Module - 4

Data-Intensive Computing: MapReduce Programming (Chapter - 8)

Data-intensive computing focuses on aa class of applications that deal with a large amount of data. Several application fields, ranging from computational science to social networking, produce large volumes of data that need to be efficiently stored, made accessible, indexed, and analyzed.

Distributed computing is definitely of help in addressing these challenges by providing more scalable and efficient storage architectures and a better performance in terms of data computation and processing.

This chapter characterizes the nature of data-intensive computing and presents an overview of the challenges introduced by production of large volumes of data and how they are handled by storage systems and computing models. It describes MapReduce, which is a popular programming model for creating data-intensive applications and their deployment on clouds.

8.1 What is data-intensive computing?

Data-intensive computing is concerned with **production, manipulation, and analysis** of **large-scale data** in the range of hundreds of **megabytes (MB) to petabytes (PB)** and **beyond**.

Dataset is commonly used to identify a collection of information elements that is relevant to one or more applications. Datasets are often maintained in repositories, which are infrastructures supporting the storage, retrieval, and indexing of large amounts of information.

To facilitate classification and search, relevant bits of information, called **metadata**, are attached to datasets.

Data-intensive computations occur in many application domains.

Computational science is one of the most popular ones. People conducting scientific simulations and experiments are often keen to produce, analyze, and process huge volumes of data. Hundreds of gigabytes of data are produced every second by telescopes mapping the sky; the collection of images of the sky easily reaches the scale of petabytes over a year.

Bioinformatics applications mine databases that may end up containing terabytes of data.

Earthquake simulators process a massive amount of data, which is produced as a result of recording the vibrations of the Earth across the entire globe.

8.1.1 Characterizing data-intensive computations

8.1.2 Challenges ahead

8.1.3 Historical perspective

1 The early age: high-speed wide-area networking

2 Data grids

3 Data clouds and “Big Data”

4 Databases and data-intensive computing

8.1.1 Characterizing data-intensive computations

Data-intensive applications dealS with huge volumes of data, also exhibit compute-intensive properties.

Figure 8.1 identifies the domain of data-intensive computing in the two upper quadrants of the graph.

Data-intensive applications handle datasets on the scale of multiple terabytes and petabytes.

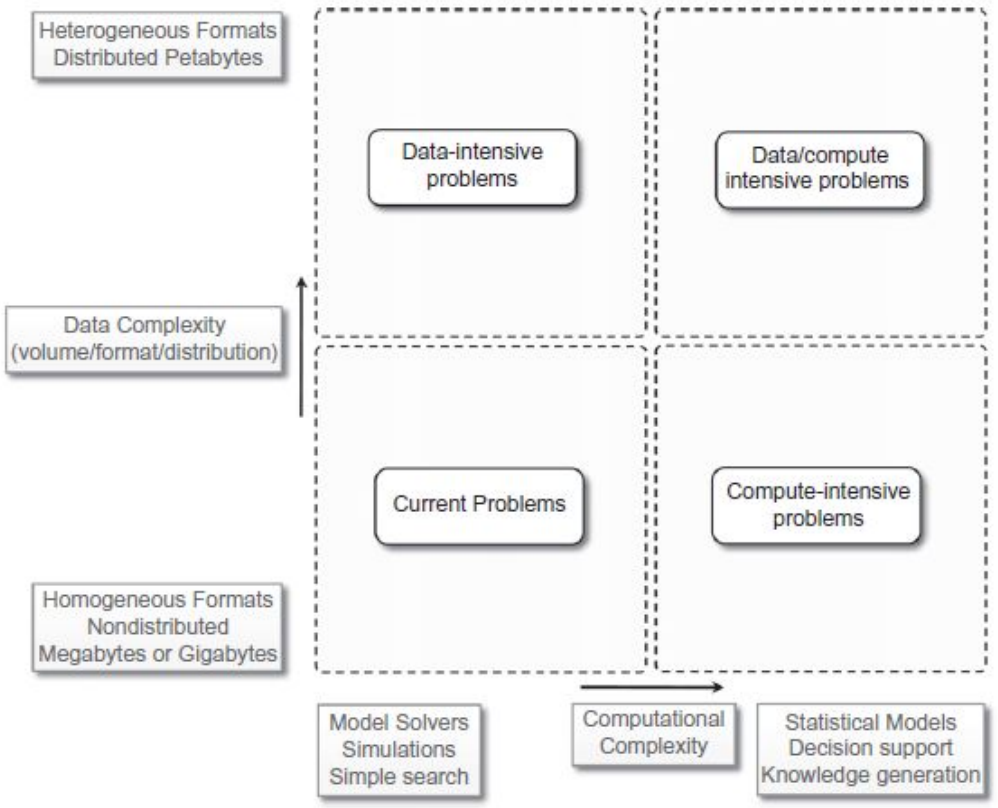


FIGURE 8.1

Data-intensive research issues.

8.1.2 Challenges ahead

The huge amount of data produced, analyzed, or stored imposes requirements on the supporting infrastructures and middleware that are hardly found in the traditional solutions.

Moving terabytes of data becomes an obstacle for high-performing computations.

Data partitioning, **content replication** and **scalable algorithms** help in improving the performance.

Open challenges in data-intensive computing given by Ian Gorton et al. are:

1. **Scalable algorithms** that can search and process massive datasets.
2. New **metadata management technologies** that can handle complex, heterogeneous, and distributed data sources.
3. Advances in **high-performance computing platforms** aimed at providing a better support for accessing in-memory multiterabyte data structures.
4. High-performance, highly reliable, petascale **distributed file systems.**
5. **Data signature-generation** techniques for data reduction and rapid processing.
6. **Software mobility** that are able to move the computation to where the data are located.
7. **Interconnection architectures** that provide better support for filtering multi gigabyte datastreams coming from high-speed networks and scientific instruments.
8. **Software integration** techniques that facilitate the combination of software modules running on different platforms to quickly form analytical pipelines.

8.1.3 Historical perspective

Data-intensive computing involves the production, management, and analysis of large volumes of data.

Support for data-intensive computations is provided by harnessing storage, networking, technologies, algorithms, and infrastructure software all together.

1 The early age: high-speed wide-area networking

In 1989, **the first experiments in high-speed networking** as a support for remote visualization of scientific data led the way.

Two years later, the potential of using high-speed wide area networks for enabling high-speed, TCP/IP-based distributed applications was demonstrated at **Supercomputing 1991 (SC91).**

Kaiser project, leveraged the **Wide Area Large Data Object (WALDO)** system, used to provide following capabilities:

1. automatic generation of metadata;
2. automatic cataloguing of data and metadata processing data in real time;
3. facilitation of cooperative research by providing local and remote users access to data; and
4. mechanisms to incorporate data into databases and other documents.

The **Distributed Parallel Storage System (DPSS)** was developed, later used to support TerraVision, a terrain visualization application that lets users explore and navigate a tridimensional real landscape.

Clipper project, the goal of designing and implementing a collection of independent, architecturally consistent service components to support data-intensive computing. The challenges addressed by Clipper project include management of computing resources, generation or consumption of high-rate and high-volume data flows, human interaction management, and aggregation of resources.

2 Data grids

Huge computational power and storage facilities could be obtained by harnessing heterogeneous resources across different administrative domains.

Data grids emerge as infrastructures that support data-intensive computing.

A data grid provides services that help users discover, transfer, and manipulate large datasets stored in distributed repositories as well as create and manage copies of them.

Data grids offer two main functionalities:

- high-performance and reliable file transfer for moving large amounts of data, and
- scalable replica discovery and management mechanisms.

Data grids mostly provide storage and dataset management facilities as support for scientific experiments that produce huge volumes of data.

Datasets are replicated by infrastructure to provide better availability.

Data grids have their own characteristics and introduce new challenges:

1. **Massive datasets**. The size of datasets can easily be on the scale of gigabytes, terabytes, and beyond. It is therefore necessary to minimize latencies during bulk transfers, replicate content with appropriate strategies, and manage storage resources.
2. **Shared data collections.** Resource sharing includes distributed collections of data. For example, repositories can be used to both store and read data.
3. **Unified namespace.** Data grids impose a unified logical namespace where to locate data collections and resources. Every data element has a single logical name, which is eventually mapped to different physical filenames for the purpose of replication and accessibility.
4. **Access restrictions.** Even though one of the purposes of data grids is to facilitate sharing of results and data for experiments, some users might want to ensure confidentiality for their data and restrict access to them to their collaborators. Authentication and authorization in data grids involve both coarse-grained and fine-grained access control over shared data collections.

As a result, several scientific research fields, including high-energy physics, biology, and astronomy, leverage data grids.

3 Data clouds and "Big Data"

Together with the diffusion of cloud computing technologies that support data-intensive computations, the term **Big Data** has become popular. **Big Data** characterizes the nature of data-intensive computations today and currently identifies datasets that grow so large that they become complex to work with using on-hand database management tools.

In general, the term Big Data applies to datasets of which the size is beyond the ability of commonly used software tools to capture, manage, and process within a tolerable elapsed time. Therefore, Big Data sizes are a constantly moving target, currently ranging from a few dozen tera-bytes to many petabytes of data in a single dataset.

Cloud technologies support data-intensive computing in several ways:

1. By providing a large amount of compute instances on demand, which can be used to process and analyze large datasets in parallel.
2. By providing a storage system optimized for keeping large blobs of data and other distributed data store architectures.
3. By providing frameworks and programming APIs optimized for the processing and management of large amounts of data.

A data cloud is a combination of these components.

Ex 1: MapReduce framework, which provides the best performance for leveraging the Google File System on top of Google's large computing infrastructure.

Ex 2: Hadoop system, the most mature, large, and open-source data cloud. It consists of the Hadoop Distributed File System (HDFS) and Hadoop's implementation of MapReduce.

Ex 3: Sector, consists of the Sector Distributed File System (SDFS) and a compute service called Sphere that allows users to execute arbitrary user-defined functions (UDFs) over the data managed by SDFS.

Ex 4: Greenplum uses a shared-nothing massively parallel processing (MPP) architecture based on commodity hardware.

4 Databases and data-intensive computing

Distributed databases are a collection of data stored at different sites of a computer network. Each site might expose a degree of autonomy, providing services for the execution of local applications, but also participating in the execution of a global application.

A distributed database can be created by splitting and scattering the data of an existing database over different sites or by federating together multiple existing databases. These systems are very robust and provide distributed transaction processing, distributed query optimization, and efficient management of resources.

8.2 Technologies for data-intensive computing

Data-intensive computing concerns the development of applications that are mainly focused on processing large quantities of data.

Therefore, storage systems and programming models constitute a natural classification of the technologies supporting data-intensive computing.

8.2.1 Storage systems

1. High-performance distributed file systems and storage clouds
2. NoSQL systems

8.2.2 Programming platforms

1. The MapReduce programming model.
2. Variations and extensions of MapReduce.
3. Alternatives to MapReduce.

8.2.1 Storage systems

Traditionally, database management systems constituted the de facto storage.

Due to the explosion of unstructured data in the form of blogs, Web pages, software logs, and sensor readings, the relational model in its original formulation does not seem to be the preferred solution for supporting data analytics on a large scale.

Some factors contributing to change in database are:

A. **Growing of popularity of Big Data.** The management of large quantities of data is no longer a rare case but instead has become common in several fields: scientific computing, enterprise applications, media entertainment, natural language processing, and social network analysis.
B. **Growing importance of data analytics in the business chain.** The management of data is no longer considered a cost but a key element of business profit. This situation arises in popular social networks such as Facebook, which concentrate their focus on the management of user profiles, interests, and connections among people.
C. **Presence of data in several forms, not only structured.** As previously mentioned, what constitutes relevant information today exhibits a heterogeneous nature and appears in several forms and formats.
D. **New approaches and technologies for computing.** Cloud computing promises access to a massive amount of computing capacity on demand. This allows engineers to design software systems that incrementally scale to arbitrary degrees of parallelism.

