

Computing Grids vs. Clouds – Key Differences

1. Workflow Management

- **Computing Grids**

- Workflows are **scientific, batch-oriented**, and often complex.
- Workflow scheduling is **manual or semi-automated**.
- Designed for **long-running jobs** in research and HPC environments.
- Workflow execution depends on **resource availability across multiple organizations**.

- **Cloud Computing**

- Workflows are **service-oriented**, dynamic, and often real-time.
- Strong support for **automated workflow orchestration** (e.g., AWS Step Functions, Azure Logic Apps).
- Designed for **on-demand, scalable, short to medium-length tasks**.
- Workflow reliability ensured through **virtualized resources and automation**.

2. Data Transport

- **Computing Grids**

- Data usually distributed across **multiple institutions or research centers**.
- Data movement relies on protocols like **GridFTP**, Globus Toolkit.
- High overhead due to **cross-domain transfers** and manual configuration.

- **Cloud Computing**

- Data stored inside centralized provider-defined datacenters.
- Fast, internal data transfer using optimized fabric (e.g., Amazon S3 → EC2).
- Native tools for seamless data integration (S3, Blob Storage, BigQuery).
- Lower latency due to **intra-cloud high-speed networks**.

3. Security

- **Computing Grids**

- Security is **complex**, as resources span multiple organizations.
- Must use certificate-based authentication (X.509), public key infrastructure.
- Requires **trust relationships** across institutions.
- More vulnerable due to **heterogeneous environments**.

- **Cloud Computing**

- Security managed centrally by the cloud provider.
- Strong identity management systems: IAM, RBAC, MFA.
- Data encryption built-in (at rest & in transit).
- Uniform security policies applied across resources.

4. Availability

- **Computing Grids**

- Availability depends on **voluntary, distributed resources**.
- Resources may be unreliable or go offline without notice.
- No strict SLAs; grid nodes may come from different administrative domains.

- **Cloud Computing**

- Provides **high availability** backed by commercial SLAs (99.9%+).
- Redundant datacenters across regions/zones.
- Automated failover, replication, and load balancing.
- Designed to support **24/7 business-critical applications**.

Parallel And Distributed Programming Paradigms

Introduction

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

- Running a parallel program on a distributed computing system (parallel and distributed programming) has several advantages for both users and distributed computing systems.
- From the users' perspective, it decreases application response time.
- From the distributed computing systems standpoint, it increases throughput and resource utilization.

System Issues for running a parallel program

- Issues for running a typical parallel program in either a parallel or a distributed manner would include the following :
 - Partitioning
 - Mapping
 - Synchronization
 - Communication
 - Scheduling

- **Partitioning:** This is applicable to both computation and data as follows:
 - Computation partitioning:
 - This splits a given job or a program into smaller tasks. Partitioning greatly depends on correctly identifying portions of the job or program that can be performed concurrently. In other words, upon identifying parallelism in the structure of the program, it can be divided into parts to be run on different workers. Different parts may process different data or a copy of the same data.
 - Data partitioning:
 - This splits the input or intermediate data into smaller pieces. Similarly, upon identification of parallelism in the input data, it can also be divided into pieces to be processed on different workers. Data pieces may be processed by different parts of a program or a copy of the same program.

- **Mapping:**

- This assigns the either smaller parts of a program or the smaller pieces of data to underlying resources.
- This process aims to appropriately assign such parts or pieces to be run simultaneously on different workers and is usually handled by resource allocators in the system.

- **Synchronization:**

- Because different workers may perform different tasks, synchronization and coordination among workers is necessary so that race conditions are prevented and data dependency among different workers is properly managed.
- Multiple accesses to a shared resource by different workers may raise race conditions, whereas data dependency happens when a worker needs the processed data of other workers.

- **Communication:**

- Because data dependency is one of the main reasons for communication among workers, communication is always triggered when the intermediate data is sent to workers.

- **Scheduling:**

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

MapReduce Framework for Parallel processing

What is MapReduce?

- MapReduce is a software framework which supports parallel and distributed computing on large data sets .
- Figure 6.1 illustrates the logical data flow from the Map to the Reduce function in MapReduce frameworks
- The abstraction layer provides two well-defined interfaces in the form of two functions: Map and Reduce
- These two main functions can be overridden by the user to achieve specific objectives.

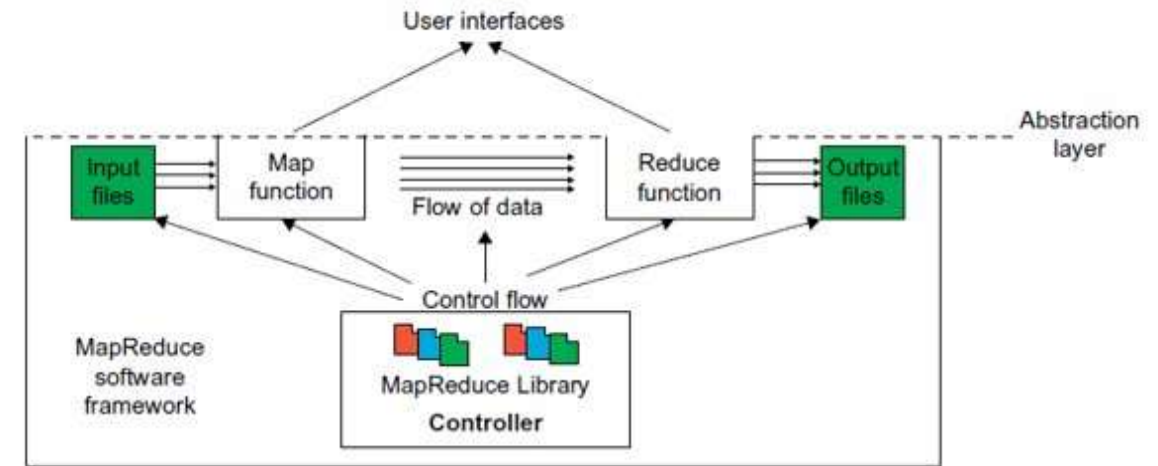


FIGURE 6.1

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.

