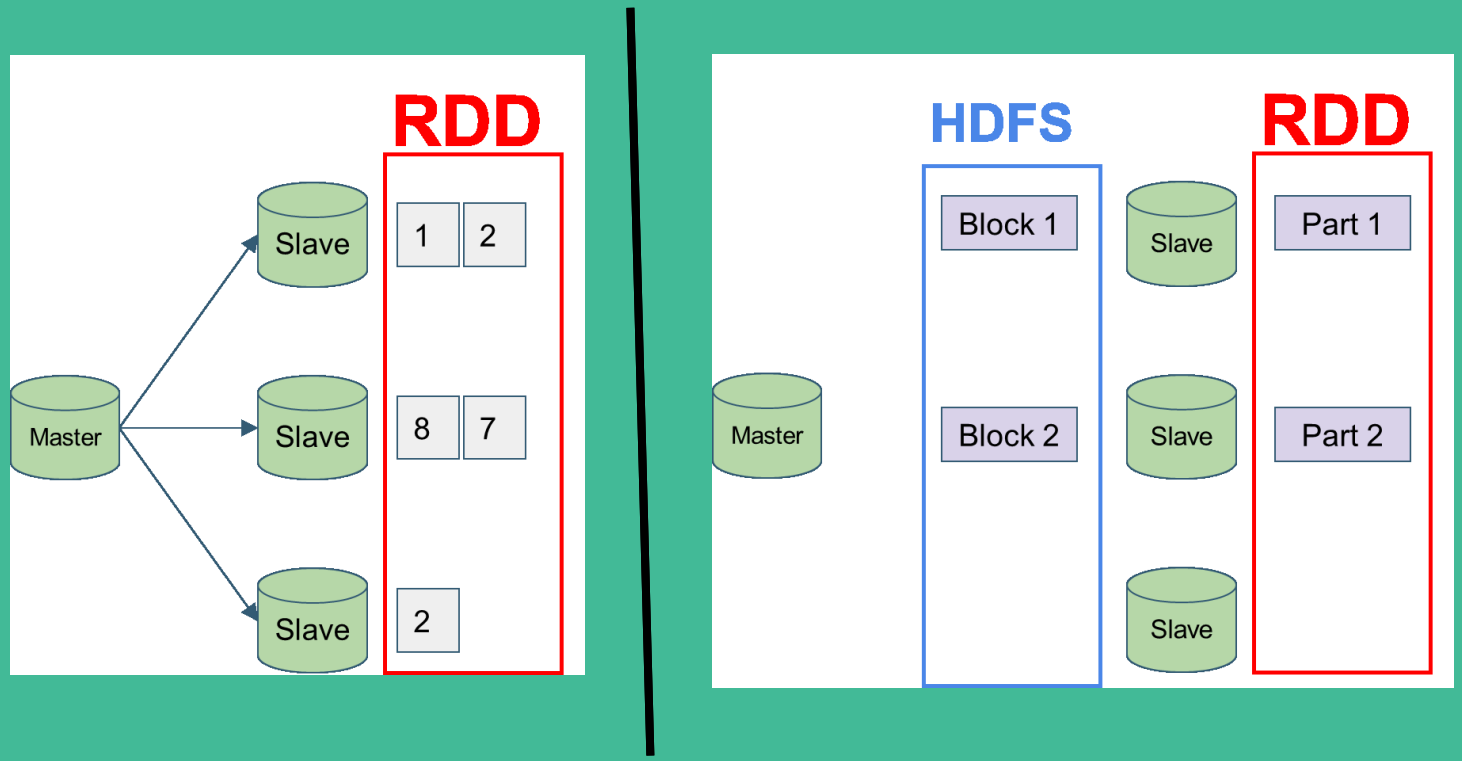
Les RDD sont des collection d'élément tolérantes aux pannes sur lesquelles on peut faire des opérations en parallèle.

Il y deux manières de créer un RDD:

- En distribuant (parallelizing) une collection qui se trouve dans votre driver (programme lance sur le master).
- En référençant un dataset qui se trouve sur un stockage externe. Par exemple : HDFS, HBase, or any data source offering a Hadoop InputFormat.

```
JavaRDD<Integer> rdd;
rdd  = context.parallelize(Arrays.asList(1, 2, 8, 7, 2),3);

String inputPath;
JavaRDD<String> rdd2;
rdd2  = context.textFile(inputPath);
```



Evaluation paresseuse

Les partitions d'un RDD ne sont matérialisées que si besoin.

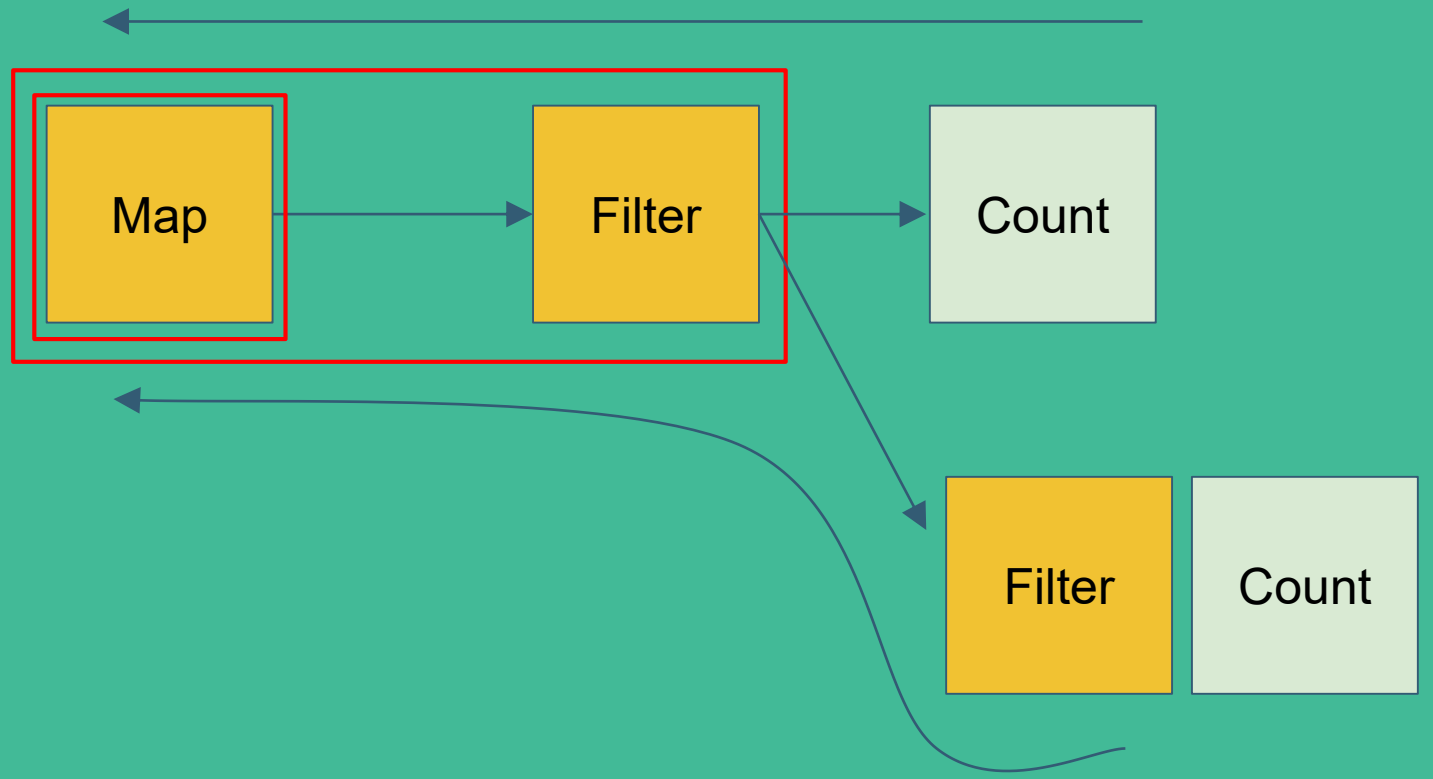
Tout calcul génère un nouveau RDD qui représente le résultat de l'évaluation de la chaîne d'opérations effectuées jusqu'à lui.

La matérialisation intervient uniquement lorsque l'on accède au résultat dans le driver ou lorsque l'on enregistre le résultat sur disque.

RIEN NE SE PASSE TANT QUE L'ON N'ACCEDE PAS AUX RESULTATS.

Les données temporaire ne sont pas stockées.

```
JavaRDD<Integer> rdd   = sc.parallelize(Arrays.asList(1, 2, 3, 4),10);
rdd = rdd.map((x) -> x * 10);
rdd = rdd.filter((x) -> x%2 == 0);
System.out(rdd.count());
rdd = rdd.filter((x) -> x > 3);
System.out(rdd.count());
```



Evaluation paresseuse

Pour améliorer les performances il est possible mettre en cache des résultats partiels de calculs.

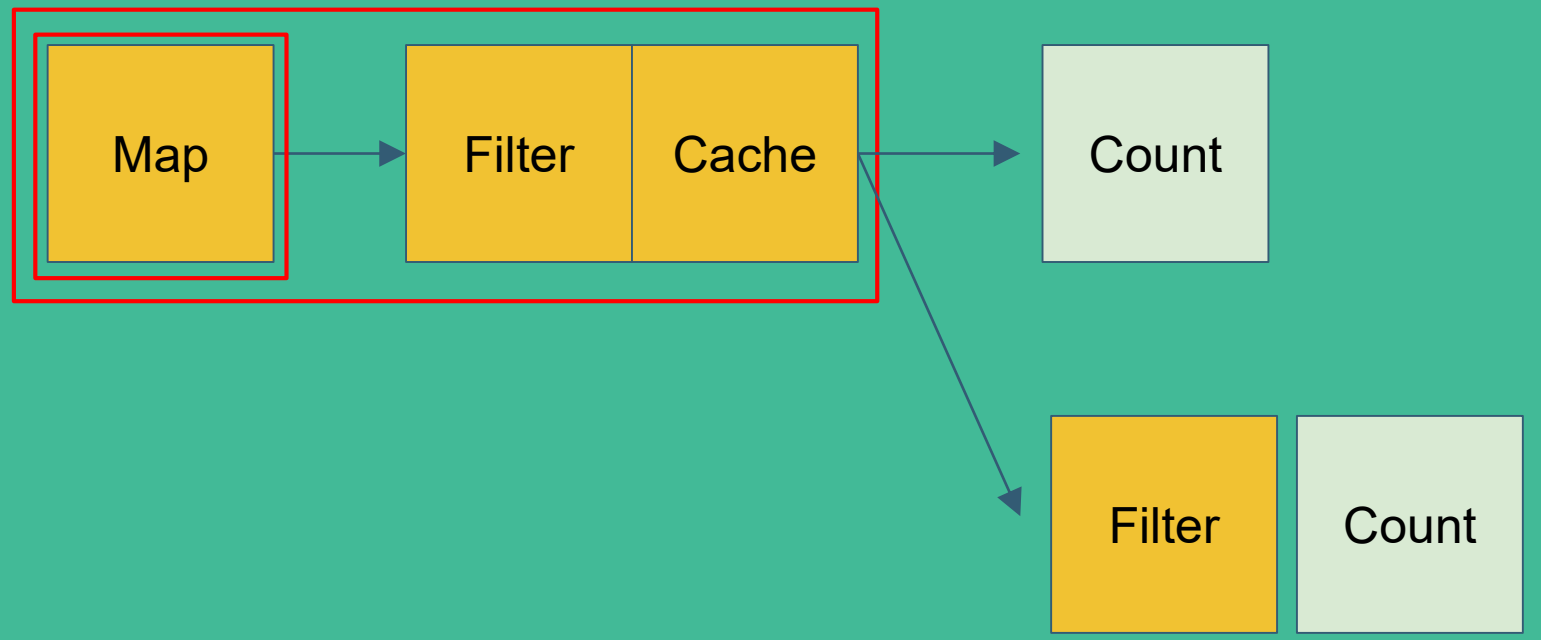
Les données mises en cache sont dite persistante et reste en mémoire vive dans les machines.