1. High-performance distributed file systems and storage clouds

Distributed file systems constitute the primary support for data management. They provide an interface whereby to store information in the form of files and later access them for read and write operations.

a. Lustre. The Lustre file system is a massively parallel distributed file system that covers the needs of a small workgroup of clusters to a large-scale computing cluster. The file system is used by several of the Top 500 supercomputing systems.

Lustre is designed to provide access to petabytes (PBs) of storage to serve thousands of clients with an I/O throughput of hundreds of gigabytes per second (GB/s). The system is composed of a metadata server that contains the metadata about the file system and a collection of object storage servers that are in charge of providing storage.

b. IBM General Parallel File System (GPFS). GPFS is the high-performance distributed file system developed by IBM that provides support for the RS/6000 supercomputer and Linux computing clusters. GPFS is a multiplatform distributed file system built over several years of academic research and provides advanced recovery mechanisms. GPFS is built on the concept of shared disks, in which a collection of disks is attached to the file system nodes by means of some switching fabric. The file system makes this infrastructure transparent to users and stripes large files over the disk array by replicating portions of the file to ensure high availability.

c. Google File System (GFS). GFS is the storage infrastructure that supports the execution of distributed applications in Google's computing cloud.

GFS is designed with the following assumptions:

1. The system is built on top of commodity hardware that often fails.
2. The system stores a modest number of large files; multi-GB files are common and should be treated efficiently, and small files must be supported, but there is no need to optimize for that.
3. The workloads primarily consist of two kinds of reads: large streaming reads and small random reads.
4. The workloads also have many large, sequential writes that append data to files.
5. High-sustained bandwidth is more important than low latency.

The architecture of the file system is organized into a single master, which contains the metadata of the entire file system, and a collection of chunk servers, which provide storage space. From a logical point of view the system is composed of a collection of software daemons, which implement either the master server or the chunk server.

d. Sector. Sector is the storage cloud that supports the execution of data-intensive applications defined according to the Sphere framework. It is a user space file system that can be deployed on commodity hardware across a wide-area network. The system's architecture is composed of four nodes: a security server, one or more master nodes, slave nodes, and client machines. The security server maintains all the information about access control policies for user and files, whereas master servers coordinate and serve the I/O requests of clients, which ultimately interact with slave nodes to access files. The protocol used to exchange data with slave nodes is UDT, which is a lightweight connection-oriented protocol.

e. Amazon Simple Storage Service (S3). Amazon S3 is the online storage service provided by Amazon.

The system offers a flat storage space organized into buckets, which are attached to an Amazon Web Services (AWS) account. Each bucket can store multiple objects, each identified by a unique key. Objects are identified by unique URLs and exposed through HTTP, thus allowing very simple get-put semantics.

2. NoSQL systems

The term **Not Only SQL (NoSQL)** was coined in 1998 to identify a set of UNIX shell scripts and commands to operate on text files containing the actual data.

NoSQL cannot be considered a relational database, it is a collection of scripts that allow users to manage most of the simplest and more common database tasks by using text files as information stores.

Two main factors have determined the growth of the NoSQL:

1. simple data models are enough to represent the information used by applications, and
2. the quantity of information contained in unstructured formats has grown.

Let us now examine some prominent implementations that support data-intensive applications.

a. Apache CouchDB and MongoDB.

Apache CouchDB and MongoDB are two examples of document stores. Both provide a schema-less store whereby the primary objects are documents organized into a collection of key-value fields. The value of each field can be of type string, integer, float, date, or an array of values.

The databases expose a RESTful interface and represent data in JSON format. Both allow querying and indexing data by using the MapReduce programming model, expose JavaScript as a base language for data querying and manipulation rather than SQL, and support large files as documents.

b. Amazon Dynamo.

The main goal of Dynamo is to provide an incrementally scalable and highly available storage system. This goal helps in achieving reliability at a massive scale, where thousands of servers and network components build an infrastructure serving 10 million requests per day. Dynamo provides a simplified interface based on get/put semantics, where objects are stored and retrieved with a unique identifier (key).

The architecture of the Dynamo system, shown in **Figure 8.3**, is composed of a collection of storage peers organized in a ring that shares the key space for a given application. The key space is partitioned among the storage peers, and the keys are replicated across the ring, avoiding adjacent peers. Each peer is configured with access to a local storage facility where original objects and replicas are stored.

Each node provides facilities for distributing the updates among the rings and to detect failures and unreachable nodes.

c. Google Bigtable.

Bigtable provides storage support for several Google applications that expose different types of workload: from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users.

Bigtable's key design goals are wide applicability, scalability, high performance, and high availability. To achieve these goals, Bigtable organizes the data storage in tables of which the rows are distributed over the distributed file system supporting the middleware, which is the Google File System.

From a logical point of view, a table is a multidimensional sorted map indexed by a key that is represented by a string of arbitrary length. A table is organized into rows and columns; columns can be grouped in column family, which allow for specific optimization for better access control, the storage and the indexing of data.
Bigtable APIs also allow more complex operations such as single row transactions and advanced data manipulation.

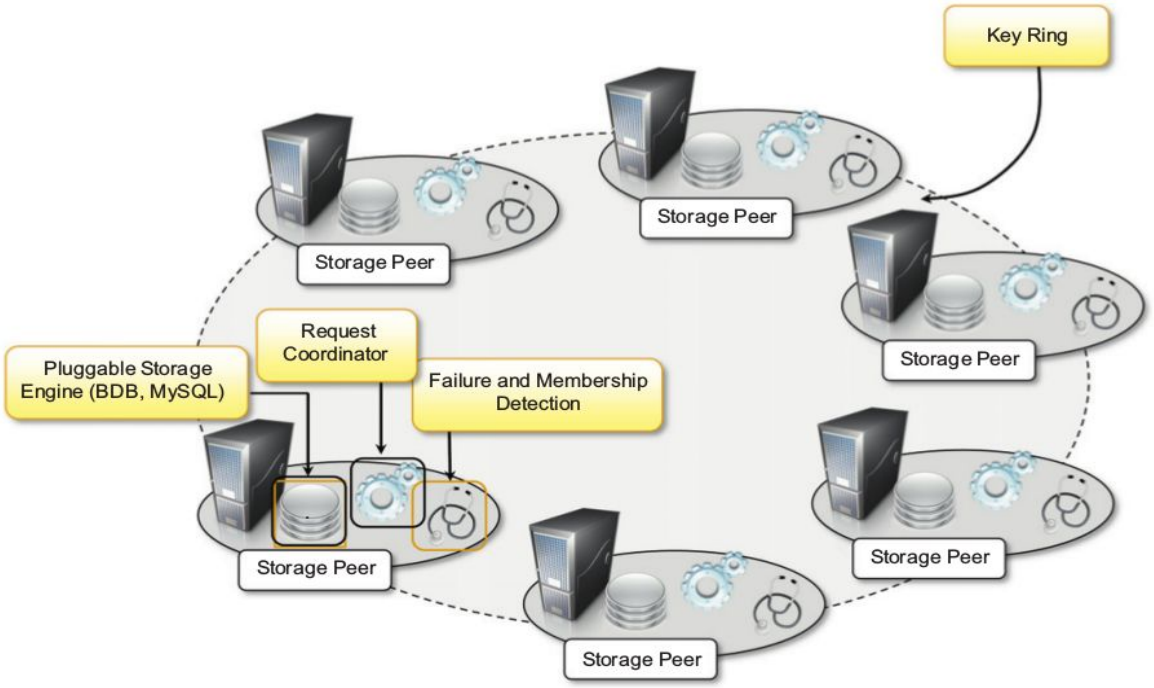


FIGURE 8.3
Amazon Dynamo architecture.

Figure 8.4 gives an overview of the infrastructure that enables Bigtable.
The service is the result of a collection of processes that coexist with other processes in a cluster-based environment. Bigtable identifies two kinds of processes: master processes and tablet server processes. A tablet server is responsible for serving the requests for a given tablet that is a contiguous partition of rows of a table. Each server can manage multiple tablets (commonly from 10 to 1,000). The master server is responsible for keeping track of the status of the tablet servers and of the allocation of tablets to tablet servers. The server constantly monitors the tablet servers to check whether they are alive, and in case they are not reachable, the allocated tablets are reassigned and eventually partitioned to other servers.

d. Apache Cassandra.

The system is designed to avoid a single point of failure and offer a highly reliable service. Cassandra was initially developed by Facebook; now it is part of the Apache incubator initiative. Currently, it provides storage support for several very large Web applications such as Facebook itself, Digg, and Twitter.
The data model exposed by Cassandra is based on the concept of a table that is implemented as a distributed multidimensional map indexed by a key. The value corresponding to a key is a highly structured object and constitutes the row of a table. Cassandra organizes the row of a table into columns, and sets of columns can be grouped into column families. The APIs provided by the system to access and manipulate the data are very simple: insertion, retrieval, and deletion. The insertion is performed at the row level; retrieval and deletion can operate at the column level.

e. Hadoop HBase.

HBase is designed by taking inspiration from Google Bigtable; its main goal is to offer real-time read/write operations for tables with billions of rows and millions of columns by leveraging clusters of commodity hardware. The internal architecture and logic model of HBase is very similar to Google Bigtable, and the entire system is backed by the Hadoop Distributed File System (HDFS).

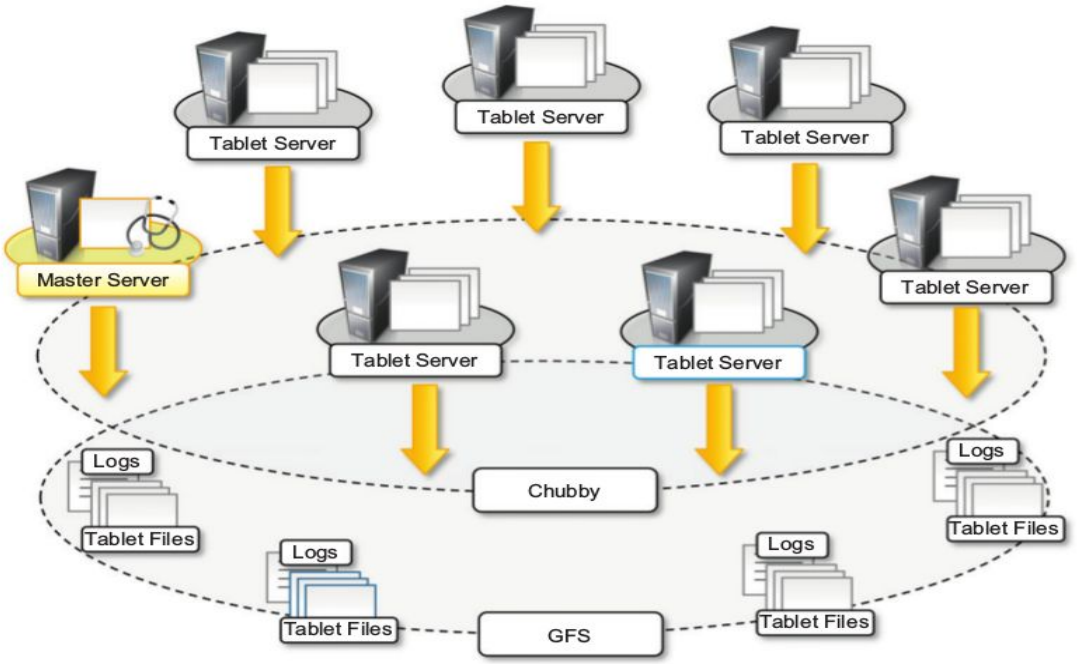


FIGURE 8.4
Bigtable architecture.

8.2.2 Programming platforms

Programming platforms for data-intensive computing provide higher-level abstractions, which focus on the processing of data and move into the runtime system the management of transfers, thus making the data always available where needed.

