SWE404/DMT413
BIG DATA ANALYTICS

Lecture 6: Spark II

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

MORE ON RESILIENT DISTRIBUTED DATASETS (RDD)

Basic RDDs: Transformations

Functions	Description
`map(func)`	Apply a function to each element in the RDD and return an RDD of the result
`flatMap(func)`	Similar to map, but each input item can be mapped to 0 or more output items
`filter(func)`	Return an RDD consisting of only elements that pass the condition passed to `filter()`
`distinct()`	Remove duplicates
`union(RDD)`	Produce an RDD containing elements from both RDDs
`intersection(RDD)`	RDD containing only elements found in both RDDs
`cartesian(RDD)`	Cartesian product with the other RDD
`sample(withReplacement, fraction, seed)`	Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed
`glom()`	Return an RDD created by coalescing all elements within each partition into a list
`coalesce(numPartitions)`	Decrease the number of partitions in the RDD to numPartitions.
`repartition(numPartitions)`	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them.

Check official document for more: https://spark.apache.org/docs/latest/api/python/pyspark.html

Basic RDDs: Actions

Functions	Description
`count()`	Gets the number of data elements in an RDD
`countByValue()`	Number of times each element occurs in the RDD
`collect()`	Gets all data elements in the RDD as an array
`reduce()`	Aggregates data elements into the RDD
`take(n)`	Used to fetch the first n elements of the RDD
`top(num)`	Return the top `num` elements the RDD
`takeOrdered(num)`	Return `num` elements based on provided ordering
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random
`aggregate(zeroValue, seqOp, combOp)`	Aggregate the elements of each partition, and then the results for all the partitions
`foreach(func)`	Apply the provided function to each element of the RDD

Check official document for more: https://spark.apache.org/docs/latest/api/python/pyspark.html

Example

- `map()` transforms RDD lines into RDD line_length.
- `first()` and `reduce()` are actions to draw results from the RDD line_length.

```
lines = sc.textFile('README.md')
lines
```

```
README.md MapPartitionsRDD[50] at textFile at NativeMethodAccessorImpl.java:0
```

```
lines.count()
```

```
104
```

```
lines.first()
```

```
'# Apache Spark'
```

```
line_length = lines.map(lambda x: len(x))
line_length
```

```
PythonRDD[53] at RDD at PythonRDD.scala:53
```

```
line_length.count()
```

```
104
```

```
line_length.first()
```

```
14
```

```
total_length = line_length.reduce(lambda a, b: a + b)
total_length
```

```
3652
```

collect()

- `collect()` is an action that returns a list that contains all of the elements in this RDD.
  - **Note:** This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
line_length.collect()
```

```
[14,
 0,
 78,
 75,
 73,
 74,
 56,
 42,
 0,
 26,
 0,
 0,
 23,
 0,
 68,
 77,
 56,
 0,
 17,
```

filter()

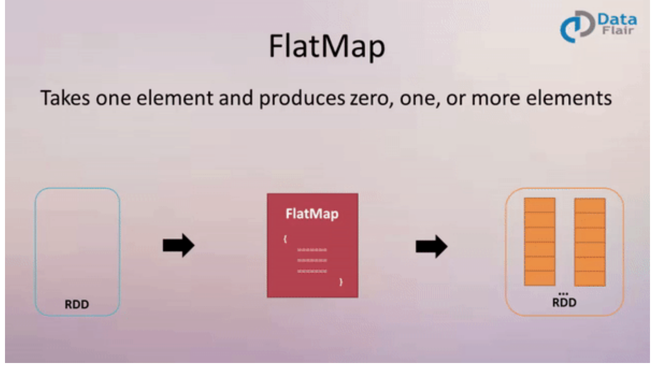
- Filter is just like WHERE condition in SQL query.

```
# filter
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.filter(lambda x: x % 2 == 0).collect()
```

```
[2, 4]
```

map() vs flatMap()

- map() will return a sequence of the same length as the original data.
- flatMap() will return a sequence whose length equals to the sum of the lengths of all sub-sequance returned by map.