MapReduce Logical Data Flow

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

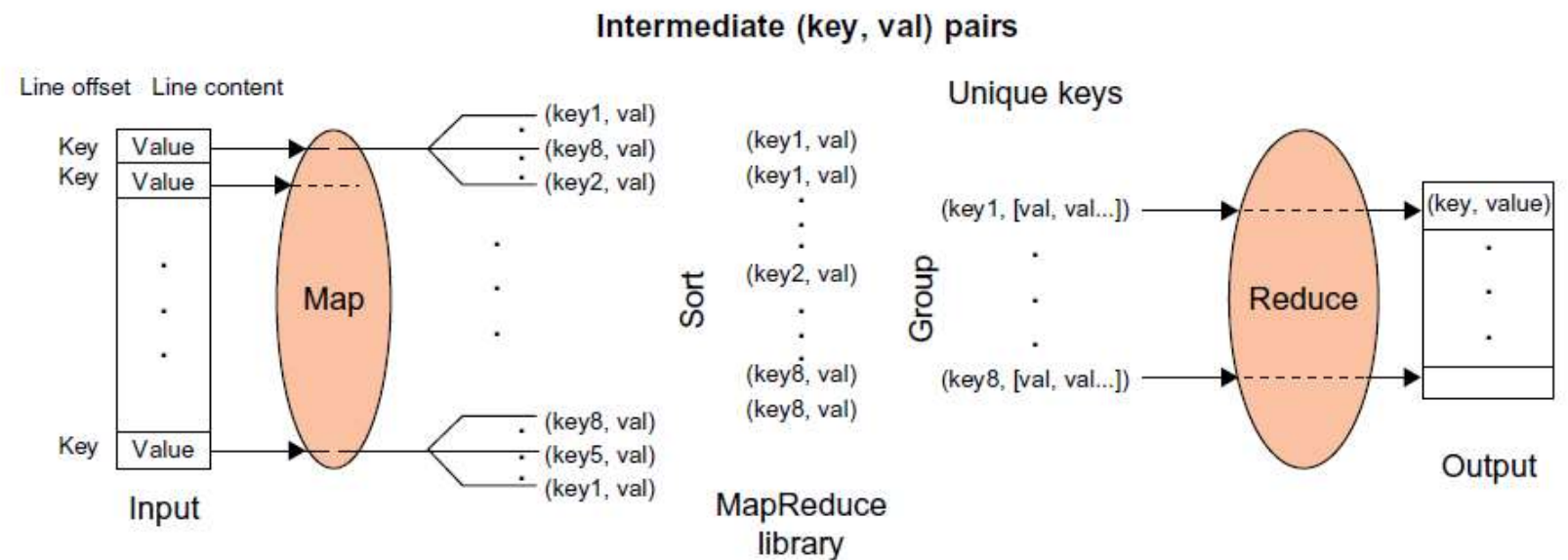


FIGURE 6.2

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

MapReduce Logical Data Flow

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

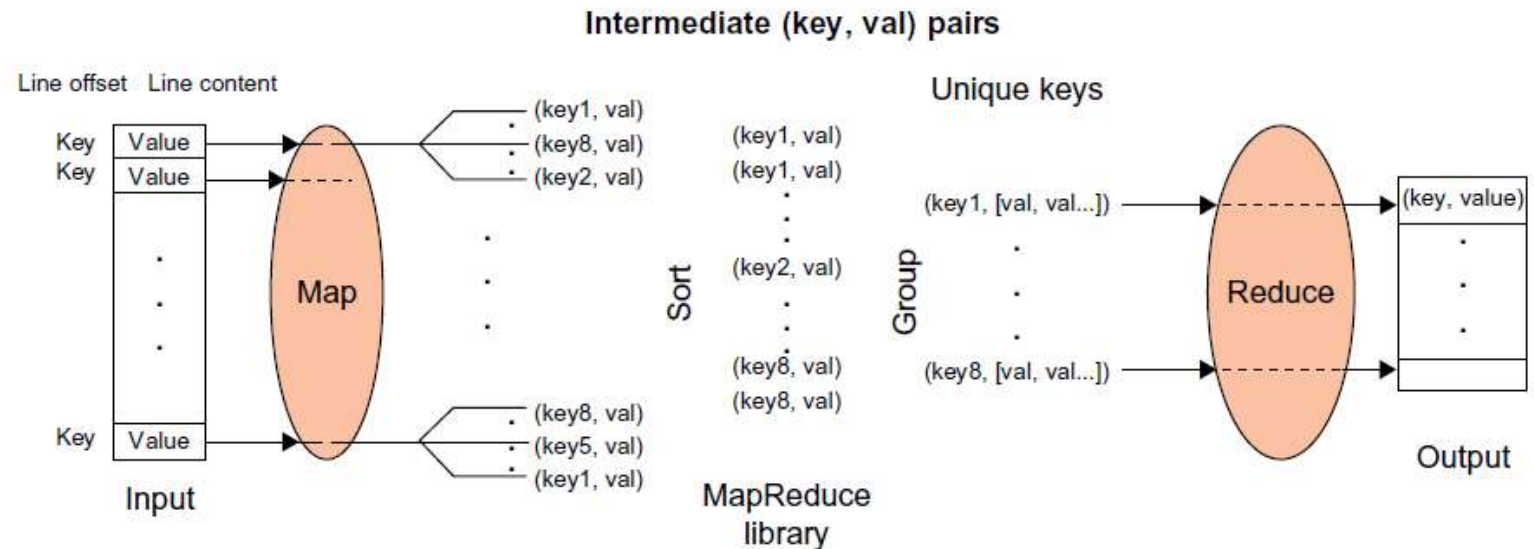


FIGURE 6.2

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

Structure of user program containing Map, Reduce functions

- The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below.
- The Map and Reduce are two major subroutines.
- They will be called to implement the desired function performed in the main program.
- Therefore, the user overrides the Map and Reduce functions first and then invokes the provided **MapReduce (Spec, & Results)** function from the library to start the flow of data.

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

Structure of user program containing Map, Reduce functions

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

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

A Word Counting Example on <Key, Count> Distribution

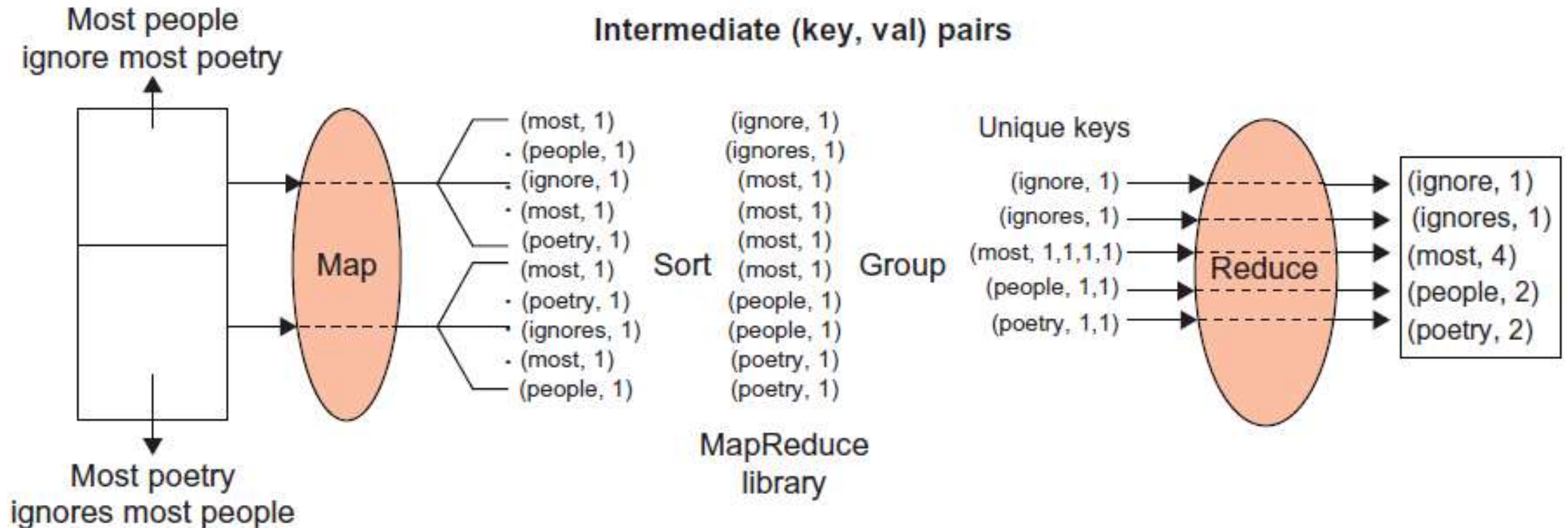


FIGURE 6.3

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

Steps in MapReduce: Actual Data and Control Flow

1. Data Partitioning

- The **MapReduce library splits the input data** stored in GFS into **M equal-size chunks**.
- Each chunk corresponds to **one map task**.

2. Computation Partitioning

- The library **creates copies** of the user program (Map + Reduce functions).
- These copies are **distributed across worker machines** and started.

3. Master–Worker Initialization

- The architecture uses a **master** and several **workers**.
- The **master assigns tasks** (map or reduce) to available idle workers.
- Each worker is a **compute node** that executes Map/Reduce functions.

4. Reading Input Data (Data Distribution)

- Each **map worker** reads the **input split assigned** to it from GFS.
- Normally, **one split per worker** is used for efficiency.

5. Map Function Execution

- The map worker passes the split as **(key, value)** pairs to the **Map function**.
- The Map function outputs **intermediate (key, value)** pairs.

6. Combiner Function (Optional)

- Acts as a **local mini-reduce** at the map worker.
- Used to **reduce data transferred over the network**.
- The MapReduce library **sorts and groups** these intermediate pairs locally before sending them.

7. Partitioning Function

- Goal: Ensure that **all identical keys go to the same reduce worker**.
- Each map worker **partitions its output into R regions**, where $R = \text{number of reduce tasks}$.
- Common partitioner:
 $\text{Partition} = \text{Hash}(\text{key}) \bmod R$
- Locations of these partitions are **sent to the master**, so reduce workers know where to retrieve their data.

8. Synchronization

- After partitioning, **all map workers must finish** before reduce workers can begin data transfer.
- This is a **global barrier**:
 - Map → Shuffle → Reduce can only begin **after all M map tasks complete**.
- This synchronization ensures that reduce workers receive **complete intermediate data** from all map tasks.

9. Communication (Shuffle Phase)

- Each reduce worker **knows the locations** of its corresponding partition (“region i ”) from every map worker.
- Reduce worker i uses **remote procedure calls (RPCs)** to fetch region i from **all** map workers.
- This results in:
 - **All-to-all communication**
(every reduce worker communicates with every map worker)
 - Heavy network traffic
 - **Shuffle bottleneck**, which is one of the largest performance limitations in MapReduce systems.
- Research improvements:
 - A **data transfer scheduler** can reduce congestion by planning transfers in a more controlled manner.

10. Sorting and Grouping (Reduce Worker Side)

- After fetching, each reduce worker stores the data in its **local disk buffer**.
- The reduce worker then:
 - **Sorts** all intermediate (key, value) pairs by key
 - **Groups** values having the same key together
- Purpose:
 - Multiple keys may appear within the same partition region.
 - Sorting ensures that the Reduce function receives keys in a predictable, grouped order.
- Output of this step:
{ key1: [v1, v2, ...], key2: [v5, v6, ...], ... }

11. Reduce Function Execution

- For each **unique grouped key**, the reduce worker calls the **Reduce function**.
- The Reduce function:
 - Processes the list of values for that key
 - Aggregates, counts, merges, or computes depending on user logic
- Final output:
 - Written to **output files**, often one file per reduce task (R files total)
 - These files become the final results of the MapReduce job

