FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

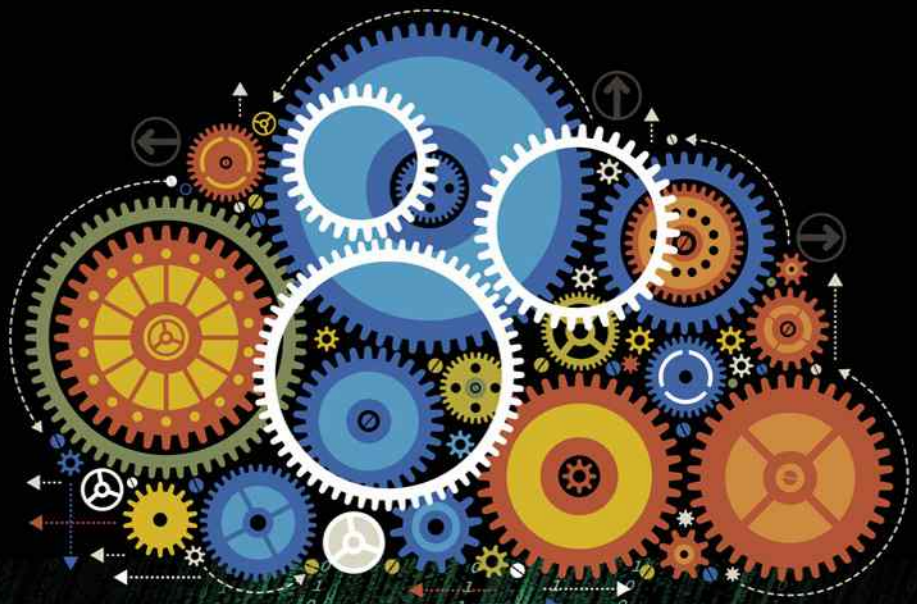
Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE



MASTERING CLOUD COMPUTING

FOUNDATIONS AND APPLICATIONS PROGRAMMING

MORGAN KAUFMANN

Rajkumar Buyya, Christian Vecchiola, S. Thamarai Selvi

CHAPTER

Data-Intensive Computing
MapReduce Programming

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. These tasks become challenging as the quantity of information accumulates and increases over time at higher rates. 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. Despite this fact, the use of parallel and distributed techniques as a support of data-intensive computing is not straightforward, but several challenges in the form of data representation, efficient algorithms, and scalable infrastructures need to be faced.

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. Practical examples of MapReduce applications for data-intensive computing are demonstrated using the Aneka MapReduce Programming Model.

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 [73]. The term *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 the 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.

Besides scientific computing, several IT industry sectors require support for data-intensive computations. Customer data for any telecom company would easily be in the range of 10−100 terabytes. This volume of information is not only processed to generate billing statements, but it is also mined to identify scenarios, trends, and patterns that help these companies provide better service. Moreover, it is reported that U.S. handset mobile traffic has reached 8 petabytes per month and it is expected to grow up to 327 petabytes per month by 2015.[1] The scale of petabytes is even more common when we consider IT giants such as Google, which is reported to process about 24 petabytes of information per day [55] and to sort petabytes of data in hours.[2] Social networking and gaming are two other sectors in which data-intensive computing is now a reality. Facebook inbox search operations involve crawling about 150 terabytes of data, and the whole uncompressed data stored by the distributed infrastructure reach to 36 petabytes.[3] Zynga, *a* social gaming platform, moves 1 petabyte of data daily and it has been reported to add 1,000 servers every week to store the data generated by games like Farmville and Frontierville.[4]

8.1.1 Characterizing data-intensive computations

Data-intensive applications not only deal with huge volumes of data but, very often, also exhibit compute-intensive properties [74]. 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. Datasets are commonly persisted in several formats and distributed across different locations. Such applications process data in multistep analytical pipelines, including transformation and fusion stages. The processing requirements scale almost linearly with the data size, and they can be easily processed in parallel. They also need efficient mechanisms for data management, filtering and fusion, and efficient querying and distribution [74].

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 for distributed computing. For example, the location of data is crucial as the need for moving terabytes of data becomes an obstacle for high-performing computations. Data partitioning as well as content replication and scalable algorithms help in improving the performance of data-intensive applications. Open challenges in data-intensive computing given by Ian Gorton et al. [74] are:

- Scalable algorithms that can search and process massive datasets
- New metadata management technologies that can scale to handle complex, heterogeneous, and distributed data sources

[1]Coda Research Consultancy, www.codaresearch.co.uk/usmobileinternet/index.htm.
[2]Google's Blog, http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.
[3]OSCON 2010, David Recordon (Senior Open Programs Manager, Facebook): Today's LAMP Stack, Keynote Speech. Available at www.oscon.com/oscon2010/public/schedule/speaker/2442.
[4]http://techcrunch.com/2010/09/22/zynga-moves-1-petabyte-of-data-daily-adds-1000-servers-a-week/.

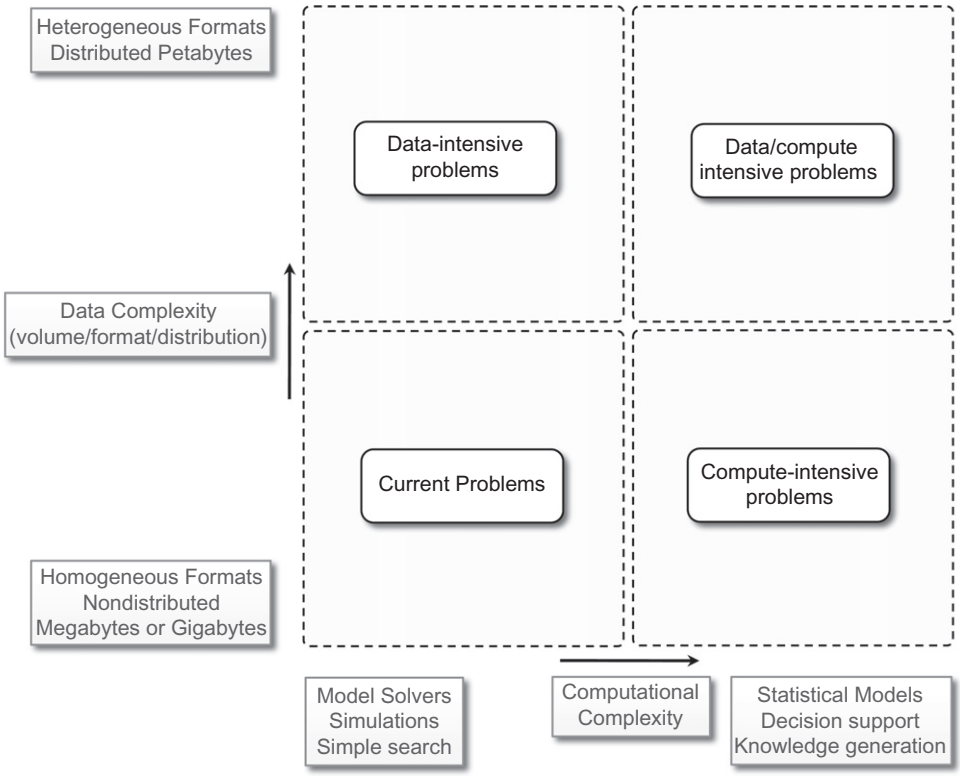


FIGURE 8.1

Data-intensive research issues.

- Advances in high-performance computing platforms aimed at providing a better support for accessing in-memory multiterabyte data structures
- High-performance, highly reliable, petascale distributed file systems
- Data signature-generation techniques for data reduction and rapid processing
- New approaches to software mobility for delivering algorithms that are able to move the computation to where the data are located
- Specialized hybrid interconnection architectures that provide better support for filtering multigigabyte datastreams coming from high-speed networks and scientific instruments
- Flexible and high-performance 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. We track the evolution of this phenomenon by highlighting the most relevant contributions in the area of storage and networking and infrastructure software.

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

The evolution of technologies, protocols, and algorithms for data transmission and streaming has been an enabler of data-intensive computations [75]. 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). On that occasion, the remote visualization of large and complex scientific datasets (a high-resolution magnetic resonance image, or MRI, scan of the human brain) was set up between the Pittsburgh Supercomputing Center (PSC) and Albuquerque, New Mexico, the location of the conference.

A further step was made by the Kaiser project [76], which made available as remote data sources high data rate and online instrument systems. The project leveraged the Wide Area Large Data Object (WALDO) system [77], which was used to provide the following capabilities: automatic generation of metadata; automatic cataloguing of data and metadata while processing the data in real time; facilitation of cooperative research by providing local and remote users access to data; and mechanisms to incorporate data into databases and other documents.

The first data-intensive environment is reported to be the MAGIC project, a DARPA-funded collaboration working on distributed applications in large-scale, high-speed networks. Within this context, the *Distributed Parallel Storage System (DPSS)* was developed, later used to support *TerraVision* [78], a terrain visualization application that lets users explore and navigate a tridimensional real landscape.

Another important milestone was set with the Clipper project,[5] a collaborative effort of several scientific research laboratories, with the goal of designing and implementing a collection of independent but architecturally consistent service components to support data-intensive computing. The challenges addressed by the Clipper project included management of substantial computing resources, generation or consumption of high-rate and high-volume data flows, human interaction management, and aggregation of disperse resources (multiple data archives, distributed computing capacity, distributed cache capacity, and guaranteed network capacity). Clipper's main focus was to develop a coordinated collection of services that can be used by a variety of applications to build on-demand, large-scale, high-performance, wide-area problem-solving environments.

8.1.3.2 Data grids

With the advent of grid computing [8], huge computational power and storage facilities could be obtained by harnessing heterogeneous resources across different administrative domains. Within this context, *data grids* [79] 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

[5] www.nersc.gov/news/annual_reports/annrep98/16clipper.html.

discovery and management mechanisms for easy access to distributed datasets [80]. Because data grids span different administration boundaries, access control and security are important concerns.

Data grids mostly provide storage and dataset management facilities as support for scientific experiments that produce huge volumes of data. The reference scenario might be one depicted in Figure 8.2. Huge amounts of data are produced by scientific instruments (telescopes, particle accelerators, etc.). The information, which can be locally processed, is then stored in repositories and made available for experiments and analysis to scientists, who can be local or, most likely, remote. Scientists can leverage specific discovery and information services, which help in determining the locations of the closest datasets of interest for their experiments. Datasets are replicated by the infrastructure to provide better availability. Since processing of this information also requires a large computational power, specific computing sites can be accessed to perform analysis and experiments.

Like any other grid infrastructure, heterogeneity of resources and different administrative domains constitute a fundamental aspect that needs to be properly addressed with security measures and the use of *virtual organizations (VO)*. Besides heterogeneity and security, data grids have their own characteristics and introduce new challenges [79]:

- *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.
- *Shared data collections.* Resource sharing includes distributed collections of data. For example, repositories can be used to both store and read data.
- *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.
- *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.

With respect to the combination of several computing facilities through high-speed networking, data grids constitute a more structured and integrated approach to data-intensive computing. As a result, several scientific research fields, including high-energy physics, biology, and astronomy, leverage data grids, as briefly discussed here:

- *The LHC Grid.* A project funded by the European Union to develop a worldwide grid computing environment for use by high-energy physics researchers around the world who are collaborating on the Large Hadron Collider (LHC) experiment. It supports storage and analysis of large-scale datasets, from hundreds of terabytes to petabytes, generated by the LHC experiment (http://lhc.web.cern.ch/lhc/).
- *BioInformatics Research Network (BIRN).* BIRN is a national initiative to advance biomedical research through data sharing and online collaboration. Funded by the National Center for Research Resources (NCRR), a component of the U.S. National Institutes of Health (NIH), BIRN provides a data-sharing infrastructure, software tools, and strategies and advisory services (www.birncommunity.org).

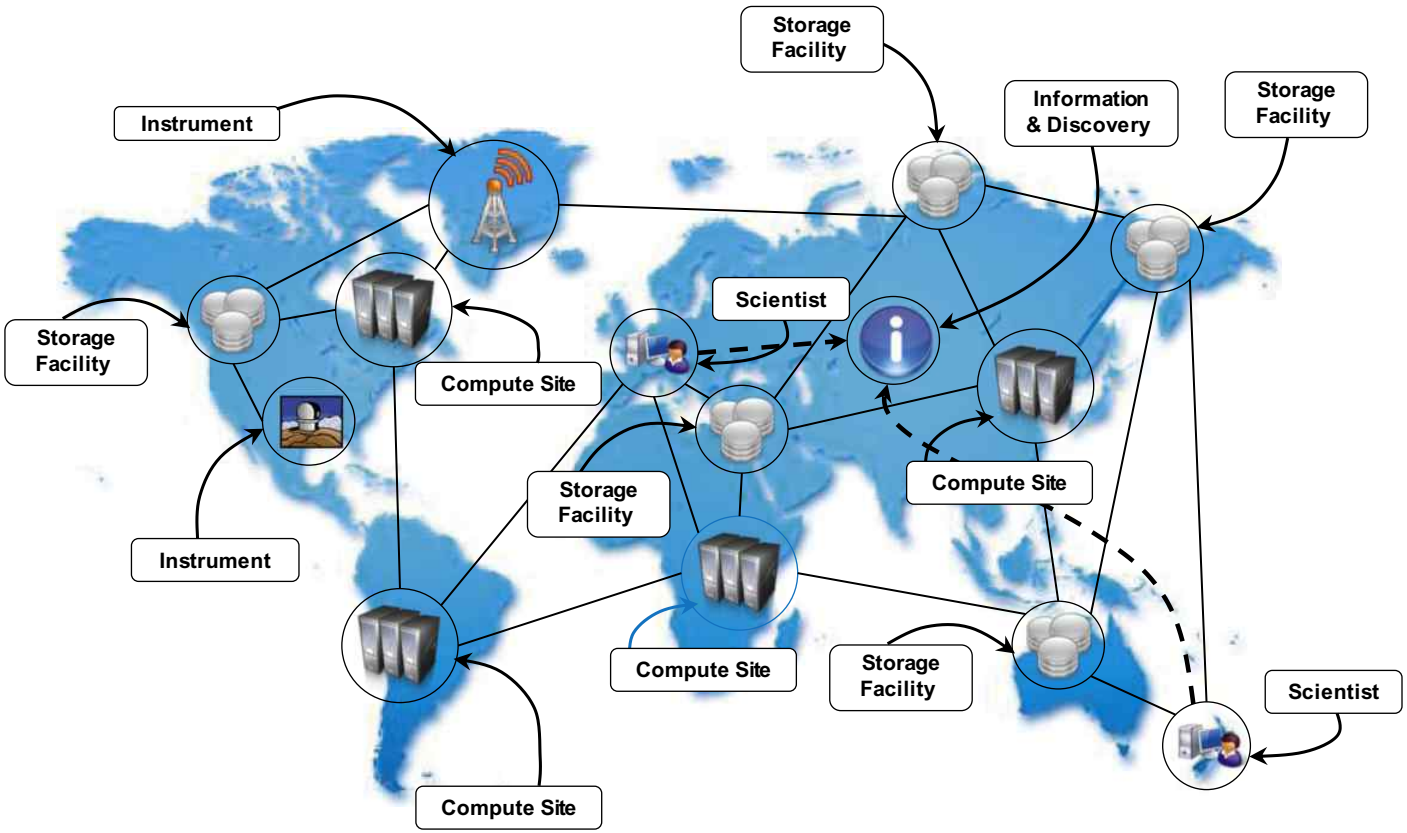


FIGURE 8.2

Data grid reference scenario.