```python
# map and flatmap
rdd = sc.parallelize([2, 3, 4])
print(rdd.map(lambda x: x + 1).collect())
print(rdd.flatMap(lambda x: range(1, x)).collect())
```

```
[3, 4, 5]
[1, 1, 2, 1, 2, 3]
```

```python
text=["a b c", "d e", "f g h"]
rdd = sc.parallelize(text)
print(rdd.map(lambda x:x.split(" ")).collect())
print(rdd.flatMap(lambda x:x.split(" ")).collect())
```

```
[['a', 'b', 'c'], ['d', 'e'], ['f', 'g', 'h']]
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```


Image source: https://data-flair.training/blogs/apache-spark-map-vs-flatmap/

Transform Operator Examples

Note:

- Union does not return distinct set.
- Sample does not return the same number of items for each run. The argument 0.1 is the expected fraction.

```
rdd1 = sc.parallelize([1, 10, 2, 3, 4, 5])
rdd2 = sc.parallelize([1, 6, 2, 3, 7, 8])
print(rdd1.union(rdd2).collect())
print(rdd1.intersection(rdd2).collect())
```

```
[1, 10, 2, 3, 4, 5, 1, 6, 2, 3, 7, 8]
[1, 2, 3]
```

```
# cartesian
rdd1 = sc.parallelize([1, 2])
rdd2 = sc.parallelize([3, 4])
rdd1.cartesian(rdd2).collect()
```

```
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

```
rdd = sc.parallelize(range(100))
print(rdd.sample(False, 0.1).collect())
print(rdd.sample(False, 0.1).collect())
print(rdd.sample(False, 0.1).collect())
```

```
[12, 24, 37, 42, 43, 48, 58, 68, 74, 76, 83, 87]
[7, 12, 15, 23, 31, 40, 46, 51, 54, 67, 70, 76, 98]
[8, 13, 23, 24, 56, 64, 75, 77, 78, 85, 96]
```

```
rdd = sc.parallelize([1, 1, 2, 3])
rdd.distinct().collect()
```

```
[1, 2, 3]
```

Action Operator Examples

```python
# reduce
from operator import add
print(sc.parallelize([1, 2, 3, 4, 5]).reduce(add))
print(sc.parallelize((2 for _ in range(10))).map(lambda x: 1).reduce(add))
```

```
15
10
```

```python
# take
print(sc.parallelize([1, 2, 3, 4, 5]).take(3))
print(sc.parallelize(range(100)).filter(lambda x: x > 90).take(3))
```

```
[1, 2, 3]
[91, 92, 93]
```

```python
# takeOrdered and top
rdd = sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7])
print(rdd.takeOrdered(6))
print(rdd.top(6))
```

```
[1, 2, 3, 4, 5, 6]
[10, 9, 7, 6, 5, 4]
```

```python
# takeSample
rdd = sc.parallelize(range(0, 10))
print(rdd.takeSample(True, 10))
print(rdd.takeSample(False, 10))
```

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

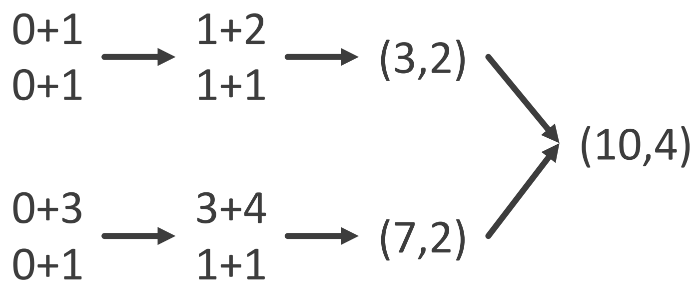
aggregate()

- `aggregate(zeroValue, seqOp, combOp)`
  - zeroValue: The initialization value, for your result, in the desired format.
  - seqOp: The operation you want to apply to RDD records. Runs once for every record in a partition.
  - combOp: Defines how the resulted objects (one for every partition), gets combined.

(0, 0)

```python
# aggregate
seqOp = (lambda x, y: (x[0] + y, x[1] + 1))
combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
rdd = sc.parallelize([1, 2, 3, 4], 2)
print(rdd.glom().collect())
print(rdd.aggregate((0, 0), seqOp, combOp))
```

