Big Data Processing

CELEBRATING 30 YEARS
Marlan and Rosemary Bourns
College of Engineering

Learning Outcomes

- Get familiar with big-data programming models and query execution engines
- Understand how big-data systems execute user queries internally
- Know the architectural differences between big-data systems

Big Data Framework

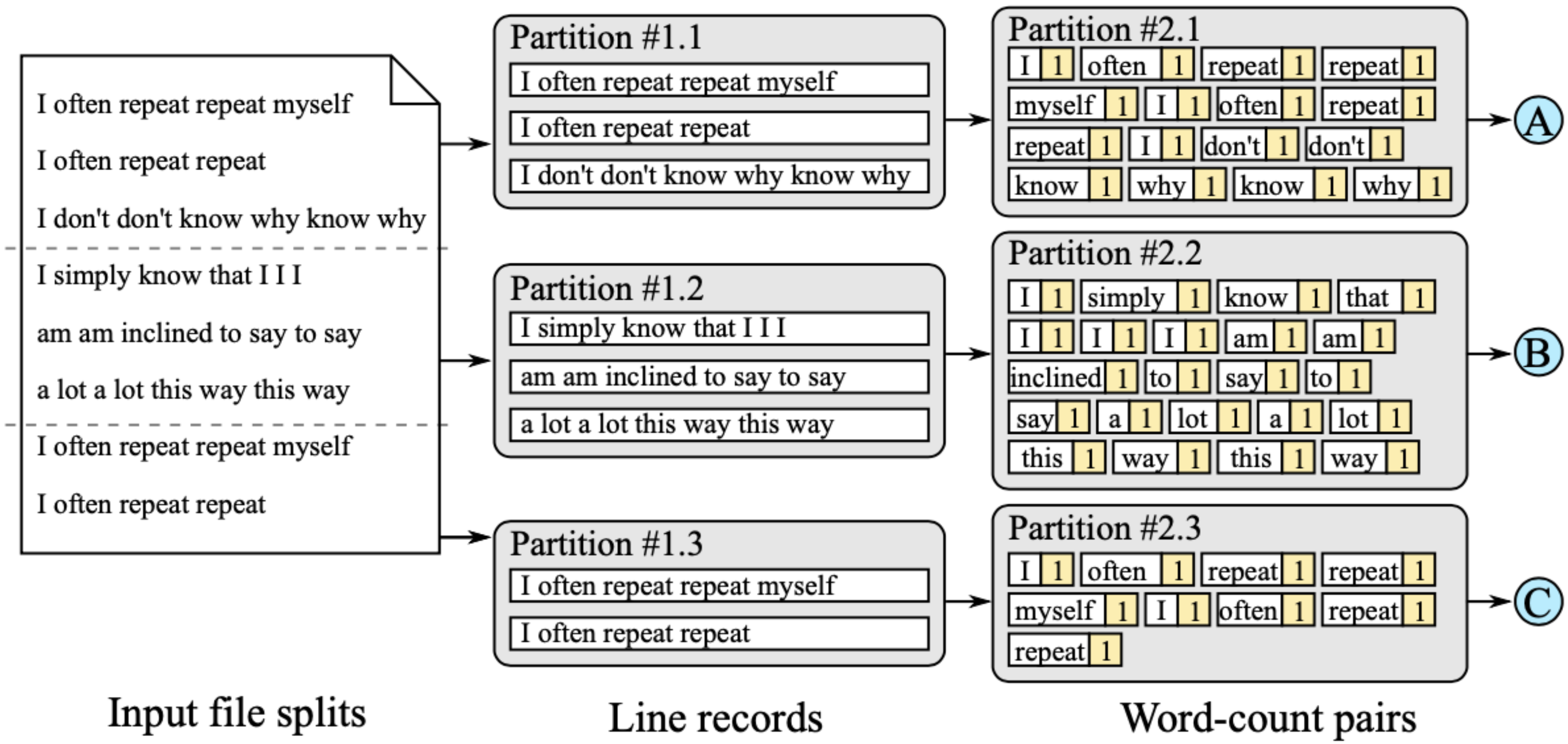
- A system that allows developers to write a program and execute it on a cluster of machines.
- Hides most of the low-level system issues such as fault tolerance, network communication, and load balancing.
- Imposes some restrictions on the developer to ensure that they can run the program efficiently.

Word Count Example

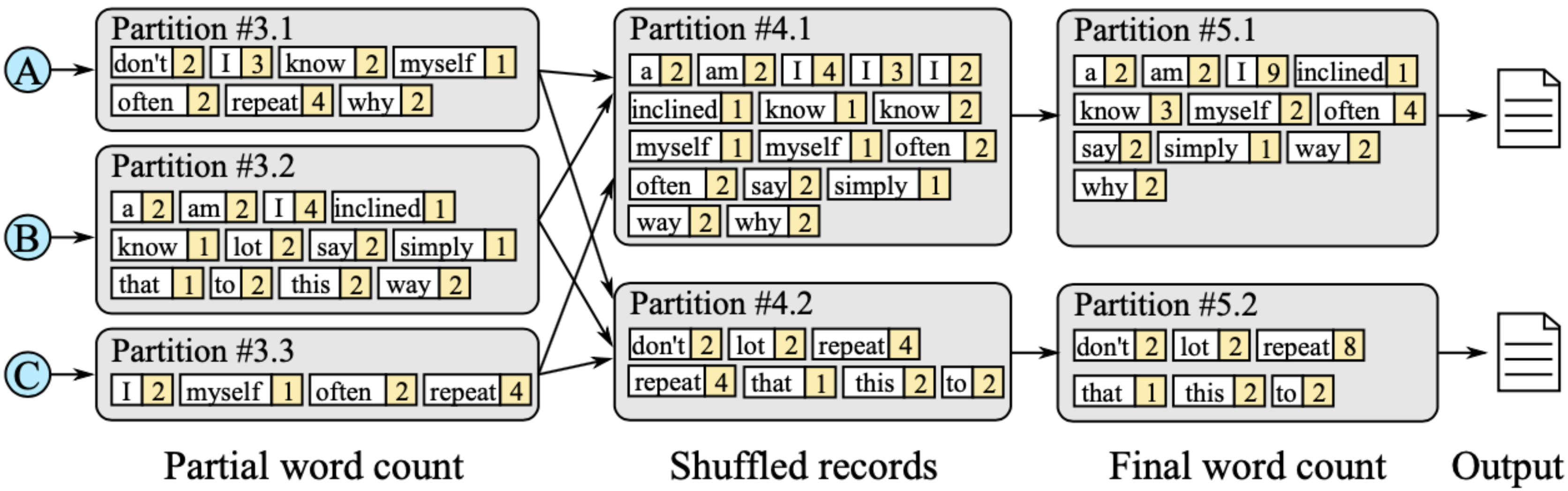
Word	Count
a	2
am	2
don't	2
I	9
inclined	1
know	3
lot	2
myself	2
often	4
repeat	8
say	2
simply	1
that	1
this	2
to	2
way	2
why	2