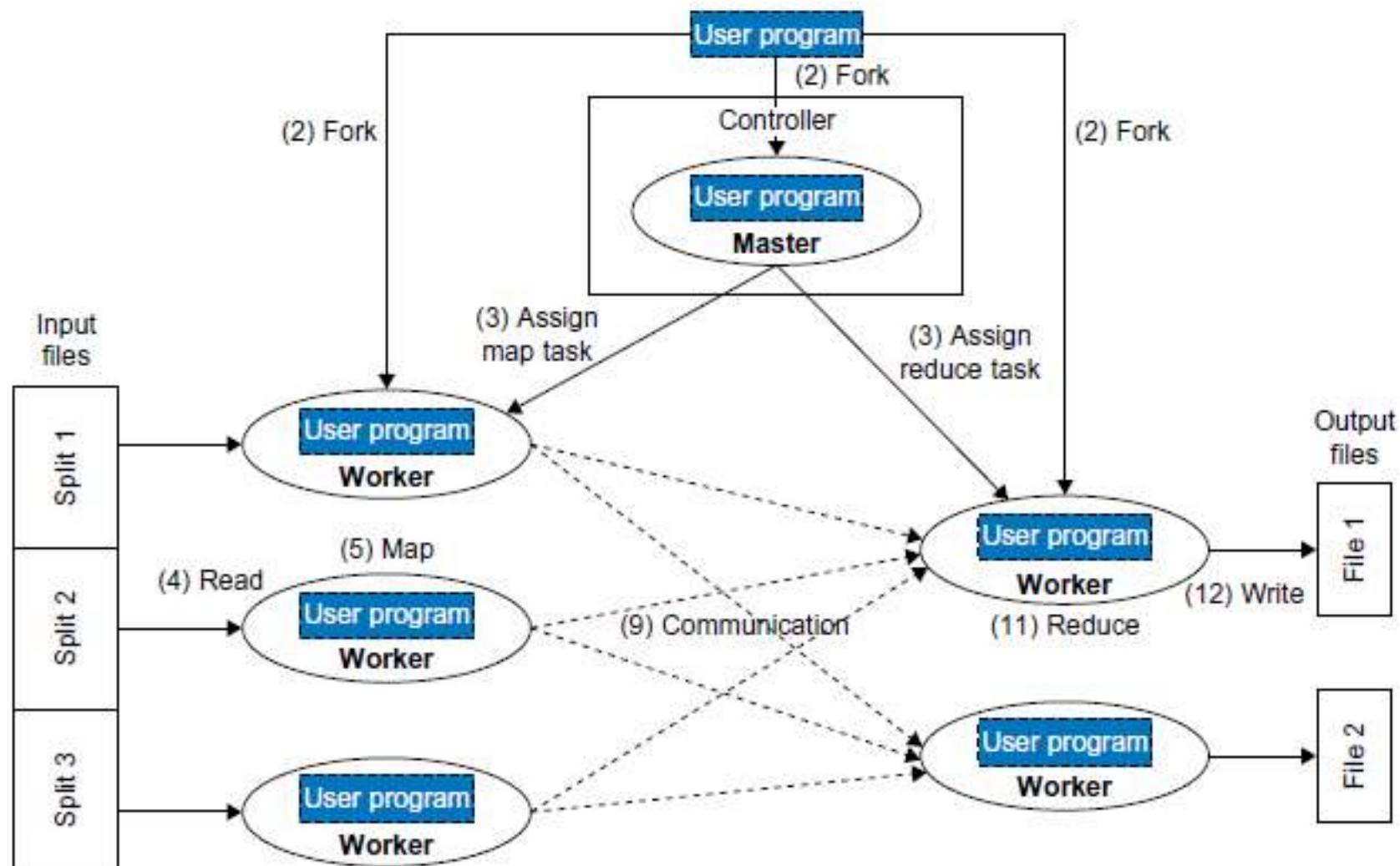


FIGURE 6.6

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.

(Courtesy of Yahoo! Pig Tutorial [54])

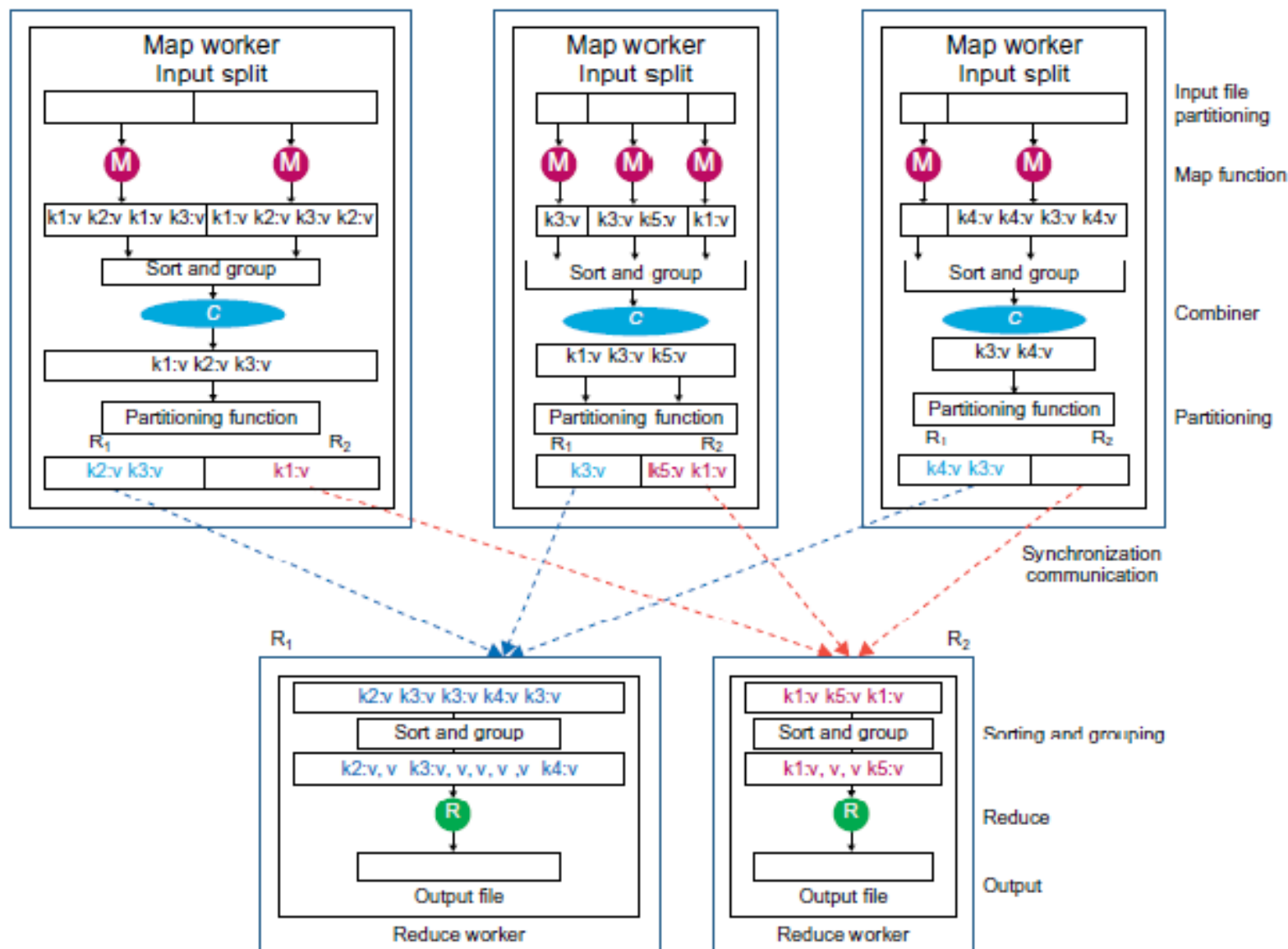



FIGURE 6.5

Data flow implementation of many functions in the Map workers and in the Reduce workers through multiple sequences of partitioning, combining, synchronization and communication, sorting and grouping, and reduce operations.



MapReduce Applications

- In research:
 - Astronomical image analysis (Washington)
 - Bioinformatics (Maryland)
 - Analyzing Wikipedia conflicts (PARC)
 - Natural language processing (CMU)
 - Particle physics (Nebraska)
 - Ocean climate simulation (Washington)
 - <Your application here>
- 

Google File System (GFS)

underlying storage system

Introduction

- Google made some special decisions regarding the design of GFS.
- A 64 MB block size was chosen.
- Reliability is achieved by using replications (i.e., each chunk or data block of a file is replicated across more than three chunk servers).
- A single master coordinates access as well as keeps the metadata.
- This decision simplified the design and management of the whole cluster.
- Developers do not need to consider many difficult issues in distributed systems, such as distributed consensus.
- There is no data cache in GFS as large streaming reads and writes represent neither time nor space locality.
- GFS provides a similar, but not identical, POSIX file system accessing interface.
- The distinct difference is that the application can even see the physical location of file blocks.
- Such a scheme can improve the upper-layer applications.
- The customized API can simplify the problem and focus on Google applications.

GFS Architecture

Overall Structure

- The GFS cluster is organized around a **single master node** that manages system-wide metadata.
- All other nodes operate as **chunk servers**, responsible for storing actual file data in fixed-size chunks.

