Apache Spark

Следующее поколение приложений пакетной обработки больших объемов данных

Зачем нам нужен Spark ?

- Классические решения первого поколения (Hadoop MapReduce и прочие) обладают рядом недостатков:
  - Не умеют эффективно использовать память вычислительных узлов
  - Промежуточные данные вычислений сохраняются на винчестер
  - Громоздкое API
  - Сложность организации кеширования – мы должны сбрасывать кеш на диск, иначе данные в случае сбоя пропадут

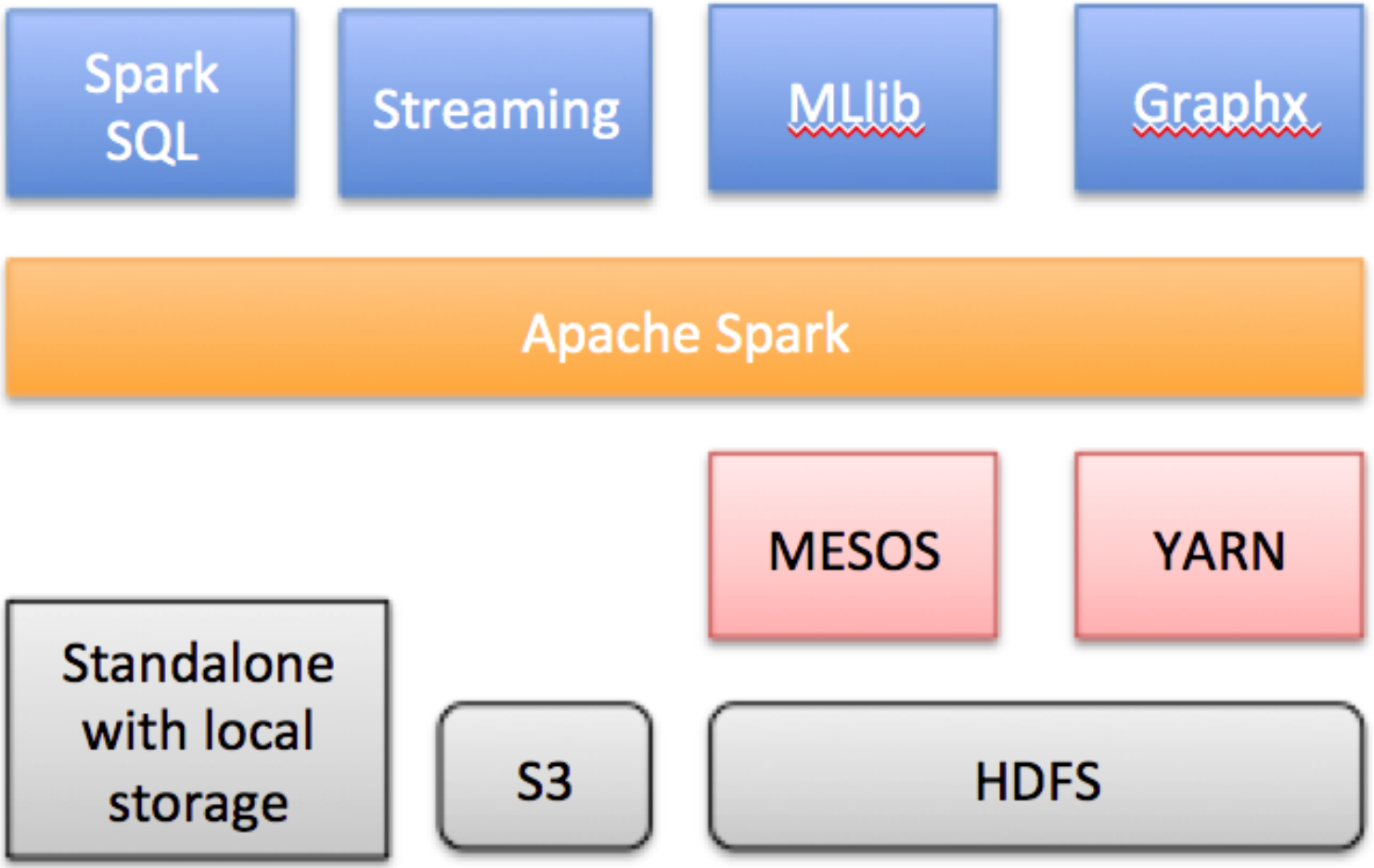
Что предлагает Spark

- Простое высокоуровневое API

```
val textFile = sc.textFile("README.md")
val count = textFile.filter(line => line.contains("Spark")).count()
```

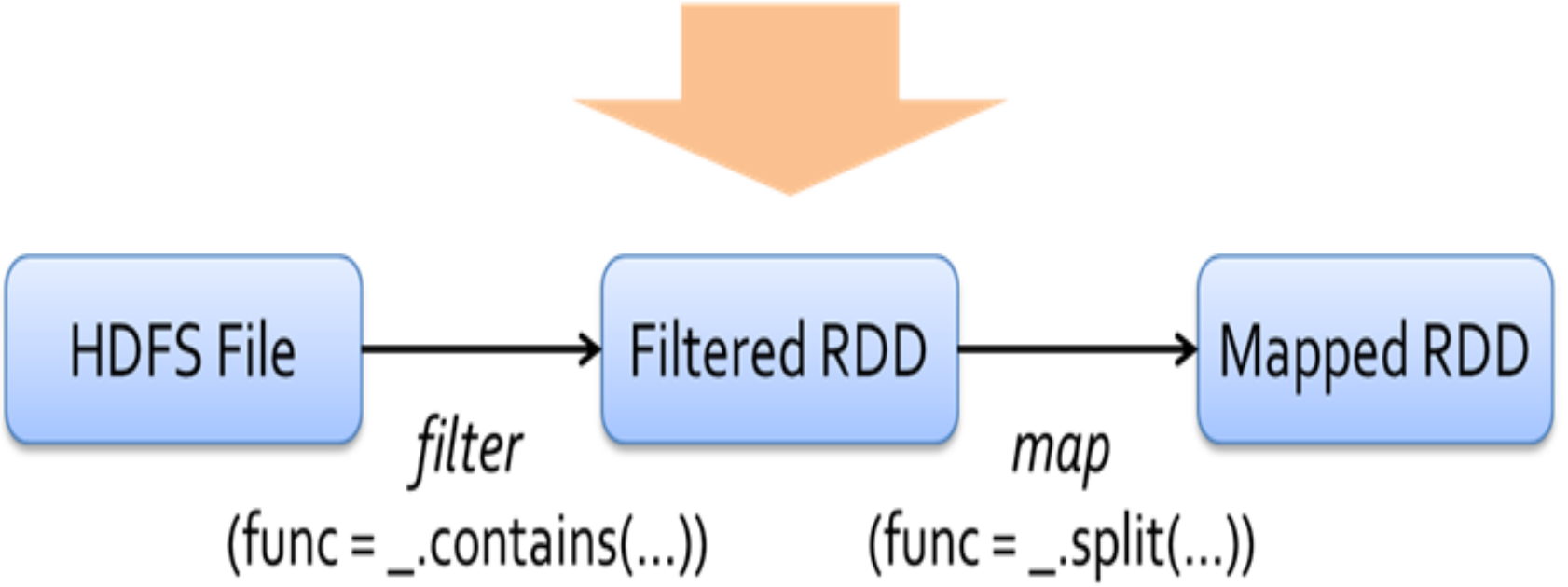
- Эффективное использование оперативной памяти – кеширование, повторное использование промежуточных результатов
