MapReduce

Distributed computing
can get very complicated

Managing resources and
memory across multiple nodes

MapReduce

Distributed computing can get very complicated

What to do if a node goes down?

MapReduce

MapReduce abstracts the programmer from all these complications

MapReduce

You just define 2 functions

map() reduce()

MapReduce

map() reduce()

The rest is taken care of by Hadoop!

Let's take an example

We have a large text file

```
next  up  previous  contents  index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing   Contents   Index


Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the
World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore,
use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is
partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing
for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-
partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for
distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on
cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with
specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time.
One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in
case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The
various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 .
First, the input data, in our case a collection of web pages, are split into $n$ splits where the size of the split is chosen to ensure
that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage
should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead
assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine
dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs .
For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed
and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent
terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here
and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also
encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to
local intermediate files, the segment files (shown as  \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5 ).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly.
This is achieved by partitioning the keys into $j$ term partitions and having the parsers write key-value pairs for each term partition
into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these
key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are
defined by the person who operates the indexing system (Exercise 4.6 ). The parsers then write corresponding segment files, one for each
term partition. Each term partition thus corresponds to $r$ segments files, where $r$ is the number of parsers. For instance, Figure 4.5
shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The
```

Objective: Create a Frequency Distribution of words in the file

We have a large text file

next up previous contents index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs . For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment files (shown as \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system (Exercise 4.6). The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to r segments files, where r is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The

This is a pretty common task in Natural Language Processing

We have a large text file

next up previous contents index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs . For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

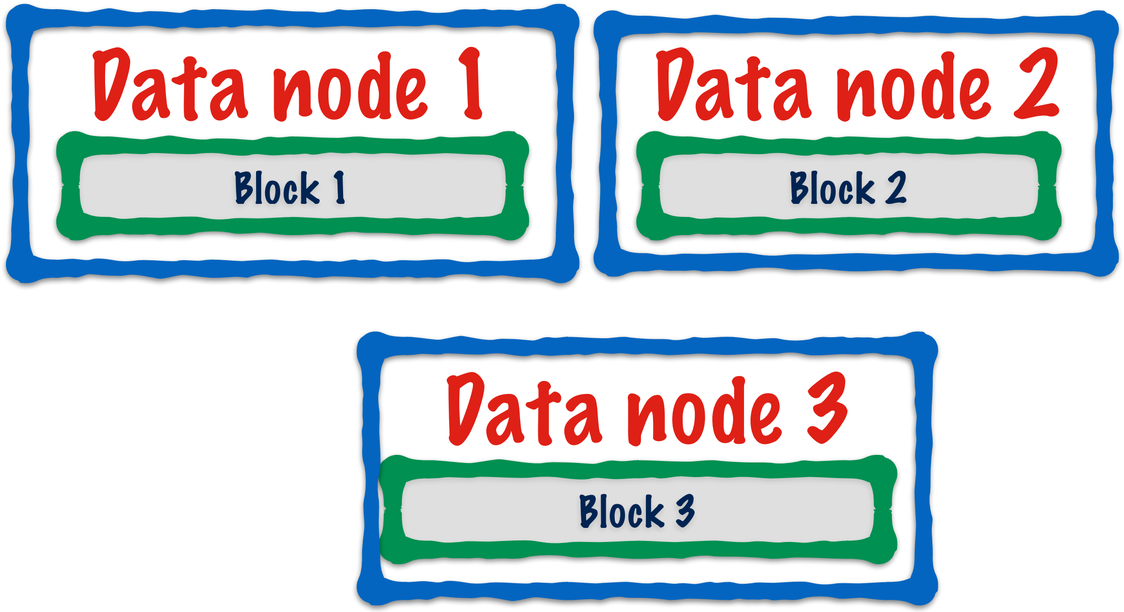
The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment files (shown as \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system (Exercise 4.6). The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to r segments files, where r is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The

Word	Count
because	1
each	4
figure	9
..	..

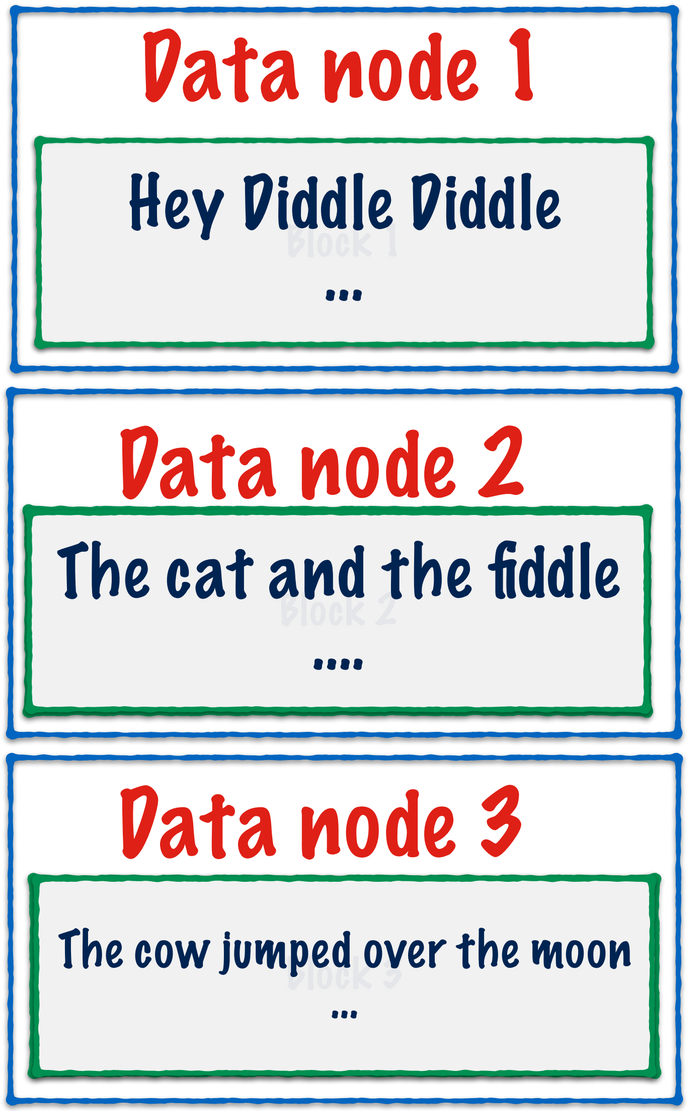
The text file has been divided into blocks and stored in HDFS



Name node

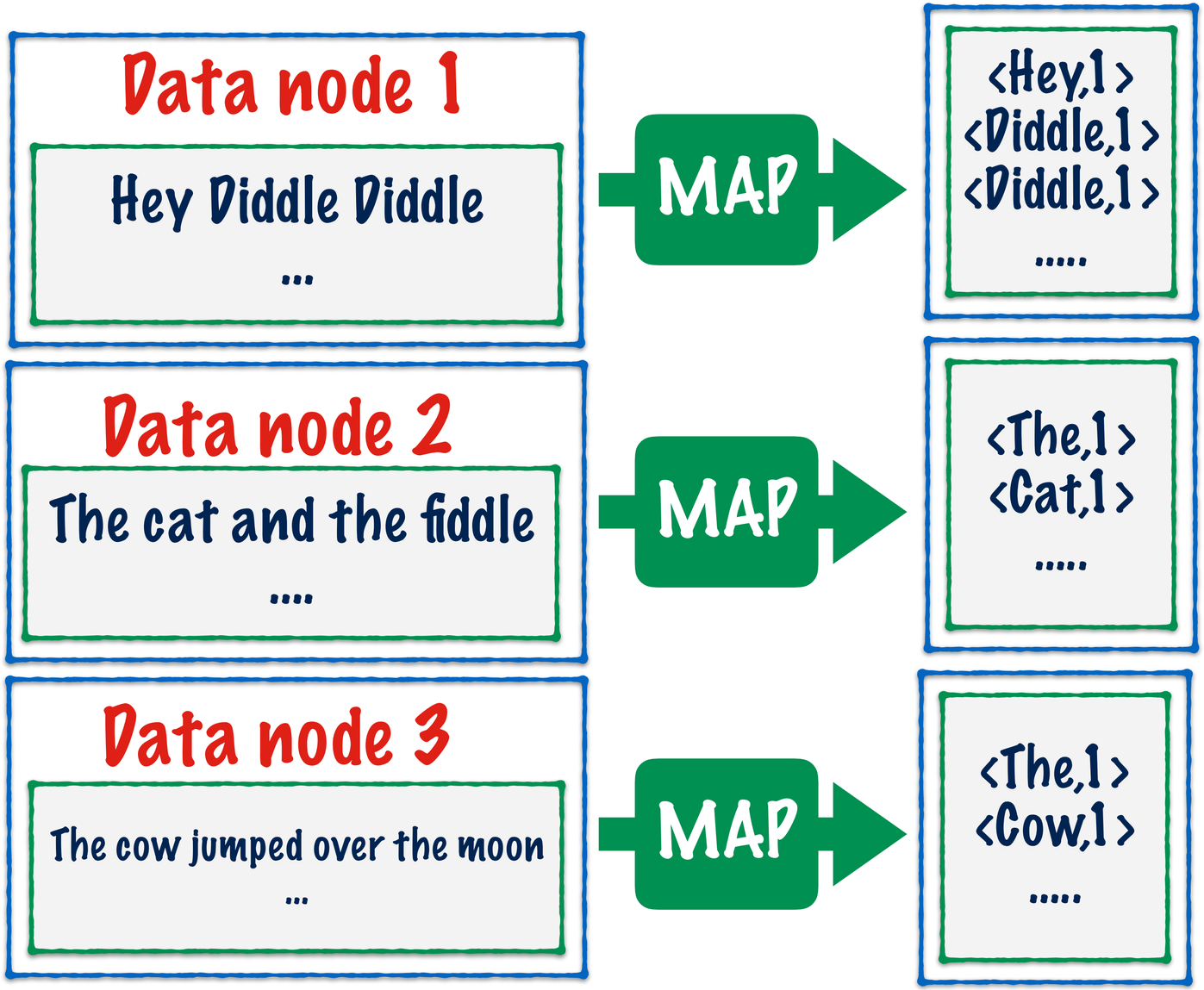
The name node stores metadata

Here is how the data flows in a MapReduce job



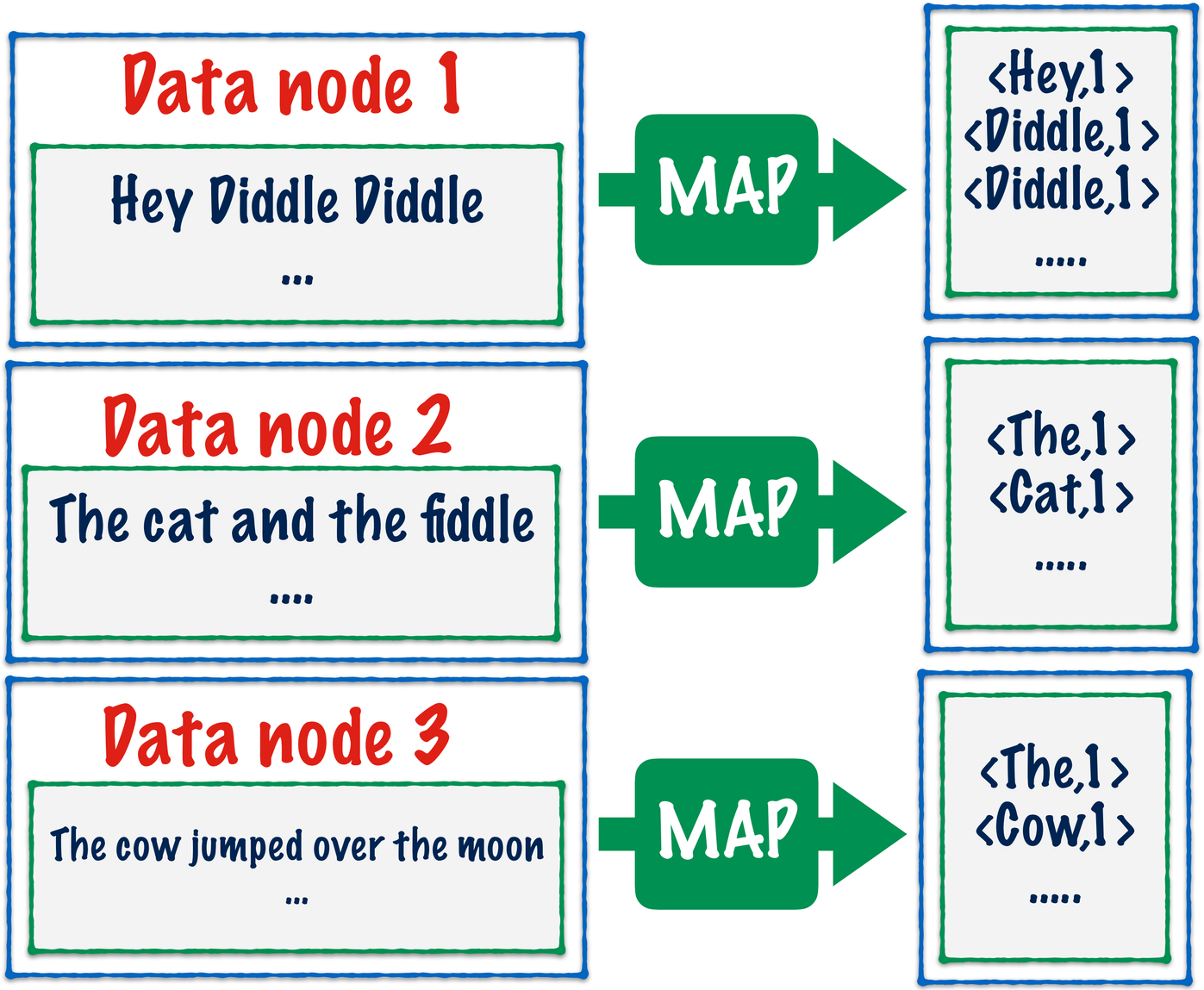
Each block here would represent a part of the text file

Here is a snapshot of some text from each block



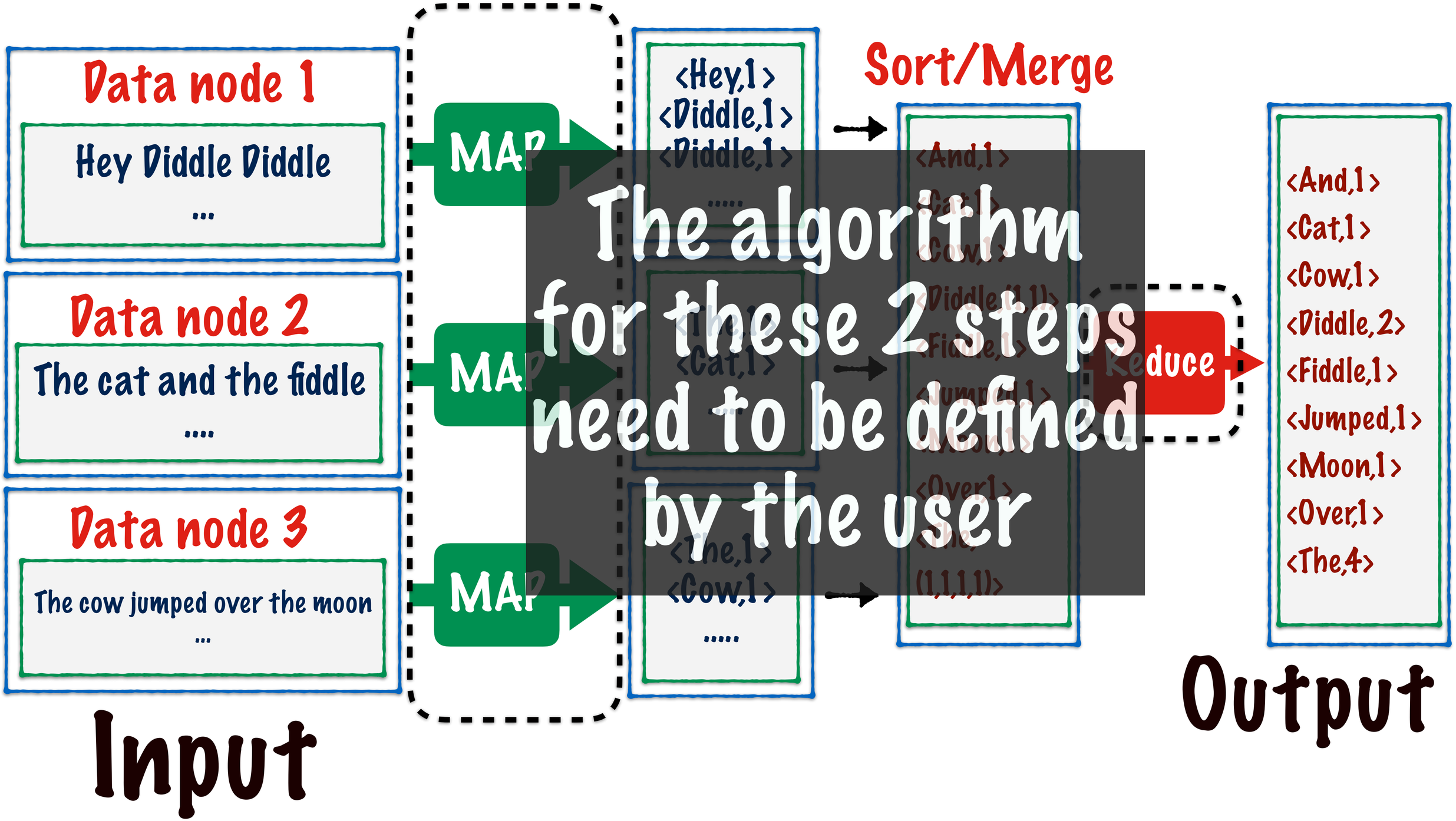
The map step will generate a list of key-value pairs on each node

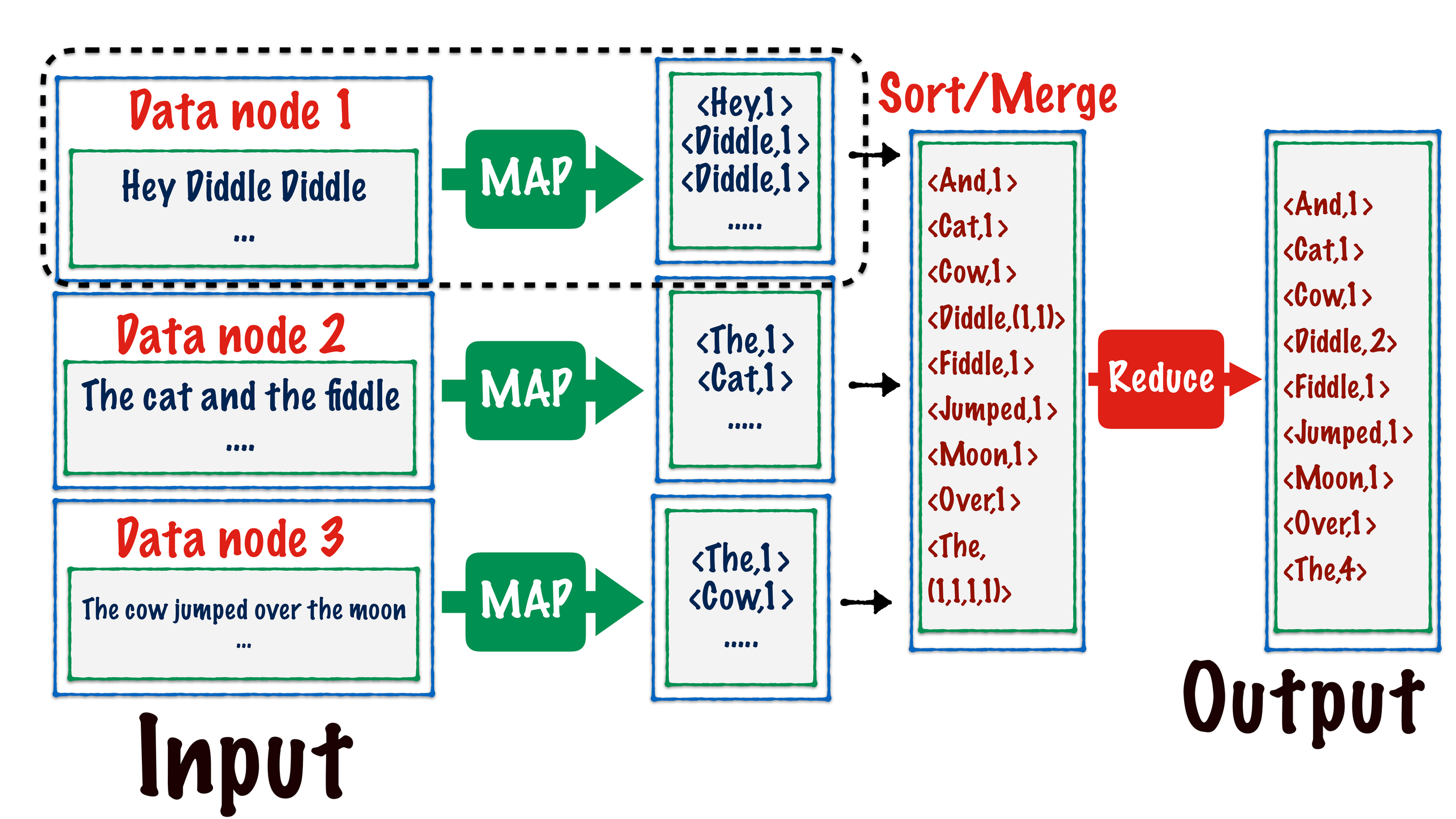
From now on all the inputs and outputs are formatted as <key,value> pairs

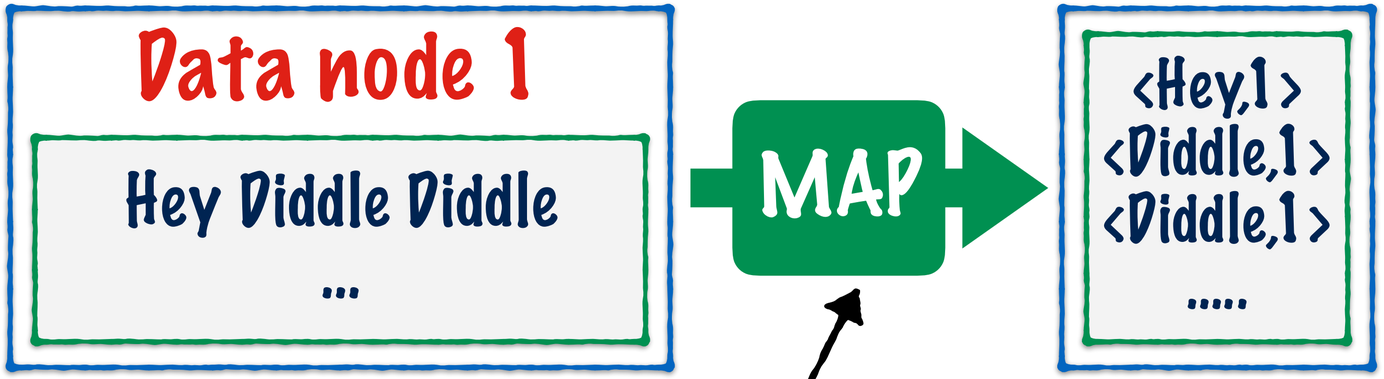


These are all copied over to one single node

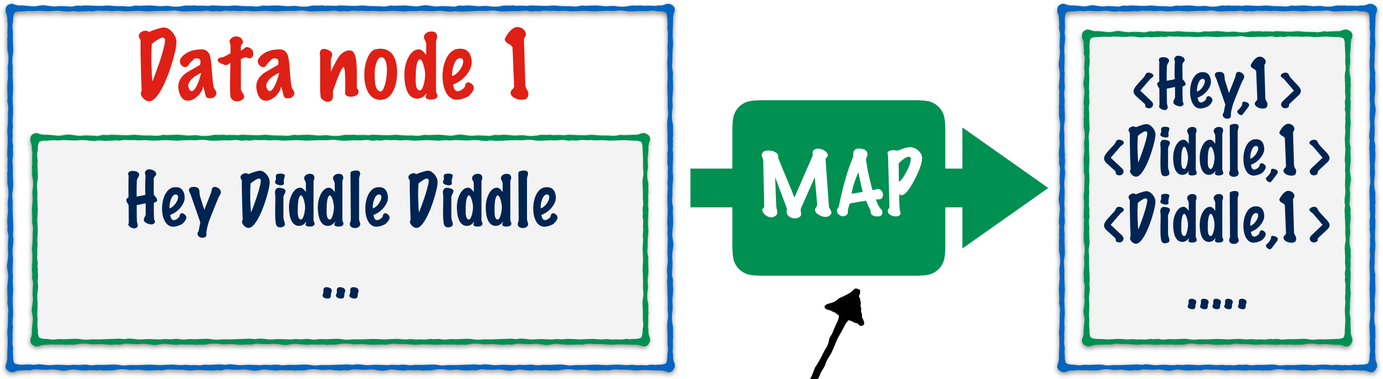
On that node an operation called Sort/Merge occurs


Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<Cat,1>
<The,1>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,(1,1,1,1)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
The algorithm for these 2 steps need to be defined by the user
Input
Output

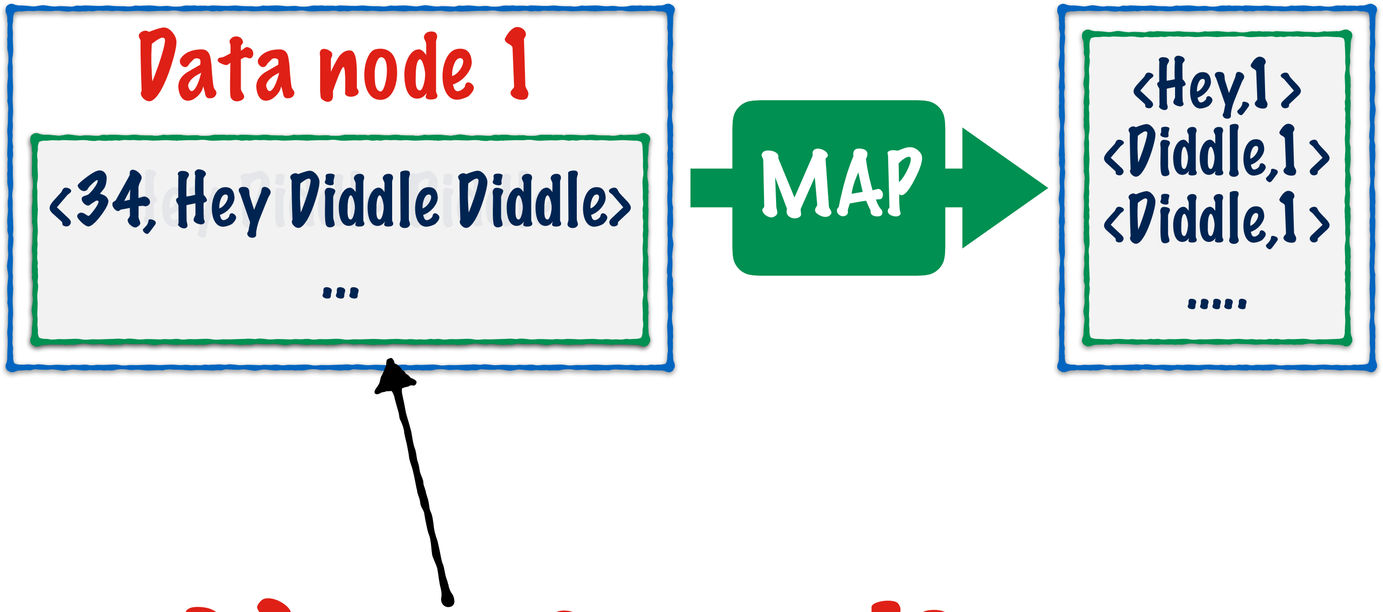

Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output



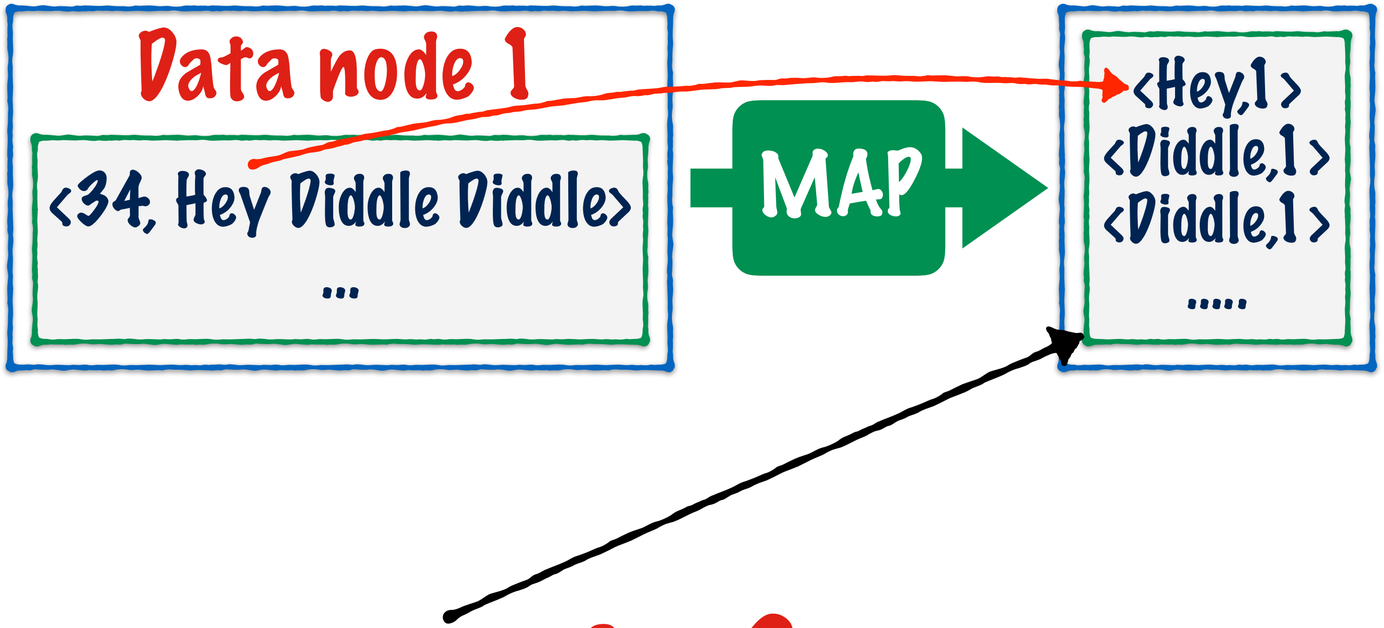
This function will run once for each line of the text file



Both the input and output need to be formatted as <key, value> pairs

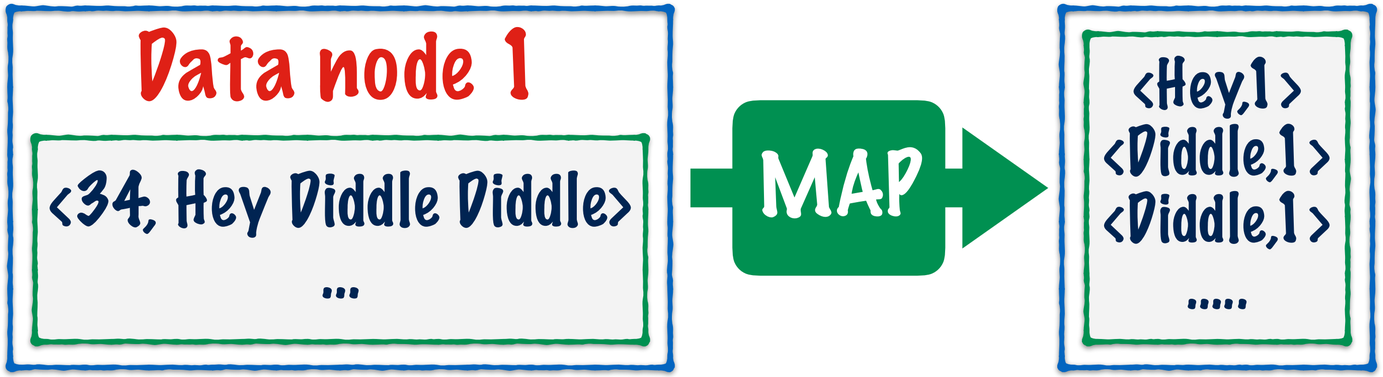


<lineNumber, line>

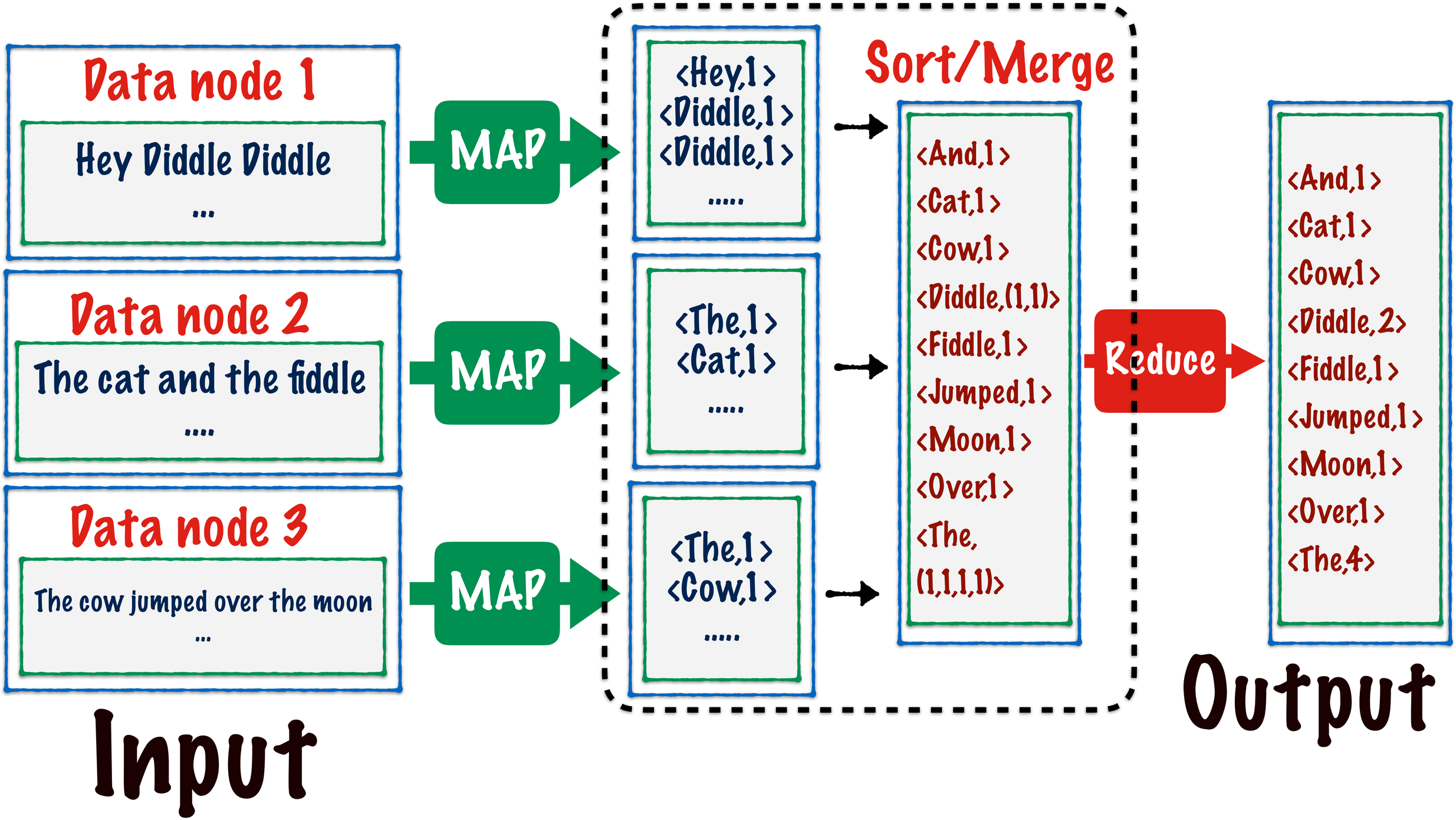


<word, 1>

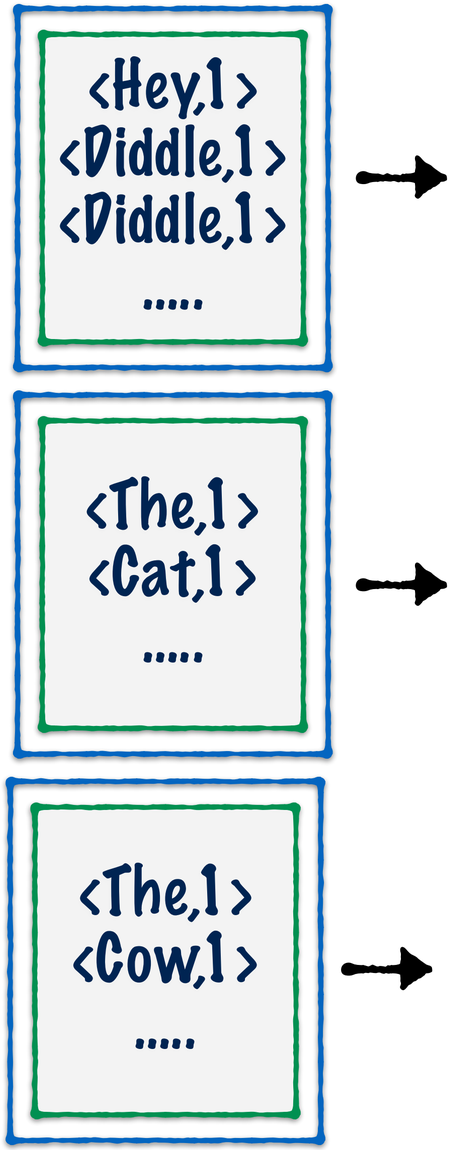
There will be one such pair for each word in the line



This operation can run in parallel on each data node, there is no interdependency in the inputs and outputs


Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output

Sort/Merge

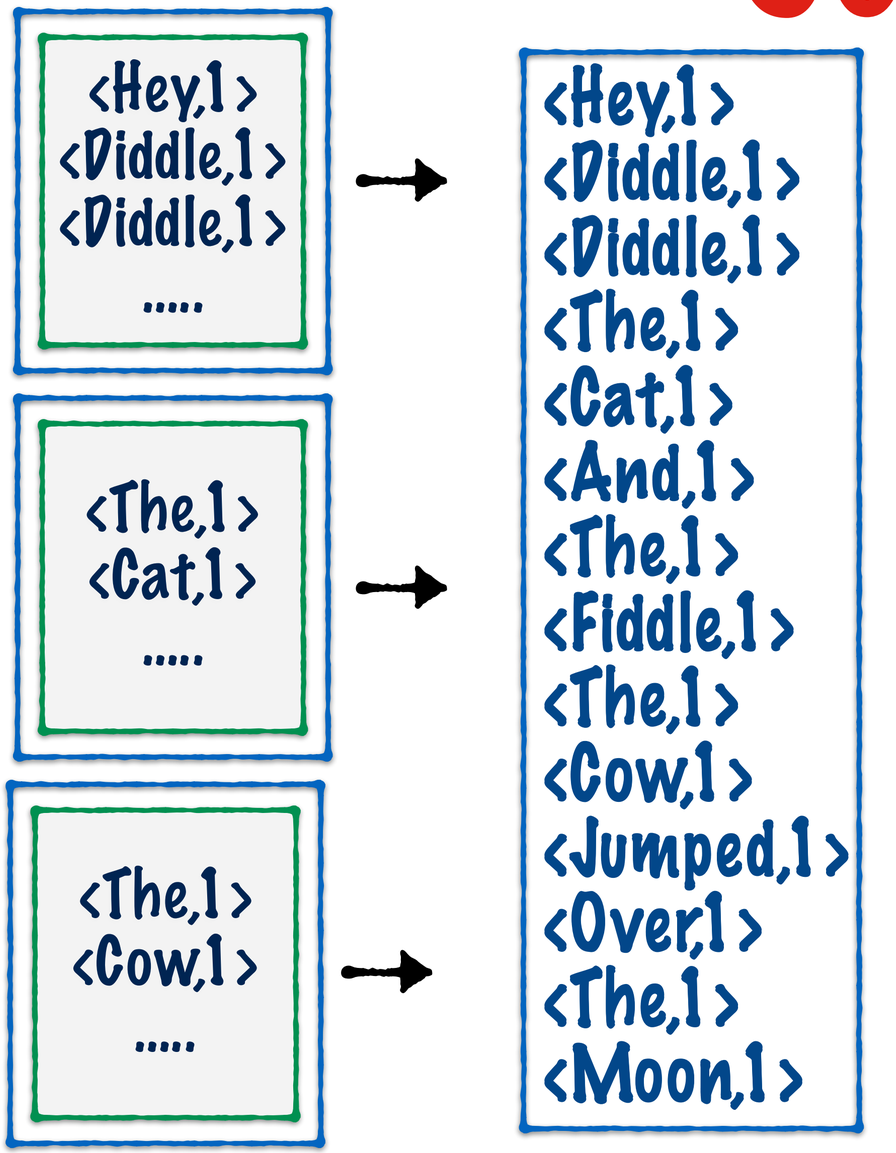


At the end of the map phase, we have a set of key-value pairs from each data node

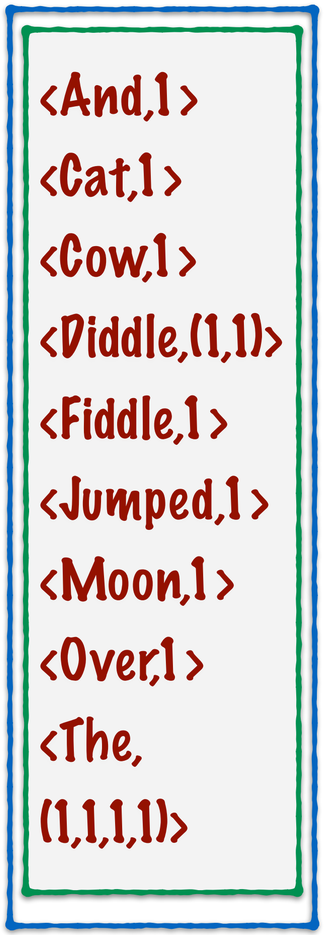
All of these results are first copied over to a single node

<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>

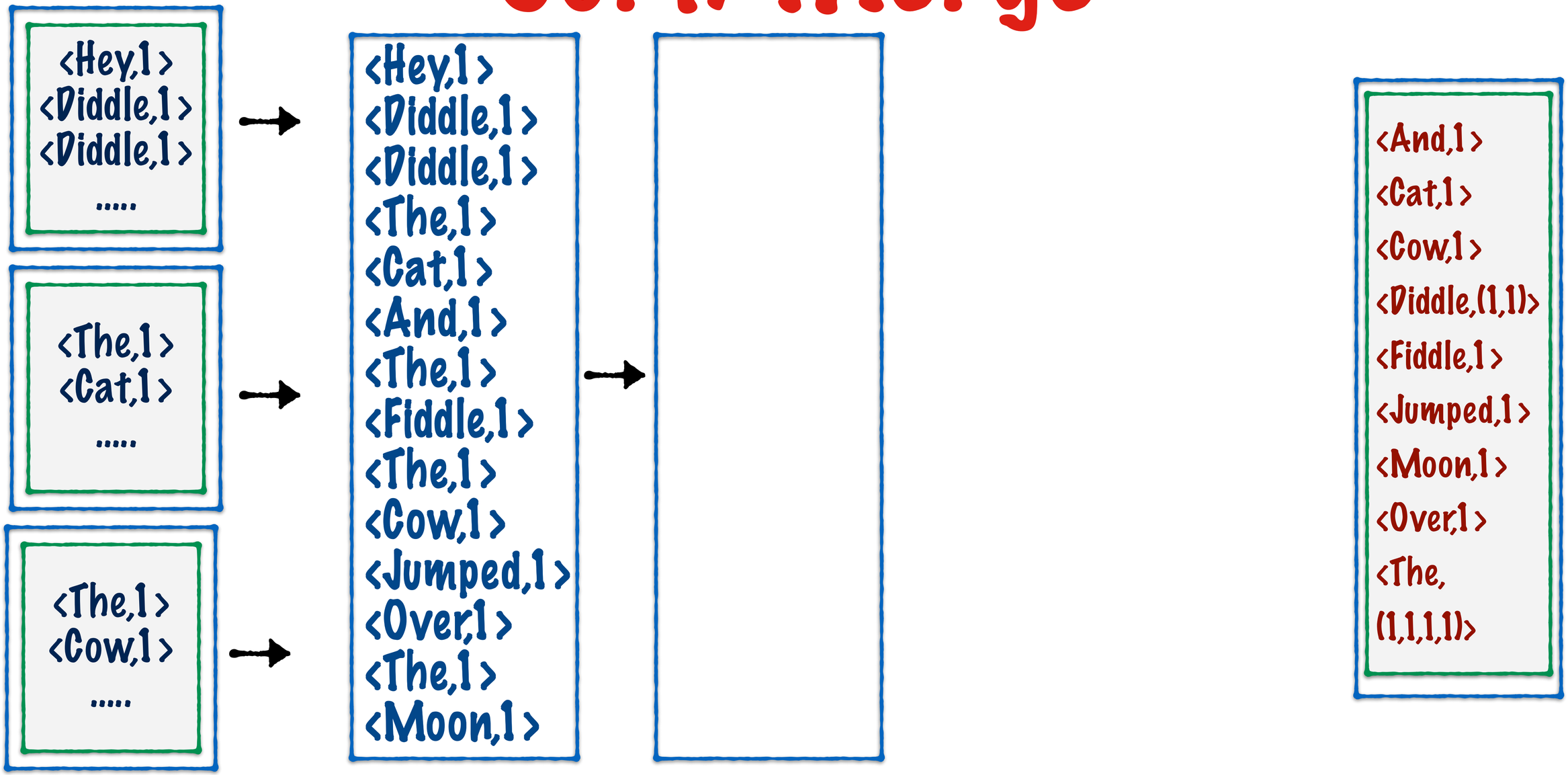
Sort/Merge



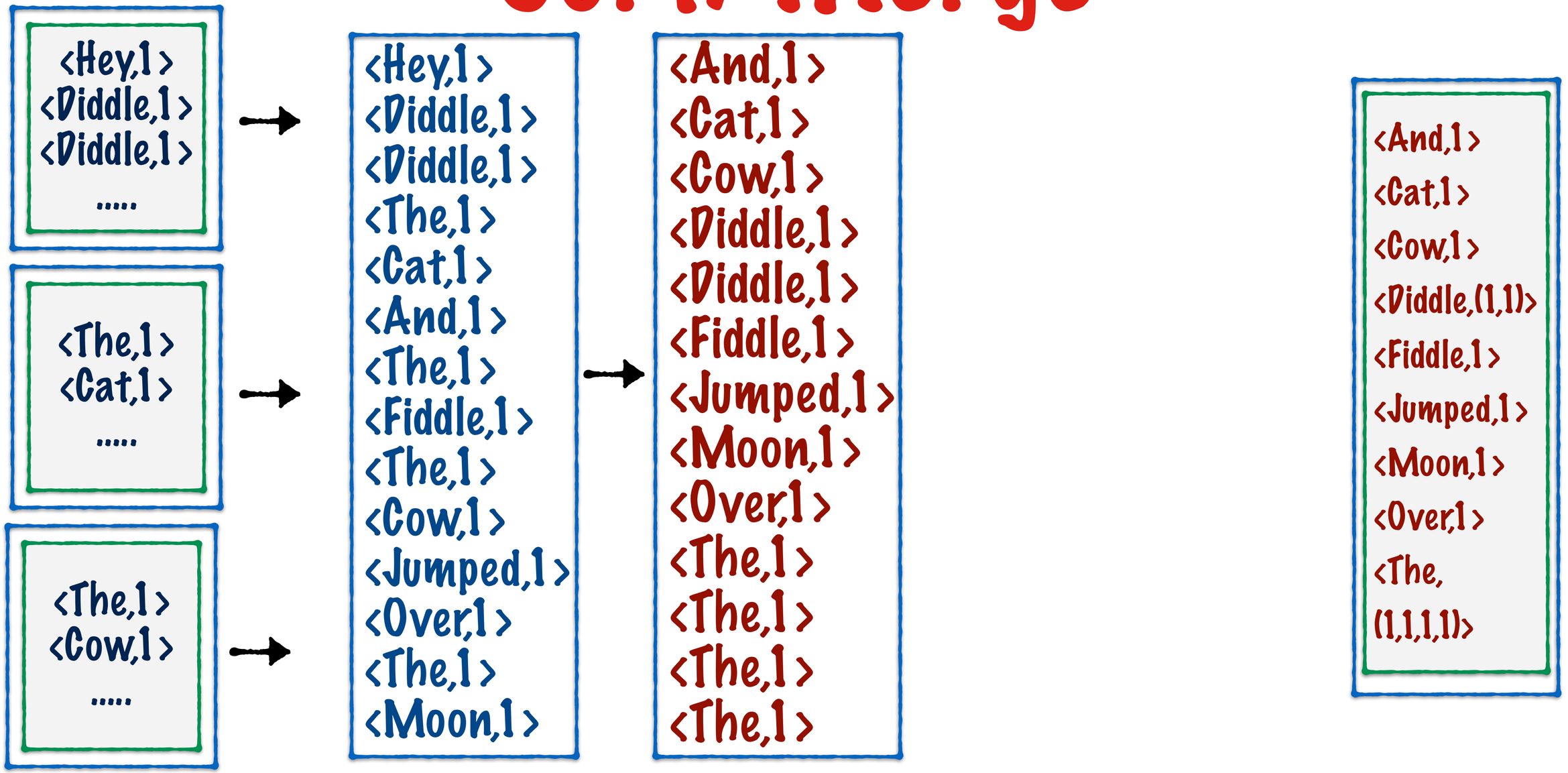
The data is sorted by **key**



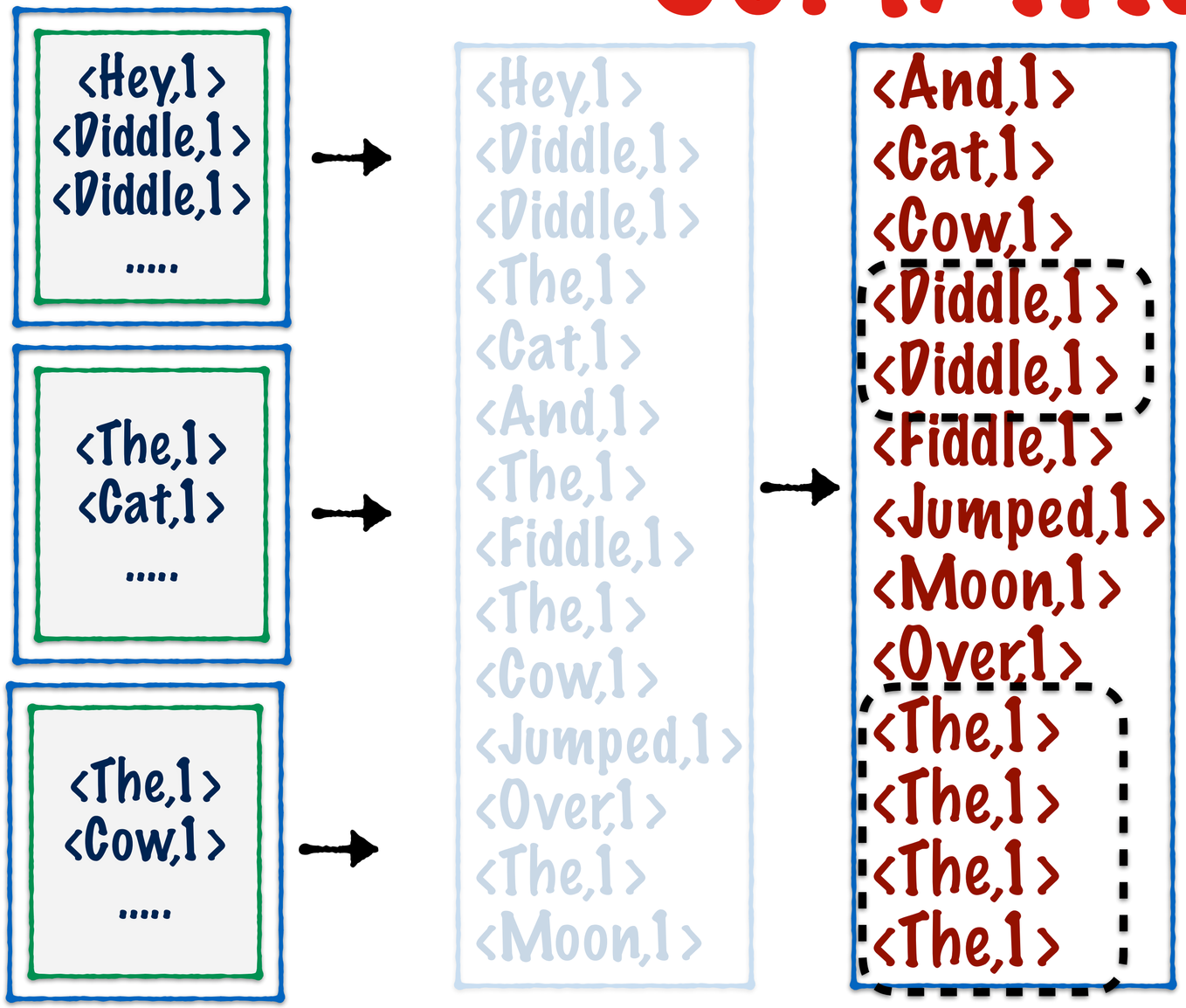
Sort/Merge


<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
<Hey,1>
<Diddle,1>
<Diddle,1>
<The,1>
<Cat,1>
<And,1>
<The,1>
<Fiddle,1>
<The,1>
<Cow,1>
<Jumped,1>
<Over,1>
<The,1>
<Moon,1>
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>

Sort/Merge

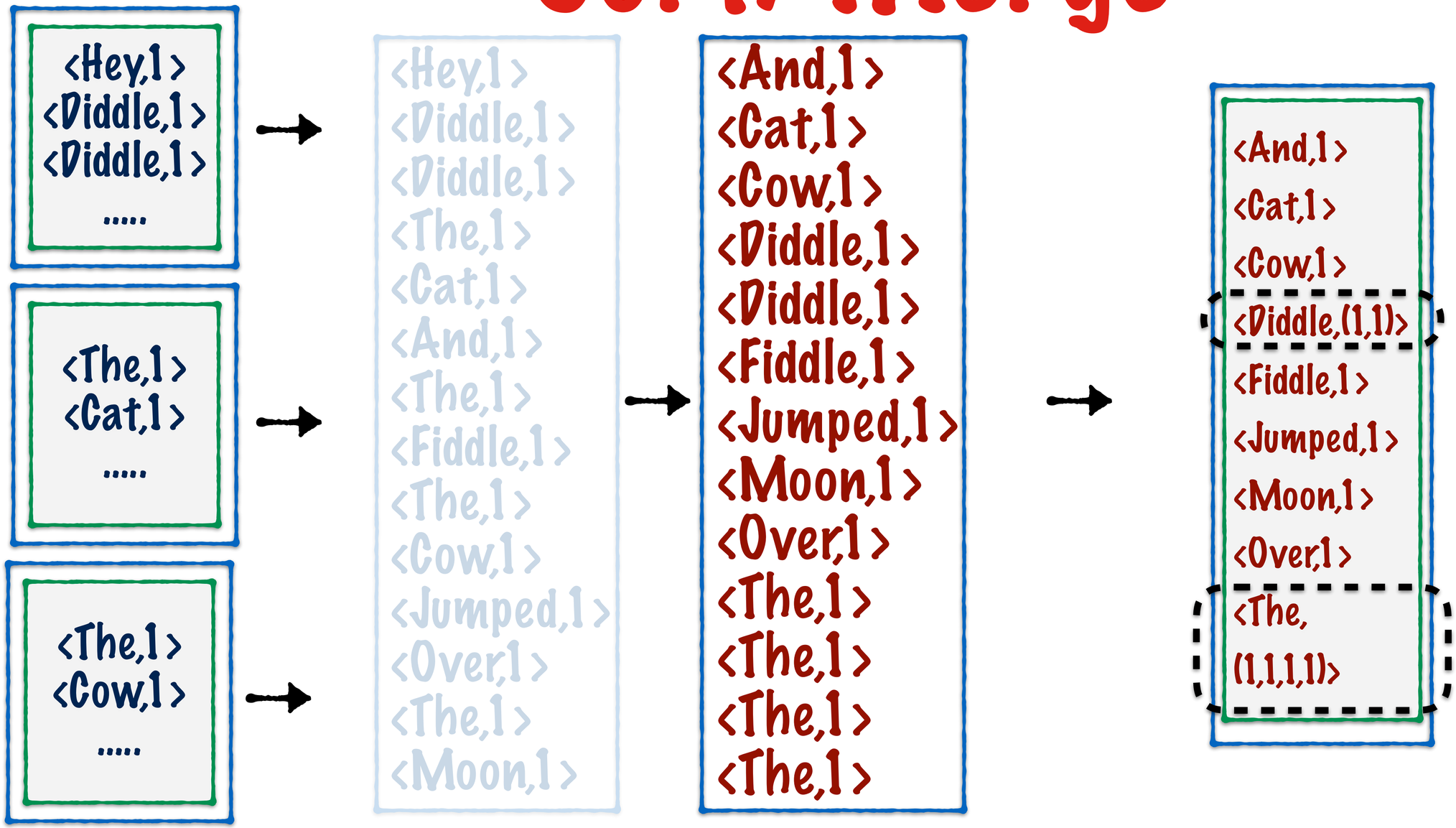

<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
<Hey,1>
<Diddle,1>
<Diddle,1>
<The,1>
<Cat,1>
<And,1>
<The,1>
<Fiddle,1>
<The,1>
<Cow,1>
<Jumped,1>
<Over,1>
<The,1>
<Moon,1>
<And,1>
<Cat,1>
<Cow,1>
<Diddle,1>
<Diddle,1>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,1>
<The,1>
<The,1>
<The,1>
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>

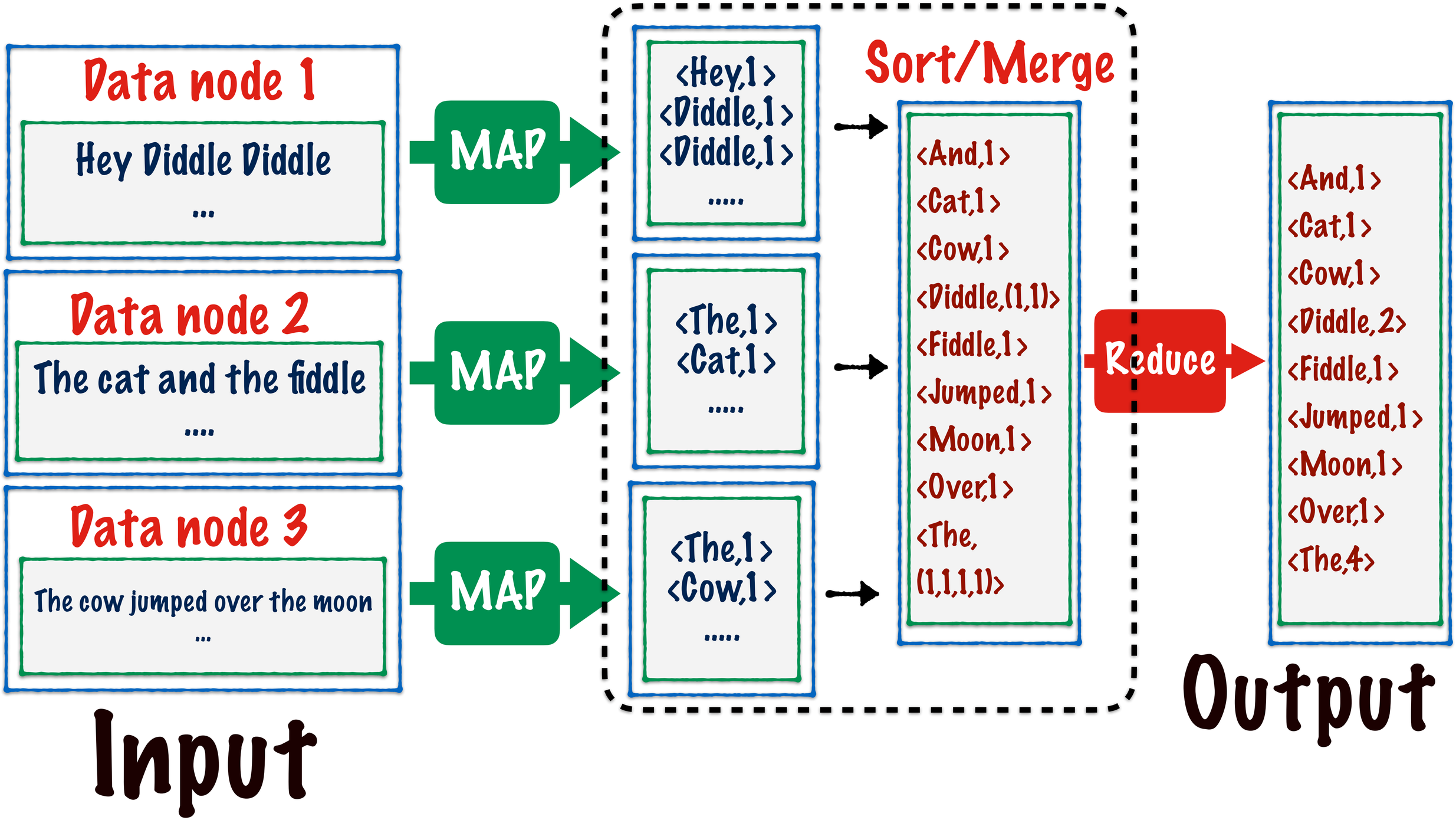
Sort/Merge

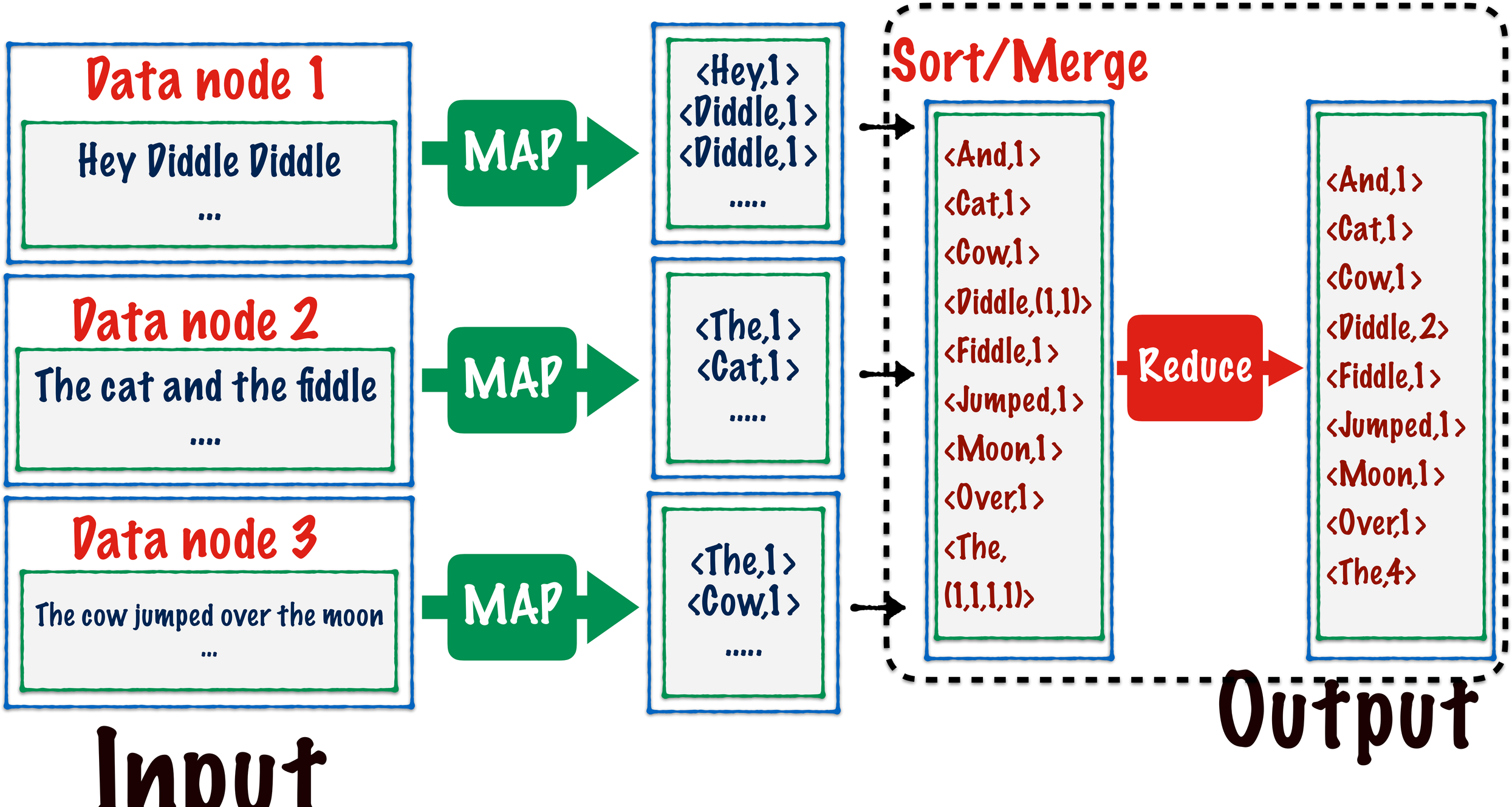

<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
<Hey,1>
<Diddle,1>
<Diddle,1>
<The,1>
<Cat,1>
<And,1>
<The,1>
<Fiddle,1>
<The,1>
<Cow,1>
<Jumped,1>
<Over,1>
<The,1>
<Moon,1>
<And,1>
<Cat,1>
<Cow,1>
<Diddle,1>
<Diddle,1>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,1>
<The,1>
<The,1>
<The,1>

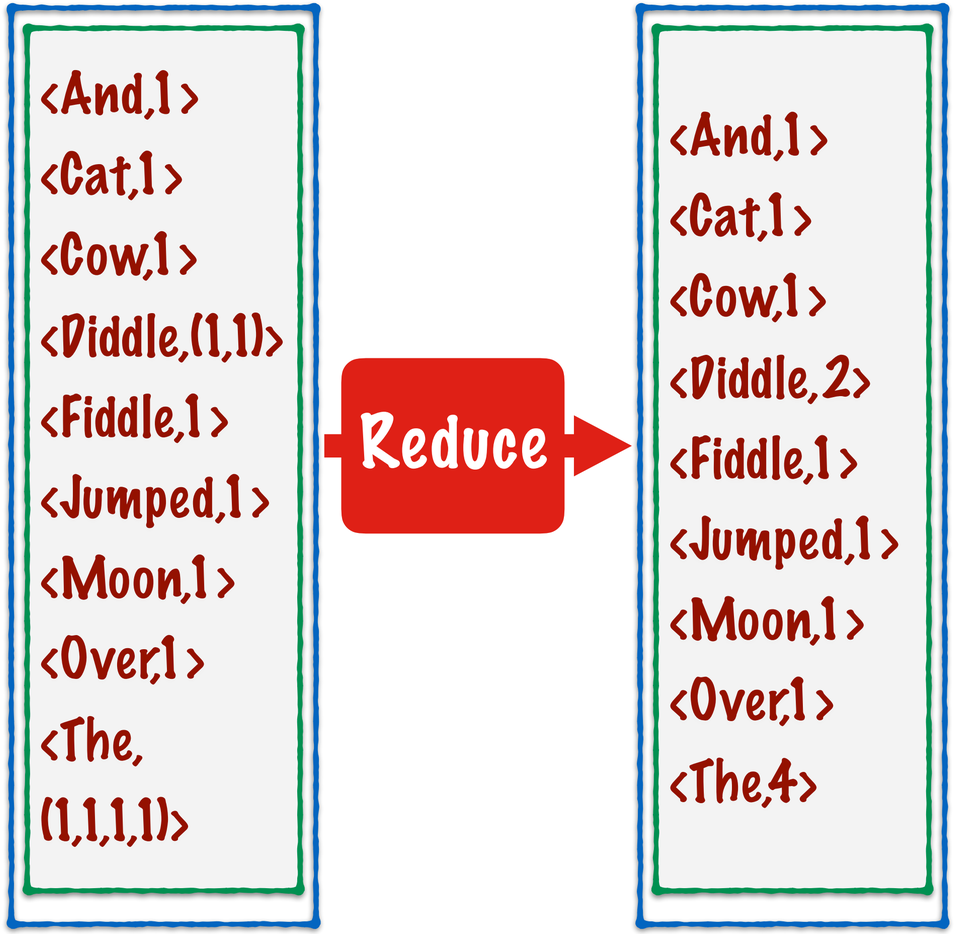

Key-Value pairs with the same key are merged

Sort/Merge

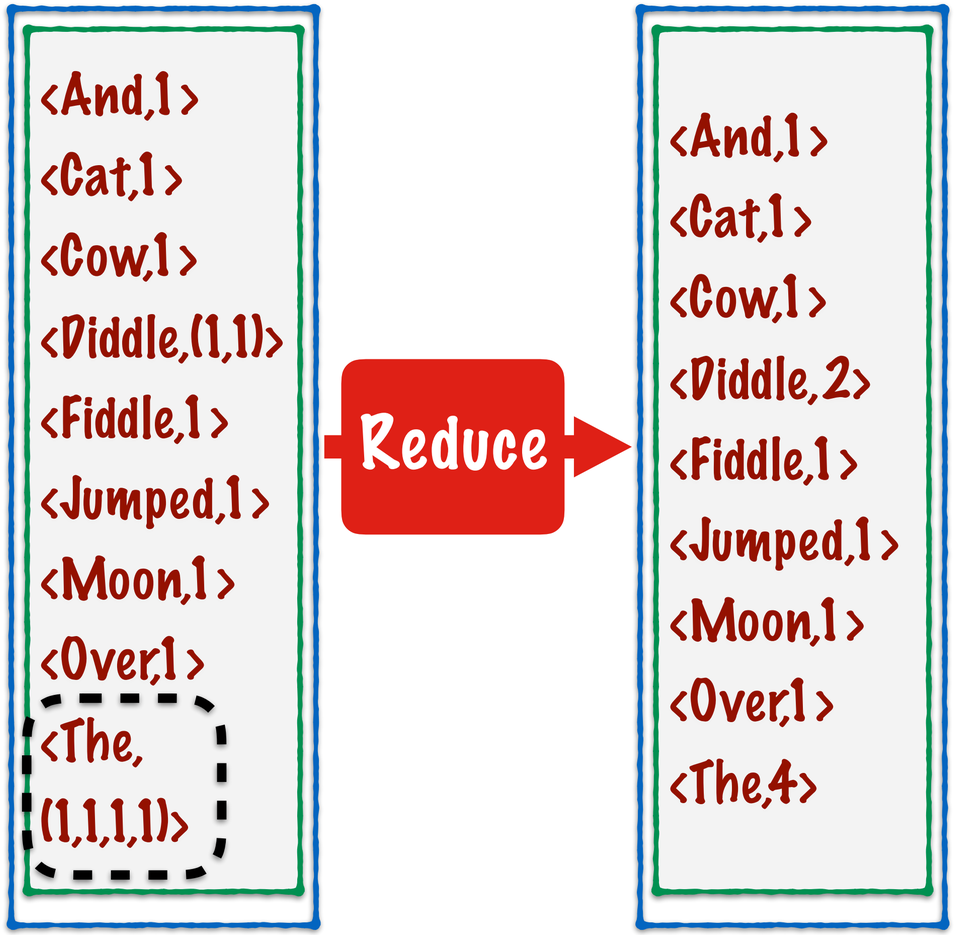

<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
<Hey,1>
<Diddle,1>
<Diddle,1>
<The,1>
<Cat,1>
<And,1>
<The,1>
<Fiddle,1>
<The,1>
<Cow,1>
<Jumped,1>
<Over,1>
<The,1>
<Moon,1>
<And,1>
<Cat,1>
<Cat,1>
<Cow,1>
<Diddle,1>
<Diddle,1>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,1>
<The,1>
<The,1>
<The,1>
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>


Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output

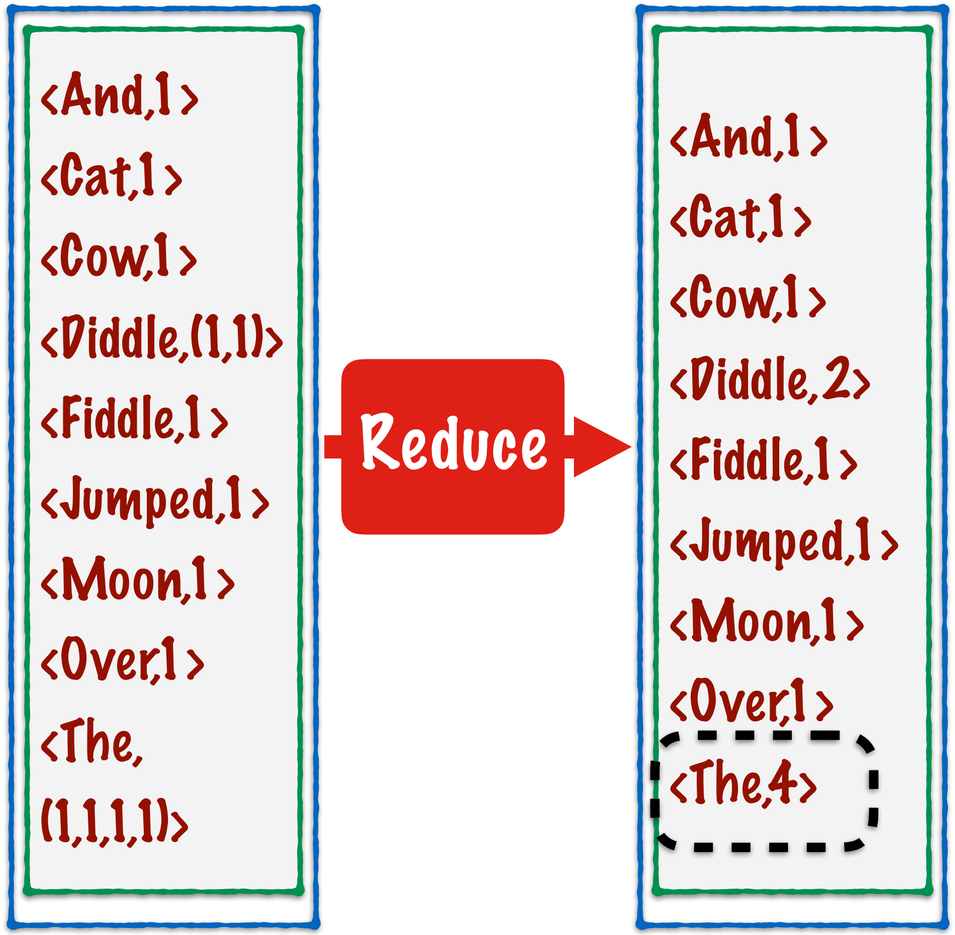

Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output



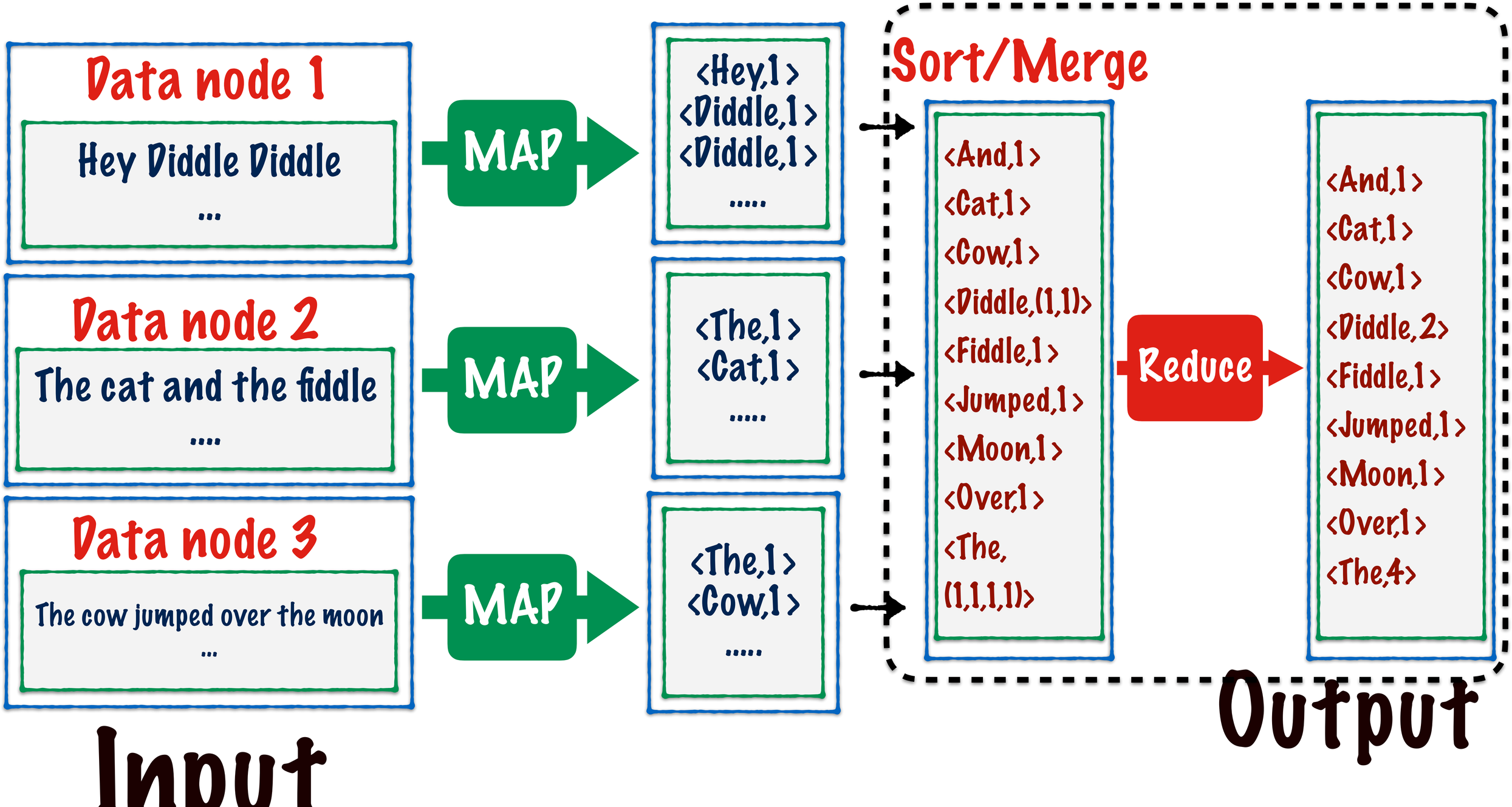
reduce() will run on each pair generated by the Sort/Merge step

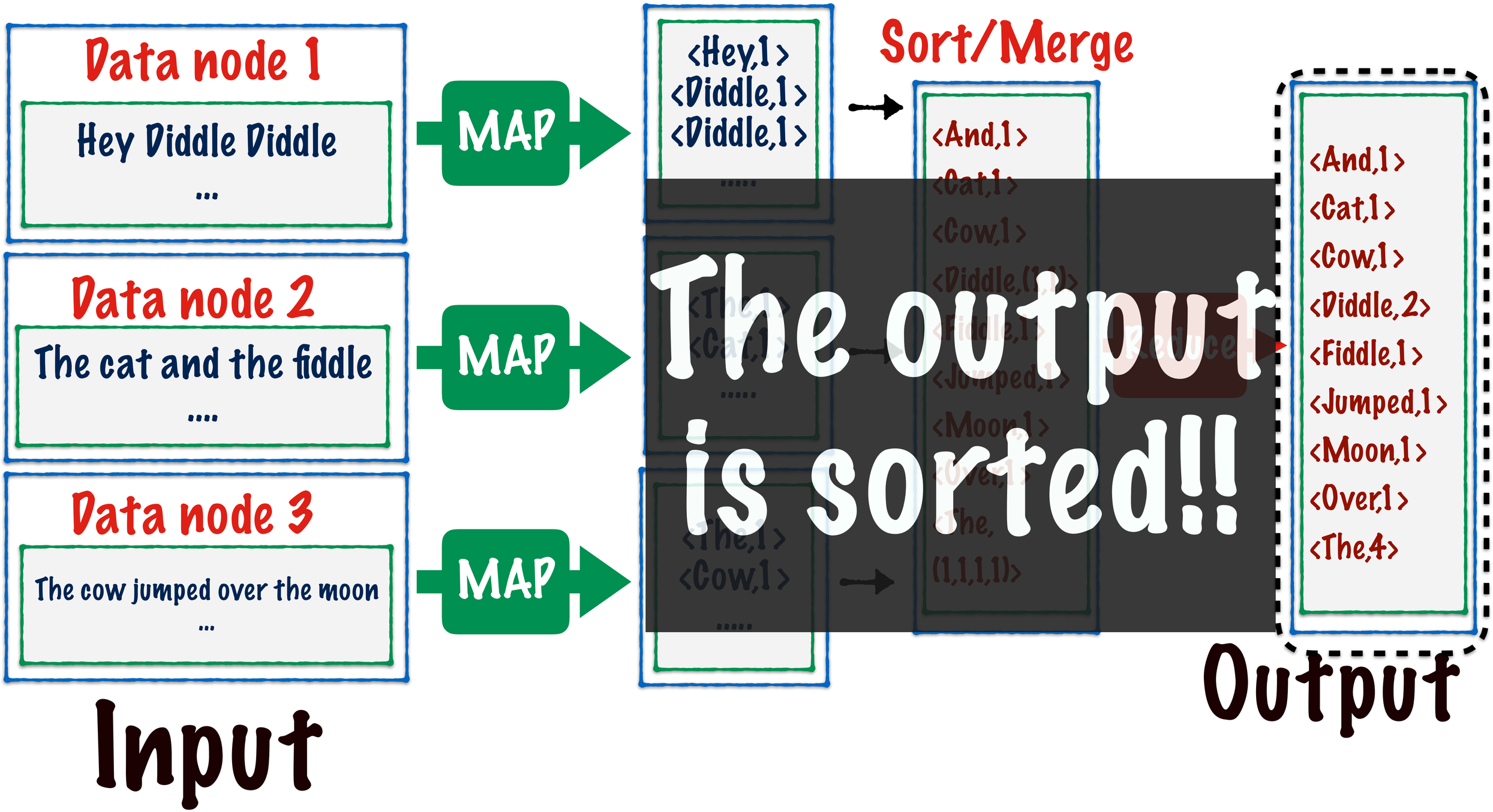

<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>

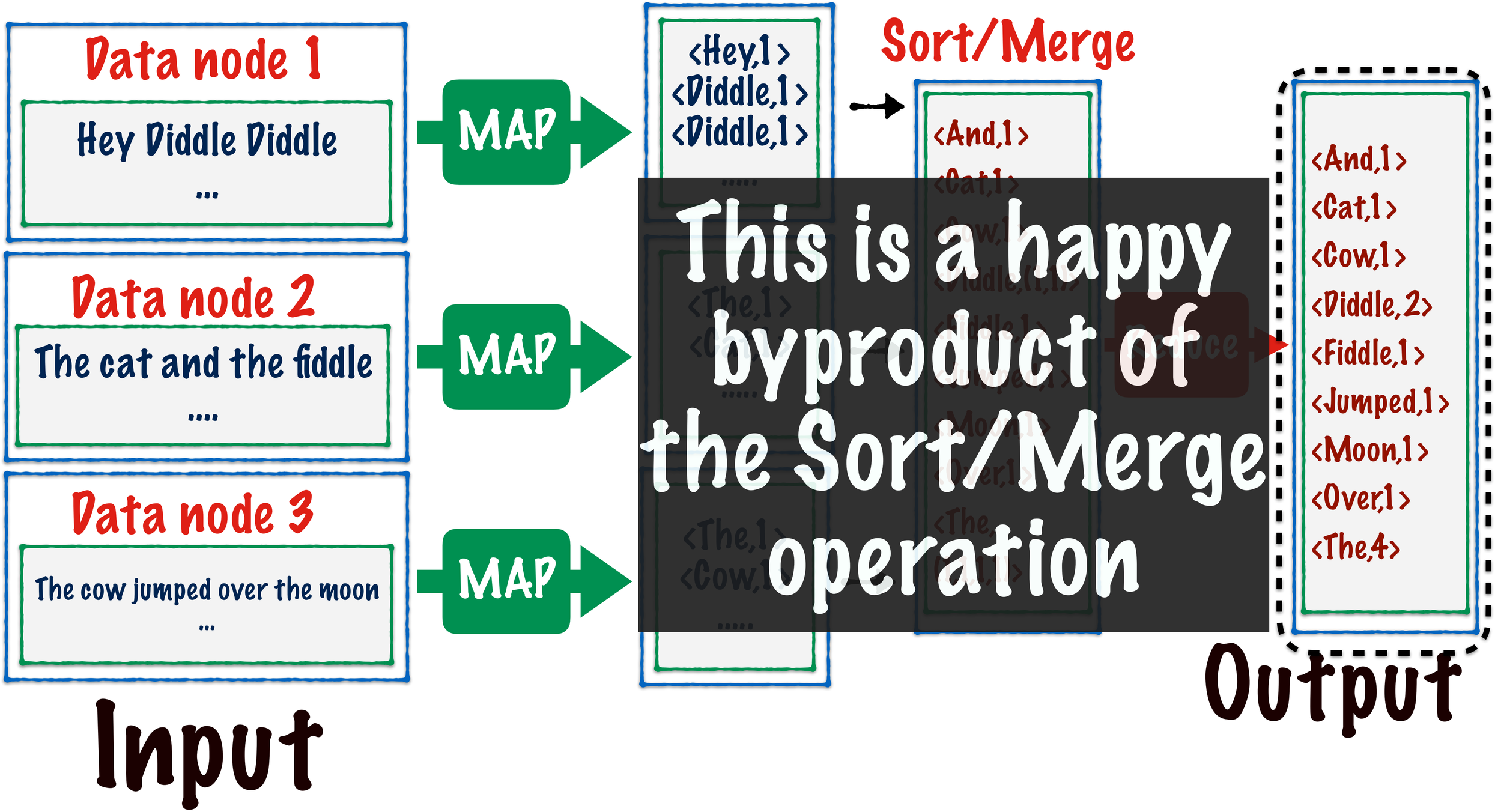
<word, list>

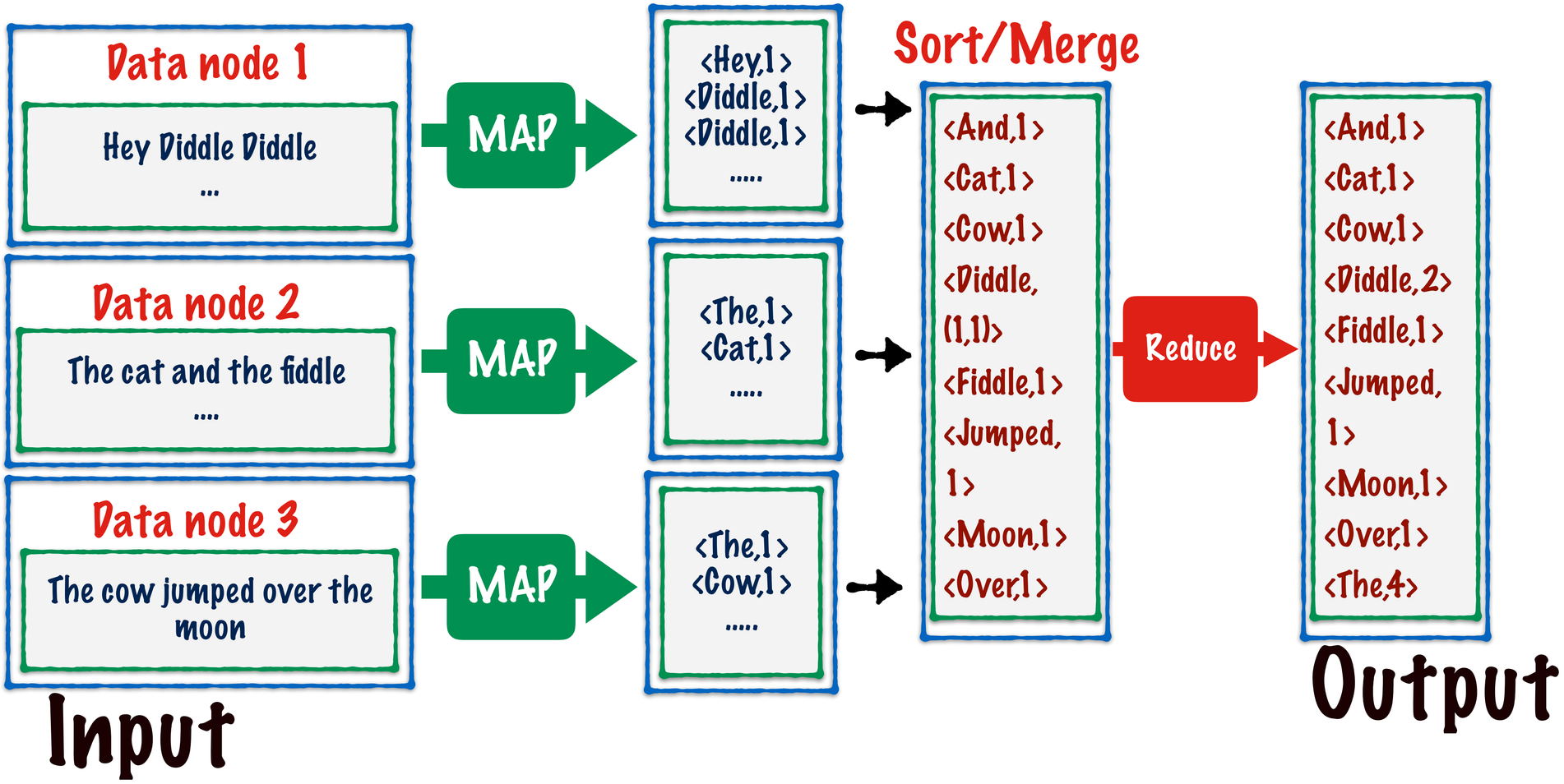


<word, list> <word, count>

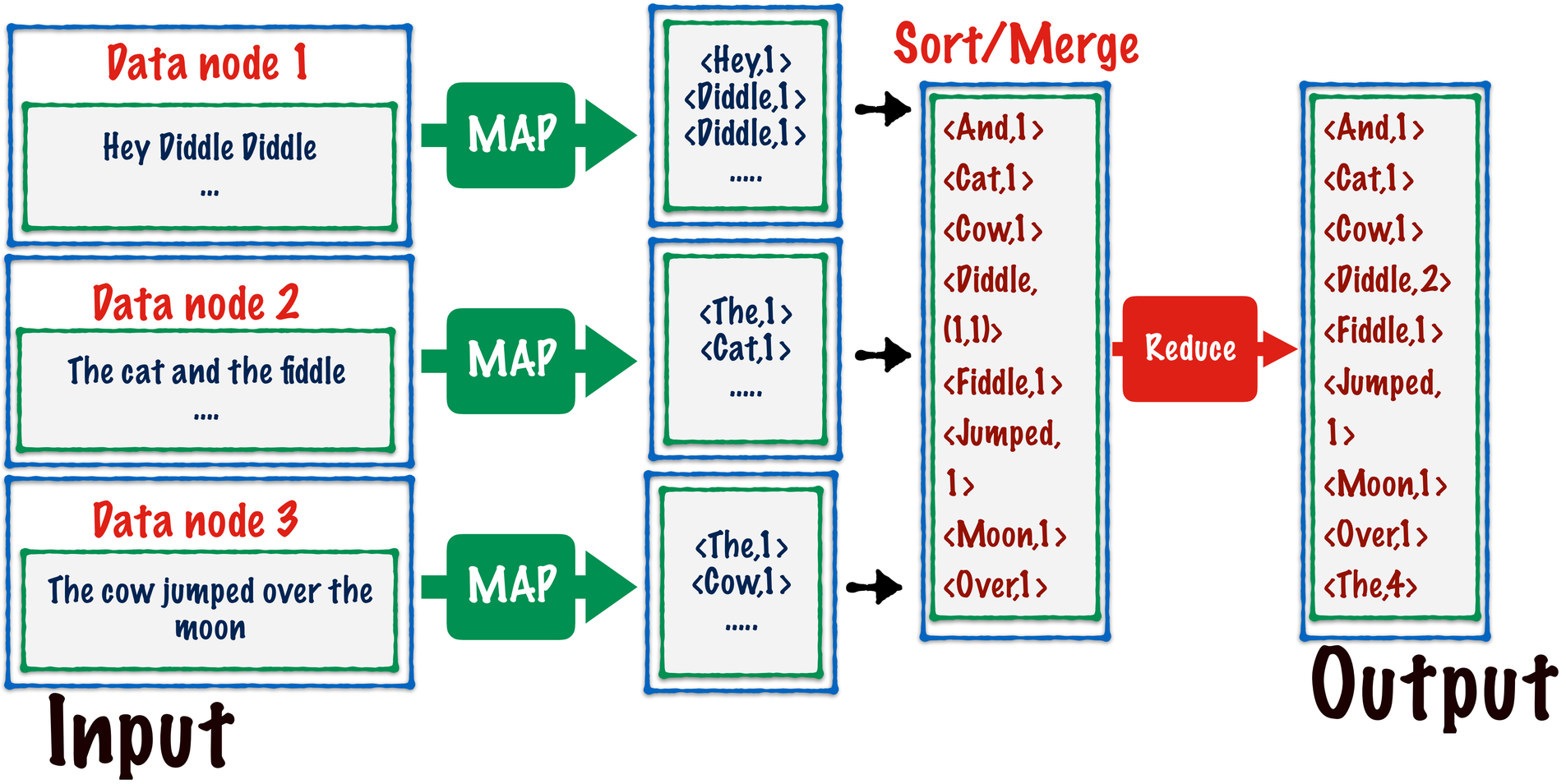

Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output


Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
<The,1>
<Cat,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,1>
<Cow,1>
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>
The output is sorted!!
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output

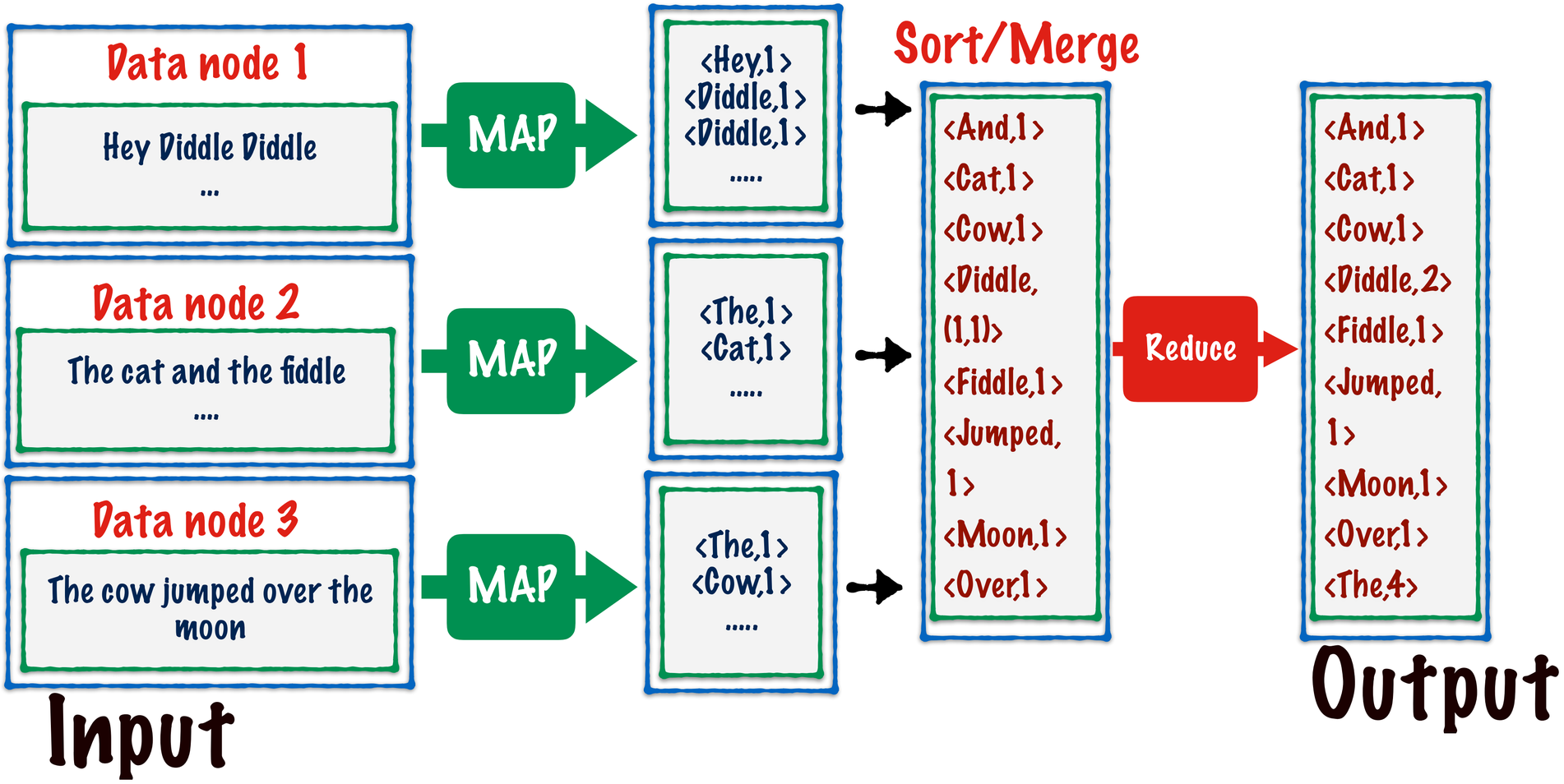

Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
Sort/Merge
<And,1>
This is a happy byproduct of the Sort/Merge operation
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output



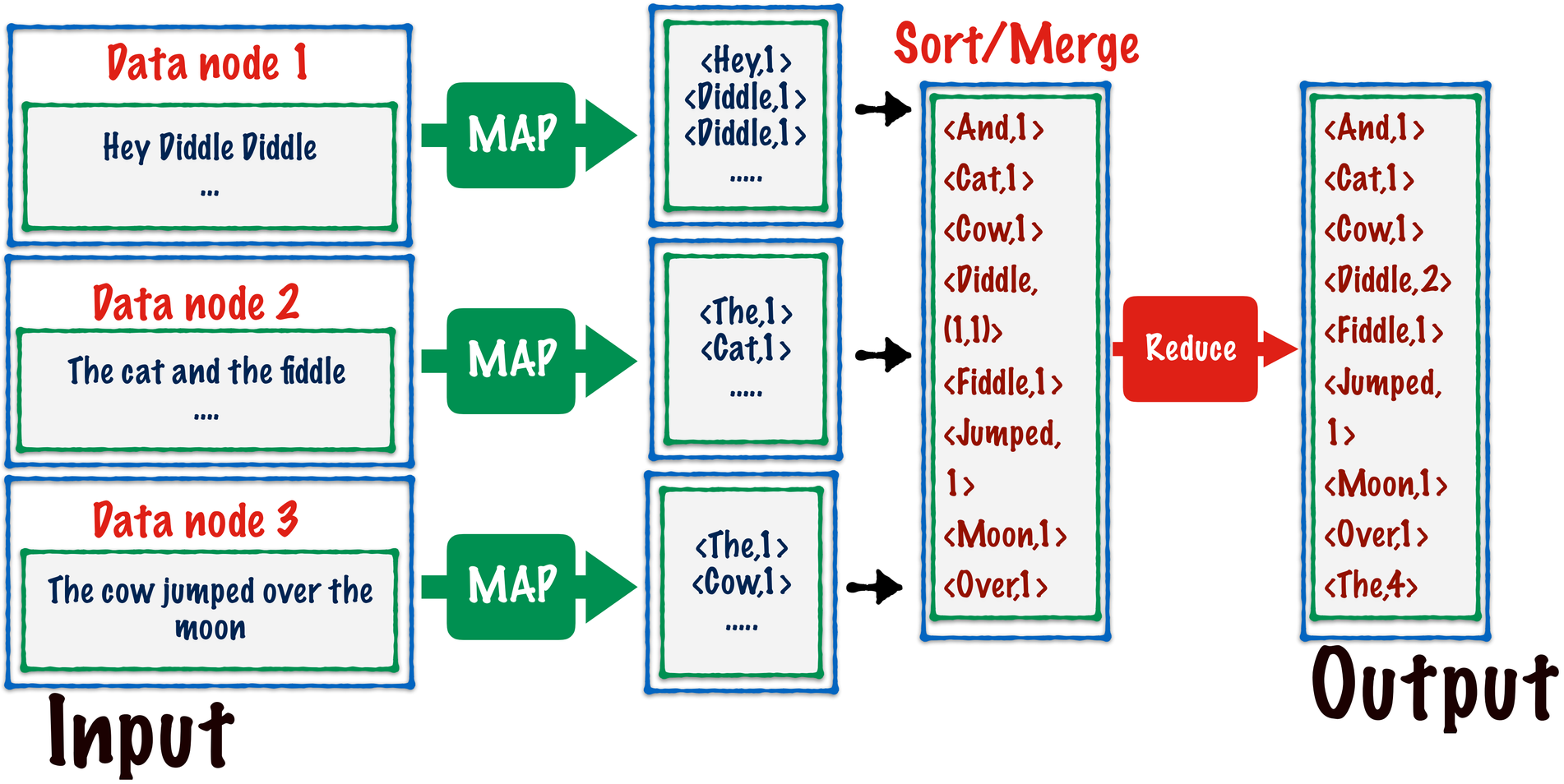
This is the general idea behind every MapReduce operation



The map function is chosen such that it can run in parallel on all the nodes



The output of map should be key, value pairs



The reduce() function will combine all the values for the same key in some way

MapReduce Hello World

Let's write our first MapReduce program

MapReduce Programs are
usually written in Java

Hadoop has an API called Streaming
that can be used for Ruby, Python etc

We'll get to that a little later

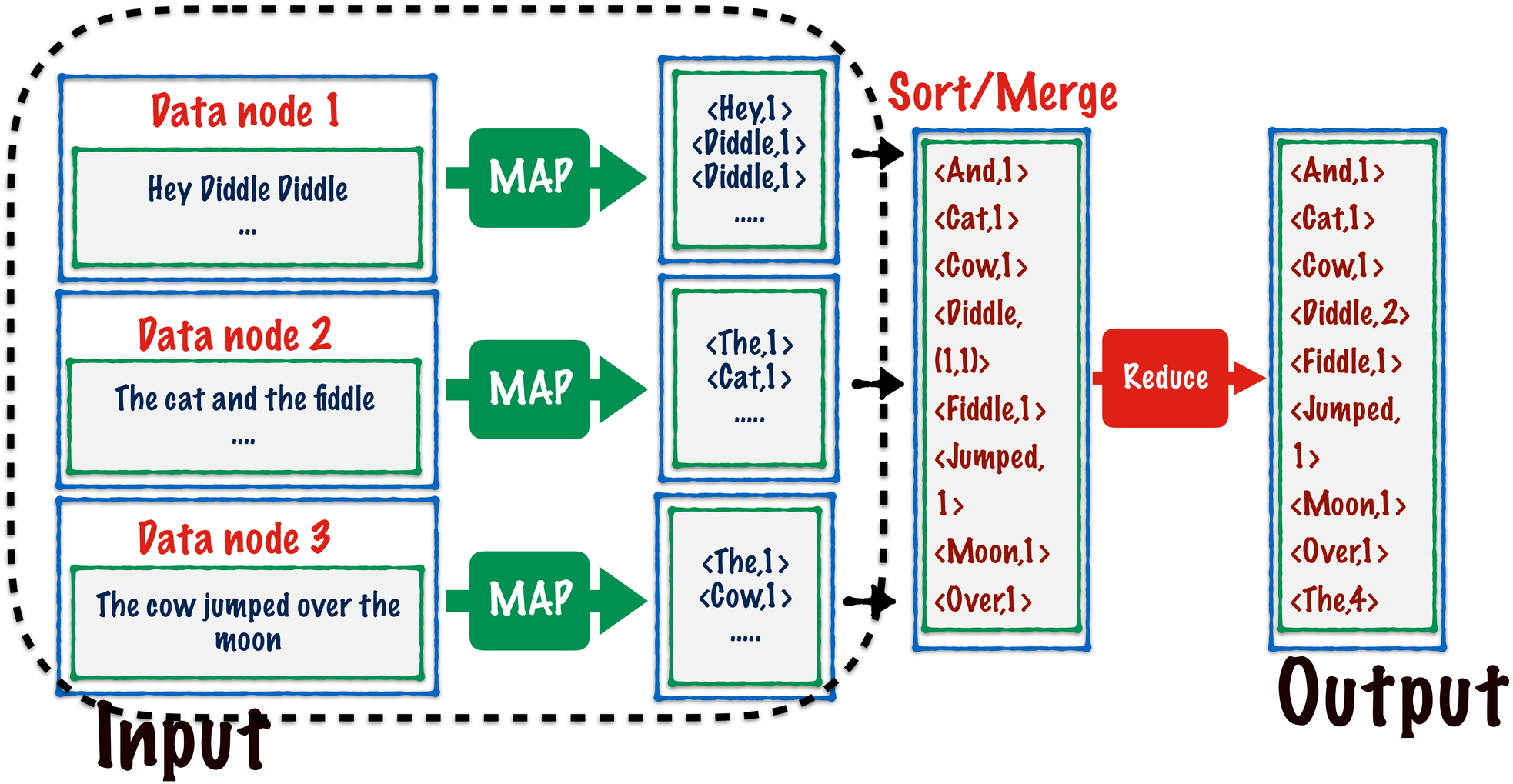
Objective: Create a **Frequency Distribution** of words in a text file

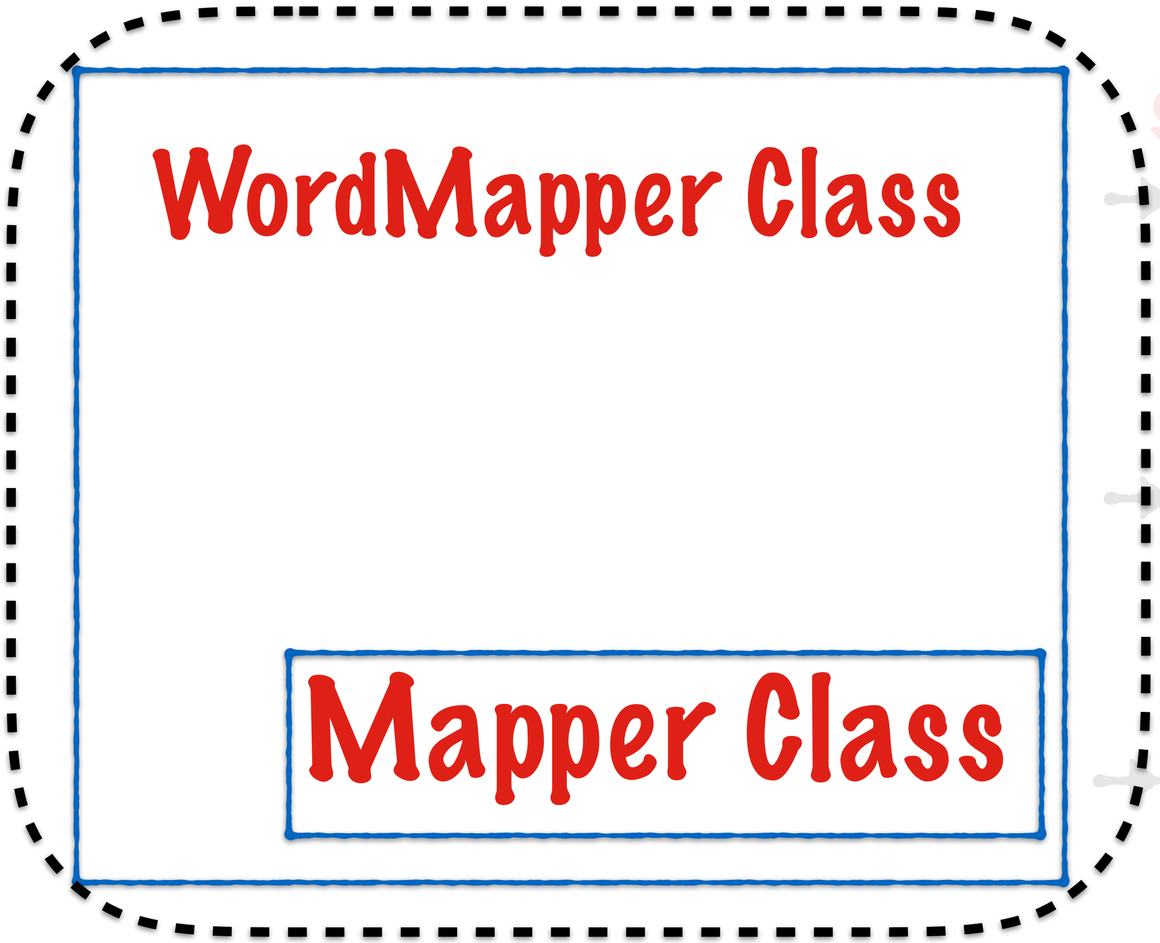
Step 1: Write a map() function

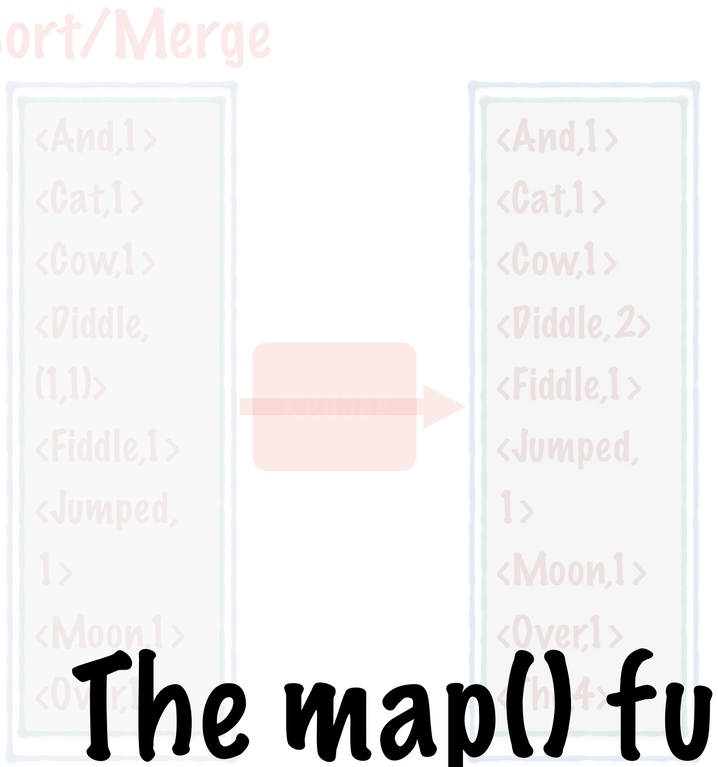
Step 2: Write a reduce() function

Step 3: Setup a driver that points to our map and reduce implementations

Objective: Create a Frequency Distribution of words in a text file





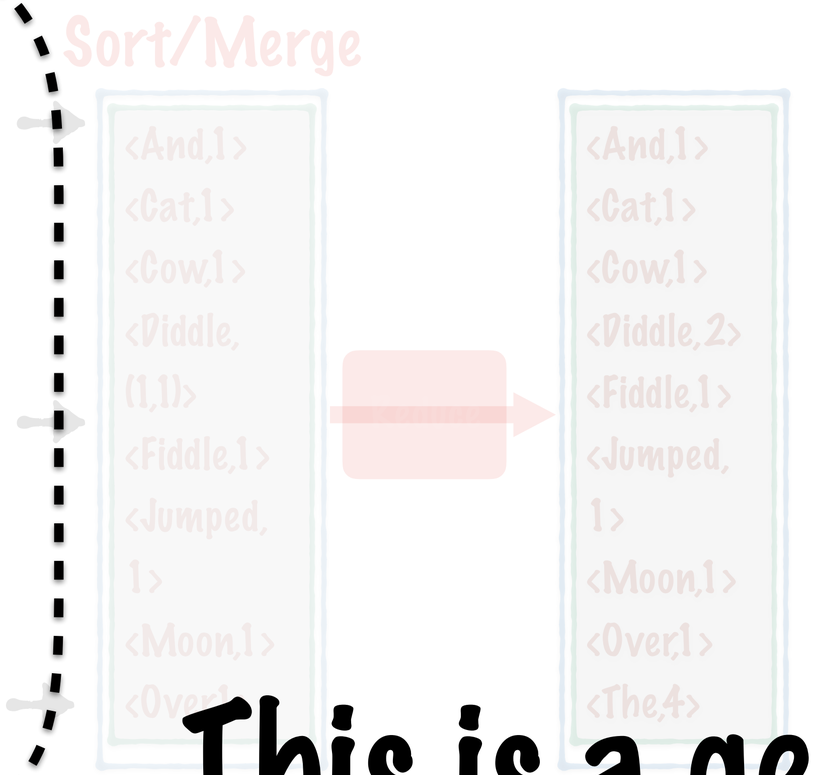


The map() function is implemented in a class that extends the Mapper Class

WordMapper Class

<input key type,
input value type,
output key type,
output value type>

Mapper Class

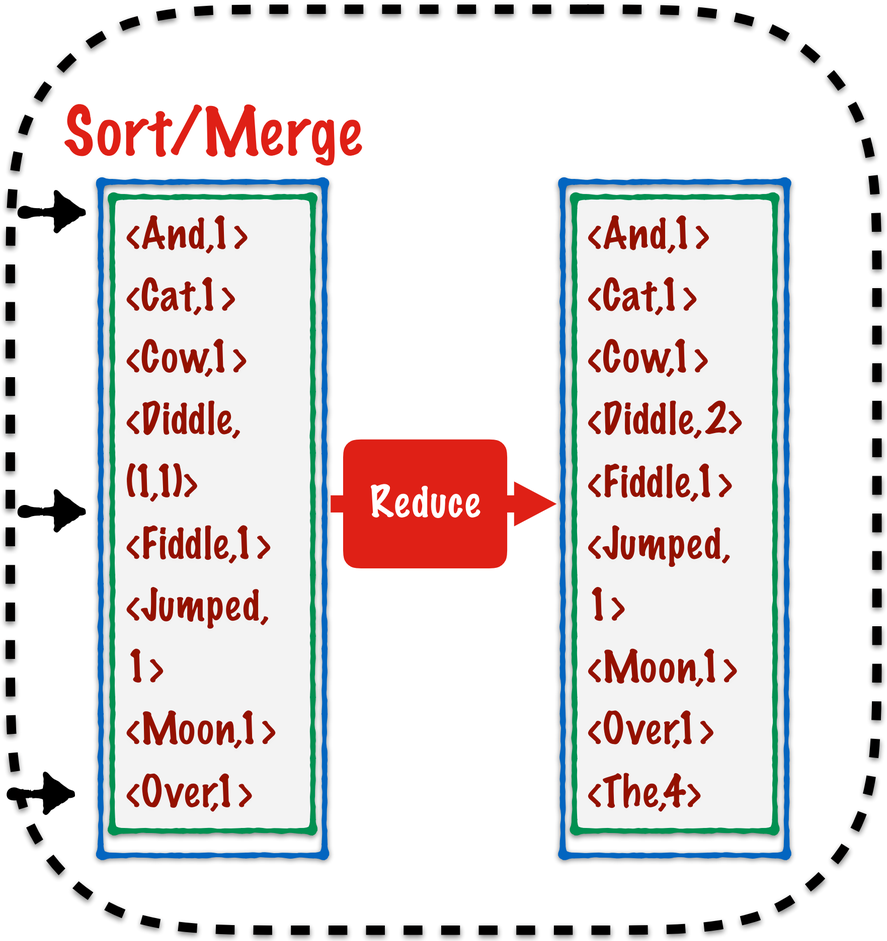


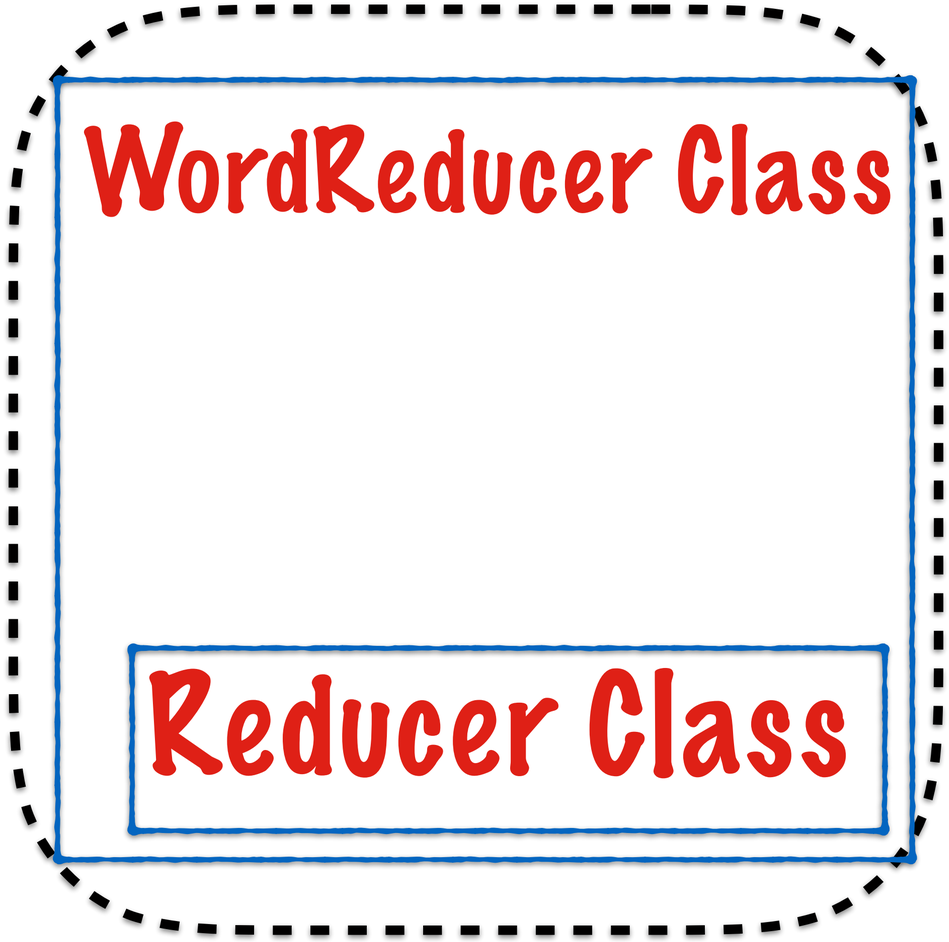
This is a generic class, with 4 type parameters

WordMapper Class

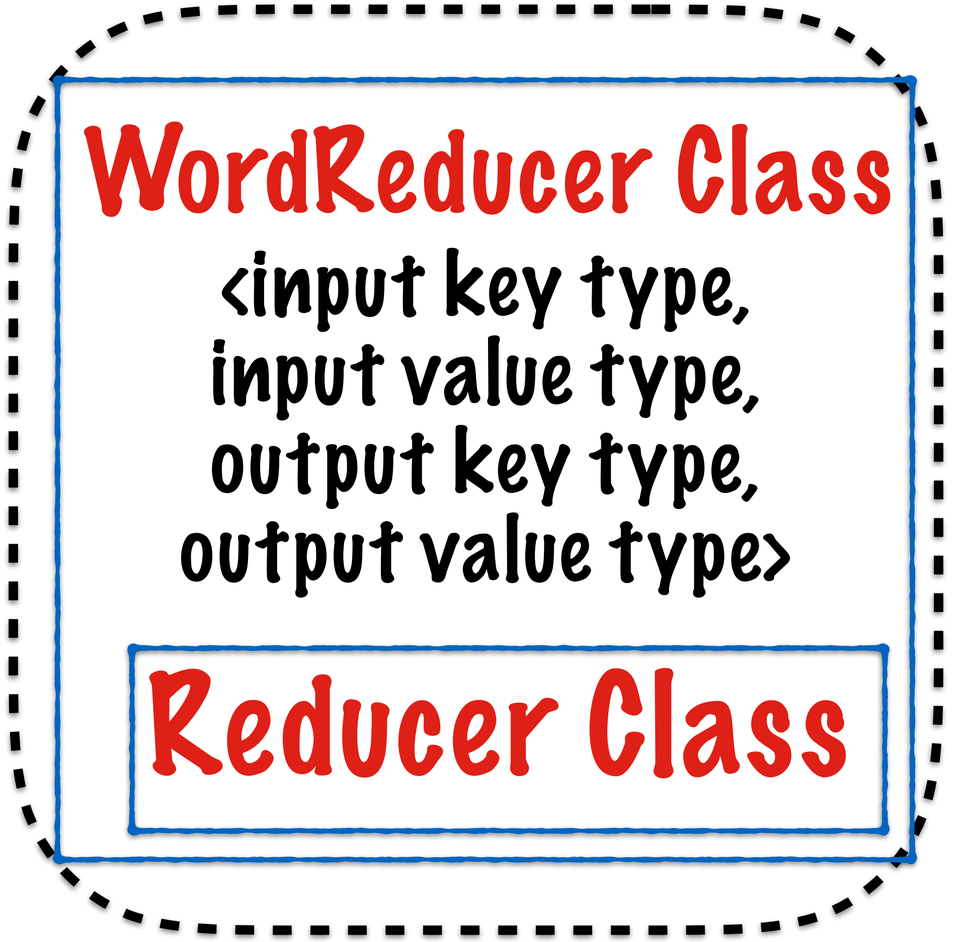
<input key type,
input value type,
output key type,
output value type>

Mapper Class





The reduce() function is implemented in a class that extends the Reducer Class



This is a generic class too,
with 4 type parameters

WordMapper Class

<input key type,
input value type,
output key type,
output value type>

Mapper Class

WordReducer Class

<input key type,
input value type,
output key type,
output value type>

Reducer Class

The output types of the Mapper should match the input types of the Reducer

WordMapper Class

<input key type,
input value type,
output key type,
output value type>

Mapper Class

WordReducer Class

<input key type,
input value type,
output key type,
output value type>

Reducer Class

The output types of the Mapper should match the input types of the Reducer

WordMapper Class

WordReducer Class

These 2 classes are used by a Job that is configured in the Main Class

WordCount Class

Job Object

WordMapper Class

WordReducer Class

The Job has a bunch of properties that need to be configured

WordCount Class

Job Object

Input filepath

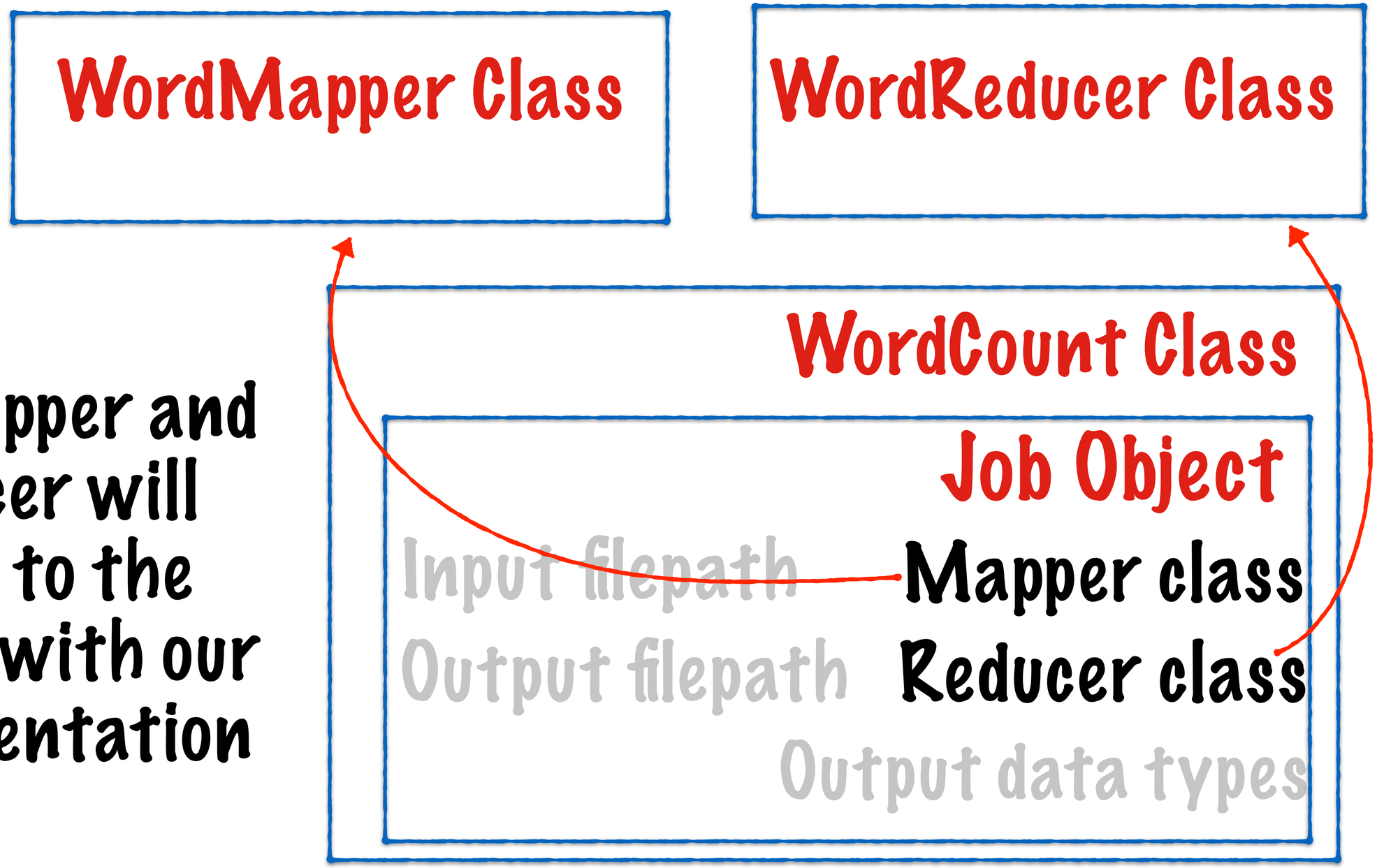
Output filepath

Mapper class

Reducer class

Output data types

The Mapper and Reducer will point to the classes with our implementation



WordMapper Class

WordReducer Class

WordCount Class

Job Object

Input filepath

Output filepath

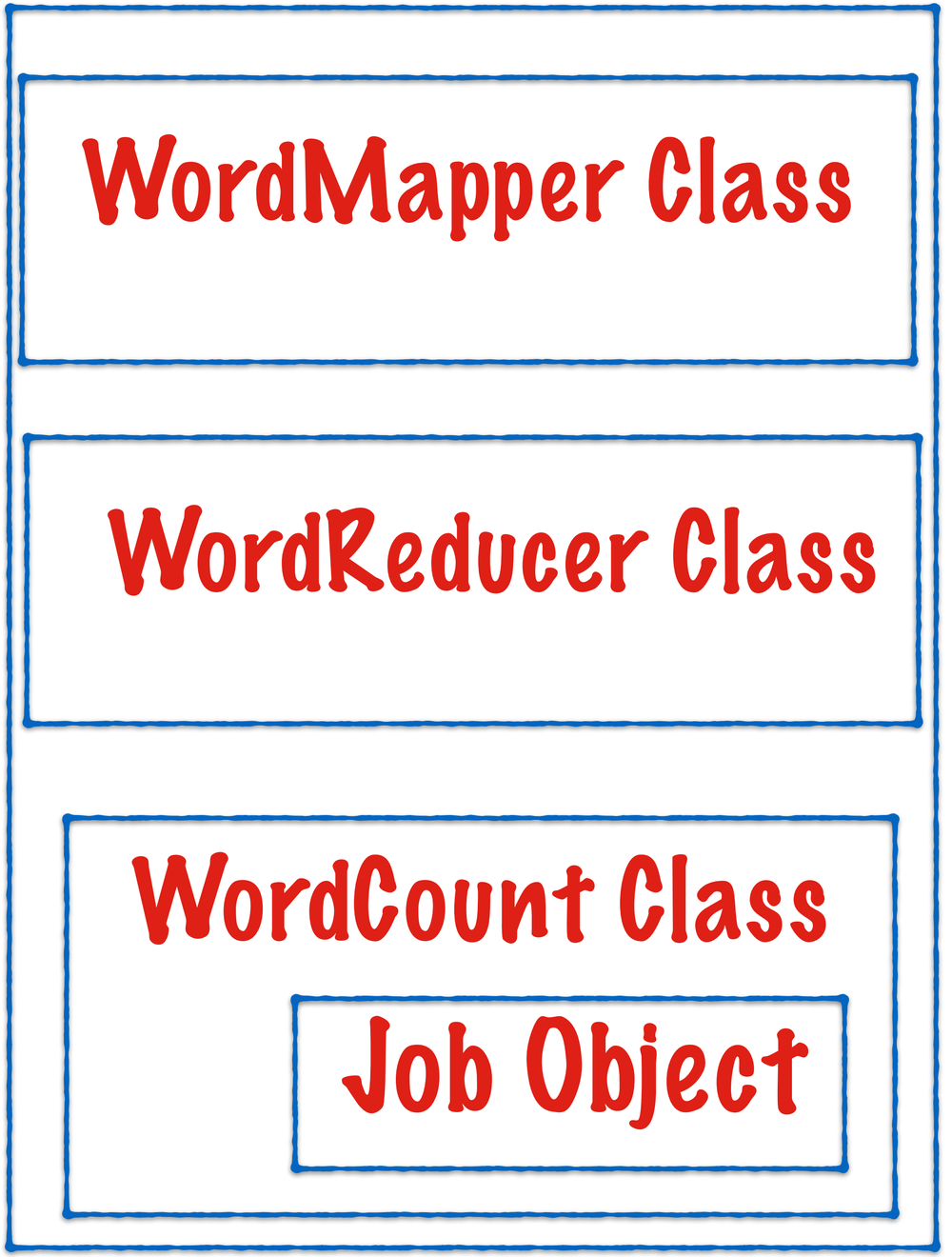
Mapper class

Reducer class

Output data types

The output data types need to match the type parameters of the Mapper and Reducer Classes

All these classes are put into a JAR file which contains all the Hadoop JARS as well



When you run a job, this JAR file gets distributed to all the nodes where the computation is run

WordCount.JAR

WordMapper Class

WordReducer Class

WordCount Class

WordMapper Class

In Java the map() function is represented by the Mapper Class

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

In Java the map() function is represented by the Mapper Class

```
package

import
import
import
import

import

public class
@Override

{

    String line = value.toString();
            context.write(
        }
}
```

This class represents the code for the step

<1,Hey diddle diddle> → map() → <Hey,1> <Diddle,1> <Diddle,1>

<Linenum, Line of text> → <Word, 1>

WordMapper Class

Mapper class is a generic

It has 4 type parameters

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable>
{
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

Input Key Type

Output key type

Input Value Type

Output Value type

WordMapper Class

Hadoop has it's own set of basic types optimized for network serialization

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable>
{
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

These are wrappers around Java primitive types

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

These types are implemented using the Writable Interface

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

Corresponding Hadoop type for Java Long

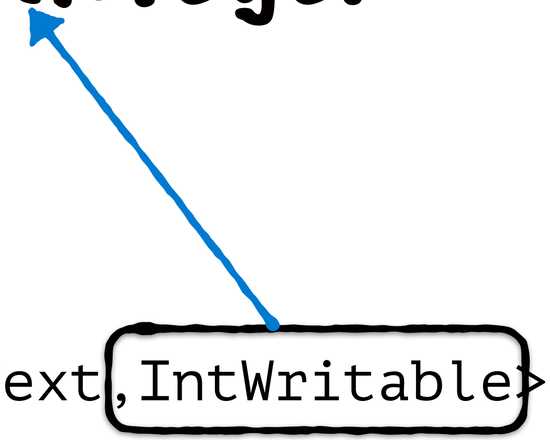
```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

Corresponding Hadoop type for Java String

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

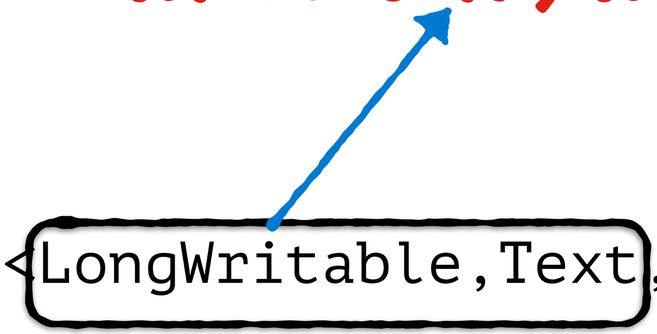
Corresponding Hadoop type for Java Integer



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;


public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

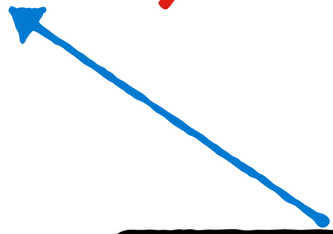
In our example, the input to map has the form <lineNum , line of Text>



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable>
{
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

In our example, the output of map has the form <Word ,1 >



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable>
{
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

The Mapper class has an abstract method called map()

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        String line = value.toString();
        for (String word : line.split(" ")){
            if(word.length()>0){
                context.write(new Text(word),new IntWritable(1));
            }
        }
}

}
```

WordMapper Class

We'll override this method with our own implementation

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

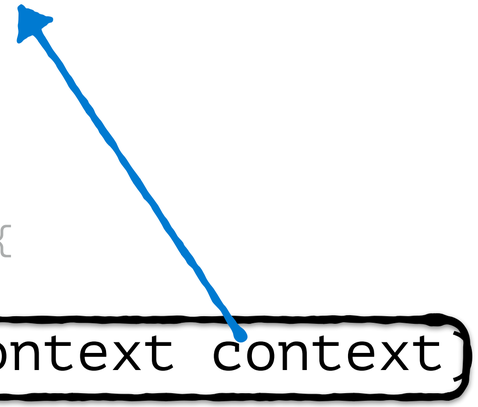
WordMapper Class

The map() method takes a Key and a Value

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        String line = value.toString();
        for (String word : line.split(" ")){
            if(word.length()>0){
                context.write(new Text(word),new IntWritable(1));
            }
        }
}

}
```

WordMapper Class

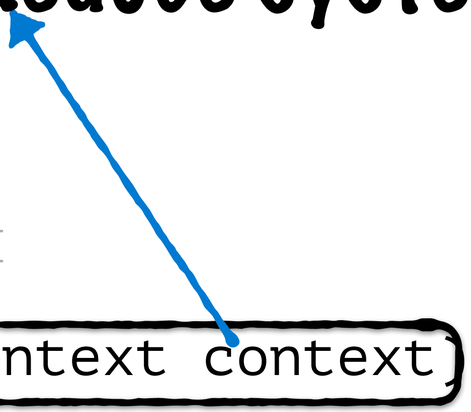
It processes the input and writes the output to a Context object



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
   public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {


    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

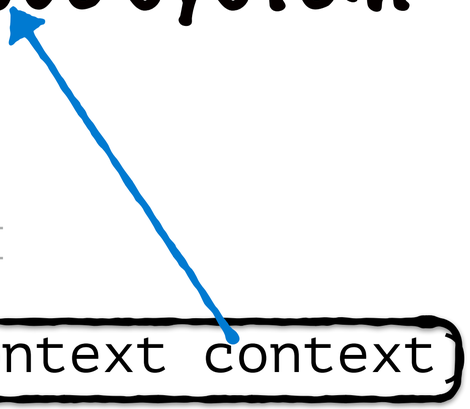
The Context object stores the output and is accessed by the rest of the MapReduce system



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
   public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

It's how the Mapper / Reducer communicate with the rest of the MapReduce system



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
   public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

    String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

Here we've implemented the map() function

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String line = value.toString();
        for (String word : line.split(" ")){
            if(word.length()>0){
                context.write(new Text(word),new IntWritable(1));
            }
        }
    }
}

}
```

WordMapper Class

Convert the Hadoop Text type to Java String

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

      String line = value.toString();
    for (String word : line.split(" ")){
        if(word.length()>0){
            context.write(new Text(word),new IntWritable(1));
        }
    }
}

}
```

WordMapper Class

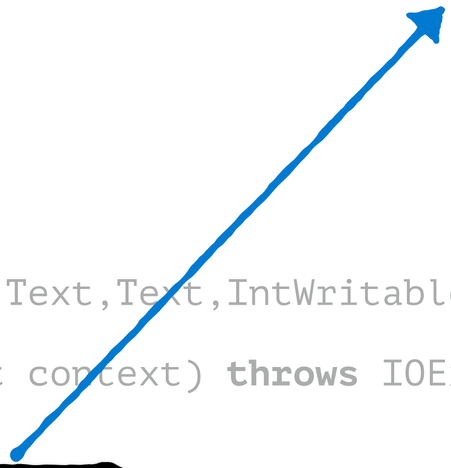
Split the line into words

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

    String line = value.toString();
        for (String word : line.split(" ")){
            if(word.length()>0){
                context.write(new Text(word),new IntWritable(1));
            }
    }
}

}
```

WordMapper Class

For each of the words, we'll write out a key-value pair

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

    String line = value.toString();
        for (String word : line.split(" ")){
            if(word.length()>0){
                context.write(new Text(word),new IntWritable(1));
            }
        }
}

}
```



WordMapper Class

For each of the words, we'll write out a key-value pair

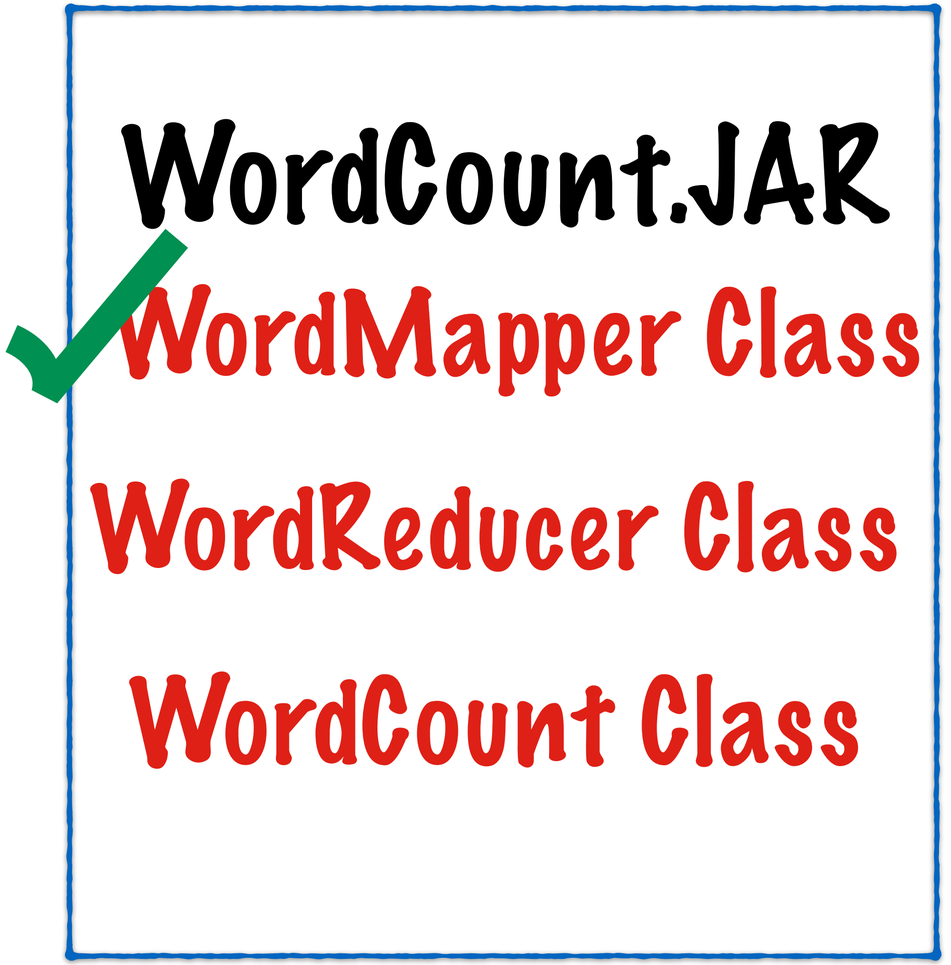
```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

    String line = value.toString();
      for (String word : line.split(" ")){
        if(word.length()>0){
              context.write(new Text(word),new IntWritable(1));
          }
    }
}

}
```

WordMapper Class

The pair (word, 1) is written to the context object

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
@Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

    String line = value.toString();
      for (String word : line.split(" ")){
        if(word.length()>0){
                context.write(new Text(word),new IntWritable(1));
            }
    }
}

}
```

When you run a job, this JAR file gets distributed to all the nodes where the computation is run



WordReducer Class

In Java the reduce() function is represented by the Reducer Class

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

In Java the reduce() function is represented by the Reducer Class

```
package

import
import
import

import

public class


InterruptedException{
        {
        }
}
```

This class represents the code for the step

<word, list>

<word, count>

This is created by taking the map output and merging all the pairs with same key

WordReducer Class

Reducer class is a generic

It has 4 type parameters

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

Input Key Type

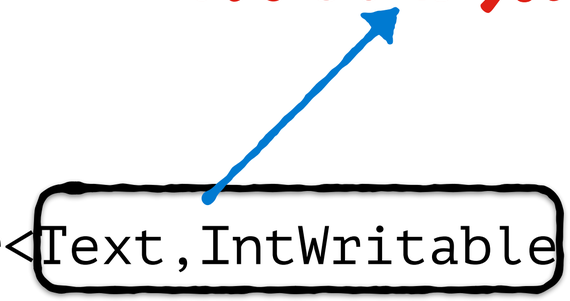
Output key type

Type of each element in the list of Input Values

Output Value type

WordReducer Class

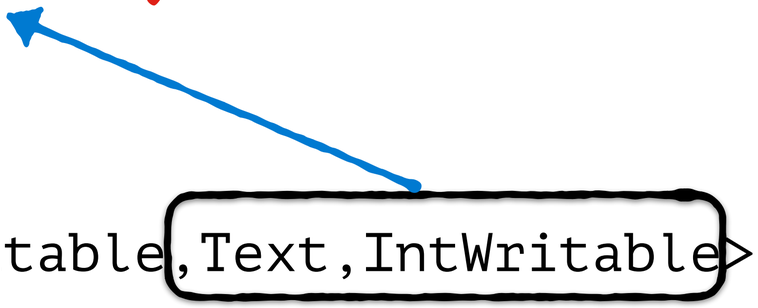
In our example, the input to map has the form <word ,list of 1s>



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

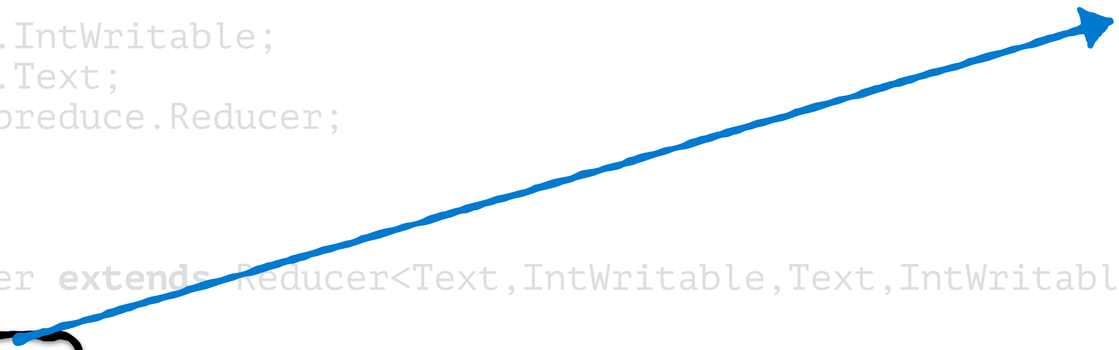
In our example, the output of reduce has the form <word , count>



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

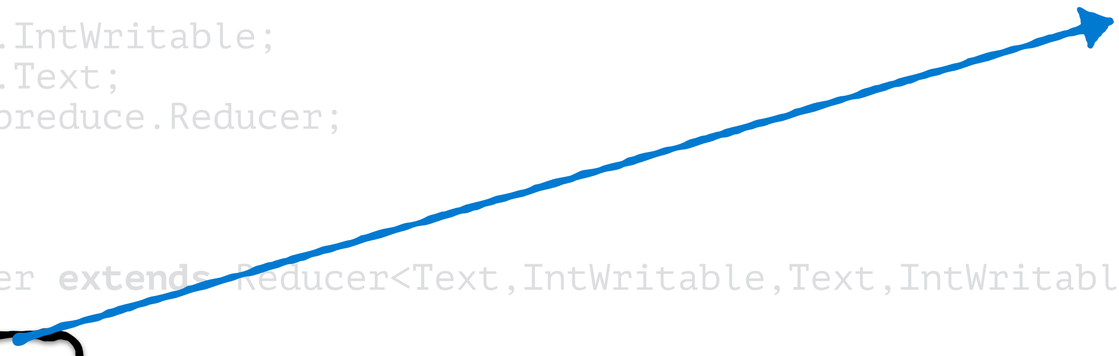
The Reducer class has an abstract method called reduce()



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
     public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

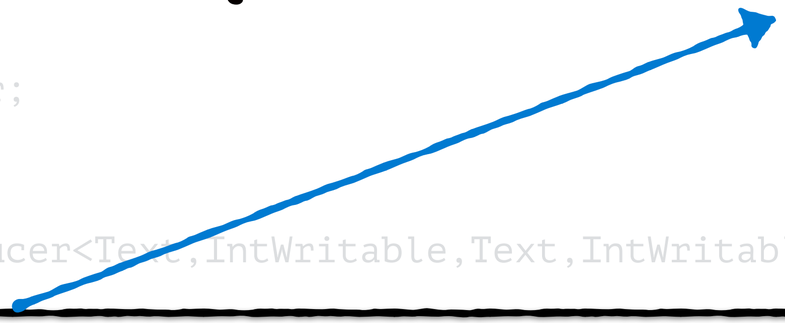
We'll override this method with our own implementation



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
     public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

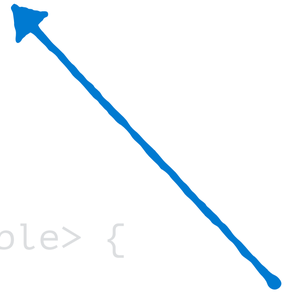
The reduce() method takes a Key and an Iterable of Values



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
     public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

It processes the input and writes the output to a Context object



```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
     public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

Here we've implemented the reduce() function

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

We iterate through the list of values

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

WordReducer Class

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```

And compute the sum of the values

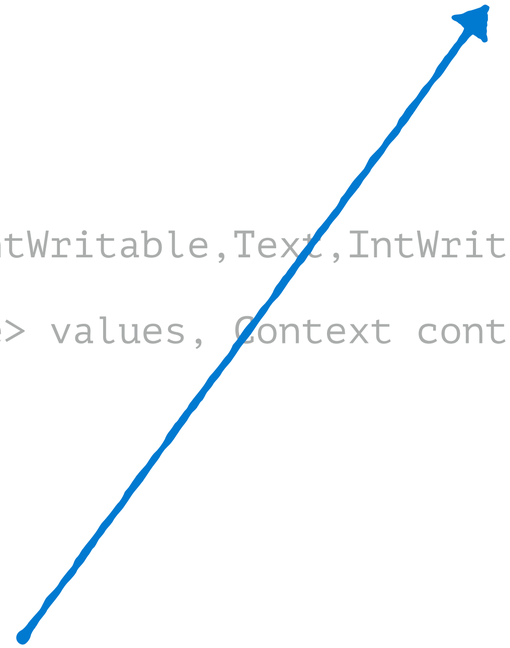
<Diddle,(1,1)>

Sum the values in this list

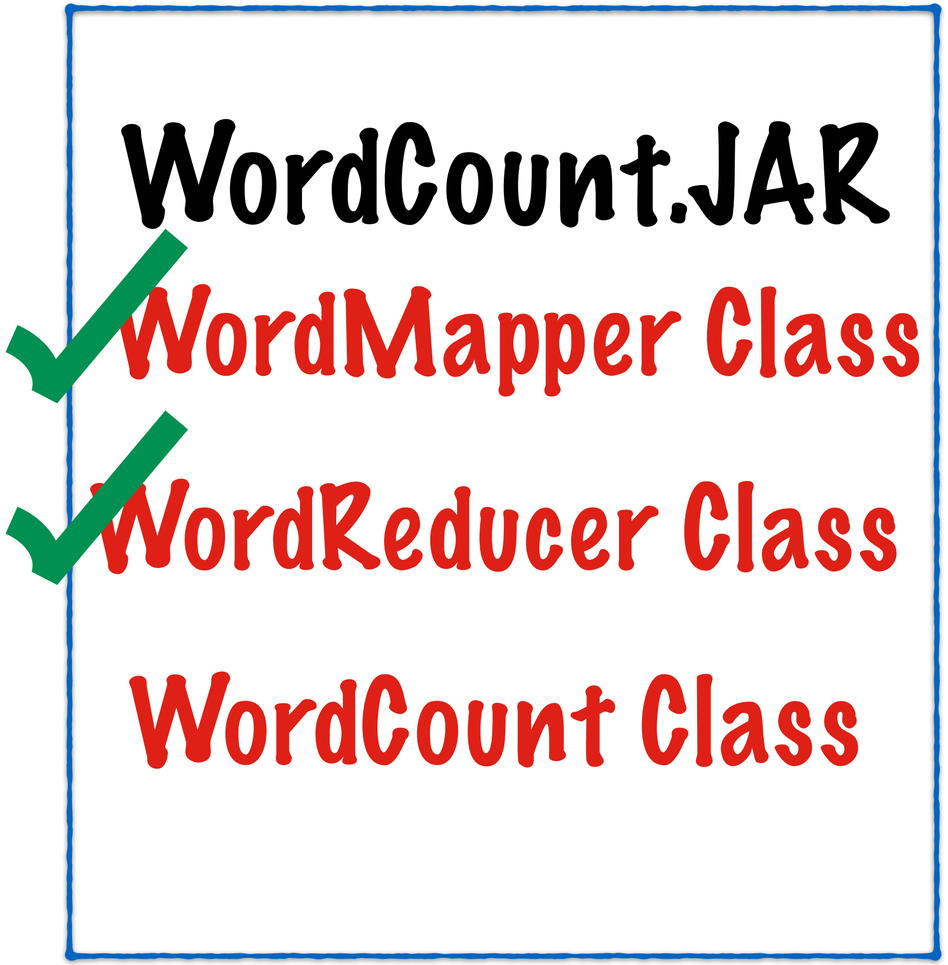
WordReducer Class

In the end, we write out a pair (word, count) to the Context object

```
package com.company;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException{
        int count = 0;
        for (IntWritable value:values)
        {
            count += value.get();
        }

        context.write(key, new IntWritable(count));
    }
}
```



When you run a job, this JAR file gets distributed to all the nodes where the computation is run



WordCount Class

We'll write a driver class whose main() method will point to our Mapper and Reducer

```java
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);



        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

The main() method will take 2 String inputs

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);



        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

2 String inputs

Input text file path,
Output file path

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);



        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

We check whether the right number of inputs are given, and if not print an error

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
                System.err.println("Invalid Command");
                System.err.println("Usage: WordCount <input path> <output path>");
                System.exit(0);
            }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);


        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

A configuration object

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);



        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

By default, this will pick up the configuration of your Hadoop deployment

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);



        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");


        job.setJarByClass(WordCount.class);


        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

We instantiate a new Job whose name is “wordcount”

WordCount Class

This object will hold all the specifications including the Mapper and Reducer to be used

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");


        job.setJarByClass(WordCount.class);


        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

We specify the JAR file that contains our code

WordCount Class

The JAR is specified by the Main class of our code

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

Hadoop will look for the JAR file that contains this class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

It will then distribute this JAR file to all the nodes where our job will run

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

We set the input file path and output file path for the job using the arguments passed to main()

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

The input path can be a file or a directory or a file name pattern

WordCount Class

The output path must be a directory that does not exist

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

We set the Mapper class and Reducer class to the classes we've written

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

Explicitly set the type of the output keys and values of the Reducer Class

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

This has to match the types we specified in our Reducer Class

```
public class WordCountReducer extends
Reducer<Text,IntWritable,Text,IntWritable>
```

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

In our case the Mapper class also has the same output type

```
public class WordCountMapper extends
Mapper<LongWritable,Text,Text,IntWritable>
```

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

Otherwise, the Mapper output types should be set using setMapOutputKeyClass() and setMapOutputValueClass()

```
public class WordCountMapper extends
Mapper<LongWritable,Text,Text,IntWritable>
```

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

The input types are set by default for text files in the way we want, so we don't need to specify that

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);


        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

waitForCompletion() method will submit the Job for execution and wait for it to complete

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);


        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

The method will return a Boolean indicating whether the Job ran successfully

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);


        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

We translate that Boolean to the System exit code

true : 0 (success)

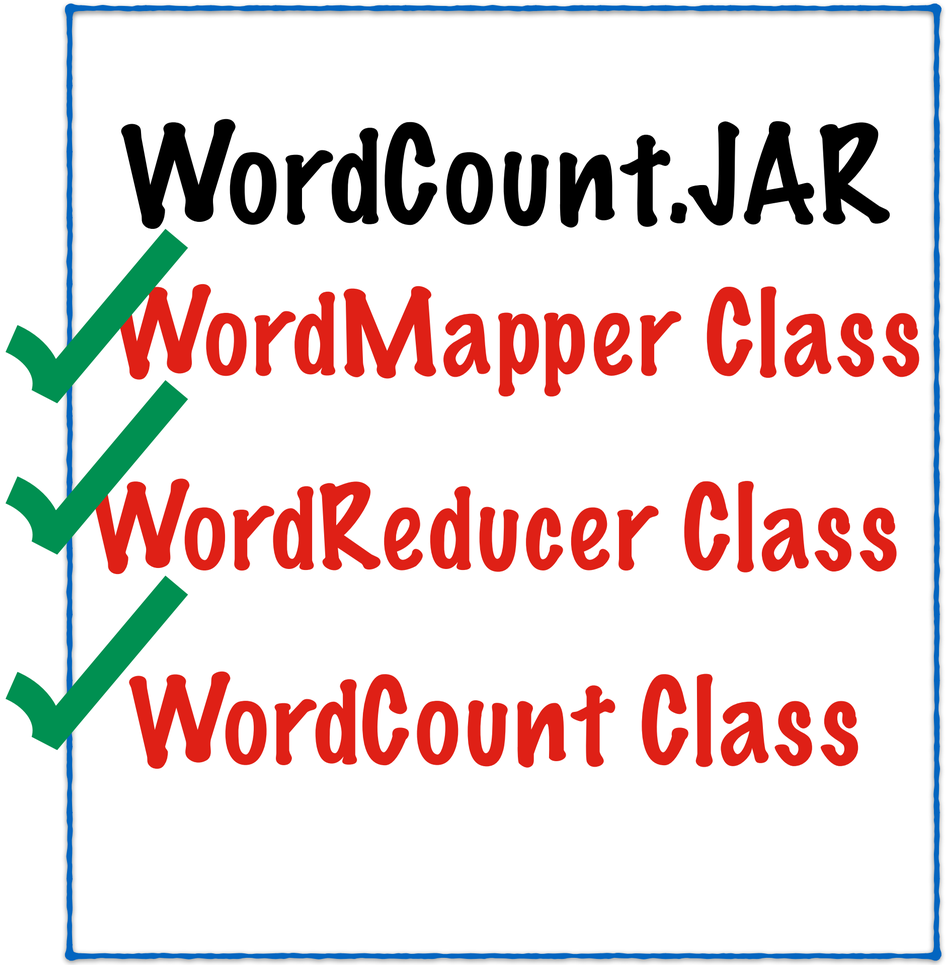
false : 1 (failure)

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);


        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

true here means that the job progress will be printed to screen

When you run a job, this JAR file gets distributed to all the nodes where the computation is run



Running a MapReduce job

Step 1: Build a JAR file that contains all the code we just wrote

Step 1a: In the JAR, include all the Hadoop JARS as dependencies

Step 2: Run the job at the command line

Step 2: Run the job at the command line

```
$ hadoop jar <JARpath> <mainclass> <inputFile> <outputDir>
```

The path to the JAR file

Step 2: Run the job at the command line

```
$ hadoop jar <JARpath> <mainclass> <inputFile> <outputDir>
```

The complete path (including package name) of the Main Class

Ex: com.company.WordCount

Step 2: Run the job at the command line

```
$ hadoop jar <JARpath> <mainclass> <inputFile> <outputDir>
```

The input file or a directory containing multiple input files

Step 2: Run the job at the command line

```
$ hadoop jar <JARpath> <mainclass> <inputFile> <outputDir>
```

A directory that does not exist