```
[[1, 2], [3, 4]]
(10, 4)
```



RDD Persistence/Caching

- In Spark, we can use some RDDs multiple times.
- We repeat the same process of **RDD evaluation** each time it required into action.
- This task can be time and memory consuming, especially for iterative algorithms that look at data multiple times.
- To solve the problem of repeated computation the technique of persistence came into the picture.

RDD Persistence/Caching

- Save the intermediate result so that we can use it further if required.
  - When we persist RDD, each node stores any partition of it in memory and makes it reusable for future use.
  - It reduces the computation overhead.
- We can make persisted RDD through `cache()` and `persist()` methods.
- The difference:
  - Using `cache()` the default storage level is **MEMORY_ONLY**.
  - Using `persist()` we can use various storage levels.

Storage levels of Persisted RDDs

By `persist()` we can use various storage levels to Store Persisted RDDs.

```
from pyspark import StorageLevel
rdd1 = sc.parallelize([1, 2, 3, 4, 5])
rdd1.persist(StorageLevel.MEMORY_AND_DISK)
rdd1.is_cached
```

```
True
```

RDD Storage Level	Store Format	When size of RDD is Greater Than Memory	Memory Usage	CPU Time
MEMORY_ONLY (default)	Deserialized Java object	Recompute	Very high	Low
MEMORY_AND_DISK		Store on the disk	High	Medium
MEMORY_ONLY_SER	Serialized Java object (one-byte array per partition)	Recompute	Low	High
MEMORY_AND_DISK_SER		Store on the disk	Low	High
DISK_ONLY	-	-	Very low	Very high

Paired RDDs

- Paired RDD = an RDD of key / value pairs.

```
lines = sc.textFile('README.md')
pairs = lines.map(lambda x: (x.split(" ")[0], x))
pairs.collect()
```

```
[('#', '# Apache Spark'),
 ('', ''),
 ('Spark',
  'Spark is a fast and general cluster computing system for Big Data. It provides'),
 ('high-level',
  'high-level APIs in Scala, Java, Python, and R, and an optimized engine that'),
 ('supports',
  'supports general computation graphs for data analysis. It also supports a'),
 ('rich',
  'rich set of higher-level tools including Spark SQL for SQL and DataFrames,'),
 ('MLlib', 'MLlib for machine learning, GraphX for graph processing,'),
 ('and', 'and Spark Streaming for stream processing.'),
 ('', ''),
 ('<http://spark.apache.org/>', '<http://spark.apache.org/>'),
 ('', ''),
 ('', ''),
 ('##', '## Online Documentation'),
 ('', ''),
```

Use the first words of RDD lines as the keys in the pair RDD pairs

Transformations on Single Paired RDDs

Method Name	Purpose
`reduceByKey(func)`	Combine values with the same key
`groupByKey()`	Group values with the same key
`combineByKey(createCombiner, mergeValue, mergeCombiners)`	Combine values with the same key using a different result type
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key
`flatMapValues(func)`	Pass each value in the key-value pair RDD through a flatMap function without changing the keys
`keys()`	Return an RDD of just the keys.
`values()`	Return an RDD of just the values.
`sortByKey()`	Return an RDD sorted by the key.

Official document: https://spark.apache.org/docs/latest/api/python/pyspark.html

keys(), values() and sortByKey()

```
# keys and values
rdd = sc.parallelize([(2, 'b'), (1, 'a'), (3, 'c')])
print(rdd.keys().collect())
print(rdd.values().collect())
```

```
[2, 1, 3]
['b', 'a', 'c']
```

```
# sortByKey
rdd = [('Mary', 1), ('had', 2), ('a', 3), ('Little', 4), ('lamb', 5)]
print(sc.parallelize(rdd).sortByKey().collect())
print(sc.parallelize(rdd).sortByKey(keyfunc=lambda k: k.lower()).collect())
```

Customized key map function for sorting