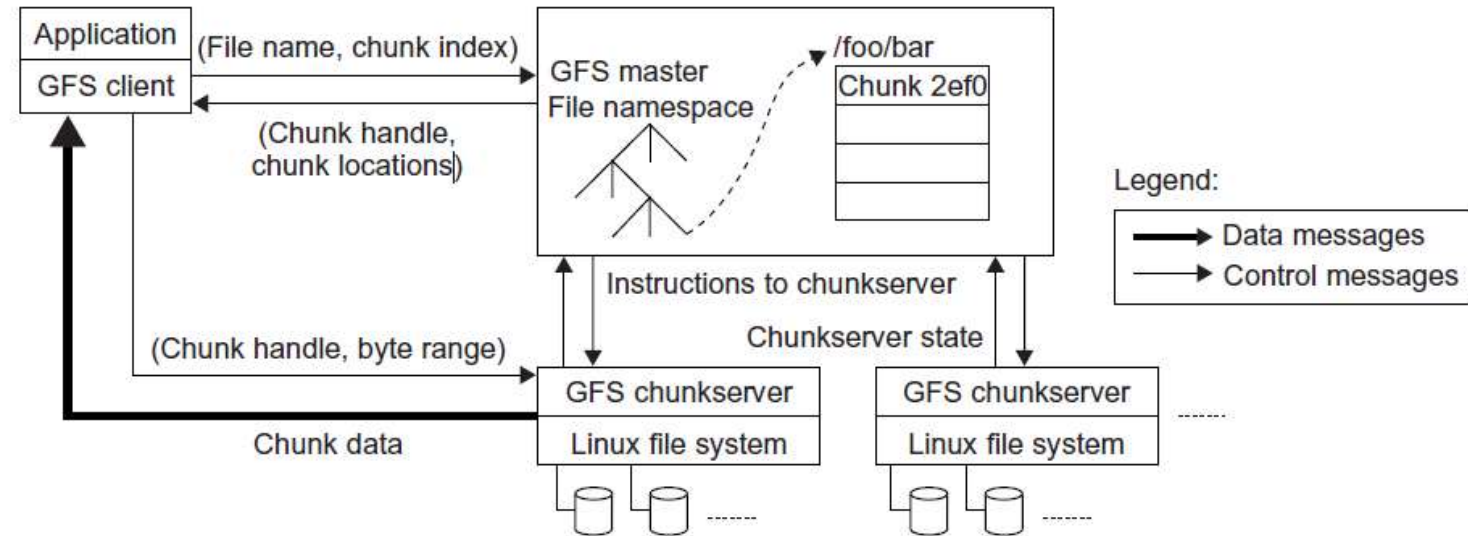


FIGURE 6.18

Architecture of Google File System (GFS).

- **Role of the Master**

- Maintains the **file system namespace**, directory hierarchy, and access-control information.
- Manages **metadata**, including:
 - File-to-chunk mappings
 - Chunk version numbers
 - Chunk replication locations
- Provides **locking services** to ensure consistency during concurrent operations.
- Periodically communicates with chunk servers to:
 - Collect system status and reports
 - Issue instructions for **load balancing**, **re-replication**, and **failure recovery**

- **Data Flow and Control Flow**

- The master handles **only control operations**, not data transfer.
- All file read/write operations occur **directly between clients and chunk servers**, reducing master load.
- Clients cache metadata obtained from the master to minimize repeated requests.

- **Reliability and Availability**

- A **shadow master** maintains a replica of the master's state to mitigate single-point-of-failure concerns.
- Shadow master supports:
 - Faster failover
 - Improved availability
 - Continual replication of master state

- **Scalability**

- Despite being a single master design, efficient control-only communication allows it to handle:
 - **1,000+ chunk servers**
 - **Tens of thousands of clients**

- **Performance Consideration**

- The master can become a potential performance bottleneck and single point of failure.
- Google's design mitigates this through:
 - Shadow master replication
 - Client–chunk server direct data paths
 - Aggressive metadata caching on clients

Data mutation in GFS

1. Client checks lease

1. Requests the master for the chunk's primary replica (lease holder) and secondary replicas.

2. Master responds

1. Provides the identity of the primary and all secondaries.
2. Client caches this metadata for future writes.

3. Client pushes data

1. Sends data to **all replicas** (any order).
2. Replicas temporarily store data in an internal LRU buffer cache.
3. Data flow is **decoupled from control flow** → improves performance.

4. Client sends write request to primary

1. After all replicas acknowledge receiving data.
2. Primary assigns **serial numbers** to mutations to establish ordering.
3. Applies mutation to its own local chunk.

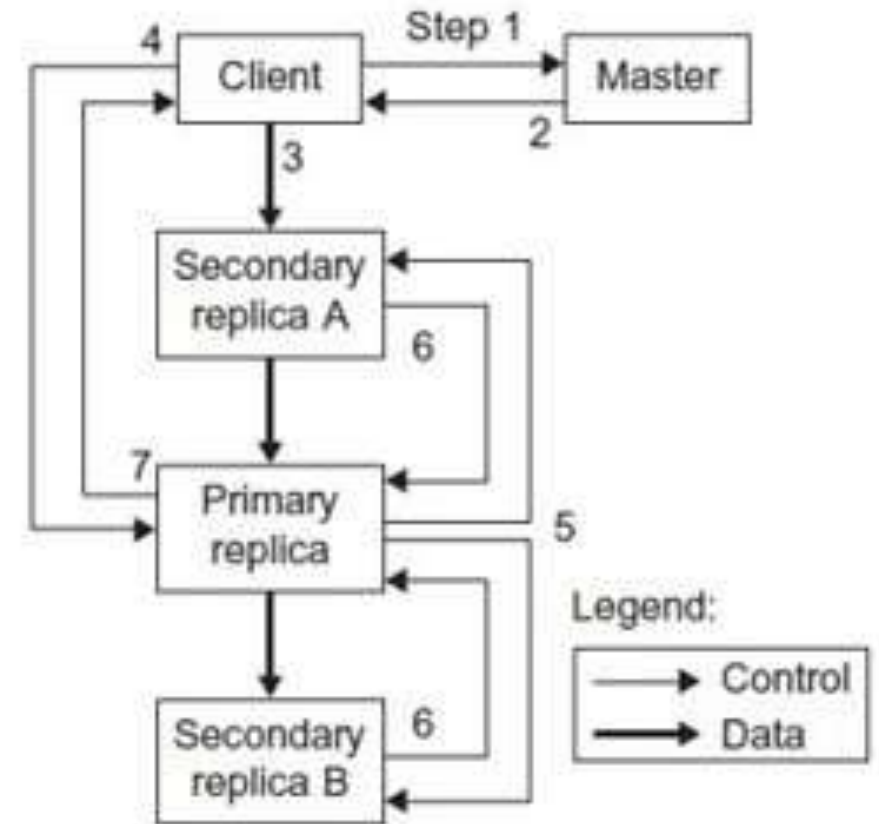


FIGURE 6.19

Data mutation sequence in GFS.

5. Primary forwards write to secondaries

1. Sends the ordered write request to all secondary replicas.
2. Ensures all replicas apply mutations in **identical serial order**.

6. Secondaries acknowledge

1. Secondary replicas respond to the primary after applying the mutation.

7. Primary responds to client

1. Success or error is reported.
2. If any replica fails, the write is considered **failed**.
3. Inconsistent chunk regions may occur temporarily.
4. Client retries the mutation (steps 3–7, or full restart if needed)

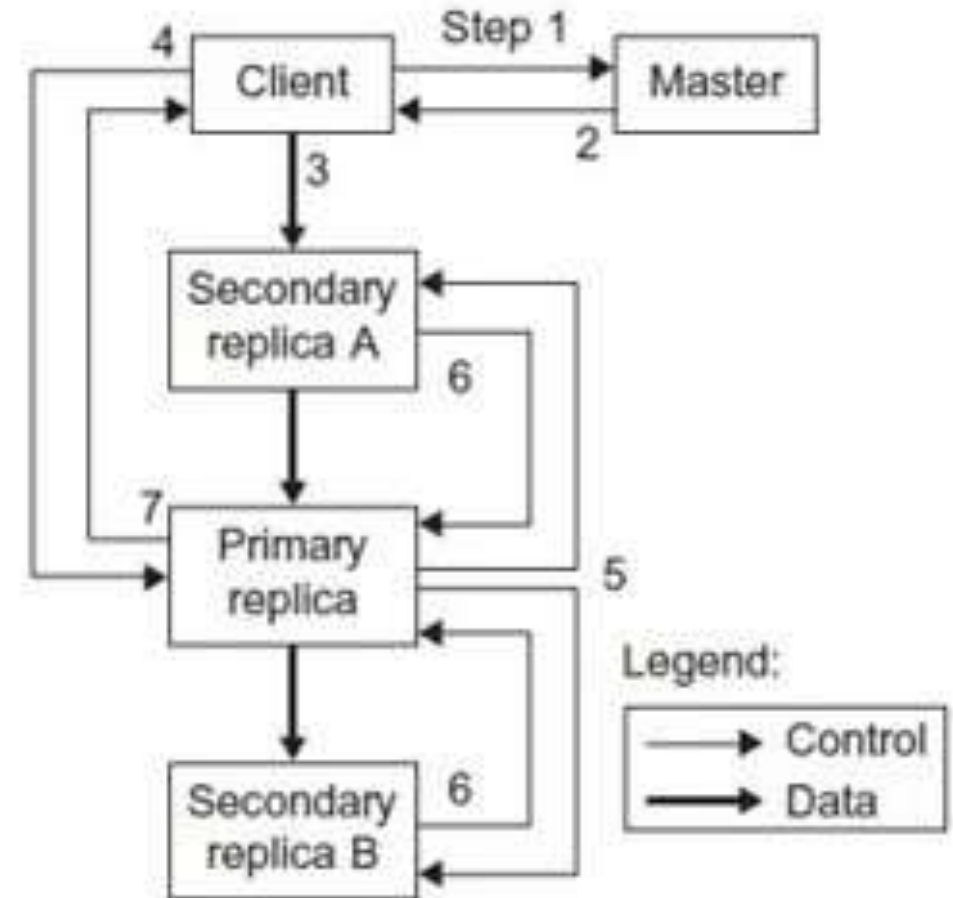


FIGURE 6.19

Data mutation sequence in GFS.

Hadoop MapReduce and HDFS

Introduction

- Hadoop is an open source implementation of MapReduce coded and released in Java (rather than C) by Apache.
- The Hadoop implementation of MapReduce uses the Hadoop Distributed File System (HDFS) as its underlying layer rather than GFS.
- The Hadoop core is divided into two fundamental layers: the MapReduce engine and HDFS.
- The MapReduce engine is the computation engine running on top of HDFS as its data storage manager.

