Karnataka ReddyJana Sangha (R)
VEMANA INSTITUTE OF TECHNOLOGY
Approved by AICTE-New Delhi, Affiliated to VTU-Belagavi, Recognized by Govt. of Karnataka
#1,Mahayogi Vemana Road, 3rd Block, Koramangala, Bengaluru - 34.
Accredited by NBA
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Module-5
Cloud Programming and Software Environments:

Prepared by
Dr. Noor Basha / Prof. Gopala Krishna P
Vemana Institute of Technlogy

5.1 Features of Cloud and Grid Platforms

In four tables, we cover the capabilities, traditional features, data features, and features for programmers and runtime systems to use. The entries in these tables are source references for anyone who wants to program the cloud efficiently.

5.1.1 Cloud Capabilities and Platform Features

The These capabilities offer cost-effective utility computing with the elasticity to scale up and down in power. However, as well as this key distinguishing feature, commercial clouds offer a growing number of additional capabilities commonly termed "Platform as a Service" (PaaS). For Azure, current platform features include Azure Table, queues, blobs, Database SQL, and web and Worker roles. Amazon is often viewed as offering "just" Infrastructure as a Service (IaaS), but it continues to add platform features including SimpleDB (similar to Azure Table), queues, notification, monitoring, content delivery network, relational database, and MapReduce (Hadoop). Google does not currently offer a broad-based cloud service, but the Google App Engine (GAE) offers a powerful web application development environment

5.1.2 Traditional Features Common to Grids and Clouds

In this section, we concentrate on features related to workflow, data transport, security, and availability concerns that are common to today's computing grids and clouds.

Table 5.1 Important Cloud Platform Capabilities

Physical or virtual computing platform	The cloud environment consists of some physical or virtual platforms. Virtual platforms have unique capabilities to provide isolated environments for different applications and users.
Massive data storage service, distributed file system	With large data sets, cloud data storage services provide large disk capacity and the service interfaces that allow users to put and get data. The distributed file system offers massive data storage service. It can provide similar interfaces as local file systems.
Massive database storage service	Some distributed file systems are sufficient to provide the underlying storage service application developers need to save data in a more semantic way. Just like DBMS in the traditional software stack, massive database storage services are needed in the cloud.
Massive data processing method and programming model	Cloud infrastructure provides thousands of computing nodes for even a very simple application. Programmers need to be able to harness the power of these machines without considering tedious infrastructure management issues such as handling network failure or scaling the running code to use all the computing facilities provided by the platforms.
Workflow and data query language support	The programming model offers abstraction of the cloud infrastructure. Similar to the SQL language used for database systems, in cloud computing, providers have built some workflow language as well as data query language to support better application logic.
Programming interface and service deployment	Web interfaces or special APIs are required for cloud applications: J2EE, PHP, ASP, or Rails. Cloud applications can use Ajax technologies to improve the user experience while using web browsers to access the functions provided. Each cloud provider opens its programming interface for accessing the data stored in massive storage.
Runtime support	Runtime support is transparent to users and their applications. Support includes distributed monitoring services, a distributed task scheduler, as well as distributed locking and other services. They are critical in running cloud applications.
Support services	Important support services include data and computing services. For example, clouds offer rich data services and interesting data parallel execution models like MapReduce.

Table 5.2 lists some low-level infrastructure features

Table 5.2 Infrastructure Cloud Features
Accounting: Includes economies; clearly an active area for commercial clouds
Appliances: Preconfigured virtual machine (VM) image supporting multifaceted tasks such as message- passing interface (MPI) clusters
Authentication and authorization: Could need single sign-on to multiple systems
Data transport: Transports data between job components both between and within grids and clouds; exploits custom storage patterns as in BitTorrent
Operating systems: Apple, Android, Linux, Windows
Program library: Stores images and other program material
Registry: Information resource for system (system version of metadata management)
Security: Security features other than basic authentication and authorization; includes higher level concepts such as trust
Scheduling: Basic staple of Condor, Platform, Oracle Grid Engine, etc.; clouds have this implicitly as is especially clear with Azure Worker Role
Gang scheduling: Assigns multiple (data-parallel) tasks in a scalable fashion; note that this is provided automatically by MapReduce
Software as a Service (SaaS): Shared between clouds and grids, and can be supported without special attention; Note use of services and corresponding service oriented architectures are very successful and are used in clouds very similarly to previous distributed systems.
Virtualization: Basic feature of clouds supporting "elastic" feature highlighted by Berkeley as characteristic of what defines a (public) cloud; includes virtual networking as in ViNe from University of Florida

Table 5.3 lists traditional programming environments for parallel and distributed systems that need to be supported in Cloud environments. They can be supplied as part of system (Cloud Platform) or user environment.

Table 5.3 Traditional Features in Cluster, Grid, and Parallel Computing Environments
Cluster management: ROCKS and packages offering a range of tools to make it easy to bring up clusters
Data management: Included metadata support such as RDF triple stores (Semantic web success and can be built on MapReduce as in SHARD); SQL and NOSQL included in
Grid programming environment: Varies from link-together services as in Open Grid Services Architecture (OGSA) to GridRPC (Ninf, GridSolve) and SAGA
OpenMP/threading: Can include parallel compilers such as Cilk; roughly shared memory technologies. Even transactional memory and fine-grained data flow come here
Portals: Can be called (science) gateways and see an interesting change in technology from portlets to HUBzero and now in the cloud: Azure Web Roles and GAE
Scalable parallel computing environments: MPI and associated higher level concepts including
ill-fated HP FORTRAN, PGAS (not successful but not disgraced), HPCS languages (X-10, Fortress, Chapel), patterns (including Berkeley dwarves), and functional languages such as F# for distributed memory
Virtual organizations: From specialized grid solutions to popular Web 2.0 capabilities such as Facebook Workflow: Supports workflows that link job components either within or between grids and clouds; relate to LIMS Laboratory Information Management Systems.

Table 5.4 presents features emphasized by clouds and by some grids. Note that some of the features in Table 5.4 have only recently been offered in a major way. In particular, these features are not offered on academic cloud infrastructures such as Eucalyptus, Nimbus, OpenNebula, or Sector/Sphere (although Sector is a data parallel file system or DPFS classified in Table 5.4).

Table 5.4 Platform Features Supported by Clouds and (Sometimes) Grids

Blob: Basic storage concept typified by Azure Blob and Amazon S3

DPFS: Support of file systems such as Google (MapReduce), HDFS (Hadoop), and Cosmos (Dryad) with compute-data affinity optimized for data processing

Fault tolerance: As reviewed in [1] this was largely ignored in grids, but is a major feature of clouds MapReduce: Support MapReduce programming model including Hadoop on Linux, Dryad on Windows HPCS, and Twister on Windows and Linux. Include new associated languages such as Sawzall, Pregel,

Pig Latin, and LINQ

Monitoring: Many grid solutions such as Inca. Can be based on publish-subscribe

Notification: Basic function of publish-subscribe systems

Programming model: Cloud programming models are built with other platform features and are related to familiar web and grid models

Queues: Queuing system possibly based on publish-subscribe

Scalable synchronization: Apache Zookeeper or Google Chubby. Supports distributed locks and used by BigTable. Not clear if (effectively) used in Azure Table or Amazon SimpleDB

SQL: Relational database

5.1.2.1 Workflow

There are commercial systems such as Pipeline Pilot, AVS (dated), and the LIMS environments. A recent entry is Trident [2] from Microsoft Research which is built on top of Windows Workflow Founda- tion. If Trident runs on Azure or just any old Windows machine, it will run workflow proxy ser- vices on external (Linux) environments. Workflow links multiple cloud and noncloud services in real applications on demand.

5.1.2.2 Data Transport

The cost (in time and money) of data transport in (and to a lesser extent, out of) commercial clouds is often discussed as a difficulty in using clouds. If commercial clouds become an important component of the national cyber infrastructure, we can expect that high-bandwidth links will be made available between clouds and TeraGrid. The special structure of cloud data with blocks (in Azure blobs) and tables could allow high-performance parallel algorithms, but initially, simple HTTP mechanisms are used to transport data [3–5] on academic systems/TeraGrid and commercial clouds.

5.1.2.3 Security , Privacy and Availability

The following techniques are related to security, privacy, and availability requirements for develop- ing a healthy and dependable cloud programming environment. We summarize these techniques here

- Use virtual clustering to achieve dynamic resource provisioning with minimum overhead cost.
- Use stable and persistent data storage with fast queries for information retrieval.
- Use special APIs for authenticating users and sending e-mail using commercial accounts.
- Cloud resources are accessed with security protocols such as HTTPS and SSL.

- Fine-grained access control is desired to protect data integrity and deter intruders or hackers.
- Shared data sets are protected from malicious alteration, deletion, or copyright violations.
- Features are included for availability enhancement and disaster recovery with life migration of VMs.
- Use a reputation system to protect data centers. This system only authorizes trusted clients and stops pirates.

5.1.3 Data Features and Databases

In the following paragraphs, we review interesting programming features related to the program library, blobs, drives, DPFS, tables, and various types of databases including SQL, NOSQL, and nonrelational databases and special queuing services.

5.1.3.1 Program Library

Many efforts have been made to design a VM image library to manage images used in academic and commercial clouds. The basic cloud environments described in this chapter also include many manage- ment features allowing convenient deployment and configuring of images (i.e., they support IaaS).

5.1.3.2 Blobes and drive

The basic storage concept in clouds is blobs for Azure and S3 for Amazon. These can be organized (approximately, as in directories) by containers in Azure. In addition to a service interface for blobs and S3, one can attach "directly" to compute instances as Azure drives and the Elastic Block Store for Amazon. This concept is similar to shared file systems such as Lustre used in TeraGrid. The cloud storage is intrinsically fault-tolerant while that on TeraGrid needs backup storage. However, the architecture ideas are similar between clouds and TeraGrid, and the Simple Cloud File Storage API [6] could become important here.

5.1.3.3 DPFS

This covers the support of file systems such as Google File System (MapReduce), HDFS (Hadoop), and Cosmos (Dryad) with compute-data affinity optimized for data processing. It could be possible to link DPFS to basic blob and drive-based architecture, but it's simpler to use DPFS as an application-centric storage model with compute-data affinity and blobs and drives as the repository-centric view. In general, data transport will be needed to link these two data views. It seems important to consider this carefully, as DPFS file systems are precisely designed for efficient execution of data-intensive applications. However, the importance of DPFS for linkage with Amazon and Azure is not clear, as these clouds do not currently offer fine-grained support for compute-data affinity. We note here that Azure Affinity Groups are one interesting capability [7]. We expect that initially blobs, drives, tables, and queues will be the areas where academic systems will most usefully provide a platform similar to Azure (and Amazon). Note the HDFS (Apache) and Sector (UIC) projects in this area.

