Thread Programming

Throughput computing focuses on delivering high volumes of computation in the form of transactions. Initially related to the field of transaction processing [60], throughput computing has since been extended beyond that domain. Advances in hardware technologies led to the creation of multicore systems, which have made possible the delivery of high-throughput computations, even in a single computer system. In this case, throughput computing is realized by means of multiprocessing and multithreading. *Multiprocessing* is the execution of multiple programs in a single machine, whereas *multithreading* relates to the possibility of multiple instruction streams within the same program.

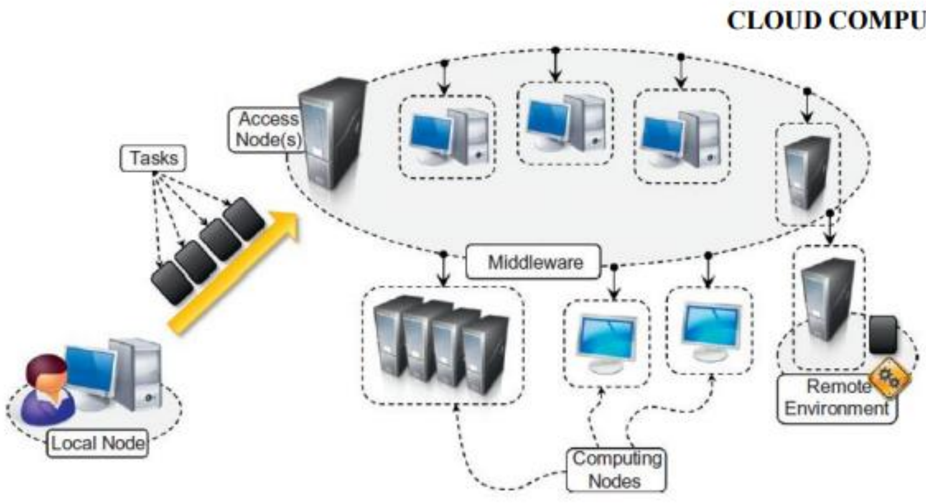
To support multithreaded programming, programming languages define the abstraction of process and thread in their class libraries. A popular standard for operations on threads and thread synchronization is POSIX, which is supported by all the Linux/UNIX operating systems and is available as an additional library for the Windows operating systems family. A common implementation of POSIX is given in C/C++ as a library of functions. New-generation languages such as Java and C# (.NET) provide a set of abstractions for thread management and synchronization that is compliant and that most closely follows the object-oriented design that characterizes these languages. These implementations are portable over any operating system that provides an implementation for the runtime environment required by these languages.

Multithreaded programming is a practice that allows achieving parallelism within the boundaries of a single machine. Applications requiring a high degree of parallelism cannot be supported by normal multithreaded programming and must rely on distributed infrastructures such as clusters, grids, or, most recently, clouds. The use of these facilities imposes application redesign and the use of specific APIs, which might require significant changes to the existing applications. To address this issue, Aneka provides the Thread Programming Model, which extends the philosophy behind multithreaded programming beyond the boundaries of a single node and allows leveraging heterogeneous distributed infrastructure for execution. To minimize application reconversion, the Thread Programming Model mimics the API of the *System. Threading* namespace, with some limitations that are imposed by the fact that threads are executed on a distributed infrastructure. High-throughput applications can be easily ported to Aneka threads with minimal or no changes at all to their logic. Examples of such features and the basic steps of converting a local multithreaded application to Aneka threads were given in the chapter by discussing simple applications demonstrating the methodology of domain and functional decomposition for parallel problems.

As a framework for distributed programming, Aneka provides many built-in features that are not generally of use while architecting an application in terms of concurrent threads. These are, for example, event notification and support for file transfer. These capabilities are available as core features of the Aneka application model but have not been demonstrated in the case of the Thread Programming Model, which is concerned with providing support for partitioning the execution of algorithms to speed up execution.

Task computing

A task identifies one or more operations that produce a distinct output and that can be isolated as a single logical unit. In practice, a task is represented as a distinct unit of code, or a program, that can be separated and executed in a remote run time environment. Multithreaded programming is mainly concerned with providing a support for parallelism within a single machine. Task computing provides distribution by harnessing the compute power of several computing nodes. Hence, the presence of a distributed infrastructure is explicit in this model. Now clouds have emerged as an attractive solution to obtain a huge computing power on demand for the execution of distributed applications. To achieve it, suitable middleware is needed. A reference scenario for task computing is depicted inn Figure



The middleware is a software layer that enables the coordinated use of multiple resources, which are drawn from a data center or geographically distributed networked computers. A user submits the collection of tasks to the access point(s) of the middleware, which will take care of scheduling and monitoring the execution of tasks. Each computing resource provides an appropriate runtime environment. Task submission is done using the APIs provided by the middleware, whether a Web or programming language interfaces. Appropriate APIs are also provided to monitor task status and collect their results upon completion. It is possible to identify a set of common operations that the middleware needs to support the creation and execution of task-based applications. These operations are:

• Coordinating and scheduling tasks for execution on a set of remote nodes

• Moving programs to remote nodes and managing their dependencies

• Creating an environment for execution of tasks on the remote nodes

• Monitoring each task's execution and informing the user about its status

• Access to the output produced by the task.

Characterizing a task

A task represents a component of an application that can be logically isolated and executed separately. A task can be represented by different elements:

• A shell script composing together the execution of several applications

• A single program

• A unit of code (a Java/C11/.NET class) that executes within the context of a specific runtime environment. A task is characterized by input files, executable code (programs, shell scripts, etc.), and output files. The runtime environment in which tasks execute is the operating system or an equivalent sandboxed environment. A task may also need specific software appliances on the remote execution nodes.

