Big Data Processing

CELEBRATING 30 YEARS
Marlan and Rosemary Bourns College of Engineering

Big Data Frameworks

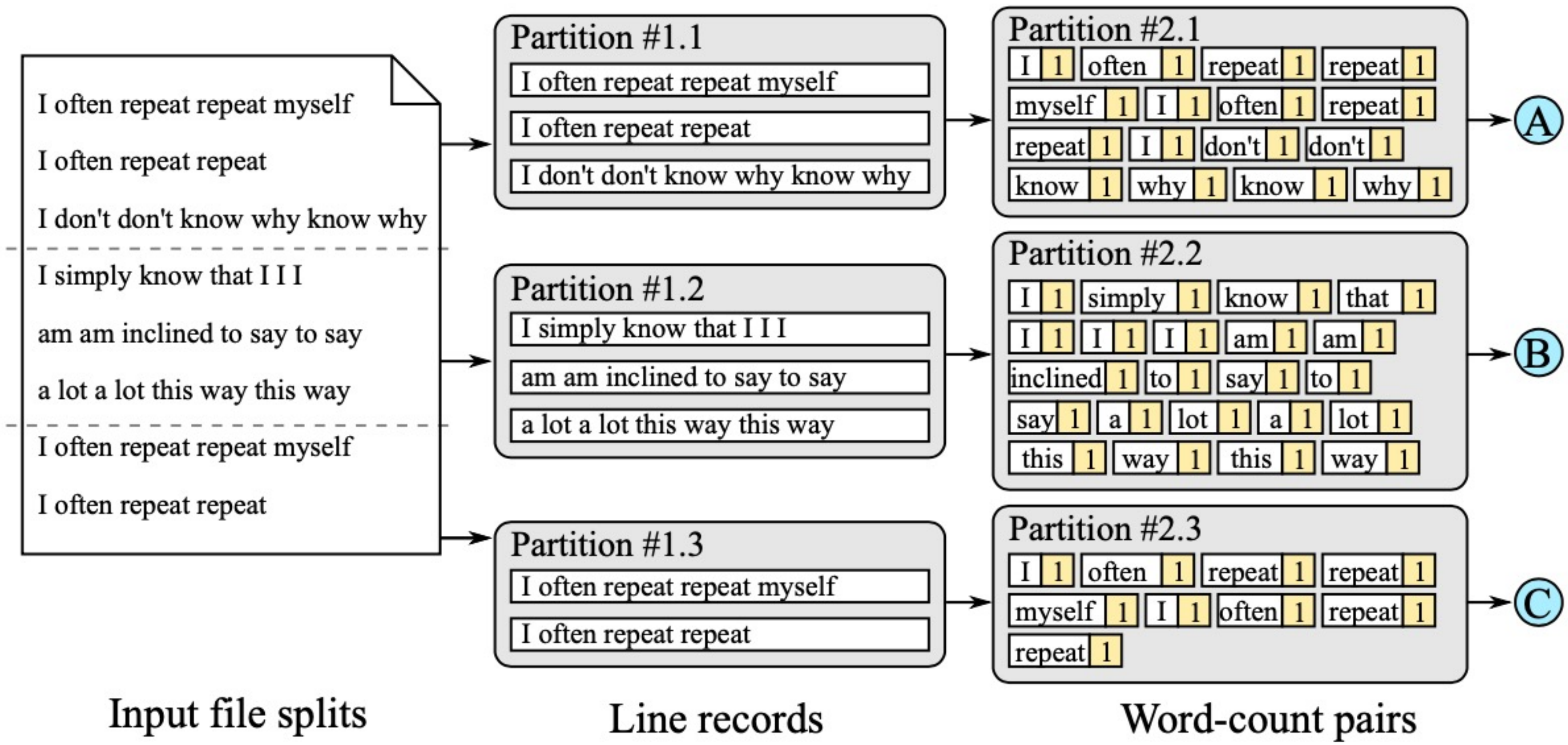
- A system that allows developers to write a program and execute it on a cluster of machines.
- Hides most of the low-level system issues such as fault tolerance, network communication, and load balancing.
- Imposes some restrictions on the developer to ensure that they can run the program efficiently.

Word Count Example

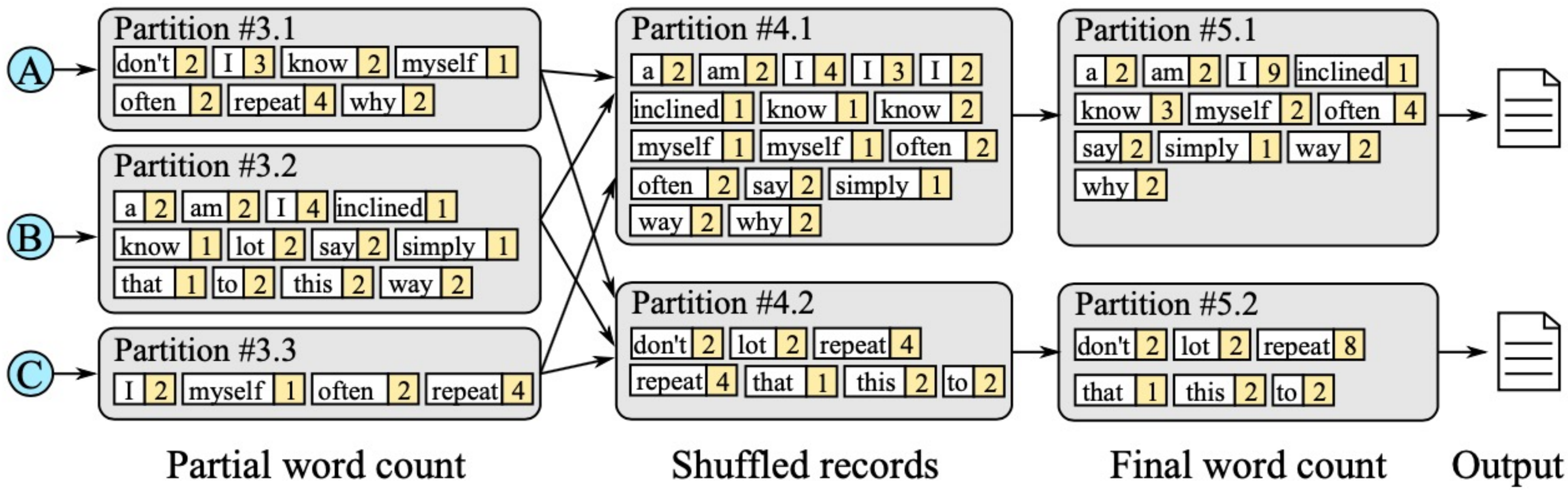
Input.txt
I often repeat repeat myself
I often repeat repeat
I don’t don’t know why know why
I simply know that I I I
am am inclined to say to say
a lot a lot this way this way
I often repeat repeat myself
I often repeat repeat

Word	Count
a	2
am	2
don't	2
I	9
inclined	1
know	3
lot	2
myself	2
often	4
repeat	8
say	2
simply	1
that	1
this	2
to	2
way	2
why	2

Word Count Walkthrough (1/2)



Word Count Walkthrough (2/2)



Word Count Logic

- The logic behind the word count example can be expressed using only two functions
  - WordExtractor: String → {(w, 1)}
  - WordSum: (w,{c}) → (w, Σc)

Complete Word Count in Hadoop

```
public static class TokenizerMapper
   extends Mapper<Object, Text, Text, IntWritable>{
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(Object key, Text value, Context context
  ) {
   StringTokenizer itr = new StringTokenizer(value.toString());
   while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
   }
  }
}

public static class IntSumReducer
   extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();
  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
  ) {
   int sum = 0;
   for (IntWritable val : values) {
    sum += val.get();
   }
   result.set(sum);
   context.write(key, result);
  }
}
```

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Source: https://hadoop.apache.org/docs/r3.2.2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0

Complete Word Count in Spark

```scala
// In Scala shell
val lines = sc.textFile("data.txt")
val pairs = lines.flatMap(s => s.split("\\b"))
  .map(w => (w,1))
val counts = pairs.reduceByKey((a, b) => a + b)
counts.saveAsTextFile("word_count_output.txt")
```

Big-data Processing

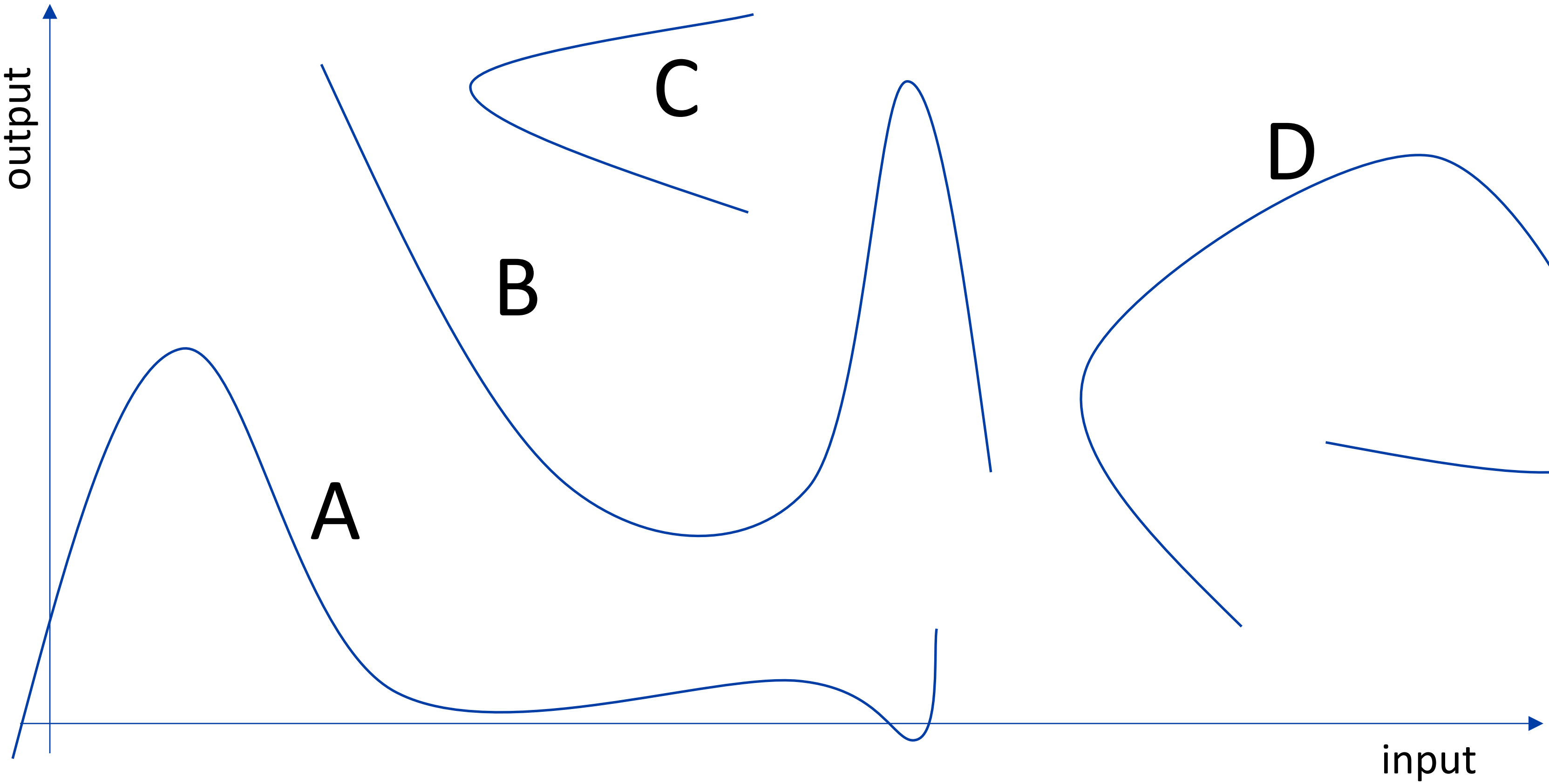
- Programming Model
  - How a developer writes a big-data program
- Application Model (Logical Model)
  - How the big-data platform internally represents a user program
- Execution Model (Physical Model)
  - How the program gets executed on the cluster

Functional Programming Model

- A big-data program consists of user-defined functions
  - E.g., Map and Reduce functions in a Hadoop MapReduce program
- A valid function must satisfy two constraints
  - Sateless/Memoryless
  - Deterministic