5.1.3.4 SQL and Relational Databases

Both Amazon and Azure clouds offer relational databases and it is straightforward for academic systems to offer a similar capability unless there are issues of huge scale where, in fact, approaches based on tables and/or MapReduce might be more appropriate [8]. As one early user, we are developing on FutureGrid a new private cloud computing model for the Observational Medical

Outcomes Partnership (OMOP) for patient-related medical data which uses Oracle and SAS where FutureGrid is adding Hadoop for scaling to many different analysis methods.

Note that databases can be used to illustrate two approaches to deploying capabilities. Traditionally, one would add database software to that found on computer disks. This software is executed, providing your database instance. However, on Azure and Amazon, the database is installed on a separate VM independent from your job (worker roles in Azure). This implements "SQL as a Service." It may have some performance issues from the messaging interface, but the "aaS" deployment clearly simplifies one's system. For N platform features, one only needs N services, whereas number of possible images with alternative approaches is a prohibitive 2N.

5.1.3.5 Table and NOSQL Nonrelational Databases

A substantial number of important developments have occurred regarding simplified database structures—termed "NOSQL" [9,10]—typically emphasizing distribution and scalability. These are present in the three major clouds: BigTable [11] in Google, SimpleDB [12] in Amazon, and Azure Table [13] for Azure. Tables are clearly important in science as illustrated by the VOTable standard in astronomy [14] and the popularity of Excel. However, there does not appear to be substantial experience in using tables outside clouds.

There are, of course, many important uses of nonrelational databases, especially in terms of triple stores for metadata storage and access. Recently, there has been interest in building scalable RDF triple stores based on MapReduce and tables or the Hadoop File System [8,15], with early success reported on very large stores. The current cloud tables fall into two groups: Azure Table and Amazon SimpleDB are quite similar [16] and support lightweight storage for "document stores," while BigTable aims to manage large distributed data sets without size limitations.

All these tables are schema-free (each record can have different properties), although BigTable has a schema for column (property) families. It seems likely that tables will grow in importance for scientific computing, and academic systems could support this using two Apache projects: Hbase[17] for BigTable and CouchDB [18] for a document store. Another possibility is the open source SimpleDB implementation M/DB [19]. The new Simple Cloud APIs [6] for file storage, document storage services, and simple queues could help in providing a common environment between academic and commercial clouds.

5.1.3.6 Queuing Services

Both Amazon and Azure offer similar scalable, robust queuing services that are used to communicate between the components of an application. The messages are short (less than 8 KB) and have a Representational State Transfer (REST) service interface with "deliver at least once" semantics. They are controlled by timeouts for posting the length of time allowed for a client to process. One can build a similar approach (on the typically smaller and less challenging academic environments), basing it on publish-subscribe systems such as ActiveMQ [20] or NaradaBrokering [21,22] with which we have substantial experience.

5.1.4 Programming and Runtime support

Programming and runtime support are desired to facilitate parallel programming and provide runtime support of important functions in today's grids and clouds. Various MapReduce systems are reviewed in this section.

5.1.4.1 Worker and Web Roles

The roles introduced by Azure provide nontrivial functionality, while preserving the better affinity

support that is possible in a nonvirtualized environment. Worker roles are basic schedulable processes and are automatically launched. Note that explicit scheduling is unnecessary in clouds for individual worker roles and for the "gang scheduling" supported transparently in MapReduce. Queues are a critical concept here, as they provide a natural way to manage task assignment in afault–tolerant, distributed fashion. Web roles provide an interesting approach to portals. GAE is largely aimed at web applications, whereas science gateways are successful in TeraGrid.

5.1.4.2 Map Reduce

There has been substantial interest in "data parallel" languages largely aimed at loosely coupled computations which execute over different data samples. The language and runtime generate and provide efficient execution of "many task" problems that are well known as successful grid applications.

Table 5.5 Comparison of MapReduce Type Systems

	Google MapReduce [28]	Apache Hadoop [23]	Microsoft Dryad [26]	Twister [29]	Azure Twister [30]
Programming Model	MapReduce	MapReduce	DAG execution, extensible to MapReduce and other patterns	Iterative MapReduce	Currently just MapReduce; will extend to Iterative MapReduce
Data Handling	GFS (Google File System)	HDFS (Hadoop Distributed File System)	Shared directories and local disks	Local disks and data management tools	Azure blob storage
Scheduling	Data locality	Data locality; rack-aware, dynamic task scheduling using global queue	Data locality; network topology optimized at runtime; static task partitions	Data locality; static task partitions	Dynamic task scheduling through global queue
Failure Handling	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of iterations	Reexecution of failed tasks; duplicated execution of slow tasks
HLL Support	Sawzall [31]	Pig Latin [32,33]	DryadLINQ [27]	Pregel [34] has related features	N/A
Environment	Linux cluster	Linux clusters, Amazon Elastic MapReduce on EC2	Windows HPCS cluster	Linux cluster, EC2	Windows Azure, Azure Local Development Fabric
Intermediate Data Transfer	File	File, HTTP	File, TCP pipes, shared-memory FIFOs	Publish-subscribe messaging	Files, TCP

5.1.4.3 Cloud Programming Models

In many ways, most of the previous sections describe programming model features, but these are "macroscopic" constructs and do not address, for example, the coding (language and libraries). Both the GAE and Manjrasoft Aneka environments represent programming models; both are applied to clouds, but are really not specific to this architecture. Iterative MapReduce is an interesting programming model that offers portability between cloud, HPC and cluster environments.

5.1.4.4 Saas

Services are used in a similar fashion in commercial clouds and most modern distributed systems. We expect users to package their programs wherever possible, so no special support is needed to enable SaaS. In addition to the technical features, such as MapReduce, BigTable, EC2, S3, Hadoop, AWS, GAE, and WebSphere2, we need protection features that may help us to achieve scalability, security, privacy, and availability.

5.2 PARALLEL AND DISTRIBUTED PROGRAMMING PARADIGMS

We define a parallel and distributed program as a parallel program running on a set of computing engines or a distributed computing system. The term carries the notion of two fundamental terms in computer science: distributed computing system and parallel computing. A distributed computing system is a set of computational engines connected by a network to achieve a common goal of running a job or an application. A computer cluster or network of workstations is an example of a distributed computing system. Parallel computing is the simultaneous use of more than one computa- tional engine (not necessarily connected via a network) to run a job or an application.

For instance, parallel computing may use either a distributed or a nondistributed computing system such as a multi- processor platform.Running a parallel program on a distributed computing system (parallel and distributed pro- gramming) has several advantages for both users and distributed computing systems. From the users' perspective, it decreases application response time; from the distributed computing systems'standpoint, it increases throughput and resource utilization. Running a parallel program on a distributed computing system, however, could be a very complicated process.

5.2.1 Parallel Computing and Programming Paradigms

Consider a distributed computing system consisting of a set of networked nodes or workers. The system issues for running a typical parallel program in either a parallel or a distributed manner would include the following [36–39]:

- Partitioning This is applicable to both computation and data as follows:
- Computation partitioning This splits a given job or a program into smaller tasks. Partitioning greatly depends on correctly identifying portions of the job or program that can be performed concurrently. In other words, upon identifying parallelism in the structure of the program, it can be divided into parts to be run on different workers. Different parts may process different data or a copy of the same data.
- Data partitioning This splits the input or intermediate data into smaller pieces. Similarly, upon identification of parallelism in the input data, it can also be divided into pieces to be processed on different workers. Data pieces may be processed by different parts of a program or a copy of the same program.
- Mapping This assigns the either smaller parts of a program or the smaller pieces of data to underlying resources. This process aims to appropriately assign such parts or pieces to be run simultaneously on different workers and is usually handled by resource allocators in the system.
- Synchronization Because different workers may perform different tasks, synchronization and coordination among workers is necessary so that race conditions are prevented and data dependency among different workers is properly managed. Multiple accesses to a shared resource by different workers may raise race conditions, whereas data dependency happens when a worker needs the processed data of other workers.
- Communication Because data dependency is one of the main reasons for communication

among workers, communication is always triggered when the intermediate data is sent to workers.

- Scheduling For a job or program, when the number of computation parts (tasks) or data pieces is more than the number of available workers, a scheduler selects a sequence of tasks or data pieces to be assigned to the workers. It is worth noting that the resource allocator performs the actual mapping of the computation or data pieces to workers, while the scheduler only picks the next part from the queue of unassigned tasks based on a set of rules called the scheduling policy. For multiple jobs or programs, a scheduler selects a sequence of jobs or programs to be run on the distributed computing system. In this case, scheduling is also necessary when system resources are not sufficient to simultaneously run multiple jobs or programs.

5.2.1 Motivation for Programming Paradigms

Because handling the whole data flow of parallel and distributed programming is very time-consuming and requires specialized knowledge of programming, dealing with these issues may affect the productivity of the programmer and may even result in affecting the program's time to market. Furthermore, it may detract the programmer from concentrating on the logic of the program itself.

Therefore, parallel and distributed programming paradigms or models are offered to abstract many parts of the data flow from users.

In other words, these models aim to provide users with an abstraction layer to hide implementation details of the data flow which users formerly ought to write codes for. Therefore, simplicity of writing parallel programs is an important metric for parallel and distributed programming paradigms. Other motivations behind parallel and distributed programming models are (1) to improve productivity of programmers, (2) to decrease programs' time to market, (3) to leverage underlying resources more efficiently, (4) to increase system throughput, and (5) to support higher levels of abstraction [40].

MapReduce, Hadoop, and Dryad are three of the most recently proposed parallel and distributed programming models. They were developed for information retrieval applications but have been shown to be applicable for a variety of important applications [41]. Further, the loose coupling of components in these paradigms makes them suitable for VM implementation and leads to much bet- ter fault tolerance and scalability for some applications than traditional parallel computing models such as MPI [42–44].

5.2.2 MapReduce, Twister, and Iterative MapReducetive

This software framework abstracts the data flow of running a parallel program on a distributed computing system by providing users with two interfaces in the form of two functions: Map and Reduce. Users can override these two functions to interact with and manipulate the data flow of running their programs.

Figure 6.1 illustrates the logical data flow from the Map to the Reduce function in MapReduce frameworks. In this framework the "value" part of the data, (key, value), is the actual data, and the "key" part is only used by the MapReduce controller to control the data flow .