This is the approach followed by the MapReduce programming platform, which expresses the computation in the form of two simple functions—map and reduce—and hides the complexities of managing large and numerous data files into the distributed file system supporting the platform. In this section, we discuss the characteristics of MapReduce and present some variations of it.

1. The MapReduce programming model.

MapReduce expresses the computational logic of an application in two simple functions: map and reduce.

Data transfer and management are completely handled by the distributed storage infrastructure (i.e., the Google File System), which is in charge of providing access to data, replicating files, and eventually moving them where needed.

the MapReduce model is expressed in the form of the two functions, which are defined as follows:

$$map\ (k1, v1) \rightarrow list(k2, v2)$$
$$reduce(k2, list(v2)) \rightarrow list(v2)$$

The map function reads a key-value pair and produces a list of key-value pairs of different types. The reduce function reads a pair composed of a key and a list of values and produces a list of values of the same type. The types (k1,v1,k2,kv2) used in the expression of the two functions provide hints as to how these two functions are connected and are executed to carry out the computation of a MapReduce job: The output of map tasks is aggregated together by grouping the values according to their corresponding keys and constitutes the input of reduce tasks that, for each of the keys found, reduces the list of attached values to a single value. Therefore, the input of a MapReduce computation is expressed as a collection of key-value pairs < k1,v1 >, and the final output is represented by a list of values: list(v2).

Figure 8.5 depicts a reference workflow characterizing MapReduce computations. As shown, the user submits a collection of files that are expressed in the form of a list of < k1,v1 > pairs and specifies the map and reduce functions. These files are entered into the distributed file system that supports MapReduce and, if necessary, partitioned in order to be the input of map tasks. Map tasks generate intermediate files that store collections of

< k2, list(v2) > pairs, and these files are saved into the distributed file system. These files constitute the input of reduce tasks, which finally produce output files in the form of list(v2).

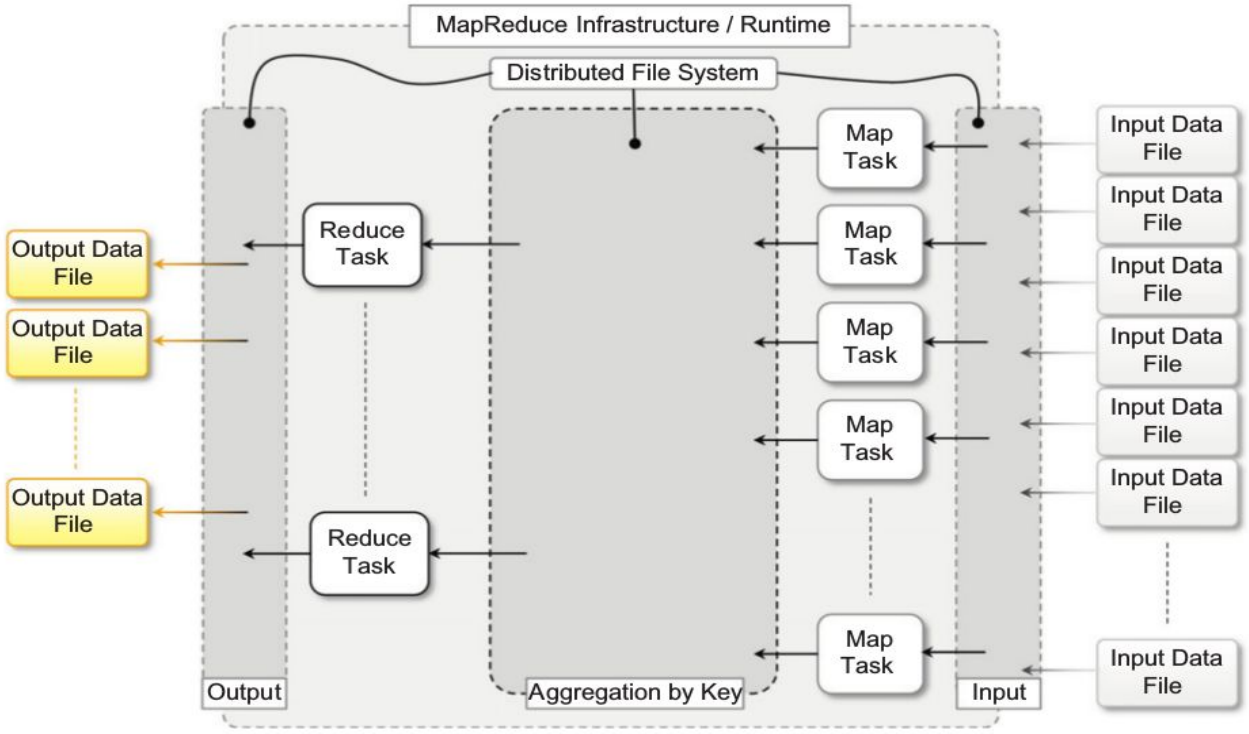


FIGURE 8.5

MapReduce computation workflow.

The computation model expressed by MapReduce is very straightforward and allows greater productivity for people who have to code the algorithms for processing huge quantities of data.

In general, any computation that can be expressed in the form of two major stages can be represented in terms of MapReduce computation.

These stages are:

1. Analysis. This phase operates directly on the data input file and corresponds to the operation performed by the map task. Moreover, the computation at this stage is expected to be embarrassingly parallel, since map tasks are executed without any sequencing or ordering.

2. Aggregation. This phase operates on the intermediate results and is characterized by operations that are aimed at aggregating, summing, and/or elaborating the data obtained at the previous stage to present the data in their final form. This is the task performed by the reduce function.

Figure 8.6 gives a more complete overview of a MapReduce infrastructure, according to the implementation proposed by Google.

As depicted, the user submits the execution of MapReduce jobs by using the client libraries that are in charge of submitting the input data files, registering the map and reduce functions, and returning control to the user once the job is completed. A generic distributed infrastructure (i.e., a cluster) equipped with job-scheduling capabilities and distributed storage can be used to run MapReduce applications.

Two different kinds of processes are run on the distributed infrastructure:

a master process and

a worker process.

The master process is in charge of controlling the execution of map and reduce tasks, partitioning, and reorganizing the intermediate output produced by the map task in order to feed the reduce tasks.

The master process generates the map tasks and assigns input splits to each of them by balancing the load.

The worker processes are used to host the execution of map and reduce tasks and provide basic I/O facilities that are used to interface the map and reduce tasks with input and output files.

Worker processes have input and output buffers that are used to optimize the performance of map and reduce tasks. In particular, output buffers for map tasks are periodically dumped to disk to create intermediate files. Intermediate files are partitioned using a user-defined function to evenly split the output of map tasks.

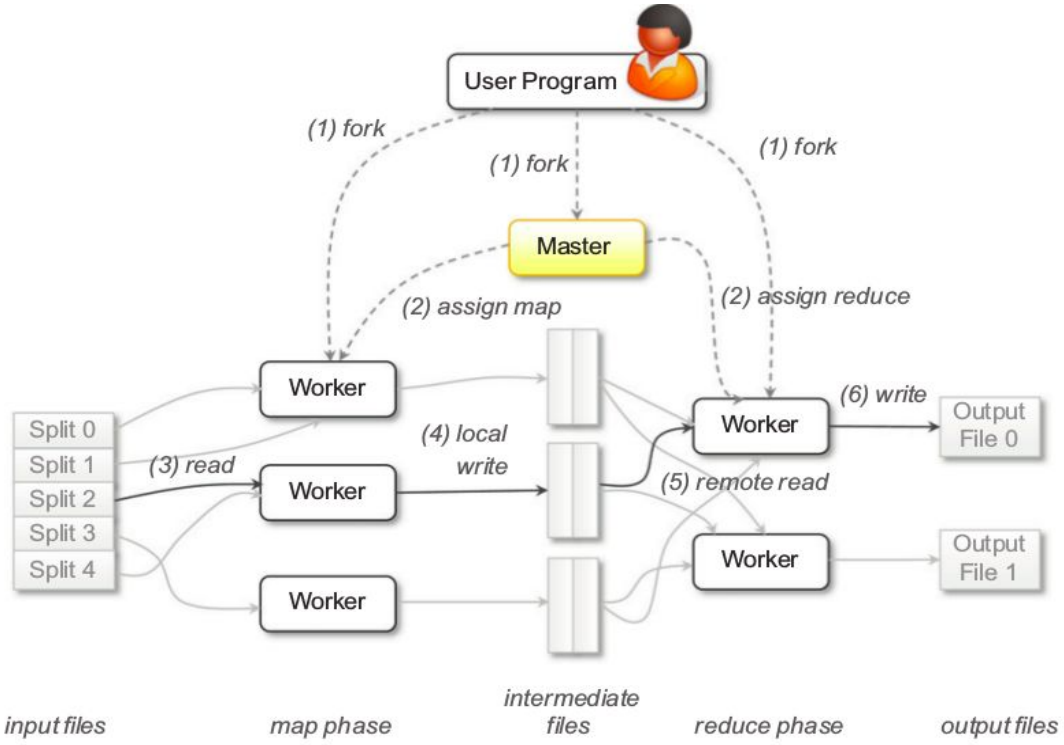


FIGURE 8.6

Google MapReduce infrastructure overview.

2. Variations and extensions of MapReduce.

MapReduce constitutes a simplified model for processing large quantities of data and imposes constraints on the way distributed algorithms should be organized to run over a MapReduce infrastructure.

Therefore, a series of extensions to and variations of the original MapReduce model have been proposed. They aim at extending the MapReduce application space and providing developers with an easier interface for designing distributed algorithms.

We briefly present a collection of MapReduce-like frameworks and discuss how they differ from the original MapReduce model.

A. Hadoop.
B. Pig.
C. Hive.
D. Map-Reduce-Merge.
E. Twister.

A. Hadoop.

Apache Hadoop is a collection of software projects for reliable and scalable distributed computing. The initiative consists of mostly two projects: Hadoop Distributed File System (HDFS) and Hadoop MapReduce. The former is an implementation of the Google File System; the latter provides the same features and abstractions as Google MapReduce.

B. Pig.

Pig is a platform that allows the analysis of large datasets. Developed as an Apache project, Pig consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The Pig infrastructure's layer consists of a compiler for a high-level language that produces a sequence of MapReduce jobs that can be run on top of distributed infrastructures.

C. Hive.

Hive is another Apache initiative that provides a data warehouse infrastructure on top of Hadoop MapReduce. It provides tools for easy data summarization, ad hoc queries, and analysis of large datasets stored in Hadoop MapReduce files.

Hive's major advantages reside in the ability to scale out, since it is based on the Hadoop framework, and in the ability to provide a data warehouse infrastructure in environments where there is already a Hadoop system running.

D. Map-Reduce-Merge.

Map-Reduce-Merge is an extension of the MapReduce model, introducing a third phase to the standard MapReduce pipeline—the Merge phase—that allows efficiently merging data already partitioned and sorted (or hashed) by map and reduce modules. The Map-Reduce-Merge framework simplifies the management of heterogeneous related datasets and provides an abstraction able to express the common relational algebra operators as well as several join algorithms.

E. Twister.

Twister is an extension of the MapReduce model that allows the creation of iterative executions of MapReduce jobs. With respect to the normal MapReduce pipeline, the model proposed by Twister proposes the following extensions:

1. Configure Map
2. Configure Reduce
3. While Condition Holds True Do
    a. Run MapReduce
    b. Apply Combine Operation to Result
    c. Update Condition
4. Close

Twister provides additional features such as the ability for map and reduce tasks to refer to static and in-memory data; the introduction of an additional phase called combine, run at the end of the MapReduce job, that aggregates the output together.

3. Alternatives to MapReduce.

a. Sphere.

b. All-Pairs.

c. DryadLINQ.

a. Sphere.

Sphere is the distributed processing engine that leverages the **Sector Distributed File System (SDFS)**.

Sphere implements the **stream processing model** (**Single Program, Multiple Data**) and allows developers to express the computation in terms of **user-defined functions (UDFs)**, which are run against the distributed infrastructure.