```
[('Little', 4), ('Mary', 1), ('a', 3), ('had', 2), ('lamb', 5)]
[('a', 3), ('had', 2), ('lamb', 5), ('Little', 4), ('Mary', 1)]
```

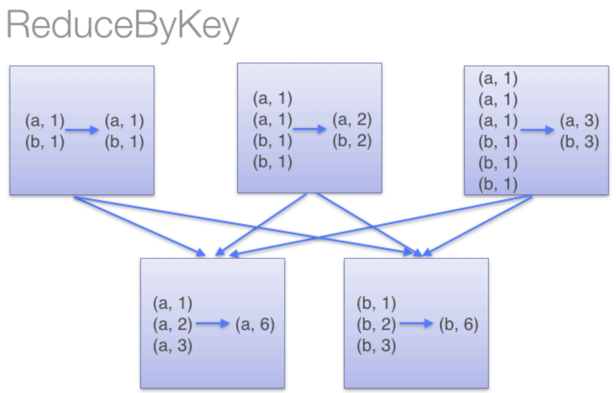
mapValues() and flatMapValues()

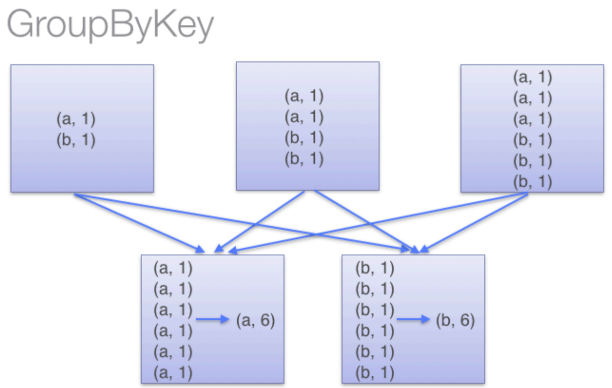
```
# mapValues and flatMapValues
x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b", ["grapes"])])
print(x.mapValues(lambda x: len(x)).collect())
print(x.flatMapValues(lambda x: x).collect())
```

```
[('a', 3), ('b', 1)]
[('a', 'apple'), ('a', 'banana'), ('a', 'lemon'), ('b', 'grapes')]
```

groupByKey() and reduceByKey()

- `reduceByKey` provide much better performance than `groupByKey` for aggregation (such as a sum or average).
  - `reduceByKey` perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce.
- `groupByKey` is usually used for non-aggregation operations like returning a list.
  - `groupByKey` is selected as the worst Spark operation, why?






```python
# groupByKey
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
print(rdd.groupByKey().mapValues(len).collect())
print(rdd.groupByKey().mapValues(list).collect())
```

```
[('a', 2), ('b', 1)]
[('a', [1, 1]), ('b', [1])]
```

```python
# reduceByKey
from operator import add
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
rdd.reduceByKey(add).collect()
```

```
[('a', 2), ('b', 1)]
```

combineByKey()

- `combineByKey(createCombiner, mergeValue, mergeCombiners)`
- Generic function to combine the elements for each key using a custom set of aggregation functions.
  - Turns an RDD[(K, V)] into a result of type RDD[(K, C)], for a "combined type" C.
- Users provide three functions:
  - *createCombiner*, which turns a V into a C (e.g., creates a one-element list, the combined type)
  - *mergeValue*, to merge a V into a C (e.g., adds it to the end of a list)
  - *mergeCombiners*, to combine two C's into a single one (e.g., merges the lists)

```python
# combineByKey
x = sc.parallelize([("a", 1), ("b", 1), ("a", 2)])
def to_list(a):
    return [a]

def append(a, b):
    a.append(b)
    return a

def extend(a, b):
    a.extend(b)
    return a

x.combineByKey(to_list, append, extend).collect()
```

```
[('a', [1, 2]), ('b', [1])]
```

Transformations on Two Paired RDDs

Method Name	Purpose
`subtractByKey(other)`	Remove elements with a key present in the other RDD.
`join(other)`	Perform an inner join between two RDDs.
`leftOuterJoin(other)`	Perform a join between two RDDs where the key must be present in the first RDD
`rightOuterJoin(other)`	Perform a join between two RDDs where the key must be present in the other RDD
`fullOuterJoin(other)`	Perform a join between two RDDs where the key must be present in the other RDD
`cogroup(other)`	Group data from both RDDs sharing the same key

Official document: https://spark.apache.org/docs/latest/api/python/pyspark.html

subtractByKey()

```
# subtractByKey
x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 3), ("c", 3)])
y = sc.parallelize([("a", 3), ("c", 4)])
print(x.subtract(y).collect())
print(x.subtractByKey(y).collect())
```

```
[('b', 5), ('b', 4), ('a', 1), ('c', 3)]
[('b', 4), ('b', 5)]
```

join()

- Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in self and (k, v2) is in other.