- *International Virtual Observatory Alliance (IVOA)*. IVOA is an organization that aims to provide improved access to the ever-expanding astronomical data resources available online. It does so by promoting standards for *virtual observatories*, which are a collection of interoperating data archives and software tools that use the Internet to form a scientific research environment in which astronomical research programs can be conducted. This allows scientists to discover, access, analyze, and combine lab data from heterogeneous data collections (www.ivoa.net).

A complete taxonomy of Data Grids can be found in Venugopal et al. [79].

8.1.3.3 Data clouds and "Big Data"

Large datasets have mostly been the domain of scientific computing. This scenario has recently started to change as massive amounts of data are being produced, mined, and crunched by companies that provide Internet services such as searching, online advertising, and social media. It is critical for such companies to efficiently analyze these huge datasets because they constitute a precious source of information about their customers. *Log analysis* is an example of a data-intensive operation that is commonly performed in this context; companies such as Google have a massive amount of data in the form of logs that are daily processed using their distributed infrastructure. As a result, they settled upon an analytic infrastructure, which differs from the grid-based infrastructure used by the scientific community.

Together with the diffusion of cloud computing technologies that support data-intensive computations, the term *Big Data* [82] has become popular. This term 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. Relational databases and desktop statistics/visualization packages become ineffective for that amount of information, instead requiring "massively parallel software running on tens, hundreds, or even thousands of servers" [82].

Big Data problems are found in nonscientific application domains such as weblogs, radio frequency identification (RFID), sensor networks, social networks, Internet text and documents, Internet search indexing, call detail records, military surveillance, medical records, photography archives, video archives, and large scale ecommerce. Other than the massive size, what characterizes all these examples is that new data are accumulated with time rather than replacing the old data. 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 terabytes to many petabytes of data in a single dataset [82].

Cloud technologies support data-intensive computing in several ways:

- By providing a large amount of compute instances on demand, which can be used to process and analyze large datasets in parallel.
- By providing a storage system optimized for keeping large blobs of data and other distributed data store architectures.
- By providing frameworks and programming APIs optimized for the processing and management of large amounts of data. These APIs are mostly coupled with a specific storage infrastructure to optimize the overall performance of the system.

A *data cloud* is a combination of these components. An example is the *MapReduce* framework [55], which provides the best performance for leveraging the Google File System [54] on top of Google's large computing infrastructure. Another example is the *Hadoop* system [83], the most mature, large, and open-source data cloud. It consists of the Hadoop Distributed File System (HDFS) and Hadoop's implementation of MapReduce. A similar approach is proposed by *Sector* [84], which consists of the Sector Distributed File System (SDFS) and a compute service called *Sphere* [84] that allows users to execute arbitrary user-defined functions (UDFs) over the data managed by SDFS. *Greenplum* uses a shared-nothing massively parallel processing (MPP) architecture based on commodity hardware. The architecture also integrates MapReduce-like functionality into its platform. A similar architecture has been deployed by *Aster*, which uses an MPP-based data-warehousing appliance that supports MapReduce and targets 1 PB of data.

8.1.3.4 Databases and data-intensive computing

Traditionally, distributed databases [85] have been considered the natural evolution of database management systems as the scale of the datasets becomes unmanageable with a single system. 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. However, they are mostly concerned with datasets that can be expressed using the relational model [86], and the need to enforce ACID properties on data limits their abilities to scale as data clouds and grids do.

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

Traditionally, database management systems constituted the *de facto* storage support for several types of applications. 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 [88]. Research on databases and the data management industry are indeed at a turning point, and new opportunities arise. Some factors contributing to this change are:

- *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. The large volume of data imposes new and more efficient techniques for data management.

- *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. This massive amount of data, which is constantly mined, requires new technologies and strategies to support data analytics.
- *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. Structured data are constantly growing as a result of the continuous use of traditional enterprise applications and system, but at the same time the advances in technology and the democratization of the Internet as a platform where everyone can pull information has created a massive amount of information that is unstructured and does not naturally fit into the relational model.
- *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. It is no longer rare to build software applications and services that are dynamically deployed on hundreds or thousands of nodes, which might belong to the system for a few hours or days. Classical database infrastructures are not designed to provide support to such a volatile environment.

All these factors identify the need for new data management technologies. This not only implies a new research agenda in database technologies and a more holistic approach to the management of information but also leaves room for alternatives (or complements) to the relational model. In particular, advances in distributed file systems for the management of raw data in the form of files, distributed object stores, and the spread of the NoSQL movement constitute the major directions toward support for data-intensive computing.

8.2.1.1 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. Among the several implementations of file systems, few of them specifically address the management of huge quantities of data on a large number of nodes. Mostly these file systems constitute the data storage support for large computing clusters, supercomputers, massively parallel architectures, and lately, storage/computing clouds.

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, including the one rated the most powerful supercomputer in the June 2012 list.[6] 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. Users access the file system via a POSIX-compliant client, which can be either mounted as a module in the

[6]Top 500 supercomputers list: www.top500.org (accessed in June 2012).

kernel or through a library. The file system implements a robust failover strategy and recovery mechanism, making server failures and recoveries transparent to clients.

IBM General Parallel File System (GPFS). GPFS [88] 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. By means of this infrastructure, the system is able to support petabytes of storage, which is accessed at a high throughput and without losing consistency of data. Compared to other implementations, GPFS distributes the metadata of the entire file system and provides transparent access to it, thus eliminating a single point of failure.

Google File System (GFS). GFS [54] is the storage infrastructure that supports the execution of distributed applications in Google's computing cloud. The system has been designed to be a fault-tolerant, highly available, distributed file system built on commodity hardware and standard Linux operating systems. Rather than a generic implementation of a distributed file system, GFS specifically addresses Google's needs in terms of distributed storage for applications, and it has been designed with the following assumptions:

- The system is built on top of commodity hardware that often fails.
- 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.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads.
- The workloads also have many large, sequential writes that append data to files.
- 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. A file is a collection of chunks for which the size can be configured at file system level. Chunks are replicated on multiple nodes in order to tolerate failures. Clients look up the master server and identify the specific chunk of a file they want to access. Once the chunk is identified, the interaction happens between the client and the chunk server. Applications interact through the file system with a specific interface supporting the usual operations for file creation, deletion, read, and write. The interface also supports *snapshots* and *record append* operations that are frequently performed by applications. GFS has been conceived by considering that failures in a large distributed infrastructure are common rather than a rarity; therefore, specific attention has been given to implementing a highly available, lightweight, and fault-tolerant infrastructure. The potential single point of failure of the single-master architecture has been addressed by giving the possibility of replicating the master node on any other node belonging to the infrastructure. Moreover, a stateless daemon and extensive logging capabilities facilitate the system's recovery from failures.

Sector. Sector [84] 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. Compared to other file systems, Sector does not partition a file into blocks but replicates the entire files on multiple nodes, allowing users to customize the replication strategy for better performance. 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 [89], which is a lightweight connection-oriented protocol optimized for wide-area networks.

Amazon Simple Storage Service (S3). Amazon S3 is the online storage service provided by Amazon. Even though its internal details are not revealed, the system is claimed to support high availability, reliability, scalability, infinite storage, and low latency at commodity cost. 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. Because of the use of HTTP, there is no need for any specific library for accessing the storage system, the objects of which can also be retrieved through the Bit Torrent protocol.[7] Despite its simple semantics, a POSIX-like client library has been developed to mount S3 buckets as part of the local file system. Besides the minimal semantics, security is another limitation of S3. The visibility and accessibility of objects are linked to AWS accounts, and the owner of a bucket can decide to make it visible to other accounts or the public. It is also possible to define authenticated URLs, which provide public access to anyone for a limited (and configurable) period of time.

Besides these examples of storage systems, there exist other implementations of distributed file systems and storage clouds that have architecture that is similar to the models discussed here. Except for the S3 service, it is possible to sketch a general reference architecture in all the systems presented that identifies two major roles into which all the nodes can be classified. Metadata or master nodes contain the information about the location of files or file chunks, whereas slave nodes are used to provide direct access to the storage space. The architecture is completed by client libraries, which provide a simple interface for accessing the file system, which is to some extent or completely compliant to the POSIX specification. Variations of the reference architecture can include the ability to support multiple masters, to distribute the metadata over multiple nodes, or to easily interchange the role of nodes. The most important aspect common to all these different implementations is the ability to provide fault-tolerant and highly available storage systems.

8.2.1.2 NoSQL systems

The term *Not Only SQL (NoSQL)* was originally coined in 1998 to identify a relational database that did not expose a SQL interface to manipulate and query data but relied on a set of UNIX shell scripts and commands to operate on text files containing the actual data. In a very strict sense, NoSQL cannot be considered a relational database since it is not a monolithic piece of software organizing information according to the relational model, but rather is a collection of scripts that

[7] Bit Torrent is a P2P file-sharing protocol used to distribute large amounts of data. The key characteristic of the protocol is the ability to allow users to download a file in parallel from multiple hosts.

allow users to manage most of the simplest and more common database tasks by using text files as information stores. Later, in 2009, the term *NoSQL* was reintroduced with the intent of labeling all those database management systems that did not use a relational model but provided simpler and faster alternatives for data manipulation. Nowadays, the term *NoSQL* is a big umbrella encompassing all the storage and database management systems that differ in some way from the relational model. Their general philosophy is to overcome the restrictions imposed by the relational model and to provide more efficient systems. This often implies the use of tables without fixed schemas to accommodate a larger range of data types or avoid joins to increase the performance and scale horizontally.

Two main factors have determined the growth of the NoSQL movement: in many cases simple data models are enough to represent the information used by applications, and the quantity of information contained in unstructured formats has grown considerably in the last decade. These two factors made software engineers look to alternatives that were more suitable to specific application domains they were working on. As a result, several different initiatives explored the use of nonrelational storage systems, which considerably differ from each other. A broad classification is reported by Wikipedia,[8] which distinguishes NoSQL implementations into:

- *Document stores* (Apache Jackrabbit, Apache CouchDB, SimpleDB, Terrastore).
- *Graphs* (AllegroGraph, Neo4j, FlockDB, Cerebrum).
- *Key-value stores*. This is a macro classification that is further categorized into key-value stores on disk, key-value caches in RAM, hierarchically key-value stores, eventually consistent key-value stores, and ordered key-value store.
- *Multivalue databases* (OpenQM, Rocket U2, OpenInsight).
- *Object databases* (ObjectStore, JADE, ZODB).
- *Tabular stores* (Google BigTable, Hadoop HBase, Hypertable).
- *Tuple stores* (Apache River).

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

Apache CouchDB and MongoDB. Apache CouchDB [91] and MongoDB [90] 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. From an infrastructure point of view, the two systems support data replication and high availability. CouchDB ensures ACID properties on data. MongoDB supports *sharding*, which is the ability to distribute the content of a collection among different nodes.

Amazon Dynamo. Dynamo [92] is the distributed key-value store that supports the management of information of several of the business services offered by Amazon Inc. 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

[8] http://en.wikipedia.com/wiki/NoSQL.