ATTENTION A NE PAS SATURER LA MÉMOIRE DES MACHINES

Il est possible de mettre des données persistante sur disque.

On peut réaliser des DAG d'évaluation très complexes…

```
JavaRDD<Integer> rdd   = sc.parallelize(Arrays.asList(1, 2, 3, 4),10);
rdd = rdd.map((x) -> x * 10);
rdd = rdd.filter((x) -> x%2 == 0);
rdd = rdd.cache();
System.out(rdd.count());
rdd = rdd.filter((x) -> x > 3);
System.out(rdd.count());
```



Evaluation paresseuse

Pour améliorer les performances il est possible mettre en cache des résultats partiels de calculs.

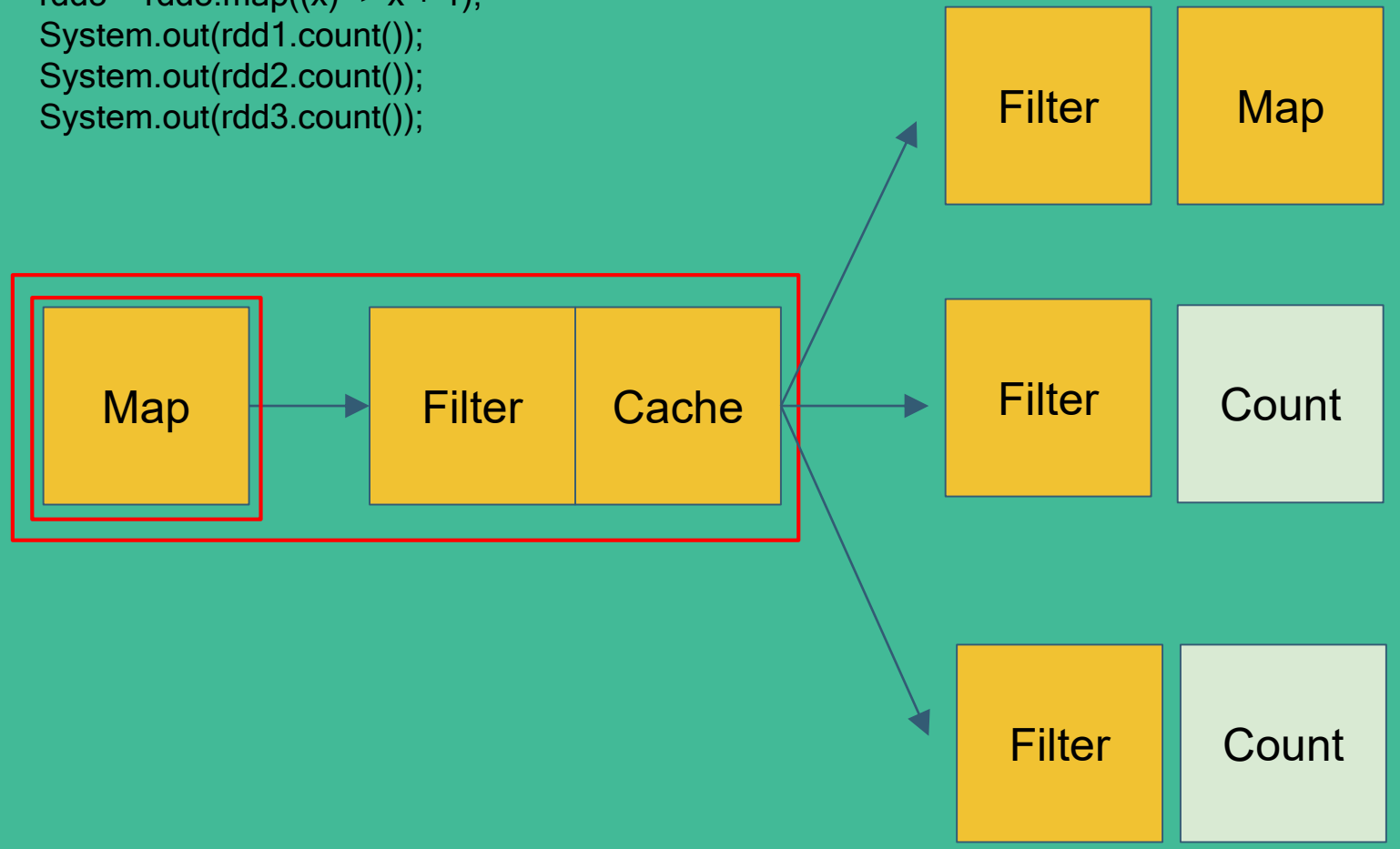
Les données mises en cache sont dite persistante et reste en mémoire vive dans les machines.

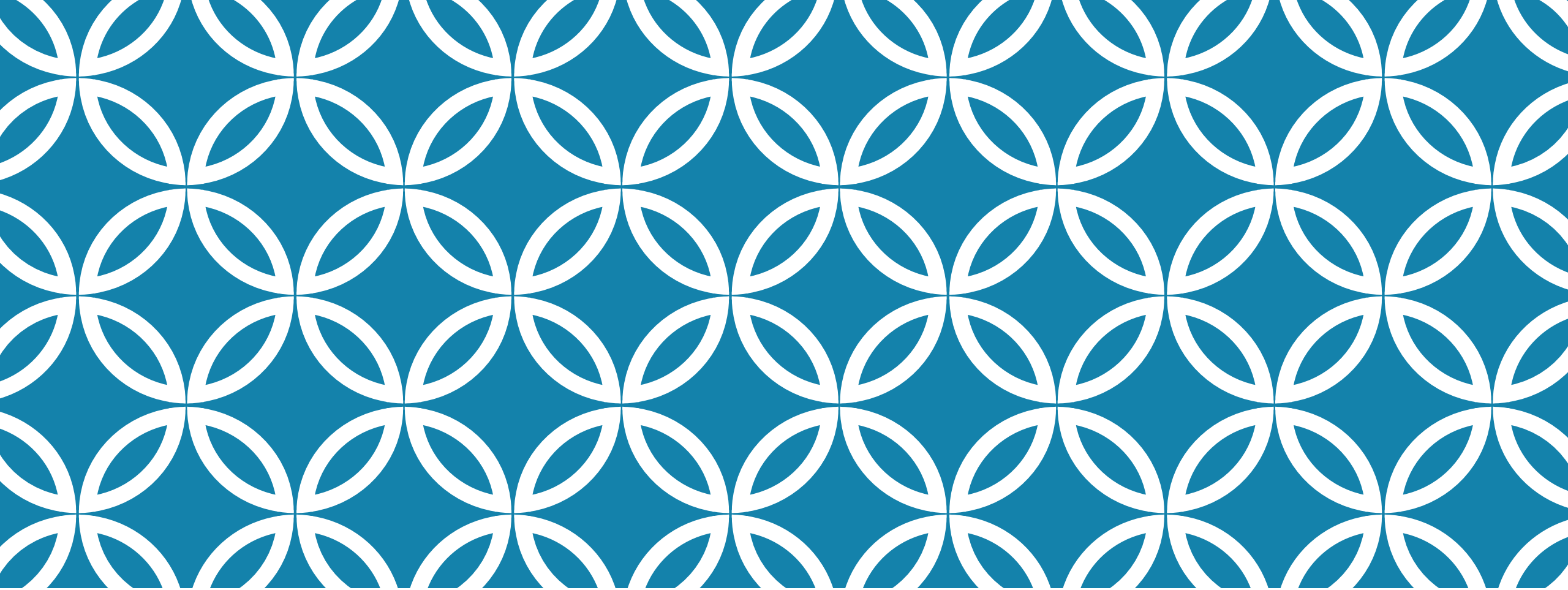
ATTENTION A NE PAS SATURER LA MÉMOIRE DES MACHINES

Il est possible de mettre des données persistante sur disque.

On peut réaliser des DAG d'évaluation très complexes…

```
JavaRDD<Integer> rdd   = sc.parallelize(Arrays.asList(1, 2, 3, 4),10);
rdd = rdd.map((x) -> x * 10);
rdd = rdd.filter((x) -> x%2 == 0);
rdd.cache();
System.out(rdd.count());
JavaRDD<Integer>   rdd1 = rdd.filter((x) -> x > 3);
JavaRDD<Integer>   rdd2 = rdd.filter((x) -> x < 3);
JavaRDD<Integer>   rdd3 = rdd.filter((x) -> x == 3);
rdd3 = rdd3.map((x) -> x + 1);
System.out(rdd1.count());
System.out(rdd2.count());
System.out(rdd3.count());
```





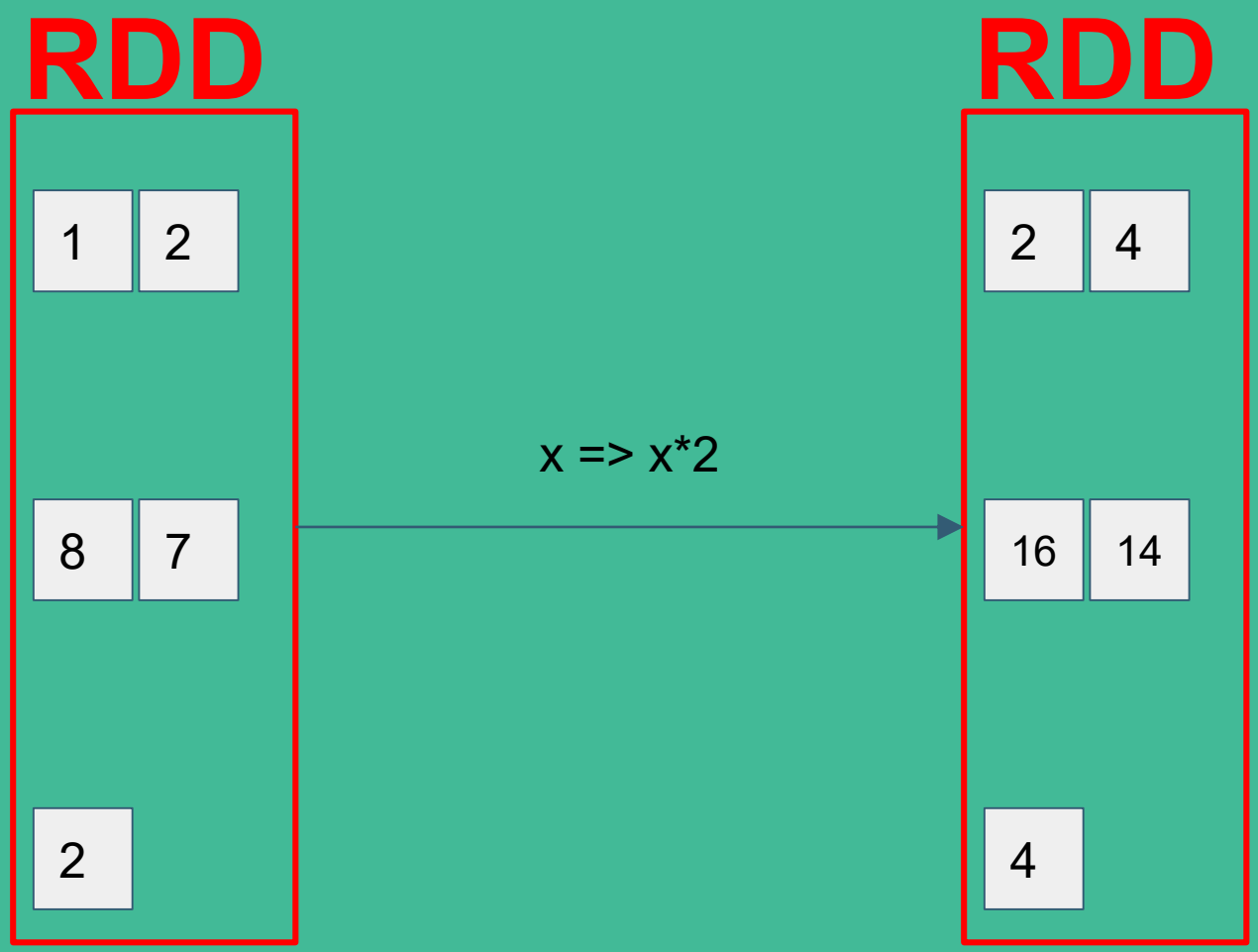
Opération sur les RDD

Spark operations on RDD[T]

map

Permet de transformer un RDD dans un autre RDD ayant la même nombre d'éléments, les types peuvent être différents.