HDFS (Hadoop Distributed File System)

- HDFS is a distributed file system inspired by GFS that organizes files and stores their data on a distributed computing system.
- **HDFS Architecture:**
 - HDFS has a master/slave architecture containing a single NameNode as the master and a number of DataNodes as workers (slaves).
 - To store a file in this architecture, HDFS splits the file into fixed-size blocks (e.g., 64 MB) and stores them on workers (DataNodes).
 - The mapping of blocks to DataNodes is determined by the NameNode.
 - The NameNode (master) also manages the file system's metadata and namespace.
 - In such systems, the namespace is the area maintaining the metadata, and metadata refers to all the information stored by a file system that is needed for overall management of all files.
 - For example, NameNode in the metadata stores all information regarding the location of input splits/blocks in all DataNodes.
 - Each DataNode, usually one per node in a cluster, manages the storage attached to the node.
 - Each DataNode is responsible for storing and retrieving its file blocks.

- **HDFS Fault Tolerance:** Since Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception. Therefore, Hadoop considers the following issues to fulfill reliability requirements of the file system:
 - Block replication: To reliably store data in HDFS, file blocks are replicated and distributed across the whole cluster. The replication factor is set by the user and is three by default.
 - Replica placement: For the default replication factor of three, HDFS stores one replica in the same node the original data is stored, one replica on a different node but in the same rack, and one replica on a different node in a different rack to provide three copies of the data.
 - Heartbeat and Blockreport messages: Heartbeats and Blockreports are periodic messages sent to the NameNode by each DataNode in a cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly, while each Blockreport contains a list of all blocks on a DataNode. The NameNode receives such messages because it is the sole decision maker of all replicas in the system.

HDFS Operation:The control flow of HDFS operations such as write and read can properly highlight roles of the NameNode and DataNodes in the managing operations.

- **Reading a file:**

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

- **Writing to a file:**

- To write a file in HDFS, a user sends a “create” request to the NameNode to create a new file in the file system namespace.
- If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the write function.
- The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode.
- Since each file block needs to be replicated by a predefined factor, the data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.
- The steamer then stores the block in the first allocated DataNode.
- Afterward, the block is forwarded to the second DataNode by the first DataNode.
- The process continues until all allocated DataNodes receive a replica of the first block from the previous DataNode.
- Once this replication process is finalized, the same process starts for the second block and continues until all blocks of the file are stored and replicated on the file system.

Architecture of MapReduce in Hadoop

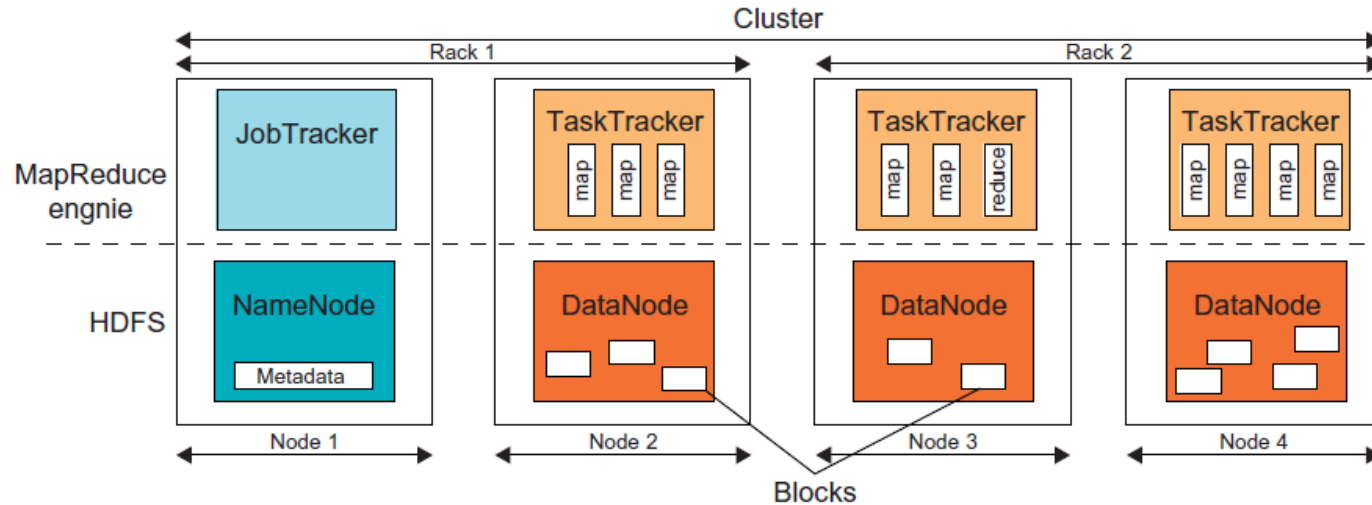


FIGURE 6.11

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

- The topmost layer of Hadoop is the MapReduce engine that manages the data flow and control flow of MapReduce jobs over distributed computing systems.
- Similar to HDFS, the MapReduce engine also has a master/slave architecture consisting of a single JobTracker as the master and a number of TaskTrackers as the slaves (workers).
- The JobTracker manages the MapReduce job over a cluster and is responsible for monitoring jobs and assigning tasks to TaskTrackers.
- The TaskTracker manages the execution of the map and/or reduce tasks on a single computation node in the cluster.

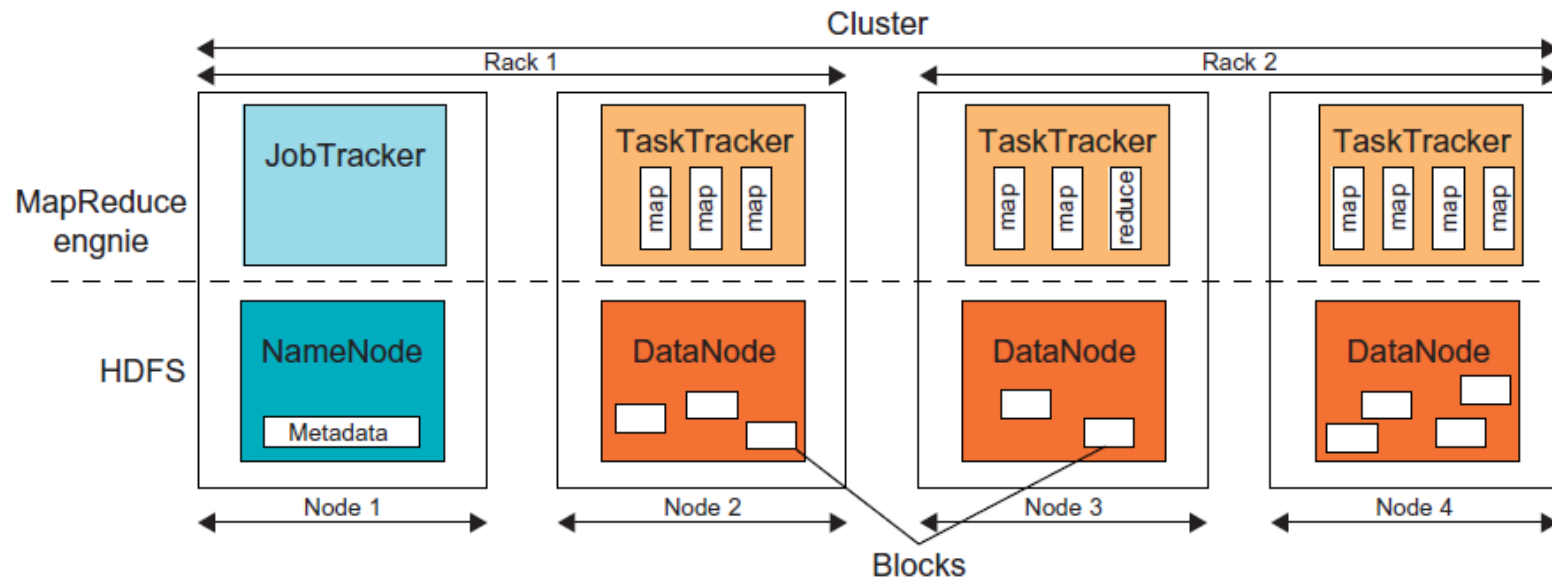


FIGURE 6.11

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

- Each TaskTracker node has a number of simultaneous execution slots, each executing either a map or a reduce task.
- Slots are defined as the number of simultaneous threads supported by CPUs of the TaskTracker node.
- For example, a TaskTracker node with N CPUs, each supporting M threads, has $M * N$ simultaneous execution slots.
- It is worth noting that each data block is processed by one map task running on a single slot.
- Therefore, there is a one-to-one correspondence between map tasks in a TaskTracker and data blocks in the respective DataNode.

Running a Job in Hadoop

- Three components contribute in running a job in Hadoop system: a user node, a JobTracker, and several TaskTrackers.
- **Job Submission:**
 - Each job is submitted from a user node to the JobTracker node that might be situated in a different node within the cluster through the following procedure:
 - A user node asks for a new job ID from the JobTracker and computes input file splits.
 - The user node copies some resources, such as the job's JAR file, configuration file, and computed input splits, to the JobTracker's file system.
 - The user node submits the job to the JobTracker by calling the `submitJob()` function.