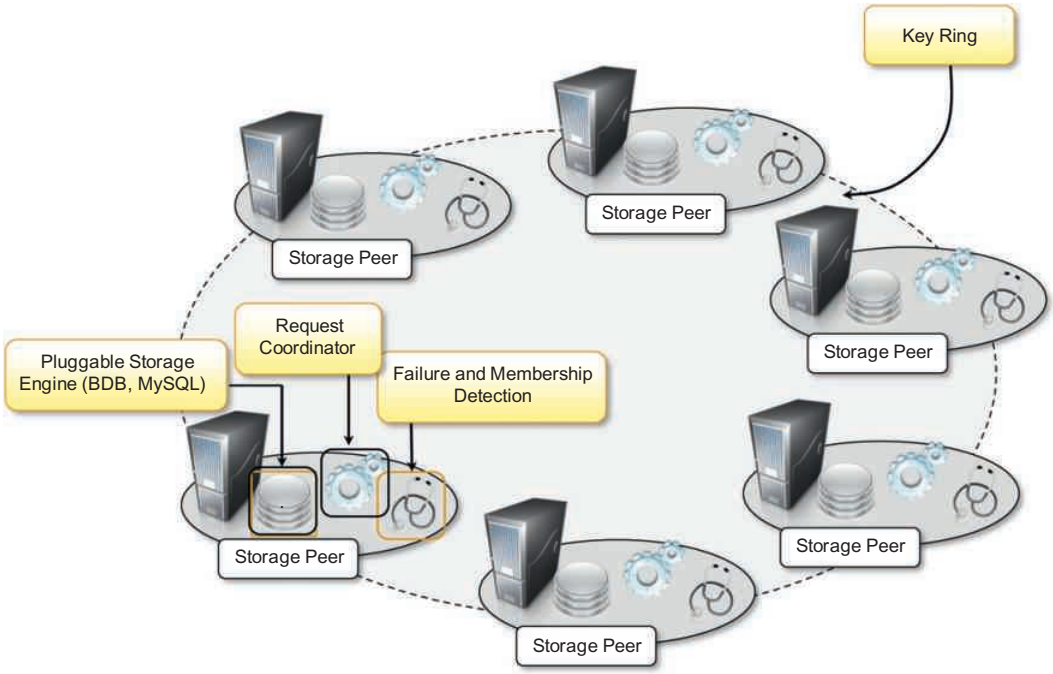


FIGURE 8.3

Amazon Dynamo architecture.

interface based on *get/put* semantics, where objects are stored and retrieved with a unique identifier (key). The main goal of achieving an extremely reliable infrastructure has imposed some constraints on the properties of these systems. For example, ACID properties on data have been sacrificed in favor of a more reliable and efficient infrastructure. This creates what it is called an *eventually consistent* model, which means that in the long term all the users will see the same data.

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. Furthermore, each node provides facilities for distributing the updates among the rings and to detect failures and unreachable nodes. With some relaxation of the consistency model applied to replicas and the use of object versioning, Dynamo implements the capability of being an *always-writable store*, where consistency of data is resolved in the background. The downside of such an approach is the simplicity of the storage model, which requires applications to build their own data models on top of the simple building blocks provided by the store. For example, there are no referential integrity constraints, relationships are not embedded in the storage model, and therefore join operations are not supported. These restrictions are not prohibitive in the case of Amazon services for which the single key-value model is acceptable.

Google Bigtable. Bigtable [93] is the distributed storage system designed to scale up to petabytes of data across thousands of servers. 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. A simple data access model constitutes the interface for client applications that can address data at the granularity level of the single column of a row. Moreover, each column value is stored in multiple versions that can be automatically time-stamped by Bigtable or by the client applications.

Besides the basic data access, Bigtable APIs also allow more complex operations such as single row transactions and advanced data manipulation by means of the Sazwall[9] [95] scripting language or the MapReduce APIs.

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.

Chubby [96]—a distributed, highly available, and persistent lock service—supports the activity of the master and tablet servers. System monitoring and data access are filtered through Chubby, which is also responsible for managing replicas and providing consistency among them. At the very bottom layer, the data are stored in the Google File System in the form of files, and all the update operations are logged into the file for the easy recovery of data in case of failures or when tablets need to be reassigned to other servers. Bigtable uses a specific file format for storing the data of a tablet, which can be compressed for optimizing the access and storage of data.

Bigtable is the result of a study of the requirements of several distributed applications in Google. It serves as a storage back-end for 60 applications (such as Google Personalized Search, Google Anlytics, Google Finance, and Google Earth) and manages petabytes of data.

Apache Cassandra. Cassandra [94] is a distributed object store for managing large amounts of structured data spread across many commodity servers. 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. Cassandra is defined as a

[9] *Sazwall* is an interpreted procedural programming language developed at Google for the manipulation of large quantities of tabular data. It includes specific capabilities for supporting statistical aggregation of values read or computed from the input and other features that simplify the parallel processing of petabytes of data.

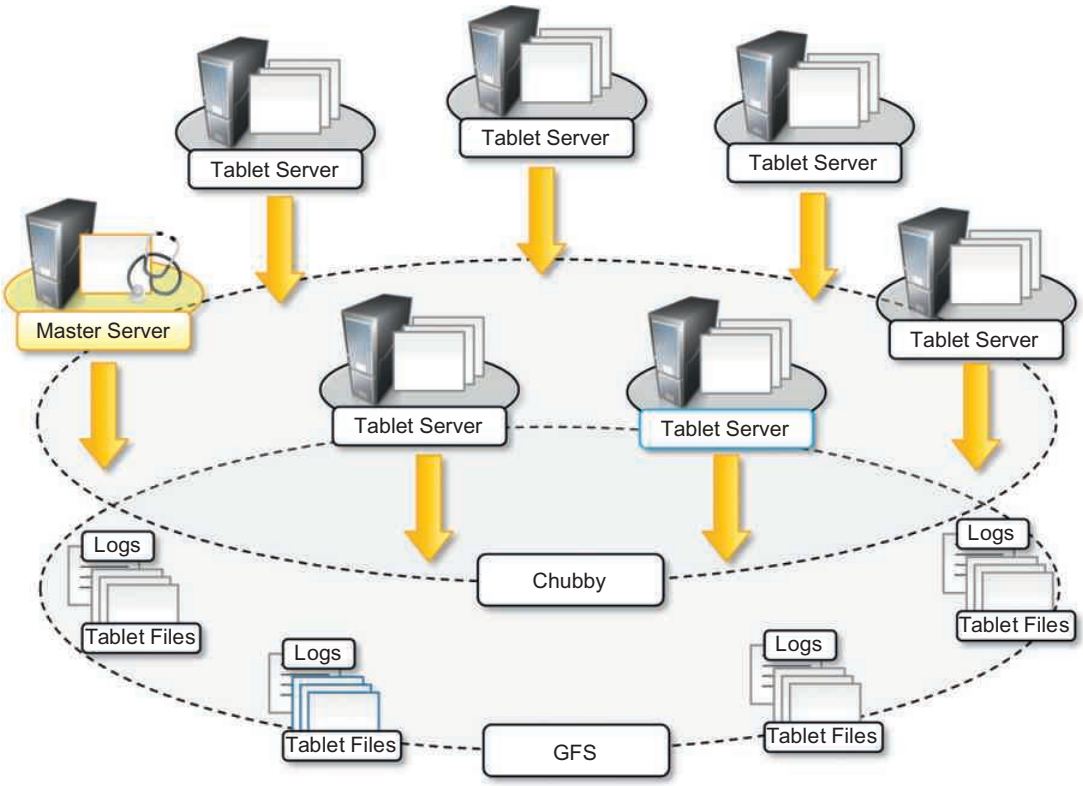


FIGURE 8.4

Bigtable architecture.

second-generation distributed database that builds on the concept of Amazon Dynamo, which follows a fully distributed design, and Google Bigtable, from which it inherits the "column family" concept. 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.

In terms of the infrastructure, Cassandra is very similar to Dynamo. It has been designed for incremental scaling, and it organizes the collection of nodes sharing a key space into a ring. Each node manages multiple and discontinuous portions of the key space and replicates its data up to N other nodes. Replication uses different strategies; it can be *rack aware*, *data center aware*, or *rack unaware*, meaning that the policies can take into account whether the replication needs to be made within the same cluster or datacenter or not to consider the geo-location of nodes. As in Dynamo,

node membership information is based on gossip protocols.[10] Cassandra makes also use of this information diffusion mode for other tasks, such as disseminating the system control state. The local file system of each node is used for data persistence, and Cassandra makes extensive use of commit logs, which makes the system able to recover from transient failures. Each write operation is applied in memory only after it has been logged on disk so that it can be easily reproduced in case of failures. When the data in memory trespasses a specified size, it is dumped to disk. Read operations are performed in-memory first and then on disk. To speed up the process, each file includes a summary of the keys it contains so that it is possible to avoid unnecessary file scanning to search for a key.

As noted earlier, Cassandra builds on the concepts designed in Dynamo and Bigtable and puts them together to achieve a completely distributed and highly reliable storage system. The largest Cassandra deployment to our knowledge manages 100 TB of data distributed over a cluster of 150 machines.

Hadoop HBase. HBase is the distributed database that supports the storage needs of the Hadoop distributed programming platform. 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), which mimics the structure and services of GFS.

In this section, we discussed the storage solutions that support the management of data-intensive applications, especially those referred as *Big Data*. Traditionally, database systems, most likely based on the relational model, have been the primary solution for handling large quantities of data. As we discussed, when it comes to extremely huge quantities of unstructured data, relational databases become impractical and provide poor performance. Alternative and more effective solutions have significantly reviewed the fundamental concepts at the base of distributed file systems and storage systems. The next level comprises providing programming platforms that, by leveraging the discussed storage systems, can capitalize on developers' efforts to handle massive amounts of data. Among them, MapReduce and all its variations play a fundamental role.

8.2.2 Programming platforms

Platforms for programming data-intensive applications provide abstractions that help express the computation over a large quantity of information and runtime systems able to efficiently manage huge volumes of data. Traditionally, database management systems based on the relational model have been used to express the structure and connections between the entities of a data model. This approach has proven unsuccessful in the case of Big Data, where information is mostly found unstructured or semistructured and where data are most likely to be organized in files of large size or a huge number of medium-sized files rather than rows in a database. Distributed workflows have often been used to analyze and process large amounts of data [66,67]. This approach introduced a plethora of frameworks for workflow management systems, as discussed in Section 7.2.4, which

[10] A *gossip protocol* is a style of communication protocol inspired by the form of gossip seen in social networks. Gossip protocols are used in distributed systems as an alternative to distributed and propagate information that is efficient compared to flooding or other kinds of algorithms.

eventually incorporated capabilities to leverage the elastic features offered by cloud computing [70]. These systems are fundamentally based on the abstraction of a *task*, which puts a big burden on the developer, who needs to deal with data management and, often, data transfer issues.

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 [55] 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, which extend its capabilities for wider purposes.

8.2.2.1 The MapReduce programming model

MapReduce [55] is a programming platform Google introduced for processing large quantities of data. It 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. Therefore, developers no longer have to handle these issues and are provided with an interface that presents data at a higher level: as a collection of key-value pairs. The computation of MapReduce applications is then organized into a workflow of *map* and *reduce* operations that is entirely controlled by the runtime system; developers need only specify how the *map* and *reduce* functions operate on the key-value pairs.

More precisely, 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. The MapReduce runtime might eventually aggregate the values corresponding to the same keys. These files constitute the input of reduce tasks, which finally produce output files in the form of *list(v2)*. The operation performed by reduce tasks is generally expressed

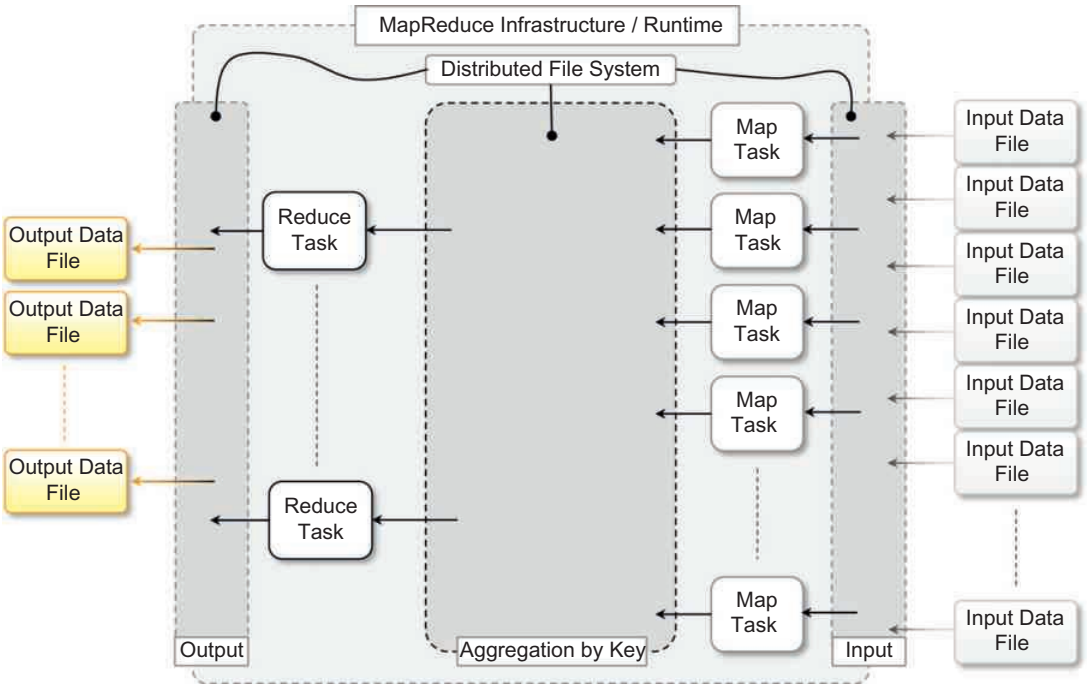


FIGURE 8.5

MapReduce computation workflow.

as an aggregation of all the values that are mapped by a specific key. The number of map and reduce tasks to create, the way files are partitioned with respect to these tasks, and the number of map tasks connected to a single reduce task are the responsibilities of the MapReduce runtime. In addition, the way files are stored and moved is the responsibility of the distributed file system that supports MapReduce.

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. This model has proven successful in the case of Google, where the majority of the information that needs to be processed is stored in textual form and is represented by Web pages or log files. Some of the examples that show the flexibility of MapReduce are the following [55]:

- *Distributed grep.* The *grep* operation, which performs the recognition of patterns within text streams, is performed across a wide set of files. MapReduce is leveraged to provide a parallel and faster execution of this operation. In this case, the input file is a plain text file, and the *map* function emits a line into the output each time it recognizes the given pattern. The reduce task aggregates all the lines emitted by the map tasks into a single file.
- *Count of URL-access frequency.* MapReduce is used to distribute the execution of Web server log parsing. In this case, the *map* function takes as input the log of a Web server and emits into the output file a key-value pair $< URL, 1 >$ for each page access recorded in the log. The

reduce function aggregates all these lines by the corresponding URL, thus summing the single accesses, and outputs a $<$*URL, total-count*$>$ pair.