Computing categories

These categories provide an overall view of the characteristics of the problems. They implicitly impose requirements on the infrastructure and the middleware. Applications falling into this category are:

1. High-performance computing
2. High-throughput computing
3. Many-task computing

High-performance computing

High-performance computing (HPC) is the use of distributed computing facilities for solving problems that need large computing power. The general profile of HPC applications is constituted by a large collection of compute-intensive tasks that need to be processed in a short period of time. The metrics to evaluate HPC systems are floating-point operations per second (FLOPS), now tera-FLOPS or even peta-FLOPS, which identify the number of floating- point operations per second. Ex: supercomputers and clusters are specifically designed to support HPC applications that are developed to solve “Grand Challenge” problems in science and engineering.

High-throughput computing

High-throughput computing (HTC) is the use of distributed computing facilities for applications requiring large computing power over a long period of time. HTC systems need to be robust and to reliably operate over a long time scale. The general profile of HTC applications is that they are made up of a large number of tasks of which the execution can last for a considerable amount of time. Ex: scientific simulations or statistical analyses. It is quite common to have independent tasks that can be scheduled in distributed resources because they do not need to communicate. HTC systems measure their performance in terms of jobs completed per month.

Many-task computing

MTC denotes high-performance computations comprising multiple distinct activities coupled via file system operations. MTC is the heterogeneity of tasks that might be of different nature: Tasks may be small or large, single processor or multiprocessor, compute-intensive or data-intensive, static or dynamic, homogeneous or heterogeneous. MTC applications include loosely coupled applications that are communication-intensive but not naturally expressed using the message-passing interface. It aims to bridge the gap between HPC and HTC. MTC is similar to HTC, but it concentrates on the use of many computing resources over a short period of time to accomplish many computational tasks.

Frameworks for task computing

Some popular software systems that support the task-computing framework are:

1. Condor

2. Globus Toolkit

3. Sun Grid Engine (SGE)

4. BOINC

5. Nimrod/G

Architecture of all these systems is similar to the general reference architecture depicted in Figure. They consist of two main components: a scheduling node (one or more) and worker nodes. The organization of the system components may vary.

Condor

Condor is the most widely used and long-lived middleware for managing clusters, idle workstations, and a collection of clusters. Condors support features of batch-queuing systems along with the capability to checkpoint jobs and manage overload nodes. It provides a powerful job resource-matching mechanism, which schedules jobs only on resources that have the appropriate runtime environment. Condor can handle both serial and parallel jobs on a wide variety of resources. It is used by hundreds of organizations in industry, government, and academia to manage infrastructures. Condor-G is a version of Condor that supports integration with grid computing resources, such as those managed by Globus.

Globus Toolkit the Globus Toolkit is a collection of technologies that enable grid computing. It provides a comprehensive set of tools for sharing computing power, databases, and other services across corporate, institutional, and geographic boundaries. The toolkit features software services, libraries, and tools for resource monitoring, discovery, and management as well as security and file management.

Sun Grid Engine (SGE) Sun Grid Engine (SGE), now Oracle Grid Engine, is middleware for workload and distributed resource management. Initially developed to support the execution of jobs on clusters, SGE integrated additional capabilities and now is able to manage heterogeneous resources and constitutes middleware for grid computing. It supports the execution of parallel, serial, interactive, and parametric jobs and features advanced scheduling capabilities such as budget-based and group- based scheduling, scheduling applications that have deadlines, custom policies, and advance reservation.

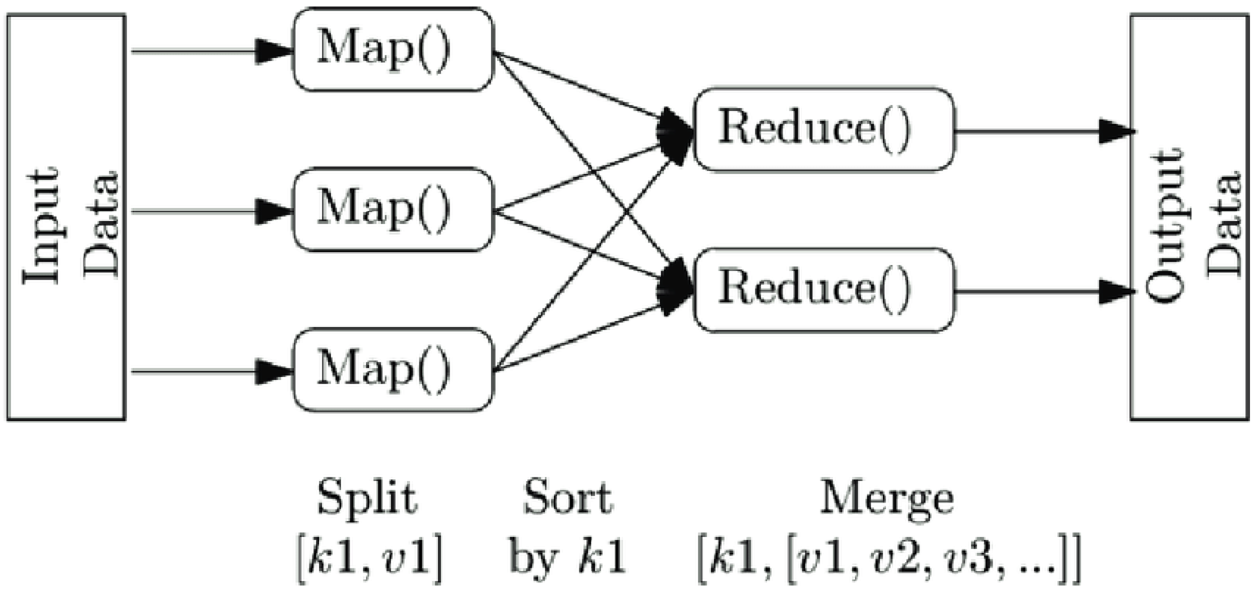
BOINC Berkeley Open Infrastructure for Network Computing (BOINC) is framework for volunteer and grid computing. It allows us to turn desktop machines into volunteer computing nodes that are leveraged to run jobs when such machines become inactive. BOINC supports job check pointing and duplication. BOINC is composed of two main components: the BOINC server and the BOINC client. The BOINC server is the central node that keeps track of all the available resources and scheduling jobs. The BOINC client is the software component that is deployed on desktop machines and that creates the BOINC execution environment for job submission. BOINC systems can be easily set up to provide more stable support for job execution by creating computing grids with dedicated machines. When installing BOINC clients, users can decide the application project to which they want to donate the CPU cycles of their computer. Currently several projects, ranging from medicine to astronomy and cryptography, are running on the BOINC infrastructure.