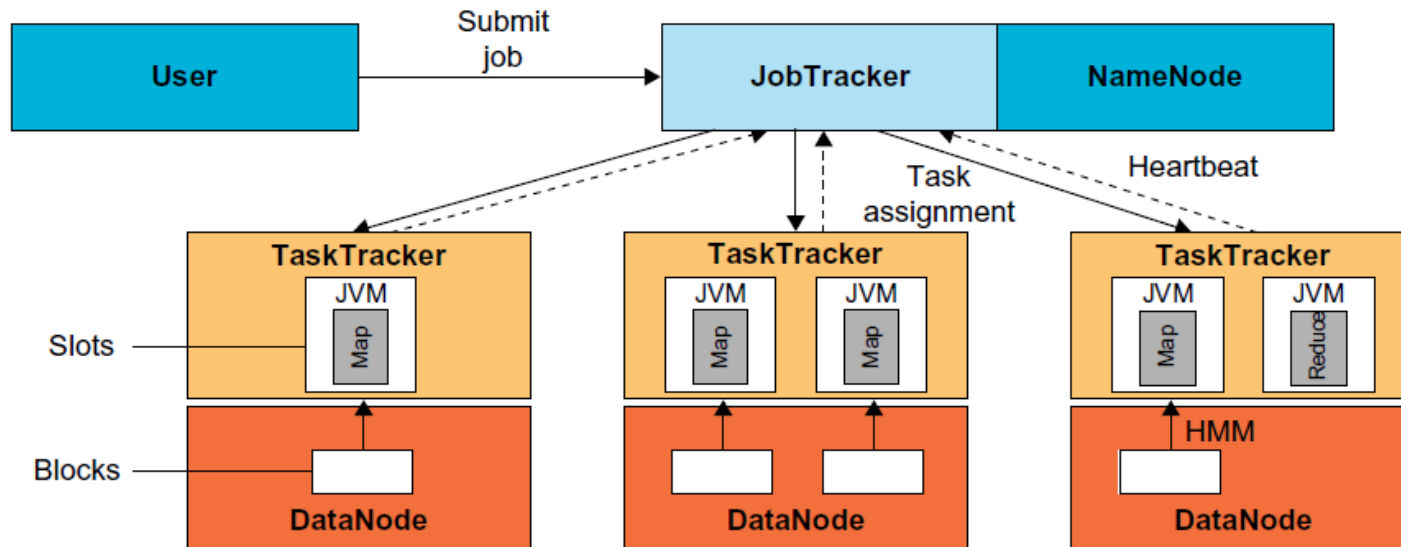


FIGURE 6.12

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

Contd...

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

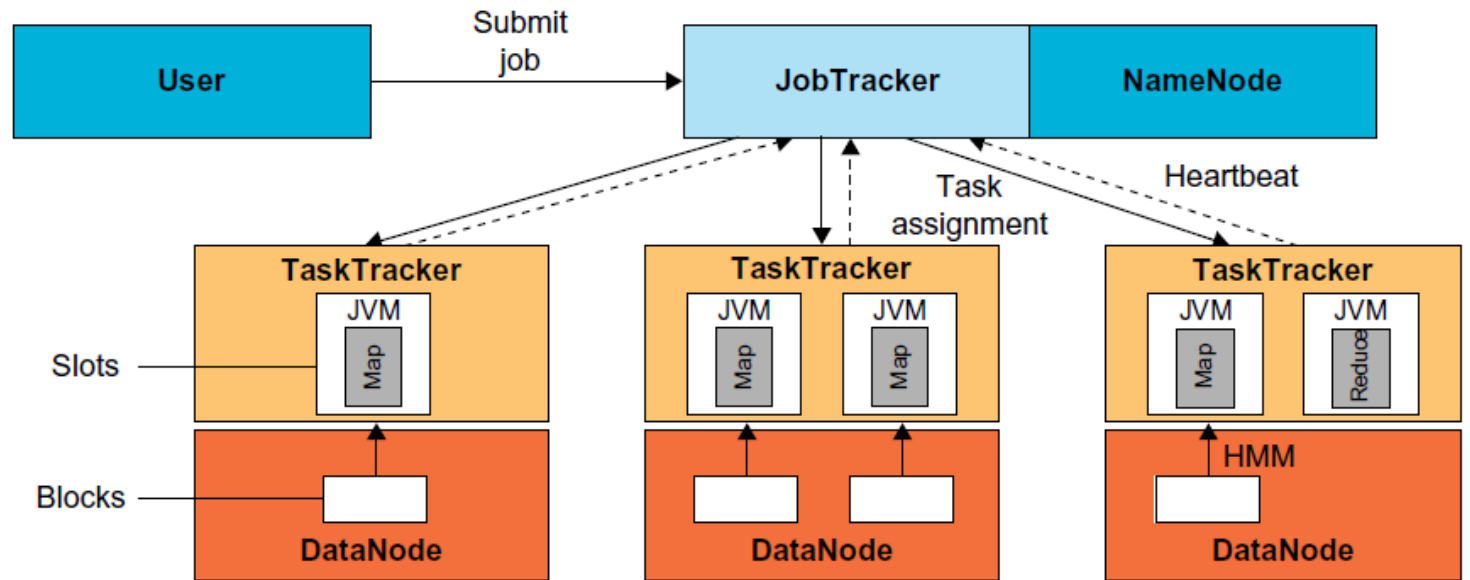


FIGURE 6.12

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

Contd..

