

Module 5

Cloud Programming and Software Environments

6.1 Features of Cloud and Grid Platform

This section provides a summary of key features found in real-world cloud and grid platforms. We present this information across four tables, each focusing on a different aspect: capabilities, traditional features, data-related features, and those relevant to programmers and runtime systems. These tables serve as a referenced guide for developers aiming to efficiently program and utilize cloud infrastructure.

6.1.1 Cloud Capabilities and Platform Features

Commercial cloud platforms are designed to offer broad capabilities, as summarized in Table 6.1. These capabilities enable cost-effective utility computing with the elasticity to scale resources up or down based on demand. Beyond this core characteristic, commercial clouds increasingly provide additional services under the umbrella of Platform as a Service (PaaS).

For example, Microsoft Azure includes platform services such as Azure Table, queues, blobs, SQL Database, and both Web and Worker roles. While Amazon Web Services (AWS) is traditionally associated with Infrastructure as a Service (IaaS), it has steadily expanded its platform offerings to include SimpleDB (analogous to Azure Table), message queues, notification services, monitoring tools, content delivery networks, relational databases, and MapReduce support through Hadoop. Google, although not offering a comprehensive cloud service like Azure or AWS, provides the Google App Engine (GAE), a robust environment for developing and hosting web applications.

Table 6.2 outlines various low-level infrastructure features. Table 6.3 presents traditional programming models and environments used for parallel and distributed systems—capabilities that are increasingly expected in cloud platforms, either as part of the system or user environment. Table 6.4 highlights newer features emphasized by commercial cloud providers and, to a lesser extent, some grid systems. Many of these features have only recently seen wide adoption and are not yet available in most academic cloud infrastructures such as Eucalyptus, Nimbus, OpenNebula, or Sector/Sphere (though Sector, a data-parallel file system or DPFS, is categorized in Table 6.4).

Module 5- Cloud Programming and Software Environments

Table 6.1 Important Cloud Platform Capabilities	
Capability	Description
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 datasets, 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 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 programming	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 and without considering tedious infrastructure management issues such as handling model network failure or scaling the running code to use all the computing facilities provided by the platforms.
Workflow and data language support	The programming model offers abstraction of the cloud infrastructure. Similar to query 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.

6.1.2 Traditional Features Common to Grids and Clouds

Common Features in Grids and Clouds

6.1.2.1 Workflow

- Enables linking of cloud and non-cloud services for real-world applications.
- Popular tools: **Pegasus**, **Kepler**, **Taverna**, **Trident (Microsoft)**.
- Workflows can run on both Windows and Linux environments.

6.1.2.2 Data Transport

- **Data transfer** in and out of commercial clouds can be slow and costly.
- Uses simple protocols like **HTTP**.
- High-speed links may be introduced for better performance in national infrastructure.

- Cloud data (e.g., **Azure blobs**) supports parallel processing.

6.1.2.3 Security, Privacy, and Availability

- Use of **HTTPS/SSL** for secure communication.
- **Virtual clustering** for dynamic resource provisioning.
- Persistent storage with fast query capabilities.
- Fine-grained access control to ensure data protection.
- **Disaster recovery** through live VM migration.
- **Reputation systems** to block unauthorized users or attackers.

6.1.3 Data Features and Databases

6.1.3.1 Program Library

- VM image libraries help deploy and configure systems in IaaS environments.

6.1.3.2 Blobs and Drives

- Azure uses **blobs**; Amazon uses **S3**.
- These can be attached like shared drives (e.g., Azure Drive, Amazon EBS).
- Fault-tolerant storage similar to HPC file systems like Lustre.

6.1.3.3 DPFS – Distributed Parallel File Systems

Examples: Google File System, HDFS, Cosmos.

- Designed for compute-data affinity.
- Used in data-intensive applications (like MapReduce).

6.1.3.4 SQL and Relational Databases

- Both Amazon and Azure support relational databases.
- Cloud model offers “**SQL as a Service**”, simplifying deployment.
- Useful for smaller scales; **MapReduce** better for huge data.

6.1.3.5 Tables and NoSQL Databases

- Examples: **BigTable (Google)**, **SimpleDB (Amazon)**, **Azure Table**.
- Schema-free, lightweight storage for scalable applications.
- Popular for scientific and metadata applications.
- Open-source options: **HBase**, **CouchDB**, **M/DB**.

6.1.3.6 Queuing Services

- Amazon and Azure provide **message queues** for communication between application components.
- REST interfaces, “**deliver-at-least-once**” semantics.
- Alternatives: **ActiveMQ**, **NaradaBrokering**.

6.1.4 Programming and Runtime Support

6.1.4.1 Worker and Web Roles

- Azure uses **Worker roles** for background tasks and **Web roles** for web portals.
- No need for explicit task scheduling.
- Uses queues for distributed task management.

6.1.4.2 MapReduce

- Processes large data sets in parallel using key-value pairs.
- Open-source: **Hadoop**; Microsoft: **Dryad**.
- Cloud-friendly and fault-tolerant.
- Supports iterative computing (e.g., **Twister**).

6.1.4.3 Cloud Programming Models

- Examples: **Aneka**, **GAE (Google App Engine)**.
- Platform-independent and suitable for cloud and HPC environments.

6.1.4.4 Software as a Service (SaaS)

- Cloud programs are delivered as services.
- Reduces the need for user-side system management.
- SaaS supports scalability, security, and convenience.

Table 6.2 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

Module 5- Cloud Programming and Software Environments

- Registry: Information resource for system (system version of meta data 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 6.3 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 meta data 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, Grid Solve) 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 port lets to HUB zero 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 6.4 Platform Features Supported by Clouds and Sometimes Grids

1. Storage and Data Models

- **Blob:**
 - Basic cloud storage (e.g., **Azure Blob**, **Amazon S3**).
 - Used for storing large unstructured data like images, backups, and videos.
- **DPFS (Distributed Parallel File Systems):**
 - Examples: **Google File System**, **HDFS (Hadoop)**, **Cosmos (Dryad)**.
 - Designed with **compute-data affinity** for efficient large-scale data processing.
- **SQL:**
 - Traditional **relational database support** (e.g., MySQL, Oracle).
 - Offered by both Amazon and Azure.
- **Table (NoSQL):**
 - Schema-free data structures like **Amazon SimpleDB**, **Azure Table**, **Apache HBase**.
 - Part of the **NoSQL movement** focused on scalability and flexibility.

2. Programming and Execution

- **MapReduce:**
 - Programming model for distributed data processing.
 - Examples: **Hadoop** (Linux), **Dryad** (Windows), **Twister**.
 - Related languages: **Sawzall**, **Pregel**, **Pig Latin**, **LINQ**.
- **Programming Model:**
 - Cloud programming built on familiar web/grid paradigms.
 - Integrates other platform features like data and task management.
- **Worker Role:**
 - Concept from Azure for background task execution.
 - Implicitly used in both Amazon and grid systems.
- **Web Role:**
 - Used in Azure to handle web interfaces or portals.
 - Comparable to **Google App Engine (GAE)** functionality.

3. Infrastructure and System Management

- **Fault Tolerance:**
 - Key feature in clouds but largely **neglected in traditional grids**.
 - Enables system resilience against node failures.
- **Monitoring:**
 - Tools like **Inca** in grid environments.

- Often implemented using **publish-subscribe mechanisms**.
- **Notification:**
 - Event or status updates delivered via **publish-subscribe systems**.
- **Queues:**
 - Used for communication and task coordination between services.
 - Often based on **publish-subscribe models**.
- **Scalable Synchronization:**
 - Tools like **Apache Zookeeper** or **Google Chubby**.
 - Supports **distributed locks** and coordination.
 - Used in **BigTable** but unclear for Azure Table or SimpleDB.

6.2 Parallel and Distributed Programming Paradigms

What is Parallel and Distributed Programming?

- Parallel and distributed programming means running a program simultaneously on multiple computing systems.
- It involves two key concepts:
 - **Parallel computing:** Running parts of a program at the same time on multiple processors (may or may not be on a network).
 - **Distributed computing:** Running a program across multiple computers connected by a network (e.g., clusters).

Benefits

- **For users:** Faster response time.
- **For systems:** Better performance and resource usage.

Challenges

- Managing these systems is **complex** due to task coordination, data sharing, and communication.

6.2.1 Key Concepts in Parallel Programming

1. Partitioning

- **Computation Partitioning:** Break the program into smaller tasks that can run at the same time.

Module 5- Cloud Programming and Software Environments

- **Data Partitioning:** Split input data into chunks that different parts of the program can process in parallel.

2. Mapping

- Assign the smaller tasks or data pieces to specific computing resources (like assigning jobs to workers).

3. Synchronization

- Coordinates tasks to avoid:
 - **Race conditions:** Two tasks trying to use the same resource.
 - **Data dependency:** One task needing results from another.

4. Communication

- When tasks need to share data, they communicate over the network (especially in distributed systems).

5. Scheduling

- Decides **which task runs when**, especially when there are more tasks than available workers.

6.2.1.1 Why Do We Use Programming Paradigms?

- Managing all of the above manually is **hard** and slows down development.
- **Programming paradigms/models** help by **hiding complex details** from the programmer.

Benefits of Using Paradigms

1. Easier to write programs.
2. Faster development (less time to market).
3. Better use of system resources.
4. Higher system performance.
5. Support for more abstraction and scalability.

Popular Paradigms/Models

- **MapReduce, Hadoop, Dryad:**
 - Originally for data processing, now used in many applications.

Module 5- Cloud Programming and Software Environments

- More fault-tolerant and scalable than older models like **MPI (Message Passing Interface)**.

The section **6.2.2** from your document elaborates on **MapReduce**, an important framework for processing large-scale data in parallel across distributed systems, and introduces its extension **Twister** for iterative computations. Here's a summarized and simplified explanation:

6.2.2 MapReduce, Twister, and Iterative MapReduce

What is MapReduce?

MapReduce is a **programming model and processing framework** designed to handle large datasets across **distributed clusters** of computers. It simplifies parallel processing by providing two main user-defined functions:

- **Map:** Processes input data and produces intermediate key-value pairs.
- **Reduce:** Aggregates the intermediate values for each key to produce final output.

Key Features:

- Users write Map() and Reduce() functions.
- A MapReduce(Spec, &Results) function runs everything.
- The Spec object tells the framework where the input/output files are and which functions to use.

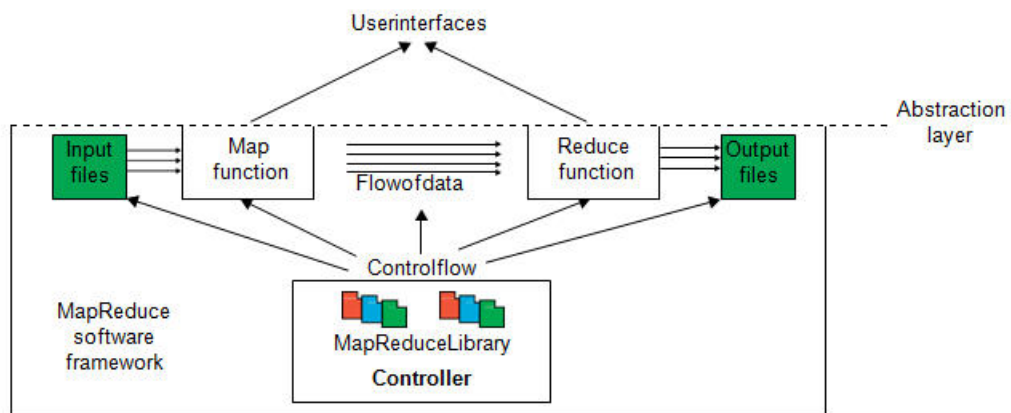


FIGURE 6.1

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.

Module 5- Cloud Programming and Software Environments

This diagram (Figure 6.1) illustrates the **MapReduce framework**, which processes large-scale data using a distributed computing model. Let's break down each component:

1. Input Files

- These are the raw data chunks (e.g., text files, logs) stored in a distributed file system (like HDFS).
- The framework splits the input data and distributes it to multiple **Map** tasks.

2. Map Function

- Each **Map** task reads a portion of the input data and processes it into intermediate key-value pairs.
- Example: For a word count task, it might emit pairs like (word, 1) for each word found.

3. Flow of Data (Map to Reduce)

- The intermediate key-value pairs are **shuffled and sorted**.
- All values associated with the same key are grouped together before being sent to the appropriate **Reduce** task.

4. Reduce Function

- Takes each key and a list of its values (e.g., (word, [1,1,1])) and performs aggregation or summarization (e.g., total count → (word, 3)).
- Produces the final output results.

5. Output Files

- The final reduced data is written to output files, usually stored again in a distributed file system.

6. Controller / MapReduceLibrary

- This is the **heart of the framework**.
- It manages:
 - **Task scheduling** (decides which node runs which task),
 - **Data distribution** (which data chunk goes to which map function),
 - **Communication** between Map and Reduce stages,
 - **Fault tolerance** (retries failed tasks),
 - And **coordination** of overall flow.

7. User Interfaces

- These provide access to the framework.
- Users write code for **Map()** and **Reduce()** functions.
- The abstraction layer hides the underlying complexity (e.g., data distribution, fault recovery).

Key Concepts Highlighted:

- **Abstraction Layer:** Users don't manage low-level system details.
- **Control Flow:** Managed by the controller to handle task coordination.
- **Data Flow:** From input → Map → Shuffle/Sort → Reduce → Output.

In Simple Terms:

The diagram shows how MapReduce **automates distributed data processing**:

- Users just define **what to do with data (Map/Reduce)**,
- The system handles **how and where** to do it.

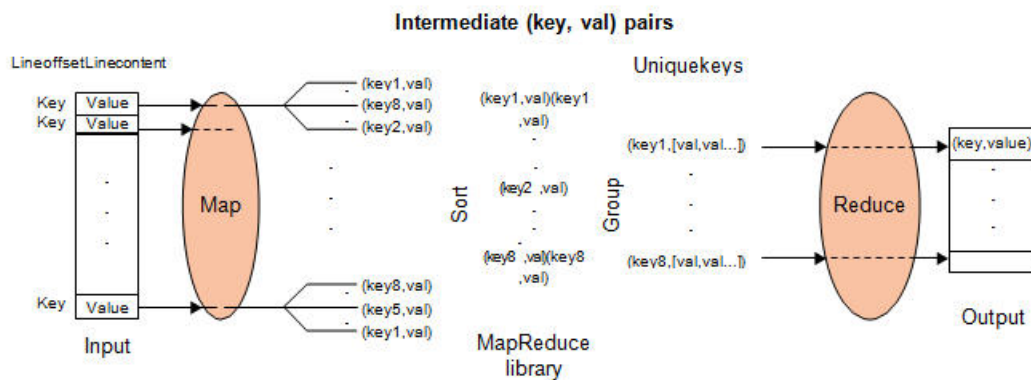


FIGURE 6.2

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

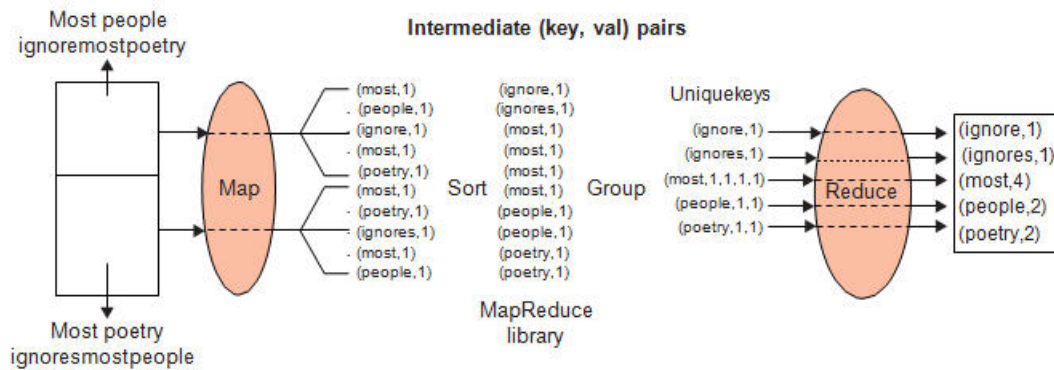


FIGURE 6.3

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

This diagram (**Figure 6.2**) shows the **logical data flow** in the **MapReduce model**, broken down into **5 processing stages** involving successive (key, value) pairs. It provides a deeper look into how data is transformed through the MapReduce pipeline.

