trainologic

Spark 2.3

Apache Spark with Java

trainologic

Agenda

1. Spark intro
2. Resilient Distributed Dataset - RDD
3. Spark architecture and more
4. Spark SQL - dataframes and datasets
5. Spark Streaming and Structured Streaming
6. Integrating Spark into your system
7. Spark performance and tuning

trainologic

What is Spark?

'Apache Spark™ is a unified analytics engine for large-scale data processing'

- ***Speed***
  Run workloads 100x faster than Hadoop
- ***Ease of Use***
  Write applications quickly in Java, Scala, Python, R, and SQL
- ***Generality***
  Combine SQL, streaming, and complex analytics
- ***Runs Everywhere***
  Spark runs on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud. It can access diverse data sources

trainologic

Spark

- Spark is a cluster computing engine
- Written in Scala
- Provides high-level API in Scala, Java, Python and R
- Provides high level tools:
  - Spark SQL
  - MLib
  - GraphX
  - Spark Streaming

trainologic

RDD - Resilient Distributed Dataset

- The basic abstraction in Spark is the RDD
- Stands for: **R**esilient **D**istributed **D**ataset
- It is a collection of items. The source can be:
  - Hadoop (HDFS)
  - S3
  - JDBC
  - ElasticSearch
  - And more…
- If possible it resides in-memory (one of the key advnatges of Spark)

trainologic

RDD - Resilient Distributed **Dataset**

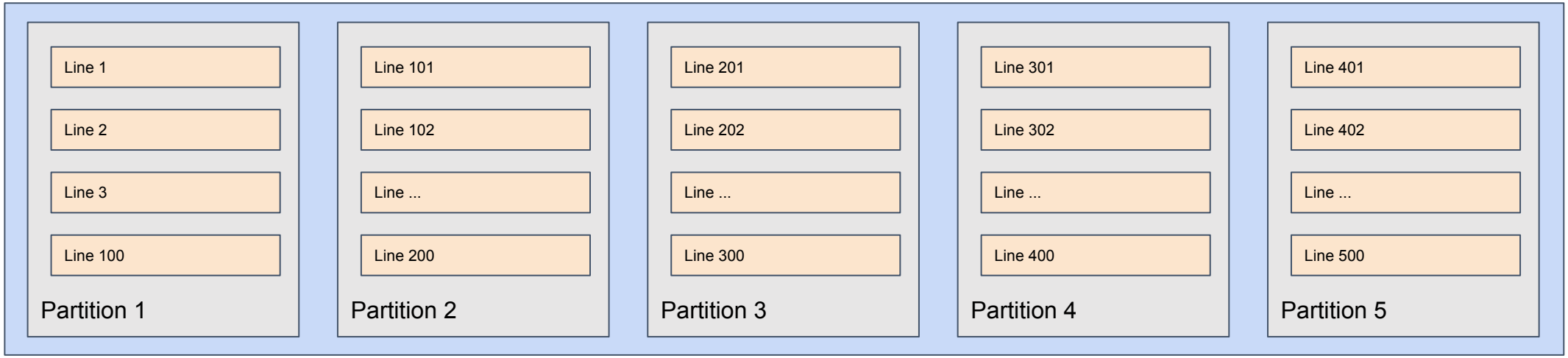
- RDD sources can vary:
  - Hadoop (HDFS)
  - S3
  - JDBC
  - ElasticSearch
  - And more…
- All RDDs looks the same without connection to their source

trainologic

RDD - Resilient **Distributed** Dataset

- Partition is a sub-collection of data that should fit into memory
- Partition can reside in a different worker or even different node
- Partition created either from source or using transformation (more on that later)
- This is the distributed part of the RDD

