RAMAIAH
Institute of Technology

Cloud Computing and Big Data

Subject Code: CS71 (Credits: 4:0:0)

Textbook:

1. Cloud Computing Theory and Practice – **DAN C**. Marinescu – Morgan Kaufmann Elsevier.
2. Cloud Computing A hands - on approach – Arshdeep Bahga & **Vijay madisetti** Universities press
3. Big Data Analytics, Seema Acharya and Subhashini Chellappan. 2nd edition, Wiley India Pvt. Ltd. 2019

NOTE: I declare that the PPT content is picked up from the prescribed course text books or reference material prescribed in the syllabus book and Online Portals.

Unit IV

- Workflows coordination of multiple activities,
- Coordination based on a state machine model
- The Zoo Keeper,
- Apache Hadoop – Introduction, Architecture and Components,
- HDFS,
- The Map Reduce programming model, YARN,
- Interacting with Hadoop ecosystem – Pig, Hive, HBase, Sqoop,
- A case study: the GrepTheWeb application.

Workflows :Coordination of multiple activities

Many cloud applications require the completion of multiple independent tasks.

- The **description of a complex activity** involving such ensemble(together) of tasks is known as **workflow.**

  - workflow models
  - life cycle of a workflow
  - desirable properties of a workflow description
  - workflow pattern
  - Reachability of goal state of a workflow

Workflow models

- Workflow models are **abstractions revealing the most important properties of the entities participating in a workflow management system.**

- **Task** – central concept in workflow modeling: **task is a unit of work to be performed on the cloud,** and its **characterized by several attributes**:

  - **Name:** String of chars. Uniquely identifying tasks

  - **Description** : a natural language description of task

  - **Actions**: Modifications of the environment caused by the execution of the task

    - **Preconditions:** boolean expressions that must be **true before** the actions of the task can take place

    - **Post Conditions:** boolean expressions that must be **true after** the actions of the task can take place

  - **Attributes**: provide indication of type and quantity of resources necessary for the execution of the task: Actors in charge of the task, the security requirements, whether the task is reversible ,other task characteristics.

Task attributes and Types :

- **Exceptions: provide information on how to handle abnormal events.**
- **Anticipated exceptions:** exceptions included in the task exception list:< event,action>
- Events not included in the exception list trigger replanning.
- **Replanning: means restructuring of a process or redefinition of the relationship among various tasks.**

Composite task : it is a structure **describing a subset of tasks** and the **order of their execution**.

- **Primitive task:** cannot be decomposed into simpler task
- **Composite task Properties:** inherits from work flow- it consists of tasks, one start symbol and several end symbols.
- **Inherits from task: it has name, preconditions, post conditions**

Routing task: a special purpose task connecting two tasks in a workflow description.

- **Predecessor task:** the task that has **just completed execution**
- **Successor task:** the one **to be initiated next**
- routing task could trigger a sequential , concurrent or iterative execution.

Routing task Types

Fork routing task: **triggers execution of several successor tasks**. Several semantics for this constructs are possible:

•All successor tasks are enabled

•Each successor task is associated with a condition. The conditions for all tasks are evaluated * and only the tasks with a *true* condition are enabled.

*but the conditions are mutually exclusive and only one condition of many be true. Thus only one task is enabled.