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 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

```java
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);
  }
}
```

```java
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

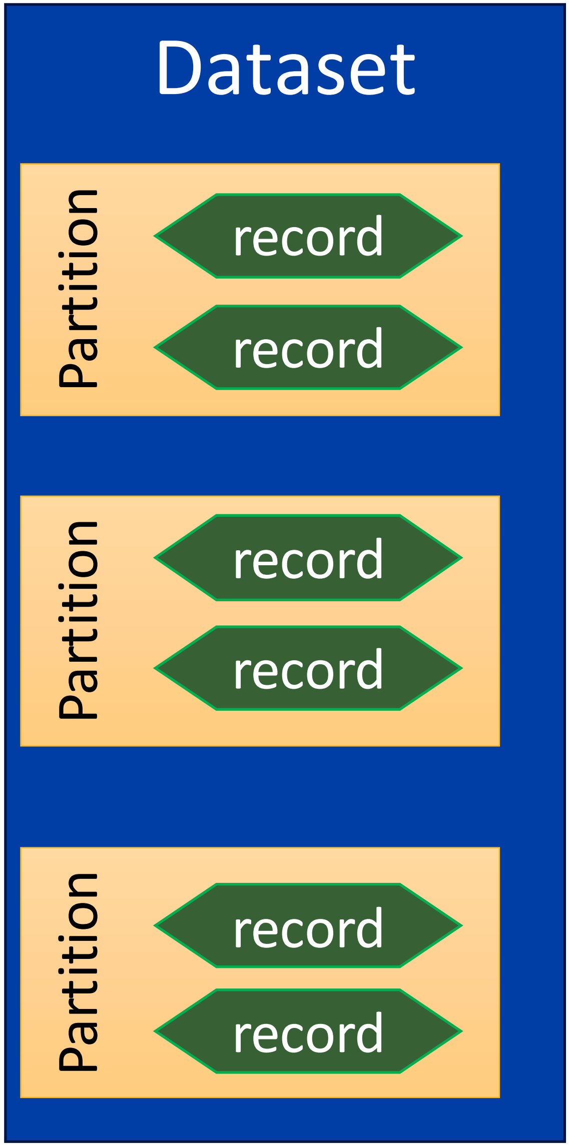
```
// In Scala shell
val lines = sc.textFile("data.txt")
val pairs = lines.flatMap(s => s.split("\\b"))
  .map(w => (w,1))
val counts = pairs.reduceByKey((a, b) => a + b)
counts.saveAsTextFile("word_count_output.txt")
```

Big-data Processing

- Data Model
  - How data is seen by system
- 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

Data Model

- Big data systems deal with multiple datasets
- Each dataset is a multiset of records (repeated values)
- Each record can be of any value
- All records in a dataset must have the same type
- Special handling is given when the object is a key-value pair, as detailed later
- A dataset could be partitioned across machines

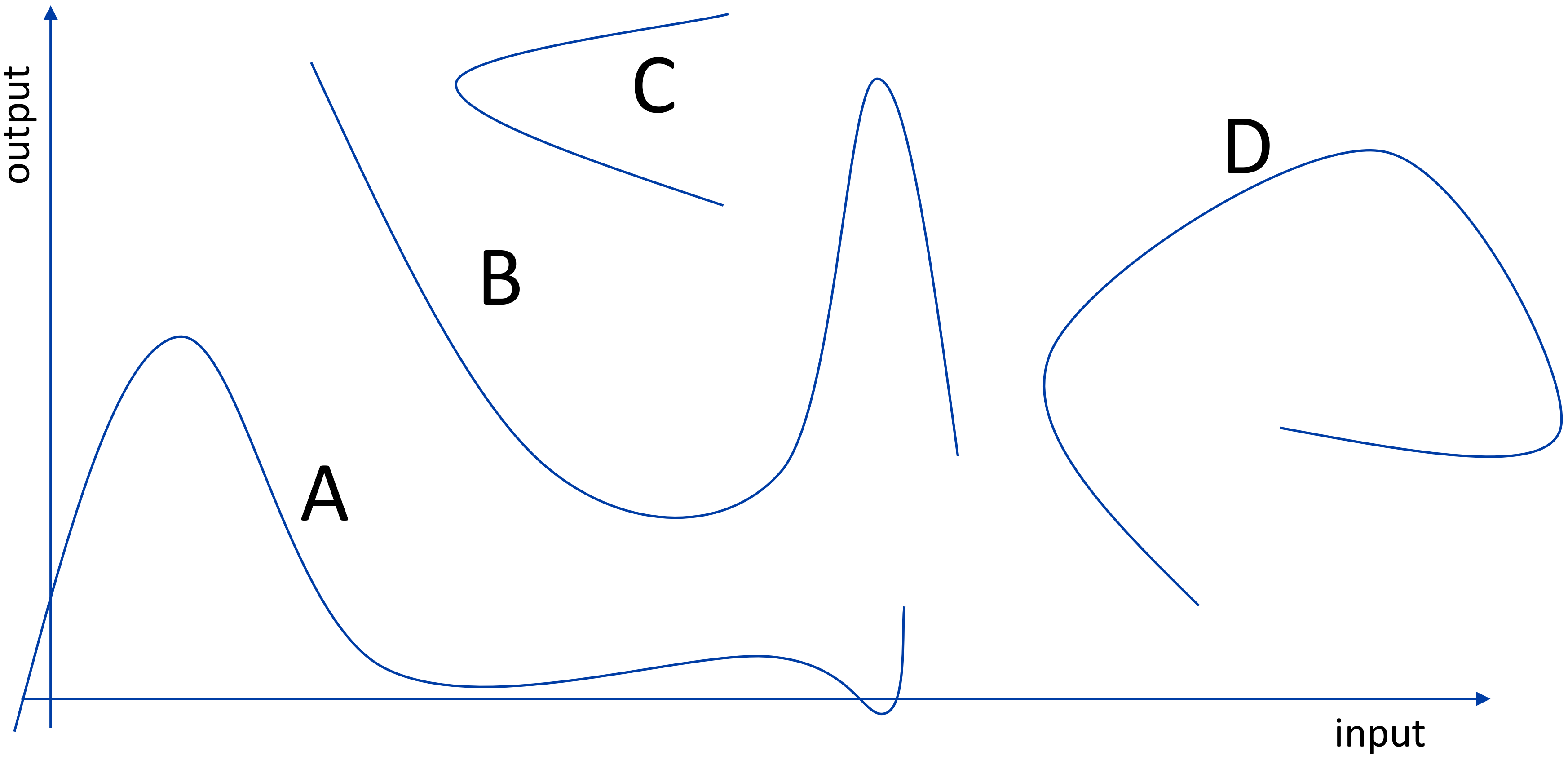


Functional Programming Model

- 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?



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;
}
```

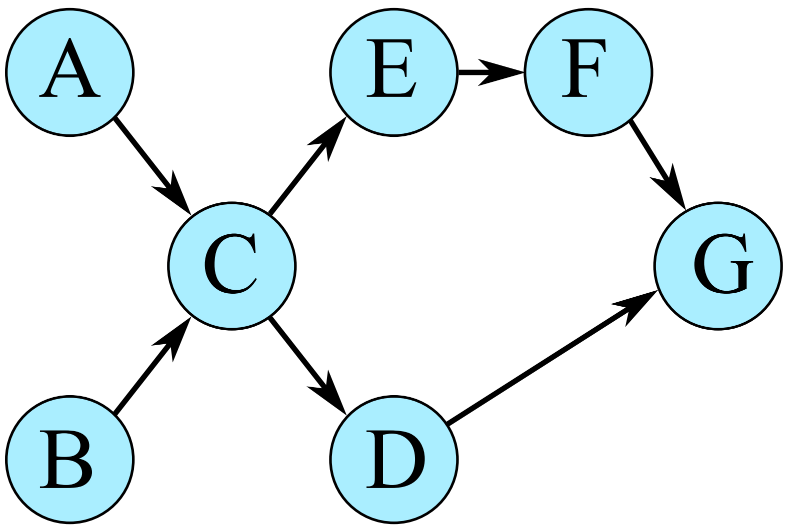
```
Int sum
Function2(x) {
  sum += x;
  return sum;
}
```

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

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

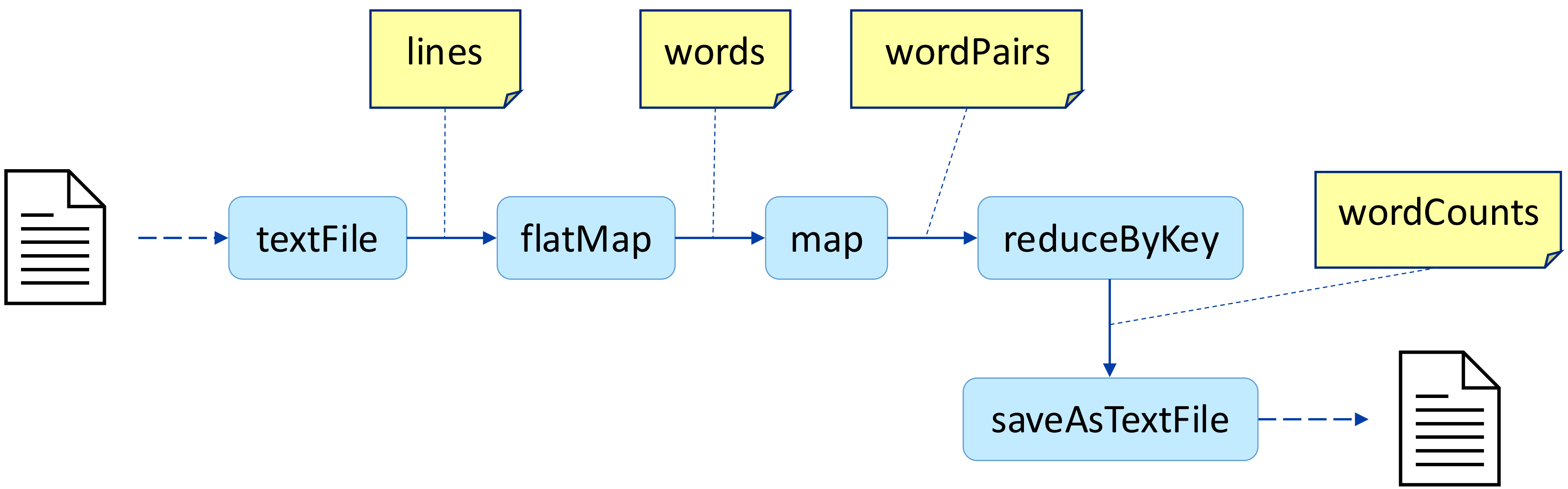
Directed Acyclic Graph

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



DAG for Word Count in Spark

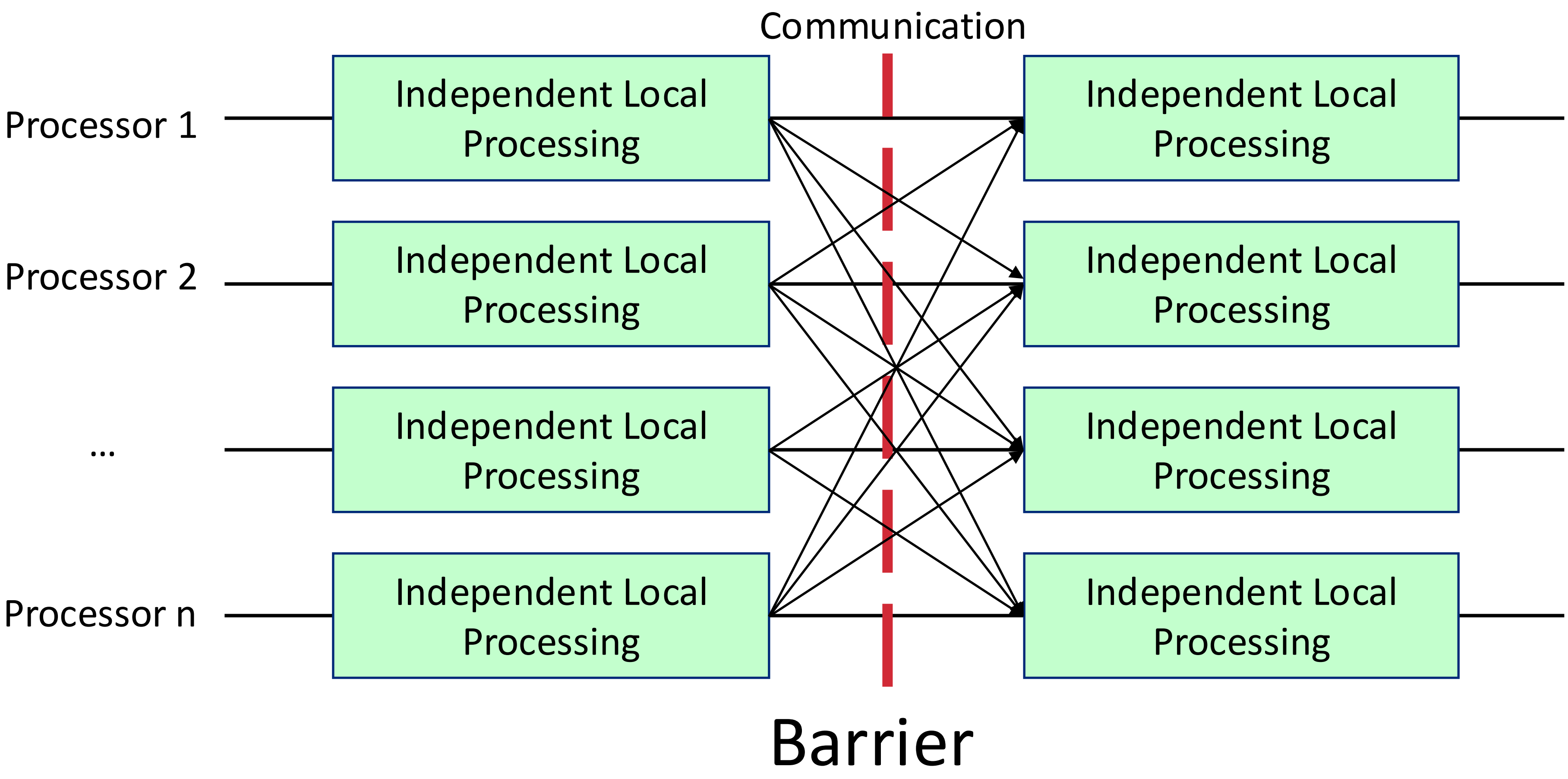
```
// In Scala shell
val lines = sc.textFile("data.txt")
val wordPairs = lines.flatMap(s => s.split("\\b"))
  .map(w => (w,1))
val wordCounts = pairs.reduceByKey((a, b) => a + b)
wordCounts.saveAsTextFile("word_count_output.txt")
```



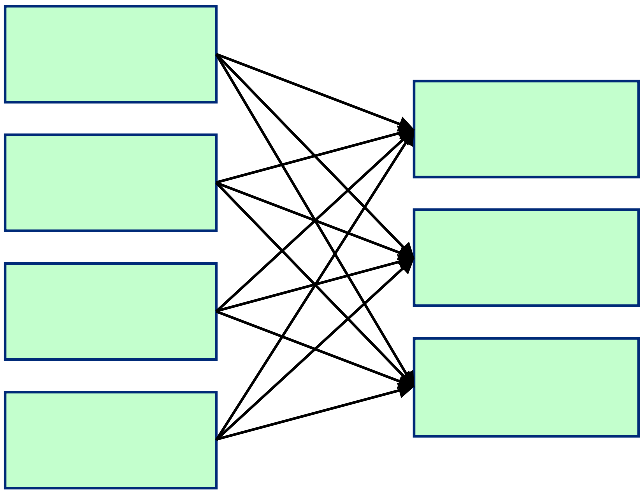
Bulk Synchronous Parallel (BSP)

- 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

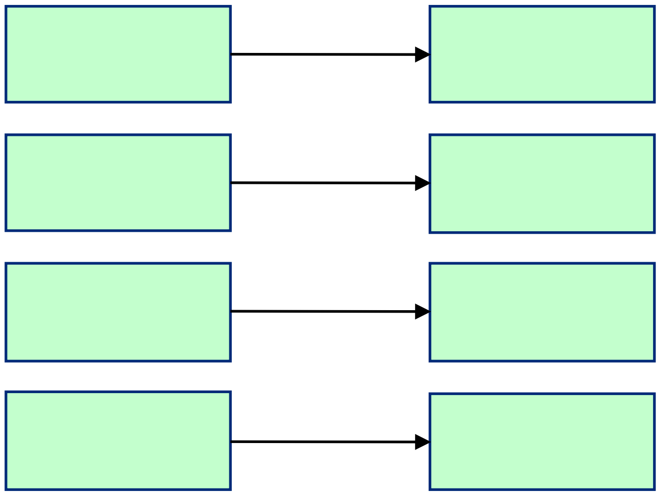
BSP Model


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

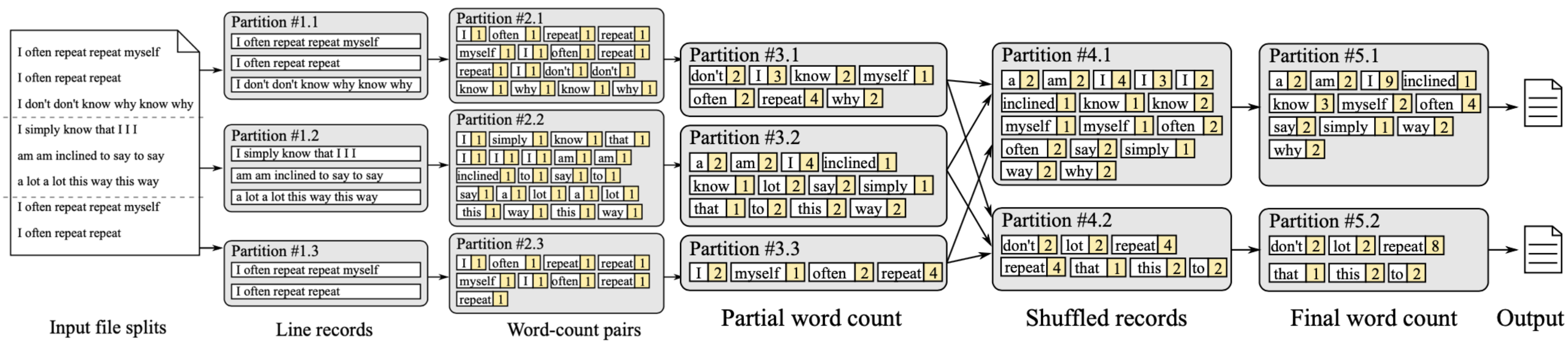


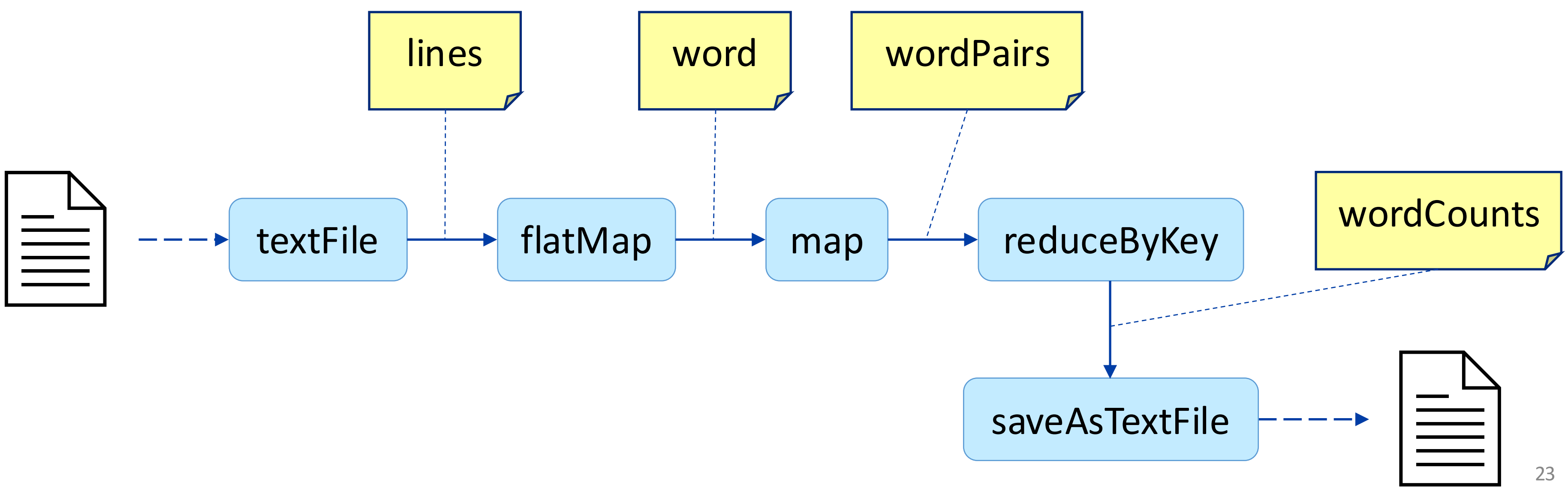
Fully Connected
(Requires network communication)



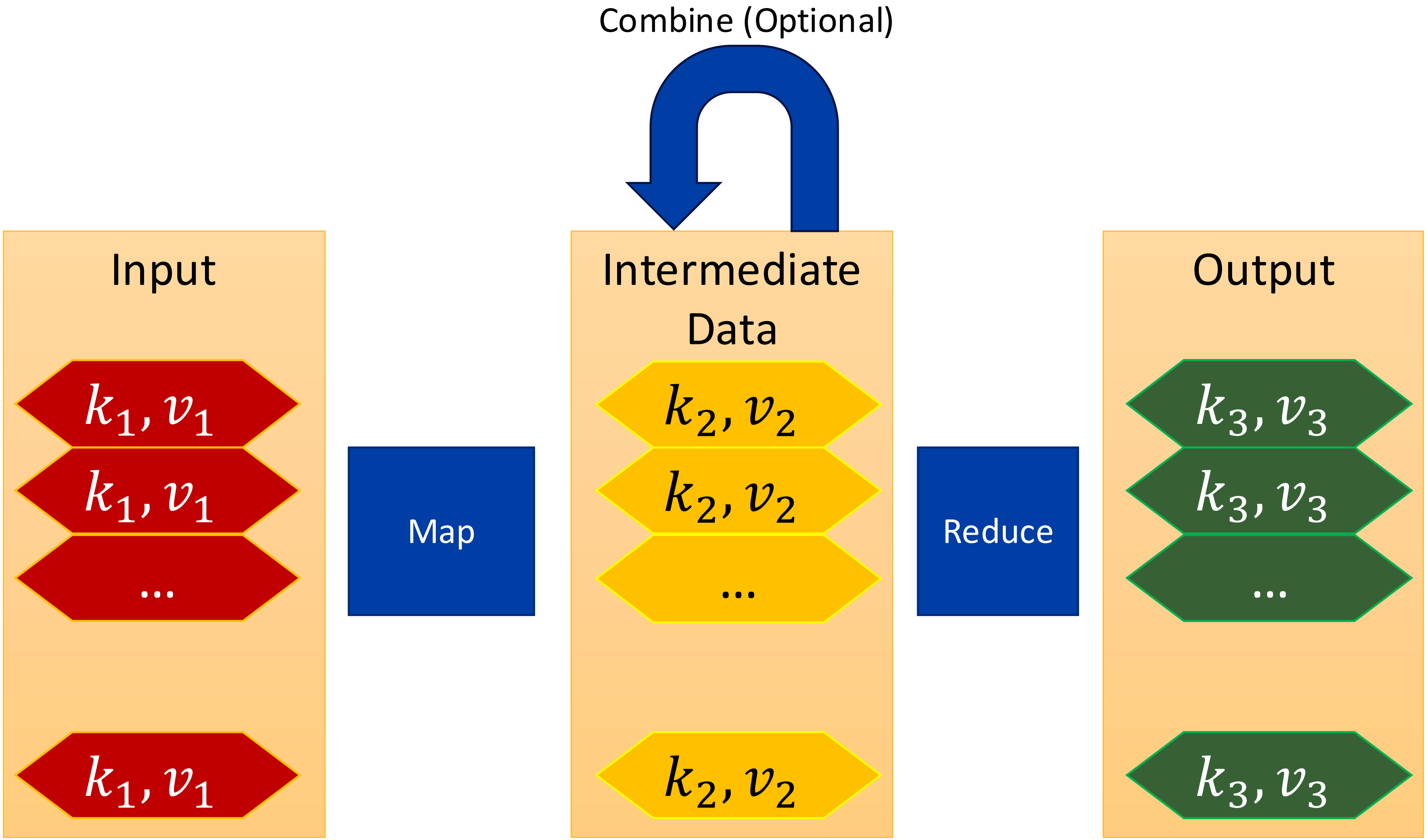
One-to-one
(Can be done locally in one stage)

Word Count Stages





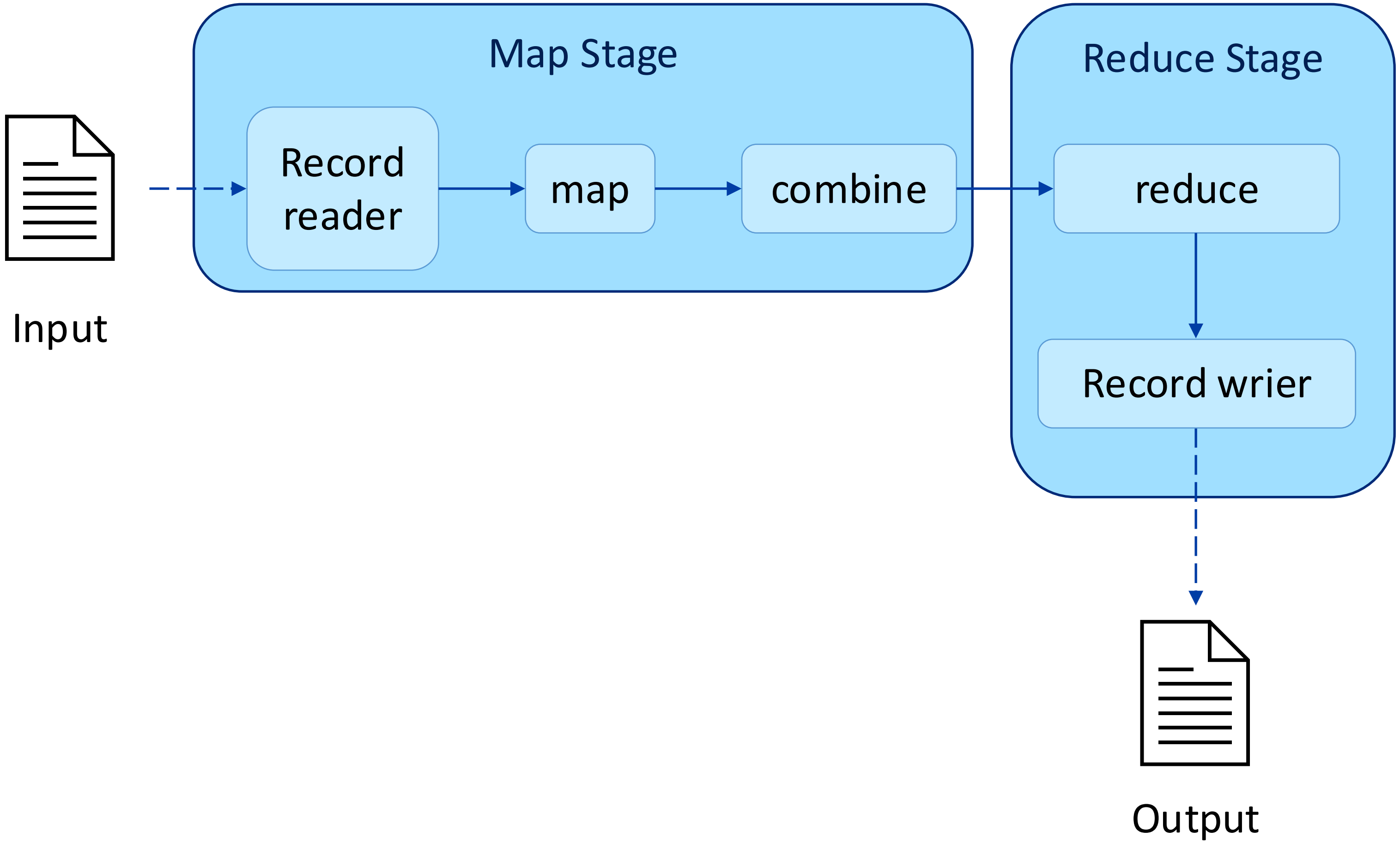
Hadoop MapReduce



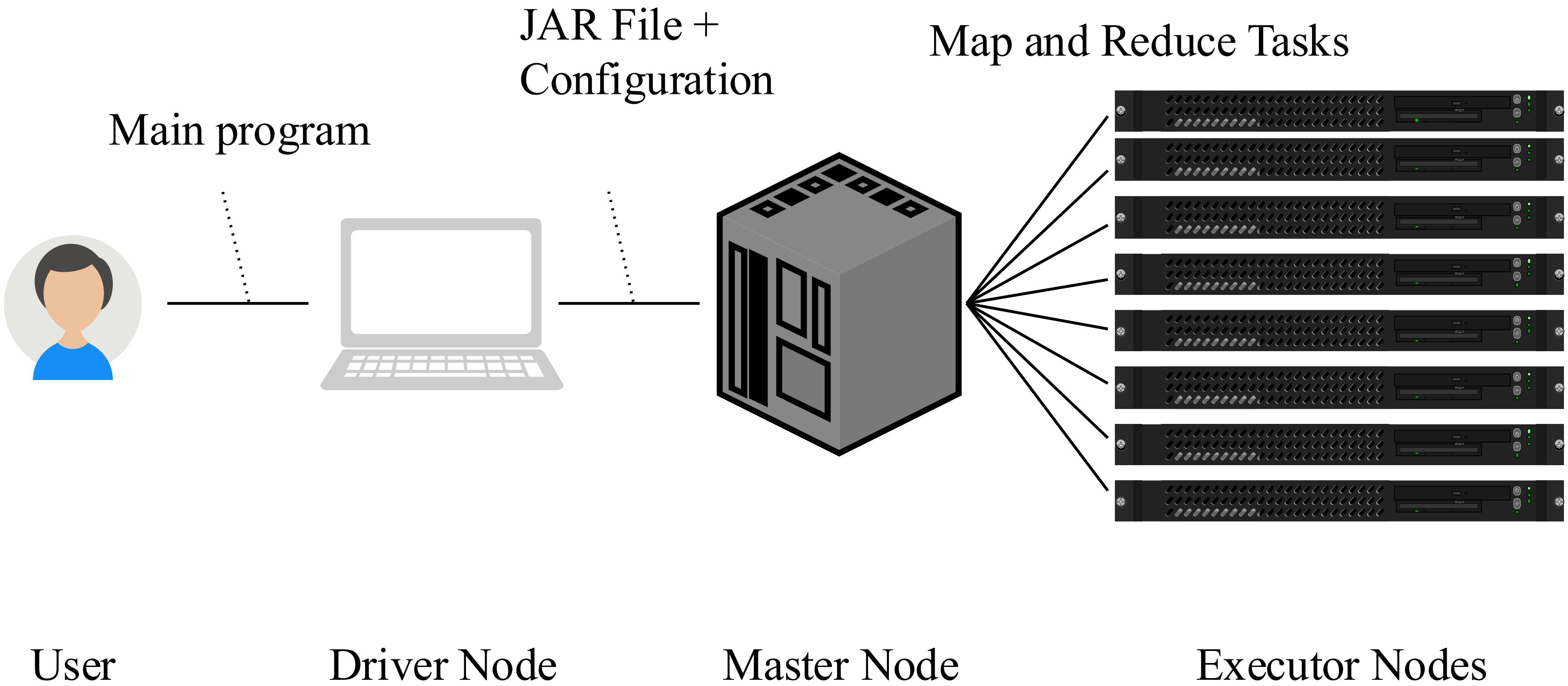
Map and Reduce Functions

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

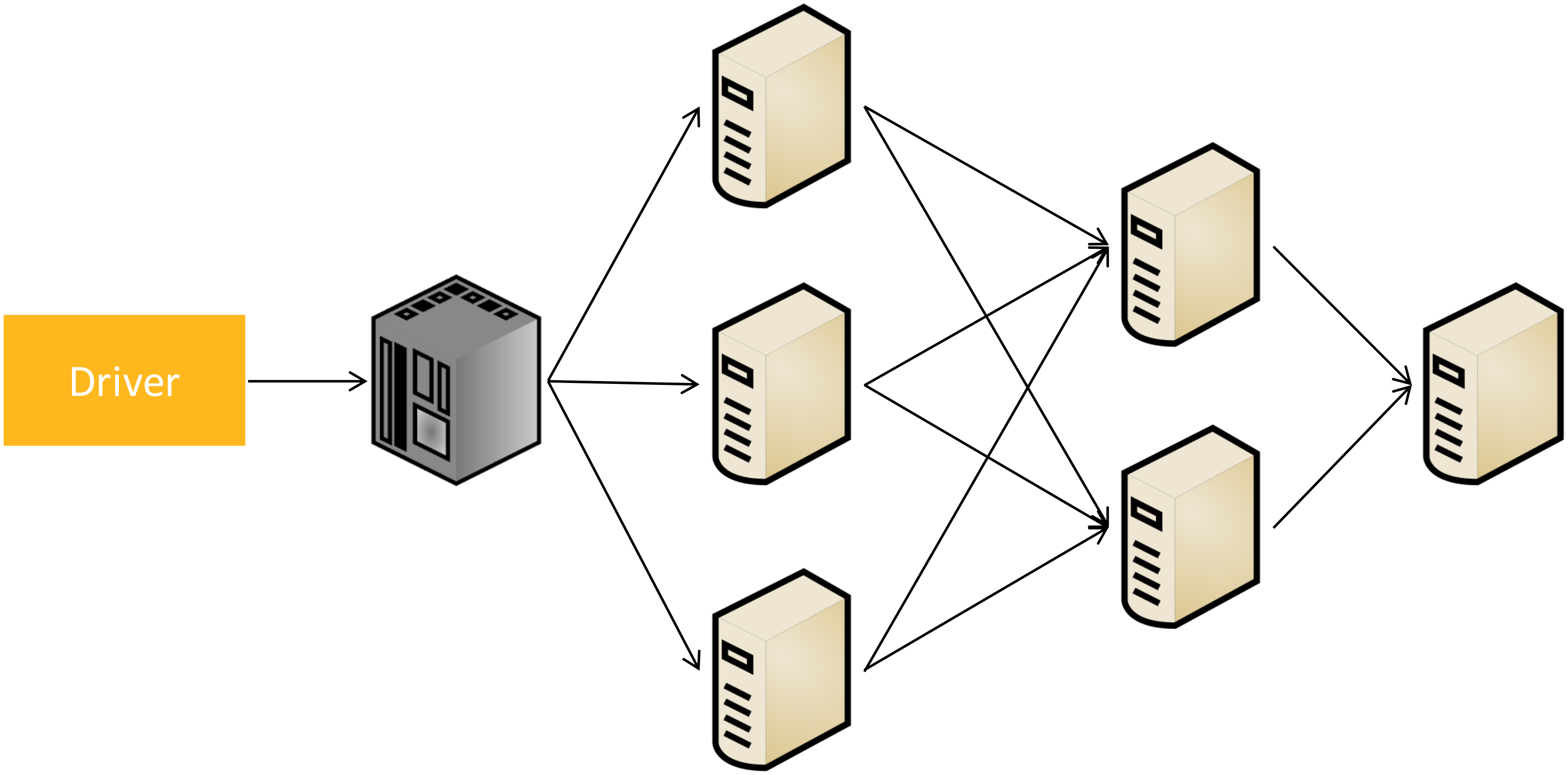
MapReduce DAG



MapReduce Program Cycle



Job Execution Overview



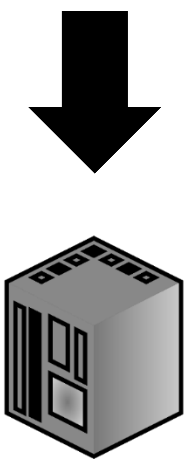
Job submission | Job preparation | Map | Shuffle | Reduce | Cleanup

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.bigdata.eldawy.WordCount
Reducer	…
JAR File	…
User-defined	User-defined



Serialized over network

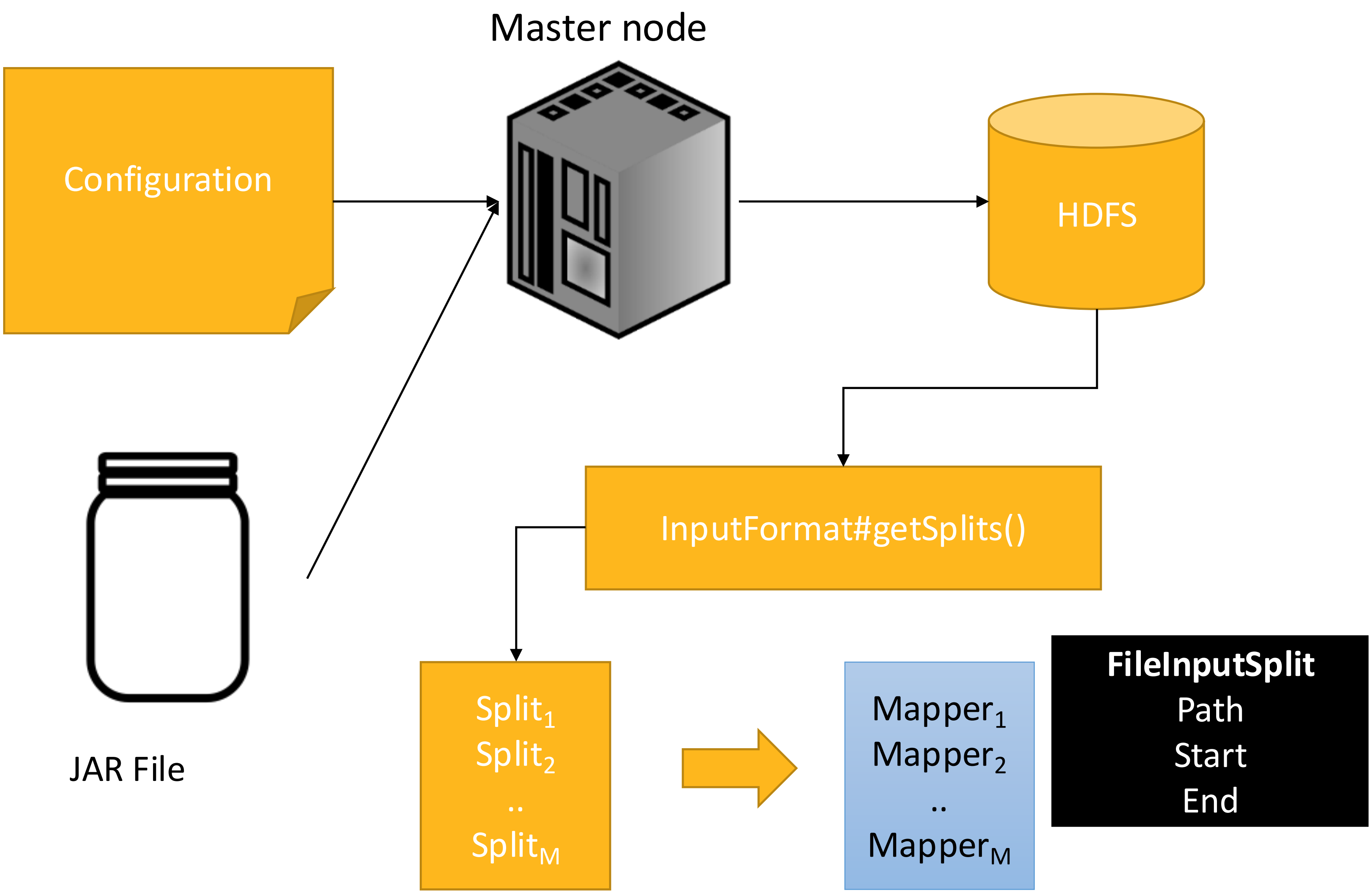
Master node

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



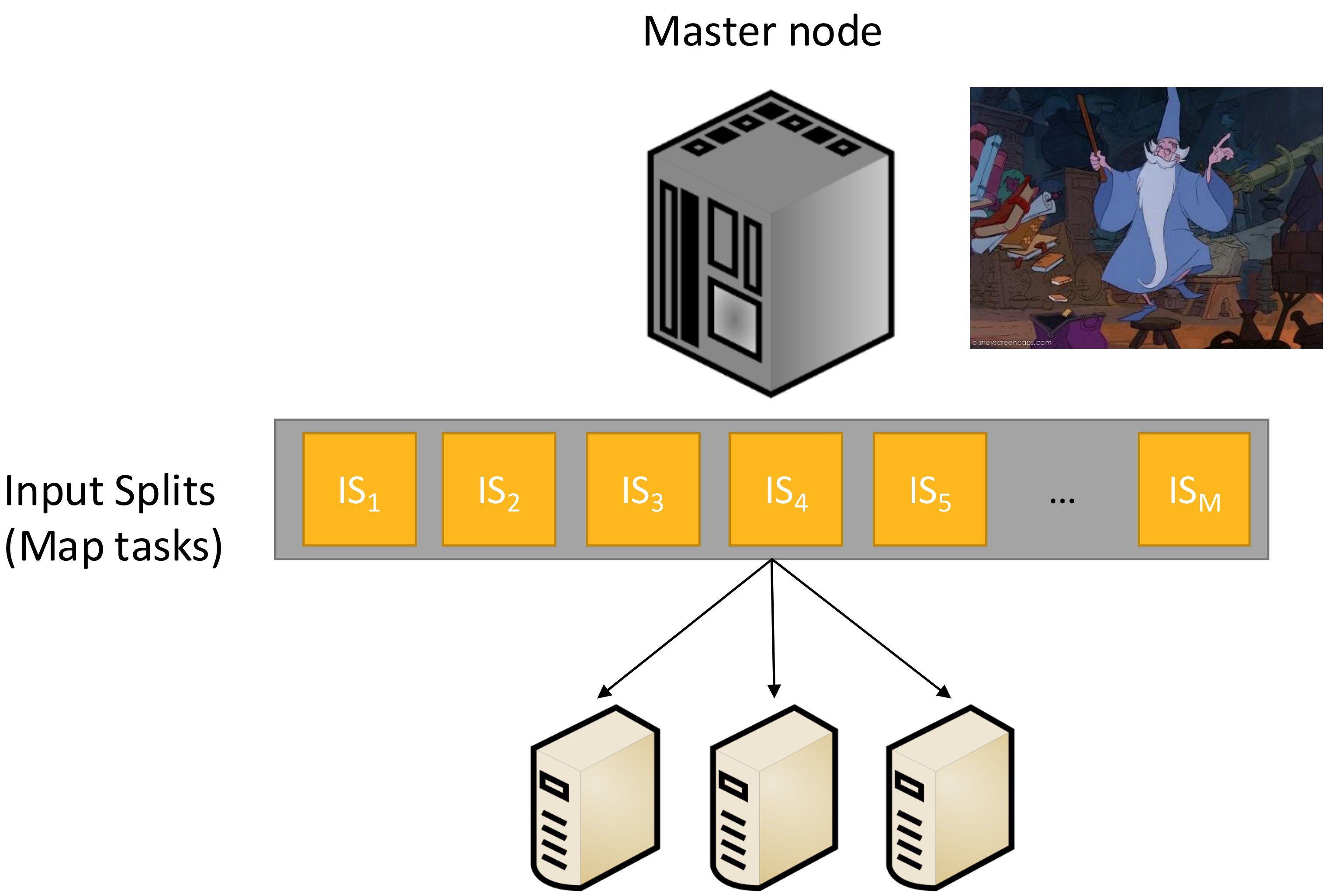
Map Phase

- 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

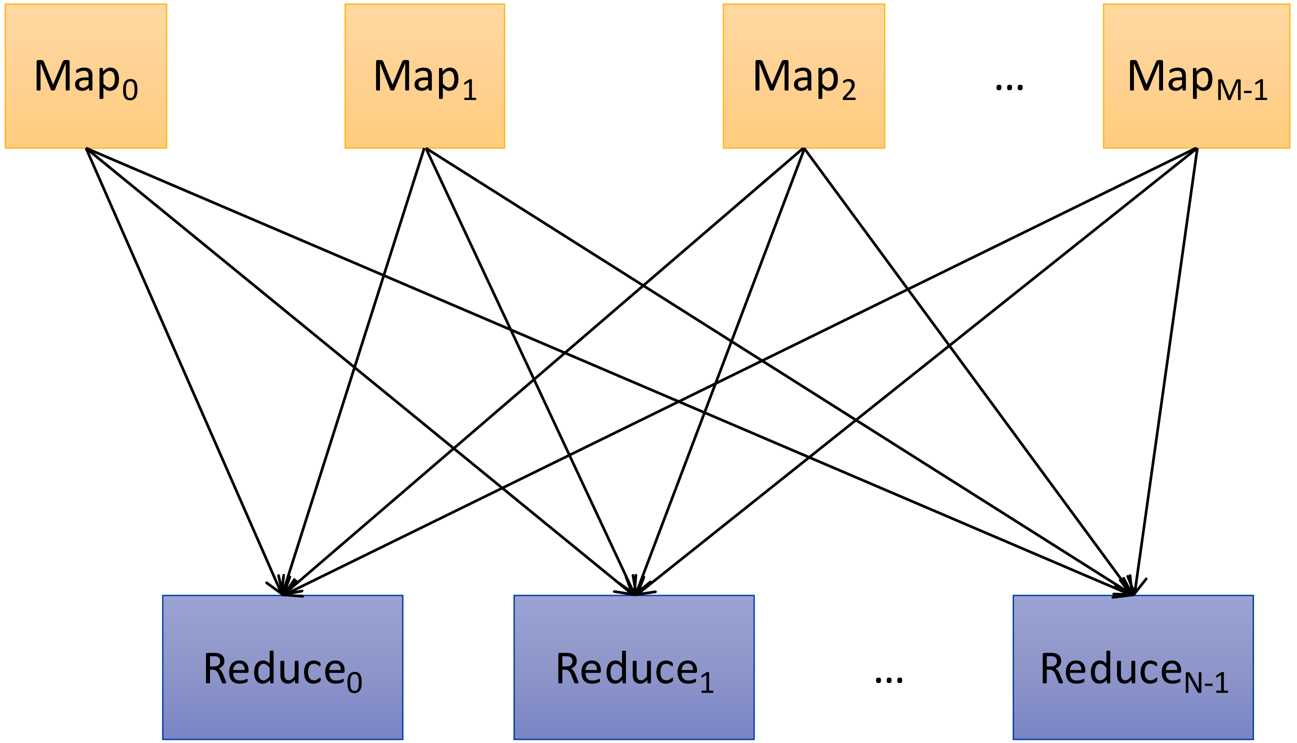
Map output

- 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

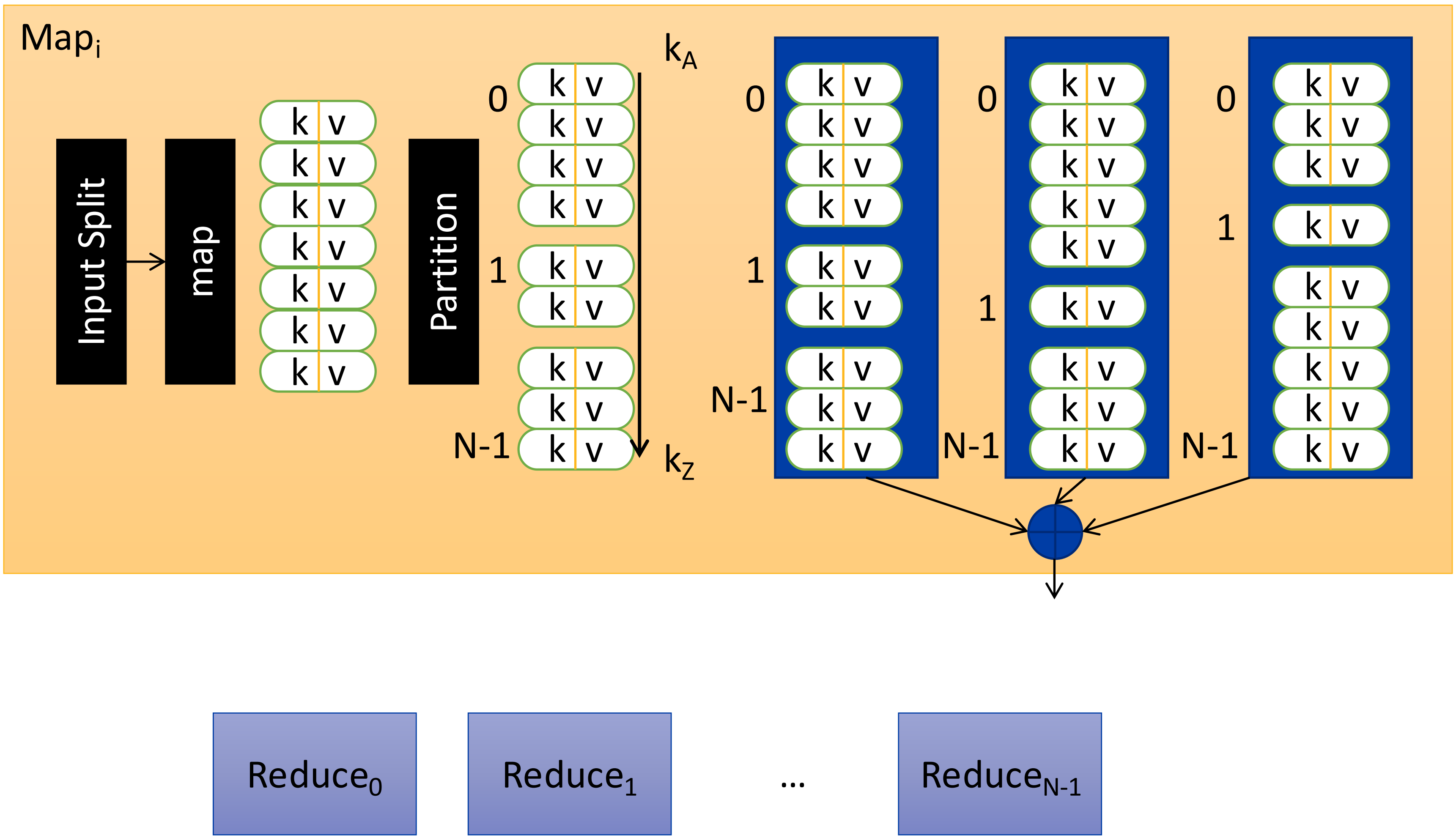
Shuffle Phase

- 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

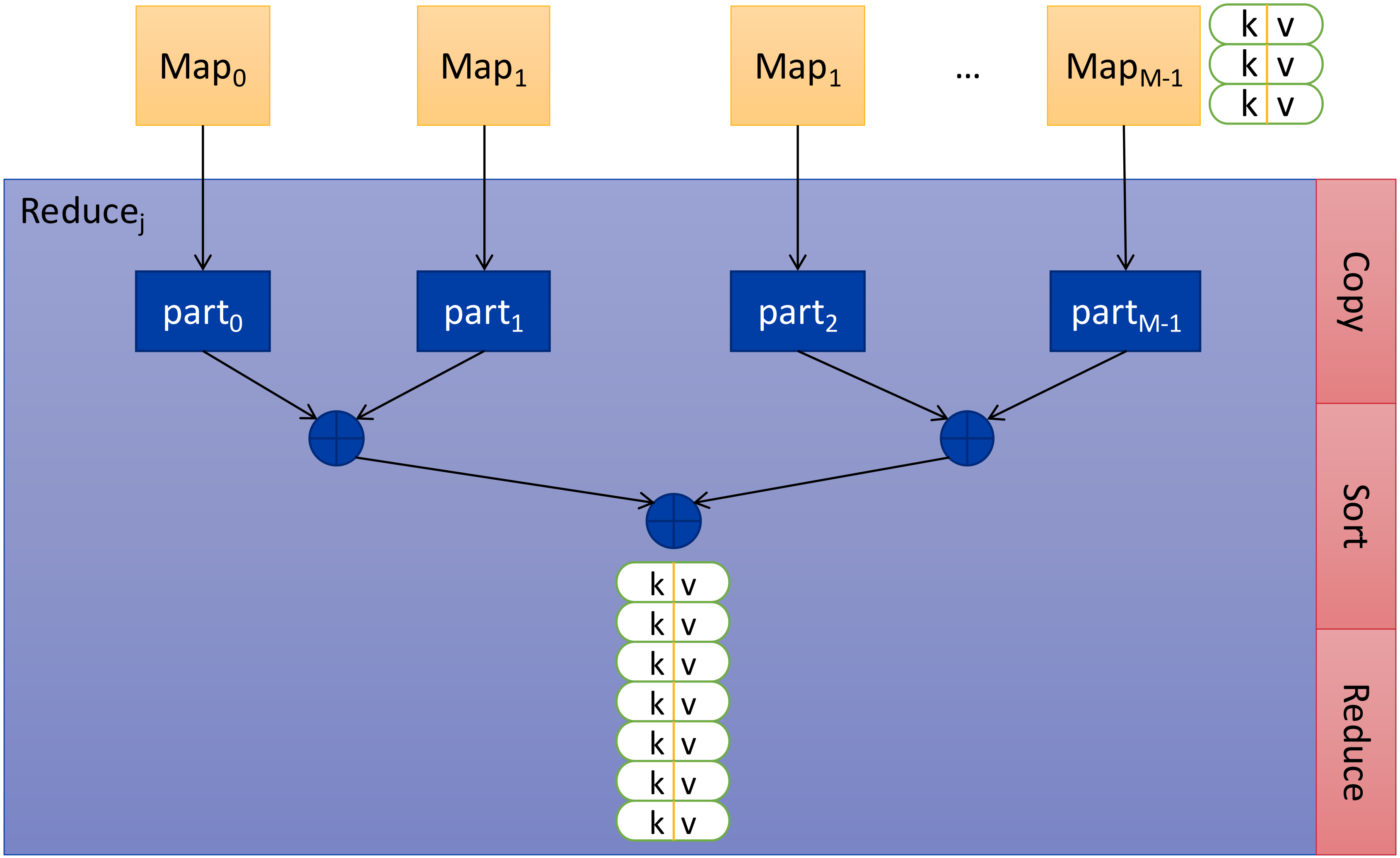
Shuffle Phase



Shuffle Phase (Map-side)

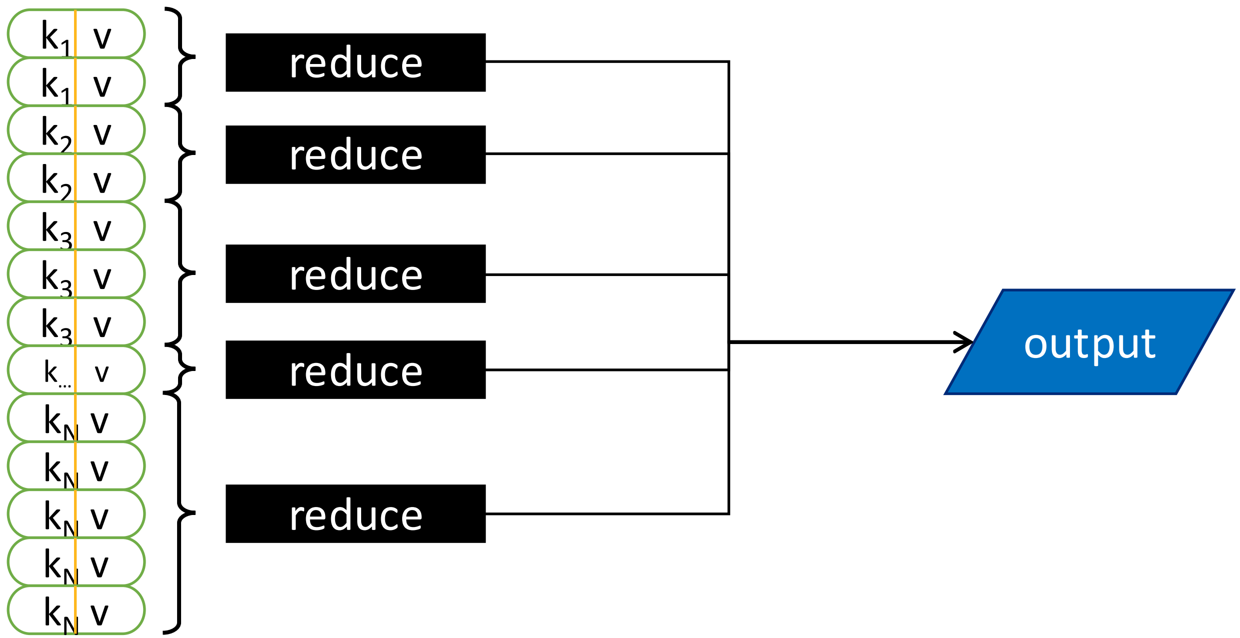


Shuffle Phase (Reduce-side)



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 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.
- The commit step typically writes an empty "_SUCCESS" file to indicate job success.

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.