→ **e.g. text file with 500 lines:**

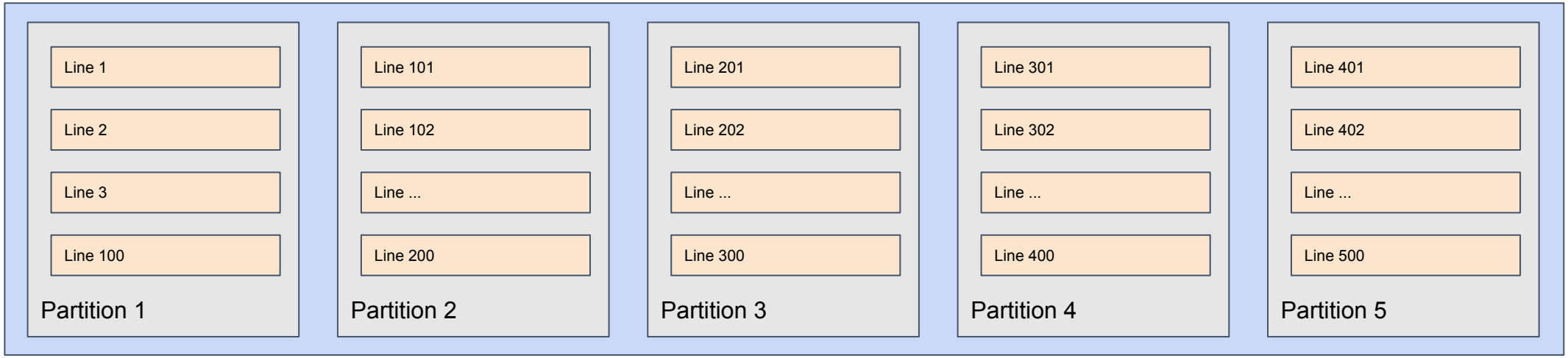


trainologic

RDD - **Resilient** Distributed Dataset

- Partition + transformation = Task
- Spark tries to cache partition in-memory for future reuse
- In case of failure partition can be recalculated (this is the ‘resilient’ part)

→ In the example below, if partition 2 failed (say the machine crashed) Spark will submit the task to a different worker



trainologic

More on RDD

- RDD can depend on other RDDs
- RDD is **lazy**

```
JavaRDD<String> lowerCaseRDD = wordRDD.map(word -> word.toLowerCase());
```

➔ lowerCaseRDD is **NEW** rdd that depends on wordRDD
➔ lowerCaseRDD contains **ONLY** metadata (i.e. data source and computing function)

The flow will compute only on a specific commands (knowen as ‘actions’)

trainologic

RDD from Collection

- You can create an RDD from a collection:

```
JavaSparkContext jsc = new JavaSparkContext(conf);
List<String> data = Arrays.asList("hello", "Spark", "Java");
JavaRDD<String> wordRDD = jsc.parallelize(data);
```

- Takes a sequence from the driver and distributes it across the nodes.
- Note, the distribution is lazy so be careful with mutable collections!
- The collection distributed, don’t count on order!

trainologic

RDD from file

- Spark supports reading files, directories and compressed files.
- The following out-of-the-box methods:
  - **textFile**

    retrieving an JavaRDD<String> (lines)
  - **wholeTextFiles**

    retrieving an JavaPairRDD<String, String> with filename and file content
  - **sequenceFile**

    Hadoop sequence files JavaPairRDD<Key, Value>

trainologic

RDD Transformations

Return pointer to new RDD with transformation meta-data

- **map(func)** - Return a new distributed dataset formed by passing each element of the source through a function func.
- **filter(func)** - Return a new dataset formed by selecting those elements of the source on which func returns true.
- **flatMap(func)** - Similar to map, but each input item can be mapped to 0 or more output items (so func should return a collection rather than a single item).

→ Taking from Spark official site:
https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations

trainologic

RDD Transformations

Return pointer to new RDD with transformation meta-data

- **sample(withReplacement, fraction, seed)** - Sample a fraction using a given random number and generator seed
- **union(otherDataset)** - Return a new dataset that contains the union of the elements in the source dataset and the argument
- **intersection(otherDataset)** - Return a new RDD that contains the intersection of elements in the source dataset and the argument
- **distinct([numPartitions]))** - Return a new dataset that contains the distinct elements of the source dataset

→ Taking from Spark official site:
https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations

trainologic

RDD Transformations - Keyed RDD

- **groupByKey([numPartitions])** - When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
  - Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using *reduceByKey* or *aggregateByKey* will yield much better performance.
  - Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.
- **reduceByKey(func, [numPartitions])** - When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

→ Taking from Spark official site:
https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations

trainologic

RDD Transformations - Keyed RDD

- **aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions])** - When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
- **sortByKey([ascending], [numPartitions])** - When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

→ Taking from Spark official site:
https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations

trainologic

RDD Actions

Return values by evaluating the RDD (not lazy)

- **collect()** – returns an list containing all the elements of the RDD
- **count()** – returns the number of the elements in the RDD.
- **first()** – returns the first element of the RDD.
- **foreach(f)** – performs the function on each element of the RDD.
- **saveAs()**…

trainologic

Lets count some words

trainologic

Lets count some words

We would like to calculate bag-of-word on *src/main/resources/books/book1.txt*

What will be the steps??