Step-by-Step Explanation

1. Input

- Input data is read as lines of text, each associated with a unique **key** (often byte offset) and **value** (content of the line).
- Format: (key, value) — e.g., (offset, "This is a line of text").

2. Map Stage

- The **Map function** is applied to each (key, value) pair.
- It processes and emits **intermediate key-value pairs**:
 - E.g., for word count: ("This", 1), ("is", 1), etc.
- Output format: (key1, val1), (key2, val2), ..., potentially many from one input.

3. Sort Stage (Shuffle & Sort)

- The MapReduce **library** sorts all intermediate pairs by key.
- So all occurrences of the same key are grouped together.
- E.g., all ("word", 1) entries are gathered before the Reduce step.

4. Group Stage

- For each unique key, values are grouped into a list:

Module 5- Cloud Programming and Software Environments

- E.g., (key1, [val1, val2, val3, ...])
- This prepares data for aggregation.

5. Reduce Stage

- The **Reduce function** takes each grouped key and list of values.
- It applies logic (like summing counts or averaging) to output a single (key, value) pair per group.
 - Example: ("word", 5) if the word appeared 5 times.

Final Output

- The final output is a simplified and aggregated set of key-value pairs.

Summary of the 5 Stages

Stage	Description
Input	Raw data split into (key, value) pairs.
Map	Processes input and emits intermediate pairs.
Sort	Rearranges intermediate pairs by key.
Group	Aggregates all values for the same key.
Reduce	Applies a function to produce final output.

Example: Word Count

If input = "cat dog cat":

- **Map output:** ("cat", 1), ("dog", 1), ("cat", 1)
- **Sort & Group:** ("cat", [1,1]), ("dog", [1])
- **Reduce output:** ("cat", 2), ("dog", 1)

Data Flow in MapReduce:

1. **Input** to Map is a (key, value) pair (e.g., line number and text).
2. Map emits intermediate (key, value) pairs (e.g., (word, 1)).
3. Intermediate pairs are **sorted and grouped** by key.

Module 5- Cloud Programming and Software Environments

4. Each group like (word, [1,1,1]) is passed to Reduce, which outputs final results like (word, 3).

Example: Word Count

For the input lines:

- “most people ignore most poetry”
- “most poetry ignores most people”

The Map function emits intermediate results like:

- (most, 1), (people, 1), ...

These are grouped and reduced to:

- (most, 4), (people, 2), ...

Formal Notation:

- **Map phase:** (key1, val1) $\rightarrow$ List(key2, val2)
- **Reduce phase:** (key2, List(val2)) $\rightarrow$ List(val2)

Solving Strategy

The key to solving MapReduce problems is defining the correct:

- **Key:** Unique identifier (e.g., word, word length, sorted letters for anagram)
- **Value:** What you want to count or process (e.g., 1 for occurrences)

Behind the Scenes (Actual Data Flow):

1. **Data Partitioning:** Splits input into chunks.
2. **Computation Partitioning:** Distributes Map and Reduce tasks to workers.
3. **Master-Worker Model:** One master assigns tasks to multiple workers.
4. **Map Worker:**
 - Reads its split of input
 - Executes Map function
 - Optionally runs a **Combiner** to reduce local data
5. **Partitioning Function:** Ensures all data with the same key goes to the same Reduce task (e.g., using $\text{hash}(\text{key}) \% R$)
6. **Synchronization:** Reduce starts only after all Map tasks finish.

7. **Communication:** Reduce workers fetch intermediate data from all Map workers.
8. **Sorting & Grouping:** Data is grouped by key on each reduce node.
9. **Reduce Function:** Final aggregation and writing of output.

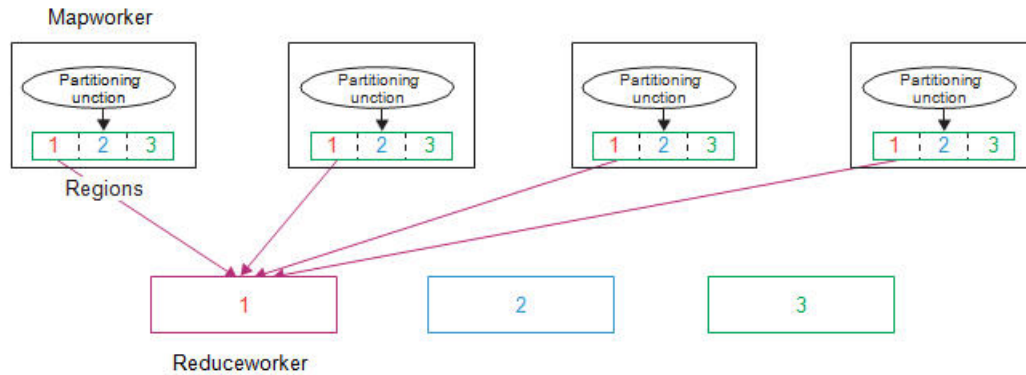


FIGURE 6.4

Use of MapReduce partitioning function to link the Map and Reduce workers.

This diagram (**Figure 6.4**) illustrates the **partitioning function** in the **MapReduce** model — specifically how **MapWorkers** assign output to the appropriate **ReduceWorkers** based on key values.

Explanation of Figure 6.4

Goal of Partitioning:

To distribute the intermediate key-value pairs generated by MapWorkers evenly and correctly to the appropriate ReduceWorkers.

Components Breakdown

MapWorkers

- Each **MapWorker** processes a chunk of the input data and emits intermediate (**key, value**) pairs.
- These key-value pairs must be passed to the right **ReduceWorker**.

Partitioning Function

- Determines **which ReduceWorker** should receive a particular key-value pair.
- Based on:

Module 5- Cloud Programming and Software Environments

```
python
CopyEdit
partition = hash(key) % num_reducers
```

- Example:
 - Keys with $\text{hash}(\text{key}) \% 3 == 0$ go to Reducer 1
 - Keys with $\text{hash}(\text{key}) \% 3 == 1$ go to Reducer 2
 - Keys with $\text{hash}(\text{key}) \% 3 == 2$ go to Reducer 3
- All MapWorkers use the **same partitioning logic** to ensure consistency.

Regions

- The diagram uses colored boxes (1, 2, 3) to represent **data partitions** (or **regions**).
- Each region is **assigned to a specific ReduceWorker**.

ReduceWorkers

- ReduceWorkers are **responsible for specific regions (partitions)**.
- All the intermediate pairs for a given region (say region 1) — **from all MapWorkers** — are sent to the **same ReduceWorker**.

Summary

Component	Role
MapWorker	Emits intermediate (key, value) pairs
Partitioning Fn	Decides which reducer a key should go to
Regions	Logical grouping of keys per partition
ReduceWorker	Processes keys from its assigned partition only

Analogy:

Imagine 4 teachers (MapWorkers) grading exam papers. Based on a **student's ID (key)**, they use a rule (Partitioning function) to decide which **department head (ReduceWorker)** the scores should be sent to. This ensures each head compiles grades only for their assigned group of students.

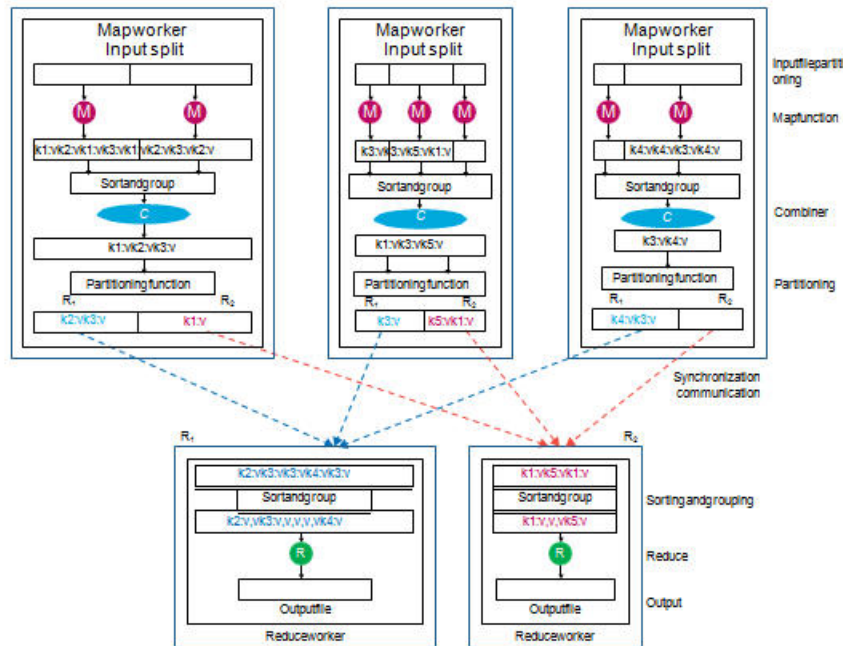


FIGURE 6.5

Dataflow implementation of many functions in the Mapworkers and in the Reduceworkers through multiple sequences of partitioning, combining, synchronization and communication, sorting and grouping, and reduce operations.

This diagram (Figure 6.5) illustrates the **dataflow implementation** in a **MapReduce job**, showing detailed internal operations in both **Map workers** and **Reduce workers**. It breaks down the workflow into stages like **partitioning**, **combining**, **synchronization**, **communication**, **sorting**, and **reducing**.

Explanation of Each Component

1. Input Splitting

- Input data is divided into chunks, and each chunk is processed by a **MapWorker**.
- Each worker receives an **Input split**.

2. Map Function (M)

- Each input record (like a line of text) is passed to the **Map function**.
- Output: A set of **intermediate key-value pairs** — e.g., (K1, V1).

3. Combiner Function (C) (Optional but common)

- A local aggregation step that reduces the volume of data transferred.
- Example: Locally counts repeated keys within the same MapWorker before sending to Reduce.

Module 5- Cloud Programming and Software Environments

- Output: Fewer intermediate pairs, e.g., (K1, V') instead of multiple (K1, V).

4. Partitioning Function

- Determines **which reducer** a key-value pair should be sent to.
- Each key is mapped to a **partition (R₀, R₁, R₂, etc.)**, often using a hash function.

5. Shuffle and Sort (Synchronization & Communication)

- Intermediate key-value pairs are sent across the network to **Reduce workers**.
- During this **Shuffle** step:
 - Data from all MapWorkers is **synchronized**.
 - Each ReduceWorker receives all values for a given key from **all Mappers**.

6. Reduce Worker

Each **ReduceWorker**:

- **Sorts and groups** the received data by key:
 - E.g., receives: (K2, [V3, V5, V4])
- Then applies the **Reduce function**, such as summing values or finding averages.
- Outputs: Final (**key, value**) pairs written to the **Output file**.

Summary Table

Stage	Role
Input Splitting	Divides input data among multiple MapWorkers
Map Function (M)	Processes data, emits intermediate key-value pairs
Combiner (C)	Optional local reduce before shuffle
Partitioning	Decides which reducer gets which key
Shuffle	Redistributes data from mappers to reducers
Reduce Worker	Aggregates and finalizes output per key

Real-World Analogy:

Imagine multiple teachers (MapWorkers) each grading student assignments (Input splits). After marking:

Module 5- Cloud Programming and Software Environments

- They **summarize scores** (Combiner).
- Each student's scores go to a specific teacher for **final tabulation** (Partitioning and Shuffle).
- The receiving teacher **calculates the final grade** (Reduce), and stores it (Output).

Compute-Data Affinity

MapReduce tries to send computation **to the data** (on the same node), not the other way around—this improves efficiency and reduces network use. Google's GFS stores files in blocks, aligning well with this strategy.

Twister and Iterative MapReduce

Standard MapReduce is **not efficient for iterative tasks** (like machine learning or graph processing) because it writes intermediate results to disk.

- **Twister** is an extension that supports **in-memory** iterative MapReduce:
 - Keeps data in memory between iterations
 - Reduces overhead from repeated data loading
 - Improves performance for applications requiring repeated computation over the same data

MapReduce vs MPI

- **MapReduce** uses file-based communication (slower).
- **MPI** (Message Passing Interface) uses direct node-to-node communication (faster).
- MPI is better for fine-grained, tightly-coupled computations.
- MapReduce is better for fault-tolerant, large-scale data analysis.

Aspect	MapReduce	MPI
Communication	Via files (read/write to disk)	Direct node-to-node over the network
Data Transfer	Transfers entire data (full data flow)	Transfers only what's needed (δ -flow)
Performance	Higher communication/synchronization overhead	More efficient due to leaner, direct communication

- **δ (delta) flow**: Represents the **minimal update information** exchanged in each iteration.
- **Full data flow**: Transferring all data, even unchanged parts, each time — as done in MapReduce.

Why Classic MapReduce is Slow for Iterative Algorithms

MapReduce's architecture:

- Creates new processes for each job step
- Writes and reads data from disk between map and reduce steps
- Lacks efficient support for **iterative** workloads like **K-means clustering** or **PageRank**

Performance Improvements Suggested

To overcome MapReduce's limitations for iterative tasks:

1. **Stream data** between steps (no intermediate disk I/O)
2. Use **long-running threads/processes** for efficient δ communication (rather than restarting jobs)

These changes:

- Reduce disk I/O
- Improve performance (especially for algorithms needing many iterations)
- Trade off: reduced fault tolerance and flexibility

Twister (MapReduce++)

Twister is a runtime framework built to **optimize iterative MapReduce computations**:

Features:

- Keeps **static data** (like input vectors) in memory
- Transfers only **dynamic data** (δ updates) each iteration
- Runs **Map and Reduce tasks in long-lived threads**
- Faster than traditional Hadoop MapReduce (e.g., in K-means)

Referenced Figures:

- **Figure 6.7:** Shows Twister's architecture and runtime
- **Figure 6.8:** K-means performance comparison (Twister vs. Hadoop)
- **Figure 6.9:** Thread/process structure comparison: Hadoop, Dryad, Twister, MPI

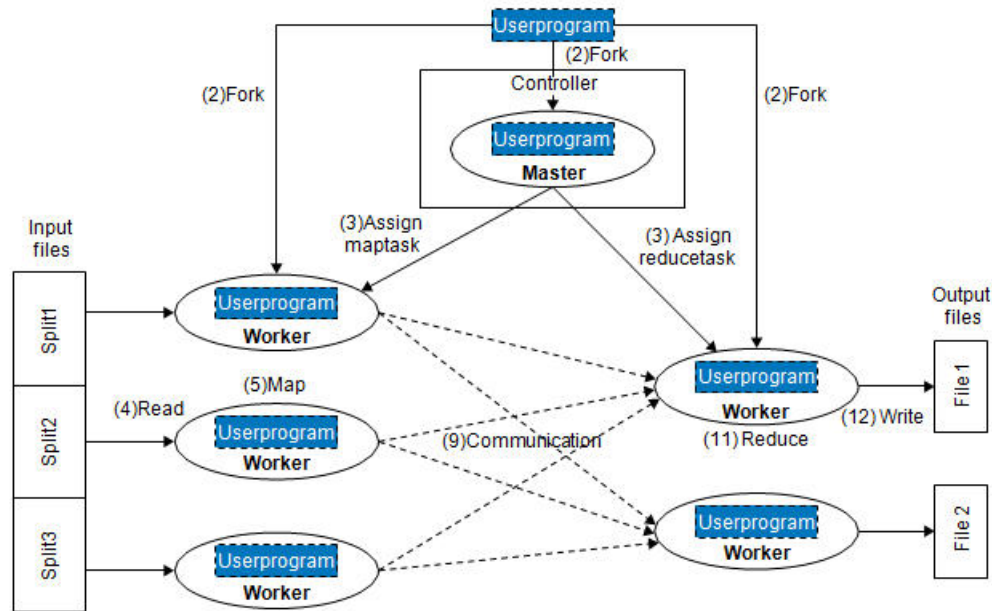


FIGURE 6.6

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.

