ACCUMULATORS

Say you were using Spark to process some logs

```
2016-06-08 12:51:29,517 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: Stopping infoServer
2016-06-08 12:51:29,521 INFO  [regionserver//192.168.0.118:16201] mortbay.log: Stopped SelectChannelConnector@0.0.0.0:1
2016-06-08 12:51:29,522 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: aborting server 192.168.0
2016-06-08 12:51:29,522 INFO  [regionserver//192.168.0.118:16201] client.ConnectionManager$HConnectionImplementation: C
2016-06-08 12:51:29,918 INFO  [regionserver//192.168.0.118:16201-SendThread(localhost:2181)] zookeeper.ClientCnxn: Open
(unknown error)
2016-06-08 12:51:30,021 INFO  [regionserver//192.168.0.118:16201] zookeeper.ZooKeeper: Session: 0x0 closed
2016-06-08 12:51:30,022 INFO  [regionserver//192.168.0.118:16201-EventThread] zookeeper.ClientCnxn: EventThread shut do
2016-06-08 12:51:30,024 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: stopping server 192.168.0
2016-06-08 12:51:30,024 INFO  [regionserver//192.168.0.118:16201] hbase.ChoreService: Chore service for: 192.168.0.118,
2016-06-08 12:51:30,026 INFO  [regionserver//192.168.0.118:16201] ipc.RpcServer: Stopping server on 16201
2016-06-08 12:51:30,375 INFO  [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Opening socket connection to serv
2016-06-08 12:51:30,375 WARN  [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Session 0x0 for server null, unex
```

The logs are stored in a text file

```
2016-06-08 12:51:29,517 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: Stopping infoServer
2016-06-08 12:51:29,521 INFO  [regionserver//192.168.0.118:16201] mortbay.log: Stopped SelectChannelConnector@0.0.0.0:1
2016-06-08 12:51:29,522 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: aborting server 192.168.0
2016-06-08 12:51:29,522 INFO  [regionserver//192.168.0.118:16201] client.ConnectionManager$HConnectionImplementation: C
2016-06-08 12:51:29,918 INFO  [regionserver//192.168.0.118:16201-SendThread(localhost:2181)] zookeeper.ClientCnxn: Open
(unknown error)
2016-06-08 12:51:30,021 INFO  [regionserver//192.168.0.118:16201] zookeeper.ZooKeeper: Session: 0x0 closed
2016-06-08 12:51:30,022 INFO  [regionserver//192.168.0.118:16201-EventThread] zookeeper.ClientCnxn: EventThread shut do
2016-06-08 12:51:30,024 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: stopping server 192.168.0
2016-06-08 12:51:30,024 INFO  [regionserver//192.168.0.118:16201] hbase.ChoreService: Chore service for: 192.168.0.118,
2016-06-08 12:51:30,026 INFO  [regionserver//192.168.0.118:16201] ipc.RpcServer: Stopping server on 16201
2016-06-08 12:51:30,375 INFO  [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Opening socket connection to serv
2016-06-08 12:51:30,375 WARN  [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Session 0x0 for server null, unex
```

You have a specific set of processing steps for this log

1. Parse the logs

2. Save the parsed logs into a new text file

1. Parse the logs

2. Save the parsed logs into a new text file

At the end of this, you also want to print the number of ERROR messages in the log

1. Parse the logs

2. Save the parsed logs into a new text file

3. Print the number of error messages

Option 1:

1 Action to parse the logs and save them to text file

Option 1:

1 Action to parse the logs and save them to text file

1. Parse the logs
2. Save the parsed logs into a new text file
3. Print the number of error messages

A 2nd Action to compute the count of Error messages

Option 1:

1 Action to parse the logs and save them to text file

A 2nd Action to compute the count of Error messages

Option 2:

Use an accumulator variable

Accumulator variable

An Accumulator is a special type of variable

It is shared among all the nodes of the Spark cluster

Accumulator variable

Broadcast variables are also shared variables

Broadcast variables are immutable though

Accumulators are not!

Accumulator variable

Accumulators have 2 important characteristics

1. Individual nodes can only write to the accumulator

2. The main program (or Spark shell) can only read it's value

1. Individual nodes can only write to the accumulator

While processing an RDD, the nodes can increment the accumulator

1. Individual nodes can only write to the accumulator

The increment is usually triggered by certain events that the user wants to track

