Mining large datasets with Apache Spark

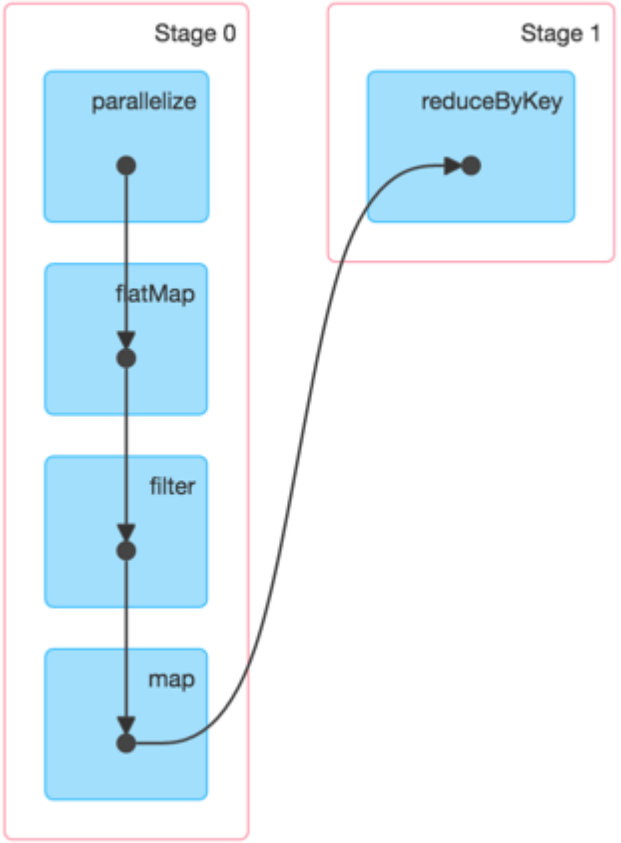
Vyacheslav Baranov, Senior Software Engineer, OK.RU

Spark на низком уровне

DAG (Directed Acyclic Graph) sheduler

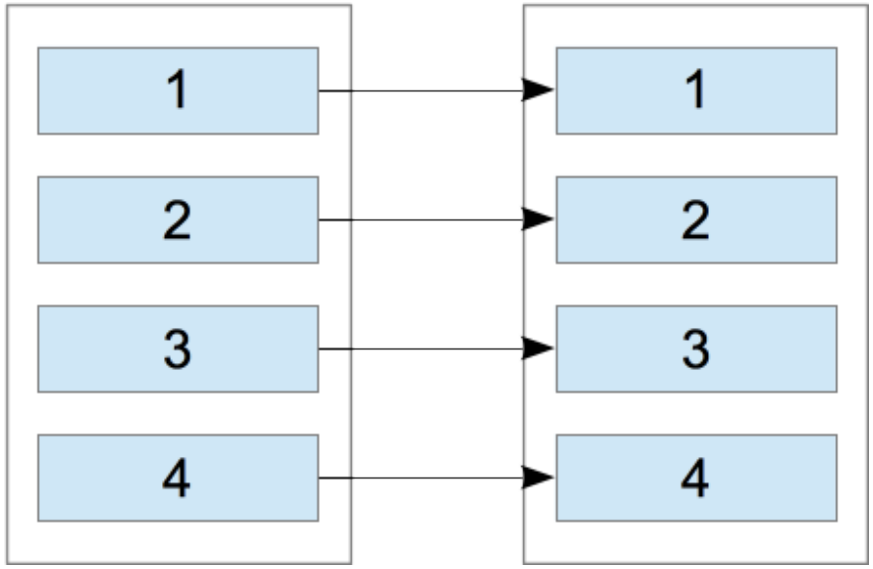
```
val txt = """The quick brown fox
    |jumps over the lazy dog""".stripMargin.toLowerCase

val rdd = sc.parallelize(txt.split("""\n"""), 2)

val res = {
  rdd
    .flatMap(_.split("""\s"""))
    .filter(_.length > 0)
    .map(_ -> 1)
    .reduceByKey(_ + _)
    .collect()
}

res.foreach { case (word, cnt) => println(s"$word -> $cnt")}
```