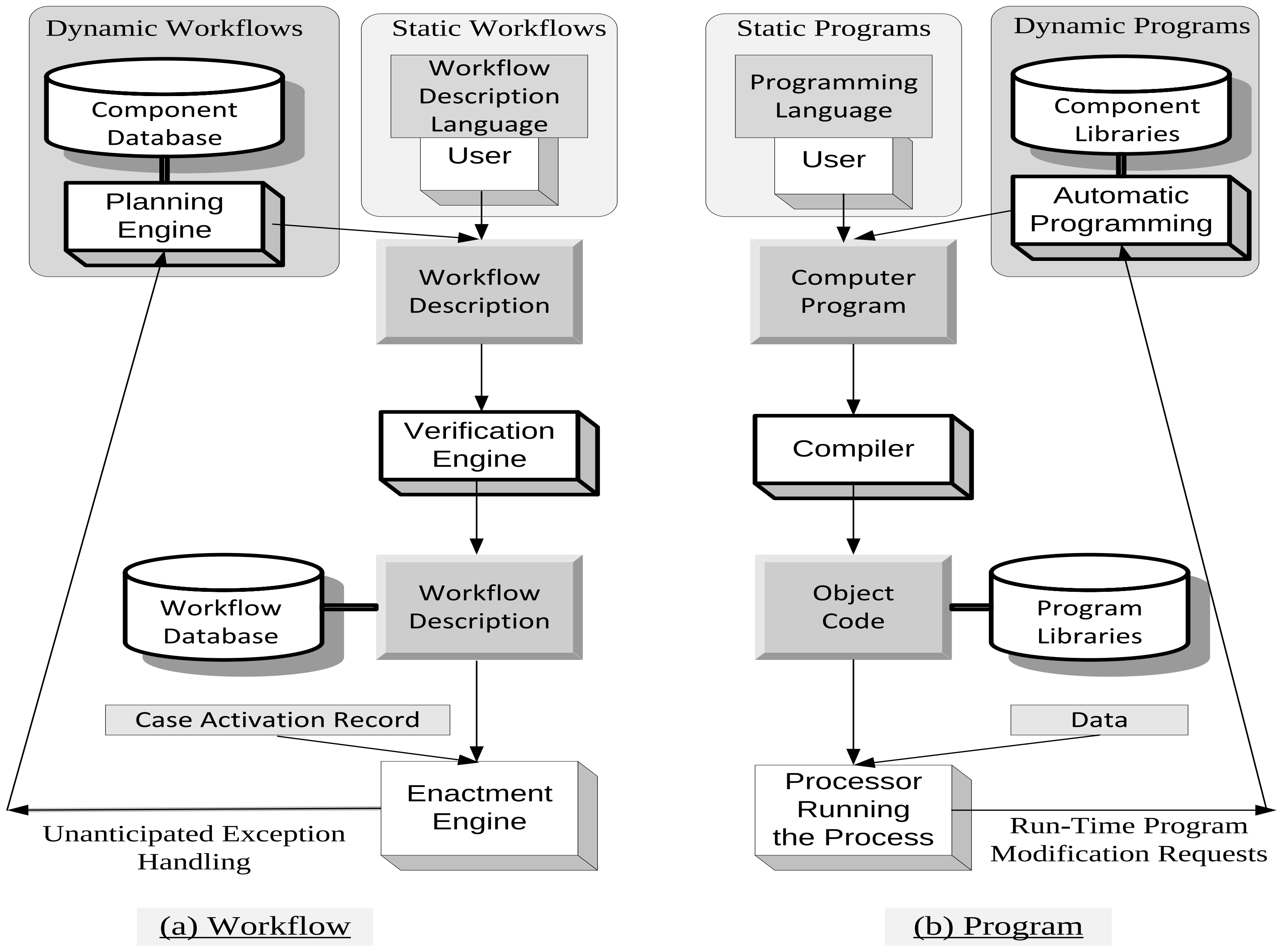
•NonDeterministic, k out of n>k successors at random are enabled.

Join routing task: waits for the completion of its predecessors tasks. Several semantics for this constructs are possible:

* The successor is enabled after all predecessors end.
* The successors is enabled after k out of n>k predecessors end.
* Iterative: The tasks between the fork and the join are executed repeatedly.

- Process description - **structure (workflow schema) describing the tasks or activities to be executed and the order of their execution.**
    - It has one start symbol and one end symbol. Resembles a flowchart.
    - A process description WFDL- workflow definition language

- **The life cycle of a workflow** - creation, definition, verification, and enactment; similar to the life cycle of a traditional program (**creation, compilation, and execution**).
    - Case - an instance of a process description.
    - State of a case at time t - defined in terms of tasks already completed at that time.
    - Events - cause transitions between states.


Dynamic Workflows
Component Database
Planning Engine
Static Workflows
Workflow Description Language
User
Workflow Description
Verification Engine
Workflow Database
Workflow Description
Case Activation Record
Enactment Engine
Unanticipated Exception Handling
(a) Workflow
Static Programs
Programming Language
User
Dynamic Programs
Component Libraries
Automatic Programming
Computer Program
Compiler
Object Code
Program Libraries
Data
Processor Running the Process
Run-Time Program Modification Requests
(b) Program