Ex: Encountering an ERROR message

Accumulator variable

Accumulators have 2 important characteristics

1. Individual nodes can only write to the accumulator

The individual nodes cannot read the accumulator's value

2. The main program (or Spark shell) can only read it's value

The main program is the one where we invoke the processing of an RDD

usually through an Action

2. The main program (or Spark shell) can only read it's value

This is called the driver program

and the processing task a Job

2. The main program (or Spark shell) can only read it's value

At the end of a Job

the driver program can request the value of the accumulator

2. The main program (or Spark shell) can only read it's value

The driver program can only

1. Create the accumulator ;
2. Read it's value

2. The main program (or Spark shell) can only read it's value

Accumulator variable

The driver program cannot change the accumulator once created

Accumulators have 2 important characteristics

1. Individual nodes can only write to the accumulator

2. The main program (or Spark shell) can only read it's value

Accumulator variable

Let's go back to our log example

```
val logs=sc.textFile(logsPath)
```

This is the logs RDD

Accumulator variable

```
def processLog(line: String): String={
    // some line processing
    //
    //



    line
}
```

This is the function to parse the logs

Accumulator variable

```scala
def processLog(line: String): String={
    // some line processing
    //
    //


    line
}
```

It does a bunch of processing

The specifics don't really matter

Accumulator variable

```
def processLog(line: String): String={
    // some line processing
    //
    //


    line
}
```

We'll use this function to process the logs and save them to a file

```
logs.map(processLog).saveAsTextFile(processedLogsPath)
```

This triggers a Job on the Spark cluster

Accumulator variable

```scala
def processLog(line: String): String={
    // some line processing
    //
    //


    line
}
```

To print a count of ERROR messages in the log, we could trigger another Job

```scala
logs.map(processLog).saveAsTextFile(processedLogsPath)
```

Or we could use an accumulator instead

Accumulator variable

```
def processLog(line: String): String={
    // some line processing
    //
    //


    line
}
```

Let's create an accumulator variable

```
val errCount=sc.accumulator(0)
```

The initial value of the accumulator

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
def processLog(line: String): String={
    // some line processing
    //
    //


    line
}
```

This variable will be shared and incremented by all the nodes

```
val errCount=sc.accumulator(0)
```

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
def processLog(line: String): String={
    // some line processing
    //
    //


    line
}
```

The increment logic has to be part of the function that's used to process the RDD

```
val errCount=sc.accumulator(0)
```

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
val errCount=sc.accumulator(0)
```

```
def processLog(line: String): String={
    // some line processing
    //
    //
  if (line.contains("ERROR")) {
      errCount+=1}
    line
}
```

Whenever an ERROR message is encountered, the accumulator is incremented

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
val errCount=sc.accumulator(0)
```

```
def processLog(line: String): String={
    // some line processing
    //
    //
  if (line.contains("ERROR")) {
      errCount+=1}
    line
}
```

+=

By default, this is the only operation allowed on accumulators

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
val errCount=sc.accumulator(0)
```

```
def processLog(line: String): String={
    // some line processing
    //
    //
  if (line.contains("ERROR")) {
      errCount+=1}
    line
}
```

Users can define custom accumulators with other operations

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
val errCount=sc.accumulator(0)

def processLog(line: String): String={
    // some line processing
    //
    //

    line
}
```

After the Job is completed, you can access the value of the accumulator

```
logs.map(processLog).saveAsTextFile(processedLogsPath)
```

Accumulator variable

```
val errCount=sc.accumulator(0)
```

```
def processLog(line: String): String={
    // some line processing
    //
    //

    line
}
```

After the Job is completed, you can access the value of the accumulator

```
logs.map(processLog).saveAsTextFile(processedLogsPath)
```

```
errCount.value
```

15

SPARK-SUBMIT

So far, we have only used the
Spark REPL environment

This is great for experimentation
and fast feedback

Once you've developed a program/
application to do specific tasks

You can submit this program
to be run on the Spark cluster

To run a Scala program, you'll need to **build and submit JAR files** with a **main** function

Spark has a tool for running the program

spark-submit

spark-submit

```
>spark-submit --class Accumulator ./target/scala-2.11/scalasbt_2.11-1.0.jar
```

This will run the main method from the Accumulator Class

spark-submit

```
>spark-submit --class Accumulator ./target/scala-2.11/scalasbt_2.11-1.0.jar
```

This is the JAR file
built from your
source code

Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

    val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
    val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/
processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

This is some code to process log files

Accumulator.scala

```scala
object Accumulator {

  def main (args: Array[String]){

    val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
    val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
      val logs=sc.textFile(logsPath)
      val errCount=sc.accumulator(0)
      def processLog(line: String): String = {

        if (line.contains("ERROR")){
          errCount += 1
        }
        line
      }
      logs.map(processLog).saveAsTextFile("hdfs:///user/
swethakolalapudi/log/processedLogs2.log")
      println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

This part is exactly the same as what you write in the Spark Shell

Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

This is a bit of setup that you need for your program to run on Spark

Accumulator.scala

```scala
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

Here we are setting up the SparkContext

Accumulator.scala

```scala
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

The SparkContext represents a connection to the Spark cluster

Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

Any Spark application has to start with setting up a SparkContext

Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

When we use Spark shell, the SparkContext is set up for us

Accumulator.scala