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at <console>:31	+details	2015/09/06 19:06:28	39 ms	2/2			475.0 B	
0	map at <console>:29	+details	2015/09/06 19:06:27	98 ms	2/2				475.0 B



Narrow dependency (Process)

Для вычисления каждой партиции требуются 1 или несколько родительских партиций, которые вычисляются локально на каждом из Worker'ов

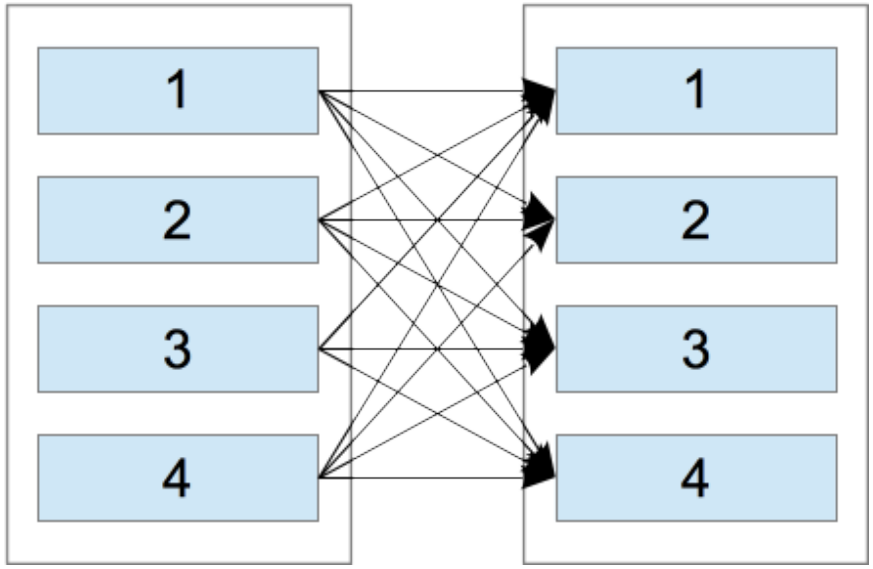


Примеры операций:

- `map`
- `sample`
- `coalesce`
- `foreach`

Shuffle dependency (Shuffle)

Для вычисления каждой партиции требуются **все** партиции родительских RDD, которые вычисляются распределенно и передаются по сети при необходимости.

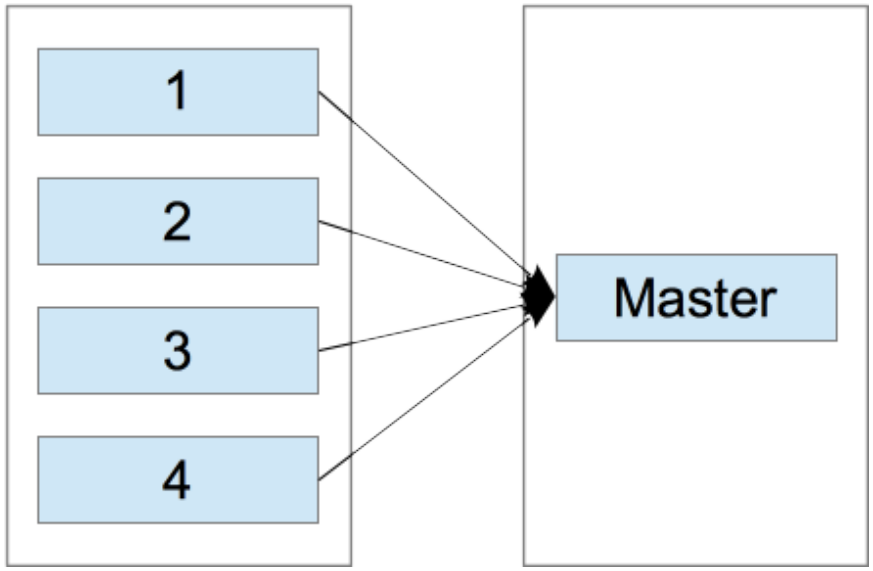


Примеры операций:

- `groupByKey`
- `join`
- `repartition`

Collect

Данные собираются с Worker'ов на Master



Примеры операций:

- `collect`
- `take`

Итого (1)

Все операции над RDD можно рассматривать, как последовательность чередующихся операций Transform & Shuffle (возможно, с последующим Collect), например:

- `reduce` - вычисляются частичные агрегаты для каждой партиции/collect/ окончательная агрегация на Master'е
- `countByValue` - локальная агрегация/при необходимости shuffle с последующей агрегацией/collect

Итого (2)

Минимальная единица вычисления - партиция. То есть, при выполнении любой операции будет вычислена хотя бы одна партиция:

- `rdd.take(n)` - Примерно эквивалентно `rdd.mapPartitions(_.take(n))` с последующим вычислением одной или нескольких (при необходимости) партиций.

Итого (3)

До начала вычислением любой партиции вычисляются **все** партиции, от которых она зависит. Есть следующие типы зависимостей:

- `OneToOneDependency`/`RangeDependency` - Каждая партиция зависит ровно от одной партиции родительского RDD
- `NarrowDependency` - Каждая партиция зависит от нескольких партиций родительских RDD (используется в `coalesce`)
- `ShuffleDependency` - Каждая партиция зависит от **всех** партиций родительских RDD

Оптимизация Shuffle

Пример Shuffle

```
val rdd = sc.parallelize(0 until 1000, 4)

val partitioner = new Partitioner {
  override def numPartitions: Int = 6
  override def getPartition(key: Any): Int = key.asInstanceOf[Int] % 6
}

val res = {
  rdd
    .map(_ -> 1)
    .partitionBy(partitioner)
    .mapPartitionsWithIndex { case (idx, iter) =>
      Iterator(idx -> iter.map(_._2).sum)
    }
    .collect()
}

res.sortBy(_._1).foreach { case (idx, cnt) => println(s"$idx -> $cnt") }
```

Как работает Shuffle

- Перед shuffle'ом вычисляются **все** партиции родительского RDD
- Каждая партиция разбивается на сегменты, соответствующие партициям результата
- Если сегменты не помещаются в память, они сериализуются и сбрасываются на диск (spill)
- Партиция считается вычисленной, когда все нужные сегменты собраны по сети
- Если до вычисления были потеряны Executor'ы, содержащие нужные сегменты, соответствующие партиции родительского RDD вычисляются повторно.

Способы оптимизации

- **Правило №1** Требуется отфильтровывать ненужные данные как можно раньше
- Отказ от полиморфизма (особенно, мелких объектов)
- Регистрация сериализаторов
- Использование массивов примитивов
- Предварительное партиционирование

Регистрация сериализаторов

```
val sparkConf = new SparkConf(...)
  ...
  .set("spark.kryo.classesToRegister",
    "mlds.serializer.SerialSeq1,mlds.serializer.SerialBean")
```

- Все, что требуется - **до** создания SparkContext'а прописать свойство конфигурации, содержащее все нужные классы (через запятую)