On utilise pour cela des lambda fonction de Java 8 (on peut utiliser des classes mais c'est beaucoup moins pratique et ça ne suit pas la philosophie Spark).



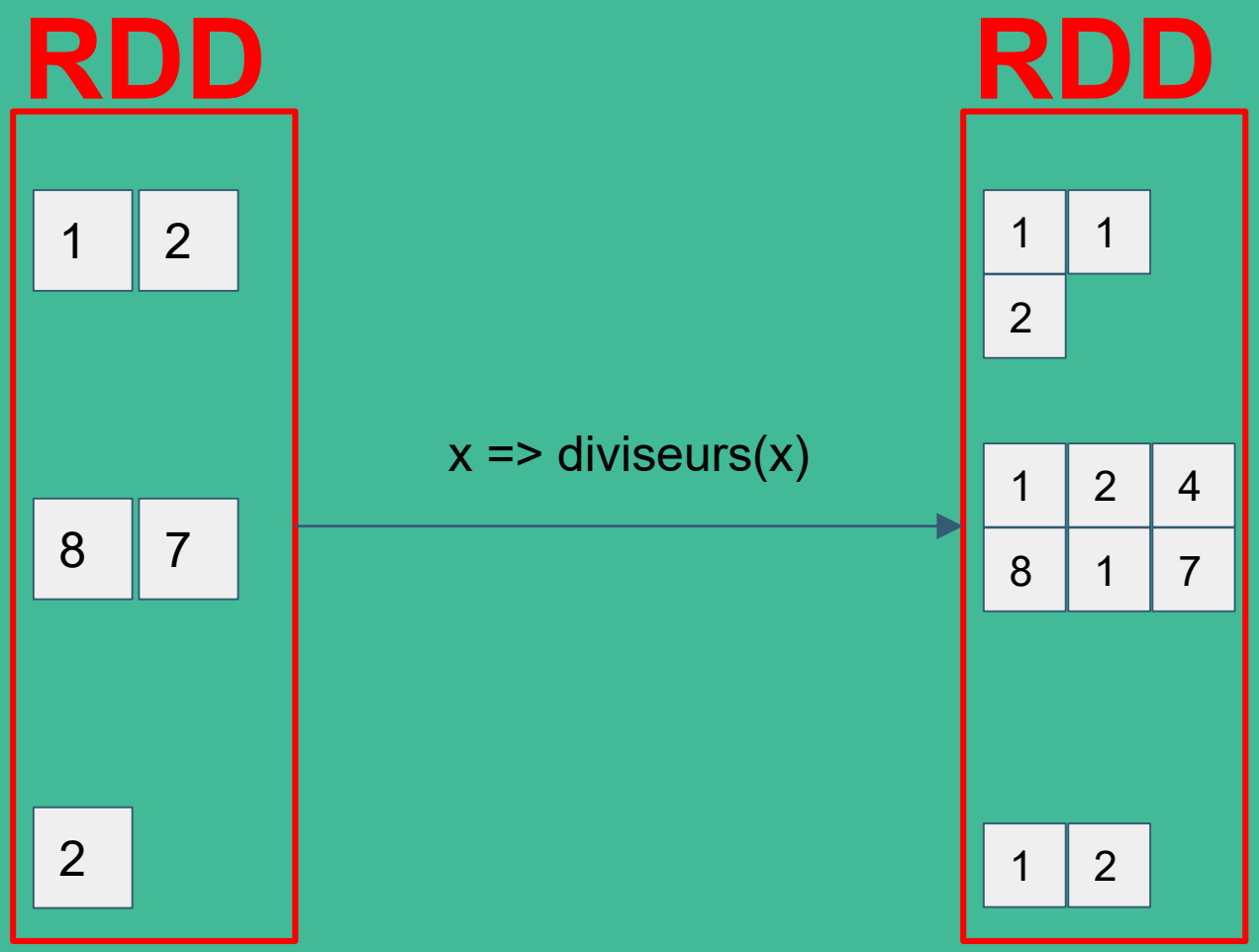

```
JavaRDD<Integer> rdd2 = rdd.map( (x) -> x * 2 )
```

Spark operations on RDD[T]

flatMap

Permet de construire un RDD de taille différente à partir du RDD existant.

La fonction de map doit retourner un itérateurs sur les valeurs à insérer.

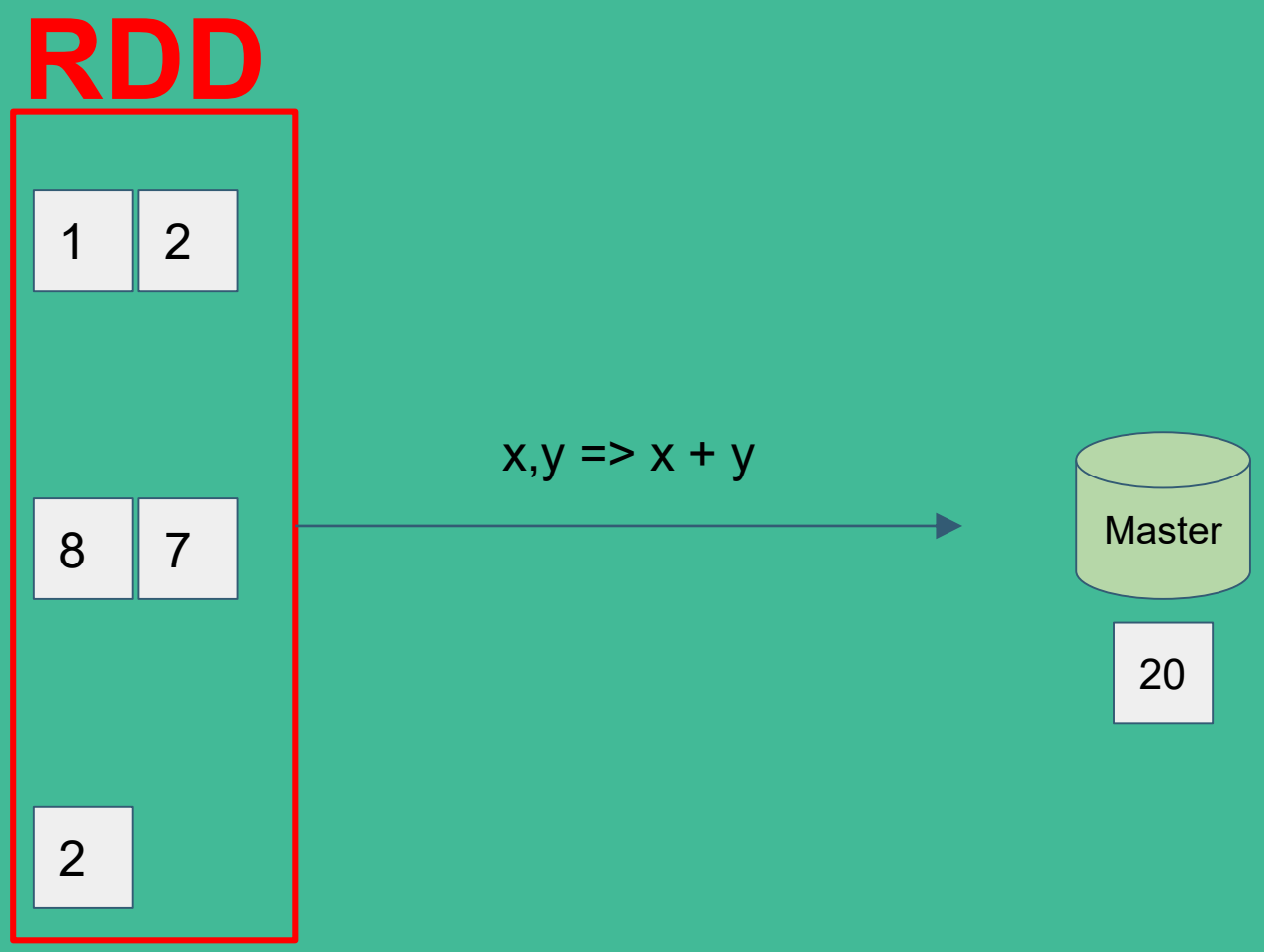



```
rdd2 = rdd2.flatMap(
(x) -> { List<Integer> res = new ArrayList<Integer>();
                for (int i=2; i<x; ++i)
                 If (x % i == 0)
                  res.add(i);
                 return res.iterator();
                } );
```

Spark operations on RDD[T]

Reduce.

La fonction reduce permet de transformer un dans une valeur. Les opérations doivent être associatives car elles sont faites en parallèle.



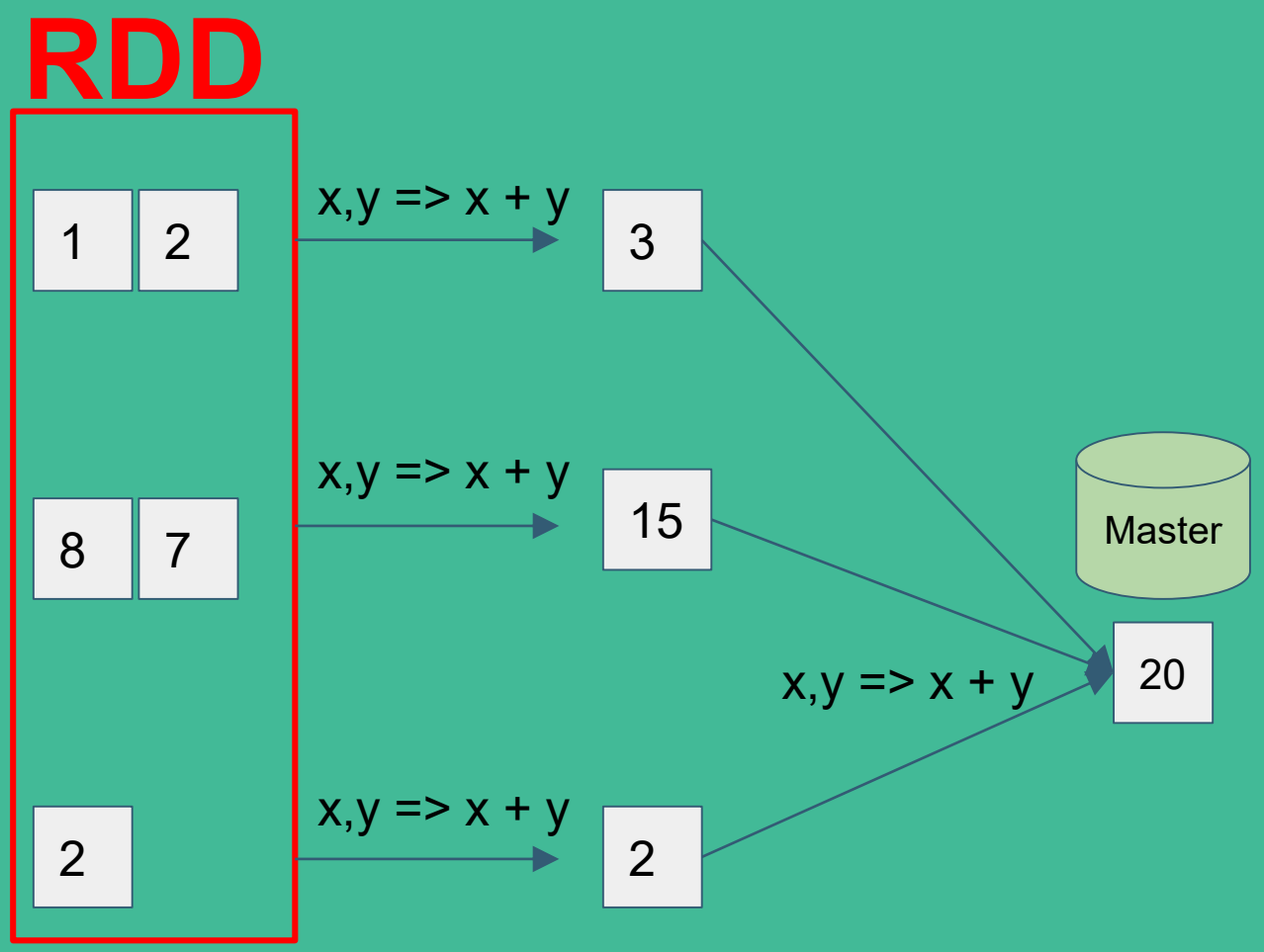

```
int i = rdd2.reduce((x,y) -> x + y);
```