Sphere is built on top of Sector's API for data access.

UDFs are expressed in terms of programs that read and write streams. A stream is a data structure that provides access to a collection of data segments mapping one or more files in the SDFS.

The execution of UDFs is achieved through **Sphere Process Engines (SPEs)**, which are assigned with a given stream segment.

Sphere **client** sends a request for processing to the **master** node, which returns the list of available slaves, and the client will choose the slaves on which to execute Sphere processes.

b. All-Pairs.

It provides a simple abstraction—in terms of the All-pairs function—that is common in many scientific computing domains:

All-pairs(A:set; B:set; F:function) -> M:matrix

Ex 1: field of biometrics, where similarity matrices are composed as a result of the comparison of several images that contain subject pictures.

Ex 2: applications and algorithms in data mining.

The All-pairs function can be easily solved by the following algorithm:

1. For each $i in A
2. For each $j in B
3. Submit job F $i $j

The execution of a distributed application is controlled by the engine and develops in four stages:

(1) model the system;

(2) distribute the data;

(3) dispatch batch jobs; and

(4) clean up the system.

c. DryadLINQ.

Dryad is a Microsoft Research project that investigates programming models for writing parallel and distributed programs to scale from a small cluster to a large datacenter.

In Dryad, developers can express distributed applications as a set of sequential programs that are connected by means of channels.

Dryad computation expressed in terms of a directed acyclic graph in which nodes are the sequential programs and vertices represent the channels connecting such programs.

Dryad is considered a superset of the MapReduce model, its application model allows expressing graphs representing MapReduce computation.

8.3 Aneka MapReduce programming

Aneka provides an implementation of the MapReduce abstractions introduced by Google and implemented by Hadoop.

8.3.1 Introducing the MapReduce programming model

1 Programming abstractions
2 Runtime support
3 Distributed file system support

8.3.2 Example application

1 Parsing Aneka logs
2 Mapper design and implementation
3 Reducer design and implementation
4 Driver program
5 Running the application

8.3.1 Introducing the MapReduce programming model

The MapReduce Programming Model defines the abstractions and runtime support for developing MapReduce applications on top of Aneka.

Figure 8.7 provides an overview of the infrastructure supporting MapReduce in Aneka.

The application instance is specialized, with components that identify the map and reduce functions to use.

These functions are expressed in terms of **Mapper and Reducer classes** that are extended from the Aneka MapReduce APIs.

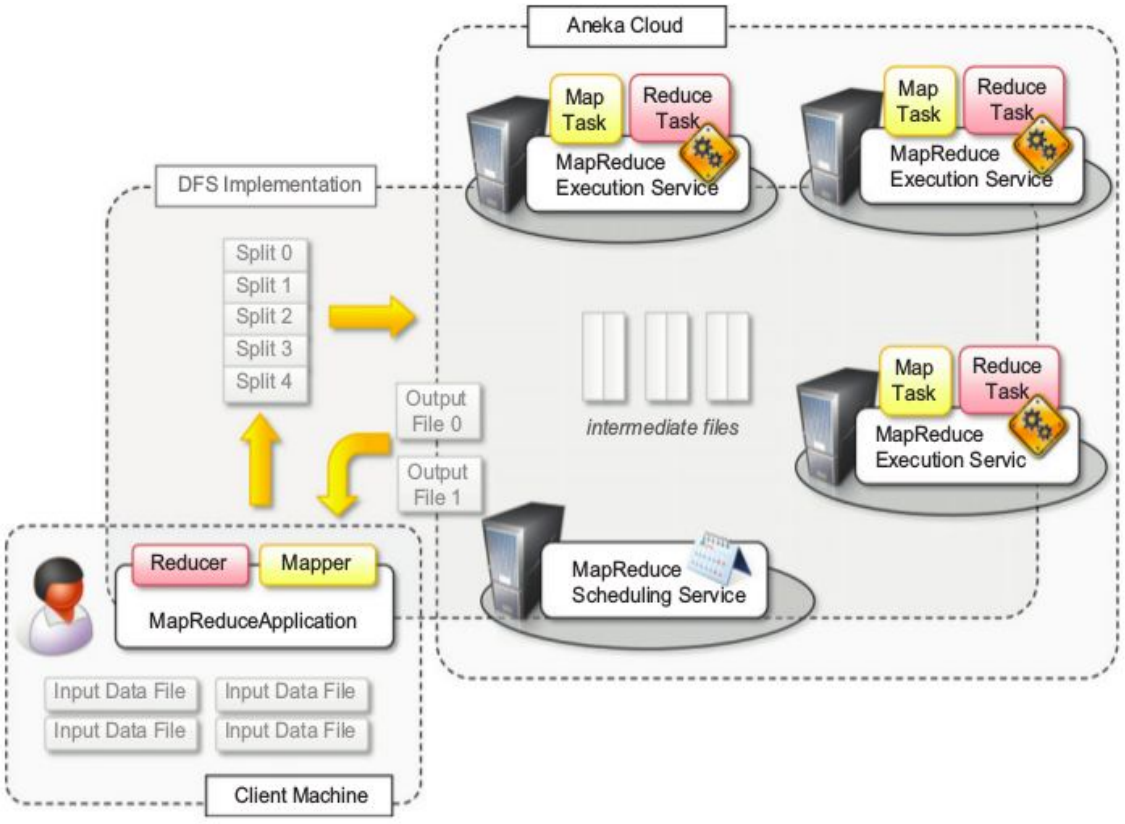


FIGURE 8.7

Aneka MapReduce infrastructure.

The runtime support is composed of **three** main elements:

1. **MapReduce Scheduling Service**, which plays the role of the master process in the Google and Hadoop implementation.
2. **MapReduce Execution Service**, which plays the role of the worker process in the Google and Hadoop implementation.
3. A specialized **distributed file system** that is used to move data files.

Client components, namely the MapReduce Application, are used to submit the execution of a MapReduce job, upload data files, and monitor it.

The management of data files is transparent: local data files are automatically uploaded to Aneka, and output files are automatically downloaded to the client machine if requested.

In the following sections, we introduce these major components and describe how they collaborate to execute MapReduce jobs.

1 Programming abstractions

Aneka executes any piece of user code within distributed application.

The task creation is responsibility of the infrastructure once the user has defined the map and reduce functions.

Therefore, the Aneka MapReduce APIs provide developers with base classes for developing Mapper and Reducer types and use a specialized type of application class—Map Reduce Application — that supports needs of this programming model.

Figure 8.8 provides an overview of the client components defining the MapReduce programming model. Three classes are of interest for application development: Mapper<K,V>, Reducer<K,V>, and MapReduceApplication<M,R>. The other classes are internally used to implement all the functionalities

required by the model and expose simple interfaces that require minimum amounts of coding for implementing the map and reduce functions and controlling the job submission. Mapper<K,V> and Reducer<K,V> constitute the starting point of the application design and implementation.

The submission and execution of MapReduce job is performed through class MapReduceApplication<M,R>, which provides the interface to Aneka Cloud to support MapReduce programming model. This class exposes two generic types: M and R. These two placeholders identify the specific types of Mapper<K,V> and Reducer<K,V> that will be used by the application.

Listing 8.1 shows in detail the definition of the Mapper<K,V> class and of the related types that developers should be aware of for implementing the map function.

Listing 8.2 shows the implementation of the Mapper<K,V> component for Word Counter sample. This sample counts frequency of words in a set of large text files. The text files are divided into lines, each of which will become the value component of a key-value pair, whereas the key will be represented by the offset in the file where the line begins.

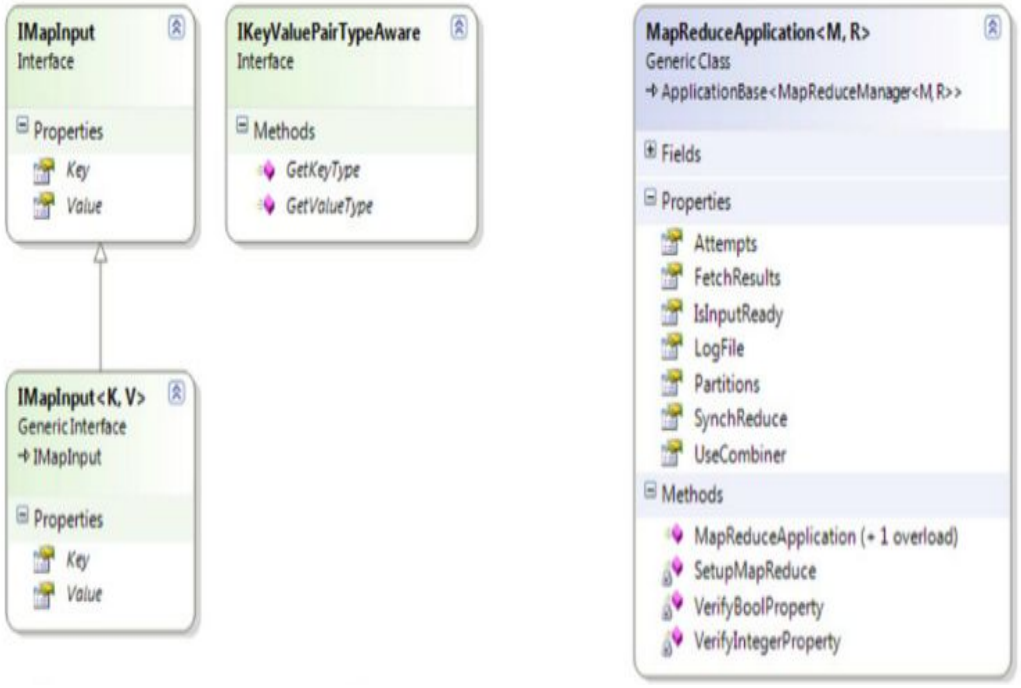
Listing 8.3 shows the definition of Reducer<K,V> class. The implementation of a specific reducer requires specializing the generic class and overriding the abstract method: Reduce(IReduceInputEnumerator<V> input).

Listing 8.4 shows how to implement the reducer function for word-counter example.

Listing 8.5 shows the interface of MapReduceApplication<M,R>.

Listing 8.6 displays collection of methods that are of interest in this class for execution of MapReduce jobs.

Listing 8.7 shows how to create a MapReduce application for running the word-counter example defined by the previous WordCounterMapper and WordCounterReducer classes.