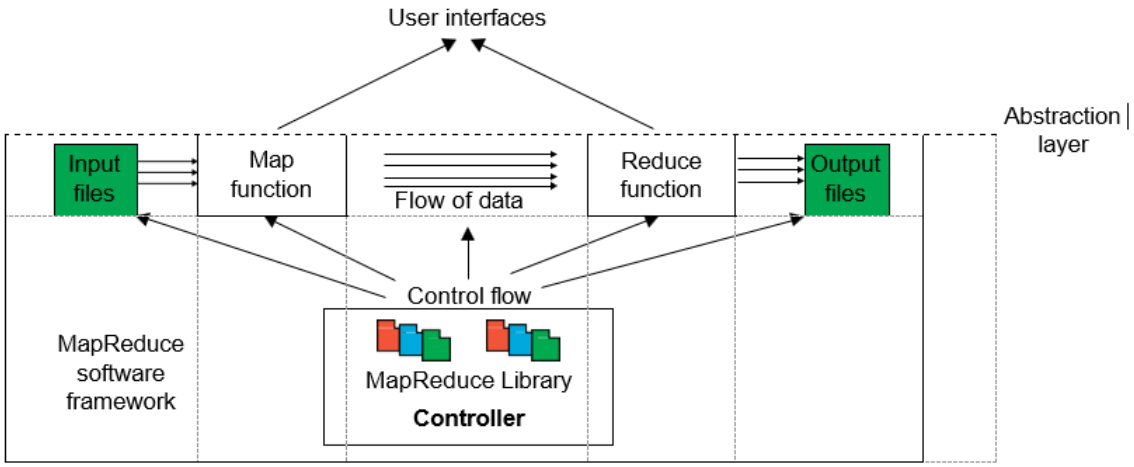


FIGURE 6.1

MapReduce framework: Input data flows through the Map and Reduce functions to generate the output result under the control flow using MapReduce software library. Special user interfaces are used to access the Map and Reduce resources.

5.2.2.1 Formal Definition of MapReduce

The MapReduce software framework provides an abstraction layer with the data flow and flow of control to users, and hides the implementation of all data flow steps such as data partitioning, mapping, synchronization, communication, and scheduling. Here, although the data flow in such frameworks is predefined, the abstraction layer provides two well-defined interfaces in the form of two functions: Map and Reduce . These two main functions can be overridden by the user to achieve specific objectives. Figure 6.1 shows the MapReduce framework with data flow and control flow.

Therefore, the user overrides the Map and Reduce functions first and then invokes the provided MapReduce (Spec, & Results) function from the library to start the flow of data. The MapReduce function, MapReduce (Spec, & Results), takes an important parameter which is a specification object, the Spec. This object is first initialized inside the user's program, and then the user writes code to fill it with the names of input and output files, as well as other optional tuning parameters. This object is also filled with the name of the Map and Reduce functions to identify these user- defined functions to the MapReduce library.

The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below. The Map and Reduce are two major subroutines. They will be called to implement the desired function performed in the main program.

```
Map Function (. . )
  {
    ... ...
  }
Reduce Function (... )
  {
```

```
    ... ...
  }
Main Function (. .. )
  {
    Initialize Spec object
    ... ...
    MapReduce (Spec, & Results)
  }
```

5.2.2.2 MapReduce Logical Data Flow

The input data to both the Map and the Reduce functions has a particular structure. This also pertains for the output data. The input data to the Map function is in the form of a (key, value) pair. For example, the key is the line offset within the input file and the value is the content of the line. The output data from the Map function is structured as (key, value) pairs called intermediate (key, value) pairs. In other words, the user-defined Map function processes each input (key, value) pair and produces a number of (zero, one, or more) intermediate (key, value) pairs. Here, the goal is to process all input (key, value) pairs to the Map function in parallel (Figure 6.2).

In turn, the Reduce function receives the intermediate (key, value) pairs in the form of a group of intermediate values associated with one intermediate key, (key, [set of values]). In fact, the MapReduce framework forms these groups by first sorting the intermediate (key, value) pairs and then grouping values with the same key. It should be noted that the data is sorted to simplify the grouping process. The Reduce function processes each (key, [set of values]) group and produces a set of (key, value) pairs as output.

To clarify the data flow in a sample MapReduce application, one of the well-known MapReduce problems, namely word count, to count the number of occurrences of each word in a collection of documents is presented here. Figure 6.3 demonstrates the data flow of the word-count problem for a simple input file containing only two lines as follows: (1) “most people ignore most poetry” and(2) “most poetry ignores most people.” In this case, the Map function simultaneously produces a number of intermediate (key, value) pairs for each line of content so that each word is the inter- mediate key with 1 as its intermediate value; for example, (ignore, 1). Then the MapReduce librarycollects all the generated intermediate (key, value) pairs and sorts them to group the 1’s for identical words; for example, (people, [1,1]). Groups are then sent to the Reduce function in parallel so that it can sum up the 1 values for each word and generate the actual number of occurrence for each word in the file; for example, (people, 2).

of (key, value) pairs as output.

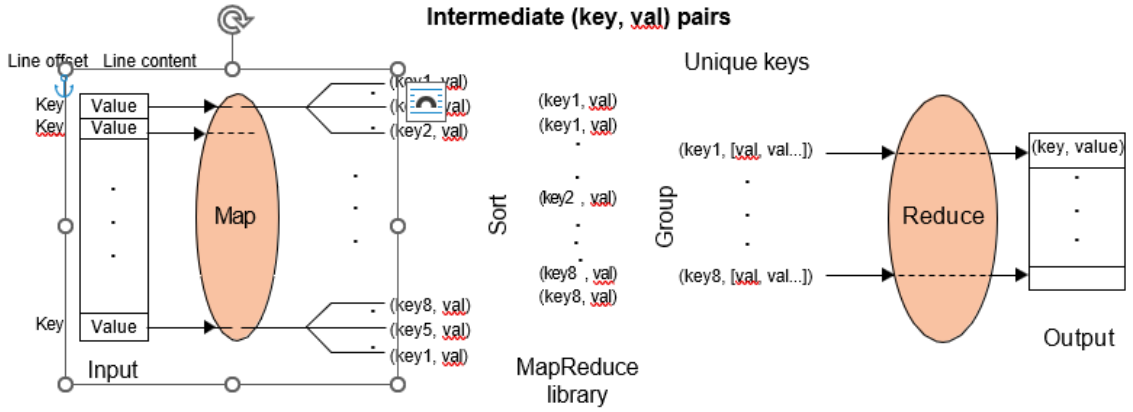


FIGURE 6.2

MapReduce logical data flow in 5 processing stages over successive (key, value) pairs.

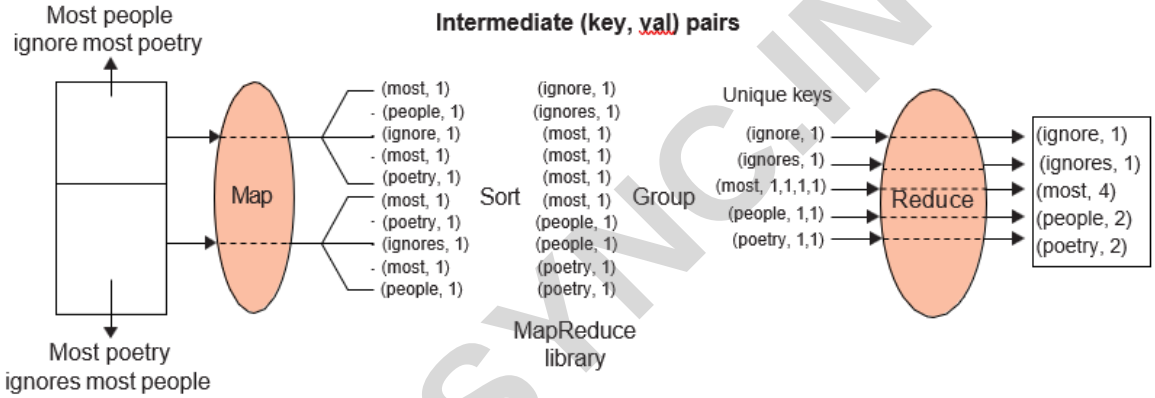


FIGURE 6.3

The data flow of a word-count problem using the MapReduce functions (Map, Sort, Group and Reduce) in a cascade operations.

5.2.2.2 Formal Notation of MapReduce Data Flow

The Map function is applied in parallel to every input (key, value) pair, and produces new set of intermediate (key, value) pairs [37] as follows:

$$(key1, val1) \xrightarrow{Map\ Function} List\ (key2, val2) \qquad (6.1)$$

Then the MapReduce library collects all the produced intermediate (key, value) pairs from all input (key, value) pairs, and sorts them based on the "key" part. It then groups the values of all occur- rences of the same key. Finally, the Reduce function is applied in parallel to each group producing the collection of values as output as illustrated here:

$$(key2, List\ (val2)) \xrightarrow{Reduce\ Function} List\ (val2) \qquad (6.2)$$

5.2.2.3 Strategy to solve MapReduce Problems

Therefore, finding unique keys is the starting point to solving a typical MapReduce problem. Then the intermediate (key, value) pairs as the output of the Map function will be automatically found. The following three examples explain how to define keys and values in

such problems:

Problem 1: Counting the number of occurrences of each word in a collection of documents
Solution: unique "key": each word, intermediate "value": number of occurrences
Problem 2: Counting the number of occurrences of words having the same size, or the same number of letters, in a collection of documents
Solution: unique "key": each word, intermediate "value": size of the word
Problem 3: Counting the number of occurrences of anagrams in a collection of documents. Anagrams are words with the same set of letters but in a different order (e.g., the words "listen" and "silent").
Solution: unique "key": alphabetically sorted sequence of letters for each word (e.g., "eilnst"), intermediate "value": number of occurrences.

5.2.2.4 MapReduce Actual data and Control Flow

The main responsibility of the MapReduce framework is to efficiently run a user's program on a distributed computing system. Therefore, the MapReduce framework meticulously handles all partitioning, mapping, synchronization, communication, and scheduling details of such data flows [48,49]. We summarize this in the following distinct steps:

1.Data partitioning The MapReduce library splits the input data (files), already stored in GFS, into M pieces that also correspond to the number of map tasks.

2. Computation partitioning This is implicitly handled (in the MapReduce framework) by obliging users to write their programs in the form of the Map and Reduce functions. Therefore, the MapReduce library only generates copies of a user program (e.g., by a fork system call) containing the Map and the Reduce functions, distributes them, and starts them up on a number of available computation engines.

3.Determining the master and workers The MapReduce architecture is based on a master-worker model. Therefore, one of the copies of the user program becomes the master and the rest become workers. The master picks idle workers, and assigns the map and reduce tasks to them.A map/reduce worker is typically a computation engine such as a cluster node to run map/ reduce tasks by executing Map/Reduce functions. Steps 4–7 describe the map workers.

4.Reading the input data (data distribution) Each map worker reads its corresponding portion of the input data, namely the input data split, and sends it to its Map function. Although a map worker may run more than one Map function, which means it has been assigned more than one input data split, each worker is usually assigned one input split only.

5. Map function Each Map function receives the input data split as a set of (key, value) pairs to process and produce the intermediated (key, value) pairs.

6. Combiner function This is an optional local function within the map worker which applies to intermediate (key, value) pairs. The user can invoke the Combiner function inside the user program. The Combiner function runs the same code written by users for the Reduce function as its functionality is identical to it. The Combiner function merges the local data of each map worker before sending it over the network to effectively reduce its communication costs. As mentioned in our discussion of logical data flow, the MapReduce framework sorts and groups the data before it is processed by the Reduce function. Similarly, the MapReduce framework

will also sort and group the local data on each map worker if the user invokes the Combiner function.