- Возможность эффективно выполнять последовательность вычислений
- Интеграция с менеджером ресурсов HADOOP YARN
- Интеграция с основными распределенными системами хранения данных :
  - HDFS
  - HBASE
  - Cassandra
  - ...

Стек технологий Spark



На что это похоже

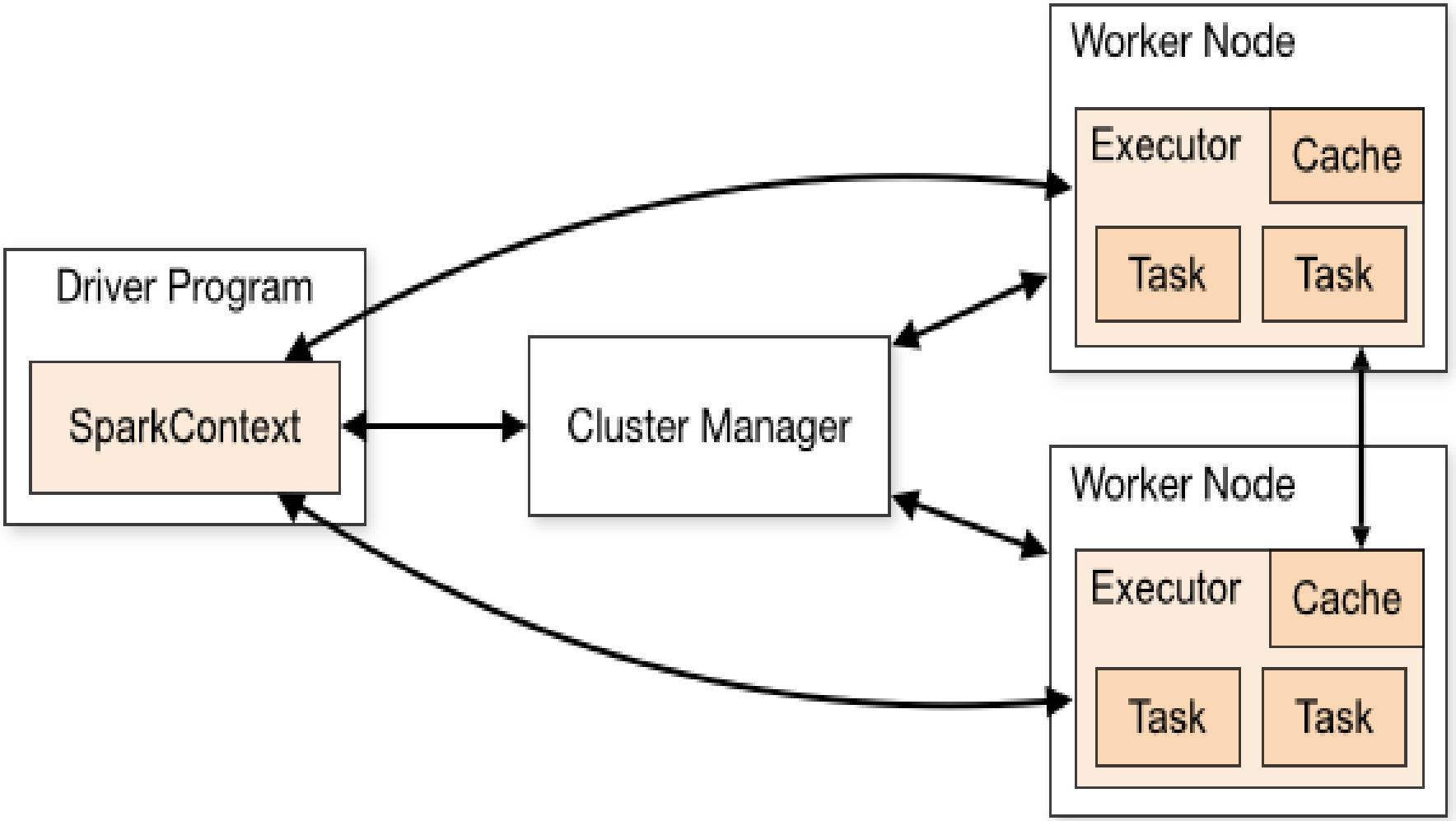
```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



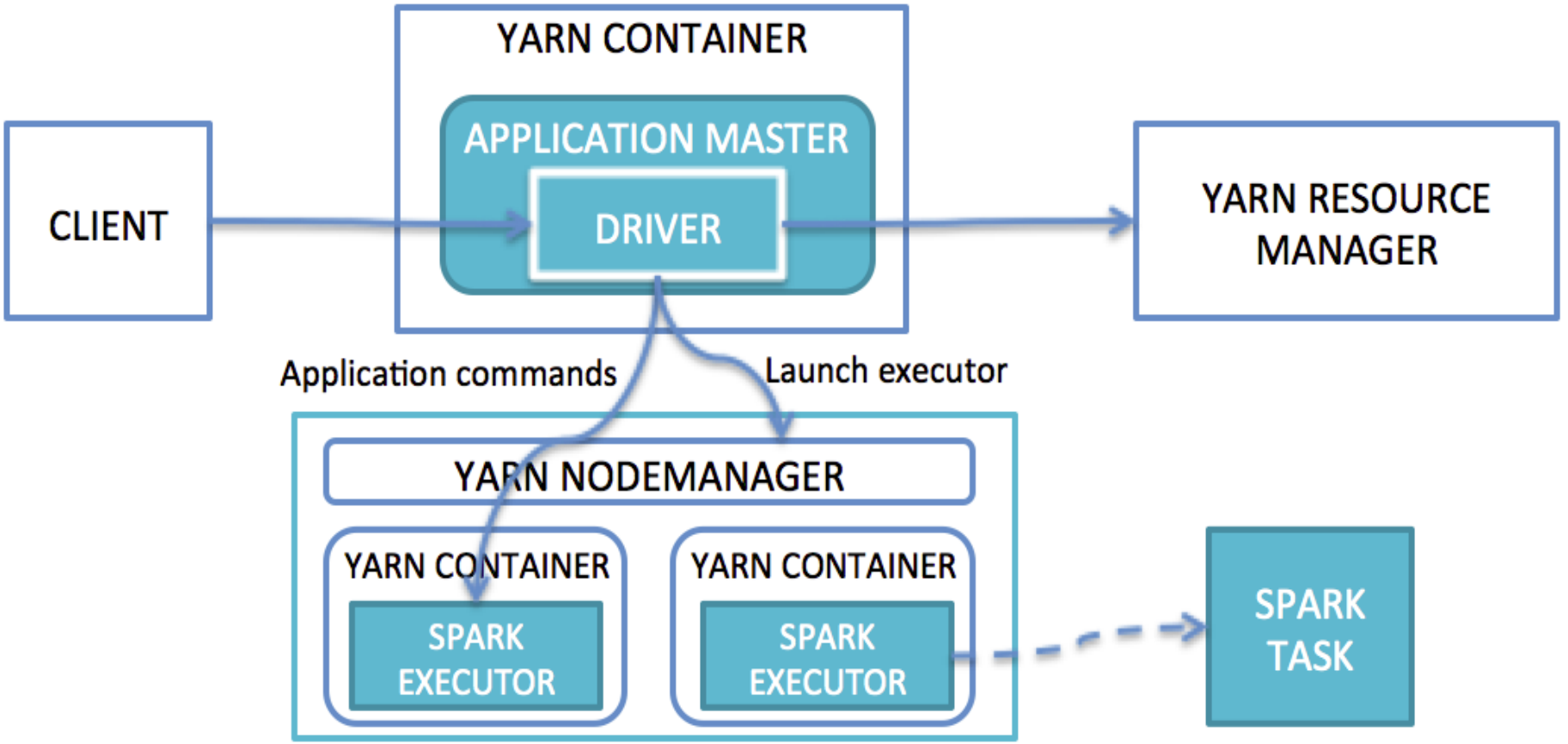