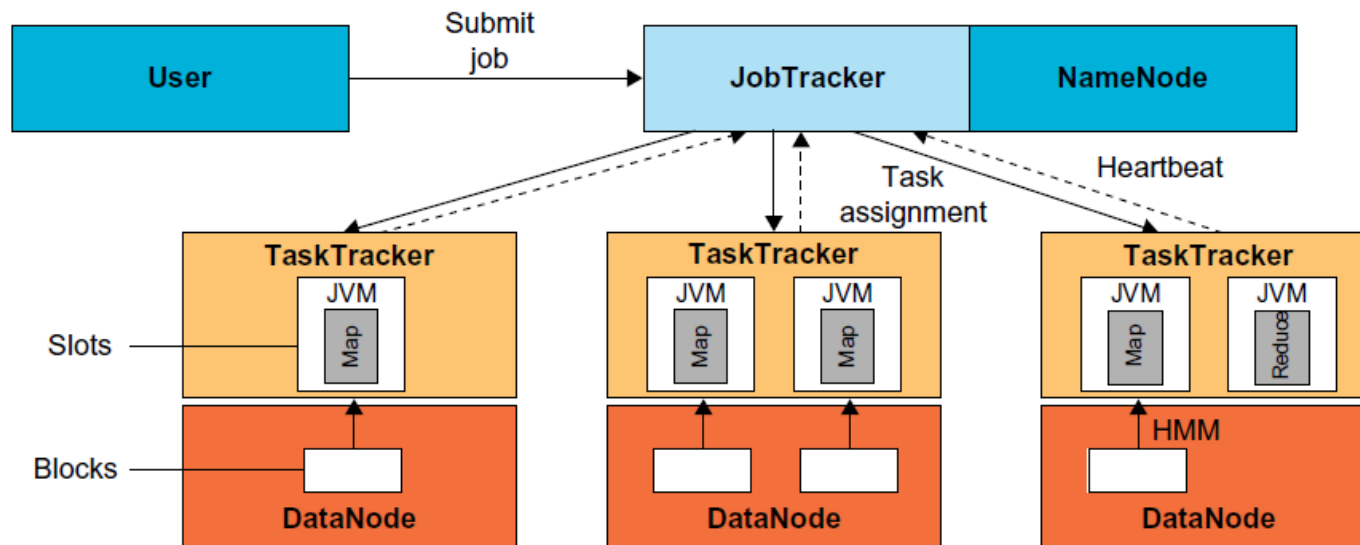


FIGURE 6.12

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

- **Task execution:**

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

- **Task running check:**

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

Big Table

Google's NOSQL System

Introduction

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

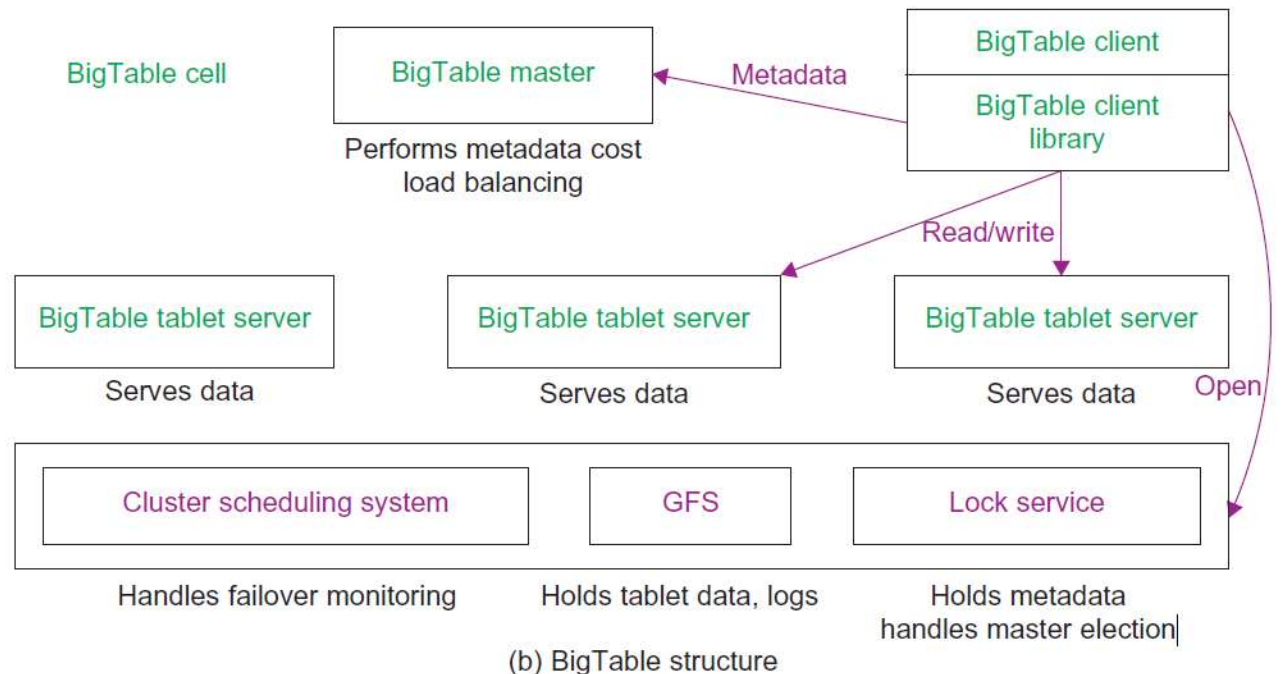
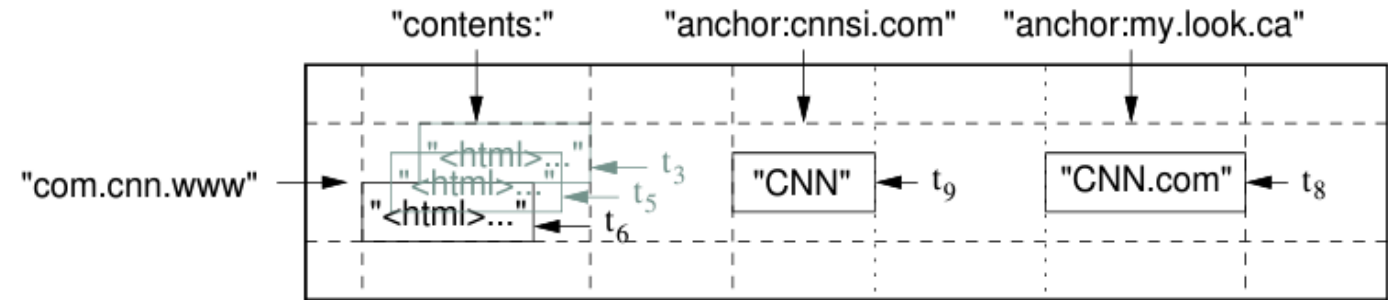
Goals in design and implementation of BigTable

- The applications want asynchronous processes to be continuously updating different pieces of data and want access to the most current data at all times.
- The database needs to support very high read/write rates and the scale might be millions of operations per second.
- Also, the database needs to support efficient scans over all or interesting subsets of data, as well as efficient joins of large one-to-one and one-to-many data sets.
- The application may need to examine data changes over time (e.g., contents of a web page over multiple crawls).

- The BigTable system is built on top of an existing Google cloud infrastructure.
- BigTable uses the following building blocks:
 - GFS: stores persistent state
 - Scheduler: schedules jobs involved in BigTable serving
 - Lock service: master election, location bootstrapping
 - MapReduce: often used to read/write BigTable data

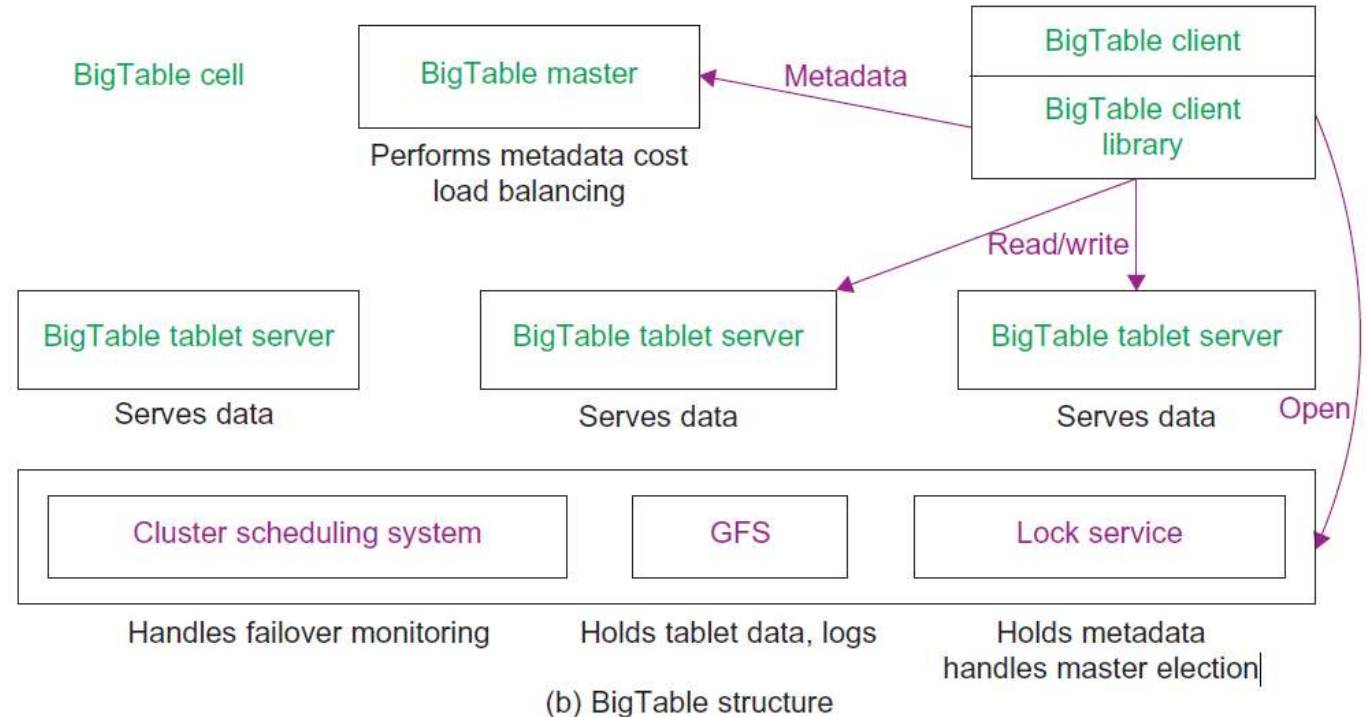
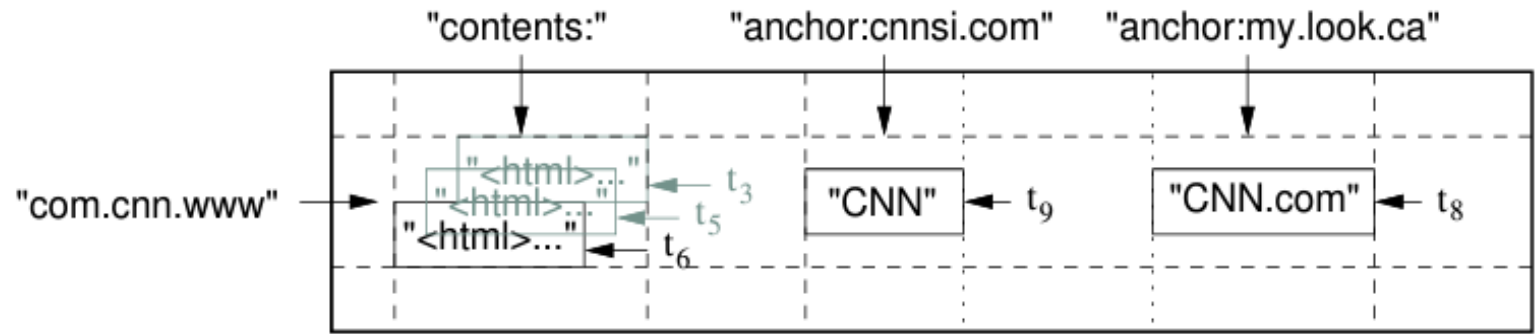
- BigTable provides a simplified data model compared to traditional database systems. Figure shows the data model of a sample table, Web Table. Web Table stores the data about a web page.
- Each web page can be accessed by the URL. The URL is considered the row index. The column provides different data related to the corresponding URL—for example, different versions of the contents, and the anchors appearing in the web page. In this sense, BigTable is a distributed multidimensional stored sparse map.
- The map is indexed by row key, column key, and timestamp—that is, (row: string, column: string, time: int64) maps to string (cell contents).

Big Table Data model



- Rows are ordered in lexicographic order by row key. The row range for a table is dynamically partitioned and each row range is called “Tablet.” Syntax for columns is shown as a (family:qualifier) pair.
- Cells can store multiple versions of data with timestamps.
- For rows, Name is an arbitrary string and access to data in a row is atomic. Row creation is implicit upon storing data. Rows are ordered lexicographically, that is, close together lexicographically, usually on one or a small number of machines.