Nimrod/G Tool for automated modeling and execution of parameter sweep applications over global computational grids. It provides a simple declarative parametric modeling language for expressing parametric experiments. It uses novel resource management and scheduling algorithms based on economic principles. It supports deadline- and budget-constrained scheduling of applications on distributed grid resources to minimize the execution cost and at the same deliver results in a timely manner. It has been used for a very wide range of applications over the years, ranging from quantum chemistry to policy and environmental impact.

Map Reduce a Programming Model for Cloud Computing Based On Hadoop Ecosystem
Cloud Computing is emerging as a new computational paradigm shift. Hadoop Map-Reduce has become a powerful Computation Model for processing large data on distributed commodity hardware clusters such as Clouds. Map-Reduce is a programming model developed for large-scale analysis. It takes advantage of the parallel processing capabilities of a cluster in order to quickly process very large datasets in a fault-tolerant and scalable manner. The core idea behind Map-Reduce is mapping the data into a collection of key/value pairs, and then reducing over all pairs with the same key. Using key/value pairs as its basic data unit, the framework is able to work with the less-structured data types and to address a wide range of problems. In Hadoop, data can originate in any form, but in order to be analyzed by Map-Reduce software, it needs to eventually be transformed into key/value pairs. 'cloud' is an elastic execution environment of resources involving multiple stakeholders and providing a metered service at multiple granularities for a specified level of quality of service to be more specific, a cloud is a platform or infrastructure that enables execution of code (services, applications etc.), in a managed and elastic fashion, whereas “managed” means that reliability according to predefined quality parameters is automatically ensured and “elastic” implies that the resources are put to use according to actual current requirements observing overarching requirement definitions implicitly, elasticity includes both up and downward scalability of resources and data, but also load balancing of data throughput with Google Docs. Due this combinatorial capability, these types are also often referred to as “components”

MapReduce programming model using two components: A Job Tracker (masternode) and many Task Trackers (slave nodes). The Job Tracker is responsible for accepting job requests, for splitting the data input, for defining the tasks required for the job, for assigning those tasks to be executed in parallel across the slaves, for monitoring the progress and finally for handling occurring failures. The Task Tracker executes tasks as ordered by the master node. The task can be either a map (takes a key/value and generates another key/value) or a reduce (takes a key and all associated values and generates a key/value pair). The map function can run independently on each key/value pair, enabling enormous amounts of parallelism. Likewise, each reducer can also run separately on each key enabling further parallelism. When a job is submitted to the Job Tracker, the Job Tracker selects a number of Task Trackers (not randomly but according to data locality) to execute a map task (Mappers) and a number of Task Trackers to execute a reduce task (Reducers). The job input data is divided into splits and is organized as a stream of keys/values records. In each split there is a matching mapper which converts the original records into intermediate results which are again in the form of key/value. The intermediate results are divided into partitions (each partition has a range of keys), which after the end of the map phase are distributed to the reducers .Finally reducers apply a reduce function on each key

Map reduce computation



A Map-Reduce paradigm is given in Figure. Map-Reduce are designed to continue to work in the face of system failures. When a job is running, Map-Reduce monitor progress of each of the servers participating in the job. If one of them is slow in returning an answer or fails before completing its work, Map-Reduce automatically starts another instance of that task on another server that has a copy of the data. The complexity of the error handling mechanism is completely hidden from the programmer. A Map Reduce Computation Map-Reduce is triggered by the map and reduce operations in functional languages, such as Lisp. This model abstracts computation problems through two functions: map and reduce. All problems formulated in this way can be parallelized automatically. Essentially, the Map-Reduce model allows users to write map/reduce components with functional-style code. These components are then composed as a dataflow graph to explicitly specify their parallelism. Finally, the Map-Reduce runtime system schedules these components to distributed resources for execution while handling many tough problems: parallelization, network communication, and fault tolerance. A map function takes a key/value pair as input and produces a list of key/value pairs as output. The type of output key and value can be different from input: map :: (key1; value1) ◊ list(key2; value2)… (1) A reduce function takes a key and associated value list as input and generates a list of new values as output: reduce :: (key2; list(value2)) -> list(value3)… (2) A Map-Reduce application is executed in a parallel manner through two phases. In the first phase, all map operations can be executed independently from each other. In the second phase, each reduce operation may depend on the outputs generated by any number of map operations. All reduce operations can also be executed independently similar to map operations.