7.Partitioning function As mentioned in our discussion of the MapReduce data flow, the intermediate (key, value) pairs with identical keys are grouped together because all values inside each group should be processed by only one Reduce function to generate the final result. However, in real implementations, since there are M map and R reduce tasks, intermediate (key, value) pairs with the same key might be produced by different map tasks, although they should be grouped and processed together by one Reduce function only. Therefore, the intermediate (key, value) pairs produced by each map worker are partitioned into R regions, equal to the number of reduce tasks, by the Partitioning function to guarantee that all (key, value) pairs with identical keys are stored in the same region. As a result, since reduce worker i reads the data of region i of all map workers, all (key, value) pairs with the same key will be gathered by reduce worker i accordingly (see Figure 6.4). To implement this technique, a Partitioning function could simply be a hash function (e.g., Hash(key) mod R) that forwards the data into particular regions. It is also worth noting that the locations of the buffered data in these R partitions are sent to the master for later forwarding of data to the reduce workers.
Figure 6.5 shows the data flow implementation of all data flow steps. The following are two networking steps:

8.Synchronization MapReduce applies a simple synchronization policy to coordinate map workers with reduce workers, in which the communication between them starts when all map tasks finish.

9.Communication Reduce worker i, already notified of the location of region i of all map workers, uses a remote procedure call to read the data from the respective region of all map workers. Since all reduce workers read the data from all map workers, all-to-all communication among all map and reduce workers, which incurs network congestion, occurs in the network. This issue is one of the major bottlenecks in increasing the performance of such systems [50–52]. A data transfer module was proposed to schedule data transfers independently [55].

Steps 10 and 11 correspond to the reduce worker domain:

10.Sorting and Grouping When the process of reading the input data is finalized by a reduce worker, the data is initially buffered in the local disk of the reduce worker. Then the reduce worker groups intermediate (key, value) pairs by sorting the data based on their keys, followed by grouping all occurrences of identical keys. Note that the buffered data is sorted and grouped because the number of unique keys produced by a map worker may be more than R regions in which more than one key exists in each region of a map worker (see Figure 6.4).

11.Reduce function The reduce worker iterates over the grouped (key, value) pairs, and

for each unique key, it sends the key and corresponding values to the Reduce function. Then this function processes its input data and stores the output results in predetermined files in the user's program.

5.2.2.5 Twister and Iterative MapReduce Flow

The two major sources of parallel overhead are load imbalance and communication (which is equivalent to synchronization overhead as communication synchronizes parallel units [threads or

processes] in Categories 2 and 6 of Table 6.10). The communication over- head in MapReduce can be quite high, for two reasons:

MapReduce reads and writes via files, whereas MPI transfers information directly between nodes over the network

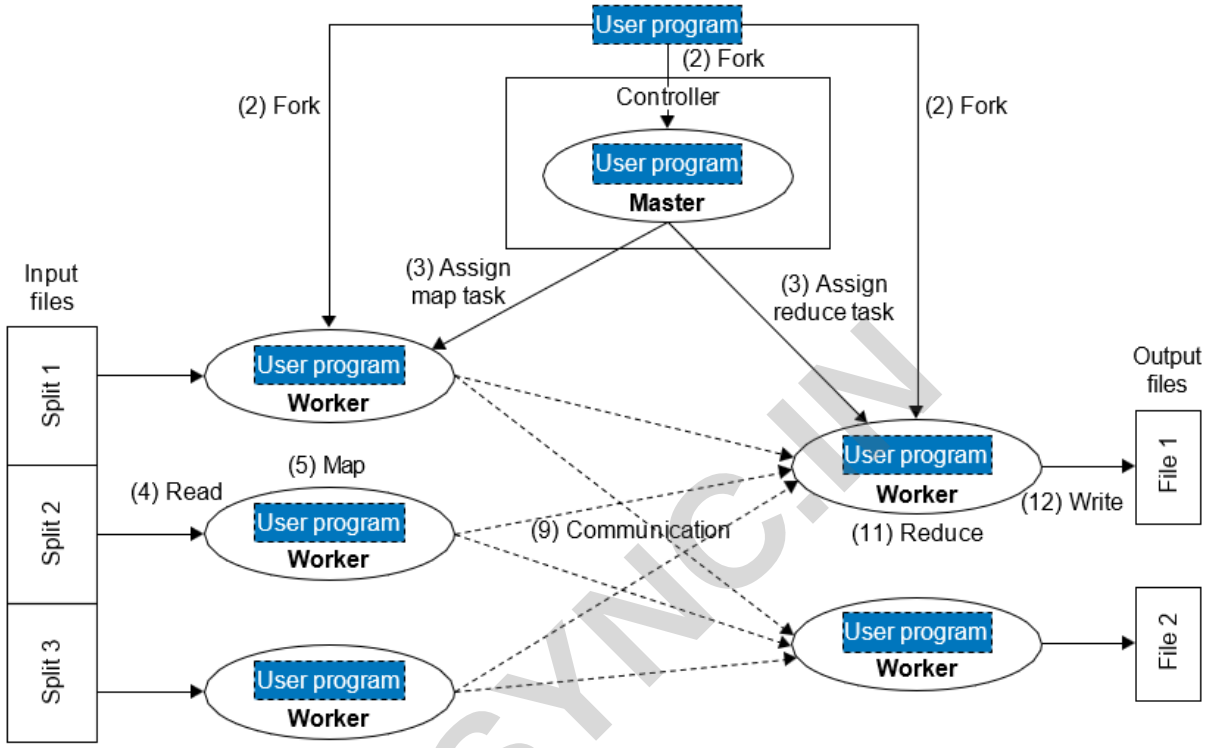


FIGURE 6.6

Control flow implementation of the MapReduce functionalities in Map workers and Reduce workers (running user programs) from input files to the output files under the control of the master user program.

The same phenomenon is seen in all "classic parallel" loosely synchronous applications which typically exhibit an iteration structure over compute phases followed by communication phases. We can address the performance issues with two important changes:

1. Stream information between steps without writing intermediate steps to disk.
2. Use long-running threads or processors to communicate the δ (between iterations) flow.

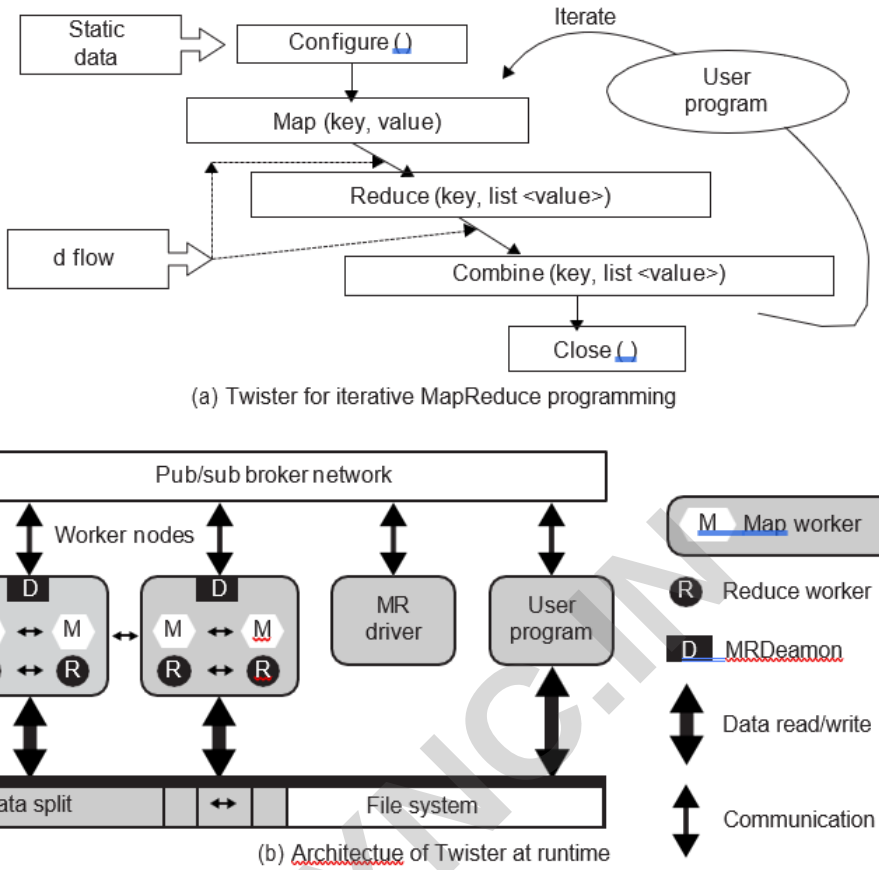


FIGURE 6.7

Twister: An iterative MapReduce programming paradigm for repeated MapReduce executions.

5.2.3 Hadoop Library from Apache

Hadoop is an open source implementation of MapReduce coded and released in Java (rather than C) by Apache. The Hadoop implementation of MapReduce uses the Hadoop Distributed File System (HDFS) as its underlying layer rather than GFS.

The Hadoop core is divided into two fundamental layers:

the MapReduce engine and HDFS.

The MapReduce engine is the computation engine run ning on top of HDFS as its data storage manager.

The following two sections cover the details of these two fundamental layers. HDFS: HDFS is a distributed file system inspired by GFS that organizes files and stores their data on a distributed computing system.

HDFS Architecture: HDFS has a master/slave architecture containing a single NameNode as the master and a number of DataNodes as workers (slaves). To store a file in this architecture, HDFS splits the file into fixed-size blocks (e.g., 64MB) and stores them on workers (DataNodes). The mapping of blocks to DataNodes is determined by the NameNode. The NameNode (master) also manages the file system's metadata and namespace. In such systems, the namespace is the area maintaining the metadata, and metadata refers to all the information stored by a file system that is needed for overall management of all files.

HDFS Features: Distributed file systems have special requirements, such as performance, scalability, concurrency control, fault tolerance, and security requirements [62], to operate efficiently. However, because HDFS is not a general-purpose file system, as it only executes specific types of applications, it does not need all the requirements of a general distributed file system.

For example, security has never been supported for HDFS systems. The following discussion highlights two important characteristics of HDFS to distinguish it from other generic distributed file systems. HDFS Fault Tolerance: One of the main aspects of HDFS is its fault tolerance characteristic. Since Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception.

Therefore, Hadoop considers the following issues to fulfill reliability requirements of the file system:

• **Block replication** To reliably store data in HDFS, file blocks are replicated in this system. In other words, HDFS stores a file as a set of blocks and each block is replicated and distributed across the whole cluster. The replication factor is set by the user and is three by default.

• **Replica placement** The placement of replicas is another factor to fulfill the desired fault tolerance in HDFS. Although storing replicas on different nodes (DataNodes) located in different racks across the whole cluster provides more reliability, it is sometimes ignored as the cost of communication between two nodes in different racks is relatively high in comparison with that of different nodes located in the same rack. Therefore, sometimes HDFS compromises its reliability to achieve lower communication costs.