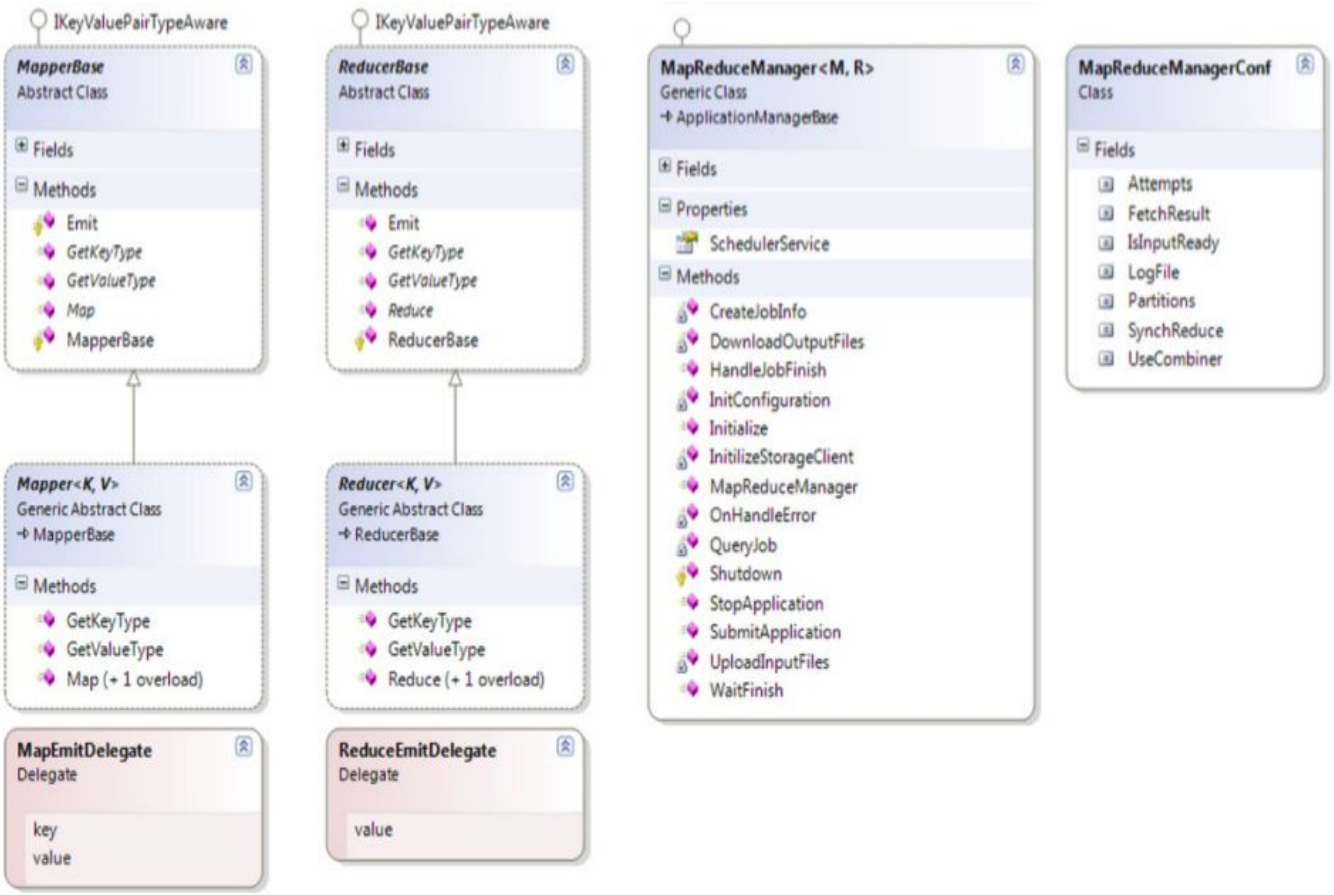


FIGURE 8.8

MapReduce Abstractions Object Model.

```
using Aneka.MapReduce.Internal;
namespace Aneka.MapReduce
{
        /// Interface IMapInput<K,V>. Extends IMapInput and provides strongly-typed version of extended
        /// interface.
        public interface IMapInput<K,V>: IMapInput
        {
                /// Property <i>Key</i> returns the key of key/value pair.
                K Key { get; }
                /// Property <i>Value</i> returns the value of key/value pair.
                V Value { get; }
        }
        /// Delegate MapEmitDelegate. Defines signature of method that is used to doEmit intermediate results
        /// generated by mapper.
        public delegate void MapEmitDelegate(object key, object value);
        /// Class Mapper. Extends MapperBase and provides a reference implementation that can be further
        /// extended in order to define the specific mapper for a given application.
        public abstract class Mapper<K,V> : MapperBase
        {
                /// Emits the intermediate result source by using doEmit.
                /// output stream the information about the output of the Map operation.</para
                public void Map(IMapInput input, MapEmitDelegate emit) { ... }
```

```
            /// Gets the type of the <i>key</i> component of a <i>key-value</i> pair.
            /// <returns>A Type instance containing the metadata
            public override Type GetKeyType(){ return typeof(K); }
            /// Gets the type of the <i>value</i> component of a <i>key-value</i> pair.
            /// <returns>A Type instance containing the metadata
            public overrideType GetValueType(){ return typeof(V); }
            #region Template Methods
            /// Function Map is overrided by users to define a map function.
            protected abstract void Map(IMapInput<K, V> input);
            #endregion
        }
}
```

LISTING 8.1 Map Function APIs.

```
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.WordCounter
{
        /// Class WordCounterMapper. Extends Mapper<K,V> and provides an
        /// implementation of the map function for the Word Counter sample.
        public class WordCounterMapper: Mapper<long,string>
        {
                /// Reads the source and splits into words. For each of the words found
                /// emits the word as a key with a vaue of 1.
                protected override void Map(IMapInput<long,string> input)
                {
                        // we don't care about the key, because we are only interested on
                        // counting the word of each line.
                        string value = input.Value;
                        string[] words = value.Split(" \t\n\r\f\"\'|!-=()[]<>:{}.#".ToCharArray(),
                        StringSplitOptions.RemoveEmptyEntries);
                        // we emit each word without checking for repetitions. The word becomes
                        // the key and the value is set to 1, the reduce operation will take care
                        // of merging occurrences of the same word and summing them.
                        foreach(string word in words)
                        {
                                this.Emit(word, 1);
                        }
                }
        }
}
```

LISTING 8.2 Simple Mapper <K,V> Implementation.

```
using Aneka.MapReduce.Internal;
namespace Aneka.MapReduce
{
        /// Delegate ReduceEmitDelegate. Defines the signature of a method
```

```
        public delegate void ReduceEmitDelegate(object value);
        /// Class <i>Reducer</i>. Extends the ReducerBase class
        public abstract class Reducer<K,V> : ReducerBase
        {
                /// Performs the <i>reduce</i> phase of the <i>map-reduce</i> model.
                public void Reduce(IReduceInputEnumerator input, ReduceEmitDelegate emit) { ... }
                /// Gets the type of the <i>key</i> component of a <i>key-value</i> pair.
                public override Type GetKeyType(){return typeof(K);}
                /// Gets the type of the <i>value</i> component of a <i>key-value</i> pair.
                public override Type GetValueType(){return typeof(V);}
                #region Template Methods
                /// Recuces the collection of values that are exposed by
                /// <paramref name="source"/> into a single value.
                protected abstract void Reduce(IReduceInputEnumerator<V> input);
                #endregion
        }
}
```

LISTING 8.3 Reduce Function APIs.

```
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.WordCounter
{
        /// Class <b><i>WordCounterReducer</i></b>. Reducer implementation for the Word
        /// Counter application.
        public class WordCounterReducer: Reducer<string,int>
        {
                /// Iterates all over the values of the enumerator and sums up
                /// all the values before emitting the sum to the output file.
                protected override void Reduce(IReduceInputEnumerator<int>input)
                {
                        int sum = 0;
                        while(input.MoveNext())
                        {
                                int value = input.Current;
                                sum += value;
                        }
                        this.Emit(sum);
                }
        }
}
```

LISTING 8.4 Simple Reducer <K,V> Implementation.

```
using Aneka.MapReduce.Internal;
namespace Aneka.MapReduce
{
        /// Class <b><i>MapReduceApplication</i></b>. Defines a distributed application
        /// based on the MapReduce Model. It extends the ApplicationBase<M> and specializes
        /// it with the MapReduceManager<M,R> application manager.
        public class MapReduceApplication<M, R> : ApplicationBase<MapReduceManager<M, R>>
        where M: MapReduce.Internal.MapperBase
```

```
        where R: MapReduce.Internal.ReducerBase
        {
                /// Default value for the Attempts property.
                public const intDefaultRetry = 3;
                /// Default value for the Partitions property.
                public const intDefaultPartitions = 10;
                /// Default value for the LogFile property.
                public const stringDefaultLogFile = "mapreduce.log";
                /// List containing the result files identifiers.
                private List<string>resultFiles = new List<string>();
                /// Property group containing the settings for the MapReduce application.
                private PropertyGroupmapReduceSetup;
                /// Gets, sets an integer representing the number of partions for the key space.
                public int Partitions { get { ... } set { ... } }
                /// Gets, sets an boolean value indicating in whether to combine the result
                /// after the map phase in order to decrease the number of reducers used in the
                /// reduce phase.
                public bool UseCombiner { get { ... } set { ... } }
                /// Gets, sets an boolean indicating whether to synchronize the reduce phase.
                public bool SynchReduce { get { ... } set { ... } }
                /// Gets or sets a boolean indicating whether the source files required by the
                /// required by the application is already uploaded in the storage or not.
                public bool IsInputReady { get { ... } set { ... } }
                /// Gets, sets the number of attempts that to run failed tasks.
                public int Attempts { get { ... } set { ... } }
                /// Gets or sets a string value containing the path for the log file.
                public string LogFile { get { ... } set { ... } }
                /// Gets or sets a boolean indicating whether application should download the
                /// result files on the local client machine at the end of the execution or not.
                public bool FetchResults { get { ... } set { ... } }
                /// Creates a MapReduceApplication<M,R> instance and configures it with
                /// the given configuration.
                public MapReduceApplication(Configurationconfiguration) :
                base("MapReduceApplication", configuration){ ... }
                /// Creates MapReduceApplication<M,R> instance and configures it with
                /// the given configuration.
                public MapReduceApplication(string displayName, Configuration configuration) : base(displayName,
                configuration) { ... }
                // here follows the private implementation…
        }
}
```

LISTING 8.5 MapReduceApplication<M,R>.

```
namespace Aneka.Entity
{
        /// Class <b><i>ApplicationBase<M></i></b>. Defines the base class for the
        /// application instances for all the programming model supported by Aneka.
        public class ApplicationBase<M> where M : IApplicationManager, new()
        {
                /// Gets the application unique identifier attached to this instance.
                public string Id { get { ... } }
                /// Gets the unique home directory for the AnekaApplication<W,M>.
```

```
                public string Home { get { ... } }
                /// Gets the current state of the application.
                public ApplicationState State{get{ ... }}
                /// Gets a boolean value indicating whether the application is terminated.
                public bool Finished { get { ... } }
                /// Gets the underlying IApplicationManager that ismanaging the execution of the
                /// application instanceon the client side.
                public M ApplicationManager { get { ... } }
                /// Gets, sets the application display name.
                public string DisplayName { get { ... } set { ... } }
                /// Occurs when the application instance terminates its execution.
                public event EventHandler<ApplicationEventArgs> ApplicationFinished;
                /// Creates an application instance with the given settings and sets the
                /// application display name to null.
                public ApplicationBase(Configuration configuration): this(null, configuration){ ... }
                /// Creates an application instance withthe given settings and display name.
                public ApplicationBase(string displayName, Configuration configuration){ ... }
                /// Starts the execution of the application instanceon Aneka.
                public void SubmitExecution() { ... }
                /// Stops the execution of the entire application instance.
                public void StopExecution() { ... }
                /// Invoke the application and wait until the application finishes.
                public void InvokeAndWait() { this.InvokeAndWait(null); }
                /// Invoke the application and wait until the application finishes, then invokes
                /// the given callback.
                public void InvokeAndWait(EventHandler<ApplicationEventArgs> handler) { ... }
                /// Adds a shared file to the application.
                public virtual void AddSharedFile(string file) { ... }
                /// Adds a shared file to the application.
                public virtual void AddSharedFile(FileData fileData) { ... }
                /// Removes a file from the list of the shared files of the application.
                public virtual void RemoveSharedFile(string filePath) { ... }
                // here come the private implementation.
        }
}
```

LISTING 8.6 ApplicationBase<M>

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.WordCounter
{
        /// Class <b><i>Program<M></i></b>. Application driver for the Word Counter sample.
        public class Program
        {
                /// Reference to the configuration object.
                private static Configuration configuration = null;
                /// Location of the configuration file.
                private static string confPath = "conf.xml";
                /// Processes arguments given to application & read runs application or shows help.
                private static void Main(string[] args)
                {
```