- *Reverse Web-link graph.* The Reverse Web-link graph keeps track of all the possible Web pages that might lead to a given link. In this case input files are simple HTML pages that are scanned by map tasks emitting $<$*target, source*$>$ pairs for each of the links found in the Web page *source*. The reduce task will collate all the pairs that have the same target into a $<$*target, list (source)*$>$ pair. The final result is given one or more files containing these mappings.
- *Term vector per host.* A term vector recaps the most important words occurring in a set of documents in the form of *list(*$<$*word, frequency*$>$*)*, where the number of occurrences of a word is taken as a measure of its importance. MapReduce is used to provide a mapping between the origin of a set of document, obtained as the host component of the URL of a document, and the corresponding term vector. In this case, the map task creates a pair $<$*host, term-vector*$>$ for each text document retrieved, and the reduce task aggregates the term vectors corresponding to documents retrieved from the same host.
- *Inverted index.* The inverted index contains information about the presence of words in documents. This information is useful to allow fast full-text searches compared to direct document scans. In this case, the map task takes as input a document, and for each document it emits a collection of $<$*word, document-id*$>$. The *reduce* function aggregates the occurrences of the same word, producing a pair $<$*word, list(document-id)*$>$.
- *Distributed sort.* In this case, MapReduce is used to parallelize the execution of a *sort* operation over a large number of records. This application mostly relies on the properties of the MapReduce runtime, which sorts and creates partitions of the intermediate files, rather than in the operations performed in the map and reduce tasks. Indeed, these are very simple: The map task extracts the key from a record and emits a $<$*key, record*$>$ pair for each record; the reduce task will simply copy through all the pairs. The actual sorting process is performed by the MapReduce runtime, which will emit and partition the key-value pair by ordering them according to the key.

The reported example are mostly concerned with text-based processing. MapReduce can also be used, with some adaptation, to solve a wider range of problems. An interesting example is its application in the field of machine learning [97], where statistical algorithms such as *Support Vector Machines (SVM)*, *Linear Regression (LR)*, *Naïve Bayes (NB)*, and *Neural Network (NN)*, are expressed in the form of *map* and *reduce* functions. Other interesting applications can be found in the field of compute-intensive applications, such as the computation of Pi with a high degree of precision. It has been reported that the Yahoo! Hadoop cluster has been used to compute the $10^{15} + 1$ bit of Pi.[11] Hadoop is an open-source implementation of the MapReduce platform.

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

- *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.

[11]The full details of this computation can be found in the Yahoo! Developer Network blog in the following blog post: http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_computes_the_10151st_bi/.

- *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.

Adaptations to this model are mostly concerned with identifying the appropriate keys, creating reasonable keys when the original problem does not have such a model, and finding ways to partition the computation between *map* and *reduce* functions. Moreover, more complex algorithms can be decomposed into multiple MapReduce programs, where the output of one program constitutes the input of the following program.

The abstraction proposed by MapReduce provides developers with a very minimal interface that is strongly focused on the algorithm to implement rather than the infrastructure on which it is executed. This is a very effective approach, but at the same time it demands a lot of common tasks, which are of concern in the management of a distributed application to the MapReduce runtime, allowing the user to specify only configuration parameters to control the behavior of applications. These tasks are managing data transfer and scheduling map and reduce tasks over a distributed infrastructure. Figure 8.6 gives a more complete overview of a MapReduce infrastructure, according to the implementation proposed by Google [55].

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 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. In a MapReduce computation, input files are initially divided into splits (generally 16 to 64 MB) and stored in the distributed file system. The master process generates the map tasks and assigns input splits to each of them by balancing the load.

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. The locations of these pairs are then notified to the master process, which forwards this information to the reduce tasks, which are able to collect the required input via a remote procedure call in order to read from the map tasks' local storage. The key range is then sorted and all the same keys are grouped together. Finally, the reduce task is executed to produce the final output, which is stored in the global file system. This process is completely automatic; users may control it through configuration parameters that allow specifying (besides the *map* and *reduce* functions) the number of map tasks, the number of partitions into which to separate the final output, and the *partition* function for the intermediate key range.

Besides orchestrating the execution of map and reduce tasks as previously described, the MapReduce runtime ensures a reliable execution of applications by providing a fault-tolerant

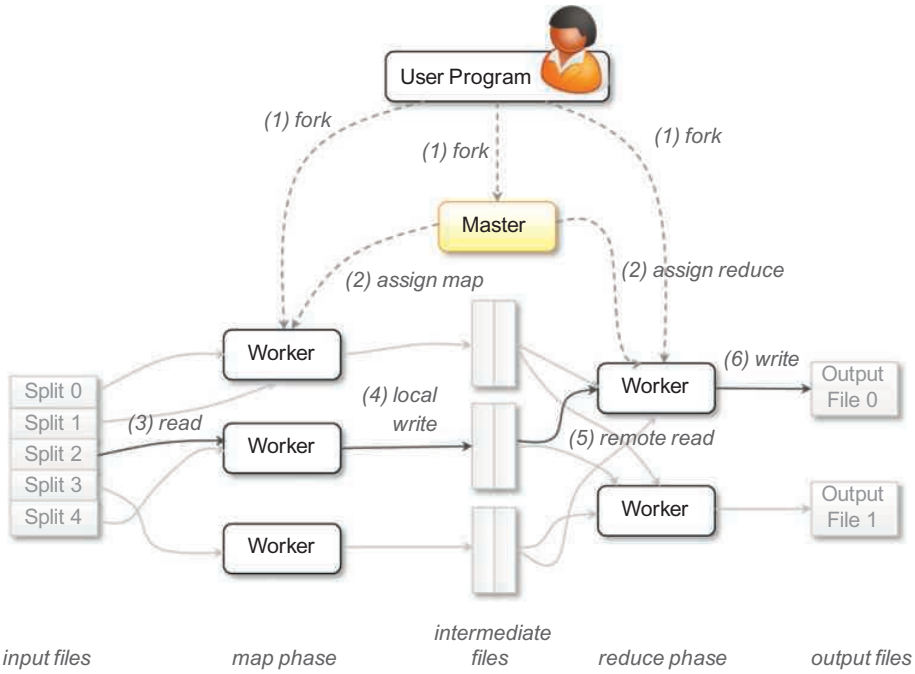


FIGURE 8.6

Google MapReduce infrastructure overview.

infrastructure. Failures of both master and worker processes are handled, as are machine failures that make intermediate outputs inaccessible. Worker failures are handled by rescheduling map tasks somewhere else. This is also the technique that is used to address machine failures since the valid intermediate output of map tasks has become inaccessible. Master process failure is instead addressed using checkpointing, which allows restarting the MapReduce job with a minimum loss of data and computation.

8.2.2.2 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. Although the model can be applied to several different problem scenarios, it still exhibits limitations, mostly due to the fact that the abstractions provided to process data are very simple, and complex problems might require considerable effort to be represented in terms of *map* and *reduce* functions only. 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. In this

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

Hadoop. Apache Hadoop [83] is a collection of software projects for reliable and scalable distributed computing. Taken together, the entire collection is an open-source implementation of the MapReduce framework supported by a GFS-like distributed file system. 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 [54]; the latter provides the same features and abstractions as Google MapReduce. Initially developed and supported by Yahoo!, Hadoop now constitutes the most mature and large data cloud application and has a very robust community of developers and users supporting it. Yahoo! now runs the world's largest Hadoop cluster, composed of 40,000 machines and more than 300,000 cores, made available to academic institutions all over the world. Besides the core projects of Hadoop, a collection of other projects related to it provides services for distributed computing.

Pig. Pig[12] 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 such as Hadoop. Developers can express their data analysis programs in a textual language called *Pig Latin*, which exposes a SQL-like interface and is characterized by major expressiveness, reduced programming effort, and a familiar interface with respect to MapReduce.

Hive. Hive[13] 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. Whereas the framework provides the same capabilities as a classical data warehouse, it does not exhibit the same performance, especially in terms of query latency, and for this reason does not constitute a valid solution for online transaction processing. 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.

Map-Reduce-Merge. Map-Reduce-Merge [98] 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.

Twister. Twister [99] 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`

[12] http://pig.apache.org/.
[13] http://hive.apache.org/.

3. `While Condition Holds True Do`
   a. `Run MapReduce`
   b. `Apply Combine Operation to Result`
   c. `Update Condition`
4. `Close`

Besides the iterative MapReduce computation, 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; and other tools for management of data.

8.2.2.3 Alternatives to MapReduce

MapReduce, other abstractions provide support for processing large datasets and execute data-intensive workloads. To different extents, these alternatives exhibit some similarities to the MapReduce approach.

Sphere. Sphere [84] is the distributed processing engine that leverages the Sector Distributed File System (SDFS). Rather than being a variation of MapReduce, 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. A specific combination of UDFs allows Sphere to express MapReduce computations. Sphere strongly leverages the Sector distributed file systems, and it 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 collective execution of UDFs is achieved through the distributed execution of *Sphere Process Engines (SPEs)*, which are assigned with a given stream segment. The execution model is a master-slave model that is client controlled; a 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 and orchestrate the entire distributed execution.

All-Pairs. All-Pairs [100] is an abstraction and a runtime environment for the optimized execution of data-intensive workloads. It provides a simple abstraction—in terms of the *All-pairs* function—that is common in many scientific computing domains:

$$All\text{-}pairs(A{:}set, B{:}set, F{:}function) \rightarrow M{:}matrix$$

Examples of problems that can be represented in this model can be found in the field of biometrics, where similarity matrices are composed as a result of the comparison of several images that contain subject pictures. Another example is several applications and algorithms in data mining. The model expressed by 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`

This implementation is quite naïve and produces poor performance in general. Moreover, other problems, such as data distribution, dispatch latency, number of available compute nodes, and probability of failure, are not handled specifically. The All-Pairs model tries to address these issues by

introducing a specification for the nature of the problem and an engine that, according to this specification, optimizes the distribution of tasks over a conventional cluster or grid infrastructure. 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. The interesting aspect of this model is mostly concentrated on the first two phases, where the performance model of the system is built and the data are opportunistically distributed in order to create the optimal number of tasks to assign to each node and optimize the utilization of the infrastructure.

DryadLINQ. Dryad [101] 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. Dryad's aim is to provide an infrastructure for automatically parallelizing the execution of applications without requiring the developer to know about distributed and parallel programming.

In Dryad, developers can express distributed applications as a set of sequential programs that are connected by means of channels. More precisely, a Dryad computation can be expressed in terms of a directed acyclic graph in which nodes are the sequential programs and vertices represent the channels connecting such programs. Because of this structure, Dryad is considered a superset of the MapReduce model, since its general application model allows expressing graphs representing MapReduce computation as well. An interesting feature exposed by Dryad is the capability of supporting dynamic modification of the graph (to some extent) and of partitioning, if possible, the execution of the graph into stages. This infrastructure is used to serve different applications and tools for parallel programming. Among them, DryadLINQ [102] is a programming environment that produces Dryad computations from the Language Integrated Query (LINQ) extensions to C# [103]. The resulting framework provides a solution that is completely integrated into the .NET framework and able to express several distributed computing models, including MapReduce.

8.3 Aneka MapReduce programming

Aneka provides an implementation of the MapReduce abstractions by following the reference model introduced by Google and implemented by Hadoop. MapReduce is supported as one of the available programming models that can be used to develop distributed applications.

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. A MapReduce job in Google MapReduce or Hadoop corresponds to the execution of a MapReduce application 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. The runtime support is composed of three main elements:

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

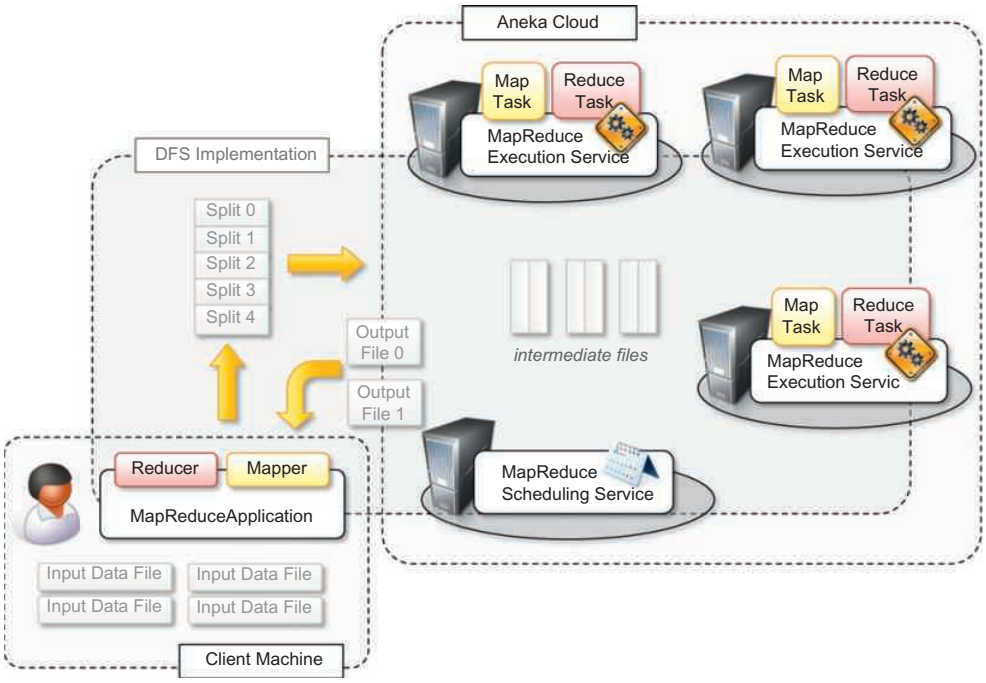


FIGURE 8.7

Aneka MapReduce infrastructure.