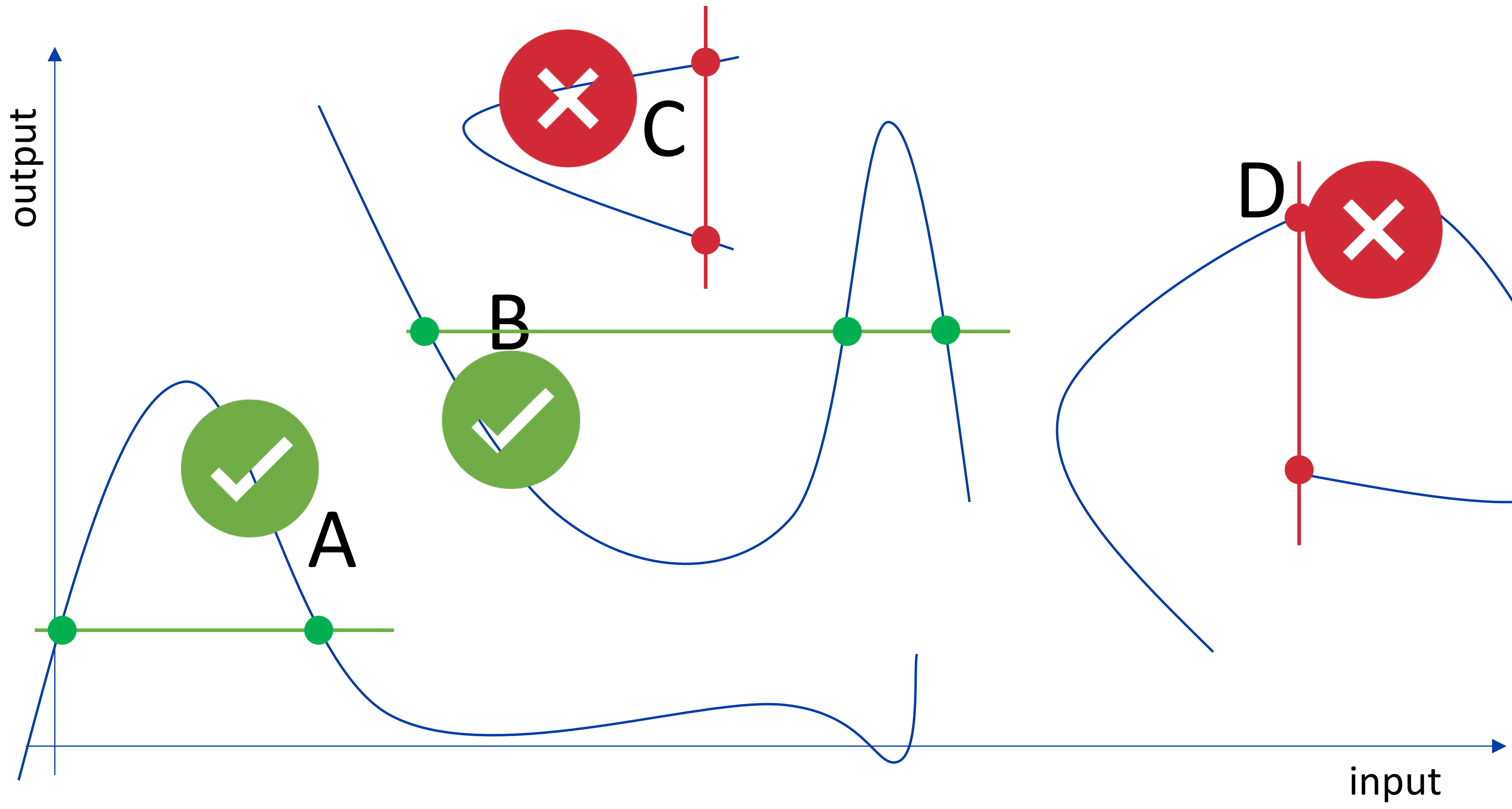
Functional Programming

- Which of these are functions?



Functional Programming

- Which of these are functions?



Word Count Functions

- Word Extractor(Line: String) {
  words = Line.split
  foreach (w ∈ words) output.write(w, 1)
  }
- SumWords(word: String, counts: int[]) {
  sum = sum(counts)
  output.write(word, sum)
  }

Examples

```
Function1(x) {
  return x + 5;
}
```

```
Int sum
Function2(x) {
  sum += x;
  return sum;
}
```

```
RNG random;
Function3(x) {
  random.randomInt(0, x);
}
```

```
Map<String, Int> lookuptable;
Function4(x) {
  return lookuptable.get(x);
}
```

Examples

```
Function1(x) {
  return x + 5;
}
```

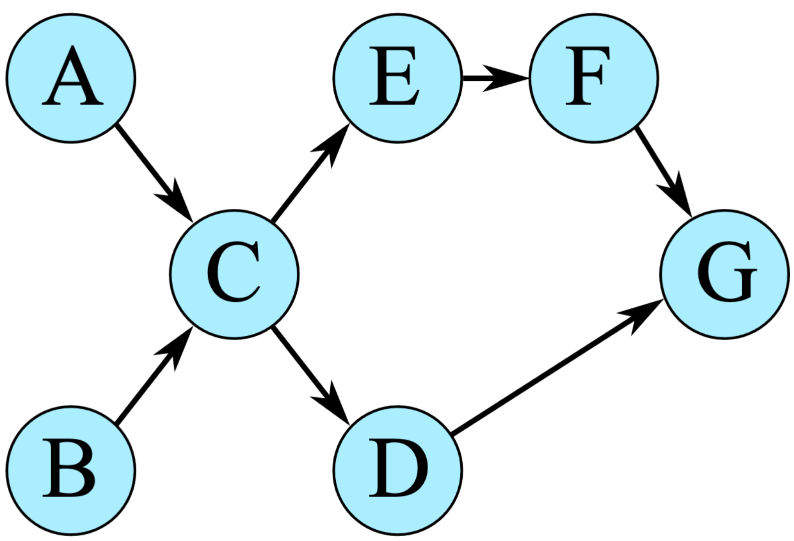
```
Int sum
Function2(x) {
  sum += x;
  return sum;
}
```

```
RNG random;
Function3(x) {
  random.randomInt(0, x);
}
```

```
Map<String, Int> lookuptable;
Function4(x) {
  return lookuptable.get(x);
}
```

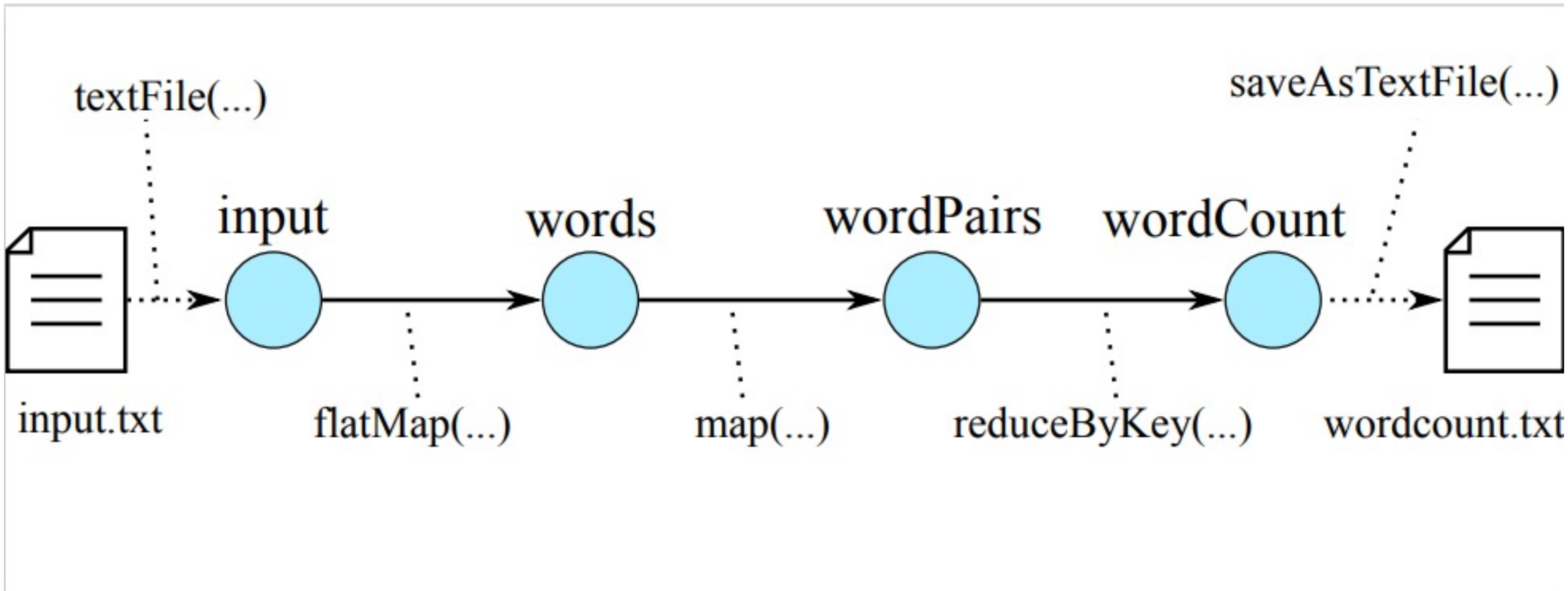
Directed Acyclic Graph

- The functional programming paradigm allows the developer to define one function
- The program consists of multiple functions



DAG for Word Count in Spark

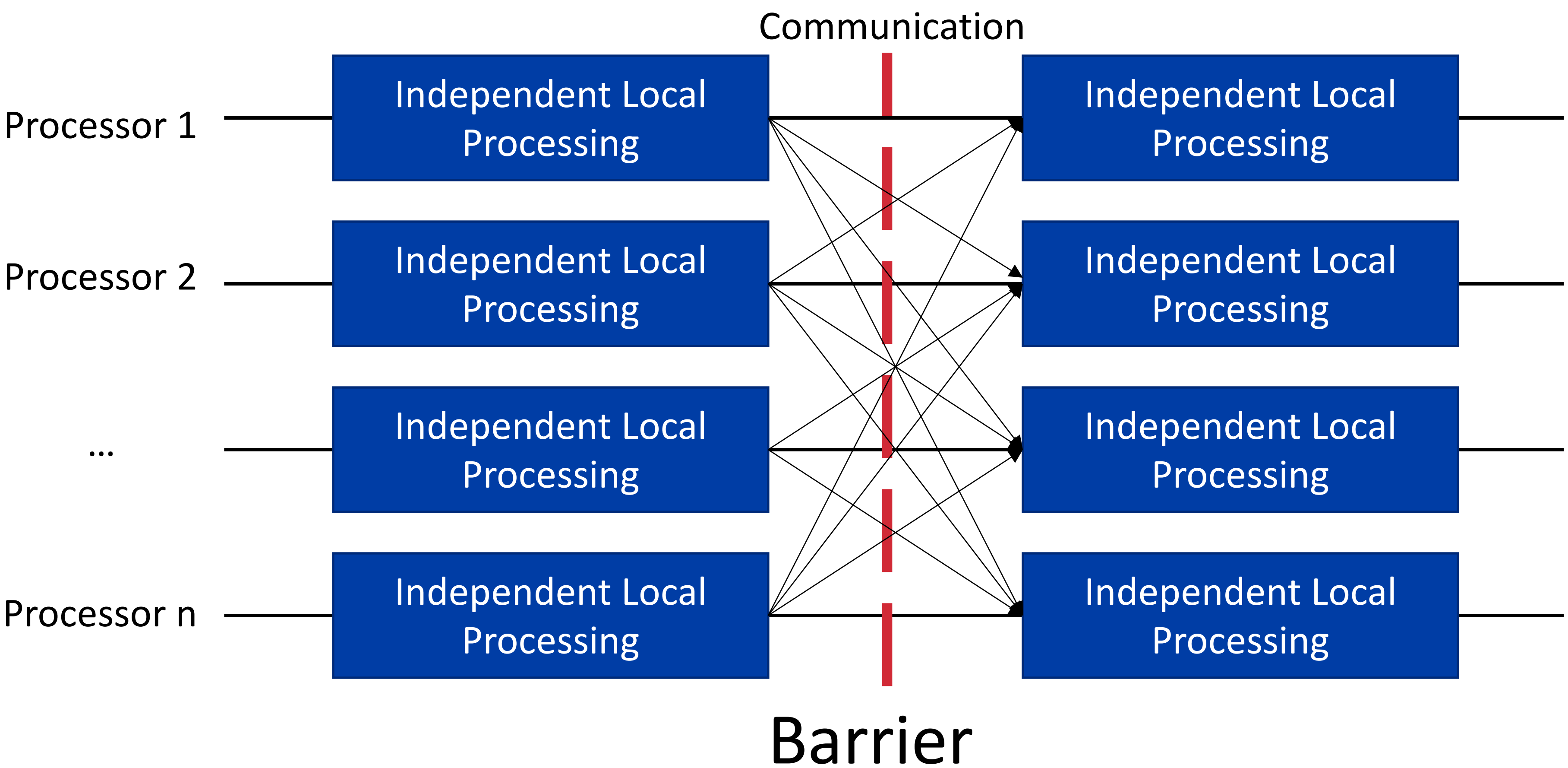
```
// In Scala shell
val lines = sc.textFile("data.txt")
val pairs = lines.flatMap(s => s.split("\\b"))
  .map(w => (w,1))
val counts = pairs.reduceByKey((a, b) => a + b)
counts.saveAsTextFile("word_count_output.txt")
```



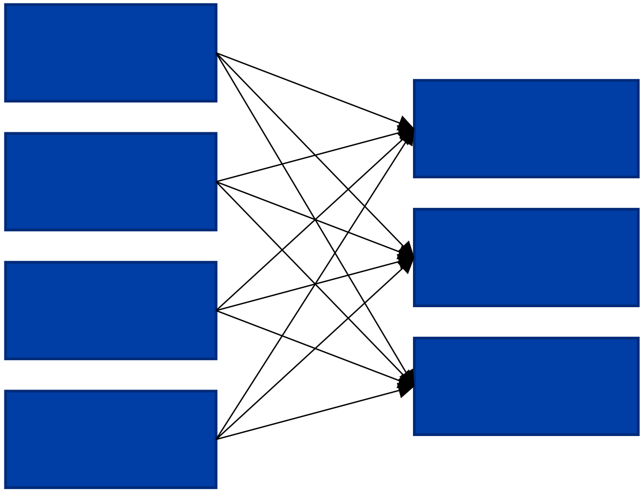
Bulk Synchronous Parallel (BSP)

- The BSP model is how big-data frameworks execute a program
- The model splits the execution into stages of local processing
- Computation stages are separated by a communication barrier

