

Apache Spark advanced

Использование Apache Spark

RDD, Сериализация

- “Из коробки” используется встроенная java сериализация
- Стандартно поддерживается библиотека сериализации Kryo, которая не требует от программиста дополнительного кода (в отличие от стандартного для hadoop api)

Инициализация

- Для управления настройками spark используется класс

`org.apache.spark.SparkConf`

- Взаимодействие с системой spark осуществляется через экземпляр класса

`org.apache.spark.api.java.JavaSparkContext`

- Пример

```
SparkConf conf = new SparkConf().setAppName("sample");  
JavaSparkContext sc = new JavaSparkContext(conf);
```

Загрузка данных в RDD

- Для разработки можно применять метод `parallelize`

```
JavaRDD<String> lines =  
    sc.parallelize(Arrays.asList("pandas", "i like pandas"));
```

- Загрузка файла из HDFS

```
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

- Загрузка данных из Hadoop InputFormat

```
JavaPairRDD<LongWritable, Text> data =  
    sc.hadoopFile("war-and-peace-1.txt",  
                  TextInputFormat.class,  
                  LongWritable.class,  
                  Text.class);
```

Функции преобразования

- Преобразование RDD производится с помощью лямбда-функций scala (и java8) или обычных функций для java7

- Пример на Scala

```
val errorsRDD=inputRDD.filter(line=>line.contains("error"))
```

- Пример java 8

```
JavaRDD<String> errorsRDD=inputRDD.filter(  
    x -> x.contains("error")  
);
```

Операции над RDD

Преобразования

| Function name | Purpose | Example | Result |
|--------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|------------------------------------|
| <code>map()</code> | Apply a function to each element in the RDD and return an RDD of the result. | <code>rdd.map(x => x + 1)</code> | <code>{2, 3, 4, 4}</code> |
| <code>flatMap()</code> | Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words. | <code>rdd.flatMap(x => x.to(3))</code> | <code>{1, 2, 3, 2, 3, 3, 3}</code> |
| <code>filter()</code> | Return an RDD consisting of only elements that pass the condition passed to <code>filter()</code> . | <code>rdd.filter(x => x != 1)</code> | <code>{2, 3, 3}</code> |
| <code>distinct()</code> | Remove duplicates. | <code>rdd.distinct()</code> | <code>{1, 2, 3}</code> |
| <code>sample(withReplacement, fraction, [seed])</code> | Sample an RDD, with or without replacement. | <code>rdd.sample(false, 0.5)</code> | Nondeterministic |

Операции над двумя RDD

| Function name | Purpose | Example | Result |
|-----------------------------|--------------------------------------------------------------|--------------------------------------|-----------------------------|
| <code>union()</code> | Produce an RDD containing elements from both RDDs. | <code>rdd.union(other)</code> | {1, 2, 3, 3, 4, 5} |
| <code>intersection()</code> | RDD containing only elements found in both RDDs. | <code>rdd.intersection(other)</code> | {3} |
| <code>subtract()</code> | Remove the contents of one RDD (e.g., remove training data). | <code>rdd.subtract(other)</code> | {1, 2} |
| <code>cartesian()</code> | Cartesian product with the other RDD. | <code>rdd.cartesian(other)</code> | {(1, 3), (1, 4), ... (3,5)} |

Actions

- Загрузка из HDFS

```
JavaRDD<String> distFile = sc.textFile("war-and-peace-1.txt");
```

- Сохранить в HDFS

```
res.saveAsTextFile("result");
```

- Скачать данные

```
res.collect()
```

- Количество записей

```
res.count()
```


Работа с парами KeyValue

- Пары KeyValue(Tuple2) применяются для операций работающих с “группами” значений:

reduceByKey – применяет функцию ко всем значениям имеющим одинаковый ключ во всем RDD

Join – генерирует пары значения для заданного ключа

...

- Создать RDD вида KeyValue