Client components, namely the *MapReduceApplication*, 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.

8.3.1.1 Programming abstractions

Aneka executes any piece of user code within the context of a distributed application. This approach is maintained even in the MapReduce programming model, where there is a natural mapping between the concept of a MapReduce job—used in Google MapReduce and Hadoop—and the Aneka application concept. Unlike other programming models, the task creation is not the responsibility of the user but 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—*MapReduceApplication*—that better supports the 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. Template specialization is used to keep track of keys and values types on which these two functions operate. Generics provide a more natural approach in terms of object manipulation from within the *map* and *reduce* methods and simplify the programming by removing the necessity of casting and other type check operations. The submission and execution of a MapReduce job is performed through the class *MapReduceApplication<M,R>*, which provides the interface to the Aneka Cloud to support the 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. To implement a specific mapper, it is necessary to inherit this class and provide actual types for key *K* and the value *V*. The *map* operation is implemented by overriding the abstract method *void Map(IMapInput<K,V> input)*, while the other methods are internally used by the framework. *IMapInput<K,V>* provides access to the input key-value pair on which the *map* operation is performed.

Listing 8.2 shows the implementation of the *Mapper<K,V>* component for the Word Counter sample. This sample counts the 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. Therefore, the mapper is specialized by using a *long integer* as the key type and a *string* for the value. To count the frequency of words, the *map* function will emit a new key-value pair for each word contained in the line by using the word as the key and the number 1 as the value. This implementation will emit two pairs for the same word if the word occurs twice in the line. It will be the responsibility of the reducer to appropriately sum all these occurrences.

Listing 8.3 shows the definition of the *Reducer<K,V>* class. The implementation of a specific reducer requires specializing the generic class and overriding the abstract method: *Reduce (IReduceInputEnumerator<V> input)*. Since the *reduce* operation is applied to a collection of values that are mapped to the same key, the *IReduceInputEnumerator<V>* allows developers to iterate over such collections. Listing 8.4 shows how to implement the reducer function for the word-counter example.

In this case the *Reducer<K,V>* class is specialized using a *string* as a key type and an *integer* as a value. The reducer simply iterates over all the values that are accessible through the enumerator and sums them. Once the iteration is completed, the sum is dumped to file.

It is important to note that there is a link between the types used to specialize the mapper and those used to specialize the reducer. The key and value types used in the reducer are those defining the key-value pair emitted by the mapper. In this case the mapper generates a key-value pair *(string,int)*; hence the reducer is of type *Reducer<string,int>*.

The *Mapper<K,V>* and *Reducer<K,V>* classes provide facilities for defining the computation performed by a MapReduce job. To submit, execute, and monitor its progress, Aneka provides

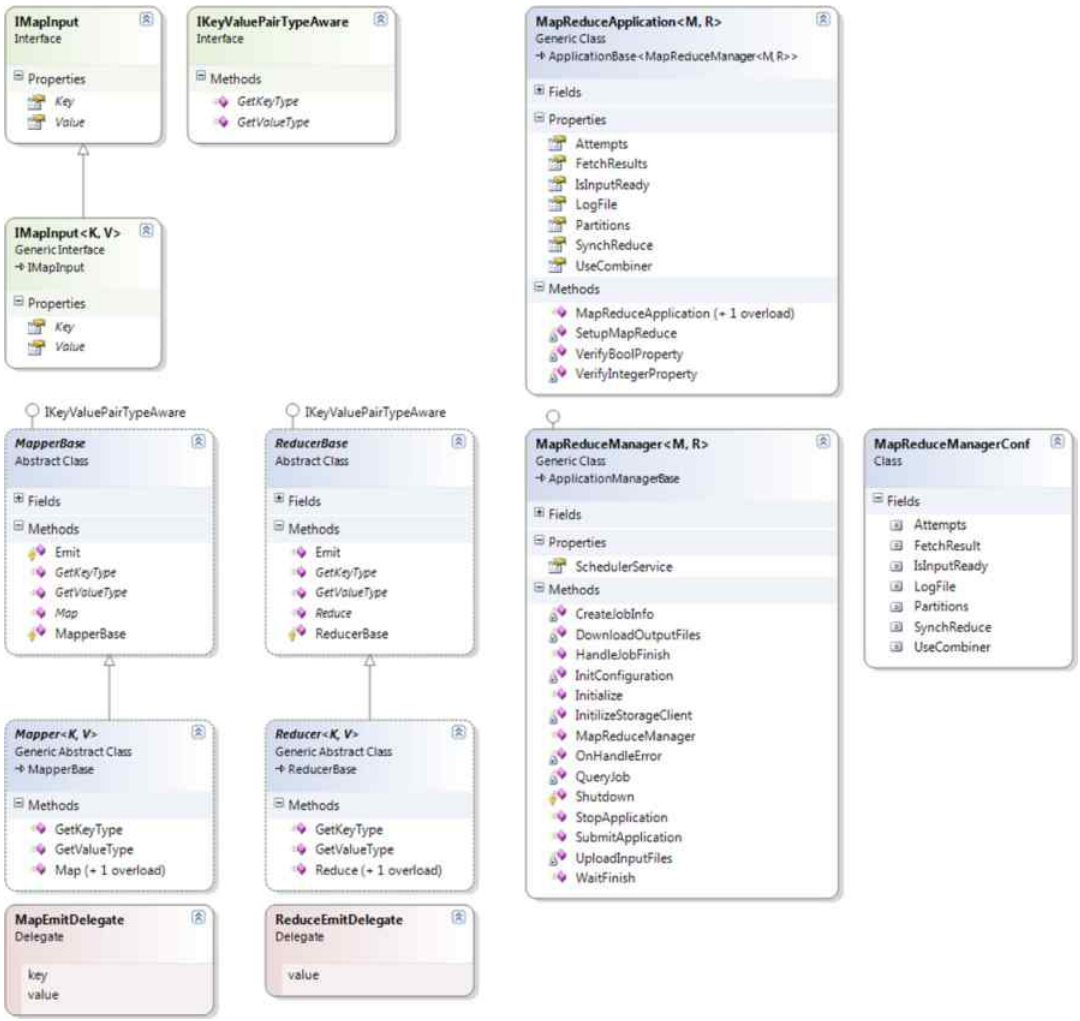


FIGURE 8.8

MapReduce Abstractions Object Model.

the *MapReduceApplication* $< M,R >$ class. As happens for the other programming models introduced in this book, this class represents the local view of distributed applications on top of Aneka. Due to the simplicity of the MapReduce model, such class provides limited facilities that are mostly concerned with starting the MapReduce job and waiting for its completion. Listing 8.5 shows the interface of *MapReduceApplication* $< M,R >$.

The interface of the class exhibits only the MapReduce-specific settings, whereas the control logic is encapsulated in the *ApplicationBase* $< M >$ class. From this class it is possible set the

```
using Aneka.MapReduce.Internal;

namespaceAneka.MapReduce
{
    /// <summary>
    /// Interface IMapInput<K,V>. Extends IMapInput and provides a strongly-
    /// typed version of the extended interface.
    /// </summary>
    public interface IMapInput<K,V>: IMapInput
    {
        /// <summary>
        /// Property <i>Key</i> returns the key of key/value pair.
        /// </summary>
        K Key { get; }
        /// <summary>
        /// Property <i>Value</i> returns the value of key/value pair.
        /// </summary>
        V Value { get; }
    }

    /// <summary>
    /// Delegate MapEmitDelegate. Defines the signature of a method
    /// that is used to doEmit intermediate results generated by the mapper.
    /// </summary>
    /// <param name="key">The <i>key</i> of the <i>key-value</i> pair.</param>
    /// <param name="value">The <i>value</i> of the <i>key-value</i> pair.</param>
    public delegate void MapEmitDelegate(object key, object value);

    /// <summary>
    /// 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. The definition of a specific mapper class only implies the
    /// implementation of the Mapper<K,V>.Map(IMapInput<K,V>) method.
    /// </summary>
    public abstract class Mapper<K,V> : MapperBase
    {
        /// <summary>
        /// Emits the intermediate result source by using doEmit.
        /// </summary>
        /// <param name="source">An instance implementing IMapInput containing the
        /// <i>key-value</i> pair representing the intermediate result.</param>
        /// <param name="doEmit">A MapEmitDelegate instance that is used to write to the
        /// output stream the information about the output of the Map operation.</param>
        public void Map(IMapInput input, MapEmitDelegate emit) { … }
        /// <summary>
        /// Gets the type of the <i>key</i> component of a <i>key-value</i> pair.
        /// </summary>
        /// <returns>A Type instance containing the metadata about the type of the
        /// <i>key</i>.</returns>
        public override Type GetKeyType(){ return typeof(K); }
```

LISTING 8.1

Map Function APIs.

```
        /// <summary>
        /// Gets the type of the <i>value</i> component of a <i>key-value</i> pair.
        /// </summary>
        /// <returns>A Type instance containing the metadata about the type of the
        /// <i>value</i>.</returns>
        public overrideType GetValueType(){ return typeof(V); }

        #region Template Methods
        /// <summary>
        /// Function Map is overrided by users to define a map function.
        /// </summary>
        /// <param name="source">The source of Map function is IMapInput, which contains
        /// a key/value pair.</param>
        protected abstract void Map(IMapInput<K, V> input);
        #endregion
    }
}
```

LISTING 8.1

(Continued)

behavior of MapReduce for the current execution. The parameters that can be controlled are the following:

- *Partitions.* This property stores an integer number containing the number of partitions into which to divide the final results. This value also determines the number of reducer tasks that will be created by the runtime infrastructure. The default value is 10.
- *Attempts.* This property contains the number of times that the runtime will retry to execute a task before declaring it failed. The default value is 3.
- *UseCombiner.* This property stores a Boolean value that indicates whether the MapReduce runtime should add a combiner phase to the map task execution in order to reduce the number of intermediate files that are passed to the reduce task. The default value is set to *true*.
- *SynchReduce.* This property stores a Boolean value that indicates whether to synchronize the reducers or not. The default value is set to *true* and currently is not used to determine the behavior of MapReduce.
- *IsInputReady.* This is a Boolean property that indicates whether the input files are already stored in the distributed file system or must be uploaded by the client manager before the job can be executed. The default value is set to *false*.
- *FetchResults.* This is a Boolean property that indicates whether the client manager needs to download to the local computer the result files produced by the execution of the job. The default value is set to *true*.
- *LogFile.* This property contains a string defining the name of the log file used to store the performance statistics recorded during the execution of the MapReduce job. The default value is *mapreduce.log*.

The core control logic governing the execution of a MapReduce job resides within the *MapReduceApplicationManager*$<M,R>$, which interacts with the MapReduce runtime.

```
using Aneka.MapReduce;

namespace Aneka.MapReduce.Examples.WordCounter
{
    /// <summary>
    /// Class WordCounterMapper. Extends Mapper<K,V> and provides an
    /// implementation of the map function for the Word Counter sample. This mapper
    /// emits a key-value pair (word,1) for each word encountered in the input line.
    /// </summary>
    public class WordCounterMapper: Mapper<long,string>
    {
        /// <summary>
        /// Reads the source and splits into words. For each of the words found
        /// emits the word as a key with a vaue of 1.
        /// </summary>
        /// <param name="source">map source</param>
        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.

Developers can control the application by using the methods and the properties exposed by the *ApplicationBase < M >* class. Listing 8.6 displays the collection of methods that are of interest in this class for the execution of MapReduce jobs.

Besides the constructors and the common properties that are of interest for all the applications, the two methods in bold in Listing 8.6 are those that are most commonly used to execute MapReduce jobs. These are two different overloads of the *InvokeAndWait* method: the first one simply starts the execution of the MapReduce job and returns upon its completion; the second one executes a client-supplied callback at the end of the execution. The use of *InvokeAndWait* is blocking; therefore, it is not possible to stop the application by calling *StopExecution* within the same thread. If it is necessary to implement a more sophisticated management of the MapReduce job, it

```
using Aneka.MapReduce.Internal;
namespace Aneka.MapReduce
{
    /// <summary>
    /// Delegate ReduceEmitDelegate. Defines the signature of a method
    /// that is used to emit aggregated value of a collection of values matching the
    /// same key and that is generated by a reducer.
    /// </summary>
    /// <param name="value">The <i>value</i> of the <i>key-value</i> pair.</param>
    public delegate void ReduceEmitDelegate(object value);

    /// <summary>
    /// Class <i>Reducer</i>. Extends the ReducerBase class and provides an

    /// implementation of the common operations that are expected from a <i>Reducer</i>.
    /// In order to define reducer for specific applications developers have to extend
    /// implementation of the Reduce(IReduceInputEnumerator<V>) method that reduces a
    /// this class and provide an collection of <i>key-value</i> pairs as described by
    /// the <i>map-reduce</i> model.
    /// </summary>
    public abstract class Reducer<K,V> : ReducerBase
    {
        /// <summary>
        /// Performs the <i>reduce</i> phase of the <i>map-reduce</i> model.
        /// </summary>
        /// <param name="source">An instance of IReduceInputEnumerator allowing to
        /// iterate over the collection of values that have the same key and will be
        /// aggregated.</param>
        /// <param name="emit">An instance of the ReduceEmitDelegate that is used to
        /// write to the output stream the aggregated value.</param>
        public void Reduce(IReduceInputEnumerator input, ReduceEmitDelegate emit) { … }
        /// <summary>
        /// Gets the type of the <i>key</i> component of a <i>key-value</i> pair.
        /// </summary>
        /// <returns>A Type instance containing the metadata about the type of the
        /// <i>key</i>.</returns>
        public override Type GetKeyType(){return typeof(K);}
        /// <summary>
        /// Gets the type of the <i>value</i> component of a <i>key-value</i> pair.
        /// </summary>
        /// <returns>A Type instance containing the metadata about the type of the
        /// <i>value</i>.</returns>
        public override Type GetValueType(){return typeof(V);}

        #region Template Methods
        /// <summary>
        /// Recuces the collection of values that are exposed by
        /// <paramref name="source"/> into a single value. This method implements the
        /// <i>aggregation</i> phase of the <i>map-reduce</i> model, where multiple
        /// values matching the same key are composed together to generate a single
        /// value.
        /// </summary>
        /// <param name="source">AnIReduceInputEnumerator<V> instancethat allows to
        /// iterate over all the values associated with same key.</param>
        protected abstract void Reduce(IReduceInputEnumerator<V> input);
        #endregion
    }
}
```

LISTING 8.3

Reduce Function APIs.