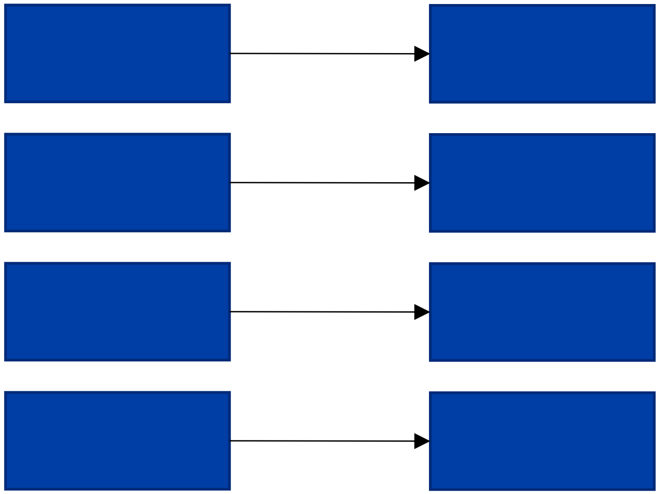
BSP Model


Communication
Processor 1
Processor 2
…
Processor n
Independent Local Processing
Independent Local Processing
Independent Local Processing
Independent Local Processing
Independent Local Processing
Independent Local Processing
Independent Local Processing
Independent Local Processing
Barrier

Communication Patterns

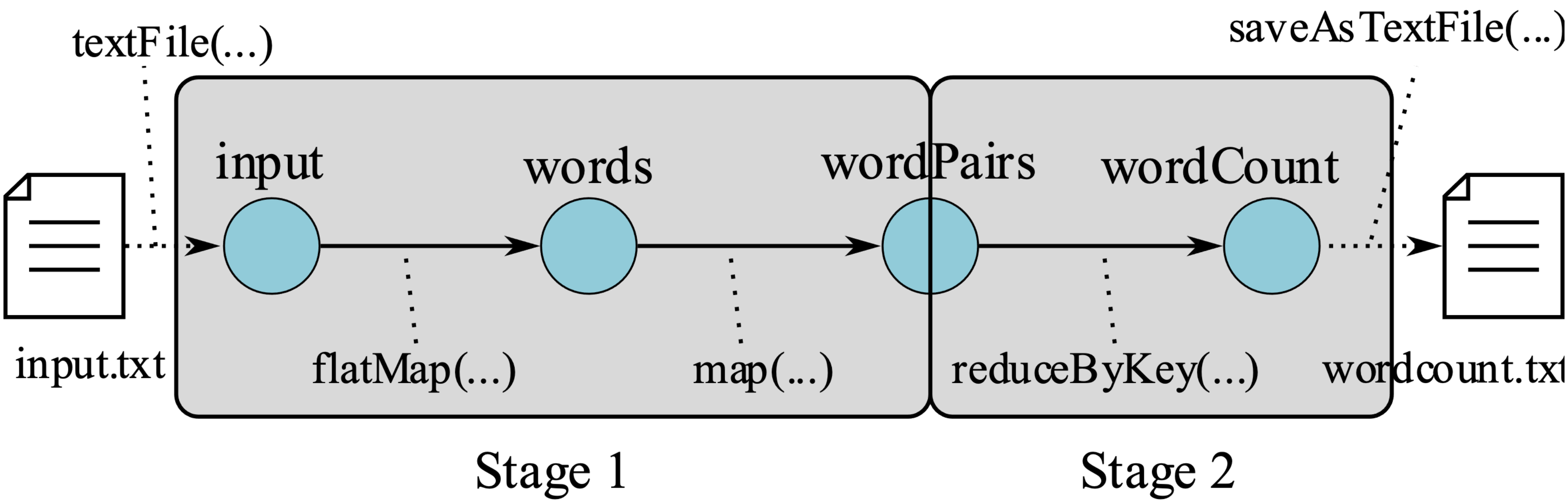


Fully Connected
(Requires network communication)

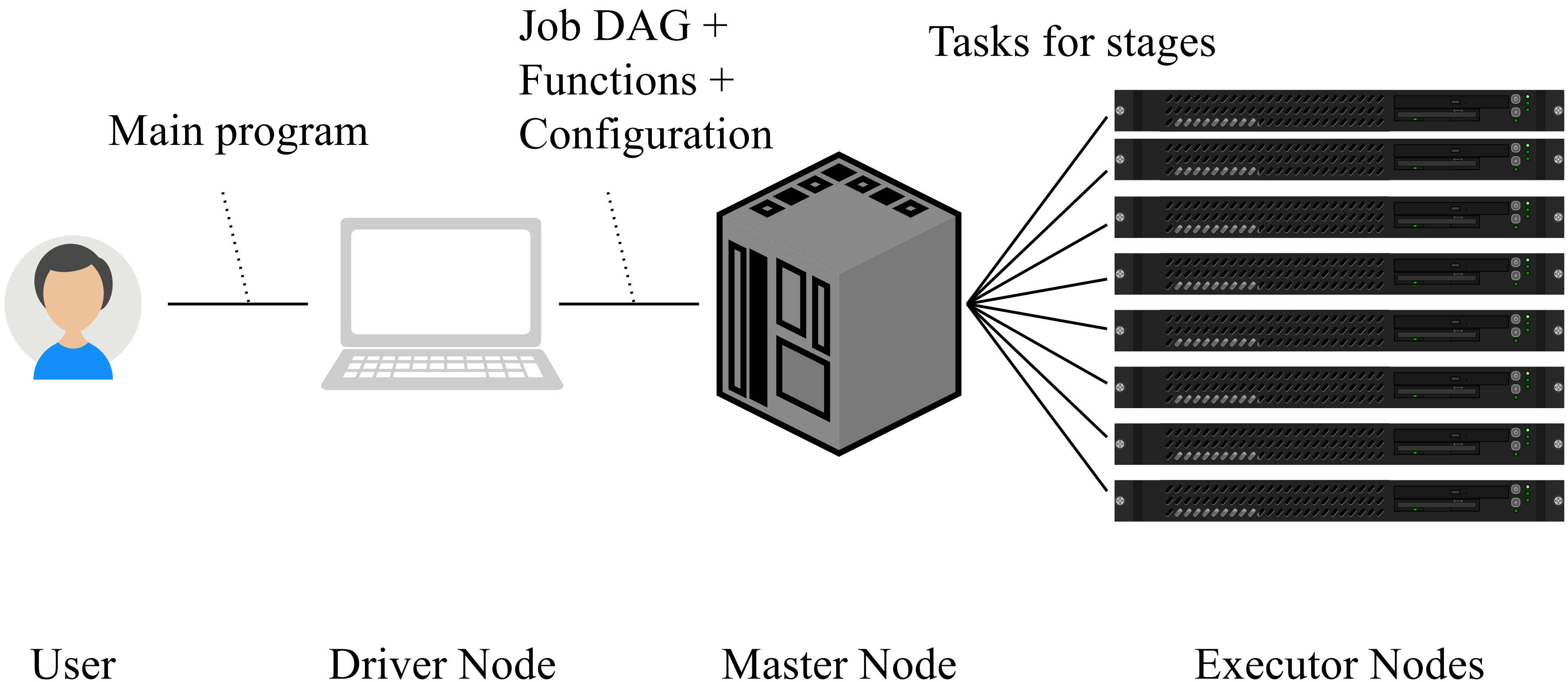


One-to-one
(Can be done locally in one stage)