Explanation of Figure 6.6: Control Flow in MapReduce

This diagram shows the **control flow of a MapReduce job**—from starting the user program to executing map and reduce tasks on distributed worker nodes and finally writing output files.

Components Involved

- **User Program:** Initiates the MapReduce job.
- **Master:** Controls and coordinates the job.
- **Workers:** Execute tasks (map or reduce).
- **Input Files:** Data split into parts for processing.
- **Output Files:** Final reduced data is stored here.

Step-by-Step Breakdown

(1) Start

- The **user program** launches and initiates the MapReduce job.

(2) Fork

- The user program **forks (spawns)**:
 - A **master** process

Module 5- Cloud Programming and Software Environments

- Multiple **worker** processes (Map and Reduce workers)

(3) Assign Tasks

- The **master** assigns:
 - **Map tasks** to workers based on **input splits** (Split1, Split2, Split3)
 - **Reduce tasks** to separate workers

(4) Read

- Each Map worker **reads its assigned input split** from the input files.

(5) Map

- Each Map worker executes the **Map function** on its split, producing intermediate (**key, value**) pairs.

(9) Communication (Shuffle Phase)

- Intermediate results from all Map workers are **shuffled and transferred** (sorted and grouped by key) to the appropriate Reduce workers.

(11) Reduce

- Reduce workers perform the **Reduce function** on received grouped data, processing keys and their associated values.

(12) Write

- Final **reduced output** is written to **output files** (e.g., File 1, File 2).

Summary

Stage	Function
Userprogram	Starts the job and forks workers
Master	Coordinates and assigns map/reduce tasks
Workers	Read, process (map/reduce), and write data
Communication	Shuffle phase moves intermediate data to reducers

Module 5- Cloud Programming and Software Environments

This figure illustrates how the **MapReduce framework orchestrates parallel data processing**, automatically handling **task assignment, data movement, and job coordination** to process large datasets efficiently.

Explanation of Figure 6.7: Twister – Iterative MapReduce Programming Paradigm

This figure explains the **Twister framework**, an enhanced version of MapReduce (often called MapReduce++) designed for **iterative applications**, such as **K-means clustering, PageRank, and machine learning algorithms**.

Part (a): Twister for Iterative MapReduce Programming

This diagram illustrates the **programming model** of Twister:

1. Configure()

- Loads **static data** (e.g., constants, static matrices) **once** before the iterative process starts.

2. Map(key, value)

- Each iteration begins by applying the **Map function** to dynamic data (e.g., input samples).
- Generates intermediate **(key, value)** pairs.

3. Reduce(key, list<value>)

- The Reduce function receives all values associated with a key, typically to **aggregate or compute a result** for that key.

4. Combine(key, list<value>)

- **Optional:** Locally combines intermediate outputs to reduce data transferred to reducers.

5. dfLow (δ -flow)

- Twister introduces **δ -flow**, a small piece of data (delta) communicated between iterations, **instead of full data** like in traditional MapReduce. This improves performance.

6. User Program Iteration

- After each Reduce (or Combine), the **User Program** evaluates results and decides whether to **iterate again** (loop back to Map).

7. Close()

- Once convergence or a stopping condition is met, the program finalizes the process and **cleans up**.

Part (b): Twister Runtime Architecture

This part shows how Twister is implemented at runtime with multiple components:

Components:

- **MR Daemon (D):** Long-running processes managing Map and Reduce workers. Unlike traditional MapReduce, **Twister reuses workers** between iterations.
- **M / R:** Map and Reduce workers.
- **MR Driver:** Central coordinator responsible for managing iterations, task assignments, and communications.
- **User Program:** Drives the iterations, convergence checks, and termination.
- **Pub/Sub Broker Network:** Used for **efficient communication** (publish/subscribe model) between distributed workers.

Data and Control Flow:

- **Data Splits:** Input data is split and fed into the system.

Module 5- Cloud Programming and Software Environments

- **File System:** Provides access to static and initial input data.
- **Data Read/Write:** Happens through MR Daemon and workers.
- **Communication:** δ -flows and task coordination occur over the **Pub/Sub network**.

Difference from Traditional MapReduce

Feature	Traditional MapReduce	Twister (MapReduce++)
Iteration Support	Requires full re-execution	Supports native iteration
Data Flow	Full data flow	δ-flow (partial update)
Disk I/O	Writes between steps	Keeps data in memory
Worker Lifecycle	Short-lived	Long-running
Communication Model	Push-based	Pub/Sub asynchronous

Summary

Twister is an **efficient iterative MapReduce framework** built for tasks that require multiple passes over data. It:

- Minimizes disk I/O,
- Reduces communication overhead (via δ -flow),
- Reuses threads across iterations (persistent workers),
- Uses pub/sub communication for flexibility and speed.

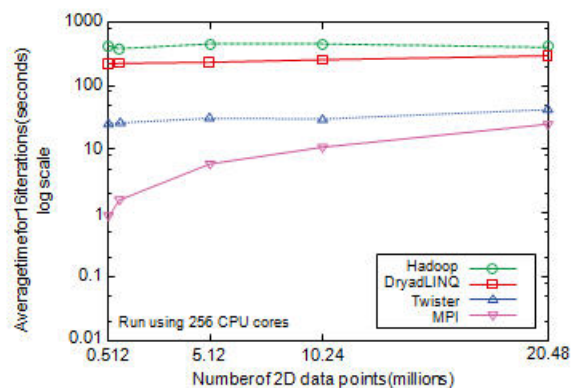


FIGURE 6.8

Performance of Kmeans clustering for MPI, Twister, Hadoop, and DryadLINQ.

Explanation of Figure 6.8: Performance of K-means Clustering for MPI, Twister, Hadoop, and DryadLINQ

Experiment Overview

- **Task:** K-means clustering
- **Environment:** 256 CPU cores
- **Metric:** Average time (in seconds) to complete **10 iterations** of K-means
- **Data Size:** Ranges from **0.512 million to 20.48 million** 2D data points
- **Y-axis (Log Scale):** Average iteration time (seconds)
- **X-axis:** Number of 2D data points in millions

Hadoop (Green Line)

- **Performance:** Consistently the **slowest**.
- **Reason:**
 - Writes intermediate data to disk
 - Creates and destroys Map and Reduce tasks every iteration
 - High I/O and setup/teardown overhead

DryadLINQ (Blue Line with Squares)

- **Performance:** Slightly better than Hadoop
- **Reason:**
 - Supports pipelining (can avoid some disk I/O)
 - Still has some overhead from task orchestration
 - Not optimized for iterative algorithms

Twister (Red Line with Triangles)

- **Performance:** Significantly **faster** than both Hadoop and DryadLINQ
- **Reason:**
 - Uses **long-running Map and Reduce tasks**
 - Avoids disk I/O between iterations
 - Sends only **δ (delta) updates**, not full data
 - Designed for **iterative applications** like K-means

MPI (Magenta Line with Circles)

- **Performance:** **Best overall**
- **Reason:**
 - Direct inter-process communication (no file I/O)

Module 5- Cloud Programming and Software Environments

- Highly optimized and low-overhead parallelism
- Transfers only necessary updates (δ flow)
- Ideal for tightly-coupled, compute-intensive tasks

Observations

- **All systems** show some increase in execution time as the data size grows.
- However:
 - **MPI** and **Twister** scale **much better** than Hadoop and DryadLINQ.
 - **MPI** consistently delivers the **fastest results**.
 - **Twister** offers a balance between ease-of-use (MapReduce-like) and **high performance**.

Conclusion

- For **iterative machine learning tasks like K-means**, traditional MapReduce frameworks like Hadoop are **inefficient**.
- **Twister** improves upon this with support for iteration and in-memory computation.
- **MPI** remains the **most performant** but is harder to program and manage.

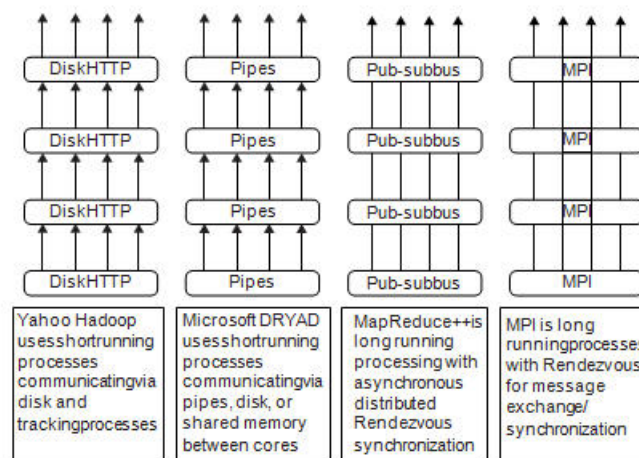


FIGURE 6.9

Thread and process structure of four parallel programming paradigms at runtimes.

Figure 6.9 presents a **comparison of thread and process structures** in four parallel programming paradigms: **Hadoop**, **Dryad**, **Twister (MapReduce++)**, and **MPI**. Here's a breakdown of what each model represents and how they differ:

1. Yahoo Hadoop

- **Structure:** Uses **short-running processes**.
- **Communication:** Via **disk** and **HTTP**, which introduces latency.
- **Synchronization:** Managed by tracking processes (JobTracker/TaskTracker).
- **Drawback:** **High overhead** due to repeated process startup and heavy disk I/O.

2. Microsoft Dryad

- **Structure:** Also uses **short-running processes**, like Hadoop.
- **Communication:** More flexible – uses **pipes**, **shared memory**, or **disk**.
- **Advantage:** Reduces disk usage slightly compared to Hadoop.
- **Limitation:** Still lacks iteration support and long-lived tasks.

3. Twister (MapReduce++)

- **Structure:** Uses **long-running processes**.
- **Communication:** Through **publish-subscribe (pub-sub) mechanisms**.
- **Synchronization:** Asynchronous, using **distributed rendezvous synchronization**.
- **Strength:** Designed for **iterative applications**, minimizing overhead between iterations.

4. MPI (Message Passing Interface)

- **Structure:** Also uses **long-running processes**.
- **Communication:** Direct **message exchange** using **rendezvous synchronization**.
- **Efficiency:** Very fast, ideal for **tightly-coupled parallel programs**.
- **Limitation:** **Low-level** and harder to use than higher-level models like Twister or Hadoop.

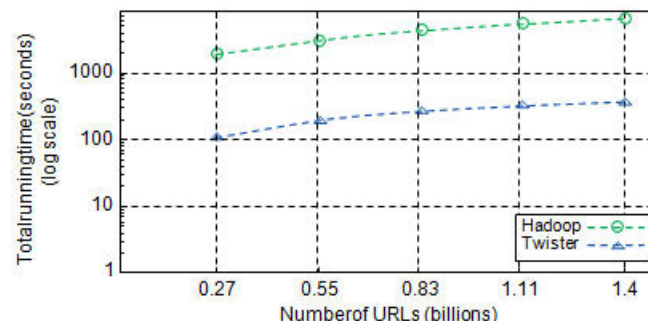


FIGURE 6.10

Performance of Hadoop and Twister on ClueWeb dataset using 256 processing cores.

Module 5- Cloud Programming and Software Environments

Key Observations:

Metric	Hadoop	Twister
Runtime	Significantly higher	Much lower
Trend	Increases rapidly with data	Increases more gradually
Efficiency	Slower for large datasets	Scales better with increasing size
Scalability	Poorer scalability	Better scalability

Interpretation:

- **Twister** performs much **faster** than **Hadoop** across all data sizes.
- Hadoop suffers from high overhead due to:
 - Writing intermediate data to disk.
 - Short-running task management.
- Twister:
 - Uses **long-running tasks** and **in-memory communication** (no disk I/O between iterations).
 - Excels in **iterative applications** and **large-scale data processing**.

Conclusion:

Twister is a **highly efficient MapReduce runtime** for iterative applications like those in machine learning or graph analytics. Its performance advantage over Hadoop grows **as dataset size increases**, making it a better fit for **big data processing at scale**.

Comparison of Programming Models

Model	Process Management	Disk Usage	Iteration Support
Hadoop	Starts new processes	High	Poor
Dryad	Pipes between operators	Low	Moderate
Twister	Long-running threads	Low	Excellent
MPI	Highly efficient messaging	None	Excellent

Summary:

- Traditional MapReduce is **not ideal for iterative applications**.
- **Twister** improves performance by **reducing I/O and using persistent threads**.
- **MPI** remains fastest but is harder to use in fault-tolerant, dynamic environments.
- Twister strikes a balance between **performance and usability** for iterative big data tasks.

6.2.3 Hadoop Library from Apache

Apache Hadoop is an open-source framework developed in Java that allows for distributed processing of large data sets across clusters of computers using a model called **MapReduce**. It was designed as an alternative to Google's proprietary systems and includes its own **file storage layer (HDFS)** and **computation engine (MapReduce engine)**.

Core Components of Hadoop

1. **MapReduce Engine:**
 - This is the processing engine that runs computations on the data stored in HDFS.
 - It breaks down tasks into **Map** and **Reduce** phases for parallel processing.
2. **HDFS (Hadoop Distributed File System):**
 - A specialized file system designed for high-throughput access to large files.
 - Inspired by Google's GFS, it is tailored for large-scale data storage and access.

HDFS Architecture

- HDFS uses a **master/slave** architecture:
 - **NameNode (Master):** Manages the file system namespace and metadata (e.g., file names, block locations).
 - **DataNodes (Slaves):** Actually store the file data in blocks and perform read/write operations.
- **How Storage Works:**
 - Files are split into **fixed-size blocks** (e.g., 64 MB).
 - These blocks are distributed and stored across different DataNodes.
 - The NameNode keeps a record of where each block is stored.

Key Features of HDFS

1. Fault Tolerance

- **Block Replication:**
 - Each block is copied to multiple DataNodes to prevent data loss.
 - Default replication factor is 3.
- **Replica Placement Strategy:**
 - 1 copy on the same node as the original.
 - 1 copy on a different node within the same rack.
 - 1 copy on a node in a different rack (for added reliability).
- **Heartbeat and Blockreport Messages:**
 - **Heartbeats:** Sent by DataNodes to let the NameNode know they are working.

- **Blockreports:** Contain lists of all the blocks stored on each DataNode, helping the NameNode manage data placement.

2. High Throughput

- HDFS is optimized for **batch processing**, not real-time use.
- Large blocks (like 64 MB) are used to:
 - Reduce the amount of metadata (fewer blocks to track).
 - Allow faster streaming of large data chunks during reads.

HDFS File Operations

Reading a File

1. The user sends a request to the NameNode to open the file.
2. The NameNode returns the list of DataNodes storing the file blocks.
3. The user then connects to the nearest DataNode to read each block one-by-one.

Writing a File

1. The user sends a “create” request to the NameNode.
2. Once approved, the user starts writing data to a queue.
3. A component called the **data streamer** writes data to one DataNode and passes it along to others (for replication).
4. This process is repeated for each block until the entire file is stored.