1. Read the file into RDD
2. Split the lines into words
3. Map into tuples of (word, 1)
4. Count ‘1’ by key

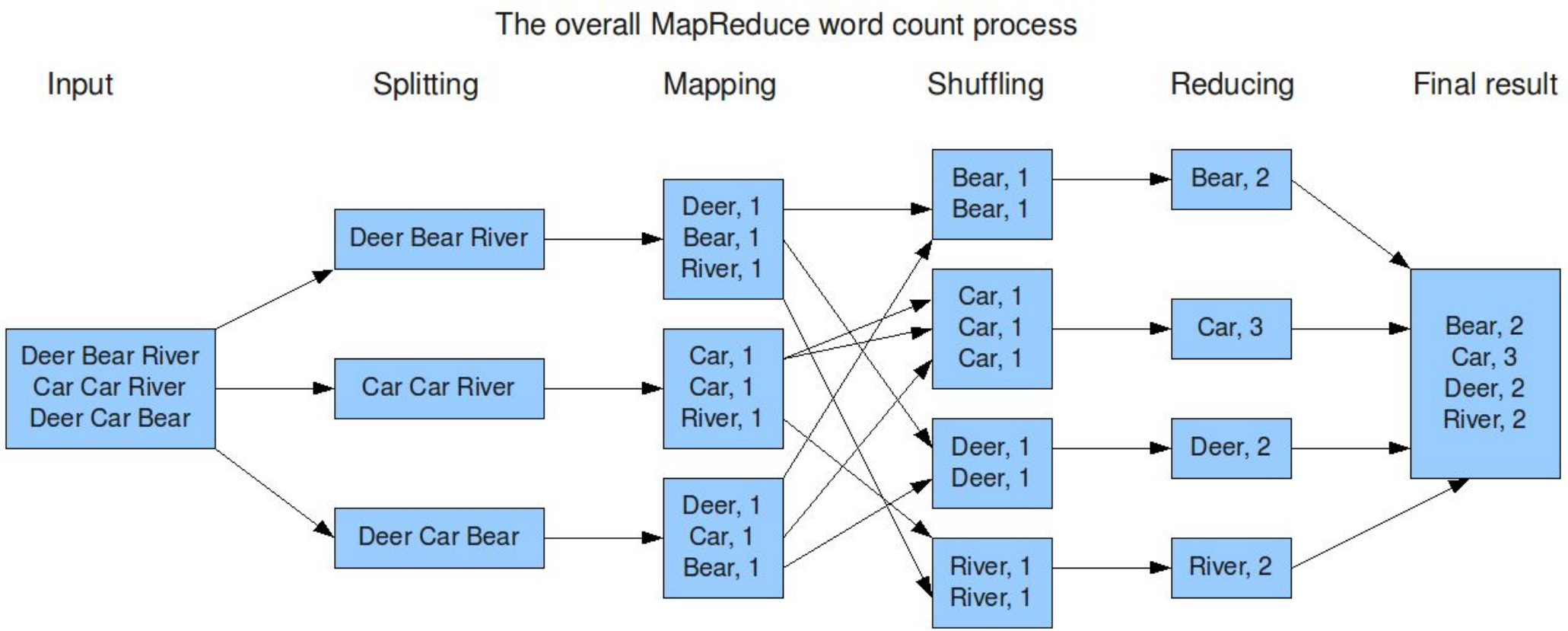
trainologic

Map Reduce

- Map-reduce was introduced by Google and it is an interface that can break a task to sub-tasks distribute them to be executed in parallel (map) , and aggregate the results (reduce).
- Between the Map and Reduce parts, an alternative phase called ‘shuffle’ can be introduced.
- In the Shuffle phase, the output of the Map operations is sorted and aggregated for the Reduce part. (Hadoop)

trainologic

Map Reduce



trainologic

Shuffle

Shuffling is a process of redistributing data across partitions (aka *repartitioning*)
The Shuffling may or may not cause moving data across JVM processes or even over the wire (between executors on separate machines)

Shuffling is the process of data transfer between stages

trainologic

Shuffle

- Shuffle operations repartition the data across the network
- Can be very expensive operations in Spark
- You must be aware where and why shuffle happens
- Order is not guaranteed inside a partition
- Popular operations that cause shuffle are: *groupBy**, *reduceBy**, *sort**, *aggregateBy** and *join/intersect* operations on multiple RDDs.

trainolo

GroupBy Requires ALL of the Previous RDD's Partitions to Compute Any Single Partition

```
val rdd = sc.cassandraTable("sc2","points")
.groupBy(_.getString("country"))
```