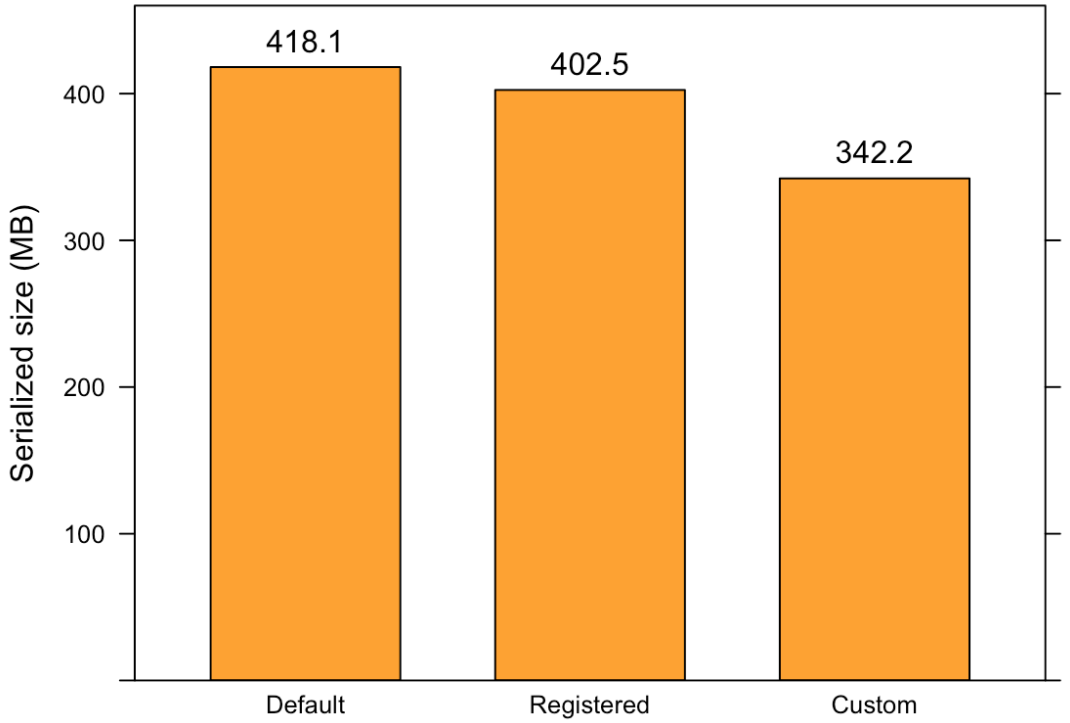
Использование массивов примитивов

```
case class SerialBean
(
  key: Int,
  value: Long
)

case class SerialSeq1
(
  id: Int,
  items: Iterable[SerialBean]
) extends SerialSeq
```

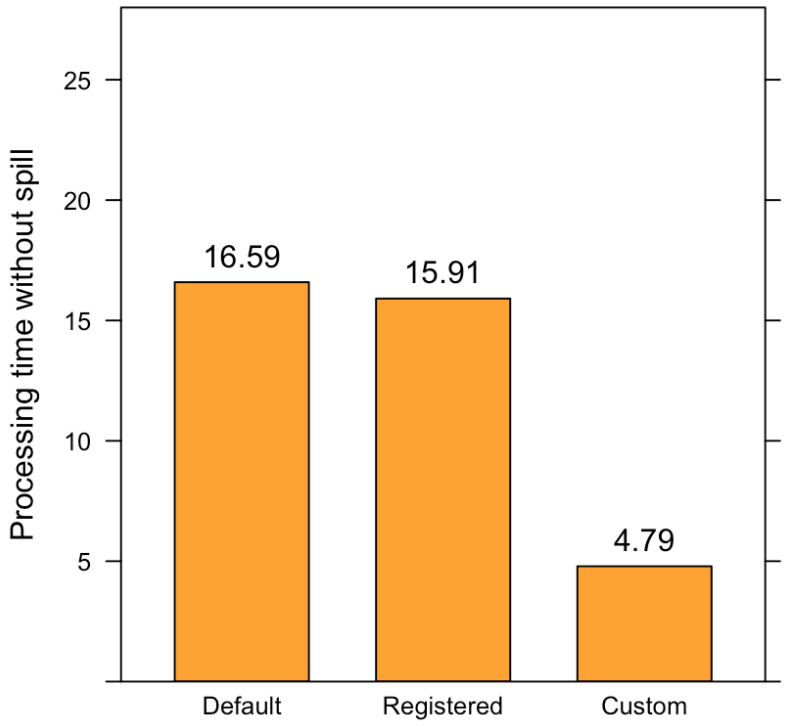
```
case class SerialSeq2
(
  id: Int,
  itemKeys: Array[Int],
  itemValues: Array[Long]
) extends SerialSeq {
  override def items =
    (0 until itemKeys.length).map { i =>
      SerialBean(itemKeys(i), itemValues(i))
    }
}
```