Summary

Apache Hadoop, through its components MapReduce and HDFS, enables efficient, reliable processing and storage of massive data sets across clusters. HDFS, with its fault tolerance and high throughput capabilities, ensures data is stored safely and accessed quickly, making it ideal for big data applications.

6.2.3.1 Architecture of MapReduce in Hadoop

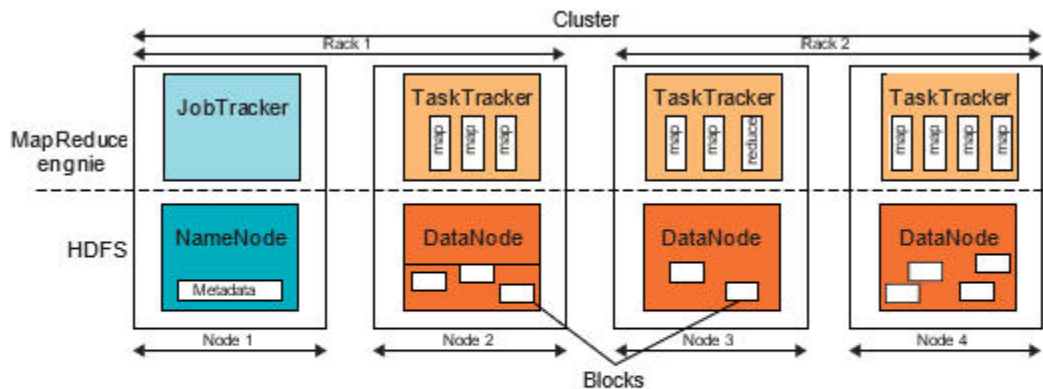


FIGURE 6.11

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

MapReduce is the **topmost layer** in the Hadoop framework. It coordinates and manages the **processing of large data sets** using parallel, distributed computing.

Architecture Overview

- **MapReduce Engine** has a **Master-Slave architecture**, similar to HDFS:
 - **JobTracker (Master):**
 - Controls the execution of MapReduce jobs.
 - Assigns map and reduce tasks to available worker nodes.
 - Monitors task progress and handles job coordination.
 - **TaskTrackers (Slaves):**
 - Run on individual nodes in the cluster.
 - Execute the actual map or reduce tasks assigned by the JobTracker.

Task Execution and Slots

- **Task Slots:**
 - Each TaskTracker has a fixed number of **execution slots** based on the node's CPU capabilities.
 - Example: A node with N CPUs, each supporting M threads, will have $N \times M$ slots.
- **Slot Usage:**
 - Each slot runs **one map or reduce task** at a time.
 - **One map task = One data block** → This means there's a **1:1 relationship** between map tasks and data blocks stored in HDFS.

Key Points

- **Map tasks** are closely tied to data blocks in HDFS.
- **Reduce tasks** process the output of the map phase after all map tasks finish.
- Efficient resource management is crucial as the performance depends on how well the JobTracker and TaskTrackers utilize CPU and memory through slots.

This section explains how Hadoop's MapReduce component works on top of HDFS to process big data efficiently through distributed execution and parallelism

The diagram in **Figure 6.11** illustrates the **architecture of Hadoop**, specifically showing how the **MapReduce engine** and **HDFS (Hadoop Distributed File System)** interact within a distributed cluster setup.

Overview

The figure is divided into two logical layers:

- **Top Half:** Represents the **MapReduce engine** (for processing data).
- **Bottom Half:** Represents **HDFS** (for storing data).

The cluster is organized into multiple **nodes**, grouped into **racks** (Rack 1 and Rack 2).

Key Takeaways

- **Node 1** is the master node with **JobTracker** (MapReduce master) and **NameNode** (HDFS master).
- **Nodes 2, 3, 4** are worker nodes with both **TaskTrackers** and **DataNodes**.
- **Blocks** are stored in DataNodes; **Tasks** are executed in TaskTrackers.
- The architecture supports **fault tolerance**, **parallel processing**, and **data locality**.

6.2.3.2Running a Job in Hadoop

When a MapReduce job is run in Hadoop, three main components are involved:

- **User Node:** Where the job is submitted from.
- **JobTracker:** Manages the job, tracks its progress, and assigns tasks.
- **TaskTrackers:** Run the actual map and reduce tasks on cluster nodes.

Step-by-Step Process:

1. Job Submission

Module 5- Cloud Programming and Software Environments

The user submits a job from their node to the JobTracker. Here's what happens:

- The user node:
 - Requests a new job ID from the JobTracker.
 - Splits the input files into chunks.
 - Uploads necessary resources (JAR file, config, input splits) to the JobTracker's file system.
 - Calls the submitJob() function to officially submit the job.

2. Task Assignment

- The JobTracker:
 - Creates one map task per input split.
 - Assigns map tasks to TaskTrackers by considering data locality (to reduce data movement).
 - Creates reduce tasks (number is set by the user).
 - Assigns reduce tasks to TaskTrackers without locality considerations.

3. Task Execution

- Each TaskTracker:
 - Copies the job's JAR file.
 - Launches a Java Virtual Machine (JVM).
 - Executes the assigned map or reduce task using instructions from the JAR.

4. Task Monitoring (Heartbeat Check)

- Each TaskTracker:
 - Sends periodic heartbeat messages to the JobTracker.
 - Heartbeats indicate that the TaskTracker is alive and whether it's ready for new tasks.

Analogy to Google's MapReduce

- The runJob(conf) in Hadoop is similar to Google's original MapReduce(Spec, &Results) function.
- The conf object in Hadoop plays a similar role to Spec in Google's implementation.

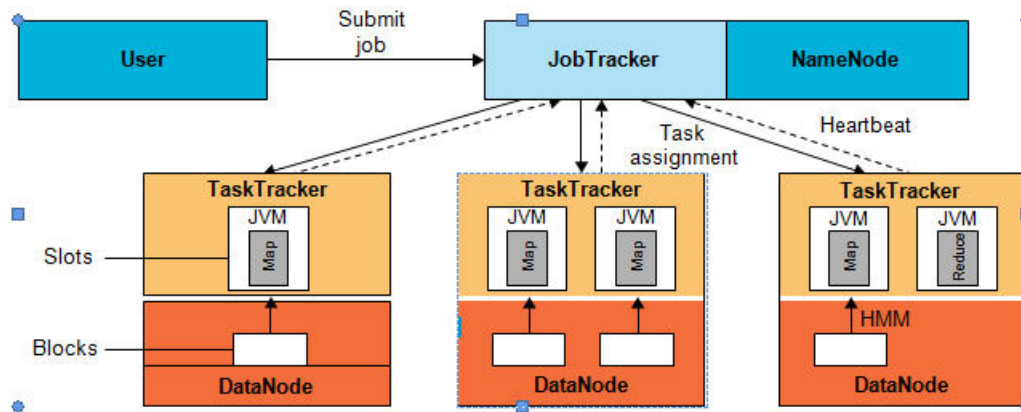


FIGURE 6.12

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

Components in the Diagram

1. **User Node (User)**
 - The job is submitted from here using `runJob(conf)`.
2. **JobTracker**
 - Central coordinator that receives the job, splits tasks, and assigns them to **TaskTrackers**.
3. **NameNode**
 - Master node of HDFS.
 - Knows where data blocks are stored across the cluster.
 - Used to locate data when JobTracker schedules map tasks (for **data locality**).
4. **TaskTrackers**
 - Nodes that execute **Map** or **Reduce** tasks.
 - Each TaskTracker:
 - Has one or more **slots** (represented by JVMs) to run tasks.
 - Connects to the **DataNode** on the same machine for accessing HDFS data blocks.
 - Periodically sends **heartbeat signals** to the JobTracker indicating it is alive and reporting task status.
5. **DataNodes**
 - Part of HDFS.
 - Store actual data blocks.
 - Accessed by the Map tasks during execution.

Process Flow

1. Job Submission

Module 5- Cloud Programming and Software Environments

- The **User** submits a MapReduce job to the **JobTracker**.
- **JobTracker**:
 - Gets metadata from **NameNode** to know data locations.
 - Splits the job into tasks (map/reduce).

2. Task Assignment

- Based on data locality (data block locations known via NameNode), the **JobTracker** **assigns tasks** to the appropriate **TaskTrackers**.
- **Map tasks** are scheduled closer to where the data blocks reside (shown in the DataNodes).
- **Reduce tasks** are assigned without considering locality.

3. Task Execution

- Each **TaskTracker** launches one or more **Java Virtual Machines (JVMs)** to execute:
 - **Map** tasks (left and middle nodes).
 - **Reduce** task (right node).
- JVMs use the code in the submitted JAR to process the task.

4. Heartbeat

- TaskTrackers send **heartbeat messages** to the **JobTracker** to:
 - Report status.
 - Confirm that they are operational.
 - Indicate availability of free task slots.

Important Notes:

- **Slots**: Represent available threads/JVMs on each TaskTracker.
- **Blocks**: Chunks of data stored in the HDFS, accessed by map tasks.
- **HMM** label in the third TaskTracker seems symbolic or placeholder—it may be an annotation or typo.

6.2.4 Dryad and DryadLINQ from Microsoft

6.2.4.1 What is Dryad?

Dryad is a **general-purpose distributed execution engine** for **parallel data processing**, developed by Microsoft. It's more **flexible than MapReduce** because:

- It allows the user to define their own **dataflow** using a **Directed Acyclic Graph (DAG)**.

- Each **vertex** in the DAG is a **computational task** (a program).
- Each **edge** is a **data channel** between tasks (for communication).

figure6.13 Dryad framework and its job structure, control and data flow

How Dryad Works

1. Job Definition

- A Dryad job is a **DAG**, where:
 - **Vertices** = processing programs (e.g., map, filter, aggregate).
 - **Edges** = communication paths (data transfer channels).
- This gives users control to define **custom workflows**, not limited to map and reduce.

2. Execution Components

- **Job Manager:**
 - Constructs the DAG from the user-defined program.
 - Schedules tasks on available nodes in the cluster.
 - Manages execution **but does not handle data movement** (avoids becoming a bottleneck).
- **Name Server:**
 - Maintains a list of **available computing resources** (cluster nodes).

Module 5- Cloud Programming and Software Environments

- Helps the job manager in **resource allocation** and understanding **network topology**.

3. Deployment

- Job manager maps the logical DAG to physical resources using info from the name server.
- A **daemon** runs on each node:
 - Acts as a **proxy**.
 - Receives the program binary.
 - Executes the assigned tasks.
 - Reports status back to the job manager.

Data Transfer and Communication

- Communication between tasks happens through **channels** (edges in DAG).
- Data channels use:
 - Shared memory
 - TCP sockets
 - Distributed file systems
- Unlike Hadoop, where all coordination happens via the JobTracker, **Dryad allows direct communication between nodes**, making it **more scalable**.

Fault Tolerance

Dryad handles two types of failures:

1. **Vertex (Task) Failure:**
 - Job manager reschedules the failed task on a new node.
2. **Channel (Edge) Failure:**
 - The vertex that initiated the channel is re-executed.
 - A new channel is created.

Advanced Features

- Dryad can **dynamically modify the DAG** at runtime:
 - Create new vertices.
 - Add/remove edges.
 - Merge graphs.
- Enables execution of complex jobs and workflows.
- Supports different programming models:
 - Scripting languages

Module 5- Cloud Programming and Software Environments

- MapReduce
- SQL-like processing (via DryadLINQ)

Analogy: 2D UNIX Pipes

- Dryad can be viewed as a **2D version of UNIX pipes**:
 - Traditional pipes: One-dimensional (linear flow).
 - Dryad: Two-dimensional (many vertices communicating in parallel).
- Each vertex may run multiple programs, enabling **parallelism at scale**.

Summary

Feature	Dryad
Programming model	DAG-based (flexible dataflow)
Core units	Vertices (programs), Edges (channels)
Coordination	Job Manager + Name Server
Fault tolerance	Re-execution of vertices or channels
Communication	Direct (channel-based), avoids bottlenecks
Use cases	Complex workflows, MapReduce, SQL (via DryadLINQ)

6.2.4.2 DryadLINQ from Microsoft

DryadLINQ is a **high-level programming interface** that combines:

- **Dryad** – Microsoft’s distributed execution engine for data-parallel computing.
- **LINQ** (Language Integrated Query) – A .NET-based query syntax for querying data in C#.

Why DryadLINQ?

DryadLINQ enables **regular .NET developers** to write scalable distributed programs using **familiar C# and LINQ syntax**, without needing to deal with low-level parallelism or distributed programming challenges.

It hides the complexity of:

- Task scheduling
- Data partitioning
- Fault tolerance
- Network communication

Execution Flow of a DryadLINQ Job

Error! Filename not specified.

The execution is divided into **9 steps**:

1. Application Starts and Builds Expression

- The user writes a .NET (e.g., C#) application.
- A **DryadLINQ expression object** is created using LINQ.
- No execution happens yet due to **deferred evaluation**.

2. Execution Triggered with ToDryadTable()

- The method ToDryadTable() is called.
- This triggers **parallel execution**.
- The LINQ expression object is handed over to DryadLINQ.

3. Compilation into Dryad Job Graph

- The LINQ expression is **compiled** into a **Dryad execution plan**.
- The expression is **split** into sub-expressions → one per **Dryad vertex**.
- DryadLINQ generates:
 - Vertex-specific code
 - Static data
 - Serialization code

4. Job Manager Starts

- A **custom Dryad job manager** is invoked.
- It **manages and monitors** the job's execution.

5. Graph Creation and Vertex Scheduling

- The job manager creates the **DAG (Directed Acyclic Graph)**.

Module 5- Cloud Programming and Software Environments

- Vertices (tasks) are **scheduled and executed** on available cluster resources.

6. Vertex Execution

- Each vertex runs its **own piece of logic** (as compiled earlier).
- These are independent and run in parallel where possible.

7. Output Generation

- **Upon completion, results are written to** output DryadTable(s).

8. Job Manager Ends, DryadLINQ Returns Control

- Job manager terminates.
- DryadLINQ returns **DryadTable objects** encapsulating results.
- These can be **input to next jobs** (i.e., multi-stage pipelines).

9. User Application Receives Results

- Control returns to the user's .NET program.
- The user can **iterate** over results using the .NET iterator interface.

Summary Table

Step	Description
1	LINQ expression created in user code
2	Execution triggered via ToDryadTable()
3	Expression compiled to Dryad DAG
4	Custom job manager launched
5	DAG mapped to physical resources
6	Vertices executed in cluster
7	Output written to DryadTable(s)
8	Job manager ends; DryadLINQ gets control

Step	Description
9	Results returned to user program

Benefits of DryadLINQ

- Seamless integration with **C# and .NET**.
- Allows **database-like query operations** on distributed data.
- Supports **automatic optimization, fault tolerance, and parallelism**.
- Simplifies **big data processing** for traditional developers.

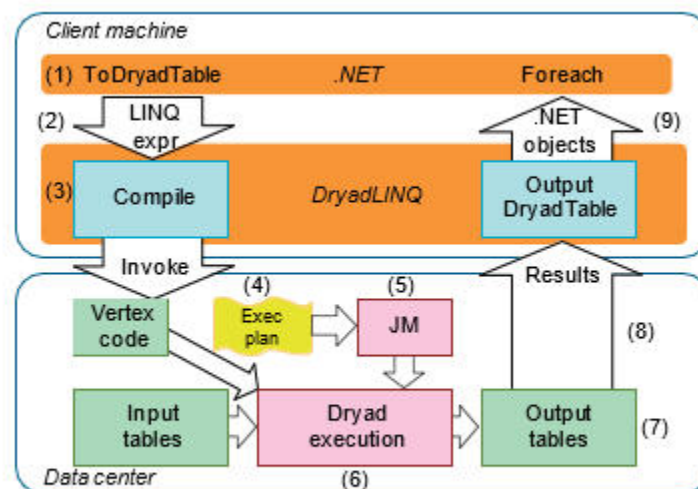


FIGURE 6.14

LINQ-expression execution in DryadLINQ.