Архитектура Spark

- Кластер Spark состоит из узлов двух видов :
  - Cluster manager – управляет выполнением задач в кластере Spark
  - Executor Node – узел для запуска преобразования RDD
- Кластер Spark может быть запущен как в standalone режиме, так и поверх менеджеров ресурсов (YARN, Mesos)
- Важным отличием Spark от стандартного MapReduce является наличие кеша на каждом из узлов executor

Архитектура Spark

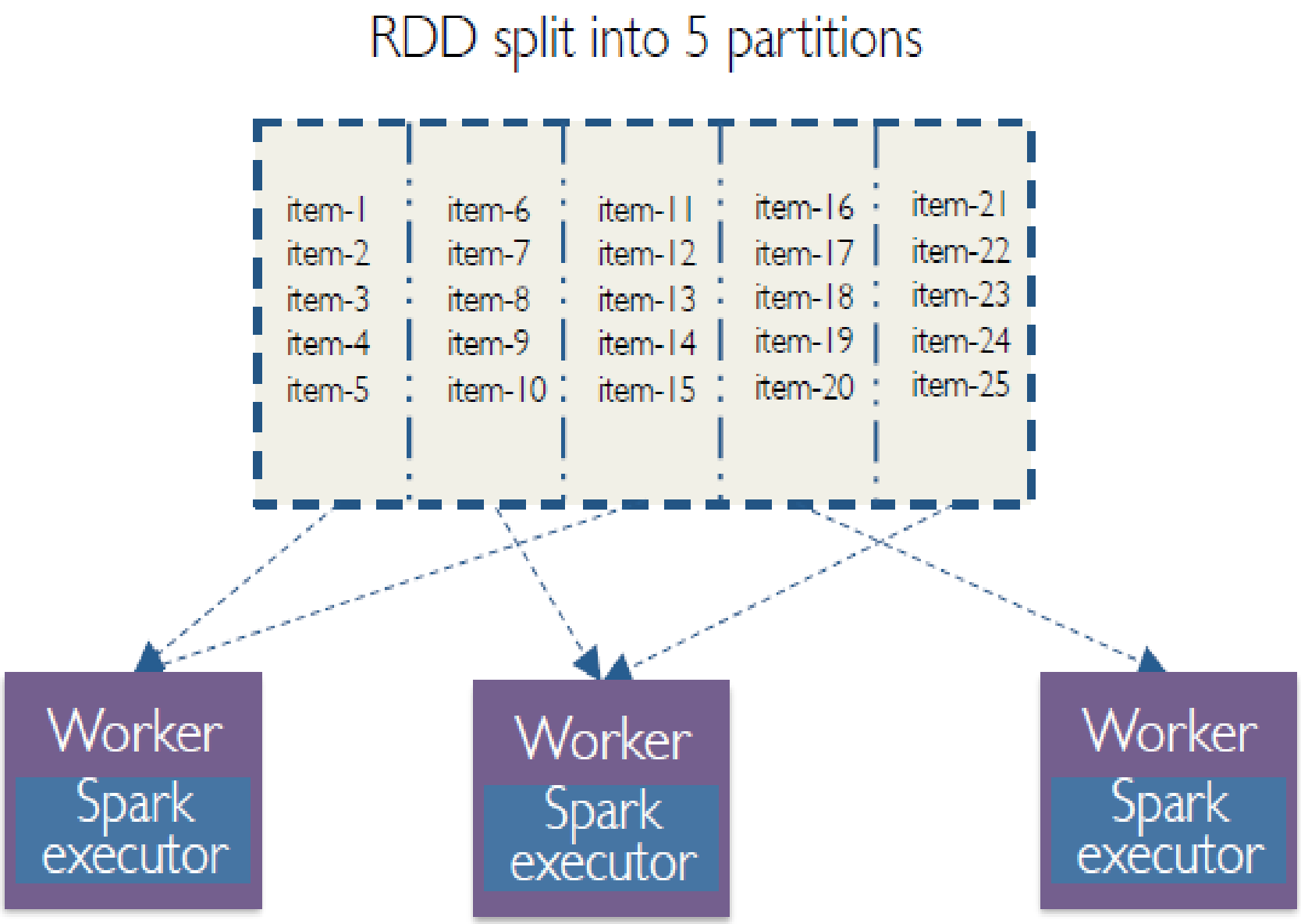


Spark в YARN


YARN CONTAINER
APPLICATION MASTER
DRIVER
CLIENT
YARN RESOURCE MANAGER
Application commands
Launch executor
YARN NODEMANAGER
YARN CONTAINER
SPARK EXECUTOR
YARN CONTAINER
SPARK EXECUTOR
SPARK TASK

RDD – Resilent Data Set

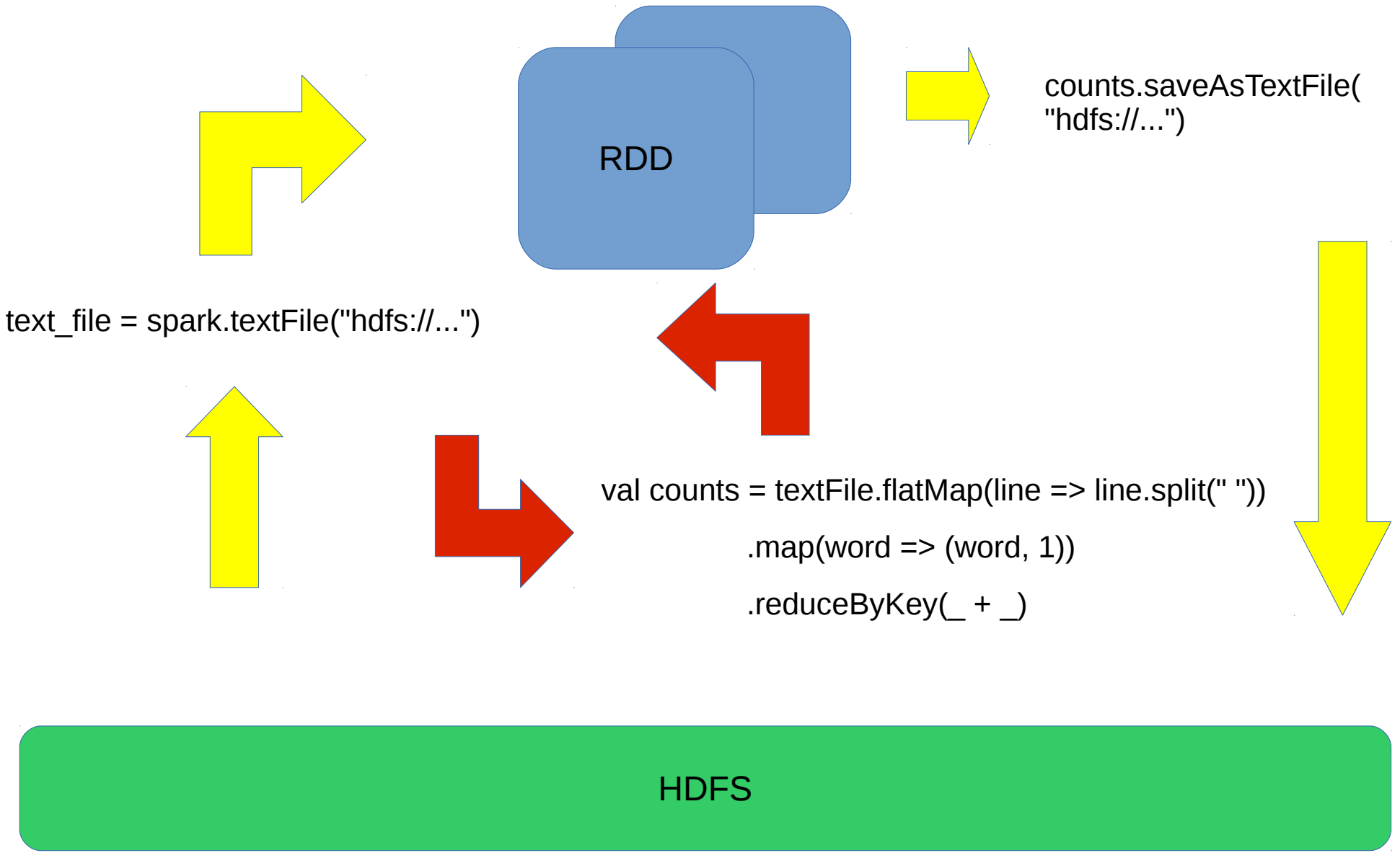
- RDD – Набор данных разбитый на блоки и “размазанный” по кластеру Spark.