```scala
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)


    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

In a script/program, we have to set it up ourselves

Accumulator.scala

```scala
object Accumulator {

  def main (args: Array[String]){
val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

The SparkConf helps us specify the parameters of the Spark Connection

Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }
}
```

setMaster is similar to the master option when we initialize the shell

```
> spark-shell --master yarn-client
```

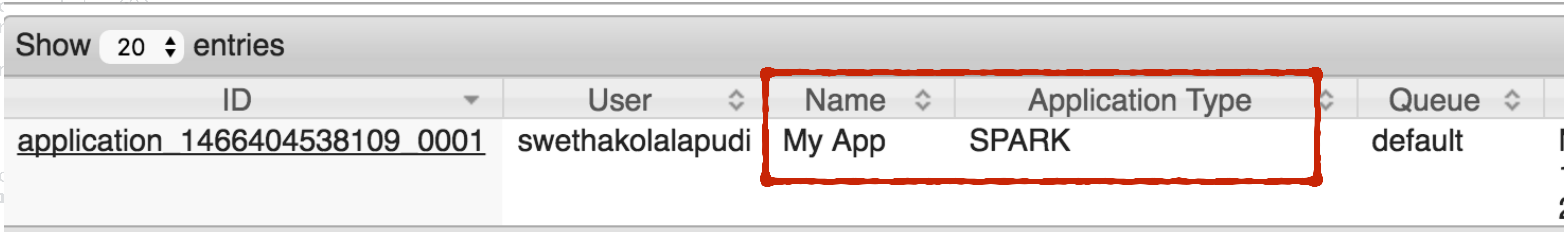
Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String ={

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

This will be the name of our application

Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.ac
    def processLog(li

      if (line.contai
        errCount += 1
      }
      line
    }
    logs.map(processLo
    println("There we
  }
}
```



This is how this application will appear on the YARN web interface

Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString +" ERROR lines")

  }

}
```

This initializes the SparkContext

Accumulator.scala

```
object Accumulator {

  def main (args: Array[String]){

val conf= new SparkConf().setMaster("yarn-client").setAppName("My App")
val sc= new SparkContext(conf)

    val logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
    val logs=sc.textFile(logsPath)
    val errCount=sc.accumulator(0)
    def processLog(line: String): String = {

      if (line.contains("ERROR")){
        errCount += 1
      }
      line
    }
    logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs2.log")
    println("There were " + errCount.value.toString + " ERROR lines")

  }
}
```

Now you can write any code, build class hierarchies etc

spark-submit

```
>spark-submit --class Accumulator ./target/scala-2.11/scalasbt_2.11-1.0.jar
```

The JAR that you submit to Spark has to be built using sbt

To run your Scala code using spark-submit

1. Build a JAR using SBT

SBT is necessary as it adds the dependencies required for Spark

2. Submit your JAR to spark-submit

1. Build a JAR using SBT

You can use your IDE to set up an SBT project and build the JAR

1. Build a JAR using SBT

SBT projects have a config file to specify dependencies

build.sbt

```
name := "ScalaSBT"

version := "1.0"

scalaVersion := "2.11.8"

//additional libraries
libraryDependencies ++= Seq(
"org.apache.spark" %% "spark-core" %  "1.6.1" % "provided"
)
```

1. Build a JAR using SBT

Add the below line to the build.sbt file

```
name := "ScalaSBT"
version := "1.0"

scalaVersion := "2.11.8"
```

```
//additional libraries
libraryDependencies ++= Seq(
"org.apache.spark" %% "spark-core" %  "1.6.1" % "provided"
)
```

To run your Scala code using spark-submit

1. Build a JAR using SBT

SBT is necessary as it adds the dependencies required for Spark

2. Submit your JAR to spark-submit

2. Submit your JAR to spark-submit

To submit your JAR and
run your program

```
>spark-submit --class Accumulator ./target/scala-2.11/scalasbt_2.11-1.0.jar
```