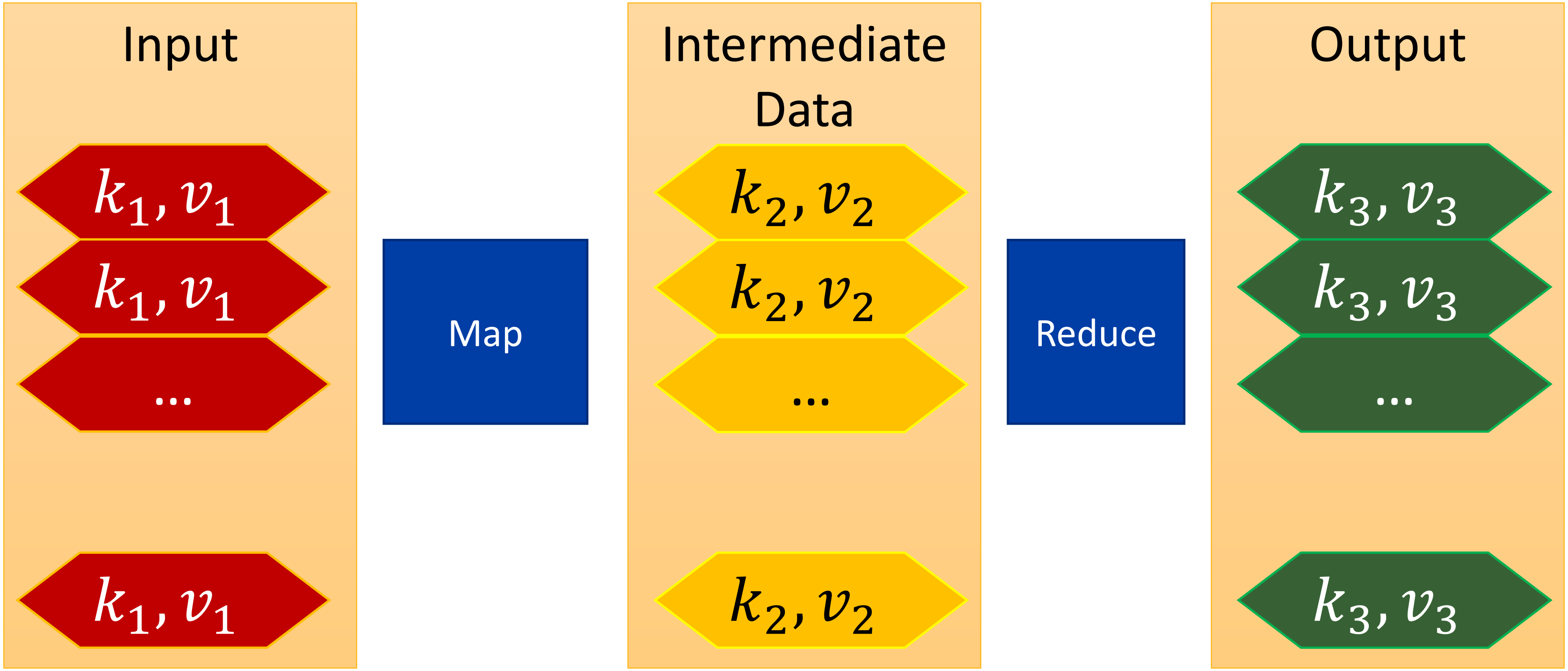
Word Count Stages



Architecture of Big-data Frameworks



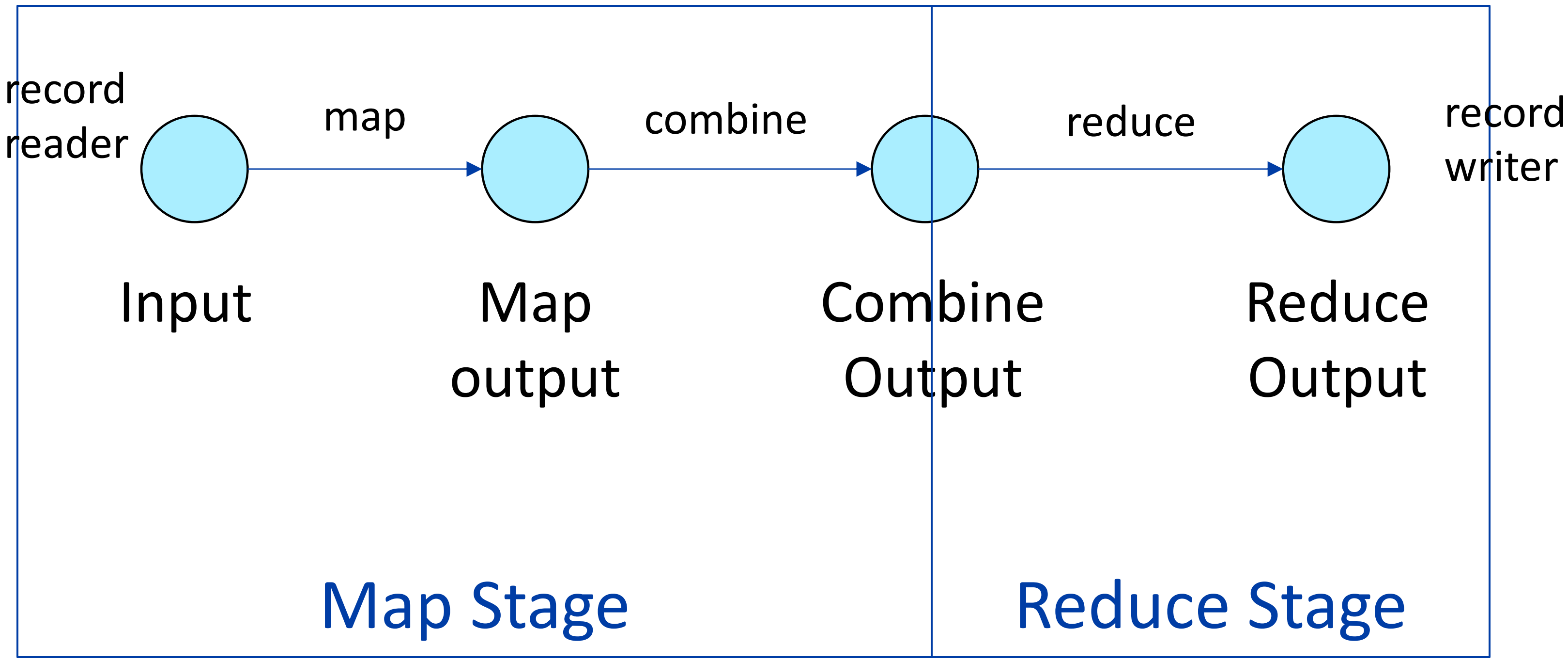
Hadoop MapReduce



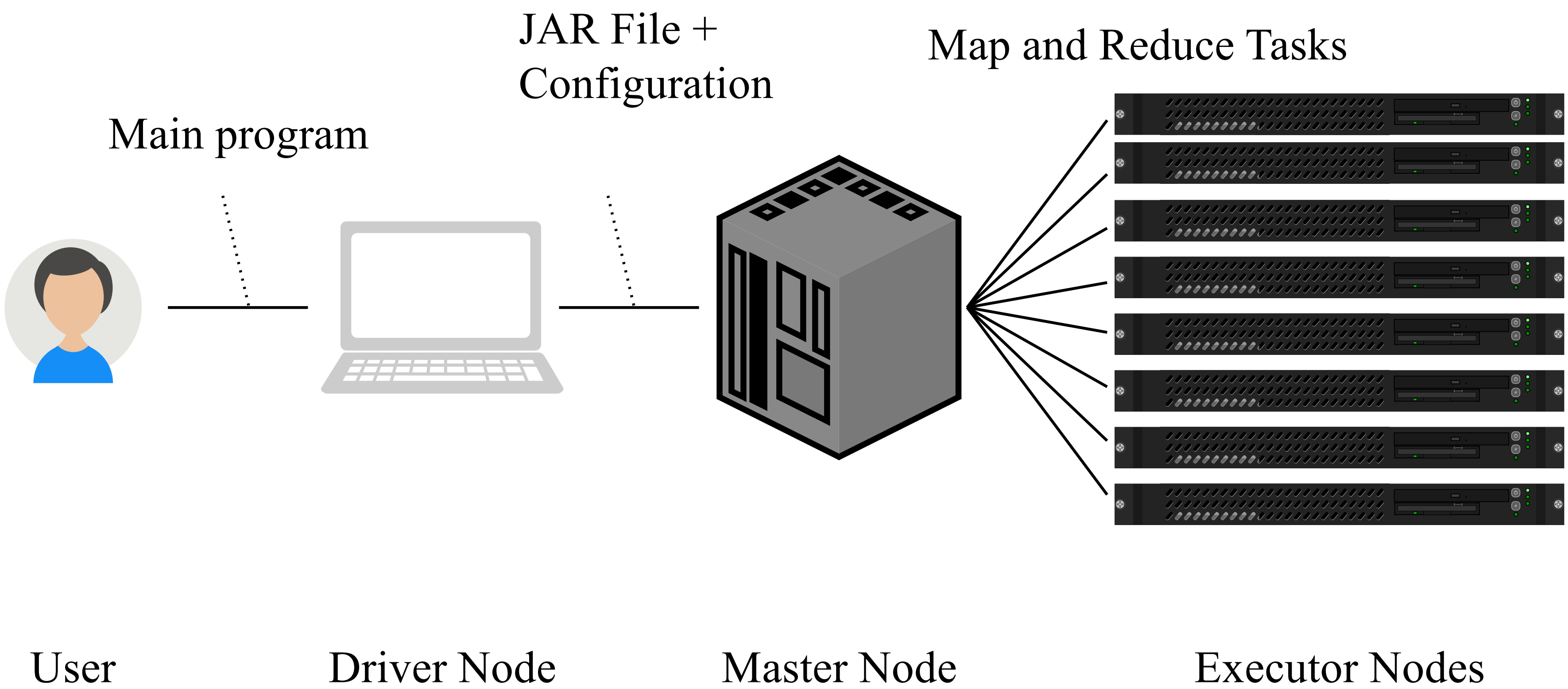
Map and Reduce Functions

- Map Function
  - Maps a single input record to a set (possibly empty) of intermediate records
  - Map: $\langle k_1, v_1 \rangle \rightarrow \{\langle k_2, v_2 \rangle\}$
- Reduce Function
  - Reduces a set of intermediate records with the same key to a set (possibly empty) of output records
  - Reduce: $\langle k_2, \{v_2\} \rangle \rightarrow \{\langle k_3, v_3 \rangle\}$
- Combine Function (Optional)
  - Partial local reduction before reduce
  - Combine: $\langle k_2, \{v_2\} \rangle \rightarrow \{\langle k_2, v_2 \rangle\}$

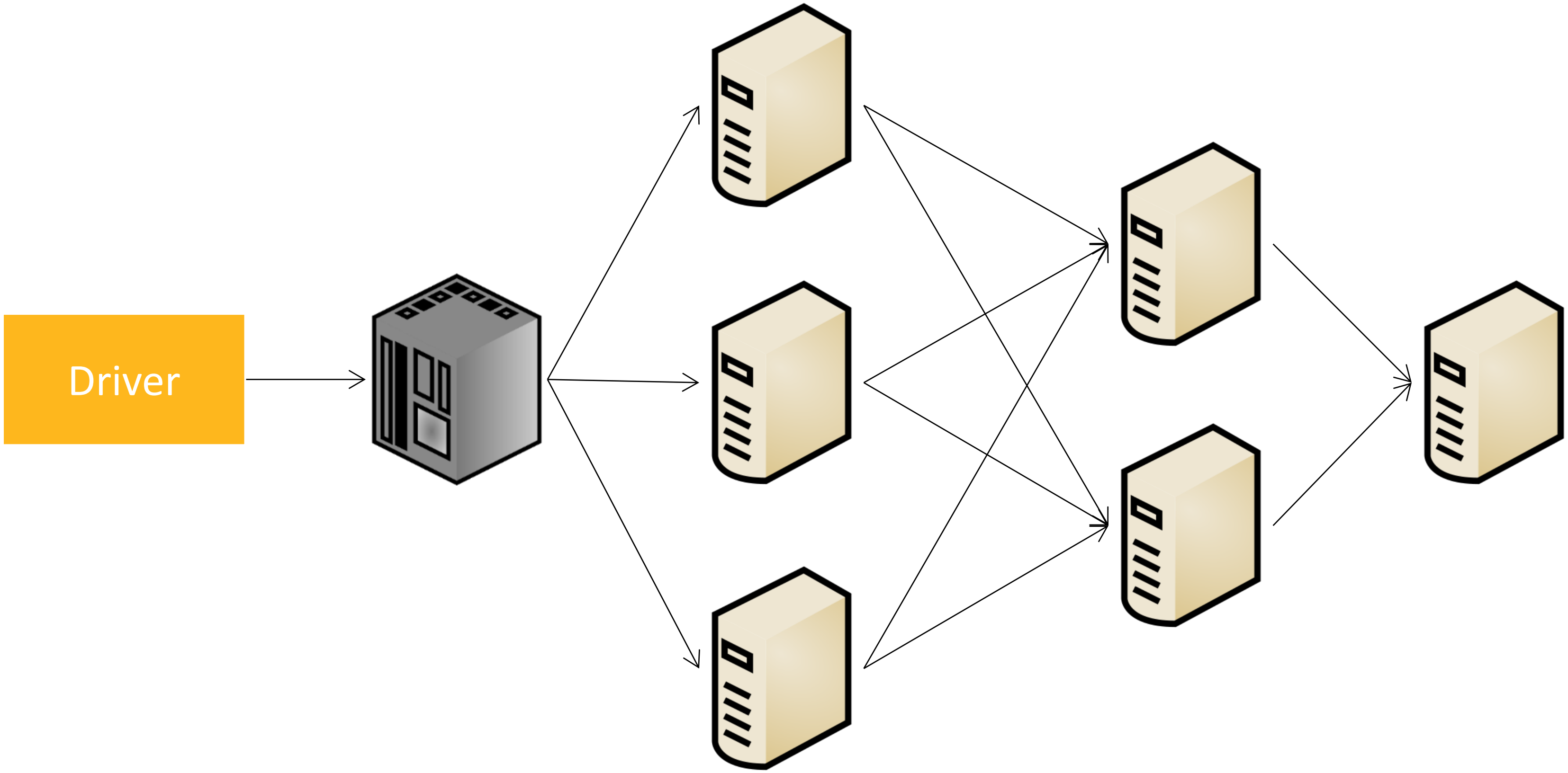
MapReduce DAG



MapReduce Program Cycle



Job Execution Overview



Job submission

Job preparation

Map

Shuffle

Reduce

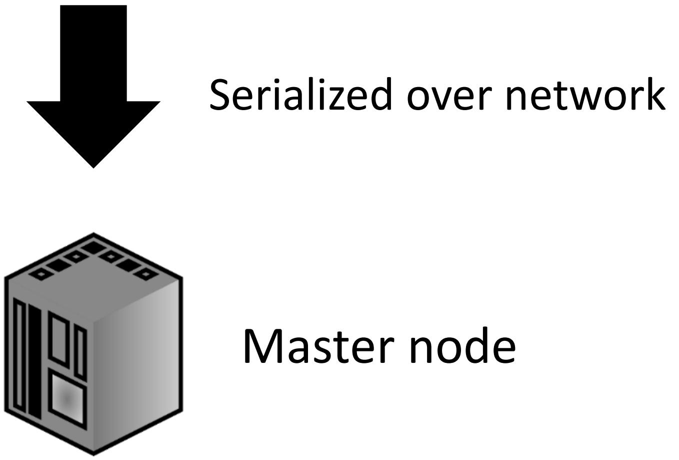
Cleanup

Job Submission

- Execution location: Driver node
- A driver machine should have the following
  - Compatible Hadoop binaries
  - Cluster configuration files
  - Network access to the master node
- Collects job information from the user
  - Input and output paths
  - Map, reduce, and any other functions
  - Any additional user configuration
- Packages all this in a Hadoop Configuration

Hadoop Configuration

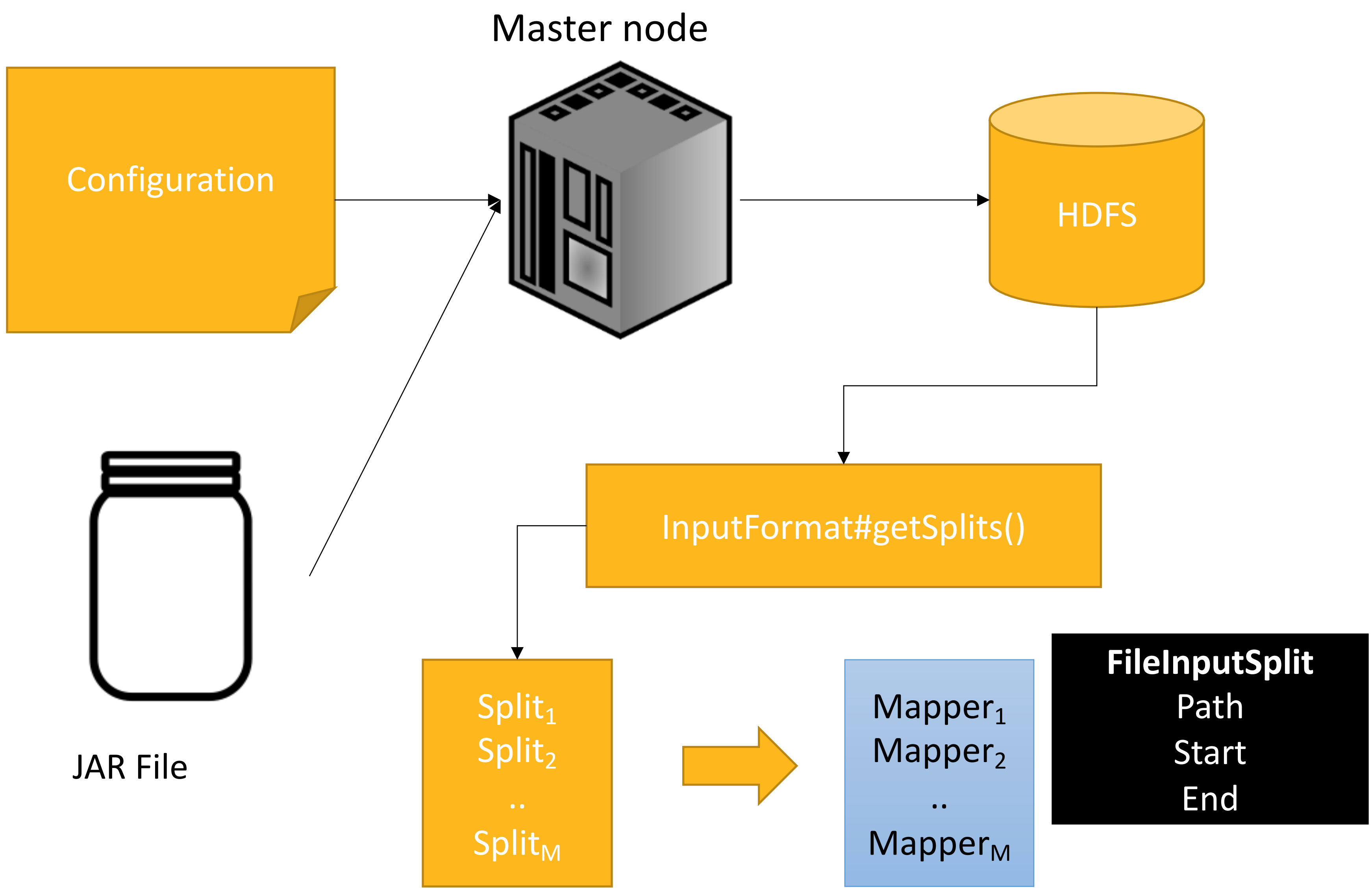
Key: String	Value: String
Input	hdfs://user/eldawy/README.txt
Output	hdfs://user/eldawy/wordcount
Mapper	edu.ucr.cs.cs167.eldawy.WordCount
Reducer	…
JAR File	…
User-defined	User-defined



Job Preparation

- Runs on the master node
- Gets the job ready for parallel execution
- Collects the JAR file that contains the user-defined functions, e.g., Map and Reduce
- Writes the JAR and configuration to HDFS to be accessible by the executors
- Looks at the input file(s) to decide how many map tasks are needed
- Makes some sanity checks
- Finally, it pushes the BRB (Big Red Button)