```
                try
                {
                        Logger.Start();
                        // get the configuration
                        configuration = Configuration.GetConfiguration(confPath);
                        // configure MapReduceApplication
                        MapReduceApplication<WordCountMapper, WordCountReducer> application =
                        new MapReduceApplication<WordCountMapper, WordCountReducer>
                ("WordCounter",
                        configuration);
                        // invoke and wait for result
                        application.InvokeAndWait(new
                        EventHandler<ApplicationEventArgs>(OnDone));
                }
                catch(Exception ex)
                {
                        Usage();
                        IOUtil.DumpErrorReport(ex, "Aneka WordCounter Demo - Error Log");
                }
                finally
                {
                        Logger.Stop();
                }
        }
        /// Hooks the ApplicationFinished events and Process the results if the application has been successful.
        private static void OnDone(object sender, ApplicationEventArgs e) { ... }
        /// Displays a simple informative message explaining the usage of the application.
        private static void Usage() { ... }
    }
}
```

LISTING 8.7 WordCounter Job.

2 Runtime support

The runtime support for the execution of MapReduce jobs comprises the collection of services that deal with scheduling and executing MapReduce tasks.

These are the MapReduce Scheduling Service and the MapReduce Execution Service.

Job and Task Scheduling. The scheduling of jobs and tasks is the responsibility of the MapReduce Scheduling Service, which covers the same role as the master process in the Google MapReduce implementation. The architecture of the Scheduling Service is organized into two major components: the MapReduceSchedulerService and the MapReduceScheduler.

Main role of the service wrapper is to translate messages coming from the Aneka runtime or the client applications into calls or events directed to the scheduler component, and vice versa. The relationship of the two components is depicted in **Figure 8.9**.

The core functionalities for job and task scheduling are implemented in the MapReduceScheduler class. The scheduler manages multiple queues for several operations, such as uploading input files into the distributed file system; initializing jobs before scheduling; scheduling map and reduce tasks; keeping track of unreachable nodes; resubmitting failed tasks; and reporting execution statistics.

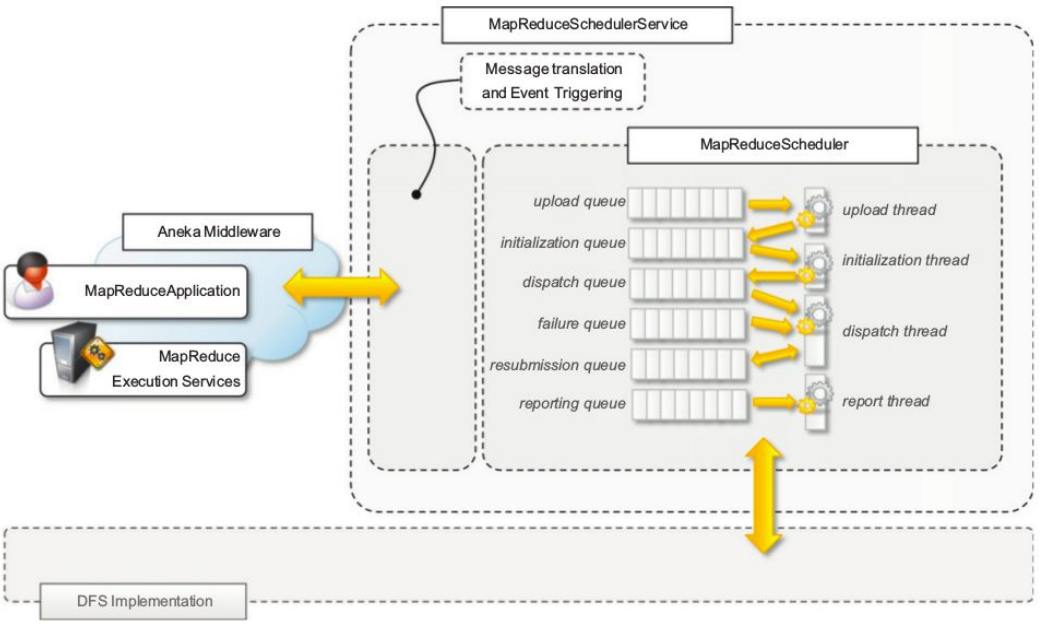


FIGURE 8.9

MapReduce Scheduling Service architecture.

Task Execution. The execution of tasks is controlled by the MapReduce Execution Service. This component plays the role of the worker process in the Google MapReduce implementation. The service manages the execution of map and reduce tasks and performs other operations, such as sorting and merging intermediate files. The service is internally organized, as described in **Figure 8.10**.

There are three major components that coordinate together for executing tasks:

1. MapReduce- SchedulerService,
2. ExecutorManager, and
3. MapReduceExecutor.

The MapReduceSchedulerService interfaces the ExecutorManager with the Aneka middleware; the ExecutorManager is in charge of keeping track of the tasks being executed by demanding the specific execution of a task to the MapReduceExecutor and of sending the statistics about the execution back to the Scheduler Service.

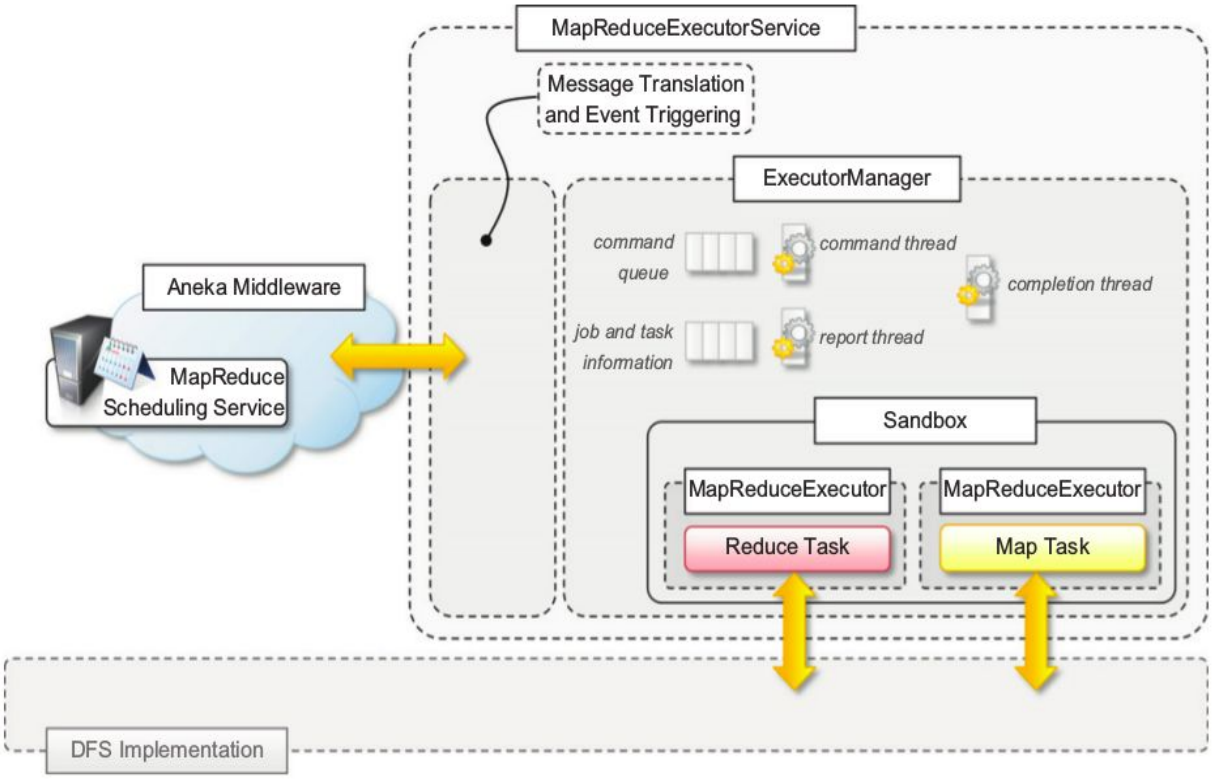


FIGURE 8.10

MapReduce Execution Service architecture.

3 Distributed file system support

Aneka supports, the MapReduce model that uses a distributed file system implementation.

Distributed file system implementations guarantee high availability and better efficiency by means of replication and distribution.

the original MapReduce implementation assumes the existence of a distributed and reliable storage; hence, the use of a distributed file system for implementing the storage layer is natural.

Aneka provides the capability of interfacing with different storage implementations and it maintains the same flexibility for the integration of a distributed file system.

The level of integration required by MapReduce requires the ability to perform the following tasks:

- Retrieving the location of files and file chunks
- Accessing a file by means of a stream

The first operation is useful to the scheduler for optimizing the scheduling of map and reduce tasks according to the location of data; the second operation is required for the usual I/O operations to and from data files.

On top of these low-level interfaces, MapReduce programming model offers classes to read from and write to files in a sequential manner. These are classes **SeqReader** and **SeqWriter**. They provide sequential access for reading and writing key-value pairs, and they expect specific file format, which is described in **Figure 8.11**.

An Aneka MapReduce file is composed of a header, used to identify the file, and a sequence of record blocks, each storing a key-value pair. The header is composed of 4 bytes: the first 3 bytes represent the character sequence SEQ and the fourth byte identifies the version of the file. The record block is composed as follows: the first 8 bytes are used to store two integers representing the length of the rest of the block and the length of the key section, which is immediately following. The remaining part of the block stores the data of the value component of the pair. The SeqReader and SeqWriter classes are designed to read and write files in this format by transparently handling the file format information and translating key and value instances to and from their binary representation.

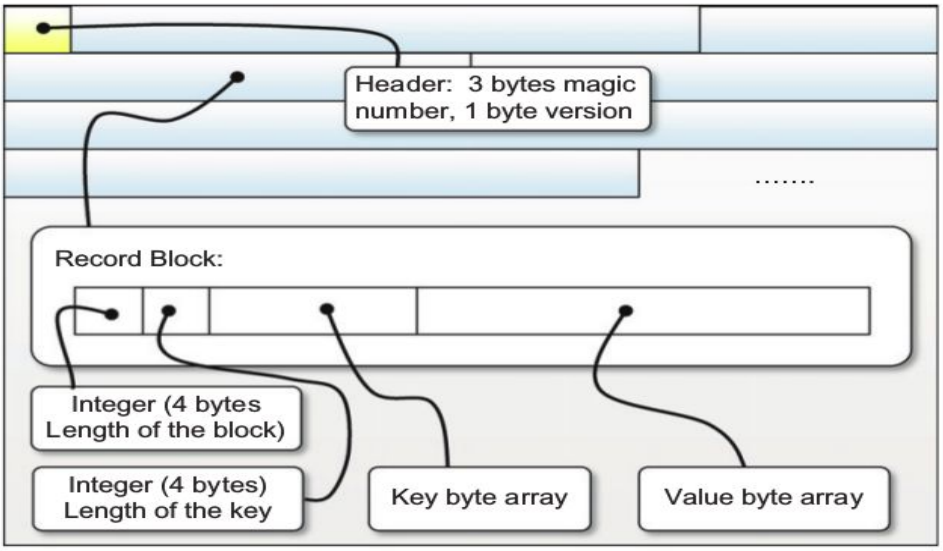


FIGURE 8.11

Aneka MapReduce data file format.

Listing 8.8 shows the interface of the SeqReader and SeqWriter classes. The SeqReader class provides an enumerator-based approach through which it is possible to access the key and the value sequentially by calling the NextKey() and the NextValue() methods, respectively. It is also possible to access the raw byte data of keys and values by using the NextRawKey() and NextRawValue(). HasNext() returns a Boolean, indicating whether there are more pairs to read or not.