The path to your JAR file

To run your Scala code using spark-submit

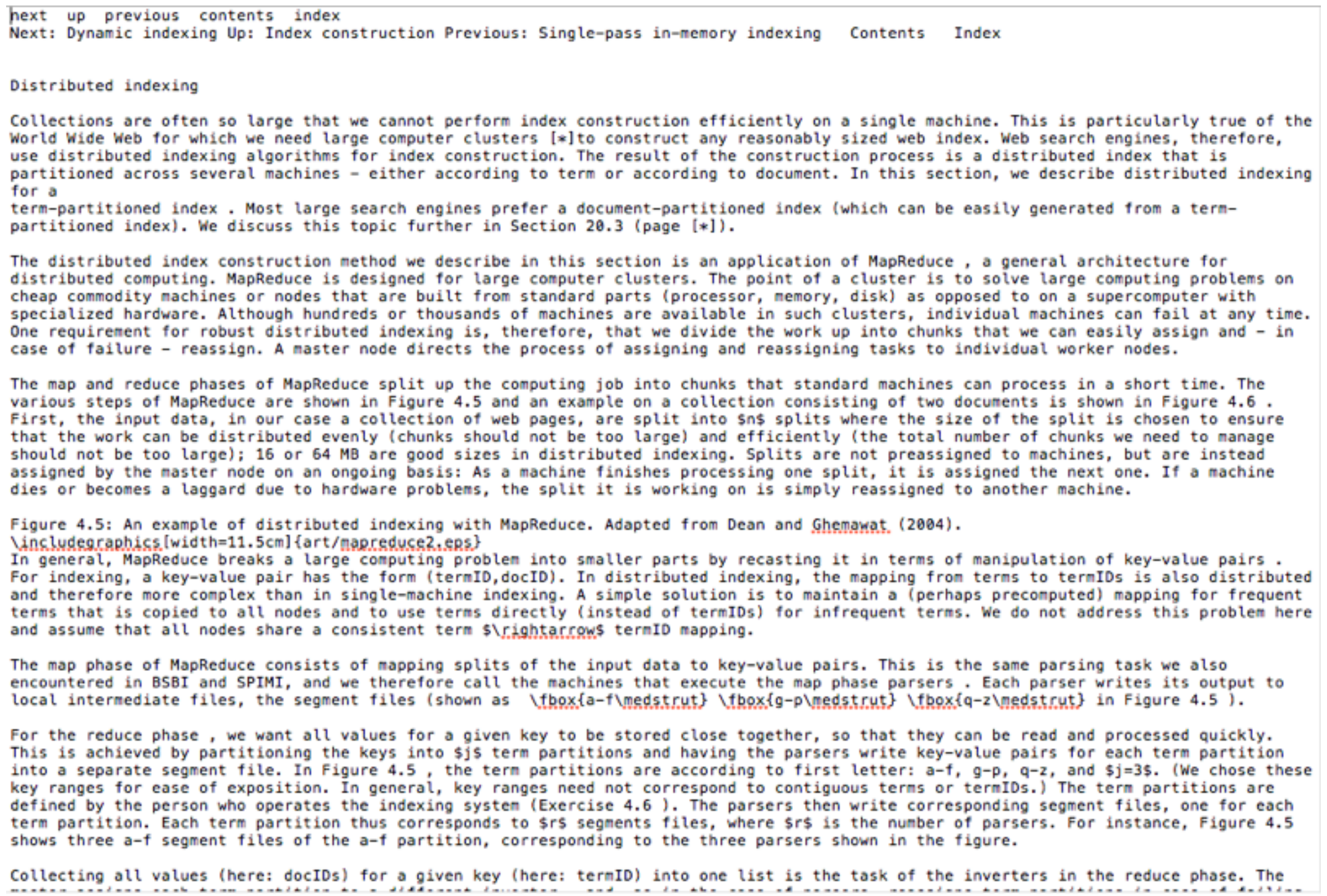
1. Build a JAR using SBT ✓

SBT is necessary as it adds the dependencies required for Spark

2. Submit your JAR to spark-submit ✓

MAPREDUCE WITH SPARK

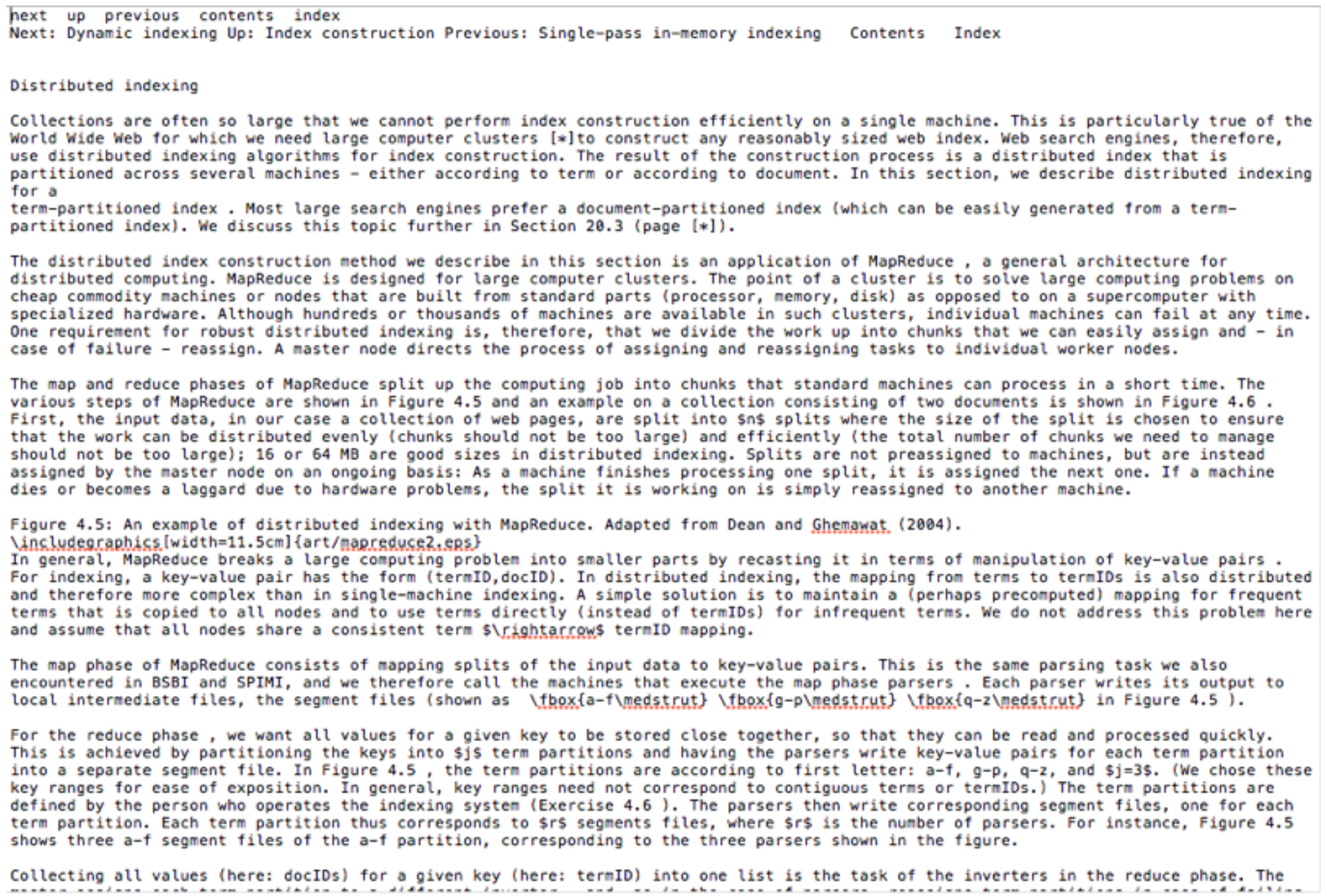
We have a large text file



```
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

```text