```
JavaPairRDD<String, Long> wordsWithCount = splitted.mapToPair(  
    s -> new Tuple2<>(s, 1)  
);
```

KeyValue transformations

Table 4-1. Transformations on one pair RDD (example: $\{(1, 2), (3, 4), (3, 6)\}$)

| Function name | Purpose | Example | Result |
|-----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|-------------------------------------------------------|-----------------------------|
| <code>reduceByKey(func)</code> | Combine values with the same key. | <code>rdd.reduceByKey((x, y) => x + y)</code> | $\{(1, 2), (3, 10)\}$ |
| <code>groupByKey()</code> | Group values with the same key. | <code>rdd.groupByKey()</code> | $\{(1, [2]), (3, [4, 6])\}$ |
| <code>combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)</code> | Combine values with the same key using a different result type. | See Examples 4-12 through 4-14. | |

KeyValue transformations

| | | | |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------|---------------------------------------------------------------|
| <code>mapValues(func)</code> | Apply a function to each value of a pair RDD without changing the key. | <code>rdd.mapValues(x => x+1)</code> | <code>{(1, 3), (3, 5), (3, 7)}</code> |
| <code>flatMapValues(func)</code> | Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization. | <code>rdd.flatMapValues(x => (x to 5))</code> | <code>{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}</code> |
| <code>keys()</code> | Return an RDD of just the keys. | <code>rdd.keys()</code> | <code>{1, 3, 3}</code> |

KeyValue transformations

`values()`

Return an RDD of just
the values.

`rdd.values()`

{2, 4,
6}

`sortByKey()`

Return an RDD sorted
by the key.

`rdd.sortByKey()`

{(1,
2), (3,
4), (3,
6)}

Пример работы с функцией combineByKey

```
public class AvgCount implements  
Serializable {
```

```
    private long total;
```

```
    private long counter;
```

```
    public long getTotal() {
```

```
        return total;
```

```
    }
```

```
    public long getCounter() {
```

```
        return counter;
```

```
    }
```

```
    public AvgCount(long total, long counter) {
```

```
        this.total = total;
```

```
        this.counter = counter;
```

```
    }
```

```
        public static AvgCount addValue  
            (AvgCount a, long value) {
```

```
            return new AvgCount(
```

```
                a.getTotal() + value,  
                a.getCounter() + 1l );
```

```
        }
```

```
        public static AvgCount add
```

```
            ( AvgCount a, AvgCount b) {
```

```
            return new AvgCount(
```

```
                a.getTotal() + b.getTotal(),  
                a.getCounter() + b.getCounter()
```

```
            );
```

```
        }
```

```
        public float avg() {
```

```
            return total / (double) counter;
```

```
        }
```

```
    }
```

Вызов combine by key

```
JavaPairRDD<String, AvgCount> avgCounts =  
    nums.combineByKey(  
        p -> new AvgCount(p.getValue(), 1),  
        (avgCount, p) → AvgCount.addValue(  
            avgCount,  
            p.getValue()),  
        AvgCount::add  
    );
```

Пример работы с функцией combineByKey

Partition 1

| | |
|--------|---|
| coffee | 1 |
| coffee | 2 |
| panda | 3 |

Partition 2

| | |
|--------|---|
| coffee | 9 |
|--------|---|

Partition 1 trace:

(coffee, 1) -> new key

accumulators[coffee] = createCombiner(1)

(coffee, 2) -> existing key

accumulators[coffee] = merge Value(accumulators[coffee], 2)

(panda, 3) -> new key

accumulators[panda] = createCombiner(3)

Partition 2 trace:

(coffee, 9) -> new key

accumulators[coffee] = createCombiner(9)

Merge Partitions:

mergeCombiners(partition1.accumulators[coffee],
partition2.accumulators[coffee])