Uses of map reduce

At Google:

– Index building for Google Search

– Article clustering for Google News

– Statistical machine translation At Yahoo!:

– Index building for Yahoo! Search

– Spam detection for Yahoo! Mail

At Facebook:

– Ad optimization

– Spam detection

Map-reduce workflow

When we write a Map-Reduce workflow, we'll have to create 2 scripts: the mapscript, and the reduce script. The rest will be handled by the Amazon Elastic Map-Reduce (EMR) framework. When we start a map/reduce workflow, the framework will split the input into segments, passing each segment to a different machine. Each machine then runs the map script on the portion of data attributed to it. The map script (which you write) takes some input data, and maps it to pairs according to your specifications. For example, if we wanted to count word frequencies in a text, we'd have be our pairs. Map-Reduce then would emit a pair for each word in the input stream. Note that the map script does no aggregation (i.e. actual counting) this is what the reduce script it for. The purpose of the map script is to model the data into pairs for the reducer to aggregate.

Emitted pairs are then “shuffled” (to use the terminology in the diagram below), which basically means that pairs with the same key are grouped and passed to a single machine, which will then run the reduce script over them. The reduce script (which you also write) takes a collection of pairs and “reduces” them according to the user-specified reduce script. In our word count example, we want to count the number of word occurrences so that we can get frequencies. Thus, we’d want our reduce script to simply sum the values of the collection of pairs which have the same key. The Figure word count example below illustrates the described scenario nicely

Work Flow of the Program

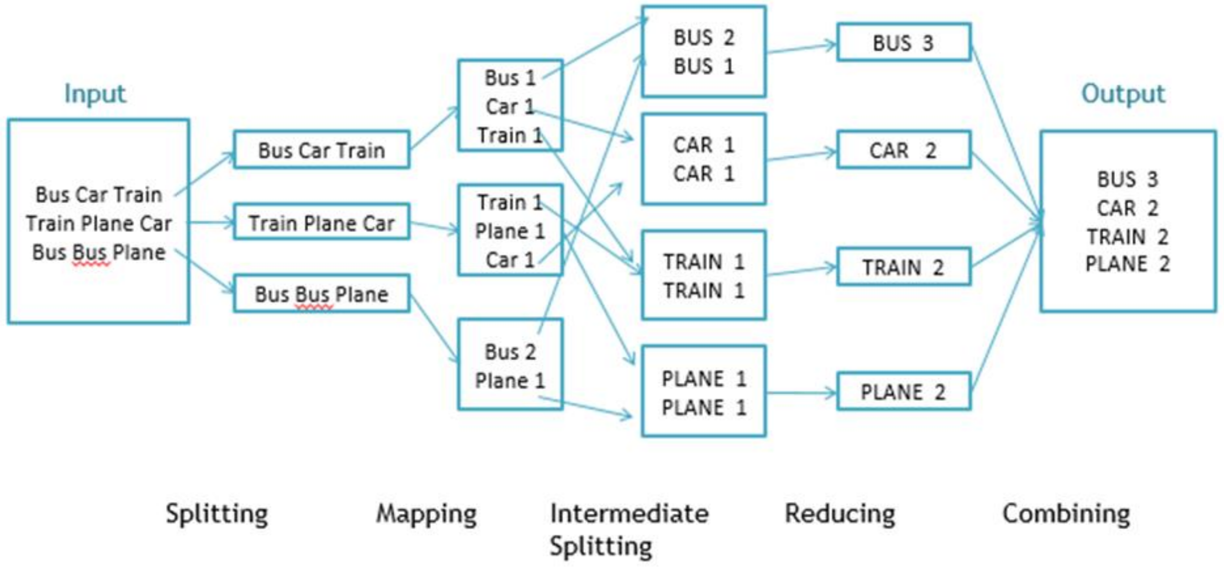


Fig. WorkFlow of MapReducing

Workflow of Map-Reduce consists of 5 steps:

1. **Splitting** – The splitting parameter can be anything, e.g. splitting by space, comma, semicolon, or even by a new line (‘\n’).

2. **Mapping** – as explained above.

3. **Intermediate splitting** – the entire process in parallel on different clusters. In order to group them in “Reduce Phase” the similar KEY data should be on the same cluster.

4. **Reduce** – it is nothing but mostly group by phase.

5. **Combining** – The last phase where all the data (individual result set from each cluster) is combined together to form a result.

Word Count Program in Java

Fortunately, we don’t have to write all of the above steps, we only need to write the splitting parameter, Map function logic, and Reduce function logic. The rest of the remaining steps will execute automatically.

Steps

1. **Open Eclipse> File > New > Java Project >(Name it – MRProgramsDemo) > Finish.**

2. **Right Click > New > Package (Name it - PackageDemo) > Finish.**

3. **Right Click on Package > New > Class (Name it - WordCount).**

4. **Add Following Reference Libraries:**

    1. Right Click on Project > Build Path> Add External

        1. */usr/lib/hadoop-0.20/**hadoop-core.jar***

        2. *Usr/lib/hadoop-0.20/lib/**Commons-cli-1.2.jar***

5. Type the following code:

```
package PackageDemo;


import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;



public class WordCount {
public static void main(String [] args) throws Exception
{
Configuration c=new Configuration();
String[] files=new GenericOptionsParser(c,args).getRemainingArgs();
Path input=new Path(files[0]);
Path output=new Path(files[1]);
Job j=new Job(c,"wordcount");
j.setJarByClass(WordCount.class);
j.setMapperClass(MapForWordCount.class);
j.setReducerClass(ReduceForWordCount.class);
j.setOutputKeyClass(Text.class);
j.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(j, input);
FileOutputFormat.setOutputPath(j, output);
System.exit(j.waitForCompletion(true)?0:1);
}
public static class MapForWordCount extends Mapper<LongWritable, Text, Text, IntWritable>{
```

```
public void map(LongWritable key, Text value, Context con) throws IOException, InterruptedEx
ception
{
String line = value.toString();
String[] words=line.split(",");
for(String word: words )
{
      Text outputKey = new Text(word.toUpperCase().trim());
  IntWritable outputValue = new IntWritable(1);
  con.write(outputKey, outputValue);
}
}
}

public static class ReduceForWordCount extends Reducer<Text, IntWritable, Text, IntWritable>
{
public void reduce(Text word, Iterable<IntWritable> values, Context con) throws IOException, I
nterruptedException
{
int sum = 0;
   for(IntWritable value : values)
   {
   sum += value.get();
   }
   con.write(word, new IntWritable(sum));
}


}
```