Spark operations on RDD[T]

La fonction aggregate permet de faire une réduction en utilisant un type intermédiaire.

Le premier paramètre représente un agrégat null (initialisation). Le deuxième une fonction qui agrège des éléments dans une agrégat, et le troisième paramètre une fonction qui permet d'agréger des agrégat entre eux.

Cette fonction permet de mettre en place des technique de calcul comme celui fait pour la moyenne en map-reduce.




```
rdd2.aggregate(0, (a, b) -> a + b, (a, c) -> a + c);
```

Spark operations on RDD[T]

Aggregate:

La fonction aggregate permet de faire une réduction en utilisant un type intermédiaire.

Le premier paramètre représente un agrégat null (initialisation). Le deuxième une fonction qui agrège des éléments dans une agrégat, et le troisième paramètre une fonction qui permet d'agréger des agrégat entre eux.

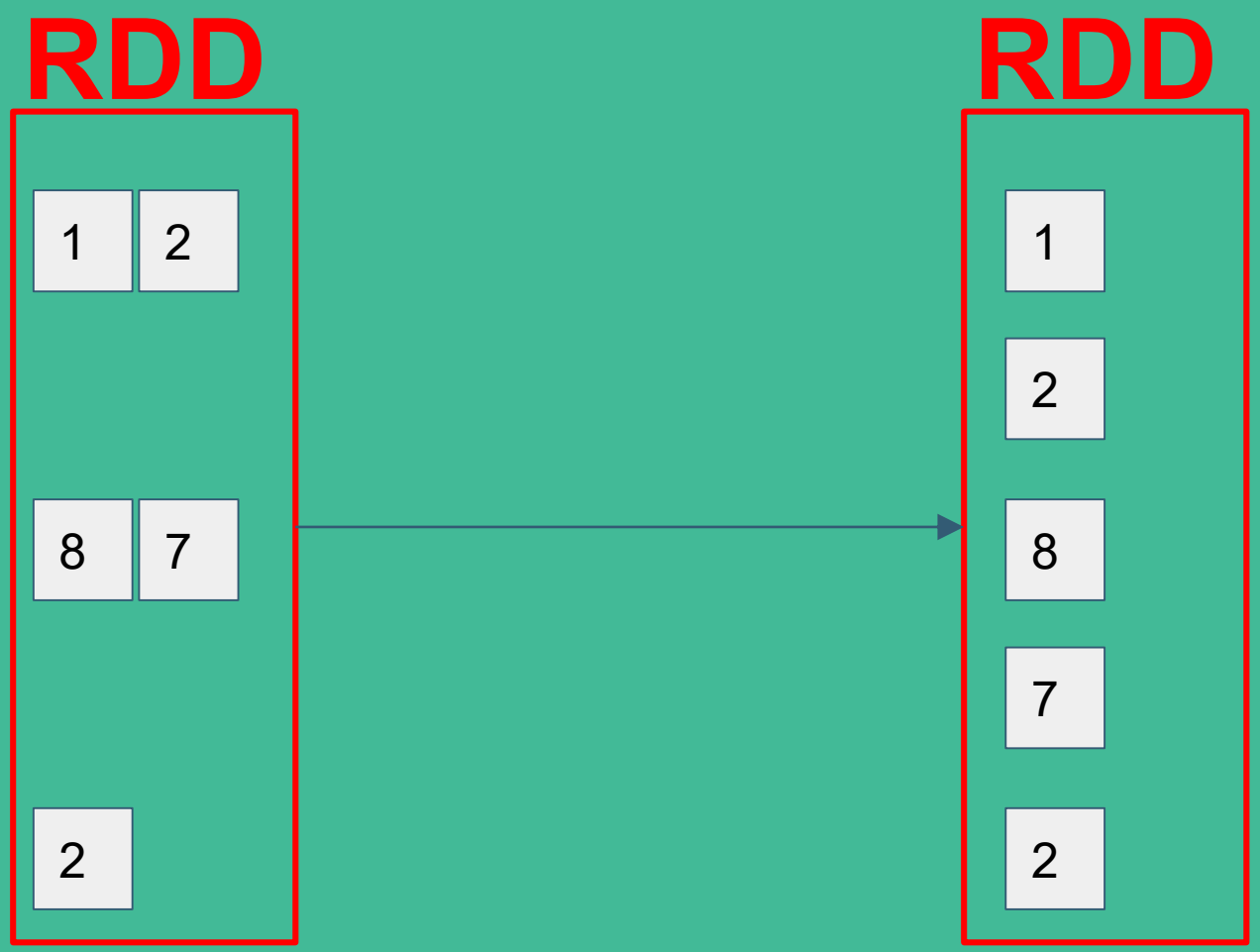
Cette fonction permet de mettre en place des technique de calcul comme celui fait pour la moyenne en map-reduce.

```
public static class AvgCount implements Serializable {
    public AvgCount(int total, int num) {
        total_ = total;
        num_ = num;
    }
    private int total_;
    private int num_;
    public float avg() {
        return total_ / (float) num_;
    }
}
//=========================
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4),10);
rdd = rdd.map((x) -> x * 10);
AvgCount initial = new AvgCount(0,0);
AvgCount result = rdd.aggregate(initial,
        (AvgCount a, Integer x) -> {
            a.total_ += x;
            a.num_++;
            return a;
        }, (AvgCount a, AvgCount b) -> {
            a.total_ += b.total_;
            a.num_   += b.num_;
            return a;
        }
 );
 System.out.println(result.avg());
```

Spark operations on RDD[T]

repartition

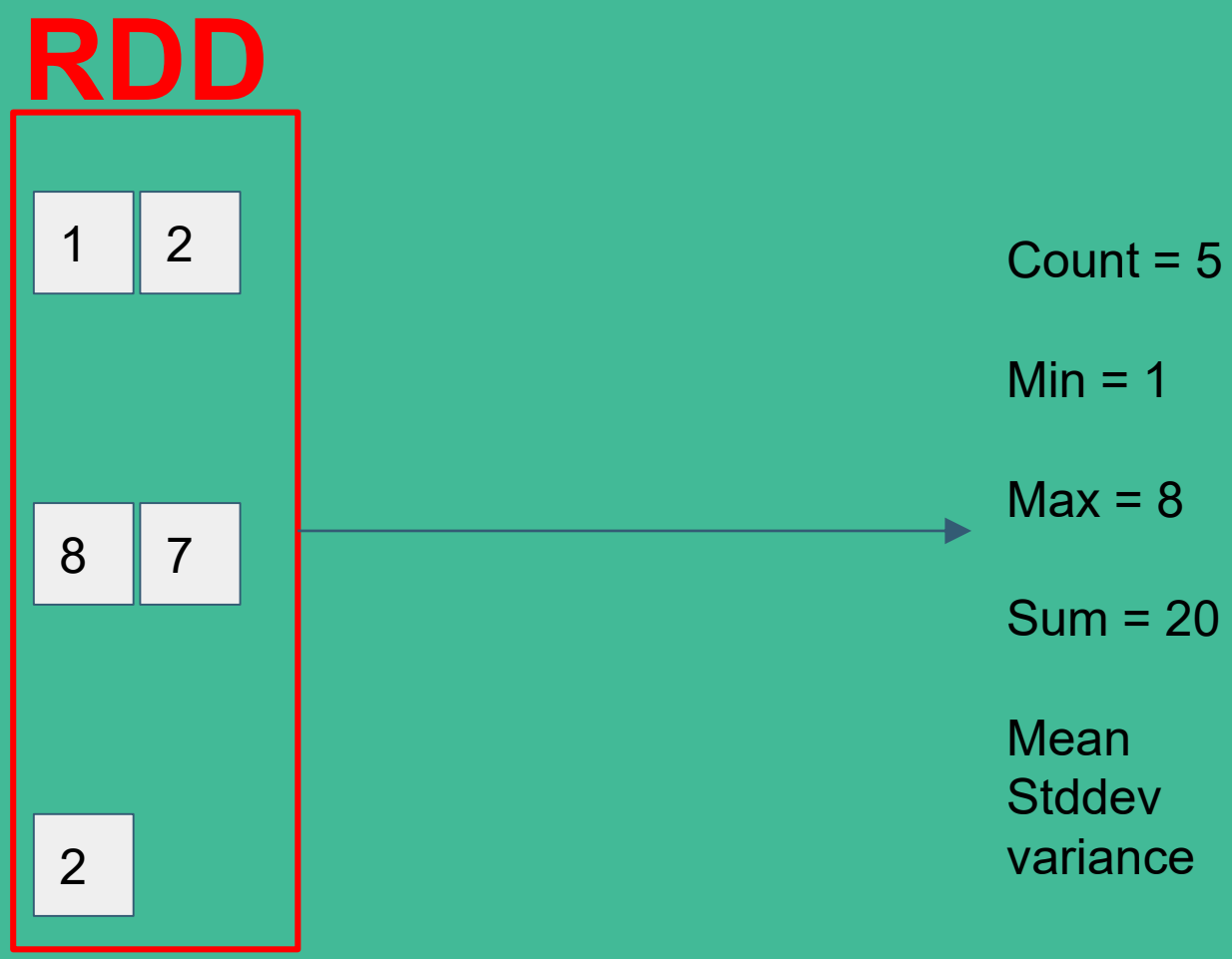
La fonction repartition permet de repartionner les données dans un RDD.



rdd = rdd.repartition(5)

Statistiques

Sur tous les RDD ont peut apperler la fonction count. Sur les RDD de type JavaDoubleRDD on peut appeler la fonction stats qui calcule de manière efficace la moyenne la médiane, l'écart, le min et le max.