```
using Aneka.MapReduce;

namespace Aneka.MapReduce.Examples.WordCounter
{
    /// <summary>
    /// Class <b><i>WordCounterReducer</i></b>. Reducer implementation for the Word
    /// Counter application. The Reduce method iterates all over values of the
    /// enumerator and sums the values before emitting the sum to the output file.
    /// </summary>
    public class WordCounterReducer: Reducer<string,int>
    {
        /// <summary>
        /// Iterates all over the values of the enumerator and sums up
        /// all the values before emitting the sum to the output file.
        /// </summary>
        /// <param name="source">reduce source</param>
        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.

is possible to use the *SubmitExecution* method, which submits the execution of the application and returns without waiting for its completion.

In terms of management of files, the MapReduce implementation will automatically upload all the files that are found in the *Configuration.Workspace* directory and will ignore the files added by the *AddSharedFile* methods.

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

The lines of interest are those put in evidence in the *try { . . . } catch { . . . } finally { . . .}* block. As shown, the execution of a MapReduce job requires only three lines of code, where the user reads the configuration file, creates a *MapReduceApplication<M,R>* instance and configures it, and then starts the execution. All the rest of the code is mostly concerned with setting up the logging and handling exceptions.

8.3.1.2 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*

```
using Aneka.MapReduce.Internal;

namespace Aneka.MapReduce
{
    /// <summary>
    /// 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. A MapReduceApplication is
    /// a generic type that is parameterized with a specific type of MapperBase and a
    /// specific type of ReducerBase. It controls the execution of the application and
    /// it is in charge of collecting the results or resubmitting the failed tasks.
    /// </summary>
    /// <typeparam name="M">Placeholder for the mapper type.</typeparam>
    /// <typeparam name="R">Placeholder for the reducer type.</typeparam>
    public class MapReduceApplication<M, R> : ApplicationBase<MapReduceManager<M, R>>
           where M: MapReduce.Internal.MapperBase
           where R: MapReduce.Internal.ReducerBase
    {
        /// <summary>
        /// Default value for the Attempts property.
        /// </summary>
        public const intDefaultRetry = 3;
        /// <summary>
        /// Default value for the Partitions property.
        /// </summary>
        public const intDefaultPartitions = 10;
        /// <summary>
        /// Default value for the LogFile property.
        /// </summary>
        public const stringDefaultLogFile = "mapreduce.log";

        /// <summary>
        /// List containing the result files identifiers.
        /// </summary>
        private List<string>resultFiles = new List<string>();
        /// <summary>
        /// Property group containing the settings for the MapReduce application.
        /// </summary>
        private PropertyGroupmapReduceSetup;

        /// <summary>
        /// Gets, sets an integer representing the number of partions for the key space.
        /// </summary>
        public int Partitions { get { … } set { … } }
        /// <summary>
        /// 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.
        /// </summary>
        public bool UseCombiner { get { … } set { … } }
        /// <summary>
        /// Gets, sets an boolean indicating whether to synchronize the reduce phase.
        /// </summary>
        public bool SynchReduce { get { … } set { … } }
```

LISTING 8.5

MapReduceApplication < M,R >.

```
        /// <summary>
        /// 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.
        /// </summary>
        public bool IsInputReady { get { … } set { … } }
        /// <summary>
        /// Gets, sets the number of attempts that to run failed tasks.
        /// </summary>
        public int Attempts { get { … } set { … } }
        /// <summary>
        /// Gets or sets a string value containing the path for the log file.
        /// </summary>
        public string LogFile { get { … } set { … } }
        /// <summary>
        /// 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.
        /// </summary>
        public bool FetchResults { get { … } set { … } }

        /// <summary>
        /// Creates a MapReduceApplication<M,R> instance and configures it with
           /// the given configuration.
        /// </summary>
        /// <param name="configuration">A Configuration instance containing the
        /// information that customizes the execution of the application.</param>
        public MapReduceApplication(Configurationconfiguration) :
                                    base("MapReduceApplication", configuration){ … }
        /// <summary>
        /// Creates MapReduceApplication<M,R> instance and configures it with
        /// the given configuration.
        /// </summary>
        /// <param name="displayName">A string containing the friendly name of the
        /// application.</param>
        /// <param name="configuration">A Configuration instance containing the
        /// information that customizes the execution of the application.</param>
        public MapReduceApplication(string displayName, Configuration configuration) :
                                        base(displayName, configuration) { … }

        // here follows the private implementation…
    }
}
```

LISTING 8.5

(Continued)

Service and the *MapReduce Execution Service*. These two services integrate with the existing services of the framework in order to provide persistence, application accounting, and the features available for the applications developed with other programming models.

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*. The former is a wrapper around the scheduler, implementing the interfaces Aneka requires to expose a software

```
Namespace Aneka.Entity
{
    /// <summary>
    /// Class <b><i>ApplicationBase<M></i></b>. Defines the base class for the
    /// application instances for all the programming model supported by Aneka.
    /// </summary>
    public class ApplicationBase<M> where M : IApplicationManager, new()
    {
        /// <summary>
        /// Gets the application unique identifier attached to this instance. The
        /// application unique identifier is the textual representation of a System.Guid
        /// instance, therefore is a globally unique identfier. This identifier is
        /// automatically created when a new instance of an application is created.
        /// </summary>
        public string Id { get { … } }
        /// <summary>
        /// Gets the unique home directory for the AnekaApplication<W,M>.
        /// </summary>
        public string Home { get { … } }
        /// <summary>
        /// Gets the current state of the application.
        /// </summary>
        public ApplicationState State{get{ … }}
        /// <summary>
        /// Gets a boolean value indicating whether the application is terminated.
        /// </summary>
        public bool Finished { get { … } }

        /// <summary>
        /// Gets the underlying IApplicationManager that ismanaging the execution of the
        /// application instanceon the client side.
        /// </summary>
        public M ApplicationManager { get { … } }

        /// <summary>
        /// Gets, sets the application display name. This is a friendly name which is
            /// to identify an application by means of a textual and human intelligible
            /// sequence of characters, but it is NOT a unique identifier and no check about
            /// uniqueness of the value of this property is done. For a unique indentifier
            /// please check the Id property.
        /// </summary>
        public string DisplayName { get { … } set { … } }

        /// <summary>
        /// Occurs when the application instance terminates its execution.
        /// </summary>
        public event EventHandler<ApplicationEventArgs> ApplicationFinished;

        /// <summary>
        /// Creates an application instance with the given settings and sets the
            /// application display name to null.
        /// </summary>
        /// <param name="configuration">Configuration instance specifying the
            /// application settings.</param>
        public ApplicationBase(Configuration configuration): this(null, configuration)
```

LISTING 8.6

ApplicationBase＜M＞.

```
        { … }
        /// <summary>
        /// Creates an application instance withthe given settings and display name. As
              /// a result of the invocation, a new application unique identifier is created
              /// and the underlying application manager is initialized.
        /// </summary>
        /// <param name="configuration">Configurationinstance specifying the application
              /// settings.</param>
        /// <param name="displayName">Application friendly name.</param>
        public ApplicationBase(string displayName, Configuration configuration){ … }
        /// <summary>
        /// Starts the execution of the application instanceon Aneka.
        /// </summary>
        public void SubmitExecution() { … }
        /// <summary>
        /// Stops the execution of the entire application instance.
        /// </summary>
        public void StopExecution() { … }
        /// <summary>
        /// Invoke the application and wait until the application finishes.
        /// </summary>
        public void InvokeAndWait() { this.InvokeAndWait(null); }
        /// <summary>
        /// Invoke the application and wait until the application finishes, then invokes
              /// the given callback.
        /// </summary>
        /// <param name="handler">A pointer to a method that is executed at the end of
        /// the application.</param>
        public void InvokeAndWait(EventHandler<ApplicationEventArgs> handler) { … }

        /// <summary>
              /// Adds a shared file to the application.
        /// </summary>
        /// <param name="file">A string containing the path to the file to add.</param>
        public virtual void AddSharedFile(string file) { … }
        /// <summary>
              /// Adds a shared file to the application.
        /// </summary>
        /// <param name="file">A FileData instance containing the information about the
        /// file to add.</param>
        public virtual void AddSharedFile(FileData fileData) { … }
        /// <summary>
              /// Removes a file from the list of the shared files of the application.
              /// </summary>
             /// <param name="file"> A string containing the path to the file to
        /// remove.</param>
        public virtual void RemoveSharedFile(string filePath) { … }

              // here come the private implementation.

    }

}
```

LISTING 8.6

(Continued)

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;

namespace Aneka.MapReduce.Examples.WordCounter
{
    /// <summary>
    /// Class <b><i>Program<M></i></b>. Application driver for the Word Counter sample.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Reference to the configuration object.
        /// </summary>
        private static Configuration configuration = null;
    /// <summary>
    /// Location of the configuration file.
    /// </summary>
    private static string confPath = "conf.xml";

    /// <summary>
    /// Processes the arguments given to the application and according
    /// to the parameters read runs the application or shows the help.
    /// </summary>
    /// <param name="args">program arguments</param>
    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));

                 // alternatively we can use the following call
                 // application.InvokeAndWait();
        }
        catch(Exception ex)
        {
            Usage();
            IOUtil.DumpErrorReport(ex, "Aneka WordCounter Demo - Error Log");
        }
        finally
        {
            Logger.Stop();
        }
```

LISTING 8.7

WordCounter Job.