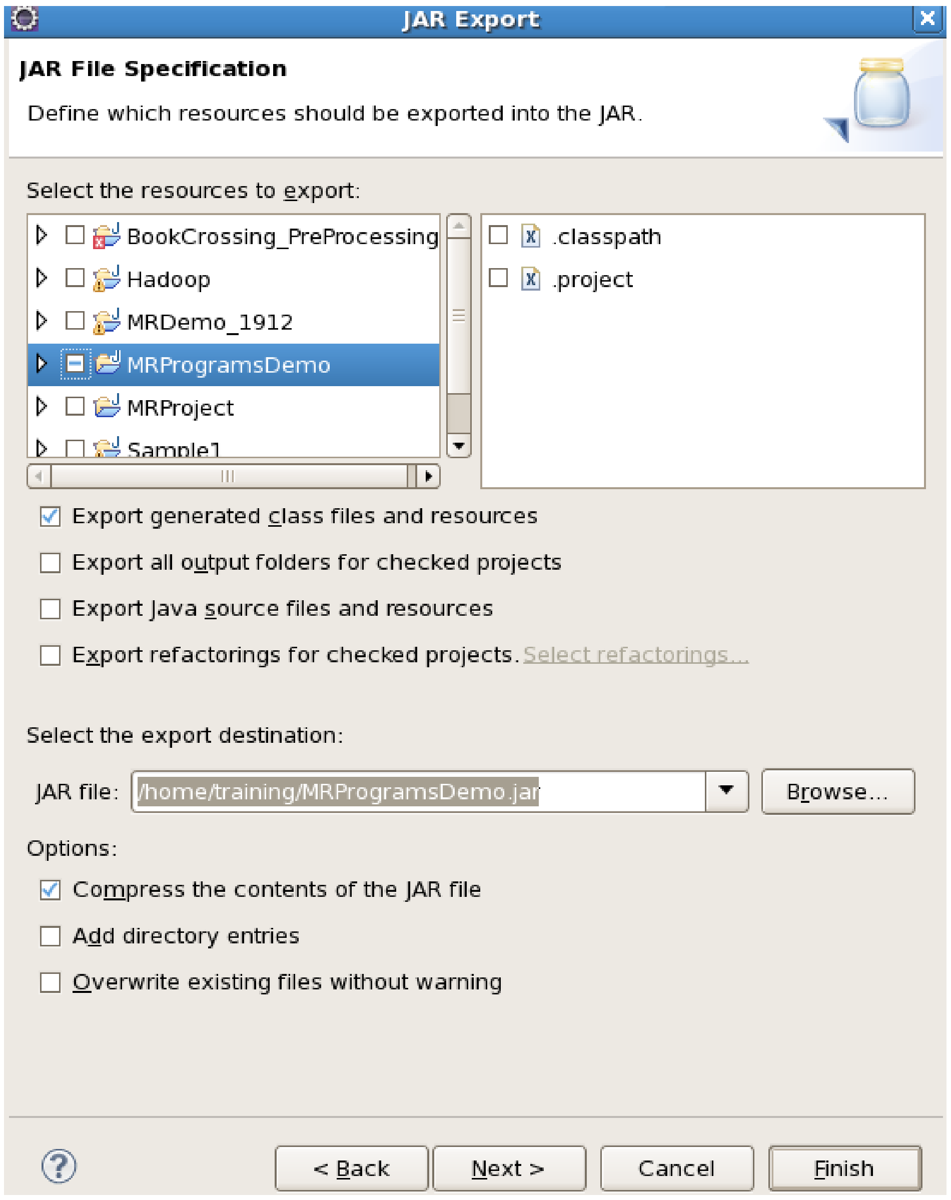
The above program consists of three classes:

- Driver class (Public, void, static, or main; this is the entry point).

- The Map class which **extends** the public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> and implements the Map function.
- The Reduce class which extends the public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> and implements the Reduce function.

6. Make a jar file

Right Click on Project> Export> Select export destination as **Jar File** > next> Finish.


JAR Export

JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

- [ ] BookCrossing_PreProcessing
- [ ] Hadoop
- [ ] MRDemo_1912
- [–] MRProgramsDemo
- [ ] MRProject
- [ ] Sample1

- [ ] .classpath
- [ ] .project

- [✓] Export generated class files and resources
- [ ] Export all output folders for checked projects
- [ ] Export Java source files and resources
- [ ] Export refactorings for checked projects. Select refactorings...

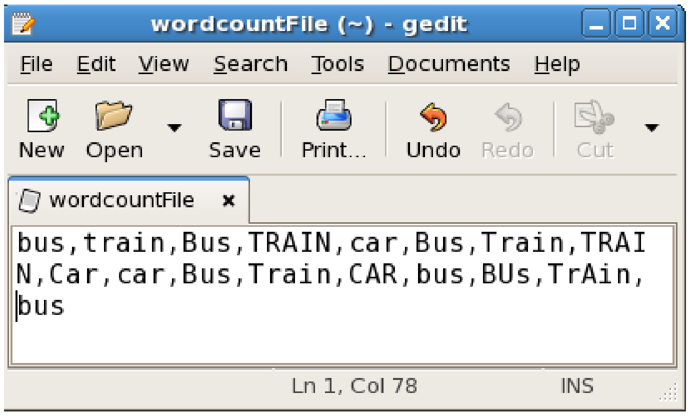
Select the export destination:

JAR file: /home/training/MRProgramsDemo.jar

Options:

- [✓] Compress the contents of the JAR file
- [ ] Add directory entries
- [ ] Overwrite existing files without warning

7. Take a text file and move it into HDFS format:


wordcountFile (~) - gedit
File Edit View Search Tools Documents Help
New Open Save Print... Undo Redo Cut
wordcountFile
Ln 1, Col 78
INS

```
bus,train,Bus,TRAIN,car,Bus,Train,TRAI
N,Car,car,Bus,Train,CAR,bus,BUs,TrAin,
bus
```

To move this into Hadoop directly, open the terminal and enter the following commands:

```
[training@localhost ~]$ hadoop fs -put wordcountFile wordCountFile
```

8. Run the jar file:

(Hadoop jar jarfilename.jar packageName.ClassName PathToInputTextFile PathToOutputDirectry)

```
[training@localhost ~]$ hadoop jar MRProgramsDemo.jar PackageDemo.WordCount wordCoun
tFile MRDir1
```

9. Open the result:

```
[training@localhost ~]$ hadoop fs -ls MRDir1

Found 3 items

-rw-r--r--   1 training supergroup          0 2016-02-23 03:36 /user/training/MRDir1/_SUCCESS

drwxr-xr-x   - training supergroup          0 2016-02-23 03:36 /user/training/MRDir1/_logs

-rw-r--r--   1 training supergroup         20 2016-02-23 03:36 /user/training/MRDir1/part-r-00000
```

```
[training@localhost ~]$ hadoop fs -cat MRDir1/part-r-00000
BUS     7
CAR     4
TRAIN   6
```

Enterprise batch processing using Map-Reduce

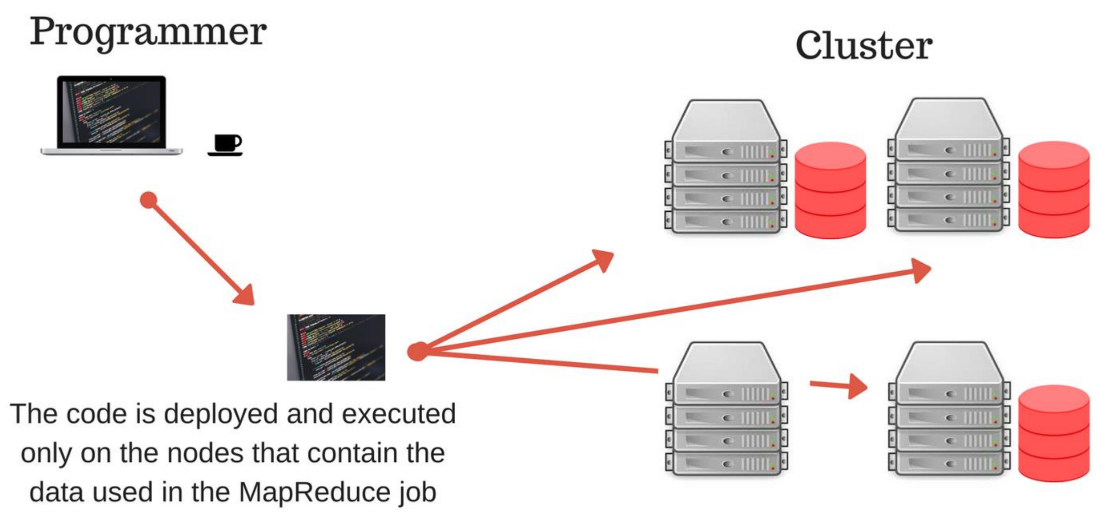
Batch processing is an automated job that does some computation, usually done as a periodical job. It runs the processing code on a set of inputs, called a batch. Usually, the job will read the batch data from a database and store the result in the same or different database. An example of a batch processing job could be reading all the sale logs from an online shop for a single day and aggregating it into statistics for that day (number of users per country, the average spent amount, etc.). Doing this as a daily job could give insights into customer trends.

For Map-Reduce to be able to do computation on large amounts of data, it has to be a distributed model that executes its code on multiple nodes. This allows the computation to handle larger amounts of data by adding more machines – **horizontal scaling**. This is different from **vertical scaling**, which implies increasing the performance of a single machine.

Execution

In order to decrease the duration of our distributed computation, Map-Reduce tries to reduce **shuffling** (moving) the data from one node to another by distributing the computation so that it is done on the same node where the data is stored. This way, the data stays on the same node, but the code is moved via the network. This is ideal because the code is much smaller than the data.