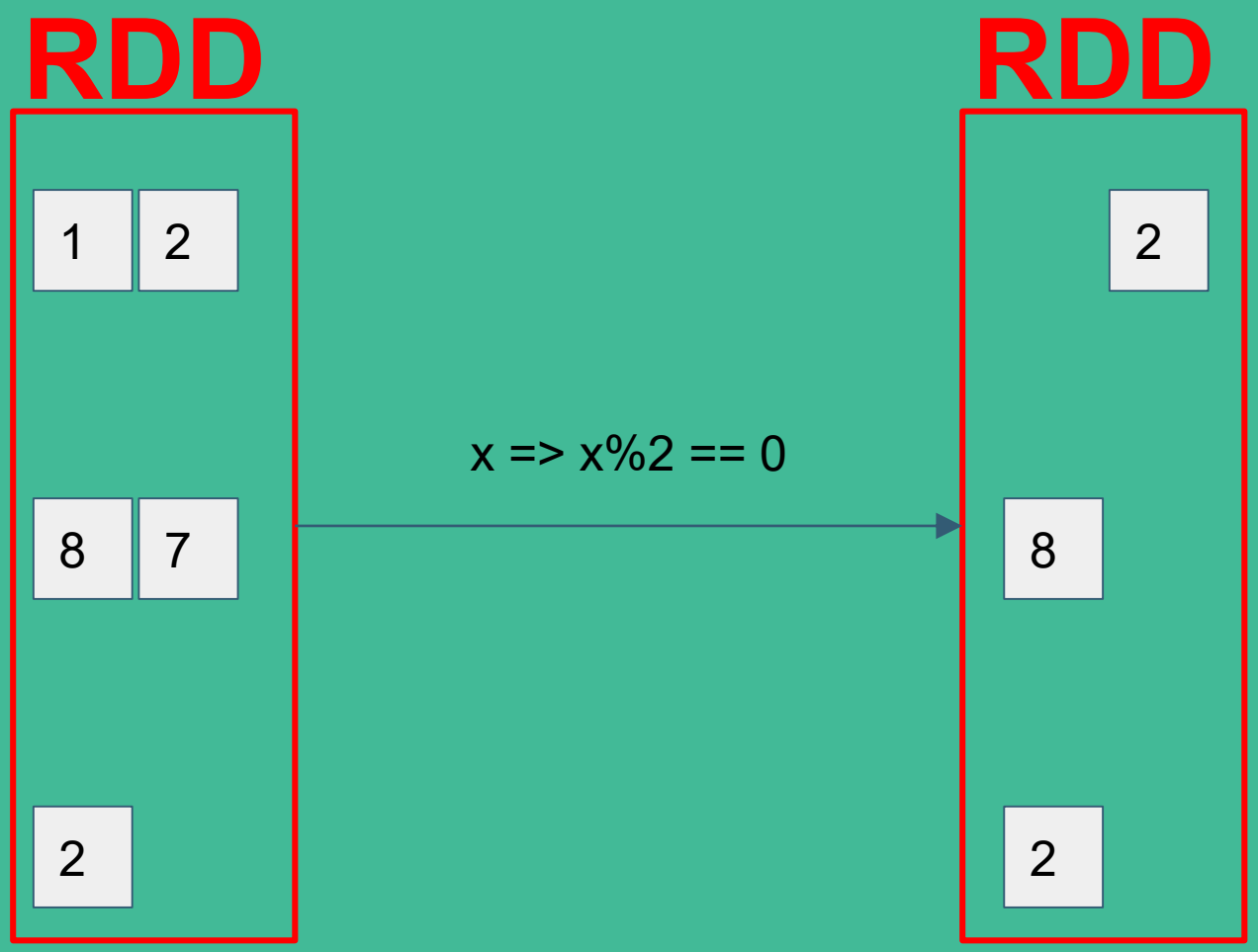



```
JavaDoubleRDD rdd3;
StatCounter stat = rdd3.stats();
stat.min();
```

Advanced operations on RDD[T]

Filter

La fonction filter permet de créer un nouveau RDD en filtrant les valeurs d'un autre RDD.

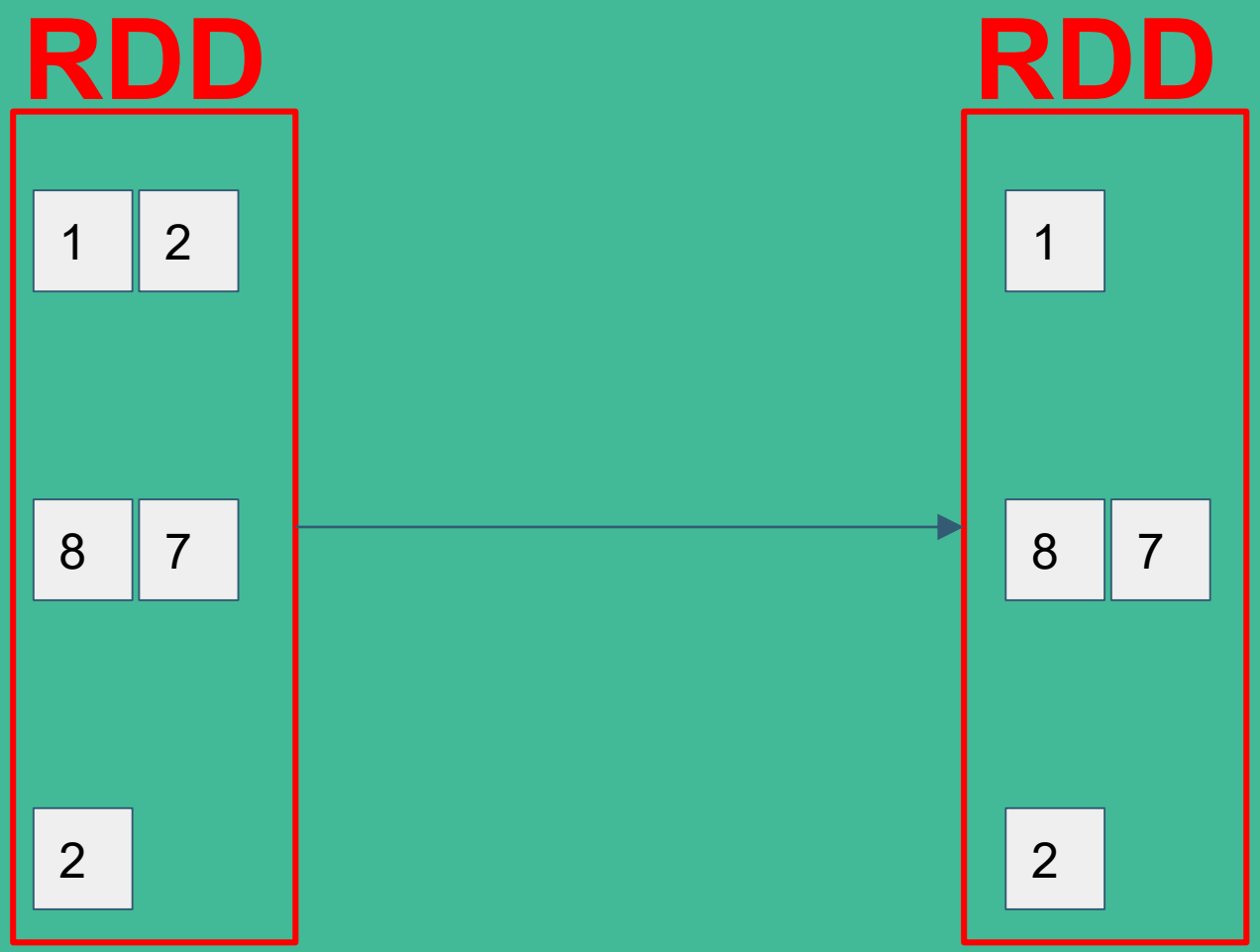



```
rdd = rdd.filter( (x) -> x%2 == 0).repartition(2)
```

Advanced operations on RDD[T]

Distinct

La fonction distrinct permet de supprimer les doublons dans un RDD.

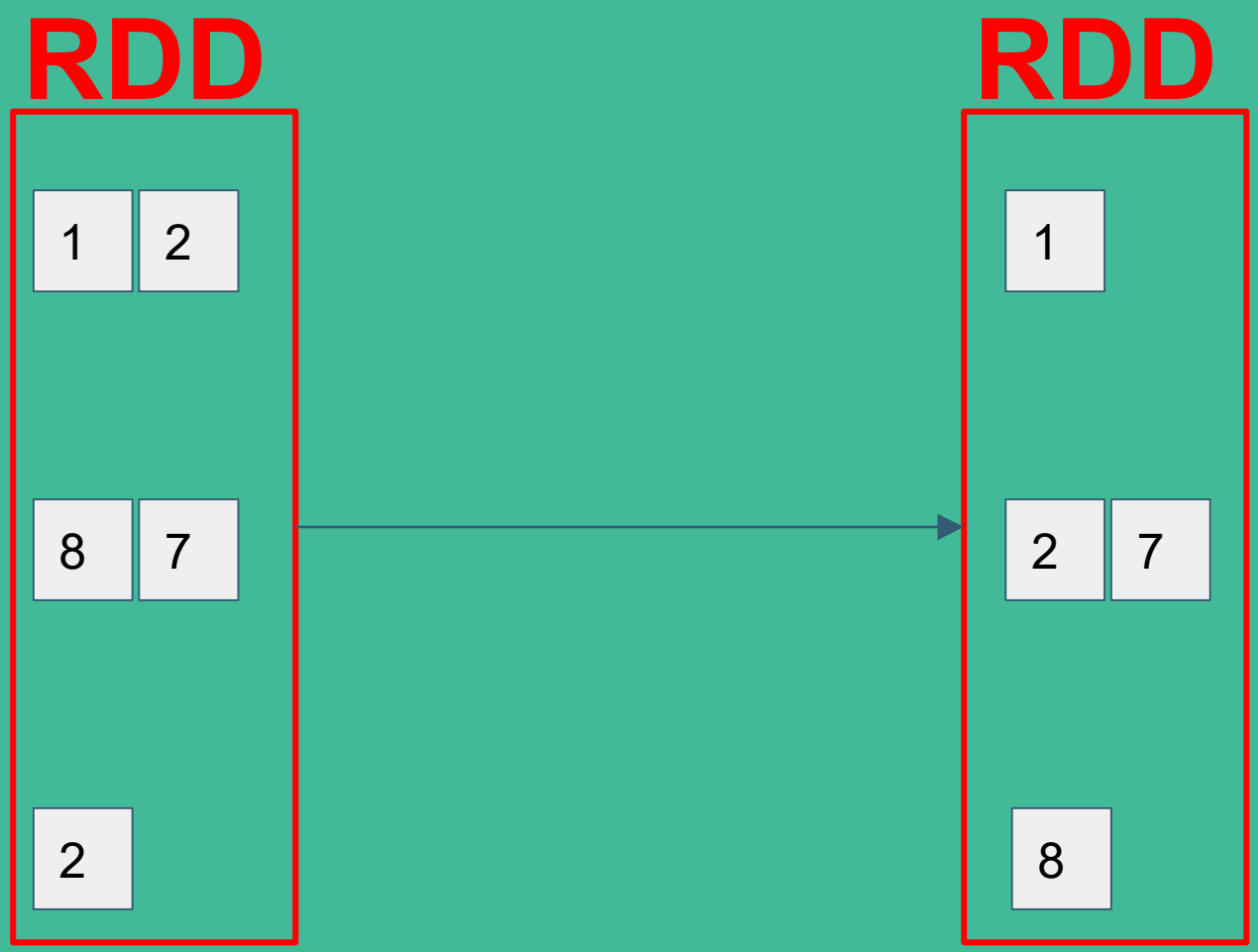


rdd = rdd.distinct()