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
..	..

How can we do this in Spark?

Word Counts in Spark

We'll start by loading the file into an RDD

```
val lines = sc.textFile(textfilePath)
```

Each record in the RDD represents a line in the text file

Word Counts in Spark

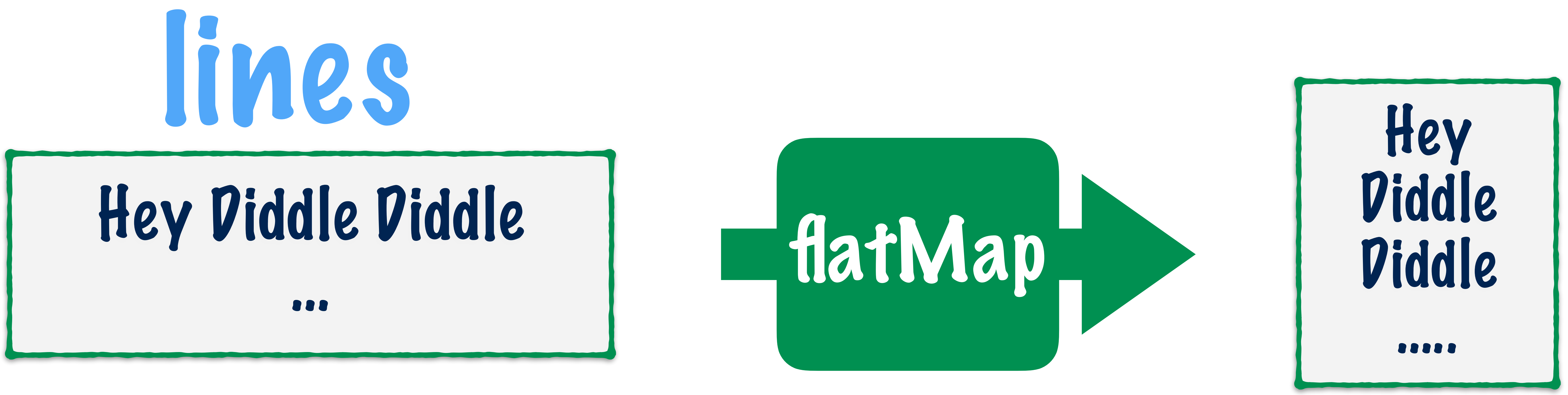
```
val lines = sc.textFile(textfilePath)
```

```
val wordsRDD=lines.flatMap(_.split(" ")).map(x => (x,1))
```