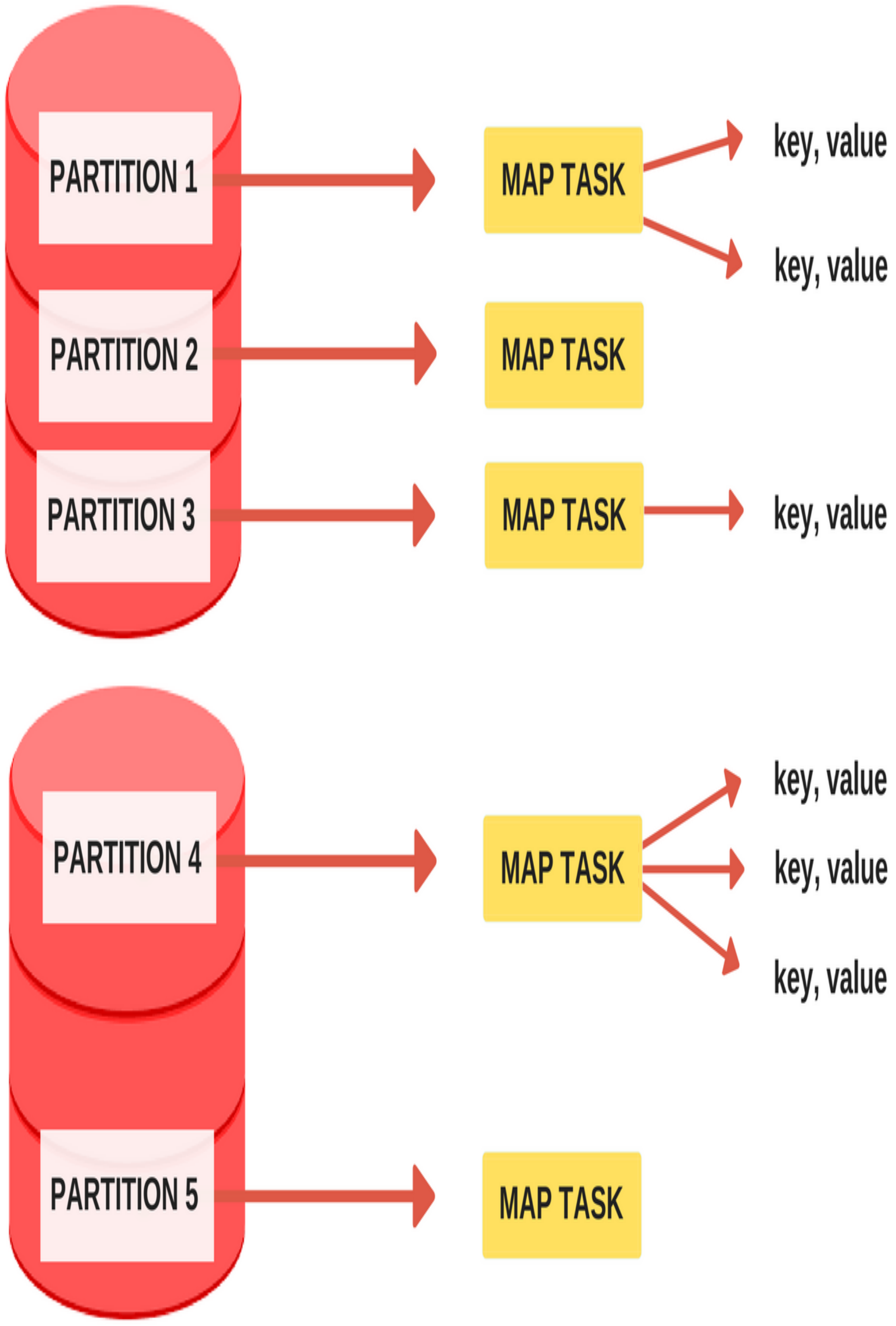
To run a Map-Reduce job, the user has to implement two functions, **map** and **reduce**, and those implemented functions are distributed to nodes that contain the data by the Map-Reduce framework. Each node runs (executes) the given functions on the data it has in order the minimize network traffic (shuffling data).


Programmer
Cluster
The code is deployed and executed only on the nodes that contain the data used in the MapReduce job

The computation performance of Map-Reduce comes at the cost of its expressivity. When writing a Map-Reduce job we have to follow the strict interface (return and input data structure) of the map and the reduce functions. The map phase generates key-value data pairs from the input data (partitions), which are then grouped by key and used in the reduce phase by the reduce task. Everything except the interface of the functions is programmable by the user.

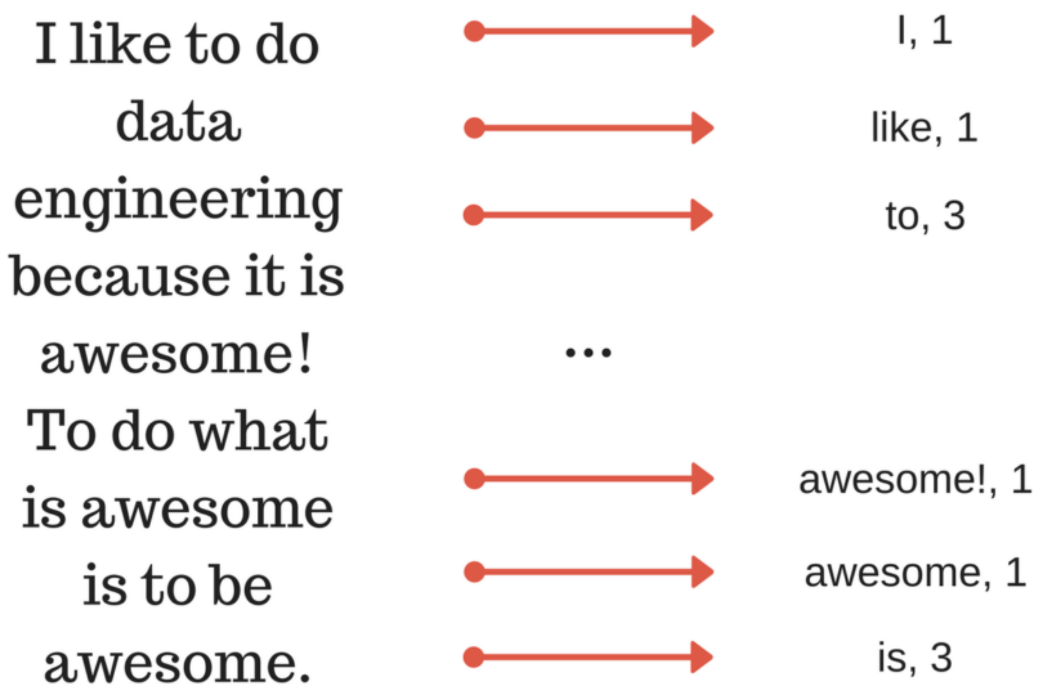
Map

Hadoop, along with its many other features, had the first open-source implementation of Map-Reduce. It also has its own distributed file storage called HDFS. In Hadoop, the typical input into a Map-Reduce job is a directory in HDFS. In order to increase parallelization, each directory is made up of smaller units called partitions and each partition can be processed separately by a map task (the process that executes the map function). This is hidden from the user, but it is important to be aware of it because the number of partitions can affect the speed of execution.