```
# join
x = sc.parallelize([("a", 1), ("b", 4)])
y = sc.parallelize([("a", 2), ("a", 3), ("c", 5)])
print(x.join(y).collect())
print(x.leftOuterJoin(y).collect())
print(x.rightOuterJoin(y).collect())
print(x.fullOuterJoin(y).collect())
```

```
[('a', (1, 2)), ('a', (1, 3))]
[('b', (4, None)), ('a', (1, 2)), ('a', (1, 3))]
[('c', (None, 5)), ('a', (1, 2)), ('a', (1, 3))]
[('b', (4, None)), ('c', (None, 5)), ('a', (1, 2)), ('a', (1, 3))]
```

cogroup()

- `cogroup` does full join and returns merged iterable values.

```
# cogroup
x = sc.parallelize([("a", 1), ("b", 4)])
y = sc.parallelize([("a", 2), ("a", 3), ("c", 5)])
```

```
cogroup_rdd = x.cogroup(y)
cogroup_rdd.collect()
```

```
[('b',
  (<pyspark.resultiterable.ResultIterable at 0x11b540d30>,
   <pyspark.resultiterable.ResultIterable at 0x11b5404a8>)),
 ('c',
  (<pyspark.resultiterable.ResultIterable at 0x11b540ba8>,
   <pyspark.resultiterable.ResultIterable at 0x11b540710>)),
 ('a',
  (<pyspark.resultiterable.ResultIterable at 0x11b540390>,
   <pyspark.resultiterable.ResultIterable at 0x11b5404e0>))]
```

```
[(x, tuple(map(list, y))) for x, y in list(cogroup_rdd.collect())]
```

```
[('b', ([4], [])), ('c', ([], [5])), ('a', ([1], [2, 3]))]
```

Actions on Pair RDDs

Method Name	Purpose
`countByKey()`	Count the number of elements for each key
`collectAsMap()`	Collect the result as a map to provide easy lookup
`lookup(key)`	Return all values associated with the provided key

Official document: https://spark.apache.org/docs/latest/api/python/pyspark.html

Examples of Actions on Pair RDDs

```
# countByKey
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 3)])
print(rdd.countByKey())
```

```
defaultdict(<class 'int'>, {'a': 2, 'b': 1})
```

```
# countAsMap
rdd = sc.parallelize([("a", 1), ("b", 2), ("c", 3)])
print(rdd.collectAsMap())
```

```
{'a': 1, 'b': 2, 'c': 3}
```

```
# lookup
rdd = sc.parallelize([("a", 1), ("b", 2), ("b", 3)])
print(rdd.lookup("a"))
print(rdd.lookup("b"))
```