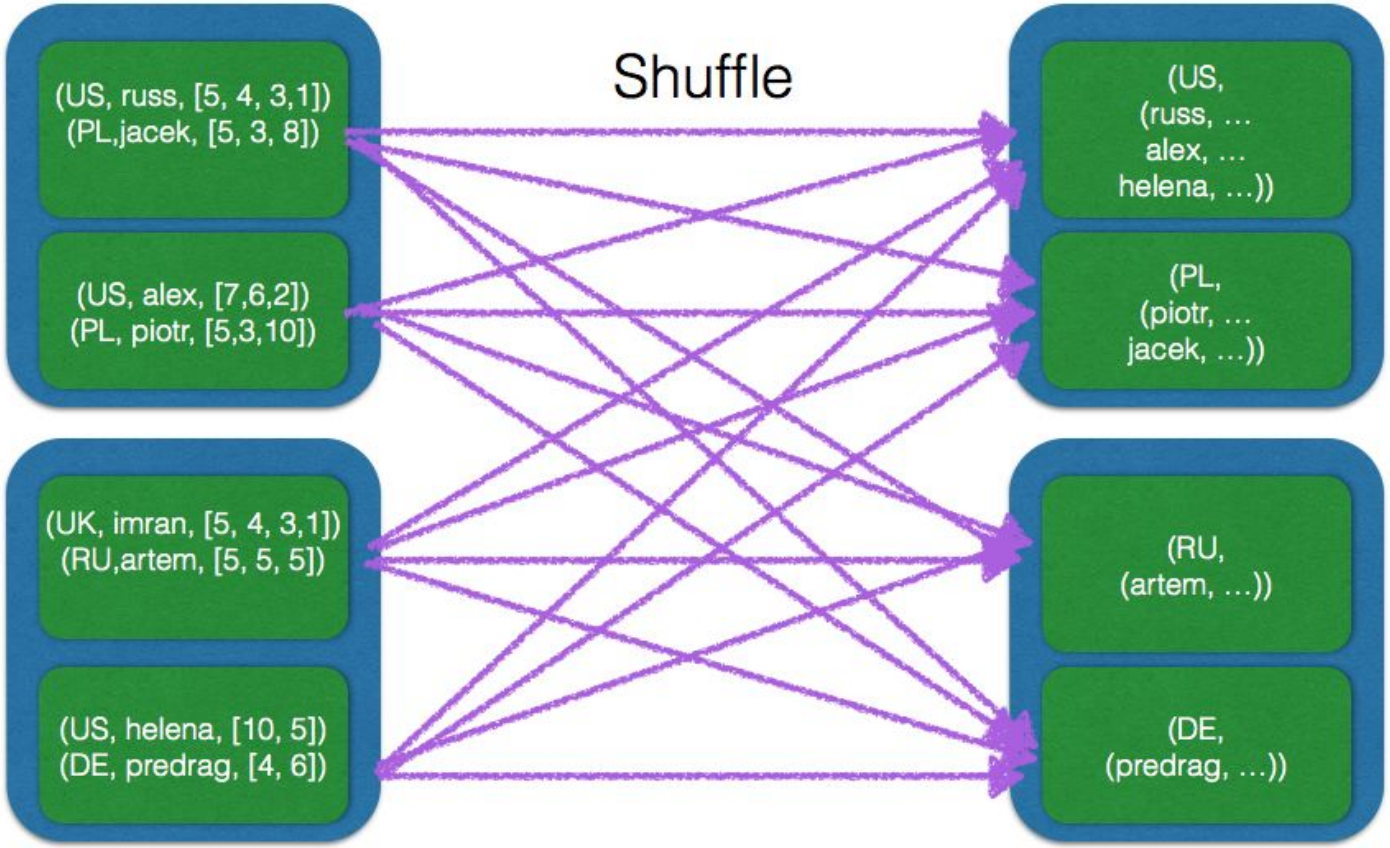
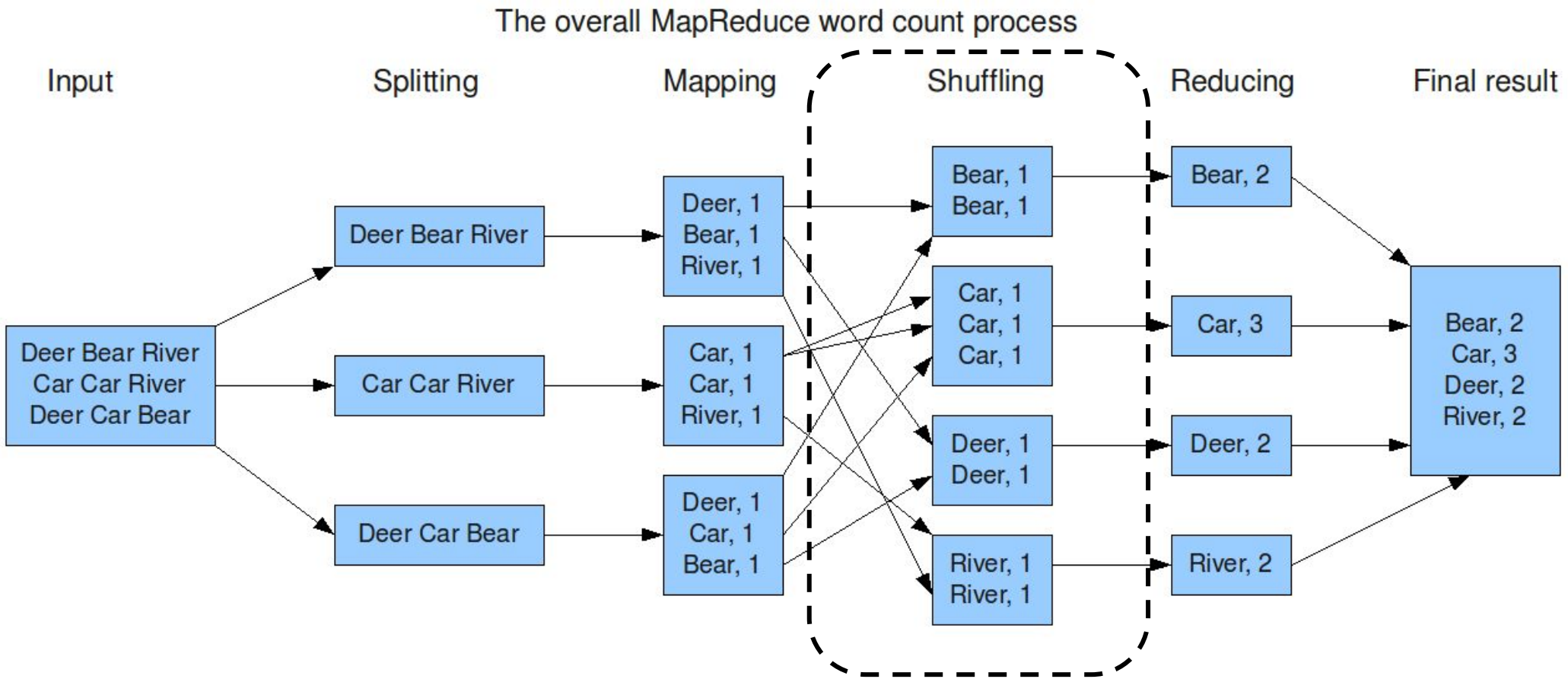