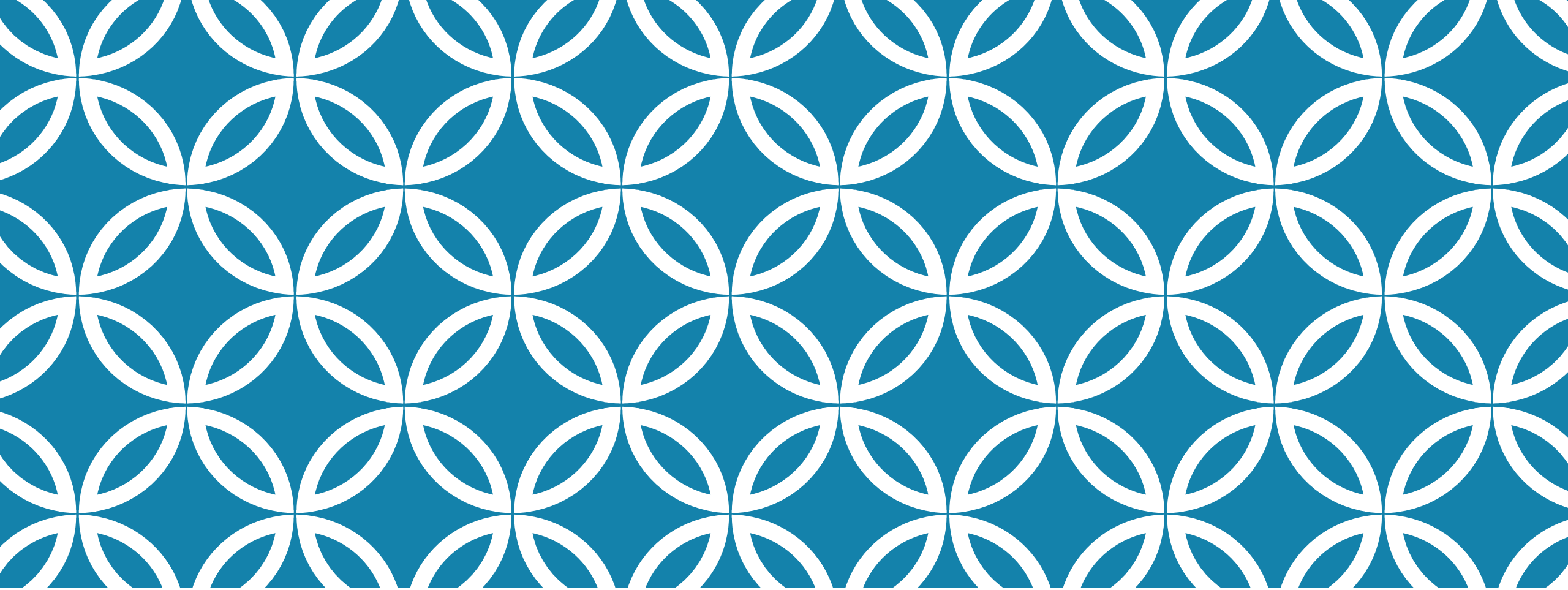
Advanced operations on RDD[T]

sortBy

La fonction sortBy permet de trier les éléments dans un RDD.



rdd2 = rdd2.sortBy((Integer x) -> (**double) x, true, rdd2.getNumPartitions());**



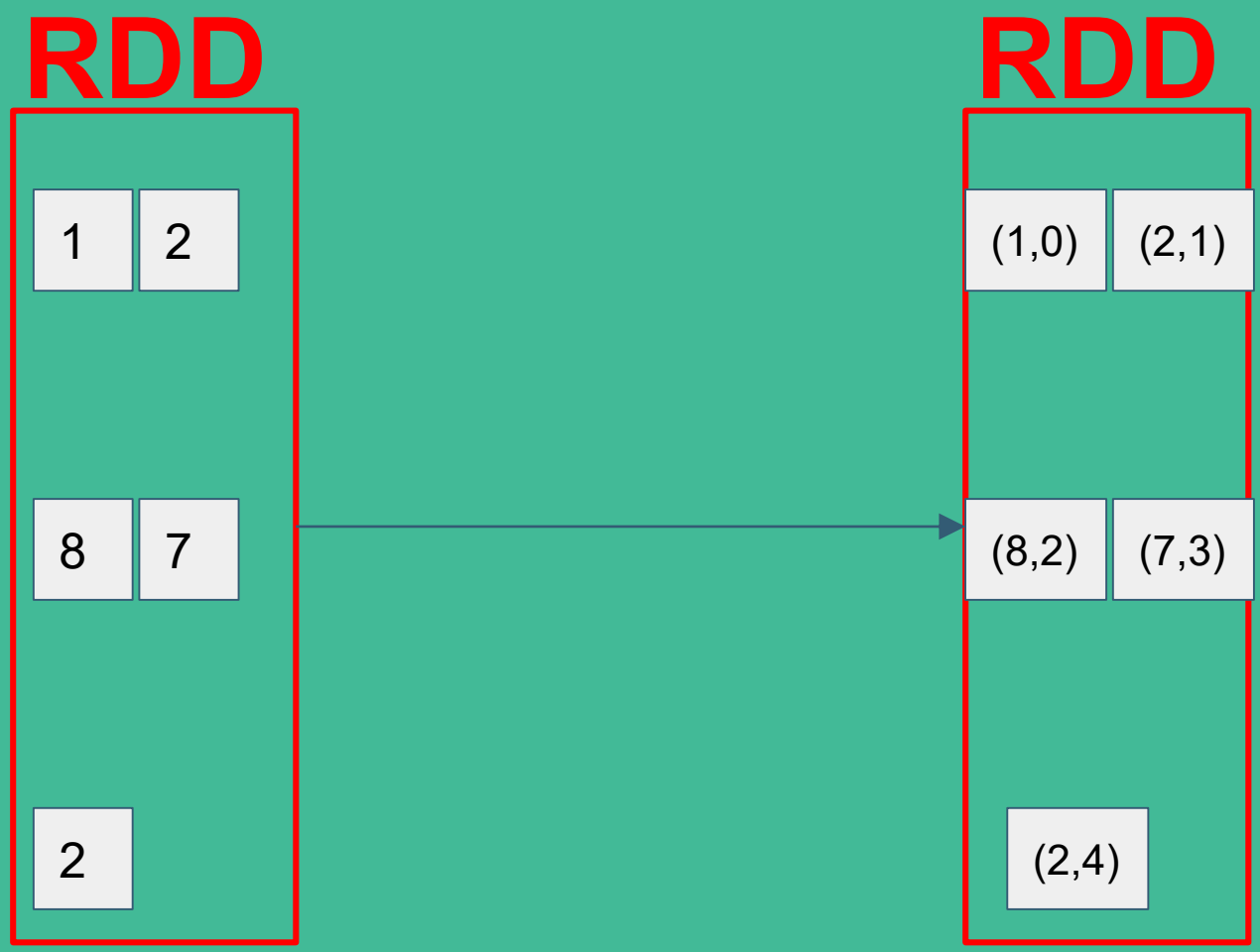
RDD -> PairRDD

Advanced operations on RDD[T]

zipWithIndex

La function zipWithIndew permet de transformer un RDD simple en pairRDD

La clé est la valeur de lancien RDD et la valeur et l'index dans l'ancient RDD.

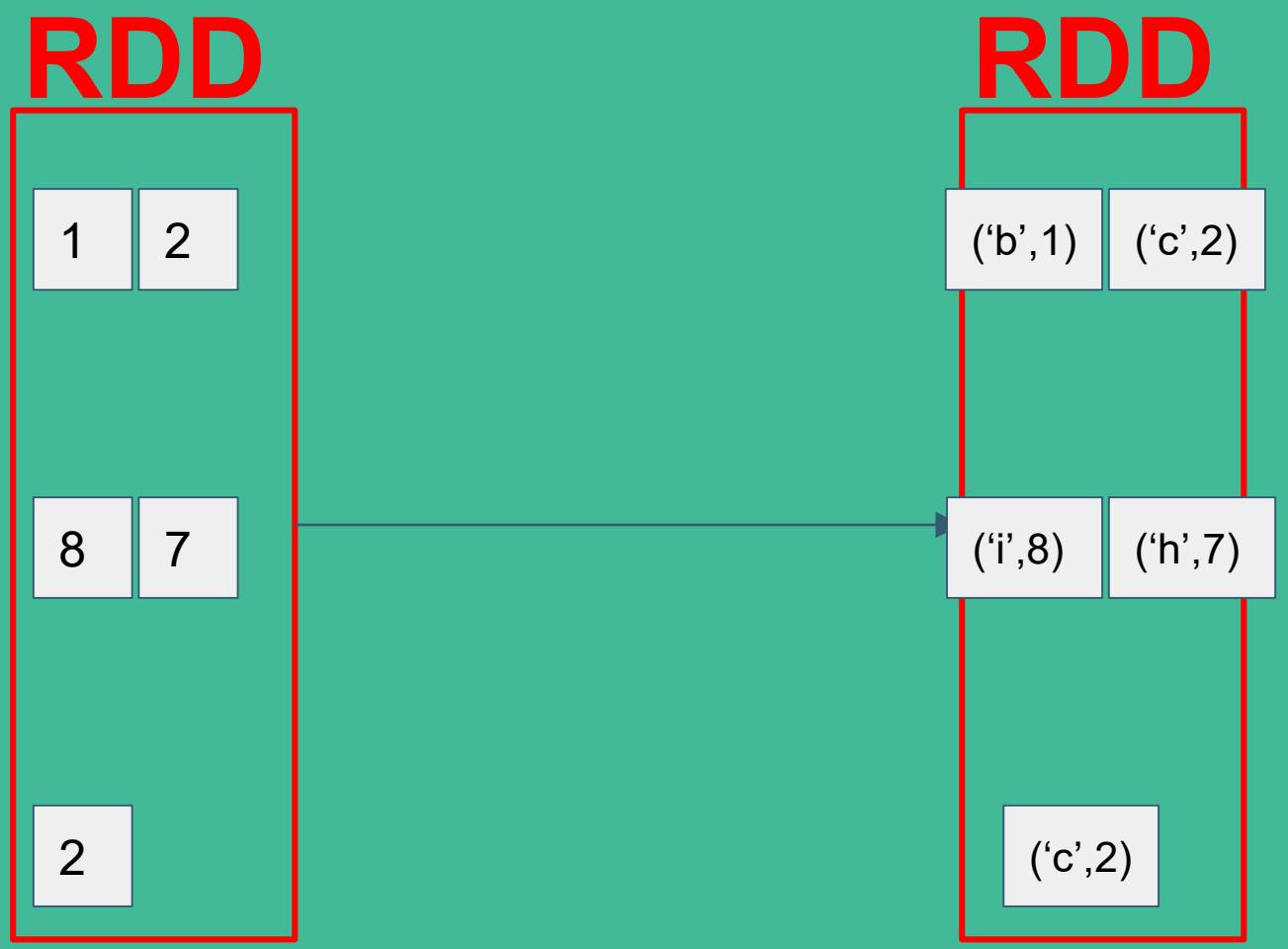



```
JavaPairRDD<String, Long> rddP = rdd.zipWithIndex();
```

Advanced operations on RDD[T]

keyBy

La fonction keyBy permet de générer une clé en fonction des éléments contenu dans un RDD.