```
[1]
[2, 3]
```

RDD Transformation Types

Narrow transformation :

- Single partition of the parent RDD is needed for computation.
- Input and output stay in the same partition.
- No data movement is needed.

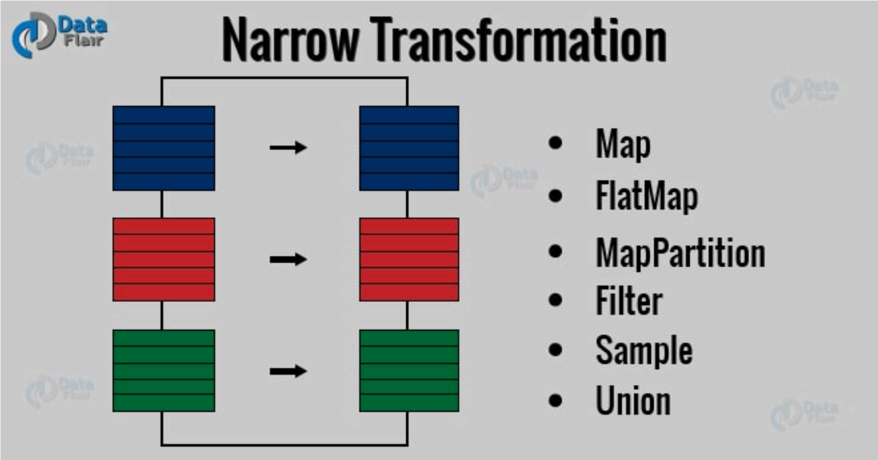


Image source: https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

RDD Transformation Types

Wide transformation :

- Multiple partitions of the parent RDD are needed for computation.
- Data shuffle is needed before processing.

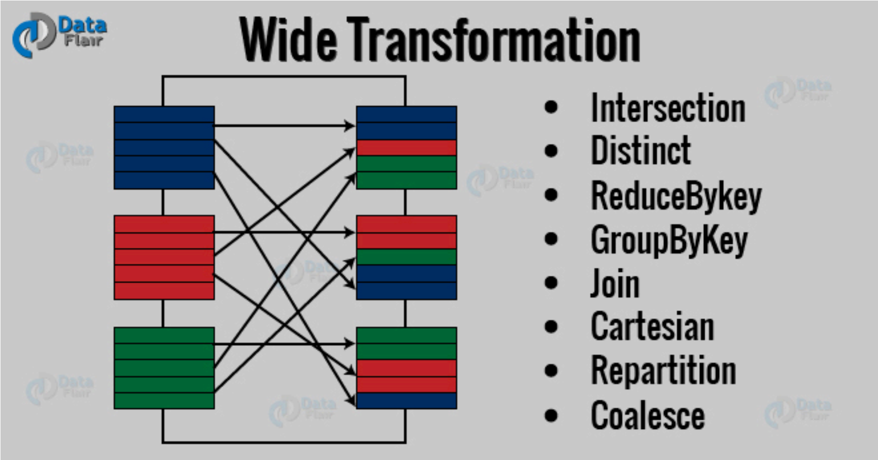
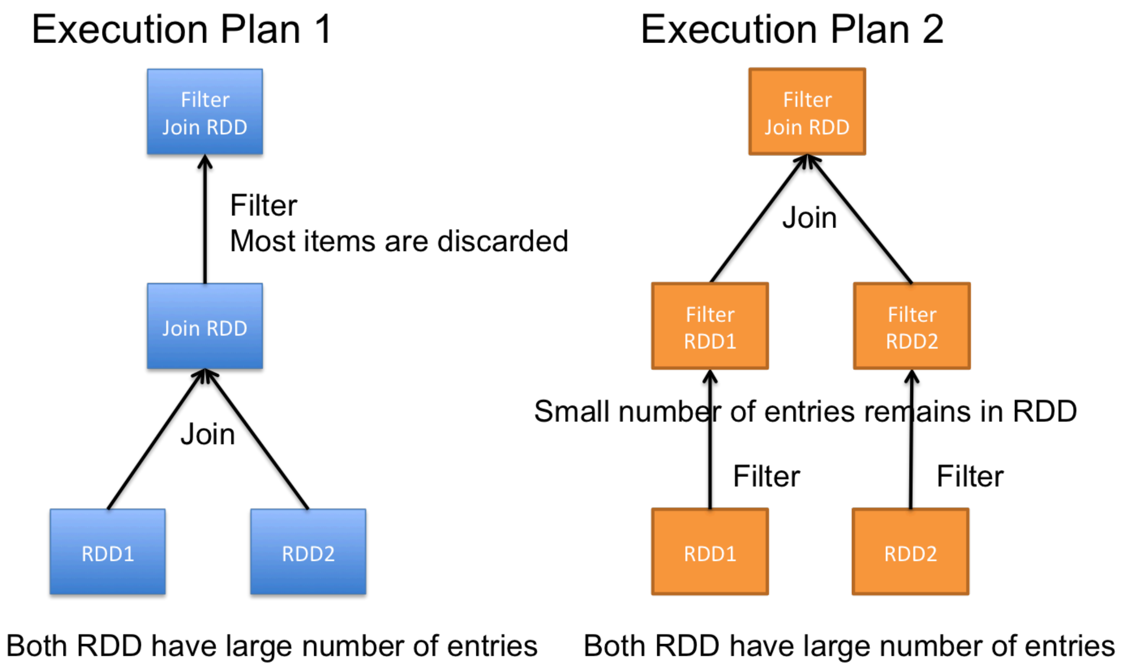


Image source: https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

Reduce the Amount of Data Shuffling

- Ideally a Spark program should avoid shuffles (wide transformations).
- In some cases, transformation can be *re-ordered* to reduce the amount of data shuffling.