6.2.5 Sawzall and Pig Latin High Level Languages

What is Sawzall?

- **Sawzall** is a **high-level scripting language** developed at Google.
- It runs on top of the **MapReduce** framework.
- Designed for **parallel, distributed, and fault-tolerant** processing of **very large-scale data sets** (e.g., Google's internet-scale log data).
- Initially created by **Rob Pike** to process Google's massive log files efficiently.

Key Features of Sawzall

- **Built for Scalability:**
 - Can handle large-scale data in parallel across many nodes.

Module 5- Cloud Programming and Software Environments

- Automatically uses cluster computing and redundant servers for performance and reliability.
- **Fast Data Analysis:**
 - Transformed batch jobs that took a full day into **interactive sessions**.
 - Enabled new, real-time ways to analyze and utilize big data.
- **Open Source:**
 - Sawzall has been made available as an open source project.

How Sawzall Works (Data Flow Model)

1. **Data Partitioning:**
 - Input data is split and processed **locally** using Sawzall scripts.
2. **Local Processing:**
 - Each node filters and processes its portion of the data using custom scripts.
3. **Aggregation:**
 - Intermediate results are **emitted** to **aggregators** (called tables).
 - Final results are generated by collecting data from these aggregators.

Sawzall Scripting Example (Conceptual)

1. **Tables** (aggregators) declared to count or sum data:

count: table sum of int;

total: table sum of float;

sum_of_squares: table sum of float;

2. **Input handling:**

```
x: float = input;    // Convert input to float
```

```
emit count <- 1;    // Count entries
```

```
emit total <- x;    // Sum values
```

```
emit sum_of_squares <- x * x; // Sum of squares
```

3. **Runtime Translation:**

The Sawzall engine translates scripts into **MapReduce jobs** that run across multiple machines.

Conclusion

Sawzall provides a powerful abstraction over MapReduce, allowing users to **write simple, high-level scripts** for complex data analysis tasks. Its automatic handling of parallelism, fault tolerance, and aggregation makes it well-suited for **large-scale log processing** and other similar applications.

6.2.5.1 Pig Latin

Pig Latin – High-Level Language on Hadoop

What is Pig Latin?

- **Pig Latin** is a **high-level data flow language** developed by **Yahoo!**.
- It is part of the **Apache Pig project**, which runs on top of **Hadoop**.
- Like Sawzall and DryadLINQ, Pig Latin provides an abstraction layer over **MapReduce**, simplifying the process of writing data processing programs.

Key Features of Pig Latin

- **Data Flow Language:**
 - Allows users to describe how data should be processed step-by-step.
 - Think of it as a **procedural scripting language** for manipulating large datasets.
- **Automated Parallelism:**
 - You focus on how to process **individual elements**.
 - Parallelism is **handled automatically** by the system.
 - Side effects (like aggregations) are only allowed in controlled, **collective operations**.
- **NoSQL Roots with SQL-Like Capabilities:**
 - Pig Latin, like Sawzall, originates from NoSQL ideas.
 - Unlike Sawzall, **Pig Latin supports SQL-like operations**, such as:
 - **JOIN**
 - **GROUP BY**
 - **FILTER**
 - **ORDER BY**
 - This makes it **more expressive and flexible** for structured data processing.

Module 5- Cloud Programming and Software Environments

Table 6.7 Comparison of High-Level Data Analysis Languages			
	Sawzall	PigLatin	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 operation + fixed aggregations	Sequence of MapReduce operations	DAGs
Target Runtime	Google MapReduce	Hadoop (Pig)	Dryad

Table 6.8 PigLatin Data Types		
Data Type	Description	Example

Module 5- Cloud Programming and Software Environments

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

Data Types in Pig Latin (as per Table 6.8, summarized):

- **Atom** – basic unit (e.g., int, float, string)
- **Tuple** – ordered set of fields
- **Bag** – collection of tuples
- **Map** – key-value pairs

Table 6.9 Pig Latin Operators	
Command	Description
<i>LOAD</i>	Read data from the filesystem.
<i>STORE</i>	Write data to the filesystem.
<i>FOREACH GENERATE</i>	Apply an expression to each record and output one or more records.
<i>FILTER</i>	Apply a predicate and remove records that do not return true.
<i>GROUP / COGROUP</i>	Collect records with the same key from one or more inputs.
<i>JOIN</i>	Join two or more inputs based on a key.
<i>CROSS</i>	Cross product of two or more inputs.
<i>UNION</i>	Merge two or more datasets.
<i>SPLIT</i>	Split data into two or more sets, based on filter conditions.
<i>ORDER</i>	Sort records based on a key.

Module 5- Cloud Programming and Software Environments

<i>DISTINCT</i>	Removeduplicate tuples.
<i>STREAM</i>	Send all records through a user-provided binary.
<i>DUMP</i>	Write output to stdout.
<i>LIMIT</i>	Limit the number of records.

Summary

- Pig Latin provides a **powerful, high-level interface** to Hadoop.
- Its SQL-like syntax and procedural style make it accessible to analysts and engineers.
- It balances the flexibility of scripting with the power of distributed parallel processing.
- Ideal for **semi-structured data processing**, log analysis, and ETL workflows in Hadoop ecosystems.

6.2.6 Mapping Applications to Parallel and Distributed Systems

This section explains how different types of applications map to parallel and distributed systems, based on six distinct **application architectures**. These help understand how various problems can be efficiently executed on different computing models like clusters, grids, and clouds.

Original Five Categories of Application Architectures

Category	Model	Key Features
1. SIMD (Single Instruction, Multiple Data)	Lock-step execution	All processors run the same instruction at the same time. Mostly outdated; used in early parallel machines.
2. SPMD (Single Program, Multiple Data)	Most common model today	Each unit runs the same program but on different data. Works well for scientific simulations (e.g., PDEs, particle dynamics). Typically uses compute-communicate phases.
3. Asynchronous Objects	Event-driven, dynamic tasks	Models interacting threads or agents (e.g., OS threads, graph algorithms, AI in games). Uses

Module 5- Cloud Programming and Software Environments

Category	Model	Key Features
		shared memory for low-latency communication. Rare in large-scale HPC systems.
4. Pleasingly Parallel (Embarrassingly Parallel)	Independent tasks	Tasks run completely independently. Great fit for clouds and grids. No inter-task communication needed. Simple and scalable.
5. Workflow / Coarse-Grained Linkage	Metaproblems + component problems	Combines smaller "atomic" tasks into larger workflows. Common in scientific workflows, big data pipelines. Two-level programming: workflow level and component level.

New Category: 6. MapReduce++ (Data-Intensive Applications)

This sixth category addresses modern **big data and data analytics** applications that emerged with MapReduce and its variants. It's a hybrid of categories 2 and 4 but focused specifically on data flow.

Subcategories of MapReduce++

- 1. Map-Only Applications:**
 - Similar to Category 4.
 - Each task reads data, processes it independently, and outputs results.
 - No reduce step involved (e.g., log scanning, data filtering).
- 2. Classic MapReduce:**
 - File-to-file processing with two phases:
 - Map phase:** processes input data in parallel.
 - Reduce phase:** aggregates intermediate outputs.
 - Automatically handles parallelism and fault tolerance.
- 3. Extended MapReduce:**
 - More complex patterns like iterations, joins, or graph processing.
 - Builds on traditional MapReduce with additional dataflow patterns (covered earlier in Section 6.2.2).

Why It's Unique

- Focuses on **data flow**, **scalability**, and **fault tolerance**.
- Loosely synchronous: operations occur in stages but not tightly coupled.
- Relies on **distributed file systems** and **batch processing models**.

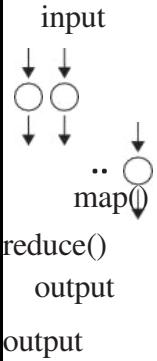
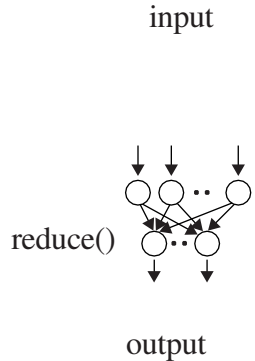
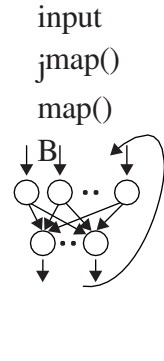
Module 5- Cloud Programming and Software Environments

Summary of Differences

- **Category 1:** Historical, rarely used today.
- **Category 2:** Widely used in scientific computing with SPMD on MIMD systems.
- **Category 3:** Suitable for OS-level or small-scale dynamic problems.
- **Category 4:** Simple, scalable, cloud-friendly tasks.
- **Category 5:** Combines multiple small problems via workflows.
- **Category 6:** Focused on modern big data applications using MapReduce-style frameworks.

This classification helps choose the right programming model and infrastructure for a given type of application—whether it's for simulations, data analysis, or event-driven systems.

Table 6.11 Comparison of MapReduce++ Subcategories along with the Loosely Synchronous Category Used in MPI

Map-Only	Classic MapReduce	Iterative MapReduce	Loosely Synchronous
			
• Document conversion scientific (e.g., PDF->HTML) • Brute force searches in cryptography • Parametric sweeps • Gene assembly	• High-energy physics applications utilizing a distributed search • Distributed sort communication constructs • Information retrieval including local interactions • Calculation of pairwise	• Expectation maximization (HEP) histograms • Linear algebra • Data mining including • Clustering • K-means	• Many algorithms • MPI • wide variety of • Solving differential equations and particle

Module 5- Cloud Programming and Software Environments

- PolarGridMatlabdata distancesforsequences •Deterministicannealing dynamicswithshort-range analysis(www.polargrid.org) (BLAST) clustering forces
 - Multidimens ional scaling (MDS)
- DomainofMapReduceandIterativeExtensions————→ MPI

Table 6.10 Application Classification for Parallel and Distributed Systems

Category	Class	Description	Machine Architecture
1	Synchronous	The problem class can be implemented with instruction-level lockstep operation as in SIMD architectures.	SIMD
2	Loosely synchronous (BSP or bulk processing)	These problems exhibit iterative compute-communication stages with independent compute (map) operations for each CPU that are synchronized with a communication step. This problem class covers many successful MPI applications including partial differential equations solutions and particle dynamics applications.	MIMD on MPP (massively parallel processor)
3	Asynchronous	Illustrated by Compute Chess and Integer Programming; combinatorial search is often supported	Shared memory

Module 5- Cloud Programming and Software Environments

		by dynamic threads. This is rarely important in scientific computing, but it is at the heart of operating systems and concurrency in consumer applications such as Microsoft Word.	
4	Pleasingly parallel	Each component is independent. In 1988, Fox estimated this at 20 percent of the total number of applications, but that percentage has grown with the use of grids and data analysis applications including, for example, the Large Hadron Collider analysis for particle physics.	Grids moving to clouds
5	Metaproblems	These are coarse-grained (asynchronous or data flow) combinations of categories 1-4 and 6. This area has also grown in importance and is well supported by grids and described by workflow in Section 3.5 .	Grids of clusters
6	MapReduce + (Twister)	This describes file (database) to file (database) operations which have three subcategories (see also Table 6.11): 6a) Pleasingly Parallel Map Only (similar to category 4) 6b) Map followed by reductions 6c) Iterative "Map followed by reductions" (extension of current technologies that support linear algebra and data mining)	Data-intensive clouds a) Master-worker or MapReduce b) MapReduce c) Twister

6.3 Programming Support of Google App Engine

6.3.1 Programming the Google App Engine

Google App Engine (GAE) is a **cloud platform** that allows developers to build and host **web applications** on Google's infrastructure. It supports languages like **Java** and **Python**, and provides built-in tools for scalable, secure, and efficient cloud development.

1. Supported Languages & Tools

- **Java Support:**
 - **Eclipse plug-in:** Enables local debugging.
 - **GWT (Google Web Toolkit):** Helps develop dynamic web apps in Java.
 - Other JVM-based languages like **JavaScript**, **Ruby** are also usable via interpreters.
- **Python Support:**
 - Common frameworks include **Django** and **CherryPy**.
 - Google provides a lightweight **webapp** framework for Python.

2. Data Management with Datastore

- **GAE Datastore:**
 - **NoSQL**, schema-less entity storage.
 - Each entity:
 - Max size: **1 MB**.
 - Identified by **key-value properties**.
 - Querying:
 - Filtered and sorted by property values.
 - Strongly **consistent** using **optimistic concurrency control**.
- **Java APIs:**
 - Use **JDO** or **JPA** via **DataNucleus Access Platform**.
- **Python API:**
 - Uses **GQL** (SQL-like query language).
- **Transactions:**
 - Multiple operations can be grouped in a single **atomic transaction**.
 - Operates within **entity groups** to maintain performance.
 - Automatic retries if conflicts occur.
- **Memcache:**
 - In-memory cache to boost performance.
 - Works with or without the datastore.
- **Blobstore:**
 - For large files (up to **2 GB**).
 - Suitable for media content (e.g., videos, images).

3. Internet & External Services Access

- **URL Fetch:**
 - Allows apps to fetch web resources using HTTP/HTTPS.
 - Uses Google's fast internal network for efficient retrieval.
- **Secure Data Connection (SDC):**
 - Tunnels through the internet to link an **intranet** with a **GAE app**.
- **Mail Service:**
 - Enables sending emails from the application.
- **Google Data API:**
 - Access services like **Maps, YouTube, Docs, Calendar**, etc., within your app.

4. User Management & Multimedia

- **Google Accounts Integration:**
 - Users can log in using existing Google accounts (e.g., Gmail).
 - Handles authentication and account creation.
- **Images API:**
 - Perform basic image operations like **resize, crop, flip, rotate**, and enhance.

5. Background Processing & Scheduling

- **Cron Service:**
 - Schedule tasks periodically (e.g., hourly, daily).
 - Ideal for maintenance, data sync, or reporting jobs.
- **Task Queues:**
 - For asynchronous background tasks triggered by application logic.
 - Helps offload long-running processes from user requests.

6. Quotas and Billing

- **Usage Limits:**
 - GAE enforces **quotas** to prevent overuse and ensure fair resource allocation.
 - **Free tier** available with limits on CPU, storage, bandwidth, etc.
 - Ensures **cost control** and **performance isolation** between apps.

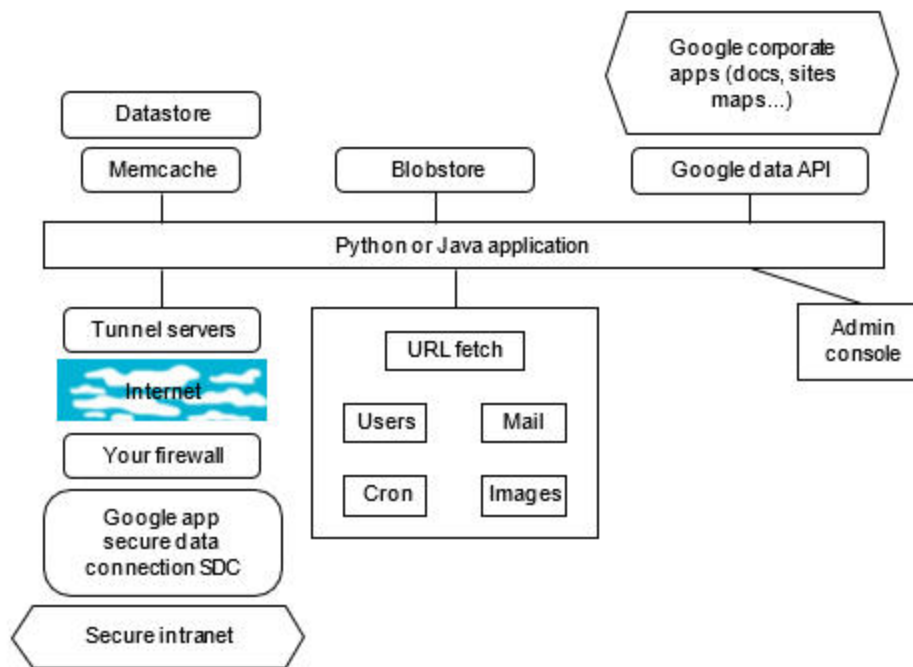


FIGURE 6.17

Programming environment for Google AppEngine.