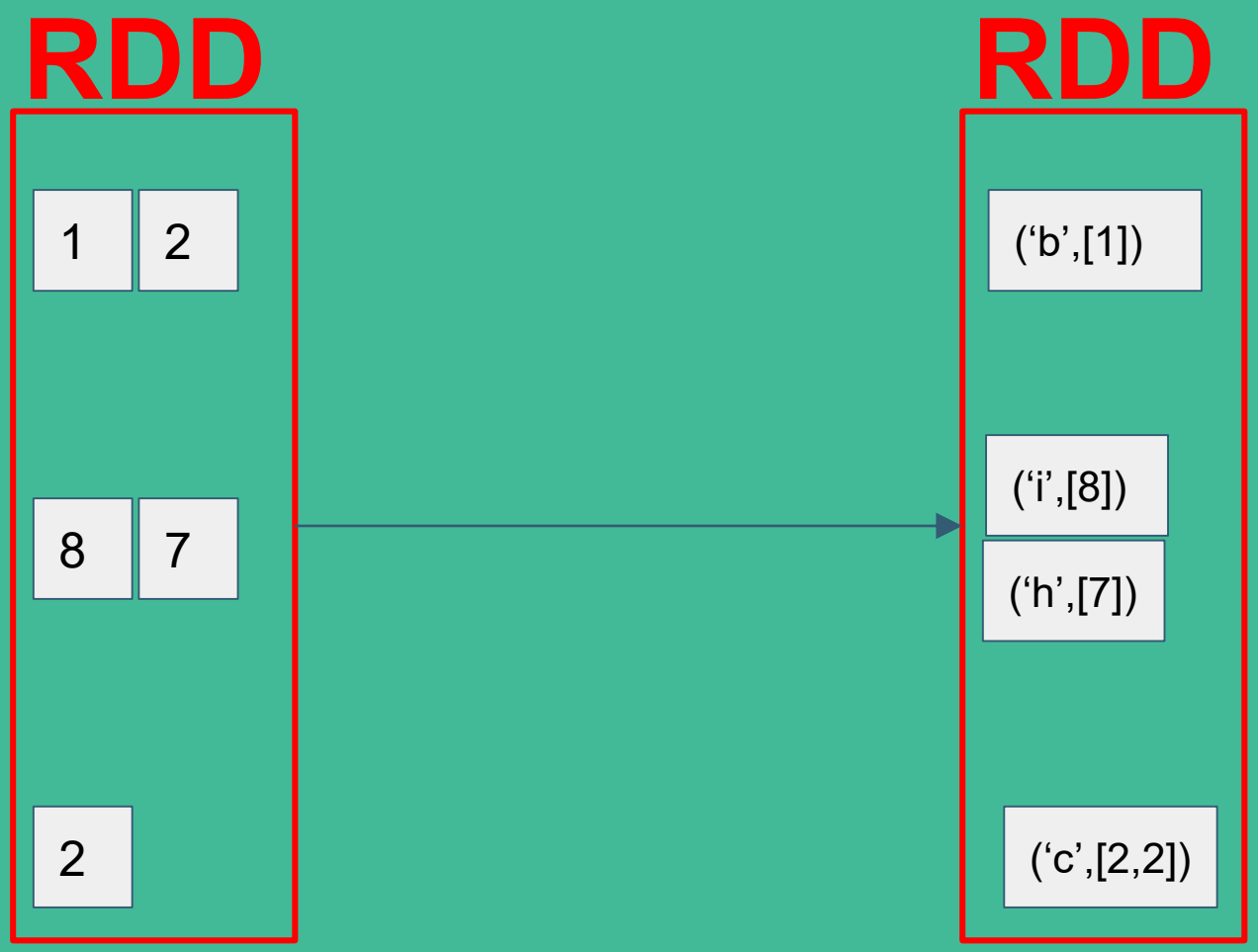

```
JavaPairRDD<Char, Double> rdd2CD = rdd.keyBy((x) -> 'a' + x);
```

```
JavaPairRDD<Integre, Double> rddID = rdd3.keyBy((x) ->
Math.floor(Math.log10(x)));
```

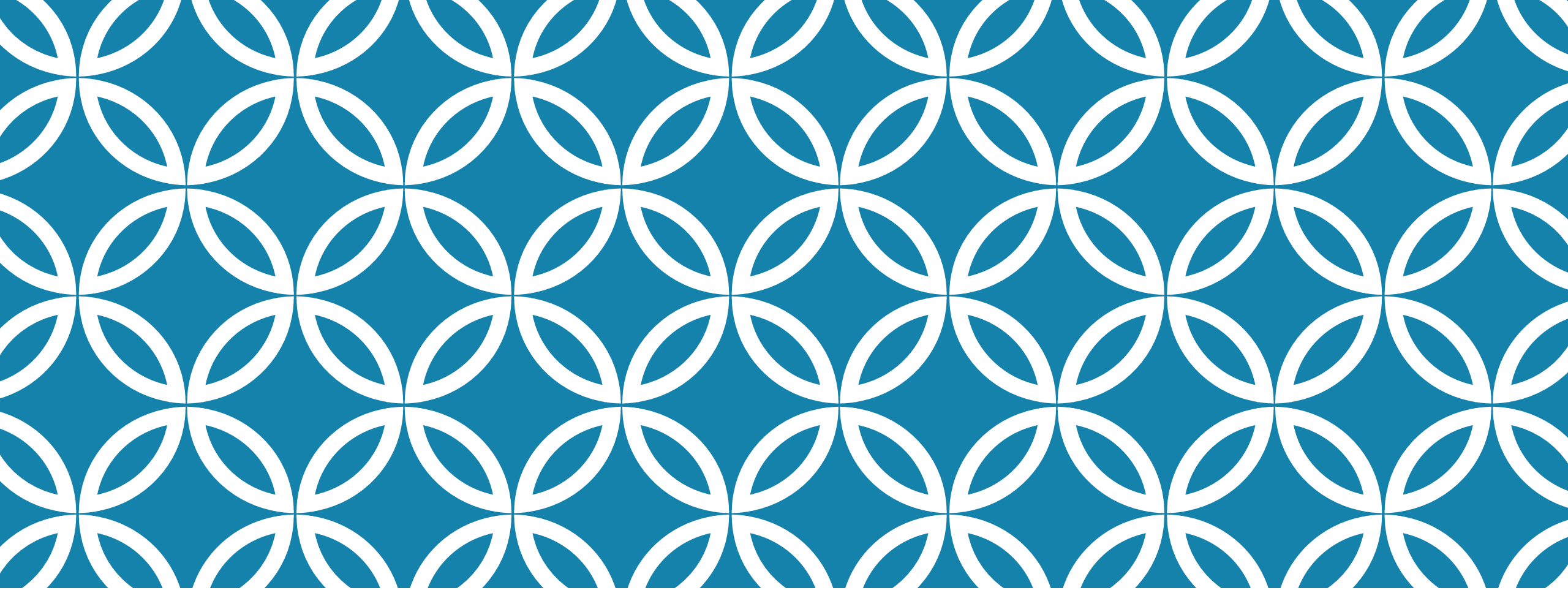
Advanced operations on RDD[T]

groupBy

La fonction groupBy permet de faire la même chose que byKey mais elle groupe en plus les éléments qui ont la même clé.



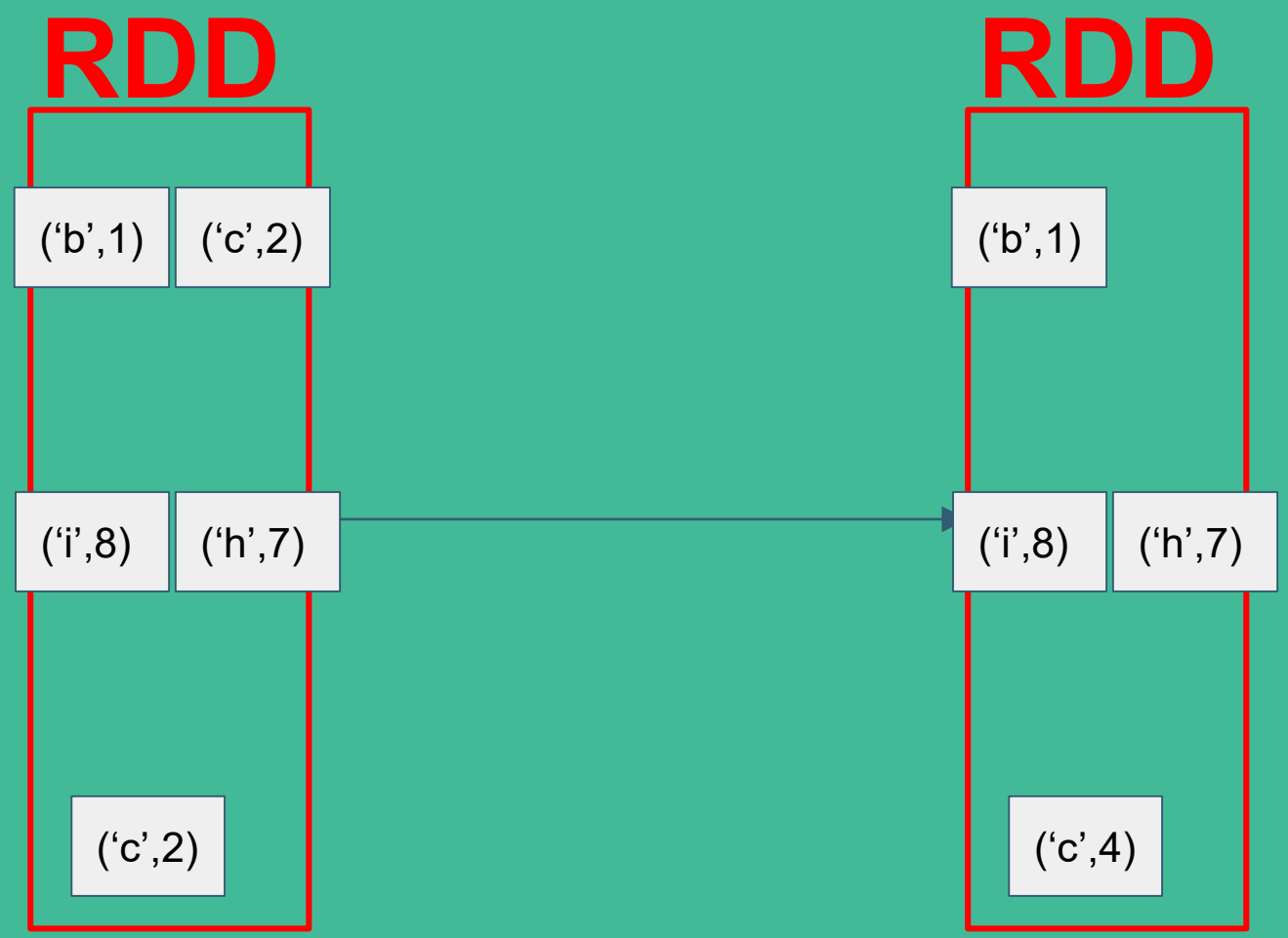

```
JavaPairRDD<Char, Iterable<Double> > rdd2CD =
rdd.groupBy((x) -> 'a' + x);
```



RDD -> PairRDD

Spark operations on RDD[(K,V)]

reduceByKey
aggregateByKey
groupByKey
Values
mapValues
join




```
var rddReduced : RDD[(Char,Int)] =
        rdd.reduceByKey((x,y) => x + y)
```