This is the **crucial step** in this exercise

Word Counts in Spark

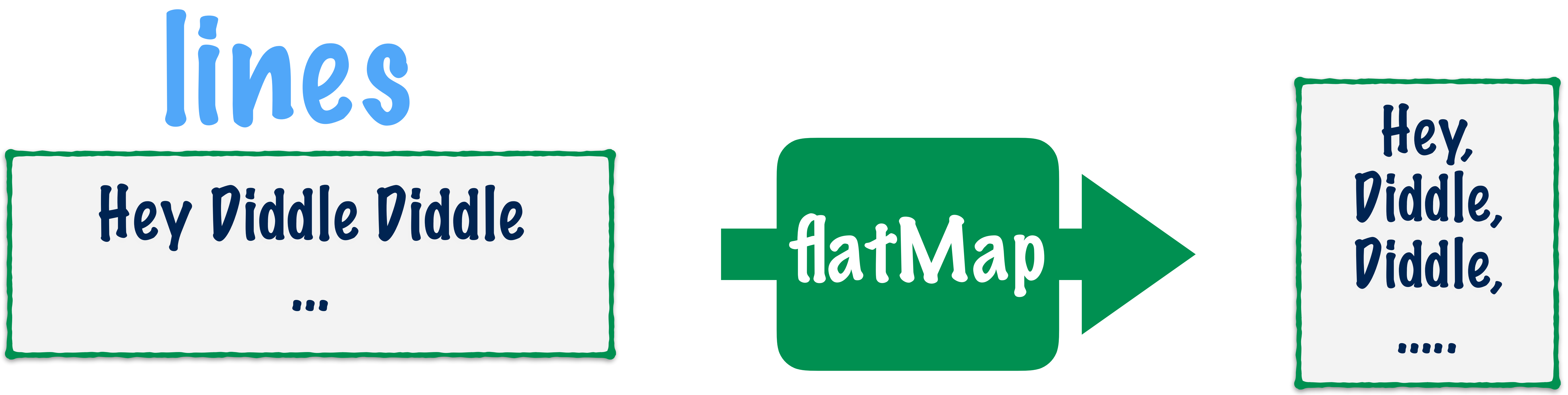
```
val wordsRDD=lines.flatMap(_.split(" ")).map(x => (x,1))
```



It creates an RDD in which each record is a word in the file

Word Counts in Spark

```
val wordsRDD=lines.flatMap(_.split(" ")).map(x => (x,1))
```



Let's parse what happened here

Word Counts in Spark

```
val wordsRDD=lines.flatMap(_.split(" ")).map(x => (x,1))
```

lines

Hey Diddle Diddle
...

[Hey,Diddle,Diddle]
...

This function creates an array for each record in the lines RDD

Word Counts in Spark

```
val wordsRDD=lines.flatMap(_.split(" ")).map(x => (x,1))
```

lines

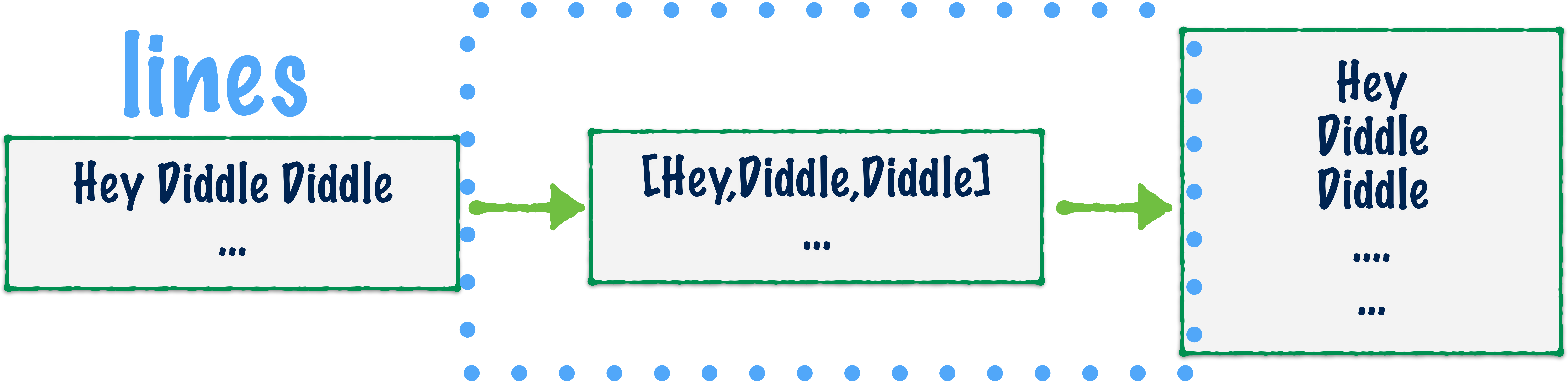
Hey Diddle Diddle ...	→	[Hey,Diddle,Diddle] ...