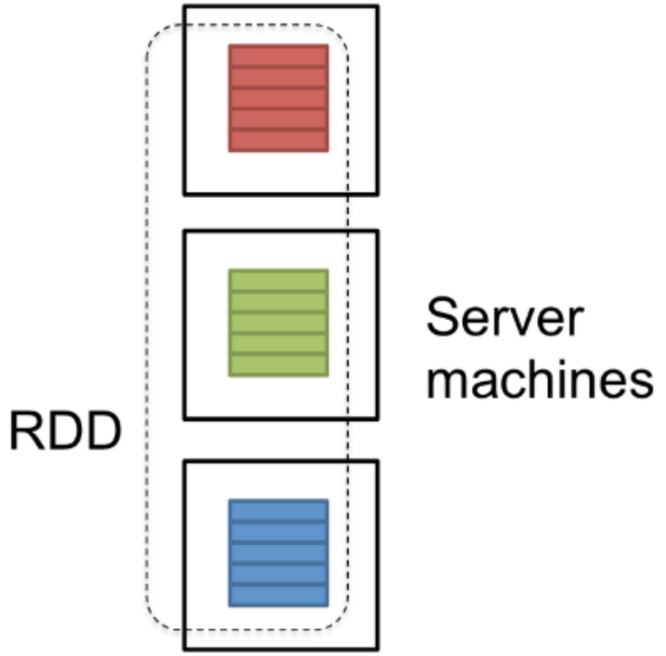
Transformation: ***Select a, b from RDD1 join RDD2 where a > 10 and b > 20***



An example of a JOIN between two huge RDDs followed by a filtering.

Partitions

- (key,value) pairs in the same partition are guaranteed to be in the same machine.
- Each node may contain more than one partition.
- Number of partitions determines parallelism.
- Location of partitions determines data locality.



glom(), coalesce() and repartition()

- `repartition` can increase or decrease the level of parallelism in this RDD. Internally, this uses a shuffle to redistribute data.
- If you are decreasing the number of partitions in this RDD, consider using `coalesce`, which can avoid performing a shuffle.
  - `coalesce` can also shuffle by setting the second argument as True, while its default value is False.