Spark operations on RDD[(K,V)]

reduceByKey
aggregateByKey
groupByKey
Values
mapValues
join

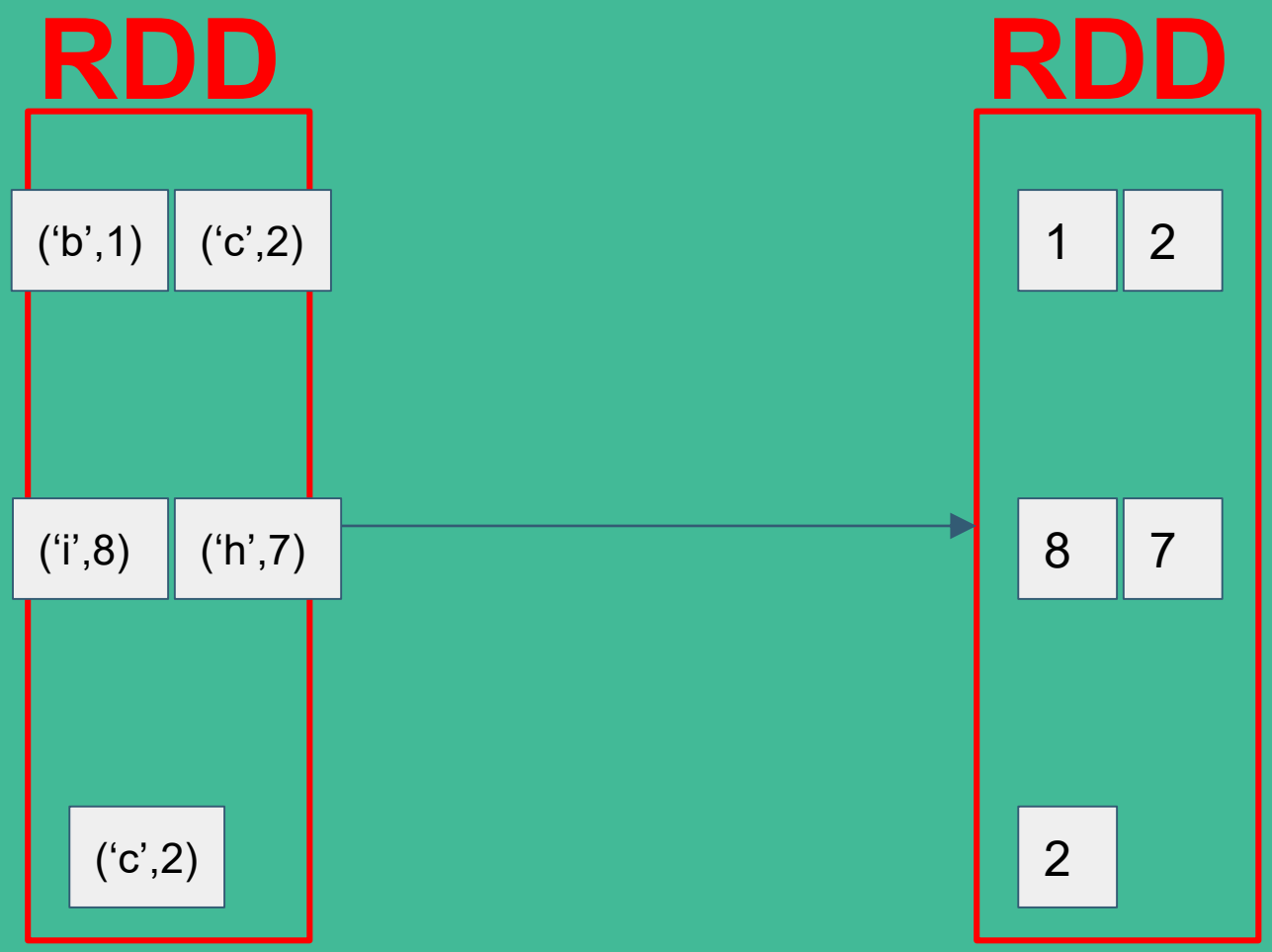
rdd.groupBy(f)

===

rdd.keyBy(f).groupByKey()

Spark operations on RDD[(K,V)]

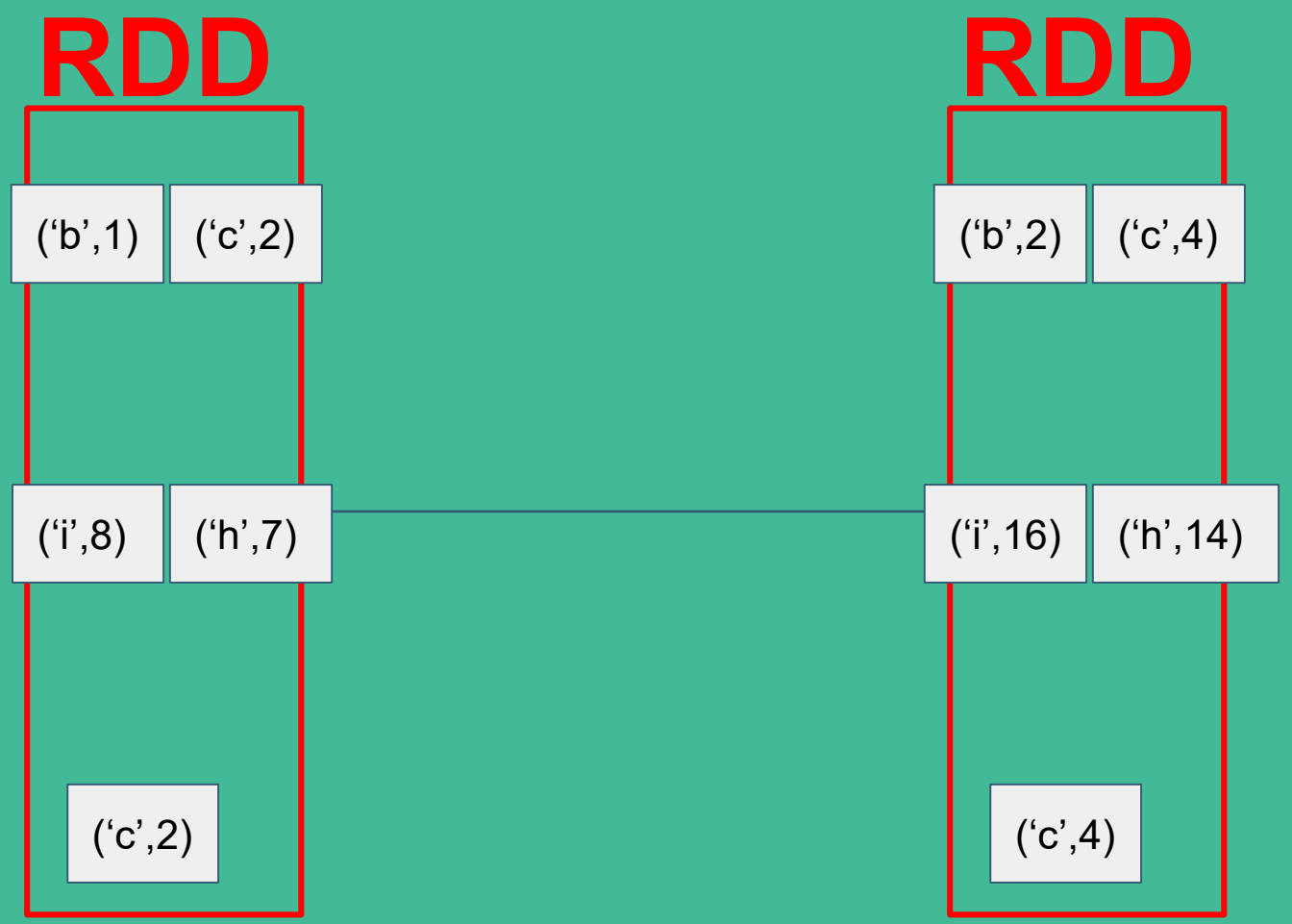
reduceByKey
aggregateByKey
groupByKey
Values
mapValues
join



var rddValues : RDD[(Char,Int)] = rdd.values

Spark operations on RDD[(K,V)]

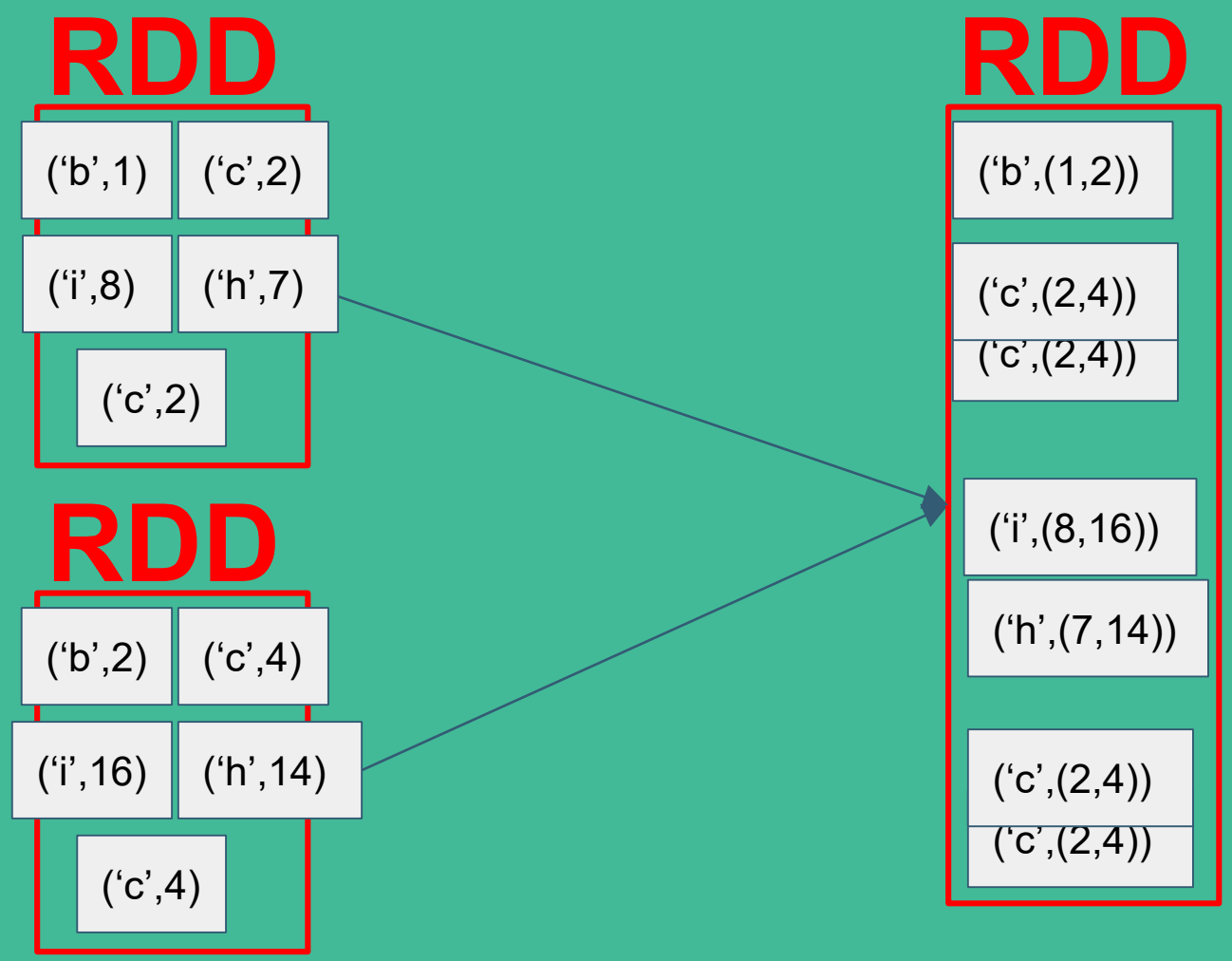
reduceByKey
aggregateByKey
groupByKey
Values
mapValues
join




```
var rddMapValues : RDD[(Char,Int)] =
rdd.mapValues(x => x * 2)
```

Spark operations on RDD[(K,V)]

reduceByKey
aggregateByKey
groupByKey
Values
mapValues
join



var rddJoined : RDD[(Char,(Int,Int)] =
rdd.join(rddMapValues)