• **Heartbeats and Blockreports** are periodic messages sent to the NameNode by each DataNode in a cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly, while each Blockreport contains a list of all blocks on a DataNode .

The NameNode receives such messages because it is the sole decision maker of all replicas in the system.

• **HDFS High-Throughput Access to Large Data Sets (Files):** Because HDFS is primarily designed for batch processing rather than interactive processing, data access throughput in HDFS is more important than latency. Also, because applications run on HDFS typically have large data sets, individual files are broken into large blocks (e.g., 64 MB) to allow HDFS to decrease the amount of metadata storage required per file. This provides two advantages: The list of blocks per file will shrink as the size of individual blocks increases, and by keeping large amounts of data sequentially within a block, HDFS provides fast streaming reads of data.

HDFS Operation: The control flow of HDFS operations such as write and read can properly highlight roles of the NameNode and DataNodes in the managing operations. In this section, the control flow of the main operations of HDFS on files is further described to manifest the interaction between the user, the NameNode, and the DataNodes in such systems [63].

• **Reading a file** To read a file in HDFS, a user sends an “open” request to the NameNode to get the location of file blocks. For each file block, the NameNode returns the address of a set of DataNodes containing replica information for the requested file. The number of addresses depends on the number of block replicas. Upon receiving such information, the user calls the read function to connect to the closest DataNode containing the first block of the file. After the first block is streamed from the respective DataNode to the user, the established connection is terminated and the same process is repeated for all blocks of the requested file until the whole file is streamed to the user.

• **Writing to a file** To write a file in HDFS, a user sends a “create” request to the NameNode to create a new file in the file system namespace. If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the write function. The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode. Since each file block needs to be replicated by a predefined factor, the

data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.

5.2.3.1 Architecture of MapReduce in Hadoop:

The topmost layer of Hadoop is the MapReduce engine that manages the data flow and control flow of MapReduce jobs over distributed computing systems. Figure 6.11 shows the MapReduce engine architecture cooperating with HDFS. Similar to HDFS, the MapReduce engine has a master/slave architecture consisting of a single JobTracker as the master and a number of TaskTrackers as the slaves (workers).

The JobTracker manages the MapReduce job over a cluster and is responsible for monitoring jobs and assigning tasks to TaskTrackers. The TaskTracker man ages the execution of the map and/or reduce tasks on a single computation node in the cluster. Each TaskTracker node has a number of simultaneous execution slots, each executing either a map or a reduce task. Slots are defined as the number of simultaneous threads supported by CPUs of the TaskTracker node.

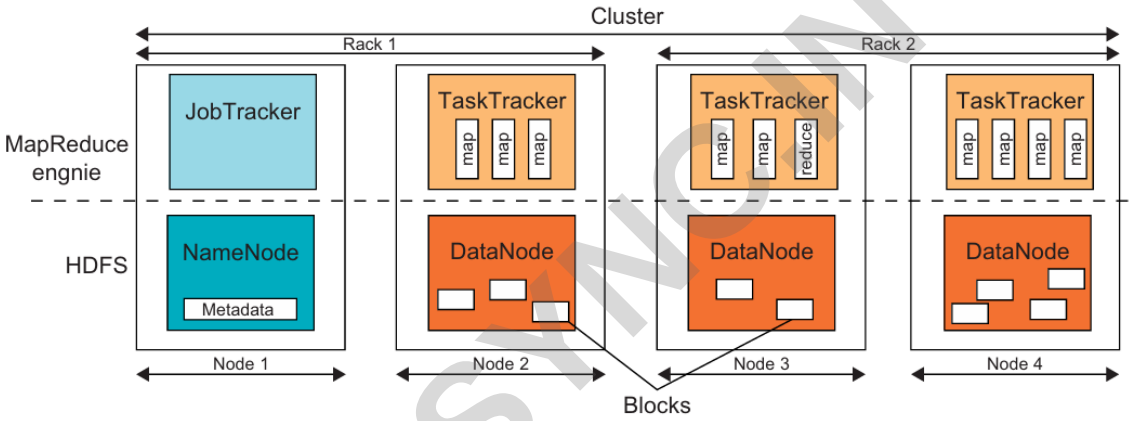


FIGURE 6.11

HDFS and MapReduce architecture in Hadoop where boxes with different shadings refer to different functional nodes applied to different blocks of data.

5.2.3.2 Running a Job in Hadoop:

Three components contribute in running a job in this system: a user node, a JobTracker, and several TaskTrackers. The data flow starts by calling the runJob(conf) function inside a user program run ning on the user node, in which conf is an object containing some tuning parameters for the MapReduce framework and HDFS. The runJob(conf) function and conf are comparable to the MapReduce(Spec, &Results) function and Spec in the first implementation of MapReduce by Google. Figure 6.12 depicts the data flow of running a MapReduce job in Hadoop.

• **Job Submission** Each job is submitted from a user node to the JobTracker node that might be situated in a different node within the cluster through the following procedure:

• Auser node asks for a new job ID from the JobTracker and computes input file splits.

• The user node copies some resources, such as the job's JAR file, configuration file, and computed input splits, to the JobTracker's file system.

• The user node submits the job to the JobTracker by calling the submitJob() function.

• **Task assignment** The JobTracker creates one map task for each computed input split by the user node and assigns the map tasks to the execution slots of the TaskTrackers. The JobTracker considers the localization of the data when assigning the map tasks to the TaskTrackers.The JobTracker also creates reduce tasks and assigns them to the TaskTrackers. The number of reduce tasks is predetermined by the user, and there is no locality consideration in assigning them.

• **Task execution** The control flow to execute a task (either map or reduce) starts inside the TaskTracker by copying the job JAR file to its file system. Instructions inside the job JAR file are executed after launching a Java Virtual Machine (JVM) to run its map or reduce task.

• **Task running check** A task running check is performed by receiving periodic heartbeat messages to the JobTracker from the TaskTrackers. Each heartbeat notifies the JobTracker that the sending TaskTracker is alive, and whether the sending TaskTracker is ready to run a new task.

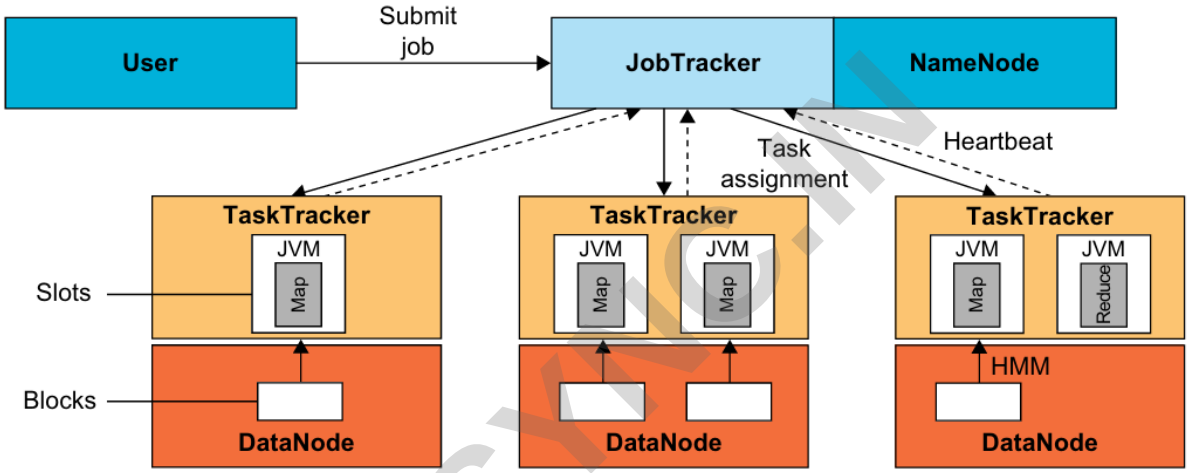


FIGURE 6.12

Data flow in running a MapReduce job at various task trackers using the Hadoop library.

5.2.4 Dryad and DryadLINQ from Microsoft:

Two runtime software environments are reviewed in this section for parallel and distributed comput ing, namely the Dryad and DryadLINQ, both developed by Microsoft.

5.2.4.1 Dryad:

Dryad is more flexible than MapReduce as the data flow of its applications is not dictated/predeter mined and can be easily defined by users. To achieve such flexibility, a Dryad program or job is defined by a directed acyclic graph (DAG) where vertices are computation engines and edges are communication channels between vertices. Therefore, users or application developers can easily specify arbitrary DAGs to specify data flows in jobs.

The job manager has the code to construct the DAG as well as the library to schedule the work running on top of the available resources. Data transfer is done via channels without involving the job manager. Thus, the job manager should not be the performance bottleneck. In summary, the job manager

1. Constructs a job's communication graph (data flow graph) using the application-specific program provided by the user.

2. Collects the information required to map the data flow graph to the underlying resources (computation engine) from the name server.

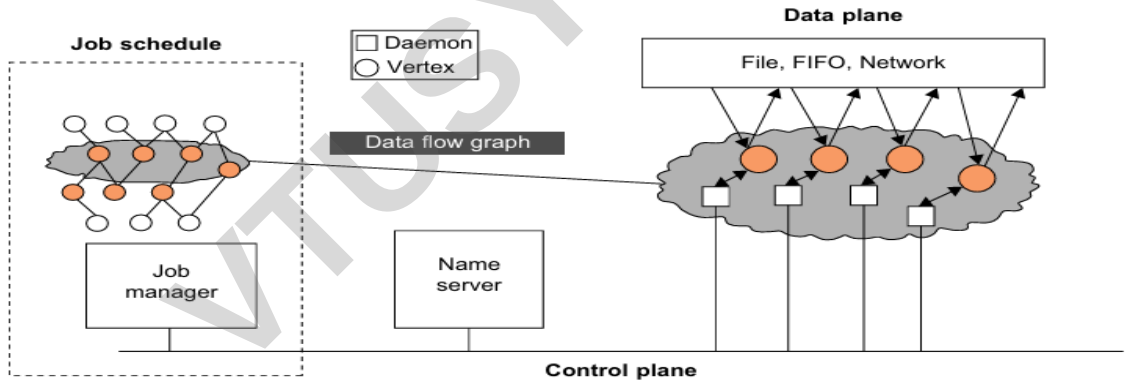
The cluster has a name server which is used to enumerate all the available computing resources in the cluster. Thus, the job manager can contact the name server to get the topology of the whole cluster and make scheduling decisions. A processing daemon runs in each computing node in the cluster. The binary of the program will be sent to the corresponding processing node directly from the job manager. The daemon can be viewed as a proxy so that the job manager can communicate with the remote vertices and monitor the state of the computation. By gathering this information, the name server provides the job manager with a perfect view of the underlying resources and network topology.