```
# glom
rdd = sc.parallelize([1, 2, 3, 4], 2)
rdd.glom().collect()
```

```
[[1, 2], [3, 4]]
```

```
# coalesce
print(sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect())
print(sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect())
```

```
[[1], [2, 3], [4, 5]]
[[1, 2, 3, 4, 5]]
```

```
# repartition
rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
print(rdd.glom().collect())
print(rdd.repartition(2).glom().collect())
print(rdd.repartition(10).glom().collect())
```

```
[[1], [2, 3], [4, 5], [6, 7]]
[[1, 4, 5, 6, 7], [2, 3]]
[[], [1], [4, 5, 6, 7], [2, 3], [], [], [], [], [], []]
```

partitionBy()

- `partitionBy()` can only be used for paird RDDs.
- `partitionBy()` is most importantly used for making shuffling functions more efficient, such as `reduceByKey()`, `join()`, `cogroup()` etc..
- It is only beneficial in cases where a RDD is used for multiple times, so it is usually followed by `persist()`.

```python
pairs = sc.parallelize([1, 2, 3, 4, 2, 4, 1, 5, 6, 7, 7, 5, 5, 6, 4])
print(pairs.partitionBy(3).glom().collect())
```

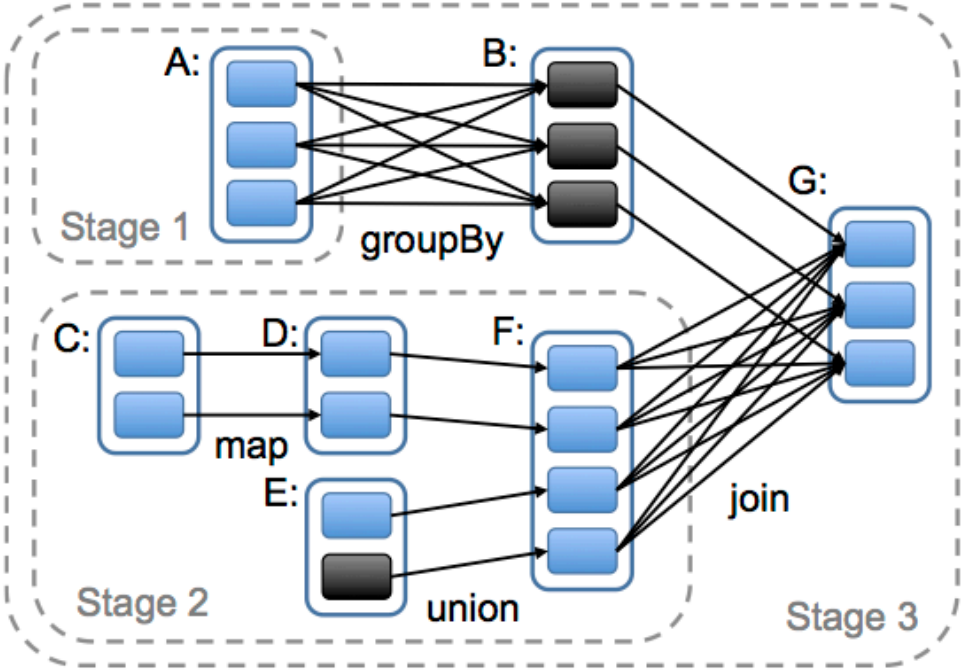
```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
<ipython-input-105-01c7bce86039> in <module>
      1 pairs = sc.parallelize([1, 2, 3, 4, 2, 4, 1, 5, 6, 7, 7, 5, 5, 6, 4])
----> 2 print(pairs.partitionBy(3).glom().collect())

/usr/local/Cellar/apache-spark/2.4.5/libexec/python/pyspark/rdd.py in collect(self)
    814         """
    815         with SCCallSiteSync(self.context) as css:
--> 816             sock_info = self.ctx._jvm.PythonRDD.collectAndServe(self._jrdd.rdd())
    817         return list(_load_from_socket(sock_info, self._jrdd_deserializer))
    818

/usr/local/Cellar/apache-spark/2.4.5/libexec/python/lib/py4j-0.10.7-src.zip/py4j/java_gateway.py in __call__(self, *a
rgs)
   1255         answer = self.gateway_client.send_command(command)
   1256         return_value = get_return_value(
-> 1257             answer, self.gateway_client, self.target_id, self.name)
   1258
```

```python
pairs = sc.parallelize([1, 2, 3, 4, 2, 4, 1, 5, 6, 7, 7, 5, 5, 6, 4]).map(lambda x: (x, x))
print(pairs.partitionBy(3).glom().collect())
print(pairs.repartition(3).glom().collect())
```

```
[[(3, 3), (6, 6), (6, 6)], [(1, 1), (4, 4), (4, 4), (1, 1), (7, 7), (7, 7), (4, 4)], [(2, 2), (2, 2), (5, 5), (5, 5),
(5, 5)]]
[[], [(1, 1), (4, 4), (2, 2), (7, 7), (7, 7), (5, 5), (5, 5), (6, 6), (4, 4)], [(2, 2), (3, 3), (4, 4), (1, 1), (5,
5), (6, 6)]]
```

Stage

- A stage is a step in a physical execution plan.
- Each job which gets divided into smaller sets of tasks is a stage.
  - Narrow transformations are grouped into stages.
- it is just like the map and reduce stages in MapReduce.
  - A stage is scheduled once all of the stages it is dependent on are available.



Black boxes are partitions that are already in memory (use `persist`)

Conclusion

After this lecture, you should know:

- Some commonly used RDD transformations and actions.
- What is RDD persistence?
- What is paired RDD?
- What are narrow and wide transformation?
- What are RDD partitions?

Thank you!

- Any question?
- Don't hesitate to send email to me for asking questions and discussion. ☺

Acknowledgement: Thankfully acknowledge slide contents shared by Dr. Ye Luo