```
        }
        /// <summary>
        /// Hooks the ApplicationFinished events and Process the results
        /// if the application has been successful.
        /// </summary>
        /// <param name="sender">event source</param>
        /// <param name="e">event information</param>
        private static void OnDone(object sender, ApplicationEventArgs e) { … }
        /// <summary>
        /// Displays a simple informative message explaining the usage of the
        /// application.
        /// </summary>
        private static void Usage() { … }
    }
}
```

LISTING 8.7

(Continued)

component as a service; the latter controls the execution of jobs and schedules tasks. Therefore, the 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. All these operations are performed asynchronously and triggered by events happening in the Aneka middleware.

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: *MapReduceSchedulerService*, *ExecutorManager*, and *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. It is possible to configure more than one *MapReduceExecutor* instance, and this is helpful in the case of multicore nodes, where more than one task can be executed at the same time.

8.3.1.3 Distributed file system support

Unlike the other programming models Aneka supports, the MapReduce model does not leverage the default Storage Service for storage and data transfer but uses a distributed file system implementation. The reason for this is because the requirements in terms of file management are significantly different with respect to the other models. In particular, MapReduce has been designed to

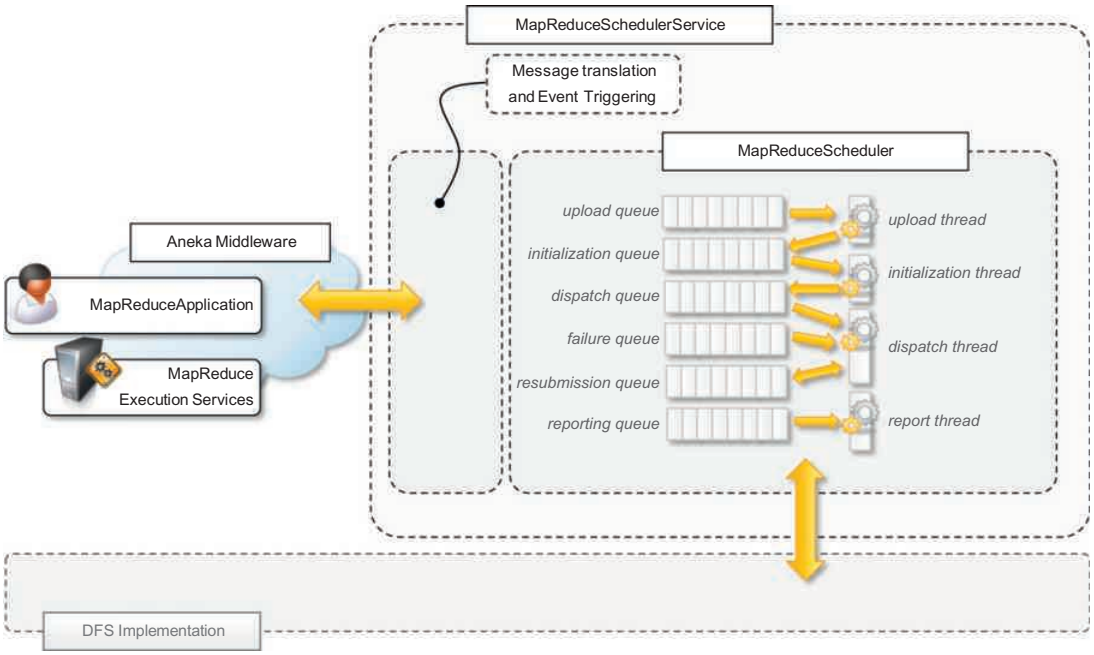


FIGURE 8.9

MapReduce Scheduling Service architecture.

process large quantities of data stored in files of large dimensions. Therefore, the support provided by a distributed file system, which can leverage multiple nodes for storing data, is more appropriate. Distributed file system implementations guarantee high availability and better efficiency by means of replication and distribution. Moreover, 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, as described in Chapter 5 (Section 5.2.3), 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. In a distributed file system, the stream might also access the network if the file chunk is not stored on the local node. Aneka provides interfaces that allow performing such operations and the capability to plug different file systems behind them by providing the appropriate implementation. The current implementation provides bindings to HDFS.

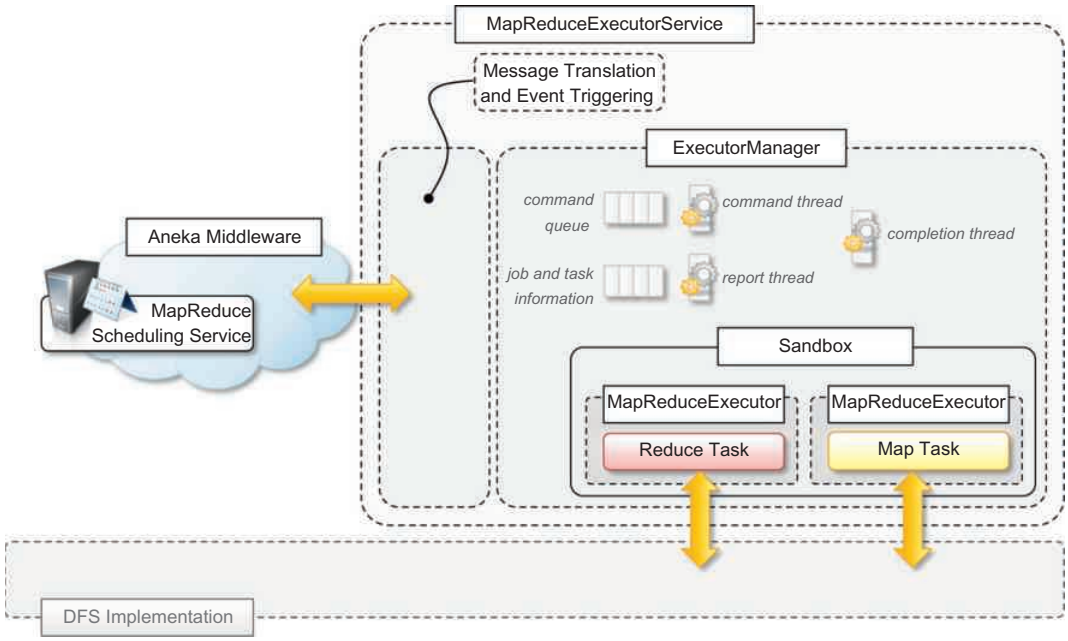


FIGURE 8.10

MapReduce Execution Service architecture.

On top of these low-level interfaces, the MapReduce programming model offers classes to read from and write to files in a sequential manner. These are the classes *SeqReader* and *SeqWriter*. They provide sequential access for reading and writing key-value pairs, and they expect a 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. All the .NET built-in types are supported. Since MapReduce jobs operate very often with data represented in common text files, a specific version of the *SeqReader* and *SeqWriter* classes have been designed to read and write text files as a sequence of key-value pairs. In the case of the read operation, each value of the pair is represented by a line in the text file, whereas the key is automatically generated and assigned to the position in bytes where the line starts in the file. In the write operation, the writing of the key is skipped and the values are saved as single lines.

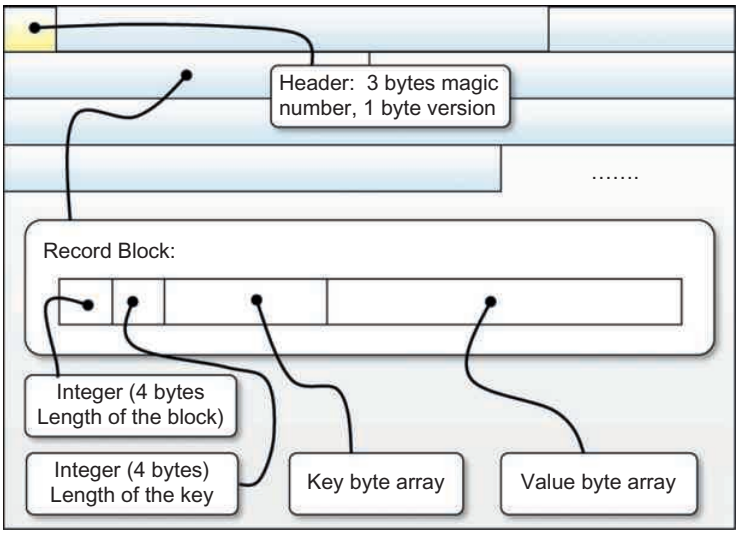


FIGURE 8.11

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. The *SeqWriter* class exposes different versions of the *Append* method.

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.

The *OnDone* callback checks to see whether the application has terminated successfully. If there are no errors, it iterates over the result files downloaded in the workspace output directory. By default, the files are saved in the *output* subdirectory of the workspace directory. For each of the result files, it opens a *SeqReader* instance on it and dumps the content of the key-value pair into a text file, which can be opened by any text editor.

8.3.2 Example application

MapReduce is a very useful model for processing large quantities of data, which in many cases are maintained in a semistructured form, such as logs or Web pages. 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. In this section, we describe in

```
using Aneka.MapReduce.Common;

namespace Aneka.MapReduce.DiskIO
{
    /// <summary>
    /// 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. This classprovides an interface for exposing the content
    /// of a file split as an enumeration of key-value pairs and offersfacilities for both
     /// accessing keys and values as objects and their corresponding binary values.
    /// </summary>
    public class SeqReader
    {
        /// <summary>
        /// Creates a SeqReader instance and attaches it to the given file. This constructor
        /// initializes the instance with the default value for the internal buffers and does
        /// not set any information about the types of the keys and values read from the
        /// file.
        /// </summary>
        public SeqReader(string file) : this(file, null, null) { … }
        /// <summary>
        /// Creates a SeqReader instance, attaches it to the given file, and sets the
        /// internal buffer size to bufferSize. This constructor does not provide any
        /// information about the types of the keys and values read from the file.
        /// </summary>
        public SeqReader(string file, int bufferSize) : this(file,null,null,bufferSize) { … }
        /// <summary>
        /// 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. The internal buffers are initialized with the default dimension.
        /// </summary>
        public SeqReader(string file, Type keyType, Type valueType)
            : this(file, keyType, valueType, SequenceFile.DefaultBufferSize) { … }
        /// <summary>
        /// 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. The internal buffers are initialized with the bufferSize dimension.
        /// </summary>
        public SeqReader(string file, Type keyType, Type valueType, int bufferSize){ … }
        /// <summary>
        /// Sets the metadata information about the keys and the values contained in the data
        /// file.
        /// </summary>
        public void SetType(Type keyType, Type valueType) { … }
        /// <summary>
        /// Checks whether there is another record in the data file and moves the current
        /// file pointer to its beginning.
        /// </summary>
        public bool HaxNext() { … }
        /// <summary>
        /// Gets the object instance corresponding to the next key in the data file.
        /// in the data file.
        /// </summary>
        public object NextKey() { … }
```

LISTING 8.8

SeqReader and *SeqWriter* Classes.

```
        /// <summary>
        /// Gets the object instance corresponding to the next value in the data file.
        /// in the data file.
        /// </summary>
        public object NextValue() { … }
        /// <summary>
        /// Gets the raw bytes that contain the value of the serializedinstance of the
        /// current key.
        /// </summary>
        public BufferInMemory NextRawKey() { … }
        /// <summary>
        /// Gets the raw bytes that contain the value of the serialized instance of the
        /// current value.
        /// </summary>
        public BufferInMemory NextRawValue() { … }

        /// <summary>
        /// Gets the position of the file pointer as an offset from its beginning.
        /// </summary>
        public long CurrentPosition() { … }
        /// <summary>
        /// Gets the size of the file attached to this instance of SeqReader.
        /// </summary>
        public long StreamLength() { … }
        /// <summary>
        /// Moves the file pointer to position. If the value of position is 0 or negative,
        /// returns the current position of the file pointer.
        /// </summary>
        public long Seek(long position) { … }
        /// <summary>
        /// Closes the SeqReader instanceand releases all the resources that have been
        /// allocated to read fromthe file.
        /// </summary>
        public void Close() { … }

        // private implementation follows
    }

    /// <summary>
    /// Class SeqWriter. This class implements a file writer for the sequence
    /// 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.
    /// </summary>
    public class SeqWriter
    {

        /// <summary>
        /// Creates a SeqWriter instance for writing to file. This constructor initializes
        /// the instance with the default value for the internal buffers.
        /// </summary>
        public SeqWriter(string file) : this(file, SequenceFile.DefaultBufferSize){ … }
        /// <summary>
        /// Creates a SeqWriter instance, attachesit to the given file, and sets the
        /// internal buffer size to bufferSize.
        /// </summary>
        public SeqWriter(string file, int bufferSize) { … }
```

LISTING 8.8

(Continued)

```
        /// <summary>
        /// Appends a key-value pair to the data file split.
        /// </summary>
        public void Append(object key, object value) { … }
        /// <summary>
        /// Appends a key-value pair to the data file split.
        /// </summary>
        public void AppendRaw(byte[] key, byte[] value) { … }
        /// <summary>
        /// Appends a key-value pair to the data file split.
        /// </summary>
        public void AppendRaw(byte[] key, int keyPos, int keyLen,
                              byte[] value, int valuePos, int valueLen) { … }

        /// <summary>
        /// Gets the length of the internal buffer or 0 if no buffer has been allocated.
        /// </summary>
        public longLength() { … }
        /// <summary>
        /// Gets the length of data file split on disk so far.
        /// </summary>
        public long FileLength() { … }
        /// <summary>
        /// Closes the SeqReader instance and releases all the resources that have been
        /// allocated to write to the file.
        /// </summary>
        public void Close() { … }

        // private implementation follows
    }
}
```

LISTING 8.8

(Continued)

detail the problem to be addressed and design the *Mapper* and *Reducer* classes that are used to execute log-parsing and data extraction operations.

8.3.2.1 Parsing Aneka logs

Aneka components (daemons, container instances, and services) produce a lot of information that is stored in the form of log files. The most relevant information is stored in the container instances logs, which store the information about the applications that are executed on the Cloud. 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. In Aneka containers, the system is configured to produce a log file that is partitioned into chunks every time the container instance restarts. Moreover, the information contained in the log file can be customized in its appearance. Currently the default layout is the following:

DD MMM YY hh:mm:ss level – message

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;

namespace Aneka.MapReduce.Examples.WordCounter
{
    /// <summary>
    /// Class Program. Application driver for the Word Counter sample.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Reference to the configuration object.
        /// </summary>
        private static Configuration configuration = null;
        /// <summary>
        /// Location of the configuration file.
        /// </summary>
        private static string confPath = "conf.xml";

        /// <summary>
        /// Processes the arguments given to the application and according
        /// to the parameters read runs the application or shows the help.
        /// </summary>
        /// <param name="args">program arguments</param>
        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
                    // application.InvokeAndWait();
            }
            catch(Exception ex)
            {
                Program.Usage();
                IOUtil.DumpErrorReport(ex, "Aneka WordCounter Demo - Error Log");
            }
            finally
            {
                Logger.Stop();
            }
        }
```

LISTING 8.9

WordCounter Job.

```
/// <summary>
/// Hooks the ApplicationFinished events and process the results
/// if the application has been successful.
/// </summary>
/// <param name="sender">event source</param>
/// <param name="e">event information</param>
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");
        }
    }
}
```

LISTING 8.9

(Continued)

```
        /// <summary>
        /// Starts the notepad process and displays the given file.
        /// </summary>
        private static voidStartNotepad(string file) { … }
        /// <summary>
        /// Displays a simple informative message explaining the usage of the
        /// application.
        /// </summary>
        private static void Usage() { … }
    }
}
```

LISTING 8.9

(Continued)

Some examples of formatted log messages are:

15 Mar 2011 10:30:07 DEBUG−SchedulerService: . . .
HandleSubmitApplication−SchedulerService: . . .
15 Mar 2011 10:30:07 INFO−SchedulerService: Scanning candidate storage . . .
15 Mar 2011 10:30:10 INFO−Added [WU: 51d55819-b211-490f-b185-8a25734ba705, 4e86fd02. . .
15 Mar 2011 10:30:10 DEBUG−StorageService:NotifyScheduler−Sending FileTransferMessage. . .
15 Mar 2011 10:30:10 DEBUG−IndependentSchedulingService:QueueWorkUnit−Queueing. . .
15 Mar 2011 10:30:10 INFO−AlgorithmBase::AddTasks[64] Adding 1 Tasks
15 Mar 2011 10:30:10 DEBUG−AlgorithmBase:FireProvisionResources−Provision Resource: 1

In the content of the sample log lines, we observe that the message parts of almost all the log lines exhibit a similar structure, and they start with the information about the component that enters the log line. This information can be easily extracted by looking at the first occurrence of the *:* character following a sequence of characters that do not contain spaces.

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:

$$map: (long, string) => (string, long)$$
$$reduce: (string, long) => (string, long)$$

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. By modifying the canonical structure we discussed, we can perform both analyses at the same time instead of developing two different MapReduce jobs. Note that the operation performed by the *Reducer* class is the same in both cases, whereas the operation of the *Mapper* class changes, but the type of the key-value pair that is generated is the same for the two jobs. Therefore, it is possible to combine the two tasks performed by the *map* function into one single *Mapper* class that will produce two key-value pairs for each input line. Moreover, we can differentiate the name of Aneka components from the log-level names by using an initial underscore character it will be very easy to post-process the output of the *reduce* function in order to present and organize data.

8.3.2.2 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. Since we decided to combine the two MapReduce jobs into one single job, each *map* task will at most emit two key-value pairs. This is because some of the log lines do not record the name of the component that entered the line; for these lines, only the key-value pair corresponding to the log level will be emitted.