This is exactly what would have happened **if we used map instead of flatMap**

flatMap goes one step further

Word Counts in Spark

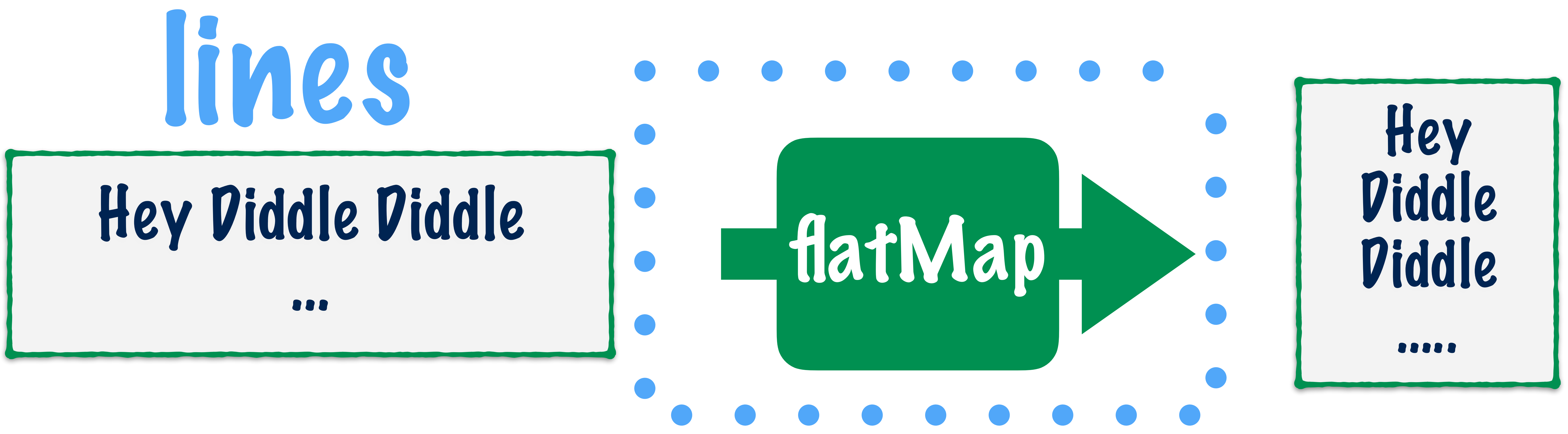
```
val wordsRDD=lines.flatMap(_.split(" ")).map(x => (x,1))
```



flatMap then creates one record for each element in the list

Word Counts in Spark

```
val wordsRDD=lines.flatMap(_.split(" ")).map(x => (x,1))
```



flatMap then creates one record for each element in the list

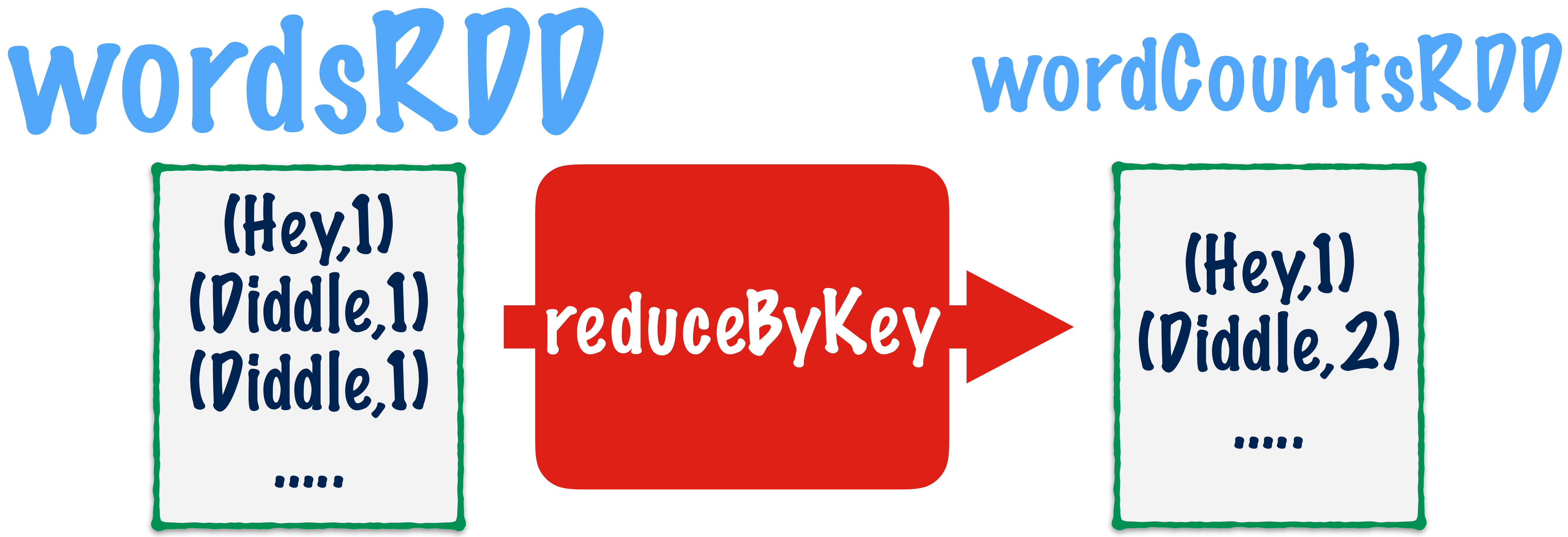
Word Counts in Spark

```
val wordsRDD=lines.flatMap(_.split(" ")).map(x => (x,1))
```

