A main advantage of cloud computing is elasticity – the ability to use as many servers as necessary to

optimally respond to the cost and the timing constraints of an application. In the case of transaction

processing systems, typically a front-end system distributes the incoming transactions to a number

of back-end systems and attempts to balance the load among them. As the workload increases, new

back-end systems are added to the pool.

For data-intensive batch applications, partitioning the workload is not always trivial. Only in some

cases can the data be partitioned into blocks of arbitrary size and processed in parallel by servers in the

cloud. We distinguish two types of divisible workloads:

• Modularly divisible. The workload partitioning is defined a priori.

• Arbitrarily divisible. The workload can be partitioned into an arbitrarily large number of smaller

workloads of equal or very close size.

Many realistic applications in physics, biology, and other areas of computational science and engineering

obey the arbitrarily divisible load-sharing model. The Divisible Load Theory (DLT) is analyzed in the

literature (see Section 4.12).

MapReduce is based on a very simple idea for parallel processing of data-intensive applications

supporting arbitrarily divisible load sharing. First, split the data into blocks, assign each block to

an instance or process, and run these instances in parallel. Once all the instances have finished, the

computations assigned to them start the second phase: Merge the partial results produced by individual instances. The so-called same program, multiple data (SPMD) paradigm, used since the early days of

parallel computing, is based on the same idea but assumes that a master instance partitions the data and

gathers the partial results.

MapReduce is a programming model inspired by the Map and the Reduce primitives of the LISP

programming language. It was conceived for processing and generating large data sets on computing

clusters [100]. As a result of the computation, a set of input <key, value> pairs is transformed into a

set of output <key, value> pairs.

Numerous applications can be easily implemented using this model. For example, one can process

logs of Web page requests and count the URL access frequency. The Map function outputs the

pairs <URL, 1> and the Reduce function produces the pairs <URL, totalcount>. Another trivial

example is distributed sort when the map function extracts the key from each record and produces a

<key, record> pair and the Reduce function outputs these pairs unchanged. The following example

[100] shows the two user-defined functions for an application that counts the number of occurrences of

each word in a set of documents.

map(String key, String value):

//key: document name; value: document contents

for each word w in value:

EmitIntermediate (w, "1");

reduce (String key, Iterator values):

// key: a word; values: a list of counts

int result = 0;

for each v in values:

result += ParseInt (v);

Emit (AsString (result));

Call M and R the number of Map and Reduce tasks, respectively, and N the number of systems used

by the MapReduce. When a user program invokes the MapReduce function, the following sequence of

actions take place (see Figure 4.6):

1. The run-time library splits the input files into M splits of 16 to 64 MB each, identifies a number N of

systems to run, and starts multiple copies of the program, one of the system being a master and the

others workers. The master assigns to each idle system either a Map or a Reduce task. The master

makesO(M+R) scheduling decisions and keepsO(M R) worker state vectors in memory. These

considerations limit the size of M and R; at the same time, efficiency considerations require that

M, R   N.

2. A worker being assigned a Map task reads the corresponding input split, parses<key, value>pairs,

and passes each pair to a user-defined Map function. The intermediate<key, value>pairs produced

by the Map function are buffered in memory before being written to a local disk and partitioned into

R regions by the partitioning function.

3. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible

for forwarding these locations to the Reduce workers. A Reduce worker uses remote procedure

calls to read the buffered data from the local disks of the Map workers; after reading all the intermediate data, it sorts it by the intermediate keys. For each unique intermediate key, the key

and the corresponding set of intermediate values are passed to a user-defined Reduce function. The

output of the Reduce function is appended to a final output file.

4. When all Map and Reduce tasks have been completed, the master wakes up the user program.

The system is fault tolerant. For each Map and Reduce task, the master stores the state (idle, in progress,

or completed) and the identity of the worker machine. The master pings every worker periodically

and marks the worker as failed if it does not respond. A task in progress on a failed worker is reset

to idle and becomes eligible for rescheduling. The master writes periodic checkpoints of its control data structures and, if the task fails, it can be restarted from the last checkpoint. The data is stored using

GFS, the Google File System, discussed in Section 8.5.

An environment for experimenting with MapReduce is described in [100]: The computers are typically

dual-processor x86 running Linux, with 2–4 GB of memory per machine and commodity networking

hardware typically 100–1,000 Mbps. A cluster consists of hundreds or thousands of machines.

Data is stored on IDE7 disks attached directly to individual machines. The file system uses replication

to provide availability and reliability with unreliable hardware. To minimize network bandwidth, the

input data is stored on the local disks of each system.