```
def createCombiner(value):  
    (value, 1)
```

```
def mergeValue(acc, value):  
    (acc[0] + value, acc[1] + 1)
```

```
def mergeCombiners(acc1, acc2):  
    (acc1[0] + acc2[0], acc1[1] + acc2[1])
```

Работа с двумя RDD KeyValue

Table 4-2. Transformations on two pair RDDs ($rdd = \{(1, 2), (3, 4), (3, 6)\}$ $other = \{(3, 9)\}$)

| Function name | Purpose | Example | Result |
|-----------------------------|---------------------------------------------------------------------------------|----------------------------------------|-------------------------------------------------------------------|
| <code>subtractByKey</code> | Remove elements with a key present in the other RDD. | <code>rdd.subtractByKey(other)</code> | <code>\{(1, 2)\}</code> |
| <code>join</code> | Perform an inner join between two RDDs. | <code>rdd.join(other)</code> | <code>\{(3, (4, 9)), (3, (6, 9))\}</code> |
| <code>rightOuterJoin</code> | Perform a join between two RDDs where the key must be present in the first RDD. | <code>rdd.rightOuterJoin(other)</code> | <code>\{(3,(Some(4),9)), (3,(Some(6),9))\}</code> |
| <code>leftOuterJoin</code> | Perform a join between two RDDs where the key must be present in the other RDD. | <code>rdd.leftOuterJoin(other)</code> | <code>\{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))\}</code> |
| <code>cogroup</code> | Group data from both RDDs sharing the same key. | <code>rdd.cogroup(other)</code> | <code>\{(1,([2],[])), (3, ([4, 6],[9]))\}</code> |

Actions с парами Key/Value

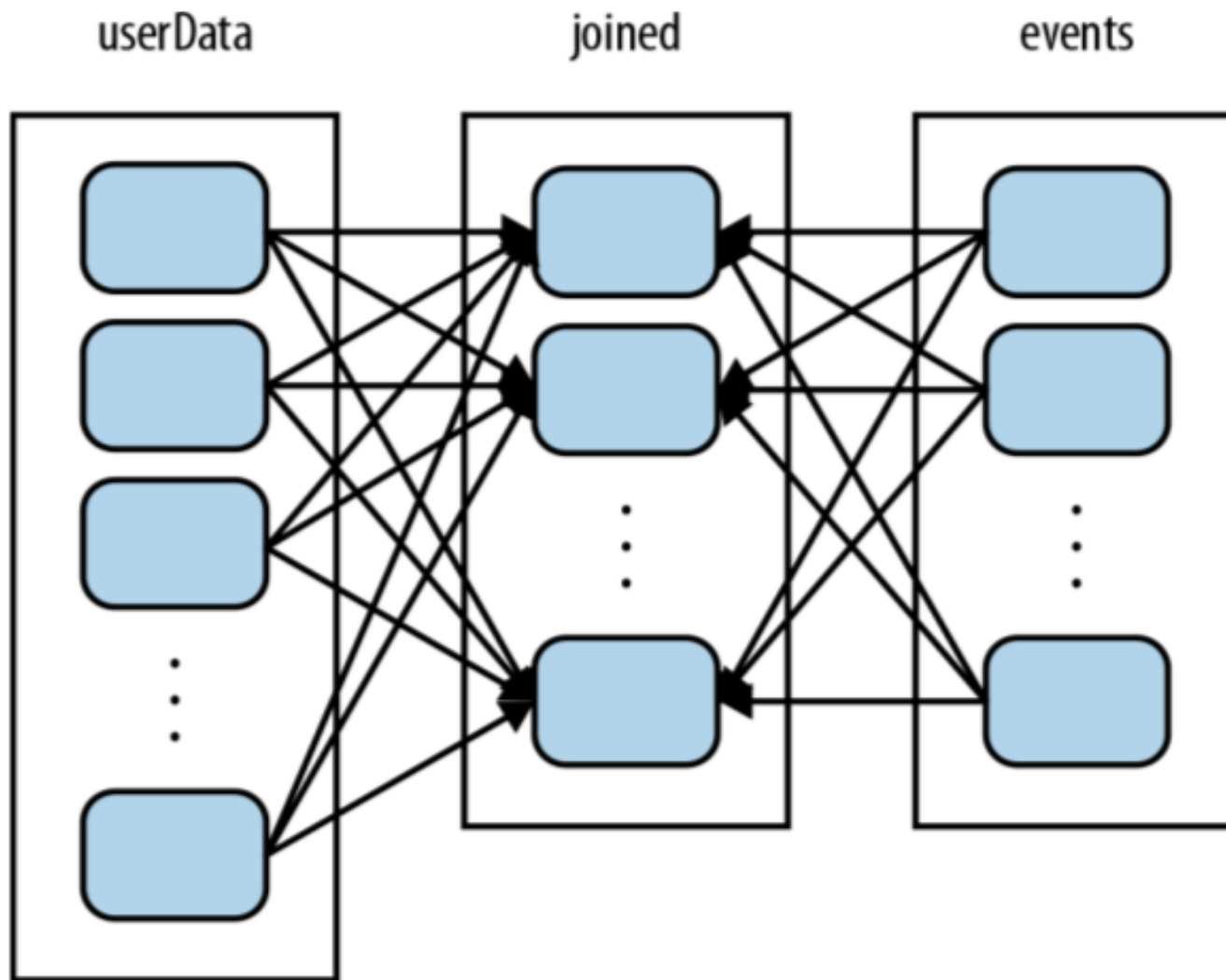
Table 4-3. Actions on pair RDDs (example $\{(1, 2), (3, 4), (3, 6)\}$)