Therefore, the job manager is able to:

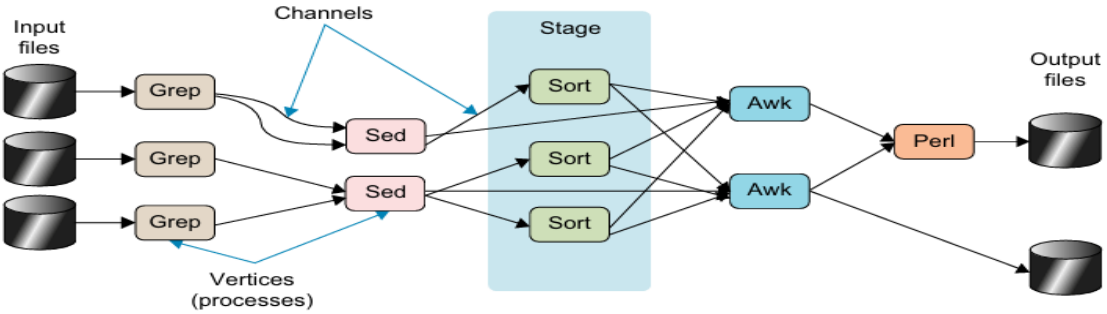
1. Map the data flow graph to the underlying resources.

2. Schedule all necessary communications and synchronization across the respective resources.

The execution of a Dryad job can be considered a 2D distributed set of pipes. Traditional UNIX pipes are 1D pipes, with each node in the pipe as a single program. Dryad's 2D distributed pipe system has multiple processing programs in each vertex node. In this way, large-scale data can be pro cessed simultaneously. Figure 6.13(b) shows the Dryad 2D pipe job structure. During 2D pipe execution, Dryad defines many operations to construct and change the DAG dynamically.



(a) Dryad control and data flow



(b) Dryad job structure

FIGURE 6.13

Dryad framework and its job structure, control and data flow.

(*Courtesy of Isard, et al., ACM SIGOPS Operating Systems Review, 2007 [26]*)

5.2.4.1 DryadLINQ from Microsoft :

DryadLINQ is built on top of Microsoft's Dryad execution framework (see http://research.microsoft .com/en-us/projects/DryadLINQ/). Dryad can perform acyclic task scheduling and run on large-scale servers. The goal of DryadLINQ is to make large-scale, distributed cluster computing available to ordinary programmers. Actually, DryadLINQ, as the name implies, combines two important components: the Dryad distributed execution engine and .NET Language Integrated Query (LINQ). LINQ is particularly for users familiar with a database programming model. Figure 6.14 shows the flow of execution with DryadLINQ. The execution is divided into nine steps as follows:

1. A .NET user application runs, and creates a DryadLINQ expression object. Because of LINQ's deferred evaluation, the actual execution of the expression has not occurred.

2. The application calls ToDryadTable triggering a data-parallel execution. The expression object is handed to DryadLINQ.

3. DryadLINQ compiles the LINQ expression into a distributed Dryad execution plan. The expression is decomposed into subexpressions, each to be run in a separate Dryad vertex. Code and static data for the remote Dryad vertices are generated, followed by the serialization code for the required data types.

4. DryadLINQ invokes a custom Dryad job manager which is used to manage and monitor the execution flow of the corresponding task.

5. The job manager creates the job graph using the plan created in step 3. It schedules and spawns the vertices as resources become available

6. Each Dryad vertex executes a vertex-specific program.

7. When the Dryad job completes successfully it writes the data to the out table(s).

8. The job manager process terminates, and it returns control back to DryadLINQ. DryadLINQ creates the local DryadTable objects encapsulating the output of the execution. The DryadTable objects here might be the input to the next phase.

9. Control returns to the user application. The iterator interface over a DryadTable allows the user to read its contents as .NET objects.

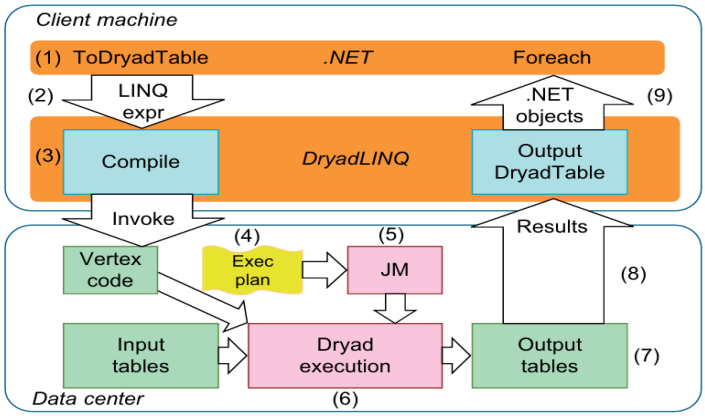


FIGURE 6.14
LINQ-expression execution in DryadLINQ.

5.2.5 Sawzall and Pig Latin High-Level Languages:

Sawzall is a high-level language built on top of Google's MapReduce framework. Sawzall is a scripting language that can do parallel data processing. As with MapReduce, Sawzall can do distributed, fault-tolerant processing of very large-scale data sets, even at the scale of the data collected from the entire Internet. Sawzall was developed by Rob Pike with an initial goal of processing Google's log files. In this regard it was hugely successful and changed a batch day long enterprise into an interactive session, enabling new approaches to using such data to be developed. Figure 6.16 shows the overall model of data flow and processing procedures in the Sawzall framework. Sawzall has recently been released as an open-source project.

First the data is partitioned and processed locally with an on-site processing script. The local data is filtered to get the necessary information. The aggregator is used to get the final results based on the emitted data. Many of Google's applications fit this model. Users write their applications using the Sawzall scripting language. The Sawzall runtime engine translates the cor responding scripts to MapReduce programs running on many nodes. The Sawzall program can harness the power of cluster computing automatically as well as attain reliability from redundant servers.

Here is a simple example of using Sawzall for data processing on clusters (the example is from the Sawzall paper at [31]). Suppose we want to process a set of files with records in each file. Each record contains one floating-point number. We want to calculate the number of records, the sum of the values, and the sum of the squares of the values. The relevant code is as follows:

```
count: table sum of int;
total: table sum of float;
```