```
namespace Aneka.MapReduce.DiskIO
{
        /// Class <b><i>SeqReader</i></b>. This class implements a file reader for the sequence
        /// file, which isa standard file split used by MapReduce.NET to store a partition of a fixed size of a data file.
        public class SeqReader
```

```
{
        /// Creates a SeqReader instance and attaches it to the given file.
        public SeqReader(string file) : this(file, null, null) { ... }
        /// Creates a SeqReader instance, attaches it to the given file, and sets the
        /// internal buffer size to bufferSize.
        public SeqReader(string file, int bufferSize) : this(file,null,null,bufferSize) { ... }
        /// Creates a SeqReader instance, attaches it to the given file, and provides
        /// metadata information about the content of the file in the form of keyType andcvalueType.
        public SeqReader(string file, Type keyType, Type valueType) : this(file, keyType, valueType,
        SequenceFile.DefaultBufferSize) { ... }
        /// Creates a SeqReader instance, attaches it to the given file, and provides
        /// metadata information about the content of the file in the form of keyType and valueType.
        public SeqReader(string file, Type keyType, Type valueType, int bufferSize){ ... }
        /// Sets the metadata information about the keys and the values contained in the data file.
        public void SetType(Type keyType, Type valueType) { ... }
        /// Checks whether there is another record in data file and moves current file pointer to its beginning.
        public bool HaxNext() { ... }
        /// Gets the object instance corresponding to the next key in the data file. in the data file.
        public object NextKey() { ... }
        /// Gets the object instance corresponding to the next value in the data file. in the data file.
        public object NextValue() { ... }
        /// Gets the raw bytes that contain the value of the serializedinstance of the current key.
        public BufferInMemory NextRawKey() { ... }
        /// Gets the raw bytes that contain the value of the serialized instance of the current value.
        public BufferInMemory NextRawValue() { ... }
        /// Gets the position of the file pointer as an offset from its beginning.
        public long CurrentPosition() { ... }
        /// Gets the size of the file attached to this instance of SeqReader.
        public long StreamLength() { ... }
        /// Moves file pointer to position. If value of position is 0 or -ve, returns current position of file pointer.
        public long Seek(long position) { ... }
        /// Closes the SeqReader instanceand releases all resources that have been allocated to read fromthe file.
        public void Close() { ... }
        // private implementation follows
}
/// Class SeqWriter. This class implements a file writer for the sequence file, which is a standard file split used by
///MapReduce.NET to store a partition of a fixed size of a data file. This classprovides an interface to add a
///sequence of key-value pair incrementally.
public class SeqWriter
{
        /// Creates a SeqWriter instance for writing to file. This constructor initializes
        /// the instance with the default value for the internal buffers.
        public SeqWriter(string file) : this(file, SequenceFile.DefaultBufferSize){ ... }
        /// Creates a SeqWriter instance, attachesit to the given file, and sets the
        /// internal buffer size to bufferSize.
        public SeqWriter(string file, int bufferSize) { ... }
        /// Appends a key-value pair to the data file split.
        public void Append(object key, object value) { ... }
        /// Appends a key-value pair to the data file split.
        public void AppendRaw(byte[] key, byte[] value) { ... }
        /// Appends a key-value pair to the data file split.
        public void AppendRaw(byte[] key, int keyPos, int keyLen,
```

```
                byte[] value, int valuePos, int valueLen) { ... }
                /// Gets the length of the internal buffer or 0 if no buffer has been allocated.
                public longLength() { ... }
                /// Gets the length of data file split on disk so far.
                public long FileLength() { ... }
                /// Closes SeqReader instance and releases all the resources that have been allocated to write to the file.
                public void Close() { ... }
                // private implementation follows
        }
}
```

LISTING 8.8 SeqReader and SeqWriter Classes.

Listing 8.9 shows a practical use of the SeqReader class by implementing the callback used in the word-counter example. To visualize the results of the application, we use the SeqReader class to read the content of the output files and dump it into a proper textual form that can be visualized with any text editor, such as the Notepad application.

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.WordCounter
{
        /// Class Program. Application driver for the Word Counter sample.
        public class Program
        {
                /// Reference to the configuration object.
                private static Configuration configuration = null;
                /// Location of the configuration file.
                private static string confPath = "conf.xml";
                /// Processes the arguments given to the application and according
                /// to the parameters read runs the application or shows the help.
                private static void Main(string[] args)
                {
                try
                {
                        Logger.Start();
                        // get the configuration
                        Program.configuration = Configuration.GetConfiguration(confPath);
                        // configure MapReduceApplication
                        MapReduceApplication<WordCountMapper, WordCountReducer> application =
                        new MapReduceApplication<WordCountMapper, WordCountReducer>("WordCounter",
                        configuration);
                        // invoke and wait for result
                        application.InvokeAndWait(newEventHandler<ApplicationEventArgs>(OnDone));
                        // alternatively we can use the following call
                }
                catch(Exception ex)
                {
                        Program.Usage();
                        IOUtil.DumpErrorReport(ex, "Aneka WordCounter Demo - Error Log");
                }
                finally
                {
```

```
                Logger.Stop();
            }
        }
        /// Hooks the ApplicationFinished events and process the results
        /// if the application has been successful.
        private static void OnDone(object sender, ApplicationEventArgs e)
        {
            if (e.Exception != null)
            {
                IOUtil.DumpErrorReport(e.Exception, "Aneka WordCounter Demo - Error");
            }
            else
            {
                string outputDir = Path.Combine(configuration.Workspace, "output");
                try
                {
                    FileStream resultFile = new FileStream("WordResult.txt",FileMode.Create,
                    FileAccess.Write);
                    Stream WritertextWriter = new StreamWriter(resultFile);
                    DirectoryInfo sources = new DirectoryInfo(outputDir);
                    FileInfo[] results = sources.GetFiles();
                    foreach(FileInfo result in results)
                    {
                        SeqReader seqReader = newSeqReader(result.FullName);
                        seqReader.SetType(typeof(string), typeof(int));
                        while(seqReader.HaxNext() == true)
                        {
                            object key = seqReader.NextKey();
                            object value = seqReader.NextValue();
                            textWriter.WriteLine("{0}\t{1}", key, value);
                        }
                        seqReader.Close();
                    }
                    textWriter.Close();
                    resultFile.Close();
                    // clear the output directory
                    sources.Delete(true);
                    Program.StartNotePad("WordResult.txt");
                }
                catch(Exception ex)
                {
                    IOUtil.DumpErrorReport(e.Exception, "Aneka WordCounter Demo - Error");
                }
            }
            }
            /// Starts the notepad process and displays the given file.
            private static voidStartNotepad(string file) { ... }
            /// Displays a simple informative message explaining the usage of the application.
            private static void Usage() { ... }
        }
}
```

LISTING 8.9 WordCounter Job.

8.3.2 Example application

To demonstrate how to program real applications with Aneka MapReduce, we consider a very common task: log parsing. We design a MapReduce application that processes the logs produced by the Aneka container in order to extract some summary information about the behavior of the Cloud.

1 Parsing Aneka logs

Aneka components produce a lot of information that is stored in the form of log files.
In this example, we parse these logs to extract useful information about the execution of applications and the usage of services in the Cloud.
The entire framework leverages the log4net library for collecting and storing the log information.
Some examples of formatted log messages are:

15 Mar 2011 10:30:07 DEBUGSchedulerService: . . .
HandleSubmitApplicationSchedulerService: . . .
15 Mar 2011 10:30:07 INFOSchedulerService: Scanning candidate storage . . .
15 Mar 2011 10:30:10 INFOAdded [WU: 51d55819-b211-490f-b185-8a25734ba705,
4e86fd02. . .
15 Mar 2011 10:30:10 DEBUGStorageService:NotifySchedulerSending
FileTransferMessage. . .
15 Mar 2011 10:30:10 DEBUGIndependentSchedulingService:QueueWorkUnitQueueing. . .
15 Mar 2011 10:30:10 INFOAlgorithmBase::AddTasks[64] Adding 1 Tasks
15 Mar 2011 10:30:10 DEBUGAlgorithmBase:FireProvisionResourcesProvision

Possible information that we might want to extract from such logs is the following:

- The distribution of log messages according to the level
- The distribution of log messages according to the components

This information can be easily extracted and composed into a single view by creating Mapper tasks that count the occurrences of log levels and component names and emit one simple key-value pair in the form (level-name, 1) or (component-name, 1) for each of the occurrences. The Reducer task will simply sum up all the key-value pairs that have the same key. For both problems, the structure of the map and reduce functions will be the following:

map: (long; string) => (string; long)
reduce: (long; string) => (string; long)

The Mapper class will then receive a key-value pair containing the position of the line inside the file as a key and the log message as the value component. It will produce a key-value pair containing a string representing the name of the log level or the component name and 1 as value. The Reducer class will sum up all the key-value pairs that have the same name.

2 Mapper design and implementation

The operation performed by the map function is a very simple text extraction that identifies the level of the logging and the name of the component entering the information in the log. Once this information is extracted, a key-value pair (string, long) is emitted by the function.
Listing 8.10 shows the implementation of the Mapper class for the log parsing task. The Map method simply locates the position of the log-level label into the line, extracts it, and emits a corresponding key-value pair (label, 1).

```
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.LogParsing
{
        /// Class LogParsingMapper. Extends Mapper<K,V> and provides an
        /// implementation of the map function for parsing the Aneka container log files .
        /// This mapper emits a ke y-value (log-level, 1) and potentially another key -value
        /// (_aneka -component-name,1) if it is able to extract such information from the
        /// input.
```

```
public class LogParsingMapper: Mapper<long,string>
{
    /// Reads the input and extracts the information about the log level and if
    /// found the name of the aneka component that entered the log line .
    protected override void Map(IMapInput<long,string>input)
    {
        // we don't care about the key, because we are only interested on
        // counting the word of each line.
        string value = input.Value;
        long quantity = 1;
        // first we extract the log level name information. Since the date is reported
        // in the standard format DD MMM YYYY mm:hh:ss it is possible to skip the first
        // 20 characters (plus one space) and then extract the next following characters
        // until the next position of the space character.
        int start = 21;
        int stop = value.IndexOf( ' ', start);
        string key = value.Substring(start, stop – start);
        this.Emit(key, quantity);
        //now we are looking for the Aneka component name that entered the log line
        //if this is inside the log line it is just right after the log level preceeded
        //by the character sequence <space><dash><space> and terminated by the <c olon> character.
        start = stop + 3; // we skip the <space><dash><space> sequence.
        stop = value.IndexOf( ':', start);
        key = value.Substring(start, stop – start);
        // we now check whether the key contains any space, if not then it is the name
        // of an Aneka component and the line does not need to be skipped.
        if (key.IndexOf(' ') == -1)
        {
            this.Emit("_" + key, quantity);
        }
    }
}
}
```

LISTING 8.10 Log-Parsing Mapper Implementation.

3 Reducer design and implementation

The implementation of the reduce function is even more straightforward; the only operation that needs to be performed is to add all the values that are associated to the same key and emit a key-value pair with the total sum.

Listing 8.11, the operation to perform is very simple and actually is the same for both of the two different key-value pairs extracted from the log lines.

```
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.LogParsing
{
    /// Class <b><i>LogParsingReducer</i></b>. Extends Reducer<K,V> and provides an
    /// implementation of the redcuce function for parsing the Aneka container log files .
    /// The Reduce method iterates all ov er values of the enumerator and sums the values
    /// before emitting the sum to the output file.
    public class LogParsingReducer : Reducer<string,long>
    {
        /// Iterates all over the values of the enum erator and sums up
        /// all the values before emitting the sum to the output file.
```

```
        protected override void Reduce(IReduceInputEnumerator<long>input)
        {
            long sum = 0;
            while(input.MoveNext())
            {
                long value = input.Current;
                sum += value;
            }
            this.Emit(sum);
        }
    }
}
```

LISTING 8.11 Aneka Log-Parsing Reducer Implementation.

4 Driver program

LogParsingMapper and LogParsingReducer constitute the core functionality of the MapReduce job, which only requires to be properly configured in the main program in order to process and produce text tiles.

Another task that is performed in the driver application is the separation of these two statistics into two different files for further analysis.

Listing 8.12 shows the implementation of the driver program. With respect to the previous examples, there are three things to be noted:

- The configuration of the MapReduce job
- The post-processing of the result files
- The management of errors

The configuration of the job is performed in the Initialize method. This method reads the configuration file from the local file system and ensures that the input and output formats of files are set to text.