| Function | Description | Example | Result |
|-----------------------------|-----------------------------------------------------|---------------------------------|----------------------------------------|
| <code>countByKey()</code> | Count the number of elements for each key. | <code>rdd.countByKey()</code> | $\{(1, 1), (3, 2)\}$ |
| <code>collectAsMap()</code> | Collect the result as a map to provide easy lookup. | <code>rdd.collectAsMap()</code> | $\text{Map}\{(1, 2), (3, 4), (3, 6)\}$ |
| <code>lookup(key)</code> | Return all values associated with the provided key. | <code>rdd.lookup(3)</code> | $[4, 6]$ |

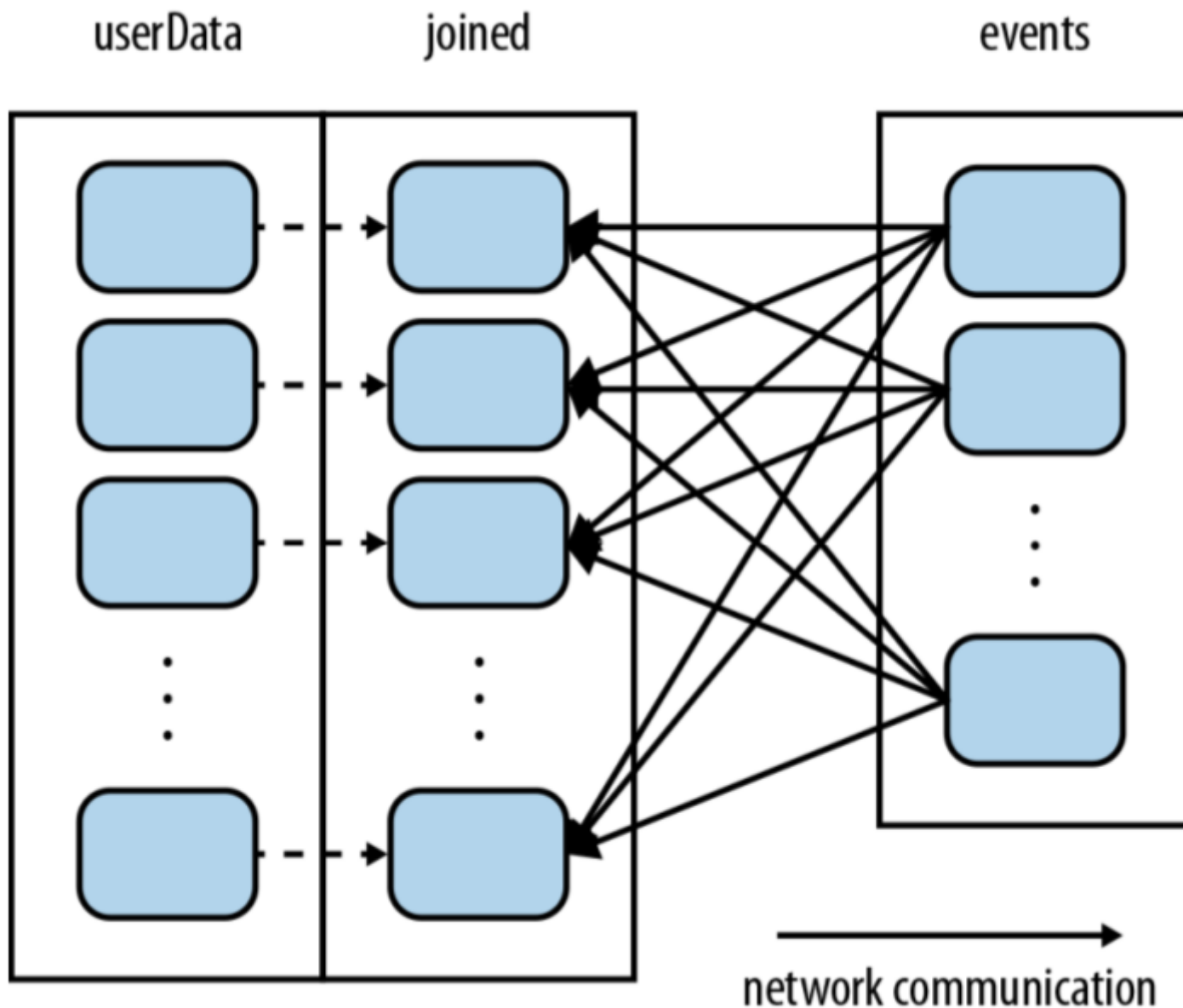
Data partitioning

- По умолчанию нельзя быть уверенным в том как распределены данные по кластеру
- В ряде операций это неэффективно
- `partitionBy(partitioner)` управляет распределением RDD по кластеру
- Операция `repartition()` генерирует новый RDD распределенный по кластеру – предназначена для равномерного распределения данных

Join непартиционированных данных



Join партиционированных данных



Persistence, Caching

- В случае если промежуточные данные требуется сохранить между вычислениями – их можно сохранить в памяти узлов
- `persist(level)` – сохраняет данные на узлах
- `unpersist()` - удаляет сохраненные данные

Аккумуляторы

- Предназначены для вычисления глобальных значений для всего вычисления
- Создаются в глобальном контексте и доступны в лямбда функциях
- Значение аккумулятора недоступно в функциях во время выполнения
- Результирующее значение аккумулятора доступно после завершения вычислений

Использование аккумулятора

....