Image taken from https://www.datastax.com/wp-content/uploads/2015/05/SparkShuffle.png

trainologic

Let’s look at shuffle again



trainologic

Transformation that shuffle

- ***distinct([numTasks])*** - Return a new dataset that contains the distinct elements of the source dataset.
- ***groupByKey([numTasks])*** - When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
- ***reduceByKey(func, [numTasks])*** -When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.

*Taken from the official Apache Spark documentation

trainologic

Transformation that shuffle

- ***join(otherDataset, [numTasks])*** - When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
- ***sort, sortByKey***…
- More at http://spark.apache.org/docs/latest/programming-guide.html
-

*Taken from the official Apache Spark documentation

trainologic

Spark architecture and more

- How does it work?
- Spark Terminology
- Per-partition Operations
- Caching
- Broadcast And Accumulators

trainologic

How does it work?

- Cluster manager (e.g. Yarn, Spark, Mesos)
  - Responsible for the resource allocation
  - Cluster management
  - Orchestrates work
- Worker nodes:
  - Tracks and spawn application processes

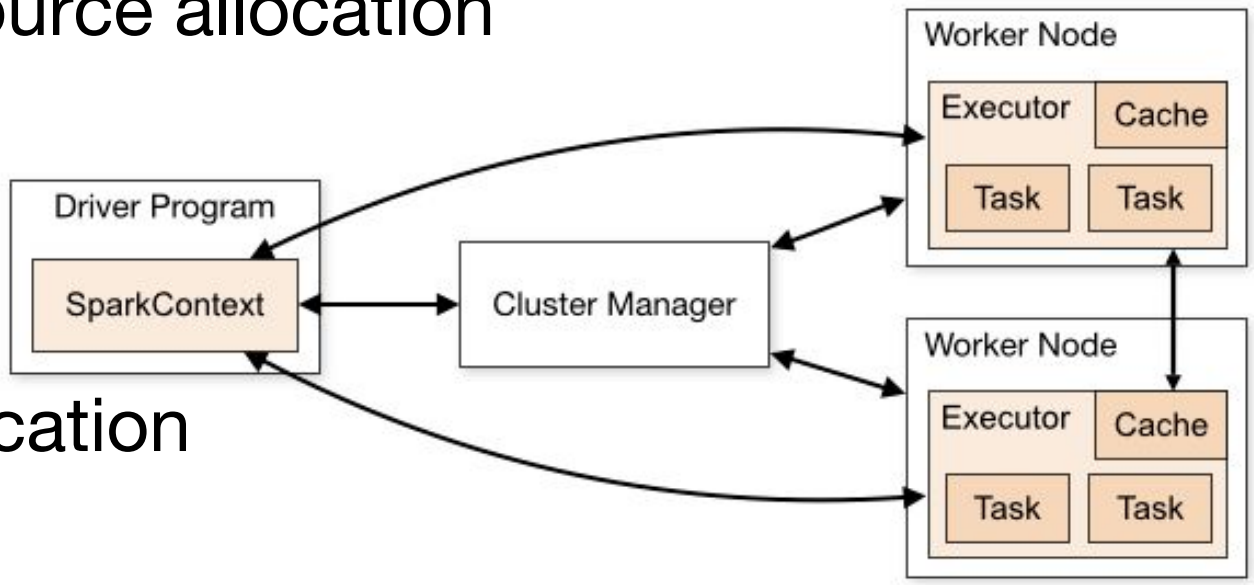


Image taken from https://spark.apache.org/docs/latest/cluster-overview.html

trainologic

How does it work?