Job Preparation



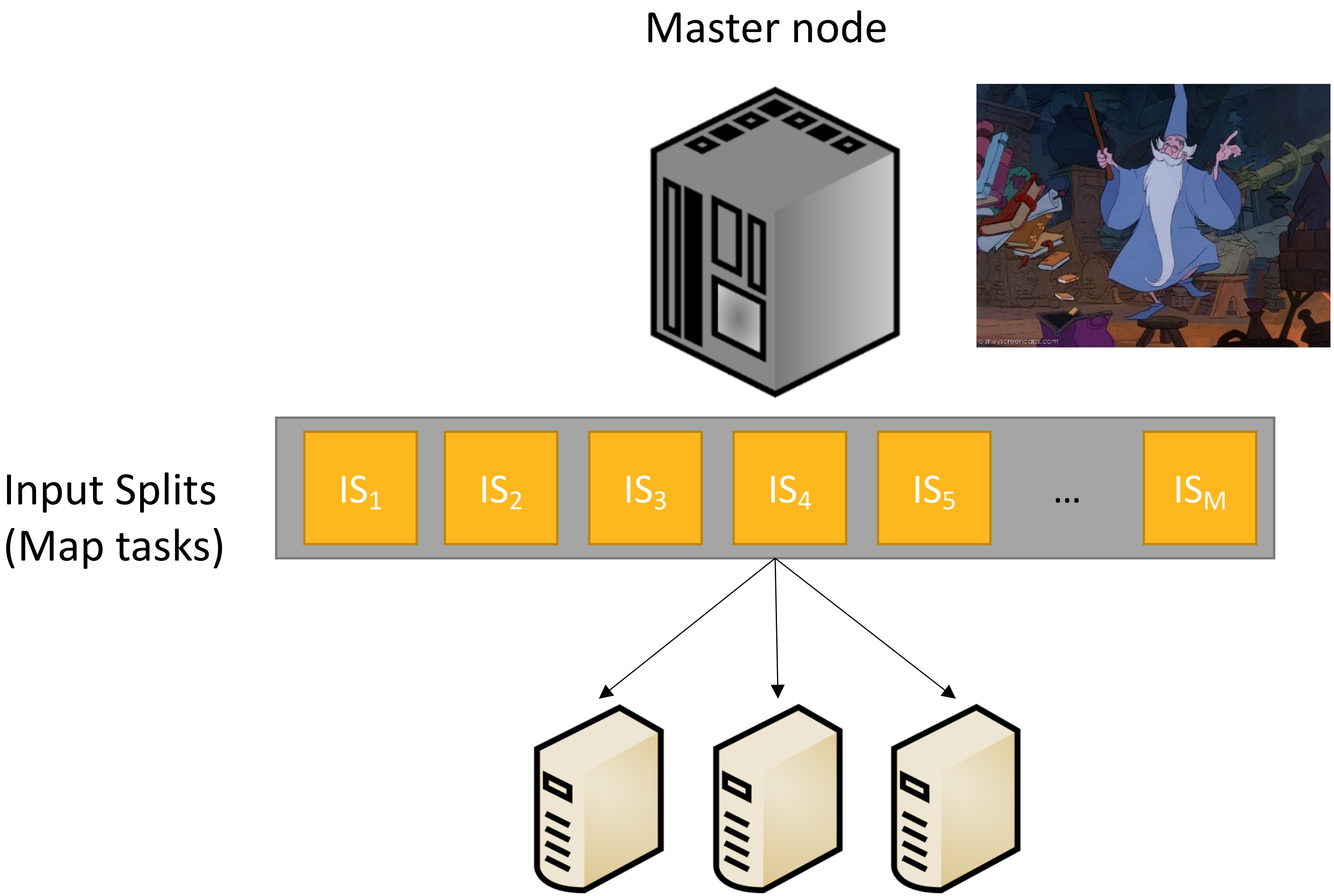
Map Phase

- Runs in parallel on worker nodes
- M Mappers:
  - Read the input
  - Apply the map function
  - Apply the combine function (if configured)
  - Store the map output
- There is no guaranteed ordering for processing the input splits

Input record reader

- Split the file based on the file metadata
  - File size, block sizes, # of nodes
- Each split is defined by:
  - File name, Start offset, Length
- For each split:
  - Seek to the start offset
  - Skip the first record (except for the first split)
  - Read until the beginning of the record goes beyond the start + length

Map Phase



Map Task

- Reads the job configuration and task information (mainly, InputSplit)
- Instantiates an object of the Mapper class
- Instantiates a record reader for the assigned input split
- Reads records one-by-one from the record reader and passes them to the user-defined map function
- Passes the map output to the next step

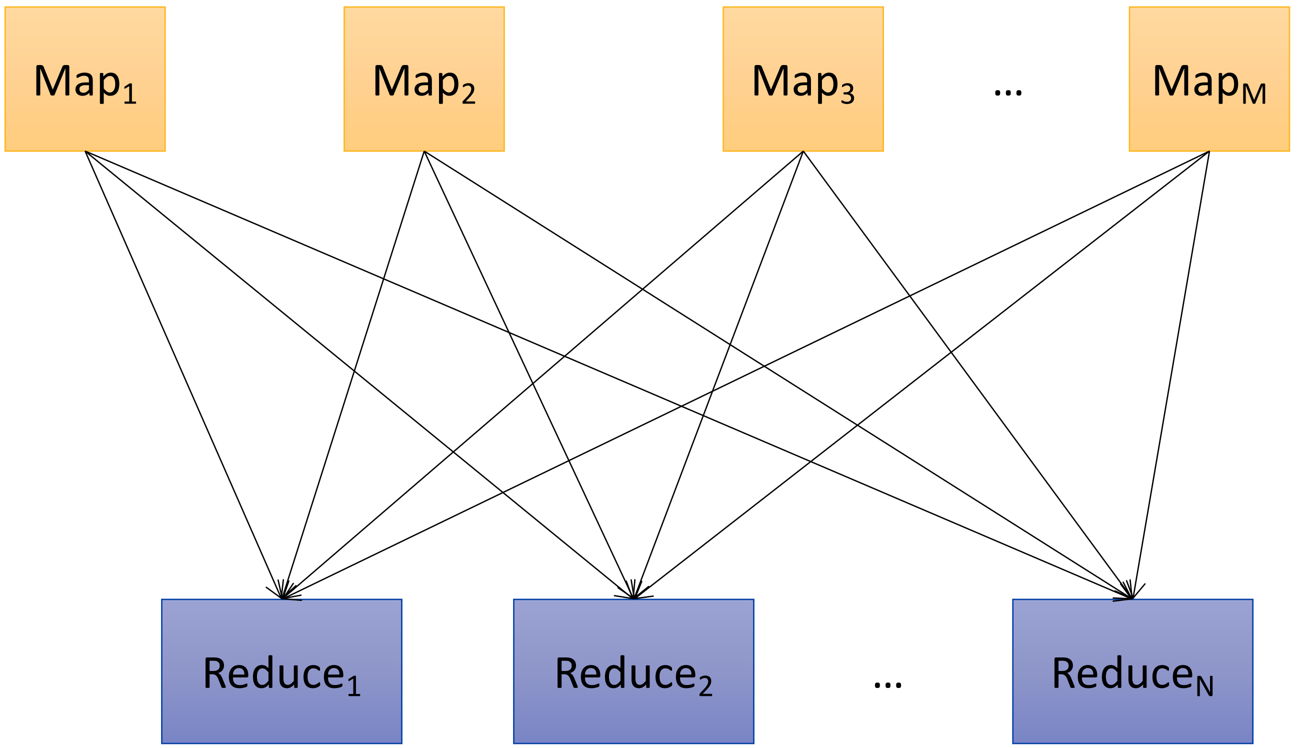
Map output

- What happens to the map output?
- It depends on the number of reducers
  - 0 reducers: Map output is written directly to HDFS as the final answer
  - 1+ reducers: Map output is passed to the shuffle phase

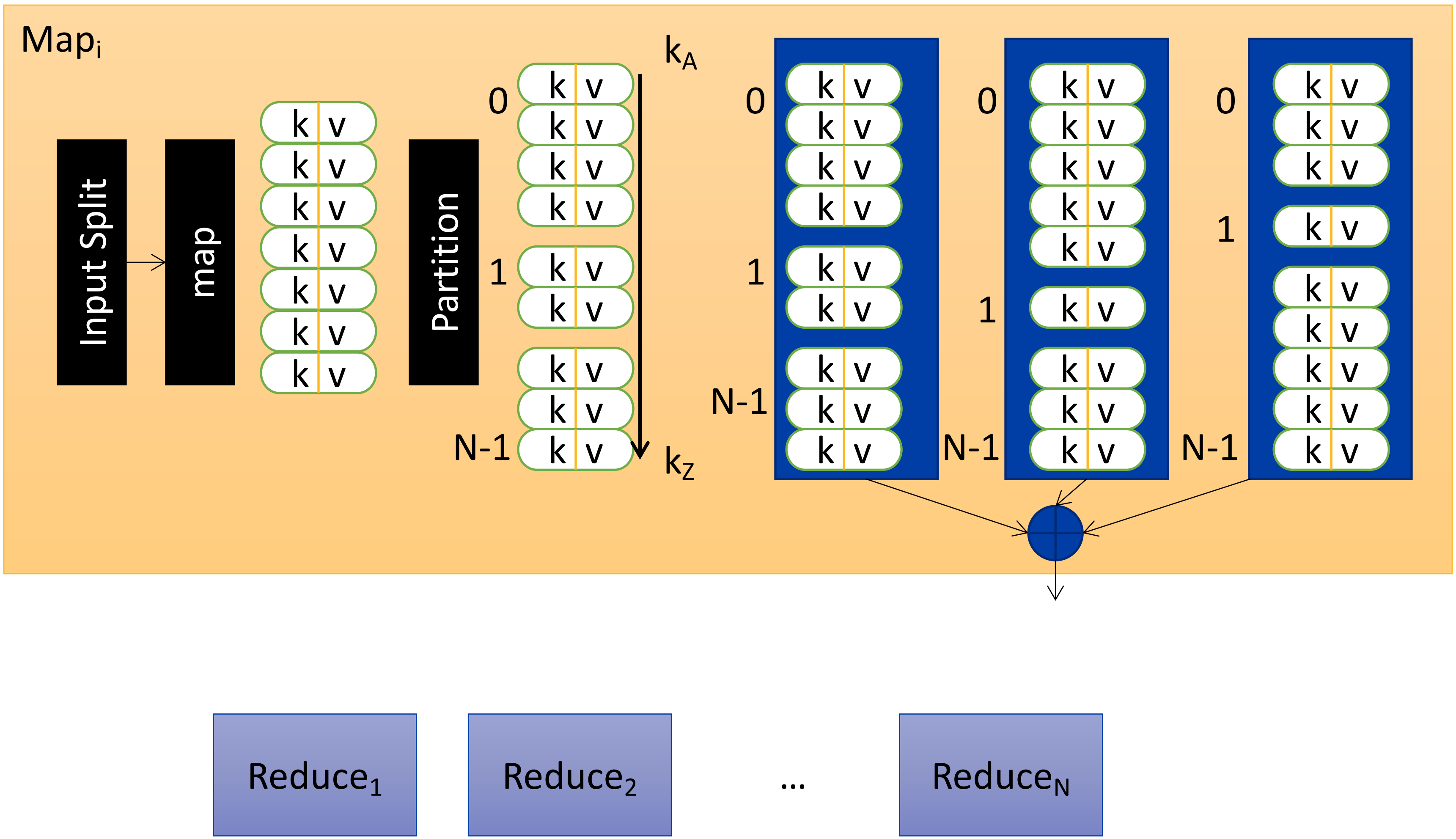
Shuffle Phase

- Executed only in the case of one or more reducers
- Transfers data between the mappers and reducers
- Groups records by their keys to ensure local processing in the reduce phase

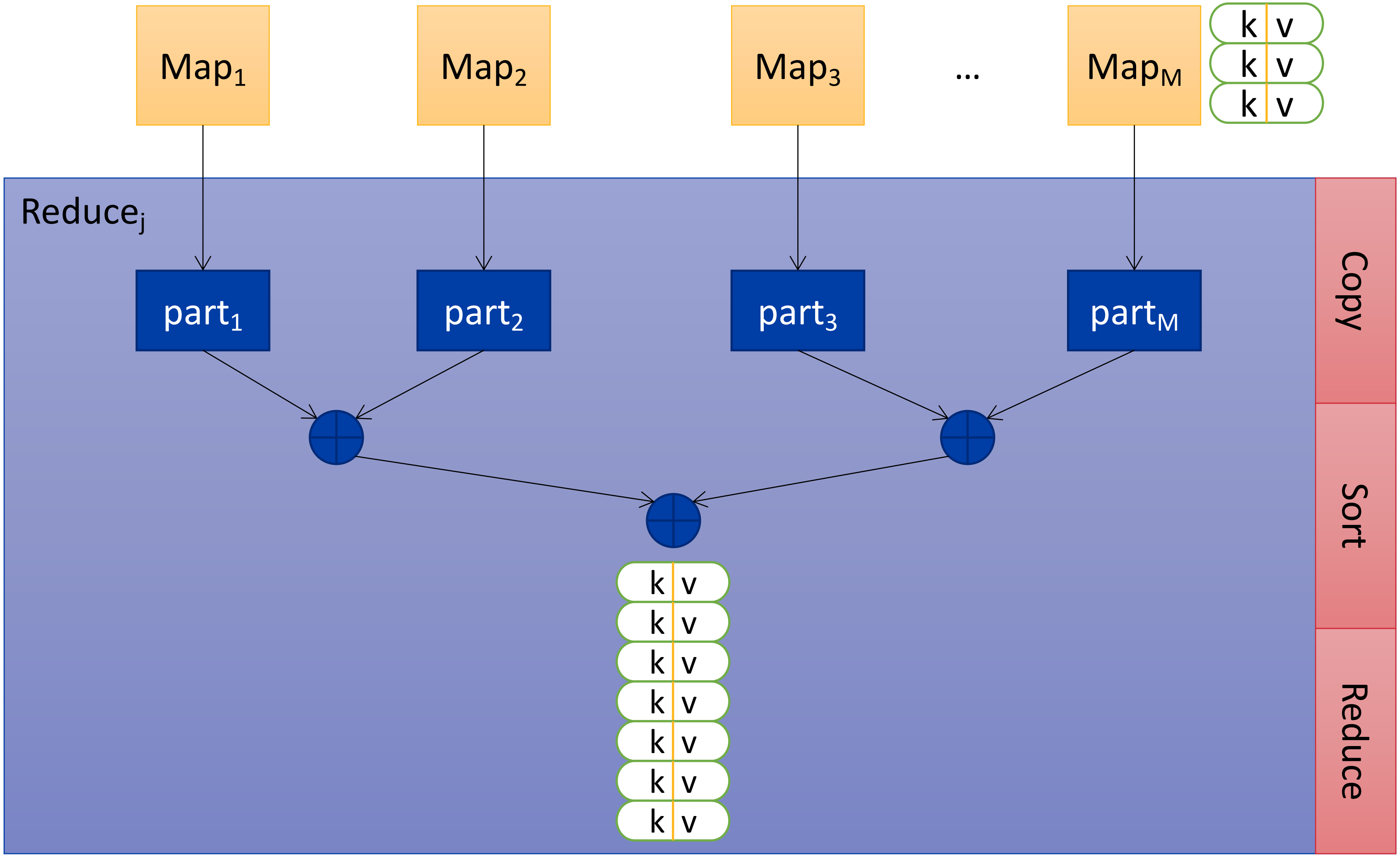
Shuffle Phase



Shuffle Phase (Map-side)