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)*. It then tries to locate the position of the name of the Aneka component that entered the log line by looking for a sequence of characters that is limited by a colon. If such a sequence does not contain spaces, it then represents the name of the Aneka component. In this case, another key-value pair *(component-name, 1)* is emitted. As already discussed, to differentiate the log-level labels from component names, an underscore is prefixed to the name of the key in this second case.

8.3.2.3 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. The infrastructure will already aggregate all the values that have been emitted for a given key; therefore, we simply need to iterate over the collection of values and sum them up.

As we see in 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. It will be the responsibility of the driver program to differentiate among the different types of information assembled in the output files.

8.3.2.4 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. Moreover, since we have designed the mapper component to extract two different types of information, another task that is performed in the driver application is the separation of these two statistics into two different files for further analysis.

```
using Aneka.MapReduce;

namespace Aneka.MapReduce.Examples.LogParsing
{
    /// <summary>
    /// 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 key-value (log-level, 1) and potentially another key-value
    /// (_aneka-component-name,1) if it is able to extract such information from the
    /// input.
    /// </summary>
    public class LogParsingMapper: Mapper<long,string>
    {
        /// <summary>
        /// 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 .
        /// </summary>
        /// <param name="input">map input</param>
        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.

```
using Aneka.MapReduce;

namespace Aneka.MapReduce.Examples.LogParsing
{
    /// <summary>
    /// 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 over values of the enumerator and sums the values
    /// before emitting the sum to the output file.
    /// </summary>
    public class LogParsingReducer: Reducer<string,long>
    {
        /// <summary>
        /// Iterates all over the values of the enumerator and sums up
        /// all the values before emitting the sum to the output file.
        /// </summary>
        /// <param name="input">reduce source</param>
        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.

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*. MapReduce jobs can be configured using a specific section of the configuration file named *MapReduce*. Within this section, two subsections control the properties of input and output files and are named *Input* and *Output*, respectively. The input and output sections may contain the following properties:

- *Format (string)* defines the format of the input file. If this property is set, the only supported value is *text*.

- *Filter (string)* defines the search pattern to be used to filter the input files to process in the workspace directory. This property only applies for the Output properties group.
- *NewLine (string)* defines the sequence of characters that is used to detect (or write) a new line in the text stream. This value is meaningful when the input/output format is set to *text* and the default value is selected from the execution environment if not set.
- *Separator (character)* property is only present in the *Output* section and defines the character that needs to be used to separate the key from the value in the output file. As with the previous property, this value is meaningful when the input/output format is set to *text*.

Besides the specific setting for input and output files, it is possible to control other parameters of a *MapReduce* job. These parameters are defined in the main *MapReduce* configuration section; their meaning was discussed in Section 8.3.1.

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.

As demonstrated, it is possible to open a $<Group\ name = "MapReduce" > \ldots <\!/Group>$ tag and enter all the properties that are required for the execution. The Aneka configuration file is based on a flexible framework that allows simply entering groups of name-value properties. The *Aneka.Property* and *Aneka.PropertyGroup* classes also provide facilities for converting the strings representing the value of a property into a corresponding built-in type if possible. This simplifies the task of reading from and writing to configuration objects.

The second element shown in Listing 8.12 is represented by the post-processing of the output files. This operation is performed in the *OnDone* method, whose invocation is triggered either if an error occurs during the execution of the MapReduce job or if it completes successfully. This method separates the occurrences of log-level labels from the occurrences of Aneka component names by saving them into two different files, *loglevels.txt* and *components.txt*, under the workspace directory, and then deletes the output directory where the output files of the reduce phase have been downloaded. These two files contain the aggregated result of the analysis and can be used to extract statistic information about the content of the log files and display in a graphical manner, as shown in the next section.

A final aspect that can be considered is the management of errors. Aneka provides a collection of APIs that are contained in the *Aneka.Util* library and represent utility classes for automating tedious tasks such as the appropriate collection of stack trace information associated with an exception or information about the types of the exception thrown. In our example, the reporting features, which are triggered in case of exceptions, are implemented in the *ReportError* method. This method utilizes the facilities offered by the *IOUtil* class to dump a simple error report to both the console and a log file that is named using the following pattern: *error.YYYY-MM-DD_hh-mm-ss.log*.

8.3.2.5 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. Moreover, this

```
using System.IO;
using Aneka.Entity;
using Aneka.MapReduce;

namespace Aneka.MapReduce.Examples.LogParsing
{
    /// <summary>
    /// 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.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Reference to the configuration object.
        /// </summary>
        private static Configuration configuration = null;
        /// <summary>
        /// Location of the configuration file.
        /// </summary>
        private static string confPath = "conf.xml";

        /// <summary>
        /// Processes the arguments given to the application and according
        /// to the parameters read runs the application or shows the help.
        /// </summary>
        /// <param name="args">program arguments</param>
        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();
            }
```

LISTING 8.12

Driver Program Implementation.

```
        Console.Readline();
    }
    /// <summary>
    /// Initializes the configuration and ensures that the appropriate input
    /// and output formats.are set
    /// </summary>
    /// <param name="configFile">A string containing the path to the config file.</param>
    /// <returns>An instance of the configuration class.</returns>
    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;
    }

    /// <summary>
    /// Hooks the ApplicationFinished events and process the results
    /// if the application has been successful.
```

LISTING 8.12

(Continued)

```
        /// </summary>
        /// <param name="sender">event source</param>
        /// <param name="e">event information</param>
        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);
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
```

LISTING 8.12

(Continued)

```
                    // 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>");
            }
        }
        /// <summary>
        /// Displays a simple informative message explaining the usage of the


        /// application.
        /// </summary>
        private static void Usage()
        {
            Console.WriteLine("Aneka Log Parsing - Usage Log.Parsing.Demo.Console.exe"
            + " [conf.xml]");
        }
        /// <summary>
        /// Dumps the error to the console, sets the exit code of the application to -1
        /// and saves the error dump into a file.
        /// </summary>
        /// <param name="ex">runtime exception</param>
        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

(Continued)

scenario also constitutes a real case study for MapReduce, since one of its most common practical applications is extracting semistructured information from logs and traces of execution.

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.

```
<?xml version="1.0" encoding="utf-8" ?>
<Aneka>
  <UseFileTransfer value="false" />
  <Workspace value="Workspace" />
  <SingleSubmission value="AUTO" />
  <PollingTime value="1000" />
  <LogMessages value="false" />
  <SchedulerUri value="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).

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.

Despite the simplicity of the parsing function implemented in the *map* task, this practical example shows how the Aneka MapReduce programming model can be used to easily perform massive data analysis tasks. The purpose of the case study was not to create a very refined parsing function but to demonstrate how to logically and programmatically approach a realistic data analysis case study with MapReduce and how to implement it on top of the Aneka APIs.

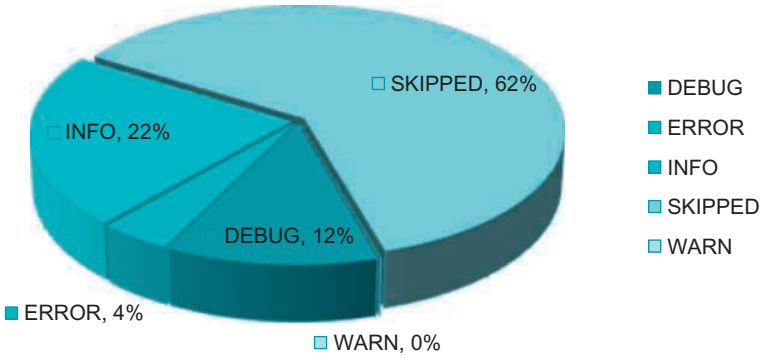


FIGURE 8.12

Log-level entries distribution.

SUMMARY

This chapter introduced the main characteristics of *data-intensive computing*. Data-intensive applications process or produce high volumes of data and may also exhibit compute-intensive properties. The amounts of data that have triggered the definition of data-intensive computation has changed over time, together with the technologies and the programming and storage models used for data-intensive computing. Data-intensive computing is a field that was originally prominent in high-speed WAN applications. It is now the domain of storage clouds, where the dimensions of data reach the size of terabytes, if not petabytes, and are referred to as *Big Data*. This term identifies the massive amount of information that is produced, processed, and mined, not only by scientific applications but also by companies providing Internet services, such as search, online advertising, social media, and social networking.

One of the interesting characteristics of our Big Data world is that the data are represented in a semistructured or unstructured form. Therefore, traditional approaches based on relational databases are not capable of efficiently supporting data-intensive applications. New approaches and storage models have been investigated to address these challenges. In the context of storage systems, the most significant efforts have been directed toward the implementation of high-performance distributed file systems, storage clouds, and NoSQL-based systems. For the support of programming data-intensive applications, the most relevant innovation has been the introduction of MapReduce, together with all its variations aiming at extending the applicability of the proposed approach to a wider range of scenarios.

MapReduce was proposed by Google and provides a simple approach to processing large quantities of data based on the definition of two functions, *map* and *reduce*, that are applied to the data in a two-phase process. First, the *map* phase extracts the valuable information from the data and stores it in key-value pairs, which are eventually aggregated together in the *reduce* phase. This model, even though constraining, proves successful in several application scenarios.

We discussed the reference model of MapReduce as proposed by Google and provided pointers to relevant variations. We described the implementation of MapReduce in Aneka. Similar to thread

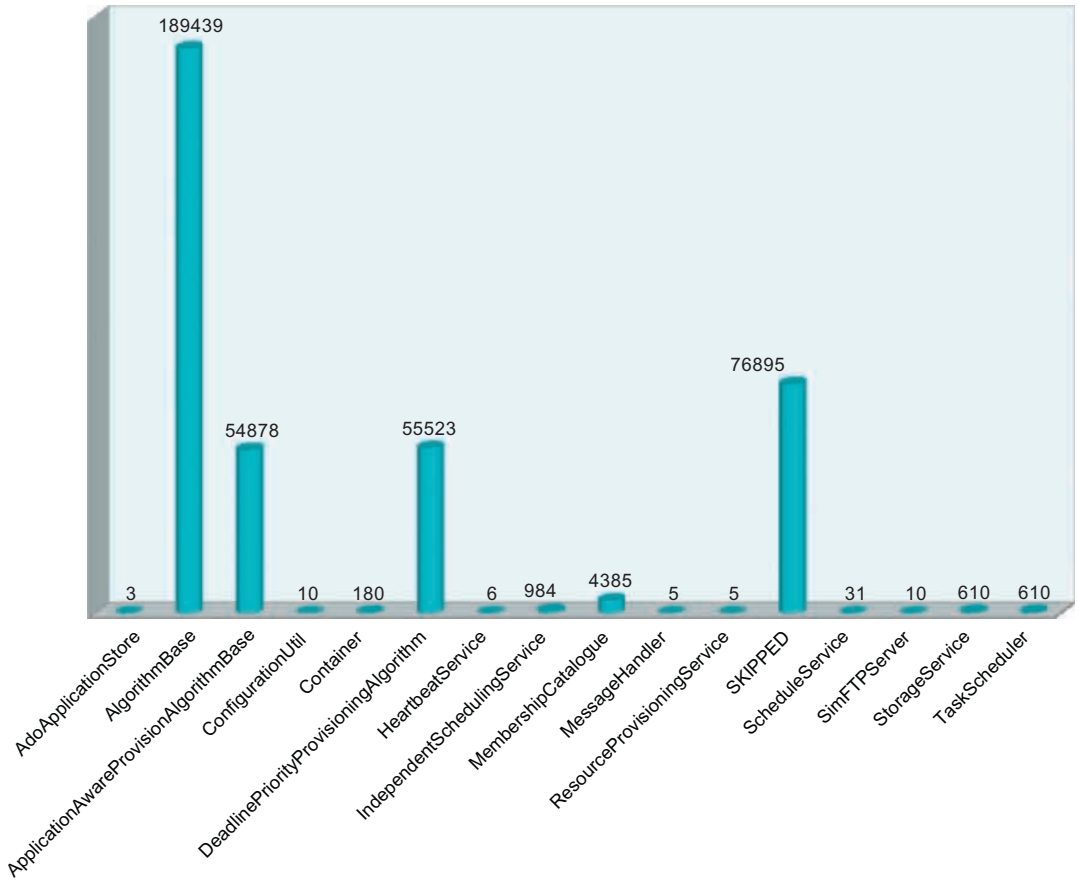


FIGURE 8.13

Component entries distribution.

and task programming models in Aneka, we discussed the programming abstractions supporting the design and implementation of MapReduce applications. We presented the structure and organization of Aneka's runtime services for the execution of MapReduce jobs. Finally, we included step-by-step examples of how to design and implement applications using Aneka MapReduce APIs.

Review questions

1. What is a data-intensive computing? Describe the characteristics that define this term.
2. Provide an historical perspective on the most important technologies that support data-intensive computing.

3. What are the characterizing features of so-called Big Data?
4. List some of the important storage technologies that support data-intensive computing and describe one of them.
5. Describe the architecture of the Google File System.
6. What does the term NoSQL mean?
7. Describe the characteristics of Amazon Simple Storage Service (S3).
8. What is Google Bigtable?
9. What are the requirements of a programming platform that supports data-intensive computations?
10. What is MapReduce?
11. Describe the kinds of problems MapReduce can solve and give some real examples.
12. List some of the variations on or extensions to MapReduce.
13. What are the major components of the Aneka MapReduce Programming Model?
14. How does the MapReduce model differ from the other models supported by Aneka and discussed in this book?
15. Describe the components of the Scheduling and Execution Services that constitute the runtime infrastructure supporting MapReduce.
16. Describe the architecture of the data storage layer designed for Aneka MapReduce and the I/O APIs for handling MapReduce files.
17. Design and implement a simple program that uses MapReduce for the computation of Pi.

This page intentionally left blank

PART

Industrial Platforms and New Developments 3