- Driver:
  - Executes the main program
  - Creates the RDDs
  - Collects the results
- Executors:
  - Executes the RDD operations
  - Participate in the shuffle

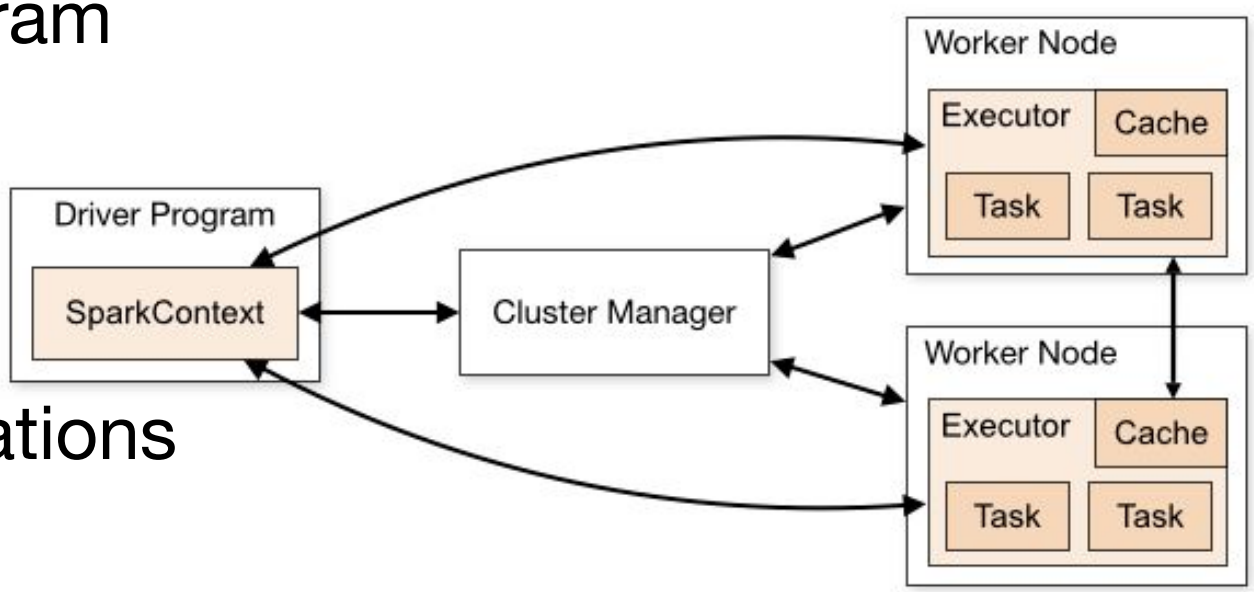


Image taken from https://spark.apache.org/docs/latest/cluster-overview.html

trainologic

Spark Terminology

- **Application –** the main program that runs on the driver and creates the RDDs and collects the results.
- **Job –** a sequence of transformations on RDD till action occurs.
- **Stage –** a sequence of transformations on RDD till shuffle occurs.
- **Task –** a sequence of transformations on a **single partition** till shuffle occurs.

trainologic

Pi? Pie?