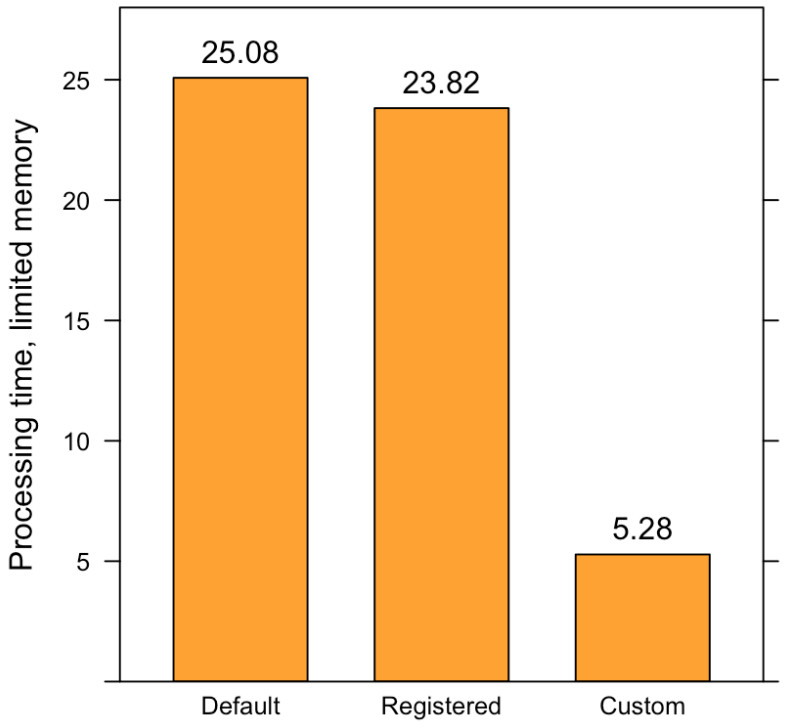
- Сериализация каждого объекта влечет накладные расходы. Сокращение **количества** сериализуемых объектов уменьшает эти накладные расходы.

Размер shuffle



Время выполнения





Предварительное партиционирование

- Позволяет **совсем** избежать shuffle
- На практике, позволяет ускорить вычисления в десятки/сотни раз
- Как этого добиться?
  - `persist/cache`
  - `checkpoint`
- Проблема: Spark не позволяет явным образом загрузить с диска партиционированные данные

PrepartitionedRDD

```
class PrepartitionedRDD[T: ClassTag]
(
  prev: RDD[T],
  part: Partitioner
) extends RDD[T](prev) {

  override val partitioner = Some(part)

  override def getPartitions: Array[Partition] = firstParent[T].partitions

  override def compute(split: Partition, context: TaskContext): Iterator[T] =
    firstParent[T].iterator(split, context)

  override protected def getPreferredLocations(split: Partition): Seq[String] =
    firstParent[T].getPreferredLocations(split)

}
```

Использование PrepartitionedRDD

```
val src = sc.objectFile[(Int, Long)]("...")

val rdd = new PrepartitionedRDD[(Int, Long)](src,
  new HashPartitioner(src.partitions.length))

val res = rdd.groupByKey().mapValues(_.sum).take(10) //No shuffle here
```

Внимание: Если партиционирование данных не соответствует объявленному, результат непредсказуем

Возможные причины:

- Что-то перепутали (функцию партиционирования и т.п.)
- Размер файла больше блока HDFS
- `InputFormat` портит порядок партиций (например, это делает `ParquetInputFormat`)

Оптимизация Transform

Эффективные коллекции

- Классы из `org.apache.spark.util.collection`:
  - `OpenHashMap`
  - `OpenHashSet`
  - `PrimitiveVector`
  - `CompactBuffer`
- GNU Trove (http://trove.starlight-systems.com/)

И еще…

- Оптимизированная библиотека BLAS на worker'ах
- Функции от итераторов, оптимизирующие распределение памяти:
  - `top(n)/bottom(n)`
  - `groupBy/groupByKey`
  - `toSortedIterator`

Q & A

Vyacheslav Baranov
Senior Software Engineer

vyacheslav.baranov@corp.mail.ru

https://github.com/SlavikBaranov/mlds

http://stackoverflow.com/users/941206/wildfire