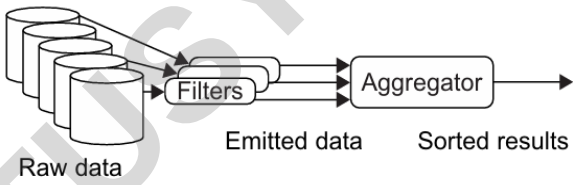


FIGURE 6.16

The overall flow of filtering, aggregating, and collating in Sawzall.


```
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```

The first three lines declare the aggregators as *count*, *total*, and *sum_of_squares*. *table* is a keyword which defines an aggregator type. These particular tables are *sum* tables and they will automatically add up the values emitted to them. For each input record, Sawzall initializes the predefined variable input to the uninterpreted byte string of the input. The line *x: float = input;* converts the input into a floating-point number and stores it in local variable *x*. The three *emit* statements send the intermediate values to the aggregators. One can translate the Sawzall scripts into MapReduce programs and run them with multiple servers.

5.2.5.1 Pig Latin:

Pig Latin is a high-level data flow language developed by Yahoo! that has been implemented on top of Hadoop in the Apache Pig project. Pig Latin, Sawzall and DryadLINQ are different approaches to building languages on top of MapReduce and its extensions. They are compared in Table 6.7.

DryadLINQ is building directly on SQL while the other two languages are of NOSQL heritage, although Pig Latin supports major SQL constructs including Join, which is absent in Sawzall. Each language automates the parallelism, so you only think about manipulation of individual elements and then invoke supported collective operations**.**

Table 6.7 Comparison of High-Level Data Analysis Languages

	Sawzall	Pig Latin	DryadLINQ
Origin	Google	Yahoo!	Microsoft
Data Model	Google protocol buffer or basic	Atom, Tuple, Bag, Map	Partition file
Typing	Static	Dynamic	Static
Category	Interpreted	Compiled	Compiled
Programming Style	Imperative	Procedural: sequence of declarative steps	Imperative and declarative
Similarity to SQL	Least	Moderate	A lot!
Extensibility (User-Defined Functions)	No	Yes	Yes
Control Structures	Yes	No	Yes
Execution Model	Record operations + fixed aggregations	Sequence of MapReduce operations	DAGs
Target Runtime	Google MapReduce	Hadoop (Pig)	Dryad

Table 6.8 Pig Latin Data Types

Data Type	Description	Example
Atom	Simple atomic value	'Clouds'
Tuple	Sequence of fields of any Pig Latin type	('Clouds', 'Grids')
Bag	Collection of tuples with each member of the bag allowed a different schema	{ ('Clouds', 'Grids') ('Clouds', ('IaaS', 'PaaS') }
Map	A collection of data items associated with a set of keys; the keys are a bag of atomic data	['Microsoft' → { ('Windows') ('Azure') } 'Redhat' → 'Linux']

Table 6.9 Pig Latin Operators

Command	Description
LOAD	Read data from the file system.
STORE	Write data to the file system.
FOREACH GENERATE	Apply an expression to each record and output one or more records.
FILTER	Apply a predicate and remove records that do not return true.
GROUP/COGROUP	Collect records with the same key from one or more inputs.
JOIN	Join two or more inputs based on a key.
CROSS	Cross product two or more inputs.
UNION	Merge two or more data sets.
SPLIT	Split data into two or more sets, based on filter conditions.
ORDER	Sort records based on a key.
DISTINCT	Remove duplicate tuples.
STREAM	Send all records through a user-provided binary.
DUMP	Write output to stdout.
LIMIT	Limit the number of records.

5.2.6 Mapping Applications to Parallel and Distributed Systems:

Mapping applications to different hardware and software in terms of five application architectures. These initial five categories are listed in Table 6.10, followed by a sixth emerging category to describe data-intensive computing .

The original classifications largely described simulations and were not aimed directly at data analysis. It is instructive to briefly summarize them and then explain the new category.

Category 1 was popular 20 years ago, but is no longer significant. It describes applications that can be parallelized with lock-step operations controlled by hardware. Such a configuration would run on SIMD (single-instruction and multiple-data) machines, whereas

category 2 is now much more important and corresponds to a SPMD (single-program and multiple data) model running on MIMD (multiple instruction multiple data) machines**.**

Category3consists of a synchronously interacting objects and is often considered the people's view of a typical parallel problem. It probably does describe the concurrent threads in a modern operating system,as well as some important applications,suchasevent-drivenven simulations and areasuch.

Category 4 is the simplest algorithmically, with disconnected parallel components. However, the importance of this category has probably grown since the original 1988 analysis when it was esti

mated to account for 20 percent of all parallel computing. Both grids and clouds are very natural for this class, which does not need high-performance communication between different nodes.
Category 5 refers to the coarse-grained linkage of different "atomic" problems, and this was fully covered in Section 5.5. This area is clearly common and is expected to grow in importance.
Remember the critical observation in Section 5.5 that we use a two-level programming model with the metaproblem (workflow) linkage specified in one fashion and the component problems with approaches such as those in this chapter. Grids or clouds are suitable for metaproblems as coarse grained decomposition does not usually require stringent performance.

5.3Programming Support of Google App Engine:

5.3.1Programming the Google App Engine:

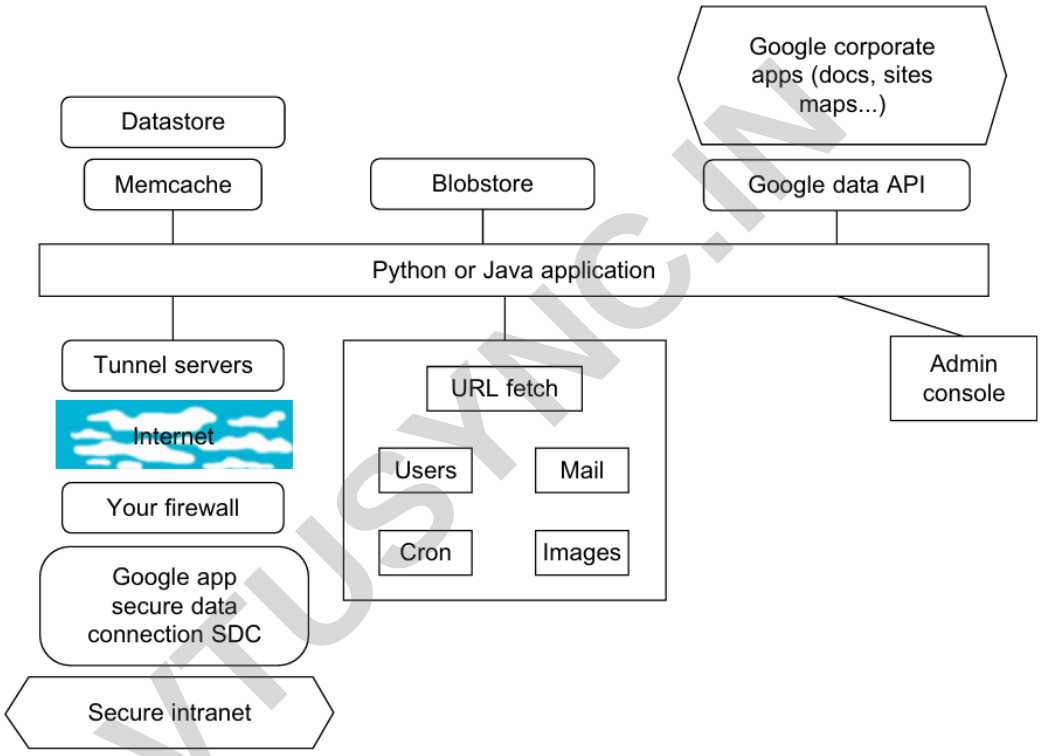


FIGURE 6.17
Programming environment for Google AppEngine.

Several web resources (e.g., http://code.google.com/appengine/) and specific books and articles (e.g., www.byteonic.com/2009/overview-of-java-support-in-google-app-engine/) discuss how to program GAE. Figure 6.17 summarizes some key features of GAE programming model for two supported languages: Java and Python.
A client environment that includes an Eclipse plug-in for Java allows you to debug your GAE on your local machine. Also, the GWT Google Web Toolkit is available for Java web application developers.
Developers can use this, or any other language using a JVM based interpreter or compiler, such as JavaScript or Ruby. Python is often used with frameworks such as Django and CherryPy, but Google also supplies a built in webapp Python environment.
There are several powerful constructs for storing and accessing data. The data store is a NOSQL data management system for entities that can be, at most, 1MB in size and are labeled by a set of

schema-less properties. Queries can retrieve entities of a given kind filtered and sorted by the values of the properties. Java offers Java Data Object (JDO) and Java Persistence API (JPA) interfaces implemented by the open source Data Nucleus Access platform, while Python has a SQL-like query language called GQL. The data store is strongly consistent and uses optimistic concurrency control.An update of an entity occurs in a transaction that is retried a fixed number of times if other processes are trying to update the same entity simultaneously.

The data store implements transactions across its distributed network using "entity groups." Atransaction manipulates entities within a single group. Entities of the same group are stored together for efficient execution of transactions.

Your GAE application can assign entities to groups when the entities are created. The performance of the data store can be enhanced by in-memory caching using the memcache, which can also be used independently of the data store. Recently, Google added the blobstore which is suitable for large files as its size limit is 2 GB.

There are several mechanisms for incorporating external resources. The Google SDC Secure Data Connection can tunnel through the Internet and link your intranet to an external GAE application. The URL Fetch operation provides the ability for applications to fetch resources and communicate with other hosts over the Internet using HTTP and HTTPS requests.

There is a specialized mail mechanism to send e-mail from your GAE application. Applications can access resources on the Internet, such as web services or other data, using GAE's URL fetch service. The URL fetch service retrieves web resources using the same high speed Google infrastructure that retrieves web pages for many other Google products. There are dozens of Google "corporate" facilities including maps, sites, groups, calendar, docs, and YouTube, among others. These support the Google Data API which can be used inside GAE.

5.3.2 Google File System (GFS):

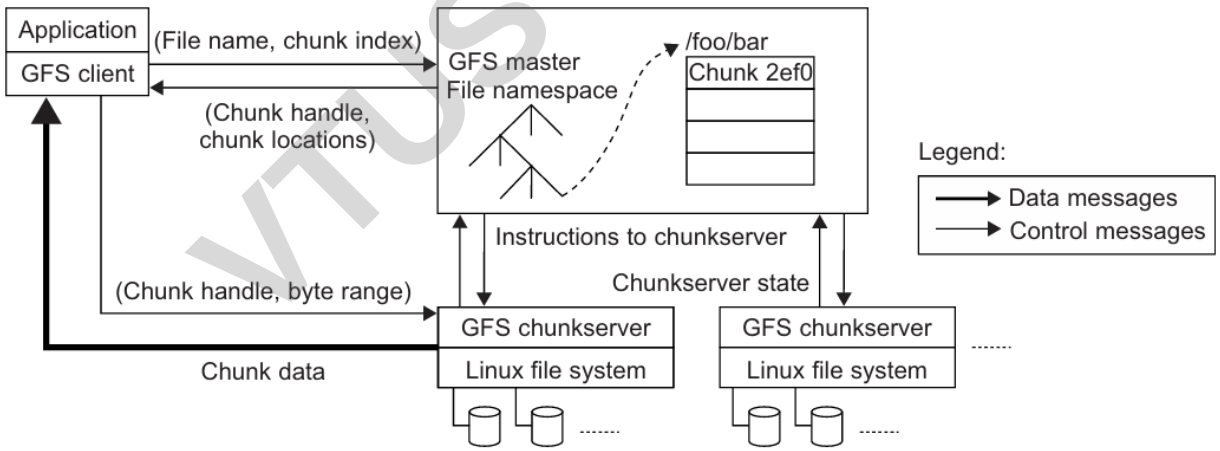


FIGURE 6.18

Architecture of Google File System (GFS).

GFS was built primarily as the fundamental storage service for Google's search engine. As the size of the web data that was crawled and saved was quite substantial, Google needed a distribu ted file system to redundantly store massive amounts of data on cheap and unreliable computers. There are several assumptions concerning GFS. One is related to the characteristic of the cloud computing hardware infrastructure (i.e., the high component failure rate). As servers are composed of inexpensive commodity components, it is the norm rather than the exception that concurrent fail ures will occur all the time. Another concerns the file size in GFS.