```
final LongAccumulator total =  
jsc.sc().longAccumulator();
```

....

```
JavaPairRDD<String, Long> collectedWords =  
wordsWithCount.reduceByKey((a, b) -> {  
    total.add(1l);  
    return a + b;  
});  
System.out.println("accumulator value="+total.value());
```

Broadcast variables

- Механизм broadcast служит для рассылки всем узлам одного и того же набора данных
- Набор данных сериализуется, эффективно рассылается на все узлы и доступен внутри функций

```
final Broadcast<Map<String, AirportData>> airportsBroadcasted =  
sc.broadcast(stringAirportDataMap);
```

```
JavaRDD<String> distFile =  
sc.textFile("664600583_T_ONTIME_sample_cutted.csv");
```

```
JavaRDD<ParsedData> splitted = distFile.map(  
    s -> new ParsedData(s, airportsBroadcasted.value())  
);
```


Операции внутри Partitions

- Функции внутри partitions применяются для всех значений RDD внутри одного partition
- Применяются для эффективной работы с внешними ресурсами которые трудоемко создавать (например подключения к базе данных)

Операции внутри Partitions

| Function name | We are called with | We return | Function signature on RDD[T] |
|---------------------------------------|-----------------------------------------------------------------------------|---------------------------------|--------------------------------------------------|
| <code>mapPartitions()</code> | Iterator of the elements in that partition | Iterator of our return elements | <code>f: (Iterator[T]) → Iterator[U]</code> |
| <code>mapPartitionsWithIndex()</code> | Integer of partition number, and Iterator of the elements in that partition | Iterator of our return elements | <code>f: (Int, Iterator[T]) → Iterator[U]</code> |
| <code>foreachPartition()</code> | Iterator of the elements | Nothing | <code>f: (Iterator[T]) → Unit</code> |