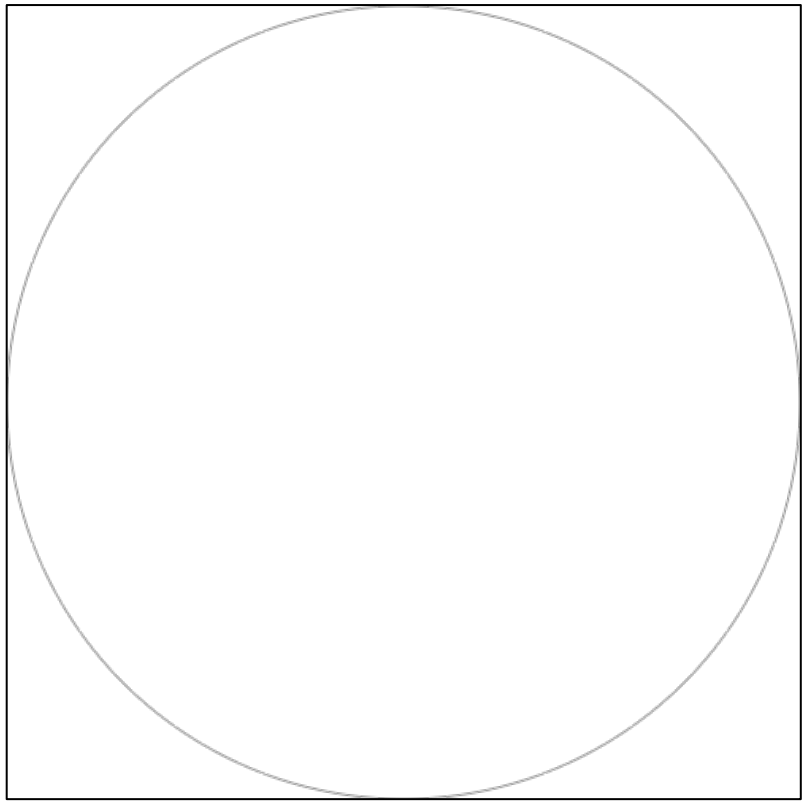
π

We would like to calculate
Pi Approximation using
Monte Carlo Method

DON'T copy from Google!

Basically it means:

1. Take Circle with Radius 0.5, enclosed by 1 X 1 square
2. Divide the area of the circle by the area of the suare (πr2=π/4)/1 == π/4
3. generate a large number of uniformly distributed random points and plot them on the graph
4. Pi estimation will be (number of inner points * 4) / total number of points



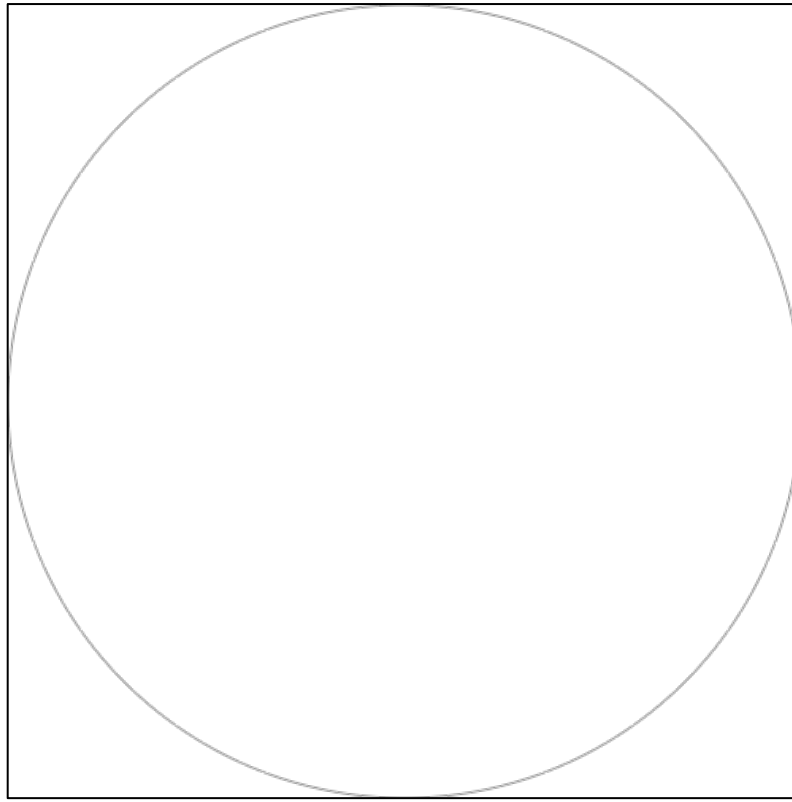
trainologic

Pi? Pie?

π

Hints:

1. This how the method signature should look:
   `public static void pi(int sampelSize)`
2. If you need random number (you need ;)):
   `Math.random()`
3. The best algorithm will be:
   For sampelSize times:
   a. Randomly choose position in the square (x,y)
   b. Count those who fell inside the circle
      → `x*x + y*y < 1;`
   c. The count of the above points * 4 / sampleSize will give good Pi Approximation



trainologic

Per-partition Operations

Spark provides several API for per-partition operations:

- **mapPartitions(func)** - Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
- **mapPartitionsWithIndex(func)** - Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.

→ The truth is that ***map()*** actually implemented using **mapPartitions()...**

trainologic

Per-partition Operations

When will we want that??

- When using external expensive resource (e.g. DB connector) for each line.
  - With ***map*** - it'll open connection every row
  - With ***mapPartitions*** - will open connection per partition
- When you you have 'hints': e.g. events on the same user kepy in the same partition.

  You can use ***mapPartitions*** to do local calculations

trainologic

Per-partition Regex

As part of the wordcount program, we would like to take only legal alphanum words.

We can do it using ‘w’ regex:

```
Pattern pattern = Pattern.compile("\\w+");

System.out.println(pattern.matcher("ophir").matches());  //true

System.out.println(pattern.matcher("ophir-cohen").matches());  //false
```

trainologic

Per-partition Regex

As part of the wordcount program, we would like to take only legal alphanum words.