Summary

Google App Engine simplifies the deployment of scalable web applications using familiar languages like Java and Python. It offers powerful features such as built-in data storage, background processing, and access to Google services — all while managing infrastructure, scaling, and cost control for you.

6.3.2 Google File System (GFS)

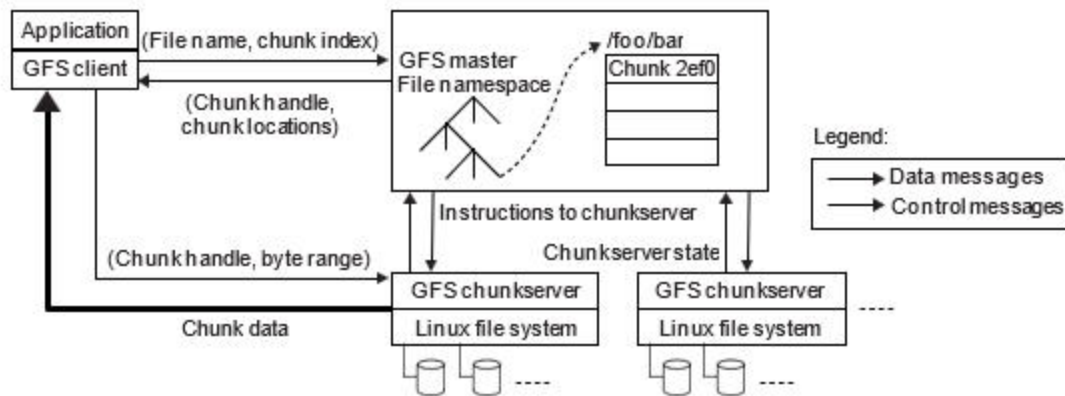


FIGURE 6.18

Architecture of Google File System (GFS).

GFS was developed by **Google** to support the massive data needs of its **search engine**, designed specifically for storing and processing **huge volumes of data** on **cheap, unreliable hardware**.

1. Design Motivations

- Traditional file systems could not handle:
 - Petabyte-scale data**
 - Frequent hardware failures**
- GFS was **co-designed with Google's applications**, leading to tight integration (non-standard, unlike POSIX-compliant systems).

2. Key Assumptions

- High Failure Rates:**
 - Commodity hardware often fails; this is expected, not exceptional.
- Large File Sizes:**
 - Files are typically 100 MB or larger (often several GB).
 - Block size = **64 MB** (vs. 4 KB in traditional systems).
- I/O Pattern:**
 - Files are written **once**, often **appended**, and then **read in large sequential streams**.
 - Low **random access** usage.

3. System Architecture (Figure 6.18)

- Single Master:**
 - Manages metadata (file names, block locations, leases).

- Simplifies cluster management but is a **potential bottleneck**.
- **Chunk Servers:**
 - Store actual file data in large chunks (64 MB).
 - Each chunk is **replicated** (default: 3 copies) across servers for **fault tolerance**.
- **Clients:**
 - Communicate with the **master** for metadata.
 - Access **chunk servers directly** for reading/writing data.

4. Fault Tolerance & Availability

- **Shadow Master:**
 - Mirrors the main master to recover from master failure.
- **Replication:**
 - Each chunk is stored in **at least three servers**.
 - Can tolerate **two simultaneous failures**.
- **Checksum Verification:**
 - Each 64 KB sub-block has a checksum for data integrity.
- **Fast Recovery:**
 - Masters and chunk servers restart in **seconds**.

5. Data Mutation Workflow (Figure 6.19)

Steps for Writing/Appending Data:

1. **Client → Master:** Asks for chunk lease and replica locations.
 2. **Master → Client:** Responds with primary + secondary chunk servers.
 3. **Client → All Servers:** Pushes data to all replicas.
 4. **Client → Primary:** Requests write.
 5. **Primary → Secondaries:** Forwards request in serial order.
 6. **Secondaries → Primary:** Confirm success.
 7. **Primary → Client:** Final success/failure notification.
- **If errors occur:** Client retries steps 3–7 or restarts the entire process.

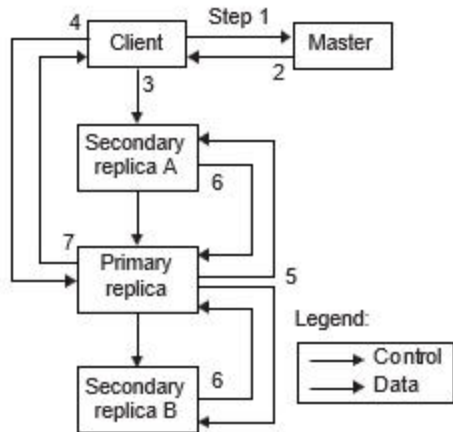


FIGURE 6.19
Data mutation sequence in GFS.

6. Special Feature: Record Append

- Used for concurrent data appending (e.g., by web crawlers).
- Ensures:
 - Data is appended **at least once**.
 - **Offset chosen by GFS**, not client.
 - Supports **atomic appends** for multiple writers.

7. Differences from Traditional File Systems

- **Not POSIX-compliant**, but has a similar API.
- Allows applications to access **physical data locations**.
- Designed for **Google-specific needs**, not general-purpose usage.

8. Advantages

- **Scalable** to thousands of nodes.
- **Highly available** and **fault-tolerant**.
- Efficient for **large streaming reads** and **append-heavy workloads**.
- Optimized for **commodity hardware** with frequent failures.

Summary

GFS is a groundbreaking distributed file system tailored for Google's massive data needs. It breaks away from traditional designs to emphasize **fault tolerance**, **high throughput**, and **scalability** on **commodity infrastructure**, making it a key foundation for systems like **MapReduce**.

6.3.3 Big Table, Google's NOSQL System

BigTable is a **distributed, scalable, NoSQL database** developed by Google for managing **structured and semi-structured data** across **massive scale**. It is designed to power a wide range of Google's applications like **Google Search, Orkut, Google Maps, and Google Earth**.

1. Purpose and Use Cases

BigTable is used for:

- **Web page storage:** URLs, content, metadata, links, and PageRank.
- **User data storage:** Preferences, search history, Gmail messages.
- **Geographic data:** Roads, satellite images, places, annotations.

2. Motivations Behind BigTable

- **Commercial databases** can't handle Google's massive scale and performance needs.
- Needed a **custom-built system** for:
 - Billions of records (e.g., URLs).
 - High user activity (e.g., thousands of queries/sec).
 - Huge data sizes (e.g., >100 TB of geographic data).

3. Key Design Goals

- **High throughput** for millions of read/write operations per second.
- **Asynchronous updates** for real-time applications.
- **Efficient scans and joins** over large datasets.
- **Historical data access** (e.g., changes in webpage content over time).

4. Conceptual View

- BigTable is like a **multilevel distributed map**:
 - **(row key, column key, timestamp) → value**
 - Stores data in a **sparse, multidimensional sorted map** format.
- Rows are **lexicographically ordered**, making **range scans** efficient.

5. System Scale and Capabilities

- Thousands of servers.
- Terabytes of in-memory data.
- Petabytes of data on disk.
- **Self-managing**:

Module 5- Cloud Programming and Software Environments

- Dynamic server addition/removal.
- **Automatic load balancing.**
- Used in Google since **2004**.
- One BigTable cell can manage **~200 TB** across thousands of machines.

6. Core Infrastructure Components

BigTable is built atop Google's internal cloud infrastructure, using:

Component	Function
GFS (Google File System)	Stores persistent data
Scheduler	Manages job scheduling for BigTable operations
Lock Service (Chubby)	Handles master node elections and service coordination
MapReduce	Used for reading/writing and bulk operations on BigTable

7. Summary

BigTable is a **high-performance, scalable, and fault-tolerant NoSQL system** tailored for Google's unique requirements. Its success lies in tight integration with the underlying Google infrastructure, and its ability to support billions of rows, flexible schema, and fast access in a cloud-native way.

Example 6.8 – BigTable Data Model in Mass Media

BigTable's **data model** is **simplified yet powerful**, designed to handle **large-scale, structured and semi-structured data**, such as web pages, user data, and media content.

1. Basic Concept: A Distributed Sparse Map

- BigTable is essentially a **sparse, multidimensional sorted map**.
- Each **cell** is addressed using a **3-part key**:
 - **Row key (string)**
 - **Column key (string)** → formatted as **family:qualifier**
 - **Timestamp (int64)**

Data mapping:

text

CopyEdit

(row: string, column: string, timestamp: int64) → string (cell value)

2. Example: Web Table

- Stores information about web pages.
- **Row Key**: URL of the web page.
- **Columns**:

- Different **content versions** of the web page.
- **Anchor text** and links on the page.
- **Multiple versions** (with timestamps) are stored in the same cell.

3. Key Features

- **Lexicographical Row Ordering:**
 - Rows are stored and accessed in sorted order.
 - Row key choice can optimize data locality and access speed.
- **Atomic Row Access:**
 - Operations on a row are **atomic**.
 - Unlike SQL databases, BigTable provides **limited atomicity**—only at row level.
- **Implicit Row Creation:**
 - Rows are automatically created when data is inserted.

4. Tablets: Partitioning and Load Balancing

- Tables are divided into **tablets** (chunks of rows).
- Each tablet:
 - Stores **~100 MB to 200 MB** of data.
 - Contains a **contiguous row range**.
- **Tablet servers:**
 - Each manages ~100 tablets.
 - Tablet migration allows **fine-grained load balancing**.
 - In case of a failure, tablets are redistributed for **fast recovery**.

5. BigTable System Architecture (Figure 6.20b)

- **BigTable Master:**
 - Manages **metadata** and **tablet assignments**.
 - Makes decisions about **load balancing**.
- **Tablet Servers:**
 - Store and serve tablets to clients.
- **Clients:**
 - Use a **BigTable client library** to communicate with master and tablet servers.
- **Chubby** (Distributed Lock Service):
 - Handles **master election**, **metadata consistency**, and **synchronization**.

Summary

BigTable's model supports high scalability, efficient large-scale data access, and fault tolerance. It uses a simple yet powerful key-value system enhanced with time and column structure, making it ideal for applications like **search indexing**, **web crawling**, **user data storage**, and **media metadata**.

6.3.3.1 – Tablet Location Hierarchy in BigTable

BigTable uses a **three-level hierarchy** to **locate tablets**, ensuring fast and reliable access to data.

Tablet Lookup Process (Figure 6.21):

1. **Root Tablet Location:**
 - Stored in a file managed by **Chubby**.
 - Root tablet contains metadata about other **METADATA tablets**.
 - This file is **never split** and guarantees a max of **three lookup levels**.
2. **METADATA Tablets:**
 - Each entry points to **user tablets**.
 - Indexed by a **row key** that encodes the table ID and end row.
 - Contains the location of all user data tablets.
3. **User Tablets:**
 - Contain actual user data rows and columns.

Key Features:

- **Fast Lookup:** Typically requires at most **3 disk accesses**.
- **Fault Tolerance:** Chubby ensures **availability** of the root tablet.
- **Tablet Compaction:** Servers use compaction to optimize data storage.
- **Shared Logs:** Used to record operations for multiple tablets, improving efficiency and consistency.

6.3.4 – Chubby: Google's Distributed Lock Service

Chubby is Google's **coarse-grained distributed lock service** used to **coordinate distributed systems** like GFS and BigTable.

Key Functions:

- Acts as a **lock manager** to elect **primary replicas**.
- Stores **small control files** in a **hierarchical namespace** (like a filesystem tree).
- Provides **high availability and consistency** using the **Paxos protocol**.

Architecture (Figure 6.22):

- Each **Chubby cell** has **5 servers**.
 - Each server has a copy of the namespace.
 - Paxos protocol ensures that **only one leader (master)** is elected at a time.
- Clients:
 - Use the **Chubby client library** to interact with servers.
 - Can read/write small files and acquire locks.
 - Communicate with any server; the leader coordinates writes.

Why Chubby Is Important:

- Used by **BigTable and GFS** to:
 - Elect master servers.
 - Manage metadata files and locks.
- **Primary Use:** Google's **internal name and configuration service**.

Summary

Component	Purpose	Key Points
Tablet Hierarchy	Fast tablet lookup in BigTable	3-level structure: Chubby → Root Tablet → METADATA → User Tablets
Chubby	Distributed lock and config service	Ensures master election, fault tolerance, and consistency using Paxos

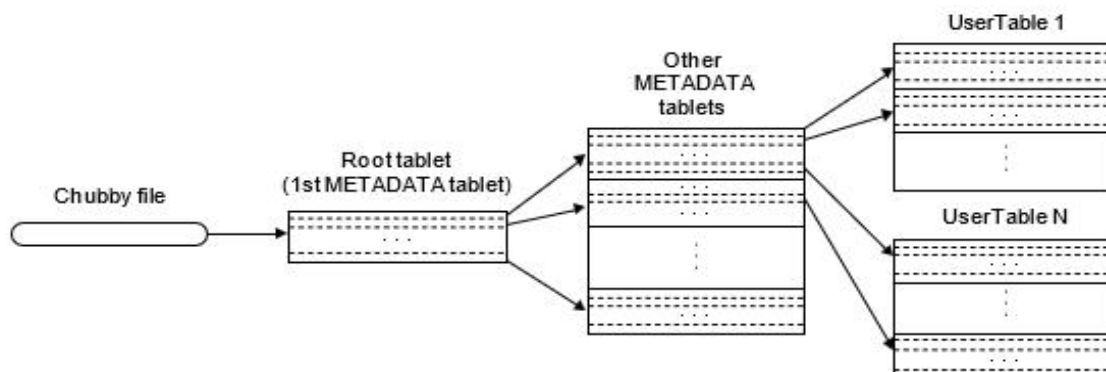


FIGURE 6.21

Tablet location hierarchy in using the BigTable.

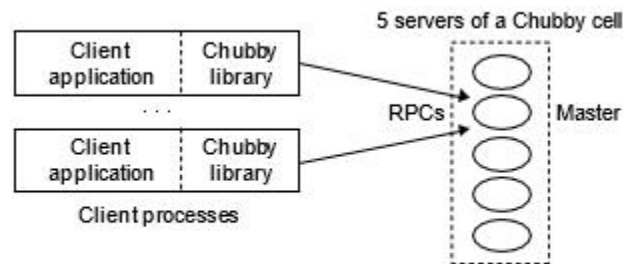


FIGURE 6.22
Structure of Google Chubby for distributed lock service.

6.4 Programming on Amazon AWS and Microsoft Azure

6.4.1 Programming on Amazon EC2

Amazon EC2 (Elastic Compute Cloud) is a key component of Amazon Web Services (AWS) that allows users to rent **virtual machines (VMs)** for running applications. It was the **first cloud service** to offer **VM-based application hosting**, pioneering the Infrastructure-as-a-Service (IaaS) model.

1. Core Features

- **Virtual Machine Hosting:**
 - Users rent VMs instead of physical hardware.
 - Install and run any software on selected operating systems.
- **Elasticity:**
 - Create, launch, and terminate VM instances on-demand.
 - Pay **only for the time** the VMs are active (per hour billing).

2. Amazon Machine Images (AMIs)

- **AMIs** are pre-configured templates for launching EC2 instances.
 - Include OS (Linux/Windows) and optional pre-installed software.
- **Types of AMIs:**
 - **Public AMIs** – Shared for general use.
 - **Private AMIs** – Only accessible to the creator.
 - **Paid AMIs** – Provided by third parties for a fee.

VM Launch Workflow (Figure 6.24):

Create AMI → Create Key Pair → Configure Firewall → Launch Instance

Module 5- Cloud Programming and Software Environments

3. EC2 Instance Classes (Table 6.13)

Class	Purpose
1. Standard	General-purpose usage.
2. Micro	Low-throughput tasks with occasional CPU bursts.
3. High-Memory	Suitable for memory-intensive apps like databases.
4. High-CPU	Ideal for compute-heavy tasks (e.g., simulations).
5. Cluster Compute	HPC and network-intensive workloads using high-speed networking (10 Gbps).

- Instance performance is measured in **EC2 Compute Units (ECUs)**.
 - 1 ECU $\approx$ 1.0–1.2 GHz CPU from a 2007 Xeon/Opteron.

4. Cost Considerations

- **Hourly billing** based on:
 - Instance type and size.
 - CPU usage (in ECUs).
- Other charges apply for:
 - **Storage** (e.g., EBS volumes),
 - **Networking**,
 - **Data transfer**, etc.
- **Current pricing**: Check AWS's official site for up-to-date costs.

5. Example 6.9 – Use in Real-World Applications

- EC2 can be used to power a range of apps, from **simple websites** to **complex enterprise solutions**.
- Real-world usage often involves:
 - Running **databases**, **web servers**, or **data processing jobs**.
 - **Scaling** up/down based on traffic or computational needs.

Summary

Amazon EC2 offers a **flexible, scalable, and cost-efficient** platform for hosting applications in the cloud. It supports a wide range of instance types to suit different workloads and allows users full control

Table 6.12 Three Types of AMI

Module 5- Cloud Programming and Software Environments

ImageType	AMIDefinition
Private	AMI Images created by you, which are private by default. You cannot 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.

Table 6.13 Instance Types Available on Amazon EC2 (October 6, 2010)					
Compute Instance	Memory GB	ECU or EC2 Units	Virtual Cores	Storage GB	32/64 Bit
Standard: small	1.7	1	1	160	32
Standard: large	7.5	4	2	850	64
Standard: extra large	15	8	4	1690	64
Micro	0.613	Up to 2		Only EBS	32 or 64
High-memory	17.1	6.5	2	420	64

Module 5- Cloud Programming and Software Environments

High-memory:double	34.2	13	4	850	64
High-memory:quadruple	68.4	26	8	1690	64
High-CPU:medium	1.7	5	2	350	32
High-CPU:extralarge	7	20	8	1690	64
Clustercompute	23	33.5	8	1690	64

6.4.2 Amazon Simple Storage Service (S3)

Amazon **S3** is a **web-based object storage service** that allows users to store and retrieve **any amount of data, anytime, from anywhere** via web protocols.

Core Concepts

- **Object Storage:** Each **object** contains data, metadata, and access control, and is stored in a **bucket**.
- **Key-Value Access:** Each object is accessed via a **unique key**.

Access Interfaces

- **REST (Web 2.0)** and **SOAP** APIs.
- Access via browsers or client applications.

Key Features

- **Durability:** 99.999999999% (11 nines) durability.
- **Availability:** 99.99% annually.
- **Redundancy:** Geo-distributed replication.
- **Security:** Object-level **ACLs**, **authentication**, and **private/public settings**.
- **Protocols:** Default access via **HTTP**, BitTorrent support for large-scale distribution.
- **Pricing** (as of 2010):
 - \$0.055–\$0.15 per GB/month (based on volume).
 - Free for the first 1 GB/month in/out; data transfers beyond that are charged.

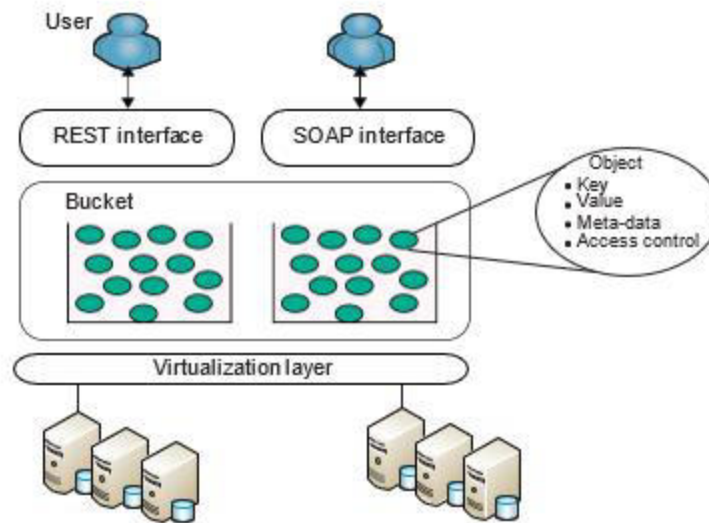


FIGURE 6.24

Amazon S3 execution environment.

6.4.3 – Amazon EBS (Elastic Block Store)

EBS offers **block-level storage volumes** for use with EC2 instances. It's akin to attaching a virtual hard disk to a virtual machine.

Key Features

- **Persistence:** Unlike EC2 instance storage, data is retained after the instance is stopped.
- **Block Device Interface:**
 - Volumes from **1 GB to 1 TB**.
 - Can be formatted with a file system or used directly.
- **Multiple Volumes:** Can be mounted on the same EC2 instance.
- **Snapshots:** Incremental backups improve save/restore efficiency.
- **Pricing** (as of 2010):
 - \$0.10 per GB/month for storage.
 - \$0.10 per million I/O requests.

6.4.3.1 – Amazon SimpleDB

SimpleDB is a **lightweight NoSQL database** designed for small-scale structured data management.

Data Model

Module 5- Cloud Programming and Software Environments

- **Domain** = Table
- **Item** = Row
- **Attribute** = Column
- **Value** = Cell (can have **multiple values** per attribute)
- No strict schema or ACID transactions; **eventual consistency** model.

Use Case

- Suited for **metadata**, configuration data, and small records.
- Simpler and more flexible than traditional relational databases.

Pricing (as of 2010)

- \$0.14 per SimpleDB machine hour.
- First 25 hours/month are free.

Comparison Summary

Service	Type	Use Case	Interface	Pricing (2010)
S3	Object Storage	Media, backups, static content	REST, SOAP	\$0.055– \$0.15/GB/month
EBS	Block Storage	Persistent storage for EC2	Block Device	\$0.10/GB + I/O costs
SimpleDB	NoSQL Database	Metadata, config, small structured data	Web API	\$0.14/hr (first 25 hrs free)

6.4.4 Microsoft Azure Programming Environment

Azure is Microsoft's cloud platform offering **virtualized compute, storage, and database services**. It supports scalable application hosting through a **role-based architecture** and integrates a range of storage models.

1. Azure Fabric and Roles

- The **Azure Fabric** is the underlying infrastructure:
 - Manages **VMs**, load balancing, fault tolerance, and resource allocation.
 - Supports **automated service management** via XML templates.

Types of Roles

Module 5- Cloud Programming and Software Environments

1. Web Role:

- Customized VM for hosting web applications (via IIS).

2. Worker Role:

- For general-purpose background processing.

Lifecycle Methods:

- OnStart(): Initialization tasks.
- OnStop(): Graceful shutdown.
- Run(): Main application logic.

Debugging:

- No live debugging on cloud instances.
- Use **trace logs** and performance counters for diagnostics.

2. SQLAzure (6.4.4.1)

- **SQLAzure** = SQL Server as a service.
- Used for **relational database management**.
- Fully managed and **scalable** with **replication** for fault tolerance.

3. Azure Storage Services

Blob Storage:

- Similar to **Amazon S3**, used for large data.
- Organized

as:

Account → Container → Block/Page Blobs

Blob Type	Purpose	Max Size
Block Blob	Streaming data (e.g., media)	200 GB
Page Blob	Random read/write access	1 TB

- **Metadata**: <name, value> pairs (up to 8 KB per blob).
- **Drives**: NTFS volumes backed by blob storage, similar to **Amazon EBS**.

4. Azure Tables (6.4.4.2)

NoSQL **key-value store** suitable for metadata and scalable structured data.

Module 5- Cloud Programming and Software Environments

- Each **entity (row)** has:
 - Up to **255 properties**.
 - **PartitionKey**: Groups entities for optimized access.
 - **RowKey**: Unique identifier for each entity.
- **Max entity size**: 1 MB (use blob links for larger data).
- **Query Support**: ADO.NET, LINQ.

5. Azure Queues

- Used for **message queuing** between web and worker roles.
- Supports reliable delivery (processed at least once).
- Each message: up to **8 KB**.
- Operations: PUT, GET, DELETE, CREATE, DELETE.

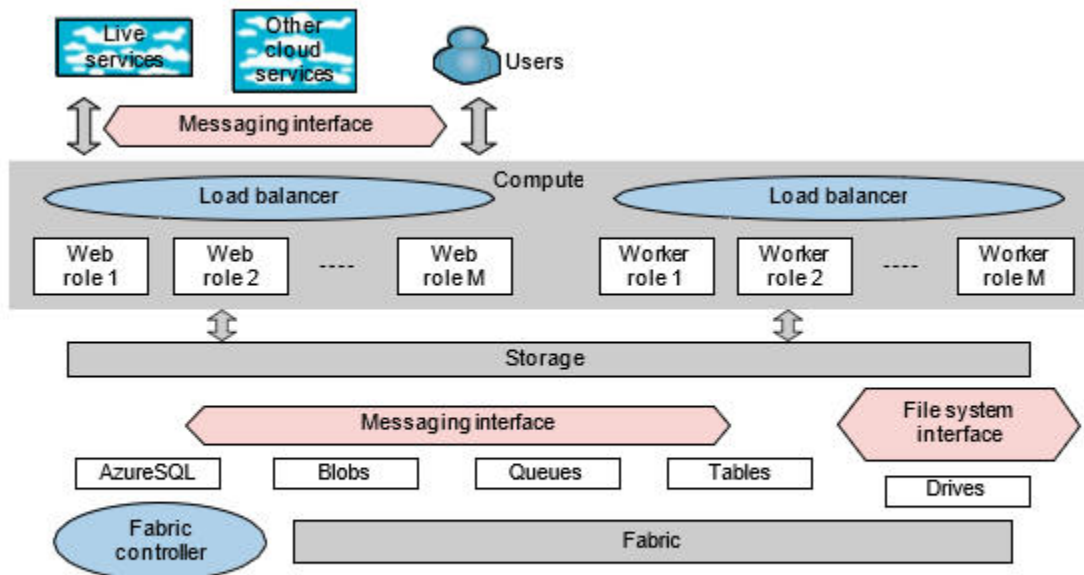


FIGURE 6.25

Features of the Azure cloud platform.

Summary Table

Module 5- Cloud Programming and Software Environments

Feature	Purpose	Analogous To
Web Role	Host web apps	AWS EC2 Web Servers
Worker Role	Background jobs	AWS Lambda / EC2
SQL Azure	Relational DB	Amazon RDS
Blob Storage	Large files	Amazon S3
Page Blobs / Drives	Disk storage	Amazon EBS
Tables	NoSQL DB	Amazon SimpleDB
Queues	Messaging	Amazon SQS

6.5 Emerging Cloud Software Environments

This section introduces **open-source and research-oriented cloud platforms** and tools designed to support **cloud programming, VM management, storage, and data processing** across diverse infrastructures.

6.5.1 – Eucalyptus and Nimbus

Eucalyptus

- Origin: UC Santa Barbara, commercialized by Eucalyptus Systems.
- Emulates **Amazon EC2 and S3**:
 - **Walrus** = S3-like storage.
- Allows users to **upload/manage VM images**, bundle root filesystems, and register them for deployment.

Architecture & Features

- Supports VM lifecycle management, image repositories.
- Available in both **open source** and **commercial** versions.

Nimbus

- Open source **IaaS cloud** solution.
- Enables leasing remote resources via **VM deployment**.
- **Nimbus Web**: Django-based web interface for admin and user tasks.

Key Features

- **Cumulus**: S3-compatible storage system with quotas.

Module 5- Cloud Programming and Software Environments

- **Client Support:** Works with EC2 clients, uses Java Jets3t, boto.
- **Resource Management:**
 - **Resource Pool Mode:** Direct control of VM nodes.
 - **Pilot Mode:** Works with local LRMS for VM provisioning.

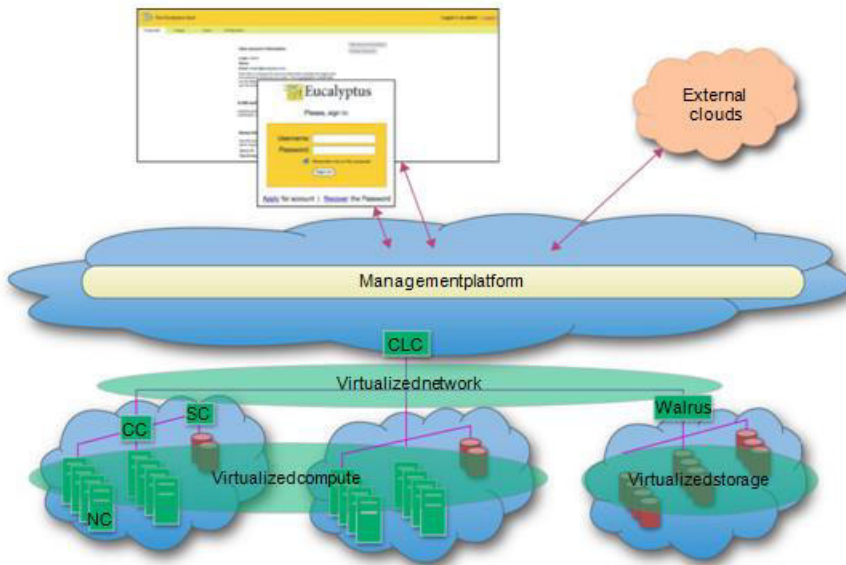


FIGURE 6.26
The Eucalyptus architecture for VM management.

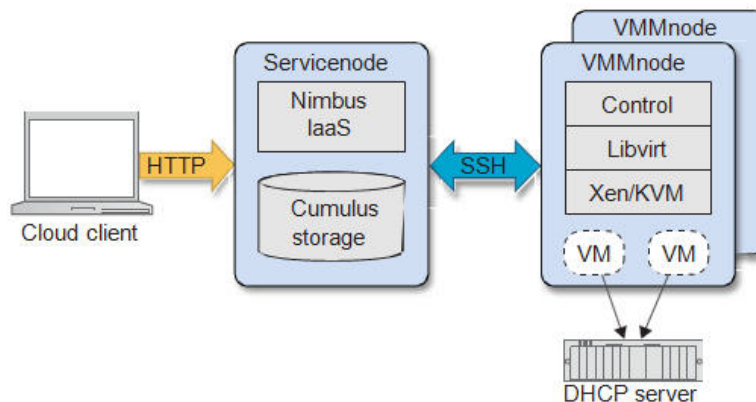


FIGURE 6.27
Nimbus cloud infrastructure.

6.5.2 – OpenNebula, Sector/Sphere, and OpenStack

OpenNebula

- **Modular IaaS cloud platform.**

Module 5- Cloud Programming and Software Environments

- Manages full VM lifecycle and **dynamic networking**.
- Uses **libvirt API**, CLI, and cloud drivers for hybrid clouds (e.g., Amazon EC2, ElasticHosts).
- Supports:
 - **VM migration**, snapshots.
 - **Capacity scheduler** with rank/requirement model.
 - **Image repository** for disk image management.

Sector/Sphere

- Designed for **big data storage** and **parallel processing**.

Sector (Storage)

- Wide-area DFS with **replica placement** based on network topology.
- Uses **UDP for control, UDT for data**.
- Integrated with FUSE and provides programming APIs.

Sphere (Processing)

- Works with Sector to process data using **user-defined functions (UDFs)**.
- Emphasizes **data locality** and **fault tolerance**.
- Supports pipelining and chaining of Sphere segments.

Space: Column-based table storage engine in Sector/Sphere, supporting a limited SQL subset.

OpenStack

- Open-source cloud platform started by **Rackspace and NASA**.
- Focus: **Compute (Nova)** and **Storage (Swift)**

OpenStack Compute (Nova)

- Message-based, **shared-nothing architecture**.
- Written in Python.
- Supports **Amazon APIs (via boto)**.
- Components:
 - **API Server, Cloud Controller, User Manager (LDAP)**
 - **Networking Components:**
 - **NetworkController** (VLANs)
 - **RoutingNode** (NAT, firewalls)
 - **TunnelingNode** (VPN)

Module 5- Cloud Programming and Software Environments

- **AddressingNode** (DHCP)

OpenStack Storage (Swift)

- Includes:
 - **Proxy Server:** Routes data requests.
 - **Ring:** Maps entity names to physical locations.
 - **Object, Container, Account Servers.**
- Supports **replication, failure isolation, and heterogeneous storage.**
- Objects stored as binary files with extended attributes.

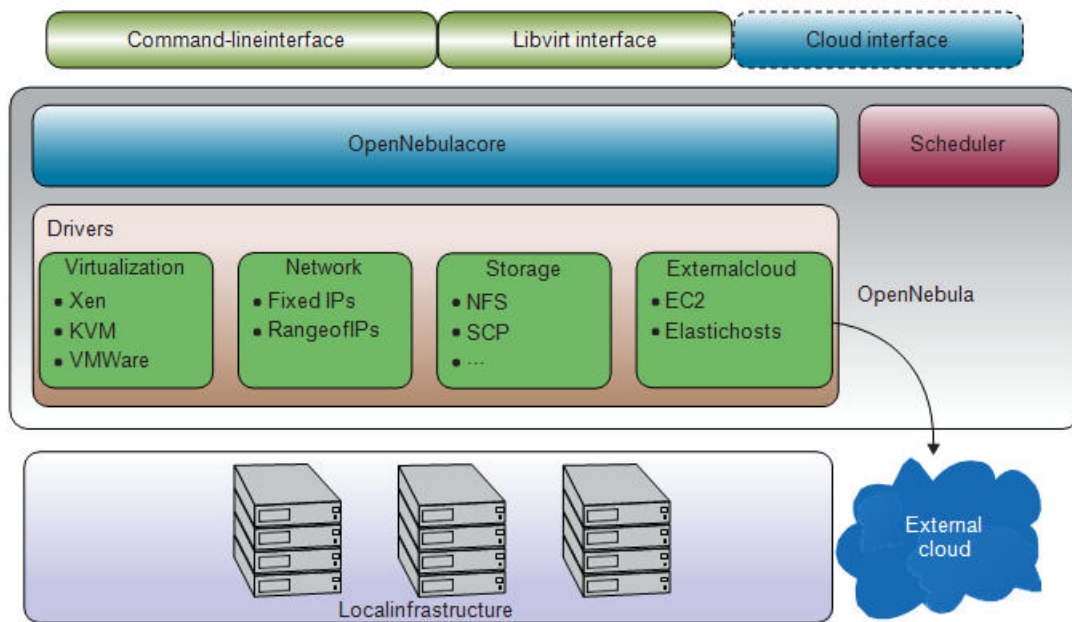


FIGURE 6.28

OpenNebula architecture and its main components.

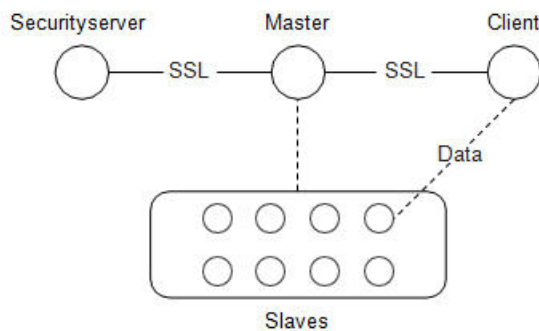


FIGURE 6.29

TheSector/Sphere system architecture.

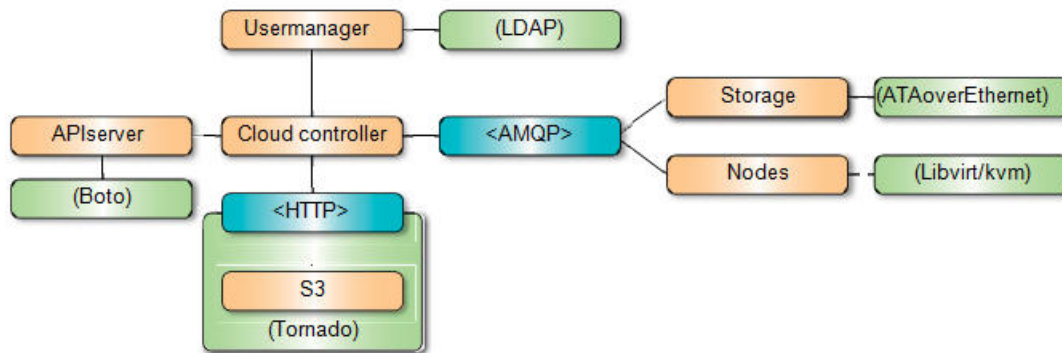


FIGURE 6.30

OpenStack Nova system architecture. The AMQP (Advanced Messaging Queuing Protocol) was described in [Section 5.2](#).

6.5.3 – Aneka Cloud (Manjrasoft)

Aneka is a cloud platform for developing and running parallel and distributed applications, built on **.NET** but supports **Linux** via Mono.

Key Capabilities

1. **Build:**
 - SDK with APIs for app development.
 - Deploy on **private**, **public**, or **hybrid** clouds.
2. **Accelerate:**
 - Rapid deployment on multiple runtime environments.
 - Dynamically lease public cloud resources to meet **QoS/SLA** deadlines.
3. **Manage:**
 - GUI + API-based infrastructure monitoring.
 - Includes accounting, SLA tracking, and dynamic provisioning.

Supported Programming Models:

1. **Thread Model:** Best for multi-core processing.
2. **Task Model:** For independent/batch tasks.
3. **MapReduce Model:** For data-parallel processing.

Comparison Snapshot

Platform	Focus	Special Features

Module 5- Cloud Programming and Software Environments

Platform	Focus	Special Features
Eucalyptus	EC2-like IaaS	Compatible APIs, Walrus (S3-like)
Nimbus	IaaS + S3 storage	Python interface, Cumulus, EC2 API
OpenNebula	Modular IaaS	VM lifecycle, hybrid cloud support
Sector/Sphere	Big Data Processing	DFS + UDF-based compute engine
OpenStack	General Cloud IaaS	Compute (Nova), Storage (Swift), strong community
Aneka	Cloud Programming Platform	Multi-model programming, SLA-aware scaling

6.5.3.1 Aneka Architecture

Aneka is a cloud computing platform that provides a flexible environment for running distributed applications. It works by connecting multiple physical or virtual machines, each running a lightweight software called the **Aneka container**. This container manages the services on each machine and interacts with the operating system through a component called the **Platform Abstraction Layer (PAL)**. PAL hides the differences between various operating systems and helps perform system tasks like monitoring performance and ensuring quality of service.

Aneka's architecture is made up of **three types of services**. First are **Fabric Services**, which handle core infrastructure tasks like monitoring hardware, managing nodes, and ensuring system reliability. Second are **Foundation Services**, which add useful features like storage management, accounting, billing, and resource reservation—helping both system administrators and developers. Lastly, **Application Services** provide the environment needed to run applications, using the other services to handle tasks like data transfer and performance tracking. Aneka can run different types of application models like distributed threads, bag of tasks, and MapReduce. It supports easy customization and expansion through its SDK and Spring framework, allowing developers to add new features quickly.

Module 5- Cloud Programming and Software Environments

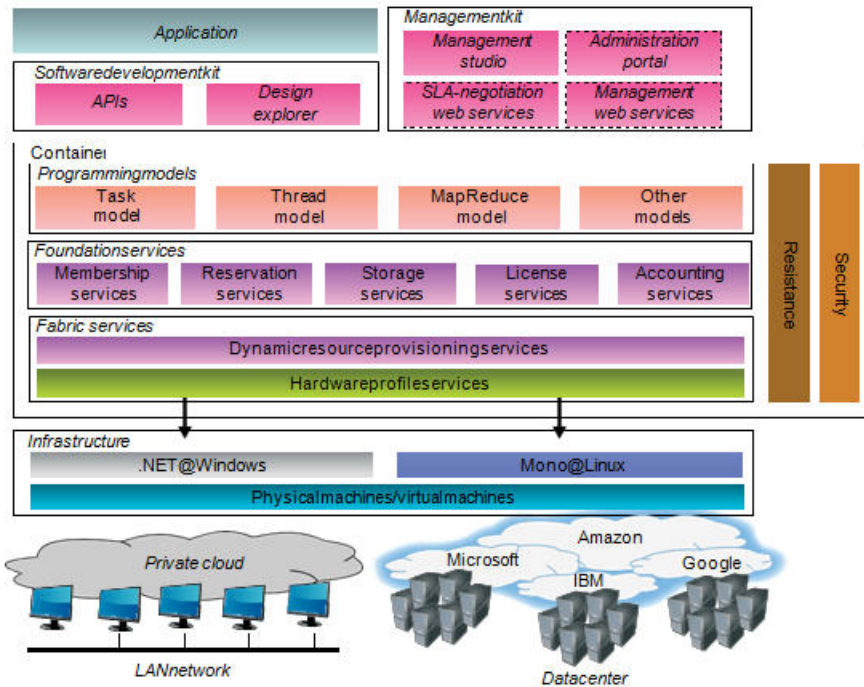


FIGURE 6.31

Architecture and components of Aneka.

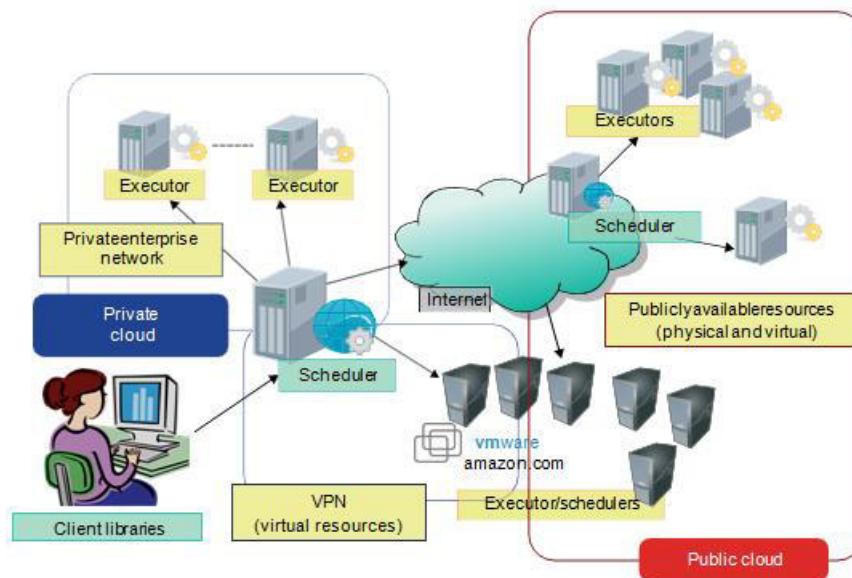


FIGURE 6.32

Aneka using private cloud resources along with dynamically leased public cloud resources.

Example 6.11: Aneka Application of Maya Rendering – Case Study:

Module 5- Cloud Programming and Software Environments

GoFront Group, a leading Chinese manufacturer of railway equipment, needed to render high-quality 3D images for the design of high-speed trains and urban transport vehicles using Autodesk Maya software. Rendering these complex designs on a single 4-core server used to take about **three days**, especially since each scene included over 2,000 frames with multiple camera angles. To solve this problem and **speed up the rendering process**, GoFront used **Aneka**, a cloud software platform, to build a **private enterprise cloud** by connecting the PCs within their company network. They used a tool called **Aneka Design Explorer**, which helps run the same program many times on different data sets—perfect for rendering many frames in parallel. A custom interface called **Maya GUI** was developed to integrate Maya with Aneka, allowing users to manage parameters, submit rendering tasks, monitor progress, and collect the results. With just **20 PCs** running Aneka, GoFront was able to reduce the rendering time **from three days to just three hours**, dramatically improving productivity and design turnaround time.

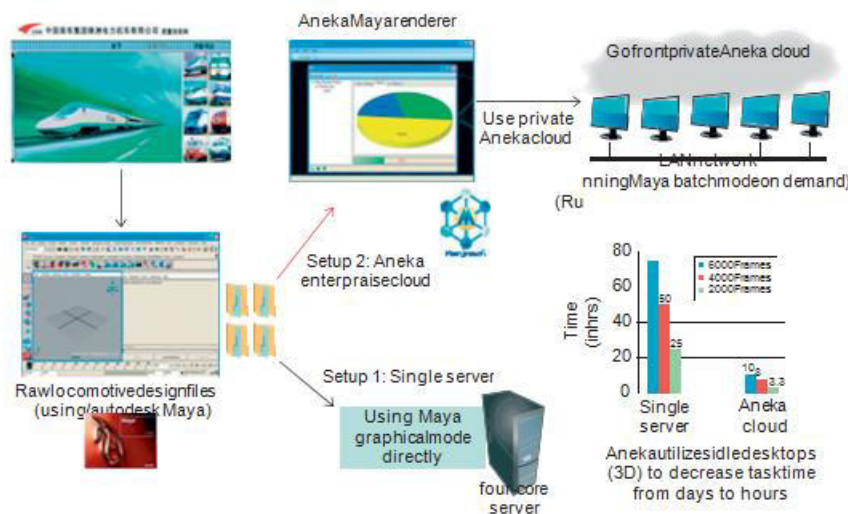


FIGURE 6.33
Rendering images of locomotive design on GoFront's private cloud using Aneka.

6.5.3.2 Virtual Appliances

Virtual appliances are specially prepared **virtual machines (VMs)** that contain everything needed to run an application, including the operating system, libraries, and setup files. These VMs are designed to run out-of-the-box, meaning they are preconfigured and ready to use immediately once started. Aneka uses virtual appliances to make application deployment easier, especially across large, mixed computing environments.

The use of **VM technology** (like VMware, VirtualBox, or Xen) allows Aneka to create virtual clusters that are consistent and easy to manage. These virtual appliances reduce software compatibility issues since the entire software stack is already inside the appliance. Even if the underlying hardware or operating systems differ, the application runs the same way. This

Module 5- Cloud Programming and Software Environments

approach is especially helpful in **grid computing**, where systems are spread across different networks and may face issues like firewalls or NAT. Virtual appliances help simplify deployment and ensure smooth performance in such distributed environments.