Figure 6.18 shows the GFS architecture. It is quite obvious that there is a single master in the whole cluster. Other nodes act as the chunk servers for storing data, while the single master stores the metadata. The file system namespace and locking facilities are managed by the master. The master periodically communicates with the chunk servers to collect management information as well as give instructions to the chunk servers to do work such as load balancing or fail recovery.

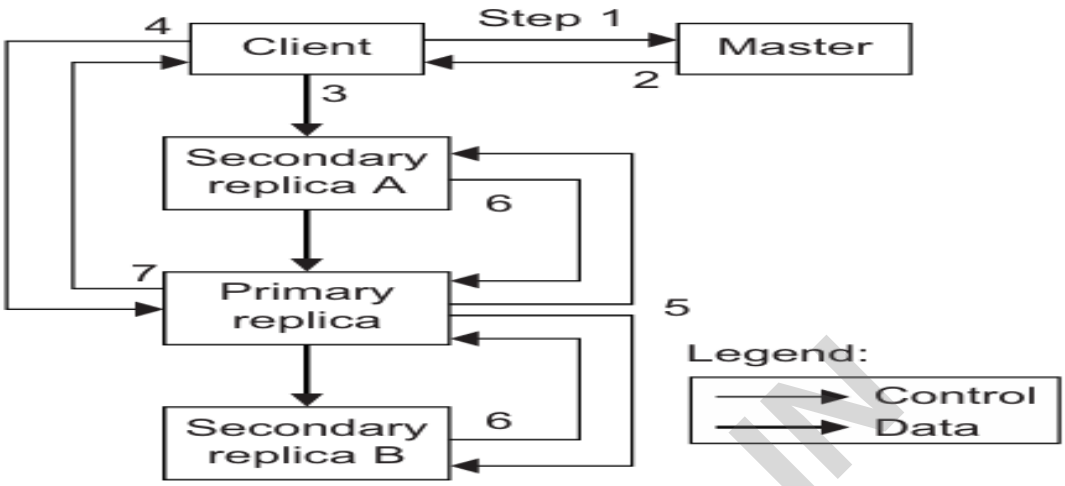


FIGURE 6.19

Data mutation sequence in GFS.

Figure 6.19 shows the data mutation (write, append operations) in GFS. Data blocks must be created for all replicas. The goal is to minimize involvement of the master. The mutation takes the following steps:

1. The client asks the master which chunk server holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).

2. The master replies with the identity of the primary and the locations of the other (secondary) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary becomes unreachable or replies that it no longer holds a lease.

3. The client pushes the data to all the replicas. A client can do so in any order. Each chunk server will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunk server is the primary.

4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial order.

5. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.

6. The secondaries all reply to the primary indicating that they have completed the operation.

7. The primary replies to the client. Any errors encountered at any replicas are reported to the client. In case of errors, the write corrects at the primary and an arbitrary subset of the secondary replicas. The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps 3 through 7 before falling back to a retry from the beginning of the write.

5.3.3 BigTable, Google's NOSQL System:

BigTable was designed to provide a service for storing and retrieving structured and semistructured data. BigTable applications include storage of web pages, per-user data, and geographic locations. Here we use web pages to represent URLs and their associated data, such as contents, crawled metadata, links, anchors, and page rank values. Per-user data has information for a specific user and includes such data as user preference settings, recent queries/search results, and the user's e-mails. Geographic locations areusedinGoogle's well-known Google Earth software. Geographic locations include physical entities (shops, restaurants, etc.), roads, satellite image data, and user annotations. The BigTable system is built on top of an existing Google cloud infrastructure. BigTable uses the following building blocks:

1. GFS: stores persistent state

2. Scheduler: schedules jobs involved in BigTable serving

3. Lock service: master election, location bootstrapping

4. MapReduce: often used to read/write BigTable data.

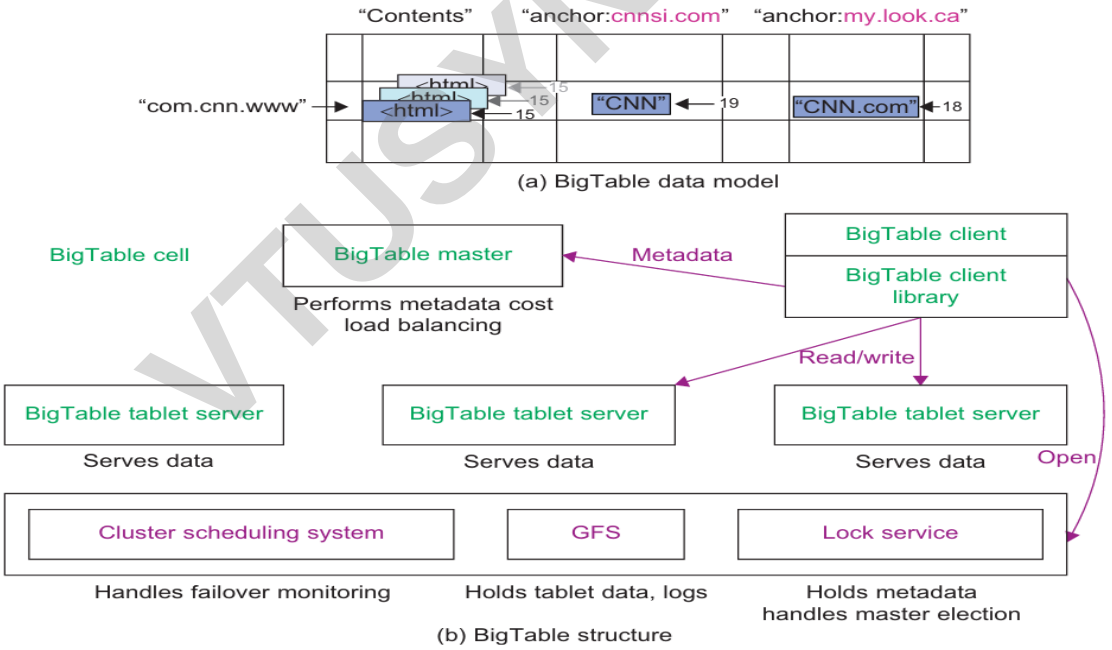


FIGURE 6.20

BigTable data model and system structure.

The map is indexed by row key, column key, and timestamp—that is, (row: string, column: string, time: int64) maps to string (cell contents). Rows are ordered in lexicographic order by row key. The row range for a table is dynamically partitioned and each row range is called "Tablet." Syntax for columns is shown as a (family:qualifier) pair. Cells can store multiple versions of data with timestamps.

Such a data model is a good match for most of Google's (and other organizations') applications. For rows, Name is an arbitrary string and access to data in a row is atomic. This is different from the traditional relational database which provides abundant atomic operations (transactions). Row creation is implicit upon storing data. Rows are ordered lexicographically, that is, close together lexicographically, usually on one or a small number of machines.

Large tables are broken into tablets at row boundaries. A tablet holds a contiguous range of rows. Clients can often choose row keys to achieve locality. The system aims for about 100MB to 200MB of data per tablet. Each serving machine is responsible for about 100 tablets. This can achieve faster recovery times as 100 machines each pick up one tablet from the failed machine. This also results in fine-grained load balan cing, that is, migrating tablets away from the overloaded machine. Similar to the design in GFS, a master machine in BigTable makes load-balancing decisions.

5.3.3.1 Tablet Location Hierarchy :

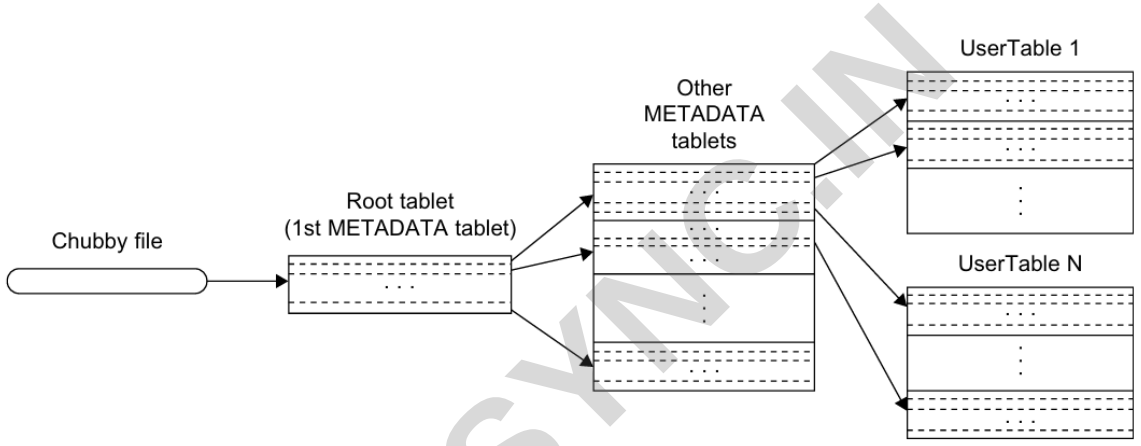


FIGURE 6.21
Tablet location hierarchy in using the BigTable.

Figure 6.21 shows how to locate the BigTable data starting from the file stored in Chubby. The first level is a file stored in Chubby that contains the location of the root tablet. The root tablet contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets. The root tablet is just the first tablet in the METADATA table, but is treated specially; it is never split to ensure that the tablet location hierarchy has no more than three levels.

The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row. BigTable includes many optimizations and fault-tolerant features. Chubby can guarantee the availability of the file for finding the root tablet. The BigTable master can quickly scan the tablet servers to determine the status of all nodes.

Tablet users use compaction to store data efficiently. Shared logs are used for logging the operations of multiple tablets so as to reduce the log space as well as keep the system consistent.

5.3.4 Chubby, Google's Distributed Lock Service

Chubby is intended to provide a coarse-grained locking service. It can store small files inside Chubby storage which provides a simple namespace as a file system tree. The files stored in Chubby

are quite small compared to the huge files in GFS. Based on the Paxos agreement protocol, the Chubby system can be quite reliable despite the failure of any member node.

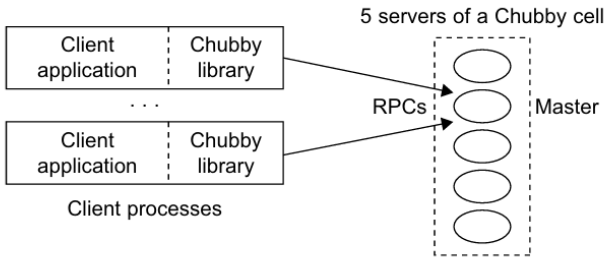


FIGURE 6.22

Structure of Google Chubby for distributed lock service.

Figure 6.22 shows the overall architecture of the Chubby system. Each Chubby cell has five servers inside. Each server in the cell has the same file system namespace. Clients use the Chubby library to talk to the servers in the cell. Client applications can perform various file operations on any server in the Chubby cell. Servers run the Paxos protocol to make the whole file system reliable and consistent. Chubby has become Google's primary internal name service. GFS and BigTable use Chubby to elect a primary from redundant replicas.

5.4 PROGRAMMING ON AMAZON AWS AND MICROSOFT AZURE

5.4.1 Programming on Amazon EC2 :

Amazon was the first company to introduce VMs in application hosting. Customers can rent VMs instead of physical machines to run their own applications. By using VMs, customers can load any software of their choice. The elastic feature of such a service is that a customer can create, launch, and terminate server instances as needed, paying by the hour for active servers. Amazon provides several types of preinstalled VMs. Instances are often called Amazon Machine Images (AMIs) which are preconfigured with operating systems based on Linux or Windows, and additional software. Table 6.12 defines three types of AMI. Figure 6.24 shows an execution environment. AMIs are the templates for instances, which are running VMs.

The workflow to create a VM is

CreateanAMI→CreateKeyPair→ConfigureFirewall→Launch (5.3)

This sequence is supported by public, private, and paid AMIs shown in Figure 6.24.The AMIs are formed from the virtualized compute, storage, and server resources shown at the bottom of Figure 6.23.

Table 6.12 Three Types of AMI

Image Type	AMI Definition
Private AMI	Images created by you, which are private by default. You can grant access to other users to launch your private images.
Public AMI	Images created by users and released to the AWS community, so anyone can launch instances based on them and use them any way they like. AWS lists all public images at http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=171.
Paid QAMI	You can create images providing specific functions that can be launched by anyone willing to pay you per each hour of usage on top of Amazon's charges.

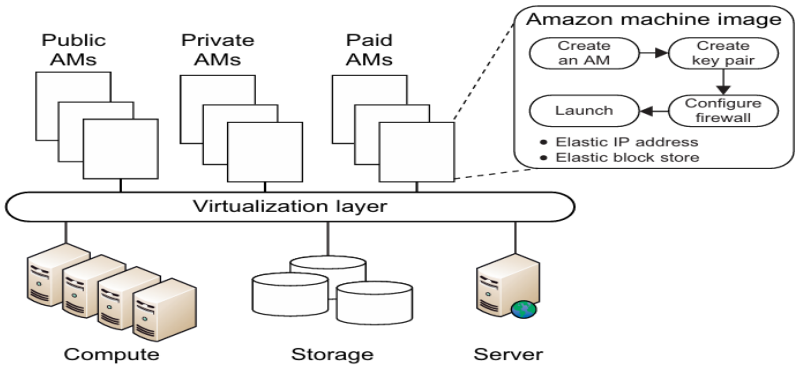


FIGURE 6.23
Amazon EC2 execution environment.

5.4.2 Amazon Simple Storage Service (S3)

Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. S3 provides the object-oriented storage service for users. Users can access their objects through Simple Object Access Protocol (SOAP) with either browsers or other client programs which support SOAP. SQS is responsible for ensuring a reliable message service between two processes, even if the receiver processes are not running. Figure 6.24 shows the S3 execution environment.

The fundamental operation unit of S3 is called an object. Each object is stored in a bucket and retrieved via a unique, developer-assigned key. In other words, the bucket is the container of the object. Besides unique key attributes, the object has other attributes such as values, metadata, and access control information. From the programmer's perspective, the storage provided by S3 can be viewed as a very coarse-grained key-value pair. Through the key-value programming interface, users can write, read, and delete objects containing from 1 byte to 5 gigabytes of data each. There are two types of web service interface for the user to access the data stored in Amazon clouds. One is a REST (web 2.0) interface, and the other is a SOAP interface. Here are some key features of S3:

• Redundant through geographic dispersion.

• Designed to provide 99.999999999 percent durability and 99.99 percent availability of objects over a given year with cheaper reduced redundancy storage (RRS).