- RDD – может быть как первичным (например считанным из HDFS), так и вычисленным в результате операций над другими блоками
- Spark помнит как был вычислен каждый из блоков промежуточных RDD
- В случае утери блока данных – блок будет перерассчитан на другом узле используя те же исходные значения


RDD split into 5 partitions
item-1
item-2
item-3
item-4
item-5
item-6
item-7
item-8
item-9
item-10
item-11
item-12
item-13
item-14
item-15
item-16
item-17
item-18
item-19
item-20
item-21
item-22
item-23
item-24
item-25
Worker
Spark executor
Worker
Spark executor
Worker
Spark executor

Элементы RDD

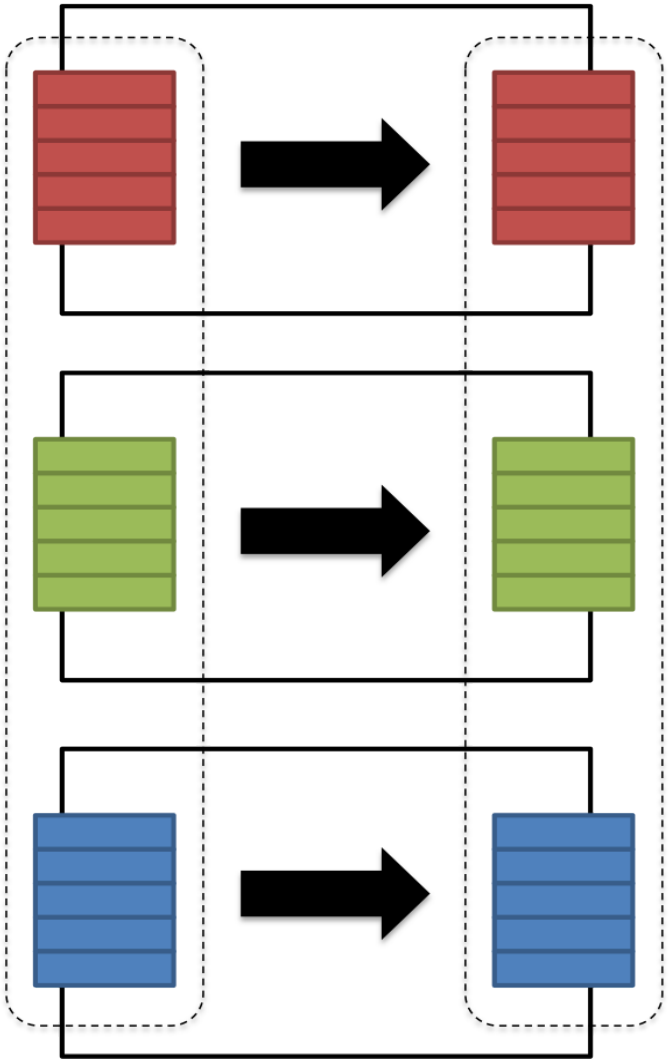
- RDD представляет собой набор записей
- Запись после загрузки из внешнего хранилища может быть двух видов :
  - Обычный java объект реализующий интерфейс serializable
  - Пара из ключа и значения. И ключ и значение должны быть serializable
    - В java api пара – scala.Tuple2
    - В scala – обычный tuple
- По умолчанию используется стандартный Java сериализатор, но возможно подключение внешней библиотеки Kryo

Transformations, Actions

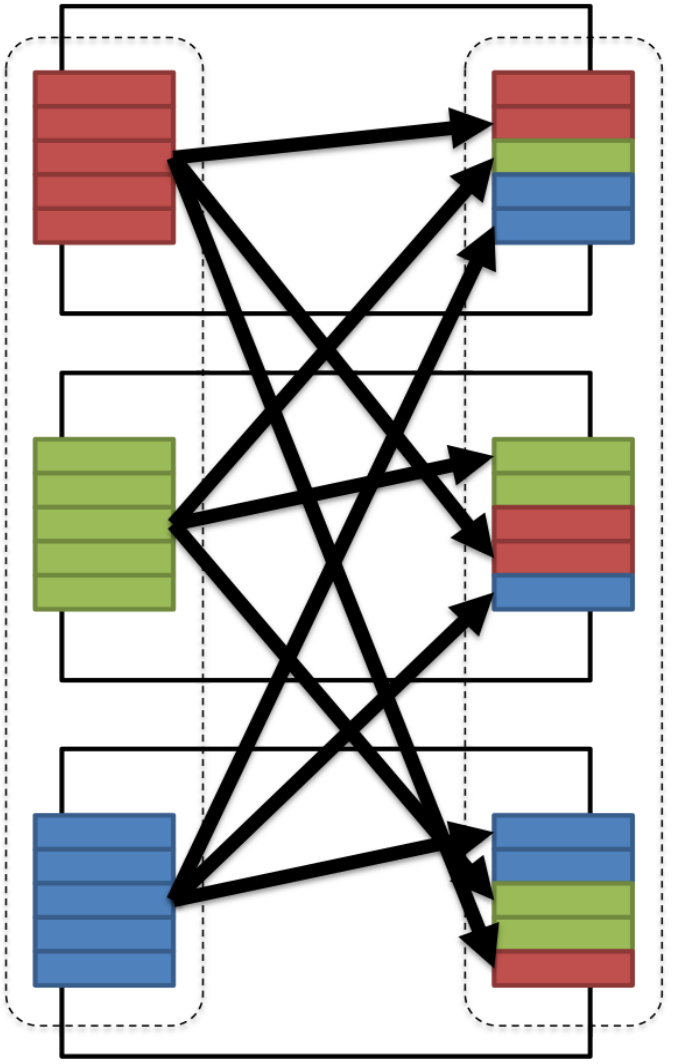


Виды оперций над RDD

Narrow transformation



Wide transformation



Narrow transformation

- “Узкое” преобразование(Narrow transformation)

  Производится над каждым блоком RDD независимо

- Операции :

  Map – отображение элемента RDD в другой вид

  FlatMap – отображение элемента RDD в несколько элементов

  Filter – фильтрация записей RDD

  Sample – случайная выборка записей RDD

  Union – объединение наборов RDD

Wide transformation

- “Широкое” преобразование(Wide transformation)

  Производится над несколькими блоками одной или нескольких RDD

- Операции :

  SortByKey – сортировка по ключу

  ReduceByKey – операция Reduce для каждого значения ключа