The map step creates a **Pair RDD** with the **value representing the count**

Word Counts in Spark

```
val wordCountsRDD=wordsRDD.reduceByKey(_+_)
```



Simply use reduceByKey to compute the sums

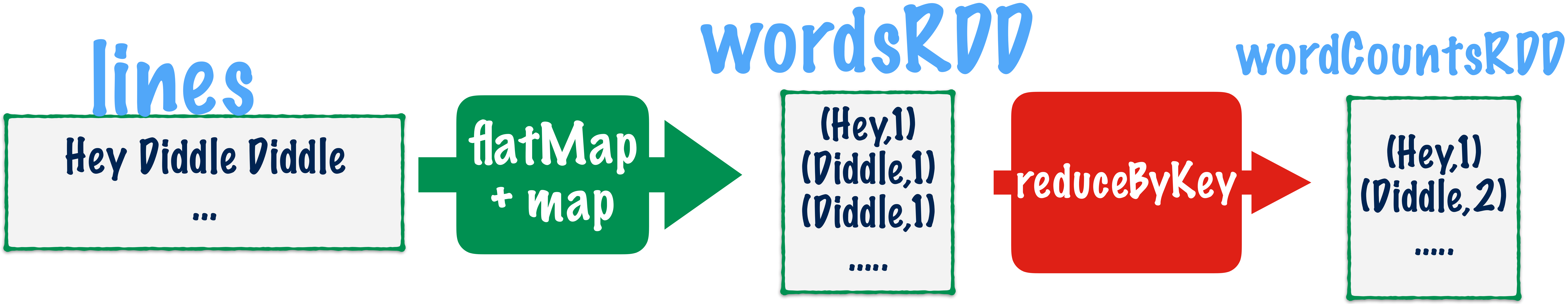
Word Counts in Spark

```
val wordCountsRDD=wordsRDD.reduceByKey(_+_)
```

This is just a concise way of writing

```
(x,y) => x+y
```

Word Counts in Spark



What we just did is a classic example of the MapReduce programming model

MapReduce

MapReduce is a programming model

Invented by Google

MapReduce

Hadoop uses MapReduce
for all it's computing tasks

MapReduce

Distributed computing can get very complicated

How to manage tasks across multiple nodes?

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()

Note. These are different from Spark's built in map and reduce operations

MapReduce

map() reduce()

The rest is taken care
of by Hadoop!

MapReduce

map() reduce()

This paradigm is driven by a key insight

MapReduce

key insight map() reduce()

ANY data processing task can be parallelized, if you express it as

<key, value> → **map()** → <key, value>

reduce() → <key, value>

MapReduce

key insight

map() reduce()

<key, value> → map() → <key, value>

reduce() → <key, value>

A map() task that transforms a key, value pair to a set of key, value pairs

MapReduce

key insight

map() reduce()

<key, value> → map() → <key, value>

reduce() → <key, value>

A reduce() task that combines values which have the same key

MapReduce

key insight map() reduce()

<key, value> → map() → <key, value>

→ reduce() → <key, value>

ANY task can be parallelized if it's expressed in this form

MapReduce

key insight

map() reduce()

<key, value> → map() → <key, value>

→ reduce() → <key, value>

ANY task can be parallelized if it's expressed in this form

or a chain of such transformations

MapReduce

While MapReduce is very powerful, it is also a little restrictive

In Hadoop, every task needs to be broken down into Map and Reduce tasks

This makes it difficult to intuitively express very complex tasks

MapReduce

With Spark, the user does not have to break down tasks into map and reduce

However, some tasks lend themselves beautifully to the MapReduce model

Word Counts in text documents, Building inverted indices

MapReduce

However, some tasks lend themselves beautifully to the MapReduce model

Word Counts in text documents,
Building inverted indices

..and so, Spark allows users to express these tasks using the MapReduce model

MapReduce

Hadoop	Spark
Map	flatMap
Reduce	reduceByKey