We can do it using ‘w’ regex:

```
Pattern pattern = Pattern.compile("\\w+");

System.out.println(pattern.matcher("ophir").matches());  //true

System.out.println(pattern.matcher("ophir-cohen").matches());  //false
```

trainologic

Per-partition Regex - map

Using ***map()***:

```
JavaRDD<Boolean> matchesRDD  = wordsRDD.map(word -> {
  Pattern pattern = Pattern.compile("\\w+");
  return pattern.matcher(word).matches();
});
```

→ How many times will Pattern.compile("\\w+") will be created??

trainologic

Per-partition Regex - II

Using ***mapPartitions()***:

```
JavaRDD<String> matchesRDD  = wordsRDD.mapPartitions(wordItr -> { Pattern pattern = Pattern.compile("\\w+");
  List<String> mathcedList = new ArrayList<>();

  while (wordItr.hasNext()){ String word = wordItr.next();
      if (pattern.matcher(word).matches()){
          mathcedList.add(word);
      }}
  //Close resource if needed
  return mathcedList.iterator();});
```

→ How many times will Pattern.compile("\\w+") will be created??

trainologic

Caching

- One of the strongest features of Spark is the ability to cache an RDD.
- Spark can cache the items of the RDD in memory or on disk.
- You can avoid expensive re-calculations this way.
- The cache() and persist() store the content in memory.
- You can provide a different storage level by supplying it to the persist method.

trainologic

In-Memory Caching - storage level

1. ***MEMORY_ONLY*** - Store RDD as deserialized Java objects in the JVM.
   If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
2. ***MEMORY_AND_DISK*** - same as ***MEMORY_ONLY.*** If RDD does not fit into memory - store it on disk
3. ***MEMORY_ONLY_SER*** - Store RDD as serialized Java objects (one byte array per partition).
   This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
4. ***MEMORY_AND_DISK_SER*** - same as ***MEMORY_ONLY_SER*** with store on disk if needed.
5. ***DISK_ONLY*** - guess what?
6. ***MEMORY_ONLY_2, MEMORY_AND_DISK_2***, etc… - Same as the levels above, but replicate each partition on two cluster nodes.

trainologic

Unpersist

- Caching is based on LRU.
- If you don’t need a cache, remove it with the unpersist() method.

trainologic

Broadcasts And Accumulators

Broadcast variables

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

e.g.
You have ‘ignored’ list that created from an external source, or its taking time to create.

trainologic

Broadcasts And Accumulators

Option 1:

```
Set<Integer> ignored = new HashSet<>(Arrays.asList(1, 3, 10));
JavaRDD<Integer> res = intRDD.filter(in -> !ignored.contains(in));
```

Is that good?

1. ignored.contains(in) will run on a different JVM than so you can't expect its behavior
2. If it takes time to create it it might be takes time to recreate it over and over

trainologic

Broadcasts And Accumulators

Option 2:

```
Set<Integer> ignored = new HashSet<>(Arrays.asList(1, 3, 10));
Broadcast<Set<Integer>> ignoredBroadcast = sc.broadcast(ignored);
JavaRDD<Integer> res = intRDD.filter(in -> !ignoredBroadcast.getValue().contains(in));
```

1. Cached locally
2. Created once in the driver side and distributed to the workers

trainologic

Broadcasts And Accumulators

Accumulators

are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel

1. Similar to Hadoop counters
2. Convenience way to monitor your application

trainologic

Broadcasts And Accumulators

Accumulators - let's count

```
int yes, no = 0;

JavaRDD<Integer> res = intRDD.map(in -> {
   if (ignored.contains(in)) { yes += 1; }
 else {no += 1; }
   return in;
});
```

Does it works??

trainologic

Broadcasts And Accumulators

Accumulators - let's count

```
LongAccumulator yes = sc.sc().longAccumulator();LongAccumulator no = sc.sc().longAccumulator();
JavaRDD<Integer> res = intRDD.map(in -> {
 if (ignored.contains(in)) {yes.add(1);}
else {no.add(1);}
   return in;});
```

1. Executes in parallel
2. Present the results in the web UI
3. 'yes' or 'no' will be added by one only after 'action'
4. One can create its own accumulator

trainologic

RDD Final Exercise

trainologic

RDD Final Exercise

On the previous word count and update the first method
(JavaRDD<String> wordsRDD = linesRDD.flatMap(line -> Arrays.*asList*(line.split("\\s")).iterator());)

1. Use *mapPartitions()* instead of flat map
2. Add Stop word filter (remove "yes", "no", "ok", "in", "you" etc…)
3. Use only alphanumric words

```
Pattern pattern = Pattern.compile("\\w+");
pattern.matcher(str).matches()
```

→ Remember to broadcast the needed variables

4. Add accumulators for: legal words, illegal words, total words and total lines.