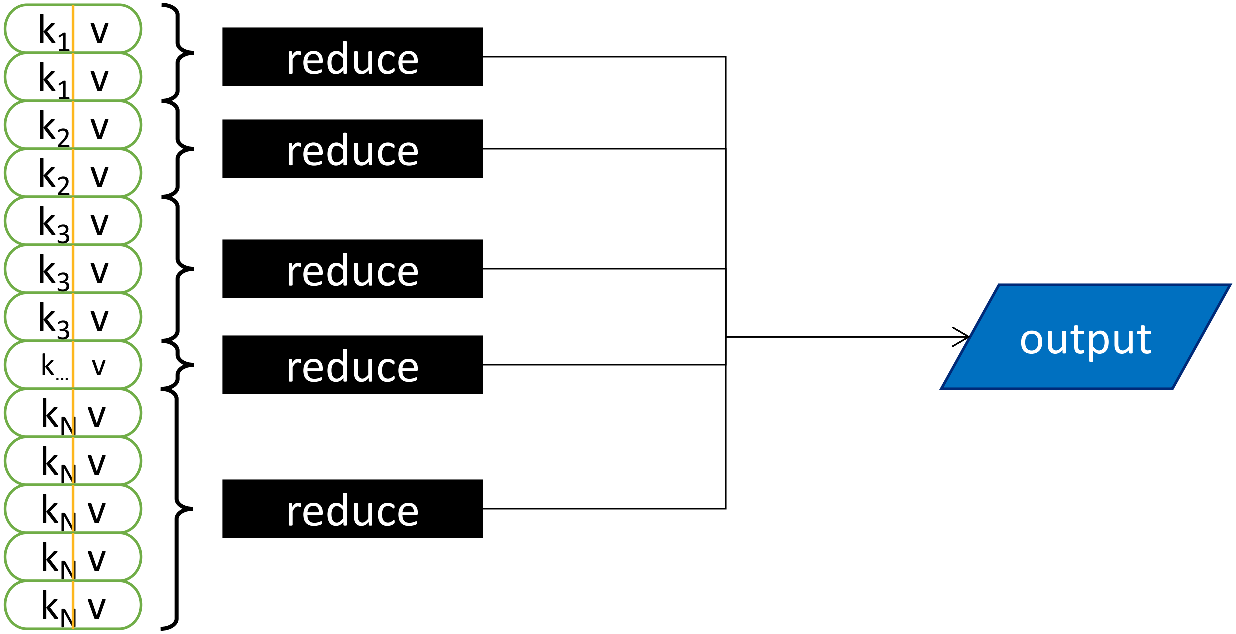


Shuffle Phase (Reduce-side)



Reduce Phase

- Apply the reduce function to each group of similar keys



Output Writing

- Materializes the final output to disk
- All results are from one process (mapper/reducer) are stored in a subdirectory
- An OutputFormat is used to
  - Create any files in the output directory
  - Write the output records one-by-one to the output
  - Merge the results from all the tasks (if needed)
- While the output writing runs in parallel, the final commit step runs on a single machine

Hadoop MapReduce Conclusion

- A MapReduce program consists of a map, a reduce, and optionally a combine function
- Hadoop distributes the program to all executor nodes
- The input is partitioned and each input split is processed independently
- The intermediate data is shuffled and reduced to produce the final output.

Spark Resilient Distributed Dataset (RDD)

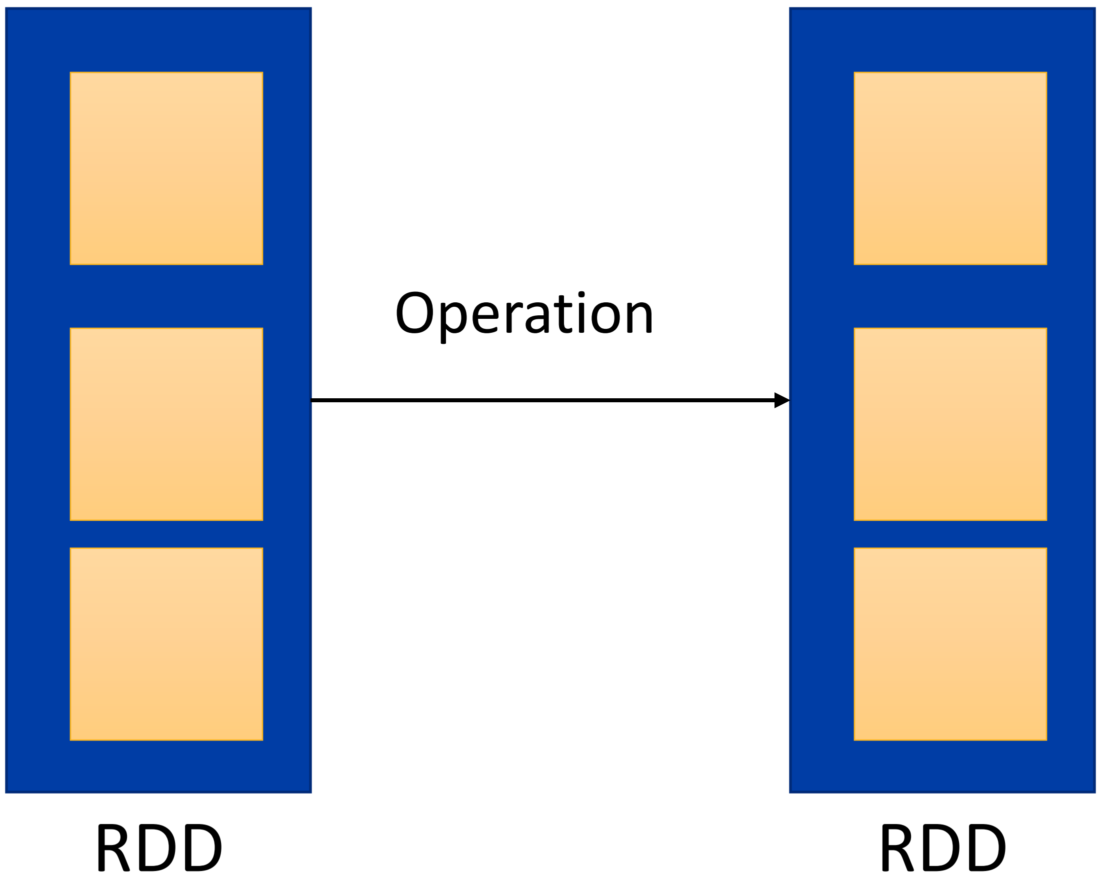
Spark

- A memory-based big-data framework
- Resilient Distributed Dataset (RDD) is an alternative query processing framework for big-data
- Utilizes more memory to speed up query processing

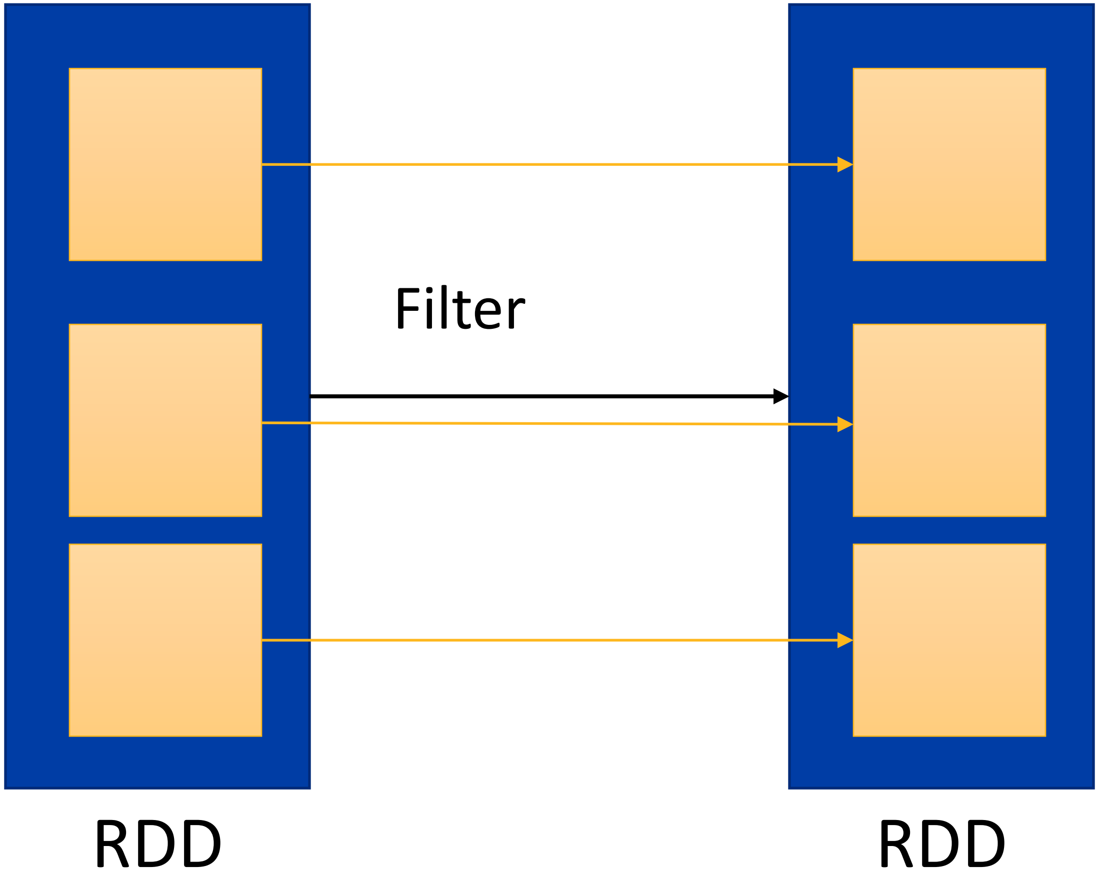
RDD Abstraction

- RDD is a pointer to a distributed dataset
- Stores information about how to compute the data or where the data is
- Transformation: Converts an RDD to another RDD
- Action: Returns an answer of an operation over an RDD