The map task (mapper) is called once for every input partition and its job is to extract key-value pairs from the input partition. The mapper can generate any number of key-value pairs from a single input (including zero, see the figure above). The user only needs to define the code inside the mapper. Below, we see an example of a simple mapper that takes the input partition and outputs each word as a key with value 1.

```
# Map function, is applied on a partition

def mapper(key, value):

# Split the text into words and yield word,1 as a pair

for word in value.split():

normalized_word = world.lower()

yield normalized_word, 1
```


I like to do data engineering because it is awesome! To do what is awesome is to be awesome.
I, 1
like, 1
to, 3
...
awesome!, 1
awesome, 1
is, 3

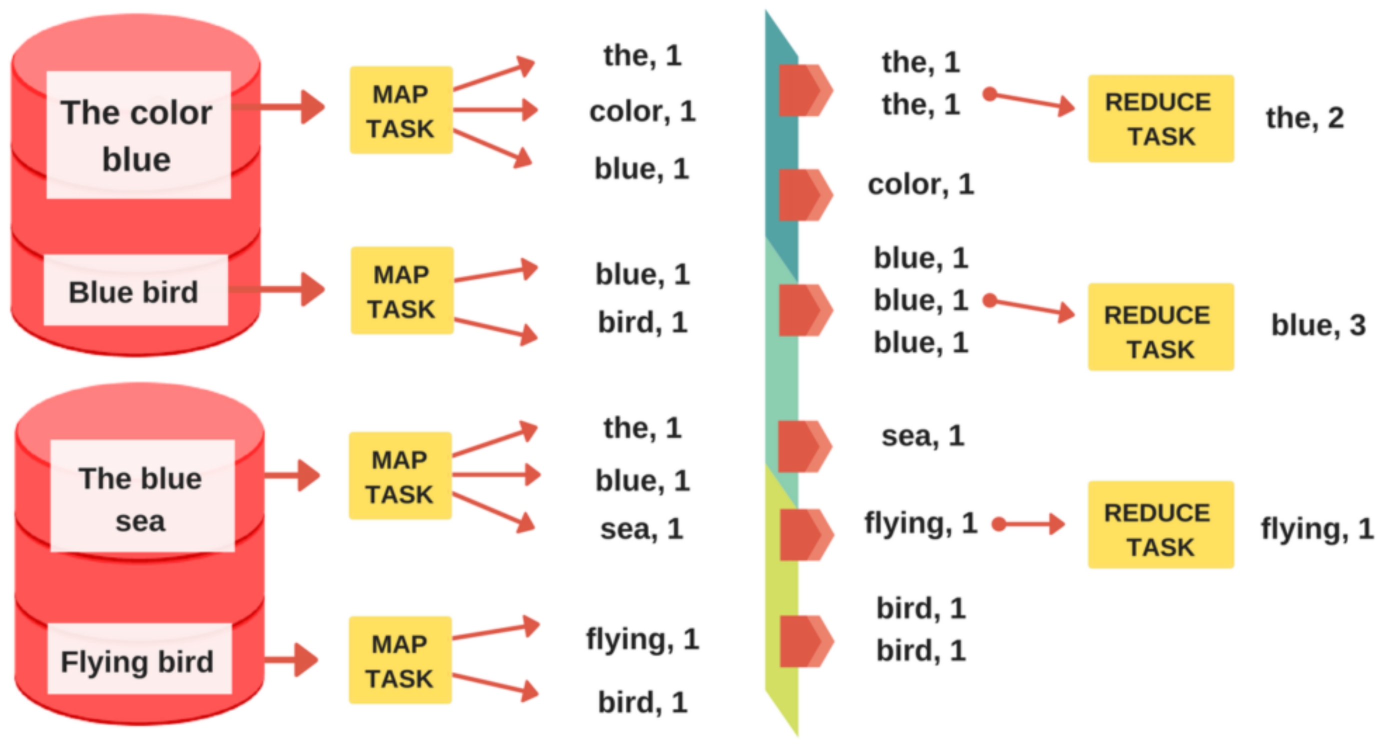
Reduce

The MapReduce framework collects all the key-value pairs produced by the mappers, arranges them into groups with the same key and applies the reduce function. All the grouped values entering the reducers are sorted by the framework. The reducer can produce output files which can serve as input into another MapReduce job, thus enabling multiple MapReduce jobs to chain into a more complex data processing pipeline.

```
# Reduce function, applied to a group of values with the same key

def reducer(key, values):

# Sum all the values with the same key

result = sum(values)

return result
```

The mapper yielded key-value pairs with the word as the key and the number 1 as the value. The reducer can be called on all the values with the same key (word), to create a distributed word counting pipeline. In the image below, we see that not every sorted group has a reduce task. This happens because the user needs to define the number of reducers, which is 3 in our case. After a reducer is done with its task, it takes another group if there is one that was not processed.

SORT and SHUFFLE


The color blue
Blue bird
The blue sea
Flying bird
MAP TASK
the, 1
color, 1
blue, 1
MAP TASK
blue, 1
bird, 1
MAP TASK
the, 1
blue, 1
sea, 1
MAP TASK
flying, 1
bird, 1
the, 1
the, 1
color, 1
blue, 1
blue, 1
blue, 1
sea, 1
flying, 1
bird, 1
bird, 1
REDUCE TASK
the, 2
REDUCE TASK
blue, 3
REDUCE TASK
flying, 1