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;
namespace Aneka.MapReduce.Examples.LogParsing
{
    /// Class Program. Application driver. This class sets up the MapReduce
    /// job and configures it with the <i>LogParsingMapper</i> and <i>LogParsingReducer</i>
    /// classes. It also configures the MapReduce runtime in order sets the appropriate
    /// format for input and output files.
    public class Program
    {
    /// Reference to the configuration object.
    private static Configuration configuration = null;
    /// Location of the configuration file.
    private static string confPath = "conf.xml";
    /// Processes the arguments given to the application and according
    /// to the parameters read runs the application or shows the help.
    private static void Main(string[] args)
    {
        try
        {
            Logger.Start();
            // get the configuration
            Program.configuration = Program.Initialize(confPath);
            // configure MapReduceApplication
```

```
                MapReduceApplication<LogParsingMapper, LogParsingReducer> application =
                new MapReduceApplication<LogParsingMapper, LogParsingReducer>("LogParsing",
                configuration);
                // invoke and wait for result
                application.InvokeAndWait(newEventHandler<ApplicationEventArgs>(OnDone));
                // alternatively we can use the following call
                // application.InvokeAndWait();
        }
        catch(Exception ex)
        {
                Program.ReportError(ex);
        }
        finally
        {
                Logger.Stop();
        }
        Console.Readline();
}
/// Initializes the configuration and ensures that the appropriate input
/// and output formats . are set
private static Configuration Initialize(stringconfigFile)
{
        Configuration conf = Configuration.GetConfiguration(confPath);
        // we ensure that the input and the output formats are simple text files.
        PropertyGroup mapReduce = conf["MapReduce"];
        if (mapReduce == null)
        {
                mapReduce = newPropertyGroup("MapReduce");
                conf.Add("MapReduce") = mapReduce;
        }
        // handling input properties
        PropertyGroup group = mapReduce.GetGroup("Input");
        if (group == null)
        {
                group = newPropertyGroup("Input");
                mapReduce.Add(group);
        }
        string val = group["Format"];
        if (string.IsNullOrEmpty(val) == true)
        {
                group.Add("Format","text");
        }
        val = group["Filter"];
        if (string.IsNullOrEmpty(val) == true)
        {
                group.Add("Filter","*.log");
        }
        // handling output properties
        group = mapReduce.GetGroup("Output");
        if (group == null)
        {
                group = newPropertyGroup("Output");
```

```
                mapReduce.Add(group);
        }
        val = group["Format"];
        if (string.IsNullOrEmpty(val) == true)
        {
                group.Add("Format","text");
        }
        return conf;
}
/// Hooks the ApplicationFinished events and process the results
/// if the application has been successful.
private static void OnDone(object sender, ApplicationEventArgs e)
{
        if (e.Exception != null)
        {
                Program.ReportError(ex);
        }
else
{
        Console.Write("Aneka Log Parsing–Job Terminated: SUCCESS");
        FileStream logLevelStats = null;
        FileStream componentStats = null;
        string workspace = Program.configuration.Workspace;
        string outputDir = Path.Combine(workspace, "output");
        DirectoryInfo sources = new DirectoryInfo(outputDir);
        FileInfo[] results = sources.GetFiles();
        try
        {
                logLevelStats = new FileStream(Path.Combine(workspace,"loglevels.txt"),
                FileMode.Create,FileAccess.Write));
                componentStats = new FileStream(Path.Combine(workspace,"components.txt"),
                FileMode.Create,FileAccess.Write));
                using(StreamWriter logWriter = new StreamWriter(logLevelStats))
                {
                        using(StreamWritercompWriter = newStreamWriter(componentStats))
                        {
                                foreach(FileInfo result in results)
                                {
                                        using(StreamReader reader =new StreamReader(result.OpenRead()))
                                        {
                                                while(reader.EndOfStream == false)
                                                {
                                                        string line = reader.ReadLine();
                                                        if (line != null)
                                                        {
                                                                if (line.StartsWith("_ ") == true)
                                                                {
                                                                    compWriter.WriteLine(line.Substring(1));
                                                                }
                                                                else
                                                                {
                                                                    logWriter.WriteLine(line);
```

```
                                                                    }
                                                                }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                            // clear the output directory
                                            sources.Delete(true);
                                            Console.WriteLine("Statistics saved to:[loglevels.txt, components.txt]");
                                            Environment.ExitCode = 0;
                                    }
                                    catch(Exception ex)
                                    {
                                            Program.ReportError(ex);
                                    }
                                    Console.WriteLine("<Press Return>");
                            }
}
/// Displays a simple informative message explaining the usage of the application.
private static void Usage()
{
        Console.WriteLine("Aneka Log Parsing – Usage Log.Parsing.Demo.Console.exe"+ " [conf.xml]");
}
/// Dumps the error to the console, sets the exit code of the application to -1 and saves the error dump into a file.
private static void ReportError(Exception ex)
{
        IOUtil.DumpErrorReport(Console.Out, ex,"Aneka Log Parsing–Job Terminated: "+ "ERROR");
        IOUtil.DumpErrorReport(ex, "Aneka Log Parsing–Job Terminated: ERROR");
        Program.Usage();j
        Environment.ExitCode = -1;
}
}
}
```

LISTING 8.12 Driver Program Implementation.

Instead of a programming approach for the initialization of the configuration, it is also possible to embed these settings into the standard Aneka configuration file, as demonstrated in **Listing 8.13.**

```
<?xml version="1.0" encoding="utf -8" ?>
<Aneka>
        <UseFileTransfervalue="false" />
        <Workspacevalue="Workspace" />
        <SingleSubmission value="AUTO" />
        <PollingTimevalue="1000"/>
        <LogMessagesvalue="false" />
        <SchedulerUrivalue="tcp://localhost:9090/Aneka" />
        <UserCredential type="Aneka.Security.UserCredentials" assembly="Aneka.dll">
                <UserCredentials username="Administrator" password=""/>
        </UserCredentials>
        <Groups>
                <Group name="MapReduce">
```

```
                    <Groups>
                        <Group name="Input">
                        <Property name="Format" value="text" />
                        <Property name="Filter" value="*.log" />
                    </Group>
                    <Group name="Output">
                    <Property name="Format" value="text" />
                </Group>
        </Groups>
        <Property name="LogFile" value="Execution.log"/>
        <Property name="FetchResult" value="true" />
        <Property name="UseCombiner" value="true" />
        <Property name="SynchReduce" value="false" />
        <Property name="Partitions" value="1" />
        <Property name="Attempts" value="3" />
        </Group>
        </Groups>
</Aneka>
```

LISTING 8.13 Driver Program Configuration File (conf.xml)

5 Running the application

Aneka produces a considerable amount of logging information. The default configuration of the logging infrastructure creates a new log file for each activation of the Container process or as soon as the dimension of the log file goes beyond 10 MB. Therefore, by simply continuing to run an Aneka Cloud for a few days, it is quite easy to collect enough data to mine for our sample application.

In the execution of the test, we used a distributed infrastructure consisting of seven worker nodes and one master node interconnected through a LAN. We processed 18 log files of several sizes for a total aggregate size of 122 MB. The execution of the MapReduce job over the collected data produced the results that are stored in the loglevels.txt and components.txt files and represented graphically in **Figures 8.12 and 8.13**, respectively.

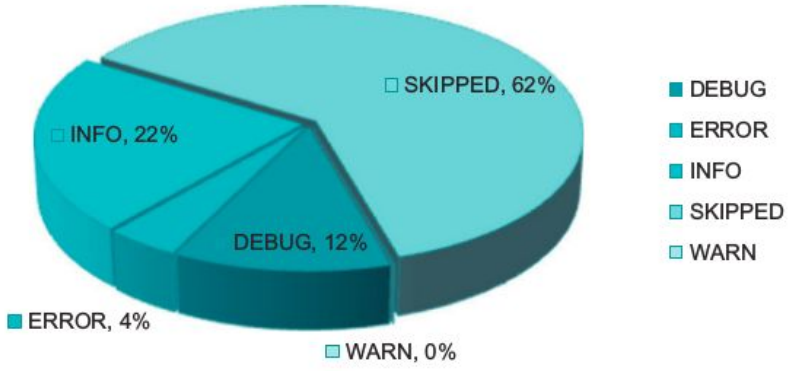


FIGURE 8.12

Log-level entries distribution.

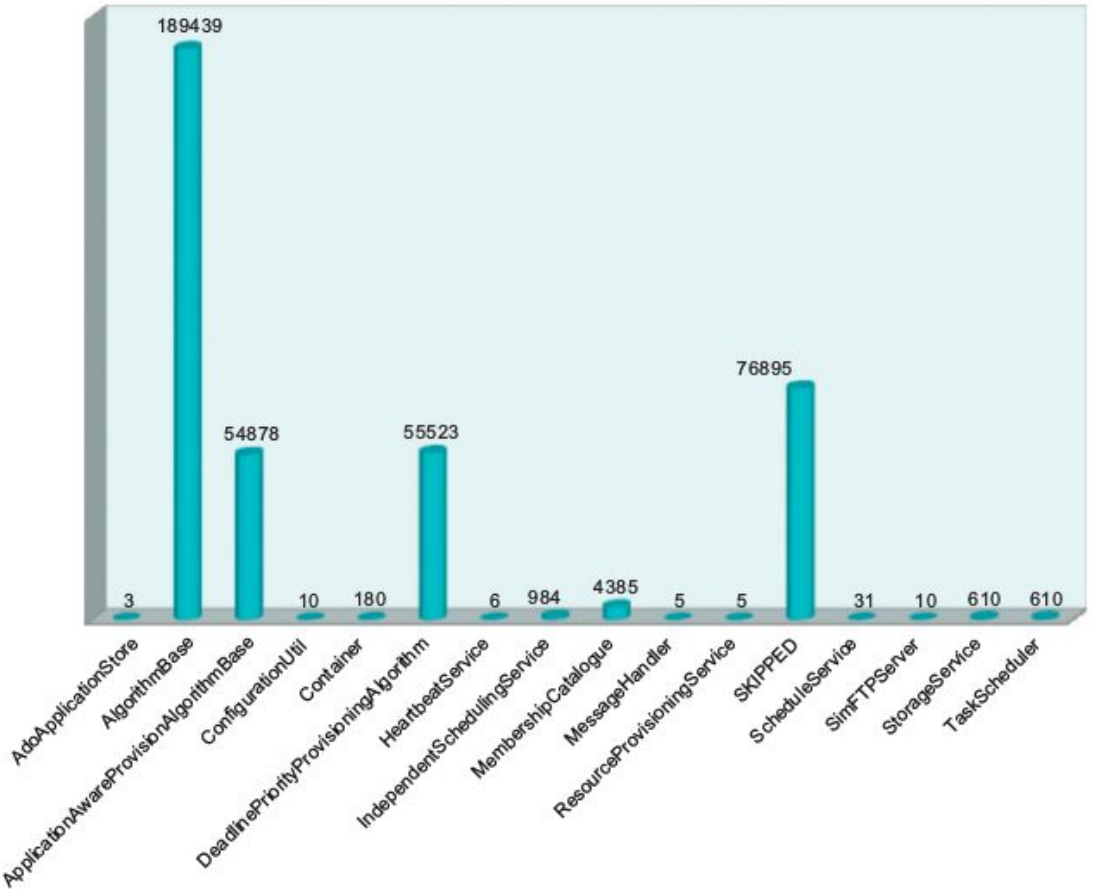


FIGURE 8.13

Component entries distribution.

The two graphs show that there is a considerable amount of unstructured information in the log files produced by the Container processes. In particular, about 60% of the log content is skipped during the classification. This content is more likely due to the result of stack trace dumps into the log file, which produces—as a result of ERROR and WARN entries—a sequence of lines that are not recognized. Figure 8.13 shows the distribution among the components that use the logging APIs. This distribution is computed over the data recognized as a valid log entry, and the graph shows that just about 20% of these entries have not been recognized by the parser implemented in the map function. We can then infer that the meaningful information extracted from the log analysis constitutes about 32% (80% of 40% of the total lines parsed) of the entire log data.