RDD

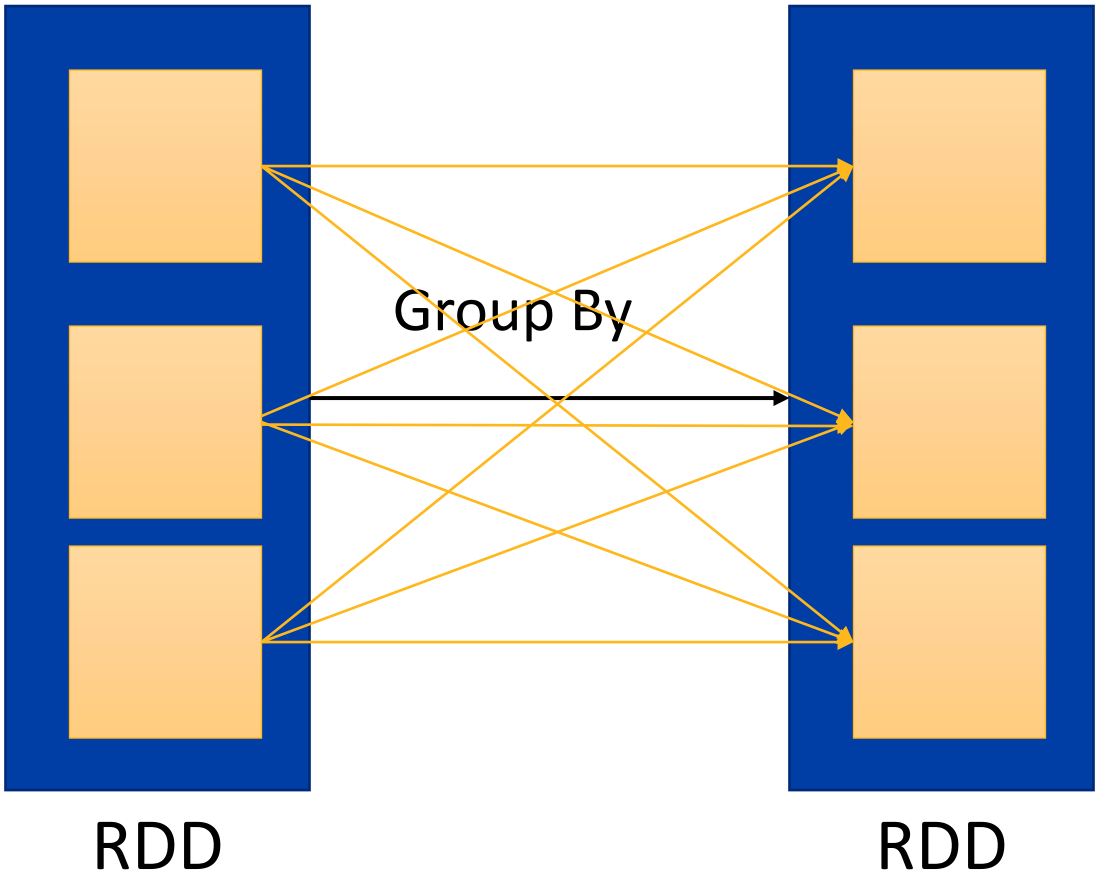


Filter Operation



Similar to the map operation

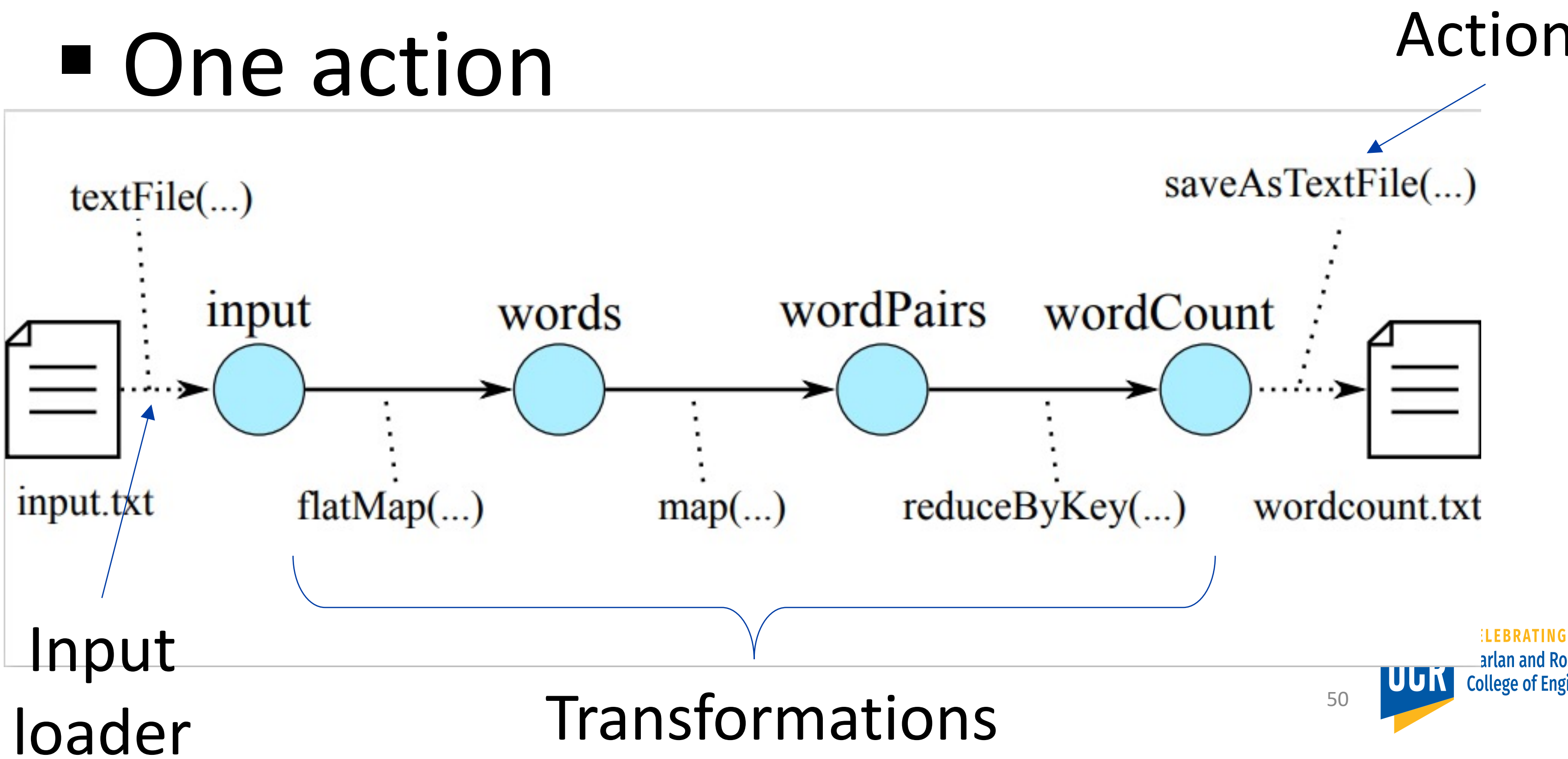
GroupBy (Shuffle) Operation



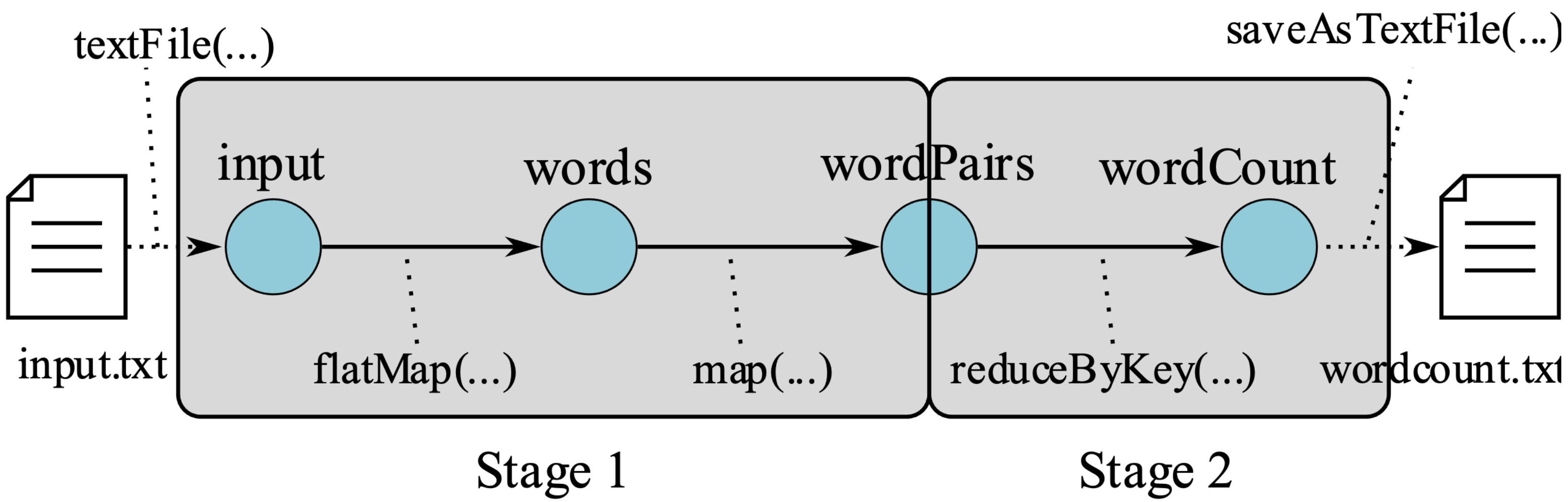
Similar operation **Join**

Application DAG

- A complete DAG consists:
  - One or more input loader
  - Zero or more transformations
  - One action



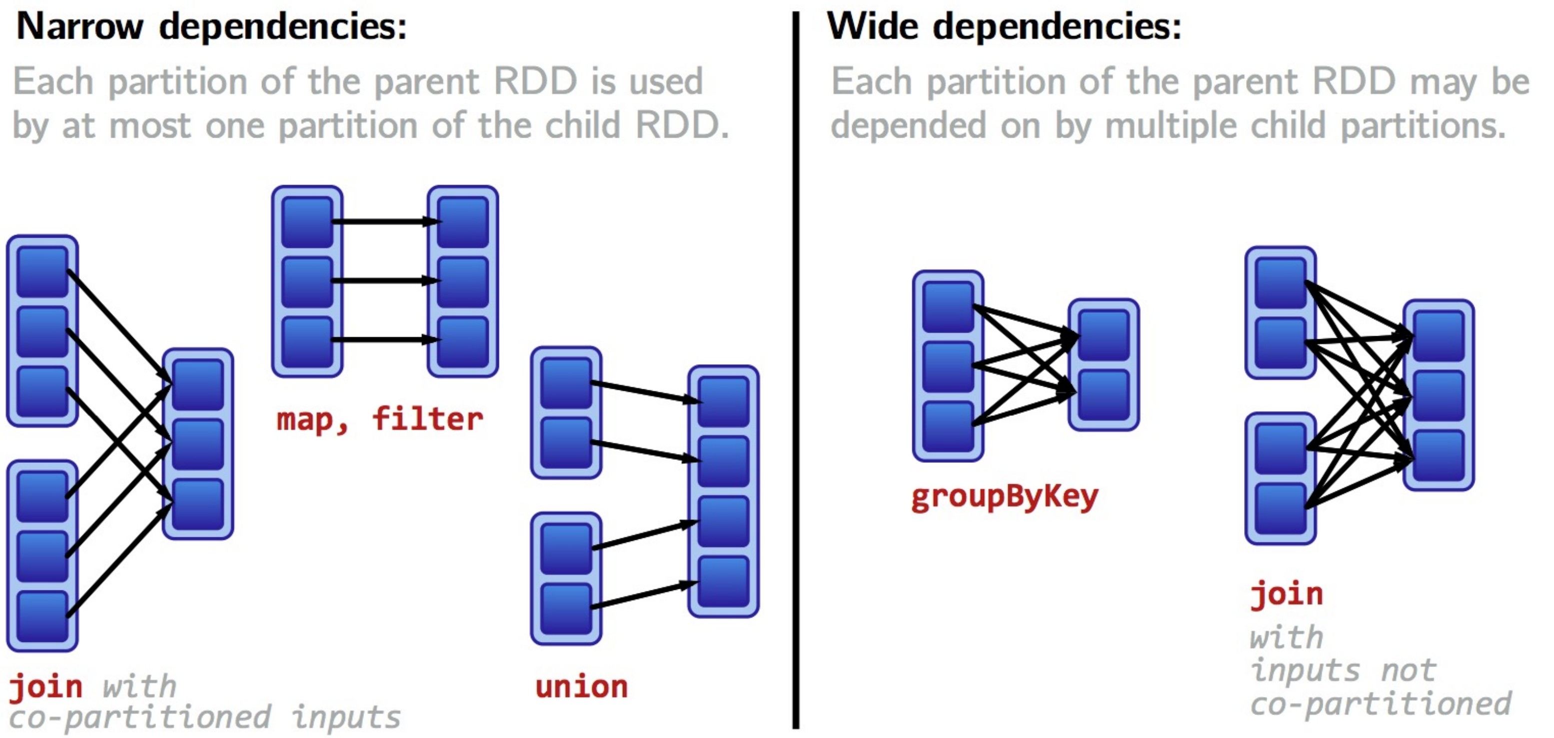
DAG Execution using BSP



How does Spark split a DAG into stages?

Types of Dependencies

- Narrow dependencies
- Wide dependencies



Source: https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies

Spark RDD Features

- Lazy execution: Collect transformations and execute on actions
- Lineage tracking: Keep track of the lineage of each RDD for fault-tolerance
- Resiliency: When an in-memory partition gets lost, Spark recomputes it

Examples of Transformations

- map
- mapToPair
- flatMap
- reduceByKey
- filter
- sample
- join
- union
- partitionBy

Examples of Actions

- count
- collect
- save(path)
- persist
- reduce

RDD Operations

- Spark is richer than Hadoop in terms of operations
- Sometimes, you can do the same logic with more than one way
- In the following part, we will explain how different RDD operations work
- The goal is to understand the performance implications of these operations and choose the most efficient one

Java Examples

- Apache Spark homepage
  - https://spark.apache.org

```
# Initialize the Spark context
JavaSparkContext spark =
    new JavaSparkContext("local", "CS167-Demo");
```

Examples

```
# Initialize the Spark context
JavaSparkContext spark =
    new JavaSparkContext("local", "CS167-Demo");


# Hello World! Example. Count the number of lines in the file
JavaRDD<String> textFileRDD =
                  spark.textFile("nasa_19950801.tsv");
long count = textFileRDD.count();
System.out.println("Number of lines is "+count);
```

Examples

```
# Count the number of OK lines (response code 200)
JavaRDD<String> okLines = textFileRDD.filter(new
Function<String, Boolean>() {
    @Override
    public Boolean call(String s) throws Exception {
        String code = s.split("\t")[5];
        return code.equals("200");
    }
});
long count = okLines.count();
System.out.println("Number of OK lines is "+count);
```

Examples

```
# Count the number of OK lines (response code 200)
# Shorten the implementation using lambdas (Java 8 and above)
JavaRDD<String> okLines =
 textFileRDD.filter(s -> s.split("\t")[5].equals("200"));


long count = okLines.count();
System.out.println("Number of OK lines is "+count);
```

Examples

```
# Make it parametrized by taking the response code as a
command line argument
String inputFileName = args[0];
String desiredResponseCode = args[1];
...
JavaRDD<String> textFileRDD = spark.textFile(inputFileName);
JavaRDD<String> okLines = textFileRDD.filter(new
Function<String, Boolean>() {
    @Override
    public Boolean call(String s) {
        String code = s.split("\t")[5];
        return code.equals(desiredResponseCode);
    }
});
```