  (K, V) → (K, V)

  GroupByKey – Группировка записей по ключу

  (K, V) → (K, Iterable<V>)

  Cogroup – Объединение двух RDD вместе с группировкой по ключу

  (K, V) + (K, W) -> (K, Iterable<V>, Iterable<W>)

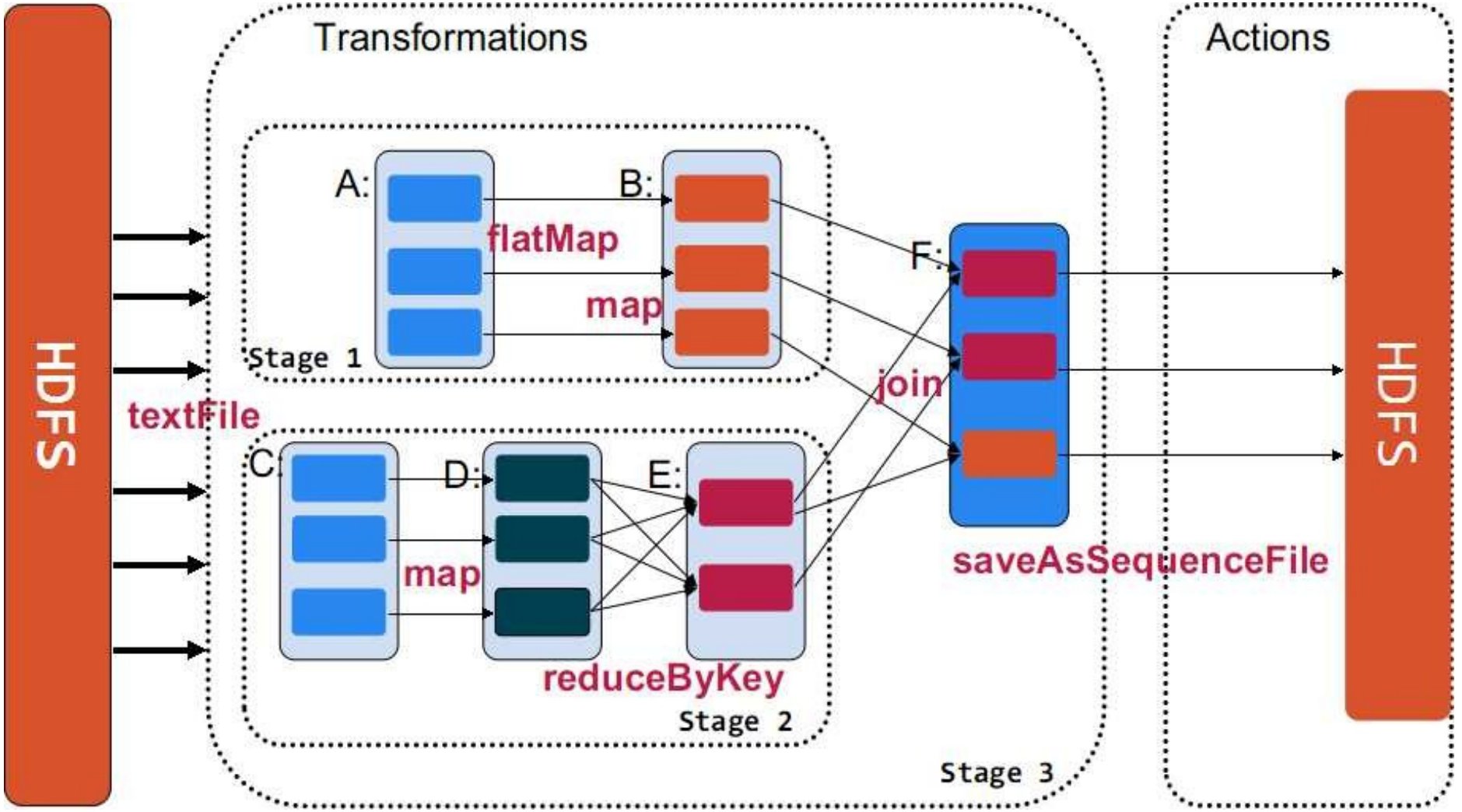
  Join – соединение таблиц

  (K, V) + (K, W) -> (K, (V, W))

  Cartesian - картезианское соединение

  T + U -> (T, U)

Операции над RDD



Операции сохранения данных

- Операции предназначенные для завершающего выода данных
- Операции :

  Collect – возвращает весь набор данных в клиентскую программу

  Take(n) – возвращает первые n элементов

  Reduce – применяет функцию reduce ко всему множеству и возвращает результат

  takeSample – производит случайную выборку данных из RDD

  Count – подсчитывает количество записей в RDD

  Save – сохраняет RDD в хранилище

Кеширование

- Spark позволяет кешировать промежуточные результаты в памяти узлов
- Управление кешированием производися явно с помощью вызовов:

  .persist(StorageLevel newLevel)

  .cache() эквивалентен .persist(MEMORY_ONLY)
- Уровни кеширования :
  - MEMORY_ONLY - хранение RDD в виде java объектов в памяти
  - MEMORY_AND_DISK – записи RDD хрянятся в виде java объектов в памяти, в случае необходимости блоки сериализуются и хранятся на диске
  - MEMORY_ONLY_SER - хранение RDD в виде десериализованных java объектов в памяти (более эффективно с точки зрения объема, но более затратно с точки зрения CPU)
  - MEMORY_AND_DISK_SER – аналогично MEMORY_ONLY_SER но с сохранением на диск в случае недостатка памяти
  - DISK_ONLY – хранение данных строго на диске

Отказоустойчивость

- При использовании цепочек вычислений в традиционном MapReduce мы вынуждены сохранять промежуточные результаты в отказоустойчивом хранилище
- Spark позволяет хранить промежуточные результаты в памяти
- Spark помнит как был вычислен каждый блок. В результате утери он может полностью воспроизвети цепочку вычисления и заново сформировать требуемый блок RDD

Загрузка данных

- Spark тесно интегрирован с основными nosql источниками данных и позволяет работать с данными hdfs, hbasr, cassandra и т.д.
- “Из коробки” предоставляются следующие способы загрузки данных:
  - .textFile – загружает файл из hdfs. Каждая запись RDD – строка.
  - .hadoopFile – загружает данные с помощью hadoop InputFormat. Каждая запись RDD – объект десериализованный с помощью InputFormat
  - .sequenceFile – заружает данные из hadoop sequence file
  - .objectFile – загружает RDD сохраненный в “родном” формате в виде сериализованных java объектов

Пример работы со Spark

- Инициализация приложения

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

- Загрузка данных

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

- Разбиение строки на слова

```
JavaRDD<String> splitted = distFile.flatMap(
                        s -> Arrays.stream(s.split(" ")).iterator()
    );
```

- Отображение слов в пару <Слово,1>

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

Пример работы со Spark

- Считаем одинаковые слова

```
JavaPairRDD<String, Long> collectedWords = wordsWithCount.reduceByKey (
        (a, b) → a + b
    );
```

- Загружаем словарь

```
JavaRDD<String> dictionaryFile = sc.textFile("words.txt");
JavaPairRDD<String, Long> dictionary = dictionaryFile.mapToPair(
        s ->  new Tuple2<>(s,1l)
    );
```

- Производим операцию join со словарем

```
JavaPairRDD<String, Tuple2<Long, Long>> joinValue = dictionary.join(collectedWords);
```

- Печатем результат

```
System.out.println("result="+joinValue.collect());
```