Big Table Data model



OpenStack

OpenStack

- **OpenStack Overview**

- Introduced by **Rackspace and NASA (2010)**
- Open-source cloud platform with a large global developer community
- Designed to create **massively scalable, secure cloud infrastructure**
- Uses **open APIs** similar to Amazon Web Services (AWS)
- Focus areas: **Compute (Nova)** and **Storage (Swift)**
- Additional services evolving: **Image repository (Glance)**

OpenStack Compute (Nova)

- **Purpose**

- Acts as the **fabric controller** for IaaS
- Manages **VM lifecycle**, scheduling, networking, and authentication

- **Architecture Characteristics**

- **Shared-nothing architecture**
- **Messaging-based communication** using message queues
- Uses **deferred objects (callbacks)** for non-blocking operations
- State information kept in a **distributed data system**
- Implemented in **Python** using external libraries:
 - **boto** (Amazon API in Python)
 - **Tornado** (HTTP server for S3 support)

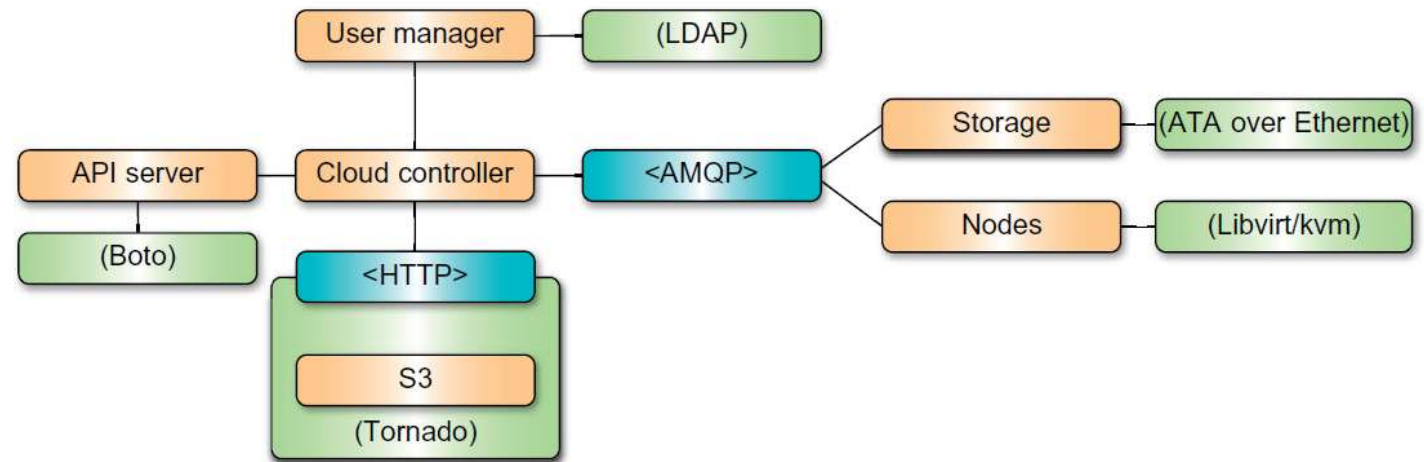


FIGURE 6.30

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

- **Key Components**

- **1. API Server**

- Receives HTTP requests
 - Converts commands to internal API format
 - Forwards requests to the cloud controller

- **2. Cloud Controller**

- Maintains global system state
 - Handles authorization (via **LDAP**)
 - Interfaces with S3-based storage
 - Manages compute nodes and storage workers via queues

- **Networking Components**

- **NetworkController**

- Manages IP addresses and VLAN allocations

- **RoutingNode**

- NAT conversion (public ↔ private IP)
 - Enforces firewall rules

- **AddressingNode**

- Provides DHCP services to private networks

- **TunnelingNode**

- Manages VPN connectivity

- **Network State (stored in distributed object store)**
 - VLAN assignments to projects
 - Private subnet assignments
 - Private IP allocations
 - Public IP allocations
 - Mapping of public \leftrightarrow private IPs for running instances

OpenStack Storage (Swift)

- **Core Idea**
 - Provides **redundant, scalable object storage**
 - Built on commodity servers
 - Designed for **terabytes to petabytes** of storage
- **Major Components**
 - 1. Proxy Server**
 - Entry point for all storage requests
 - Performs **lookups** in the storage rings
 - Routes requests to correct object/container/account servers
 - 2. Rings**
 - Map logical entity names to physical storage locations
 - Separate rings for:
 - Accounts
 - Containers
 - Objects
 - Support:
 - **Zones** (rack / server / data center)
 - **Replication**
 - **Partitioning**
 - **Weights** (balance storage load across devices)

3. Object Server

- Stores and retrieves object data
- Objects stored as **binary files**
- Metadata stored in **extended file attributes**

4. Container Server

- Lists objects in containers

5. Account Server

- Manages listing of containers under each account

Aneka

Aneka

- Aneka is a **cloud application platform** developed by *Manjrasoft (Australia)* for building and deploying distributed and parallel applications.
- Supports deployment on:
 - **Public clouds** (e.g., Amazon EC2)
 - **Private clouds** (on-premise clusters with restricted access)
- **Purpose & Capabilities**
 - Enables **rapid development and execution** of distributed applications.
 - Provides **rich APIs** to access distributed resources transparently.
 - Lets developers express application logic using their **preferred programming abstractions**.
 - Offers tools for system administrators to **monitor and control** cloud resources and applications.

- **Architecture & Platform Support**
 - Functions as a **workload distribution and management platform**.
 - Supports:
 - Microsoft .NET framework
 - Linux environments (via Mono)
 - Can run on **virtual or physical machines**, forming hybrid environments.

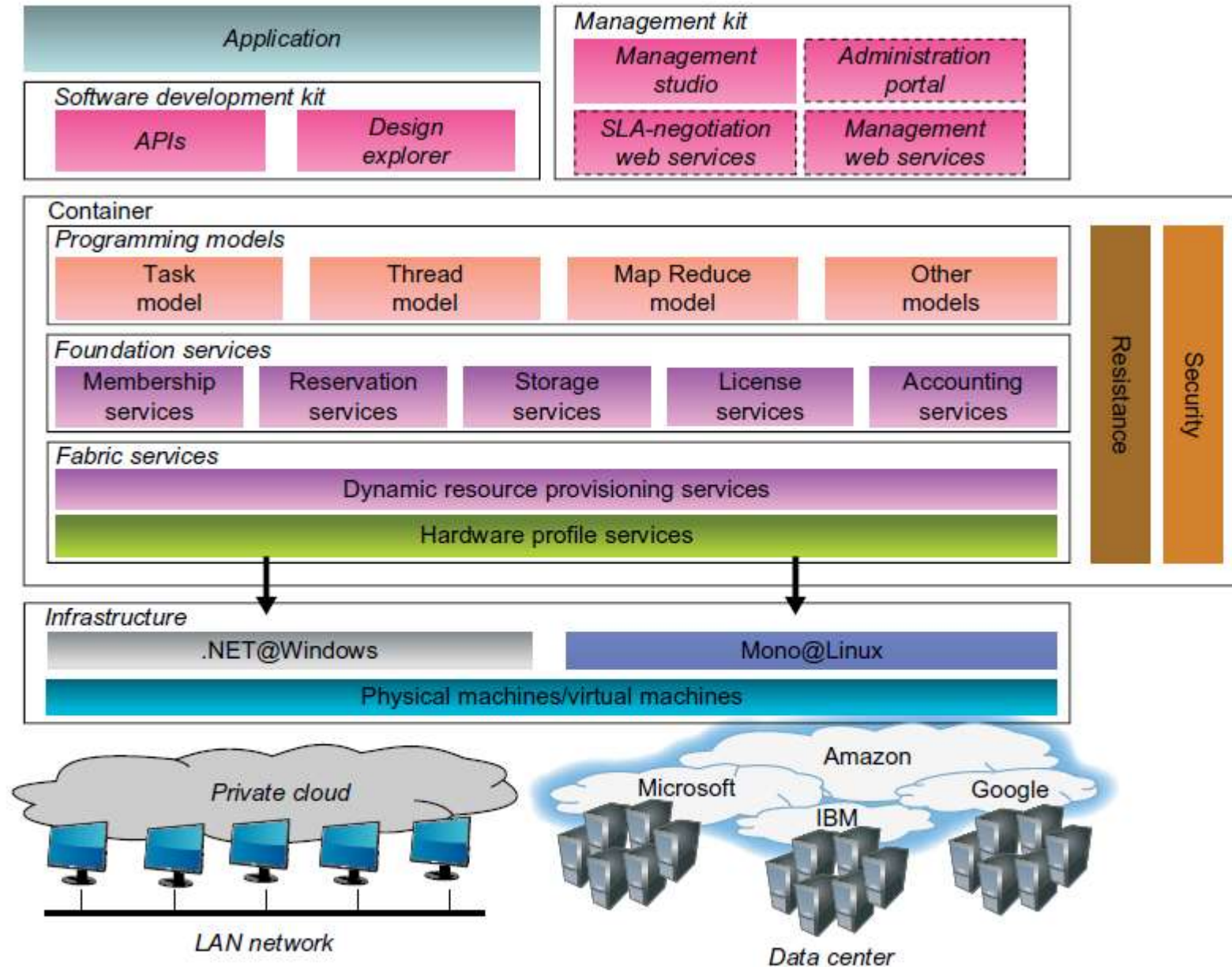


FIGURE 6.31

Architecture and components of Aneka.

- **Key Advantages of Aneka**

- **Multiple Programming Models**

- Supports diverse programming paradigms to build various types of applications.

- **Simultaneous Multi-Runtime Support**

- Can run different runtimes together to support heterogeneous applications.

- **Rapid Deployment Framework**

- Provides tools to easily set up and configure the cloud environment.

- **QoS/SLA-Aware Resource Provisioning**

- Allocates resources based on user QoS requirements and SLAs.

- **High Flexibility**

- Harnesses multiple virtual/physical machines to accelerate applications.