Examples

```
# Count by response code
# Important! Not all transformations and actions are on the getting started guide
JavaPairRDD<Integer, String> linesByCode = textFileRDD.mapToPair(new PairFunction<String, Integer, String>() {
    @Override
    public Tuple2<Integer, String> call(String s) {
        String code = s.split("\t")[5];
        return new Tuple2<Integer, String>(Integer.valueOf(code), s);
    }
});
Map<Integer, Long> countByCode = linesByCode.countByKey();
System.out.println(countByCode);
```

RDD<T>#filter

- func: T → Boolean
- Applies the predicate function on each record and produces that tuple only of the predicate returns true
- Result RDD<T> with same or fewer records than the input
- In Hadoop:
  - map(T value) {
      if (func(value))
        context.write(value)
    }

RDD<T>#map(func)

- func: T → U
- Applies the map function to each record in the input to produce one record
- Results in RDD<U> with the same number of records as the input
- In Hadoop:
  - map(T value) {
    context.write(func(value));
    }

RDD<T>#flatMap(func)

- func: T → Iterator<V>
- Applies the map function to each record and add all resulting values to the output RDD
- Result: RDD<V>
- This is the closest function to the Hadoop map function
- In Hadoop:
  - ```
    map(T value) {
      Iterator<V> results = func(value);
      for (V result : results)
        context.write(result)
    }
    ```
    ```

RDD<T>#mapPartition(func)

- func: Iterator<T> → Iterator<U>
- Applies the map function to a list of records in one partition in the input and adds all resulting values to the output RDD
- Can be helpful in two situations
  - If there is a costly initialization step in the function
  - If many records can result in one record
- Result: RDD<U>

RDD<T>#mapPartition(func)

- In Hadoop, the mapPartition function can be implemented by overriding the run() method in the Mapper, rather than the map() function
  - run(context) {

```
      // Initialize
      Array<T> values;
      for (T value : context)
        values.add(value);
      Iterator<V> results = func(values);
      for (V value : results)
        context.write(value);
    }
```

RDD<T>#mapPartitionWithIndex(func)

- func: (Integer, Iterator<T>) → Iterator<U>
- Similar to mapPartition but provides a unique index for each partition
- In Hadoop, you can achieve a similar functionality by retrieving the InputSplit or taskID from the context.

RDD<T>#sample(r, f, s)

- r: Boolean: With replacement (true/false)
- f: Float: Fraction [0,1]
- s: Long: Seed for random number generation
- Returns RDD<T> with a sample of the records in the input RDD
- Can be implemented using `mapPartitionWithIndex` as follows
  - Initialize the random number generator based on seed and partition index
  - Select a subset of records as desired
  - Return the sampled records

RDD<T>#distinct()

- Removes duplicate values in the input RDD
- Returns RDD<T>
- Implemented as follows
  map(x => (x, null)).
  reduceByKey((a, b) => a, numPartitions).
  map(_._1)
- Note: Both a and b are null in the reduceByKey function above

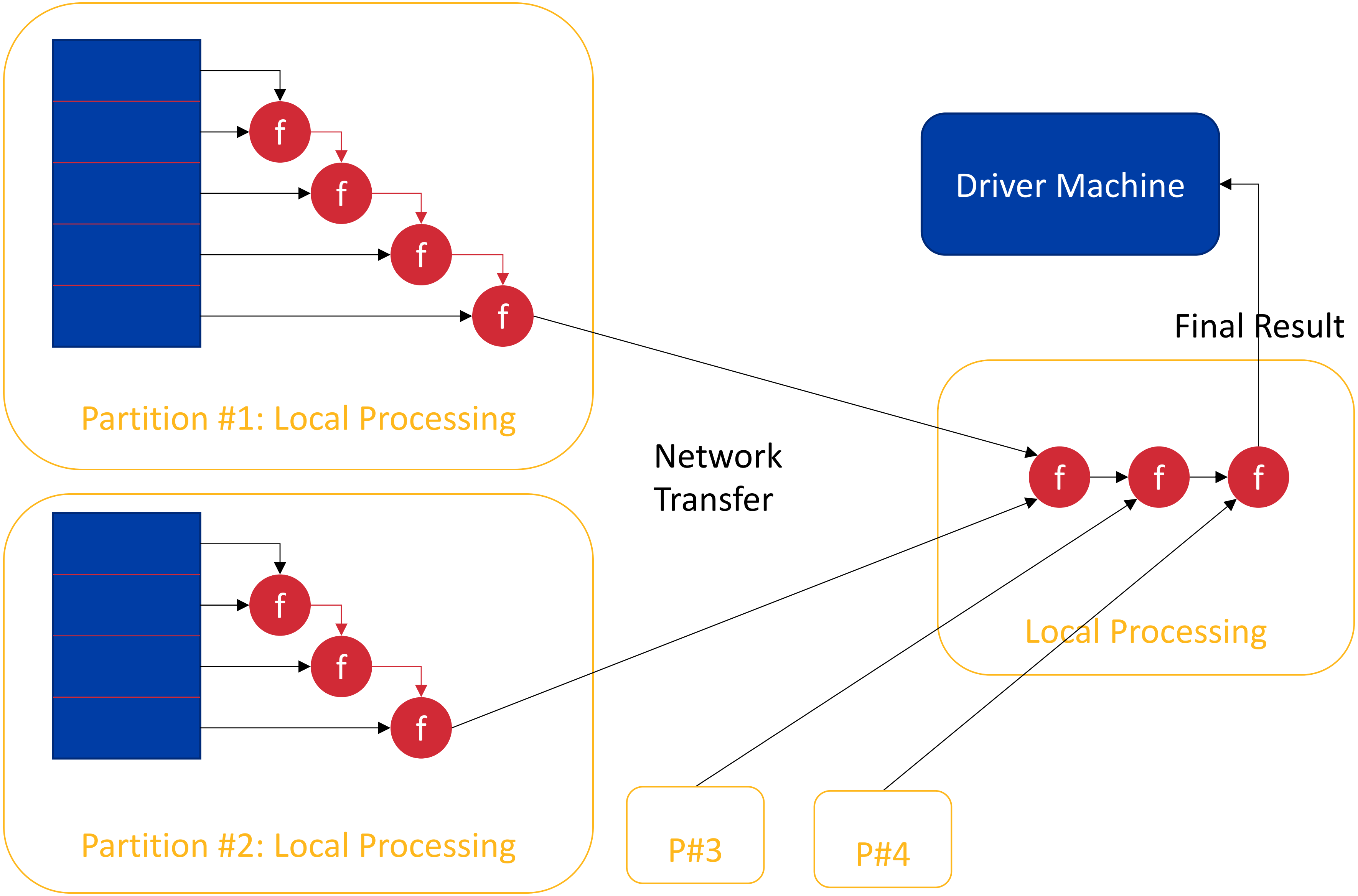
RDD<T>#reduce(func)

- func: (T, T) → T
- This is not the same as the reduce function of Hadoop even though it has the same name
- Reduces all the records to a single value by repeatedly applying the given function
- Result: T
- This is an action

RDD<T>#reduce(func)

- In Hadoop
  - map(T value) {
    context.write(NullWritable.get(), value);
    }
  - combine, reduce(key, Iterator<T> values)
    {
    T result = values.next();
    while (values.hasNext())
    result = func(result, values.next());
    context.write(result);
    }

RDD<T>#reduce(func)



RDD<K,V>#reduceByKey(func)

- func: (V, V) → V
- Similar to reduce but applies the given function to each group separately
- Since there could be so many groups, this operation is a transformation that can be followed by further transformations and actions
- Result: RDD<K,V>
- By default, number of reducers is equal to number of input partitions but can be overridden

RDD<K,V>#reduceByKey(func)

- In Hadoop:
  - map(K key, V value) {
    ```
    context.write(key, value);
    }
    ```
  - combine, reduce(K key, Iterator<V> values) {
    ```
    V result = values.next();
    while (values.hasNext())
      result = func(result, values.next());
    context.write(key, result);
    }
    ```

Limitation of reduce methods

- Both reduce methods have a limitation is that they have to return a value of the same type as the input.
- Let us say we want to implement a program that operates on an RDD<Integer> and returns one of the following values
  - 0: Input is empty
  - 1: Input contains only odd values
  - 2: Input contains only even values
  - 3: Input contains a mix of even and odd values

RDD<T>#aggregate(zero, seqOp, combOp)

- zero: U - Zero value of type U
- seqOp: (U, T) → U – Combines the aggregate value with an input value
- combOp: (U, U) → U – Combines two aggregate values
- Returns U

- Similarly, aggregateByKey operates on RDD<K,V> and returns RDD<K,U>

RDD<T>#aggregate(zero, seqOp, combOp)

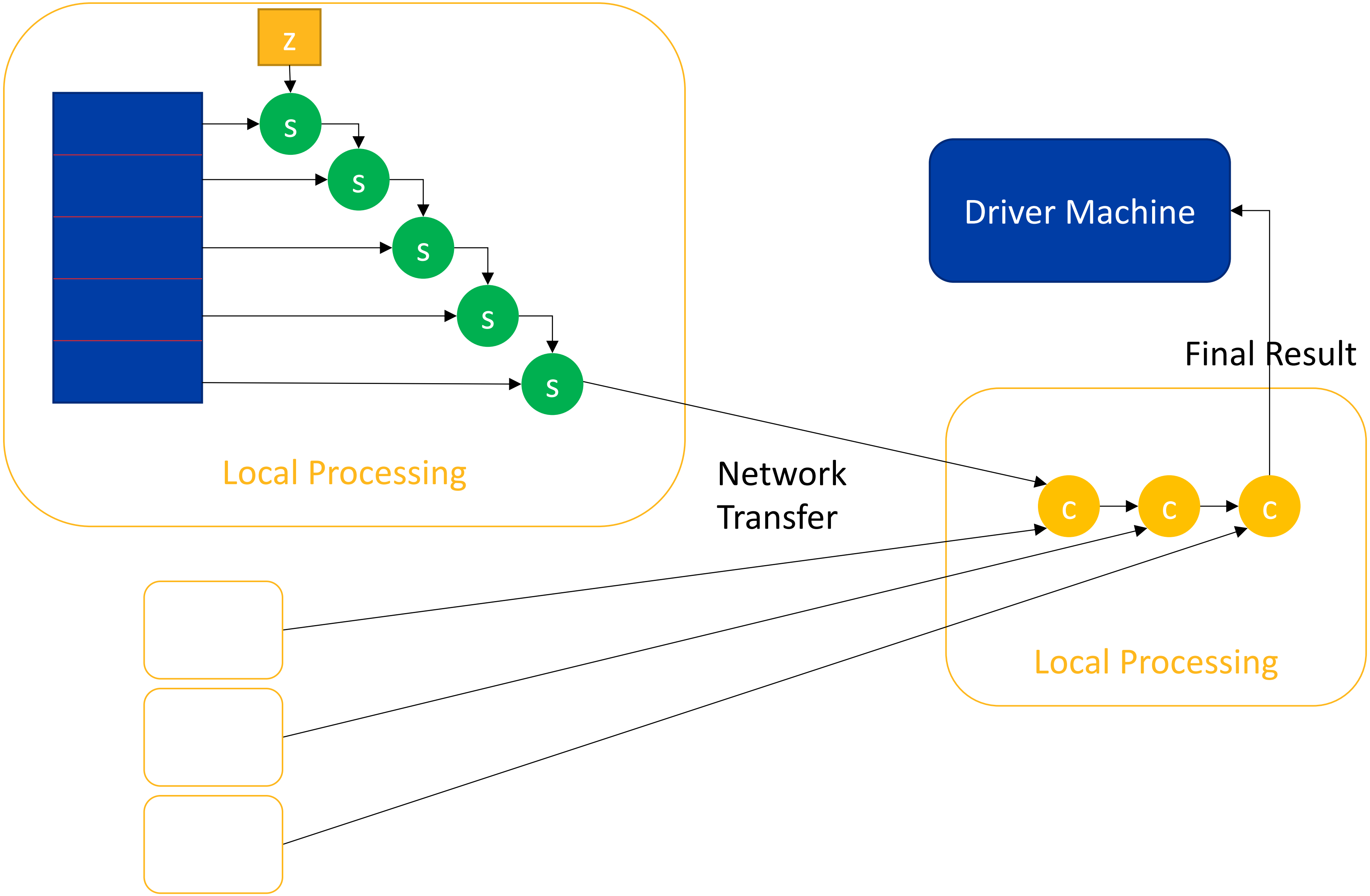
- In Hadoop:
  - run(context) {
    ```
    U result = zero;
    for (T value : context)
      result = seqOp(result, value);
    context.write(NullWritable.get(), result);
    ```
    }
  - combine,reduce(key, Iterator<U> values) {
    ```
    U result = values.next();
    while (values.hasNext())
      result = combOp(result, values.next());
    context.write(result);
    ```
    }

RDD<T>#aggregate(zero, seqOp, combOp)

- Example:
- RDD<Integer> values
- Byte marker = values.aggregate((Byte)0,

```
  (result: Byte, x: Integer) => {
    if (x % 2 == 0) // Even
      return result | 2;
    else
      return result | 1;
  },
  (result1: Byte, result2: Byte) => result1 |
result2
);
```

RDD<T>#aggregate(zero, seqOp, combOp)



RDD<K,V>#groupByKey()

- Groups all values with the same key into the same partition
- Closest to the shuffle operation in Hadoop
- Returns RDD<K, Iterator<V>>
- Performance notice: By default, all values are kept in memory so this method can be very memory consuming.
- Unlike the reduce and aggregate methods, this method does not run a combiner step, i.e., all records get shuffled over network

Further Readings

- List of common transformations and actions
  - http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations
- Spark RDD Scala API
  - http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD