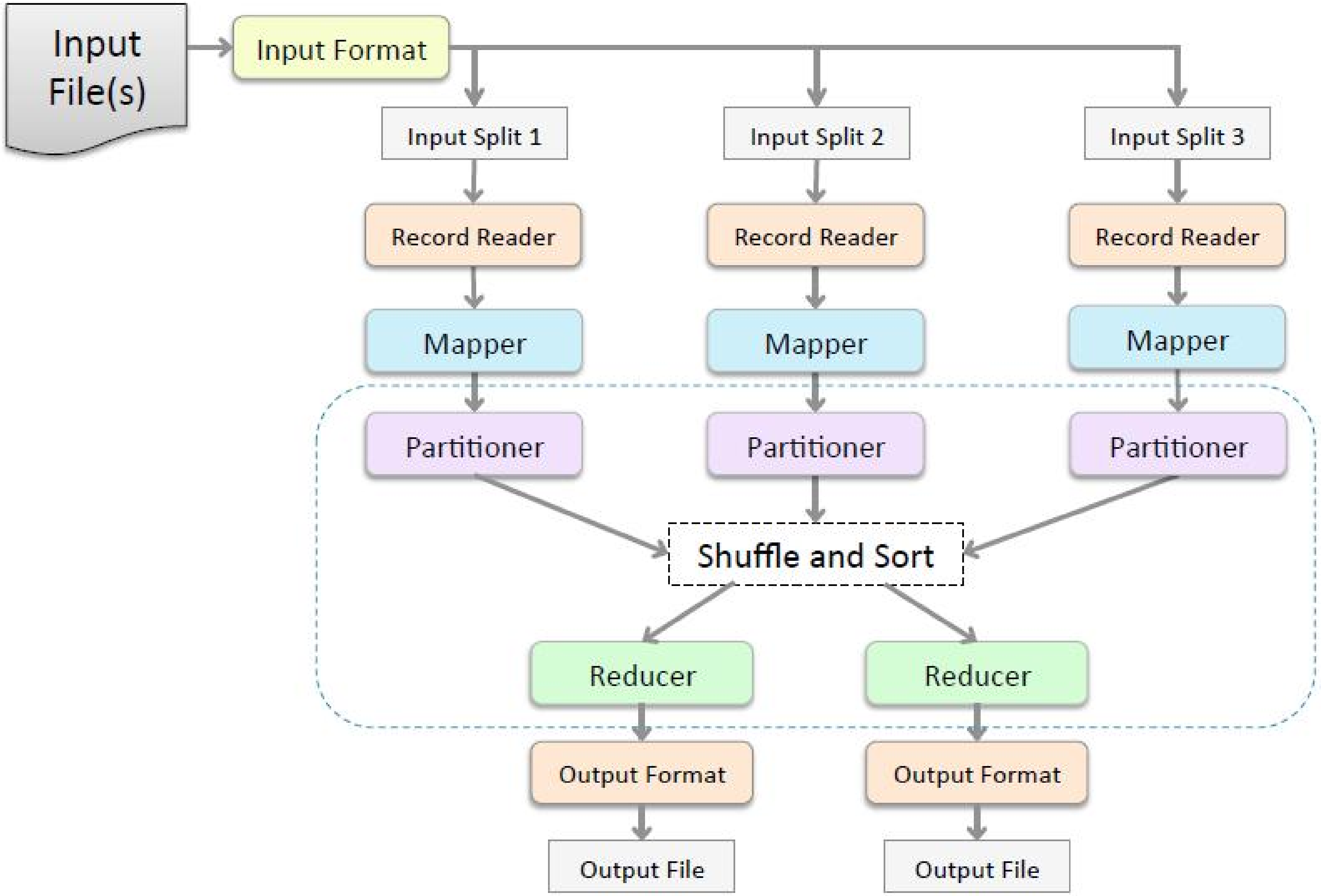
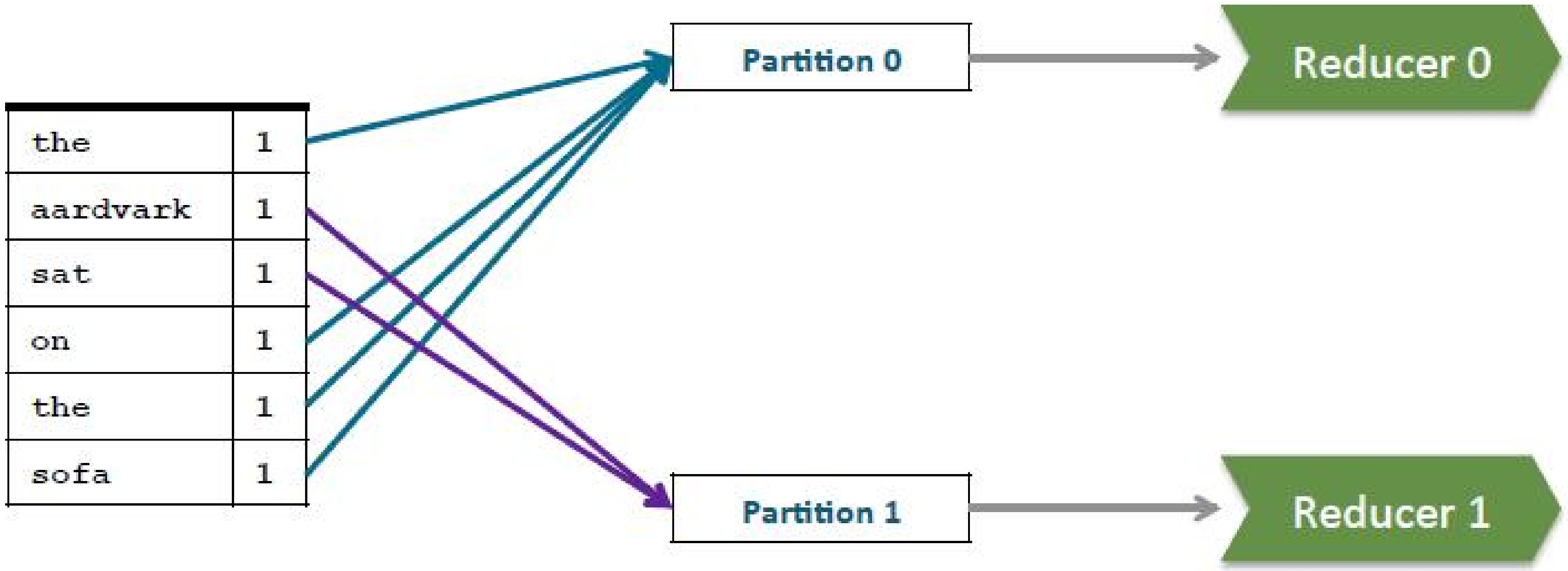

Input File(s)
Input Format
Input Split 1
Input Split 2
Input Split 3
Record Reader
Record Reader
Record Reader
Mapper
Mapper
Mapper
Partitioner
Partitioner
Partitioner
Shuffle and Sort
Reducer
Reducer
Output Format
Output Format
Output File
Output File

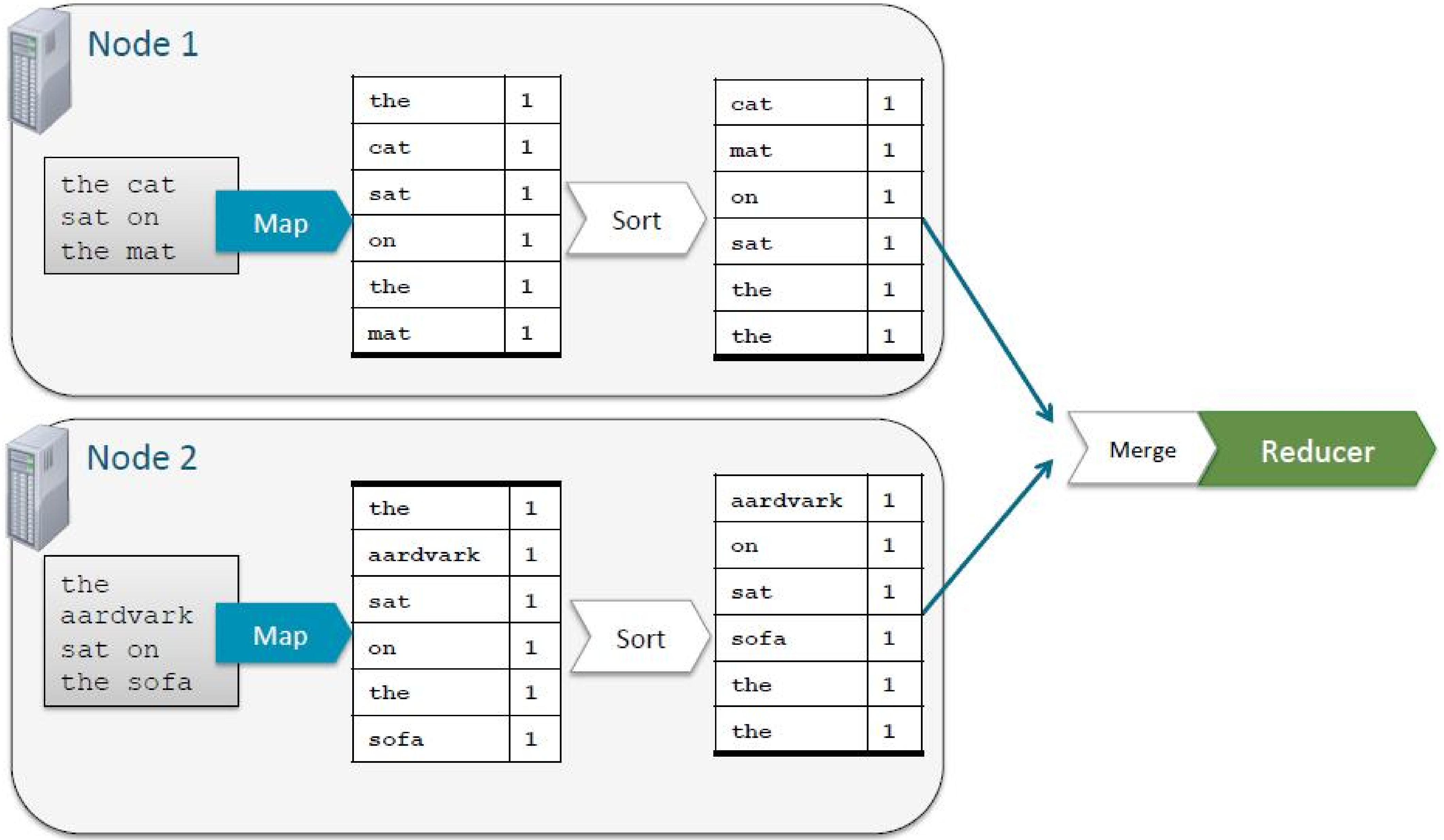
What Does the Partitioner Do?

- **The Partitioner determines which Reducer each intermediate key and its associated values goes to**

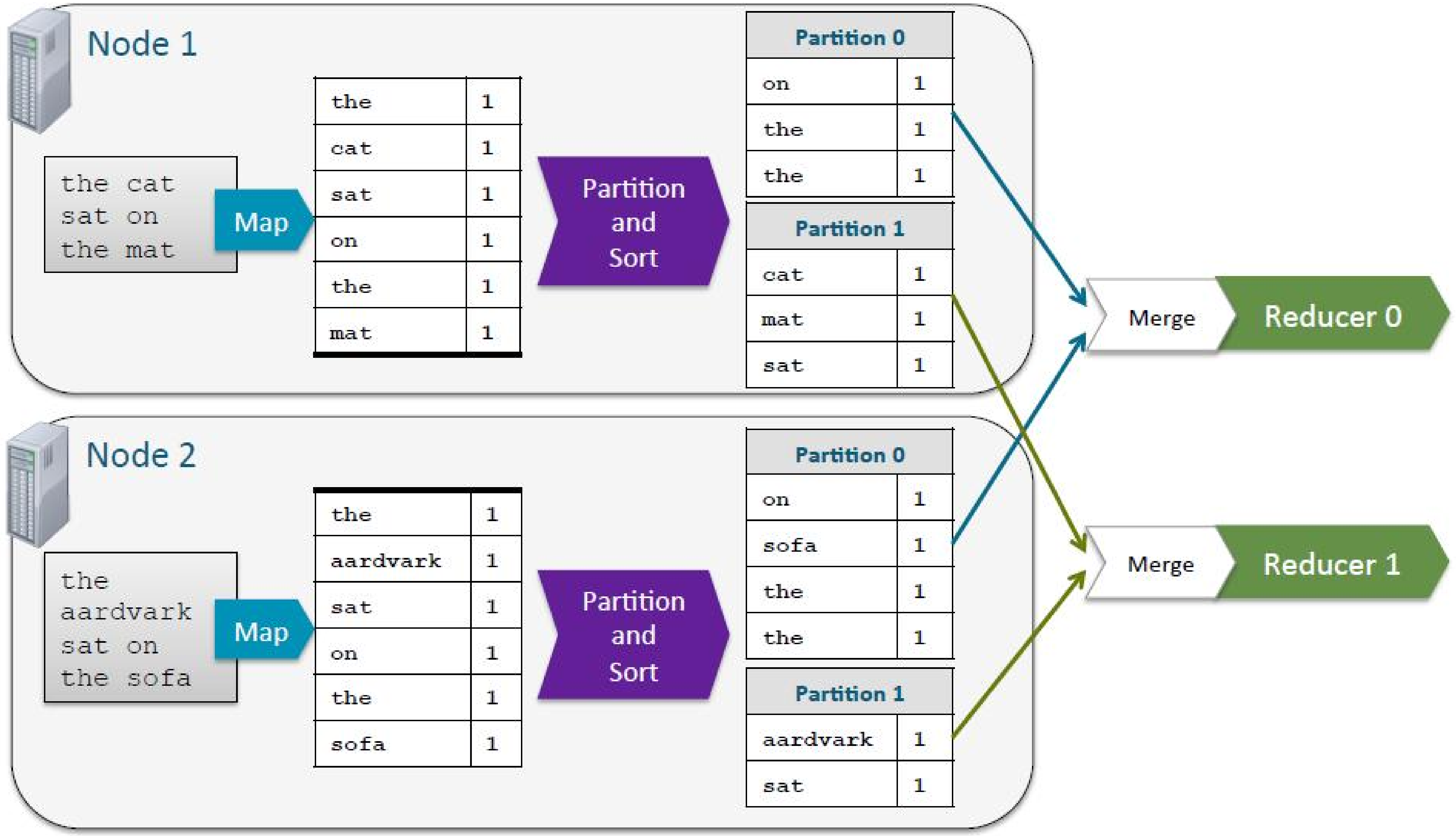
```
getPartion:
    (inter_key, inter_value, num_reducers)→ partition
```



Example: WordCount with One Reducer



Example: WordCount with Two Reducers



The Default Partitioner

- **The default Partitioner is the `HashPartitioner`**
  - Uses the Java `hashCode` method
  - Guarantees all pairs with the same key go to the same Reducer

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {

    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() % numReduceTasks);
    }
}
```

How Many Reducers Do You Need?

- **An important consideration when creating your job is to determine the number of Reducers specified**
- **Default is a single Reducer**
- **With a single Reducer, one task receives *all* keys in sorted order**
  - This is sometimes advantageous if the output must be in completely sorted order
  - Can cause significant problems if there is a large amount of intermediate data
    - Node on which the Reducer is running may not have enough disk space to hold all intermediate data
    - The Reducer will take a long time to run

Jobs Which Require a Single Reducer

- **If a job needs to output a file where all keys are listed in sorted order, a single Reducer must be used**
- **Alternatively, the `TotalOrderPartitioner` can be used**
  - Uses an externally generated file which contains information about intermediate key distribution
  - Partitions data such that all keys which go to the first Reducer are smaller than any which go to the second, etc
  - In this way, multiple Reducers can be used
  - Concatenating the Reducers' output files results in a totally ordered list

Jobs Which Require a Fixed Number of Reducers

- **Some jobs will require a specific number of Reducers**
- **Example: a job must output one file per day of the week**
  - Key will be the weekday
  - Seven Reducers will be specified
  - A Partitioner will be written which sends one key to each Reducer

Jobs With a Variable Number of Reducers (1)

- **Many jobs can be run with a variable number of Reducers**
- **Developer must decide how many to specify**
  - Each Reducer should get a reasonable amount of intermediate data, but not too much
- **Typical way to determine how many Reducers to specify:**
  - Test the job with a relatively small test data set
  - Extrapolate to calculate the amount of intermediate data expected from the ‘real’ input data
  - Use that to calculate the number of Reducers which should be specified

Custom Partitioners

- **Sometimes you will need to write your own Partitioner**
- **Example: your key is a custom `WritableComparable` which contains a pair of values `(a, b)`**
  - You may decide that all keys with the same value for `a` need to go to the same Reducer
  - The default Partitioner is not sufficient in this case

Custom Partitioners

- **Custom Partitioners are needed when performing a secondary sort**
- **Custom Partitioners are also useful to avoid potential performance issues**
  - To avoid one Reducer having to deal with many very large lists of values
  - Example: in our word count job, we wouldn't want a single Reducer dealing with all the three- and four-letter words, while another only had to handle 10- and 11-letter words

Creating a Custom Partitioner

1. **Create a class that extends Partitioner**
2. **Override the `getPartition` method**
   - Return an int between 0 and one less than the number of Reducers
     - e.g., if there are 10 Reducers, return an int between 0 and 9

```
import org.apache.hadoop.mapreduce.Partitioner;

public class MyPartitioner<K,V> extends Partitioner<K,V> {

    @Override
    public int getPartition(K key, V value, int numReduceTasks) {
        //determine reducer number between 0 and numReduceTasks-1
        //...
        return reducer;
    }
}
```

Using a Custom Partitioner

- **Specify the custom Partitioner in your driver code**

```
job.setPartitionerClass(MyPartitioner.class);
```

Aside: Setting up Variables for your Partitioner

- **If you need to set up variables for use in your partitioner, it should implement `Configurable`**
- **If a Hadoop object implements `Configurable`, its `setConf()` method will be called once, when it is instantiated**
- **You can therefore set up variables in the `setConf()` method which your `getPartition()` method will then be able to access**

Aside: Setting up Variables for your Partitioner

```
class MyPartitioner extends Partitioner<K, V> implements Configurable {

    private Configuration configuration;
    // Define your own variables here

    @Override
    public void setConf(Configuration configuration) {
        this.configuration = configuration;
        // Set up your variables here
    }
    @Override
    public Configuration getConf() {
        return configuration;
    }
    public int getPartition(K key, V value, int numReduceTasks) {
        // Use variables here
    }
}
```