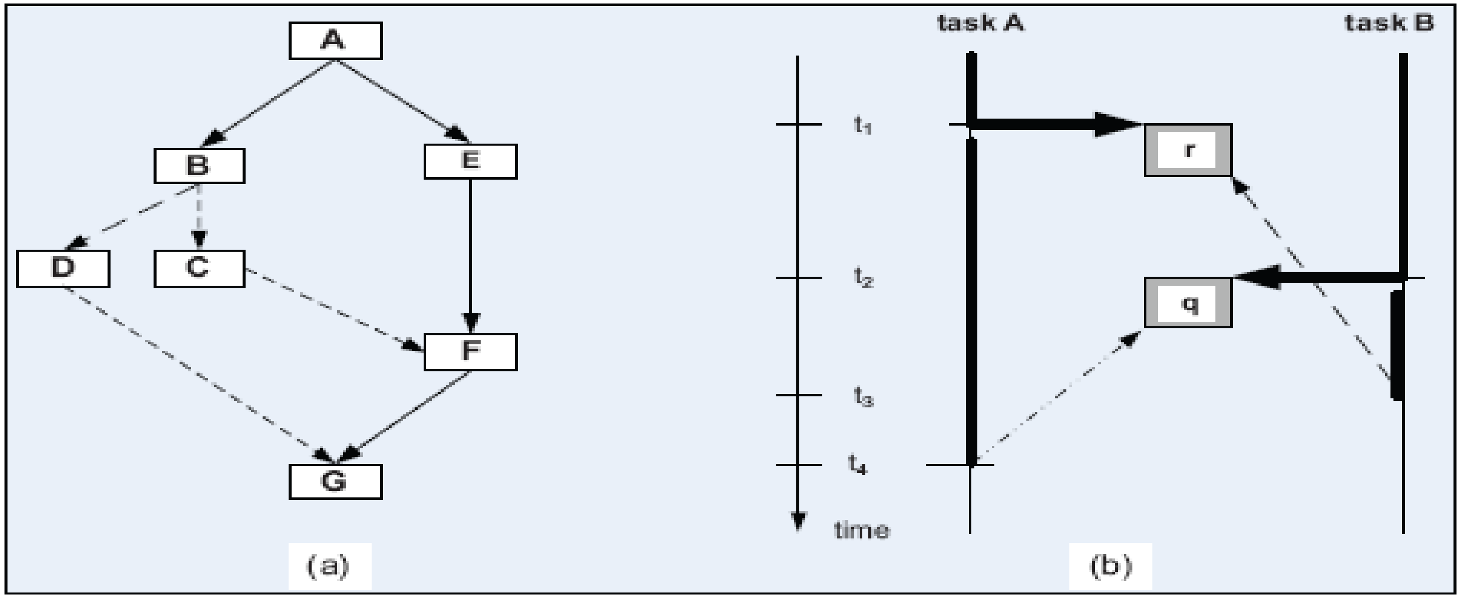


Figure 35: (a) A process description which violates the liveness requirement; if task C is chosen after completion of B, the process will terminate after executing task G; if D is chosen, then F will never be instantiated because it requires the completion of both C and E. The process will never terminate because G requires completion of both D and F. (b) Tasks A and B need exclusive access to two resources r and q and a deadlock may occur if the following sequence of events occur: at time t_1 task A acquires r, at time t_2 task B acquires q and continues to run; then, at time t_3, task B attempts to acquire r and it blocks because r is under the control of A; task A continues to run and at time t_4 attempts to acquire q and it blocks because q is under the control of B.

Basic workflow patterns:

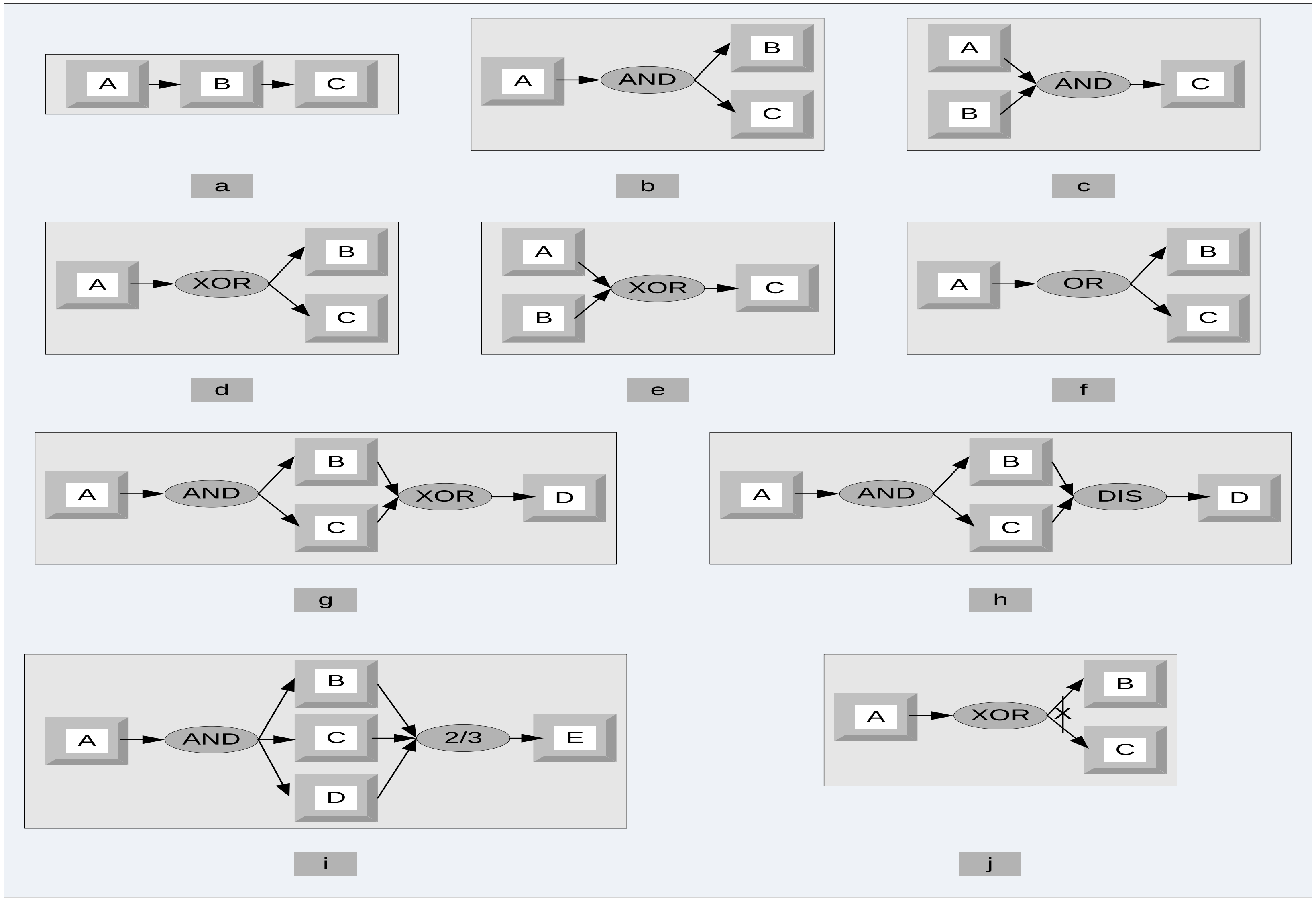
- **Sequence -** several tasks have to be **scheduled one after the completion of the other.**
- **AND split -** both **tasks B and C are activated when task A terminates**.
- **Synchronization -** task **C can only start after tasks A and B terminate**.
- **XOR split -** after **completion of task A, either B or C can be activated**.
- **XOR merge -** task **C is enabled when either A or B terminate**.
- **OR split - after completion of task A one could activate either B, C, or both.**
- **Multiple Merge -** once task A terminates, B and C execute concurrently; when the first of them, say B, terminates, then D is activated; then, when C terminates, D is activated again.

Basic workflow patterns:

- **Discriminator –** wait for a number of incoming branches to complete before activating the subsequent activity; then wait for the remaining branches to finish without taking any action until all of them have terminated. Next, resets itself.

- **N out of M join -** barrier synchronization. Assuming that M tasks run concurrently, N (N<M) of them have to reach the barrier before the next task is enabled. In our example, any two out of the three tasks A, B, and C have to finish before E is enabled.

- **Deferred Choice -** similar to the XOR split but the choice is not made explicitly; the run-time environment decides what branch to take.


A
B
C
a
A
AND
B
C
b
A
B
AND
C
c
A
XOR
B
C
d
A
B
XOR
C
e
A
OR
B
C
f
A
AND
B
C
XOR
D
g
A
AND
B
C
DIS
D
h
A
AND
B
C
D
2/3
E
i
A
XOR
B
C
j

Reachability of Goal state

➢ A system Σ, an initial **state ,$\sigma_{initial}$, and goal state σ_{goal}** .

➢ A process group **P={p_1,p_2,…,p_n};**

- **each process pi in the process group is characterized by set of preconditions, pre(p_i),post conditions ,post(p_i) and attribtes ,attr(p_i).**

➢ A workflow described by a directed activity graph A or by a procedure π capable of constructing A given the tuple <P, $\sigma_{initial}$, σ_{goal} >.

- The nodes of A are processes in P and edges define precedence relations among processes. **P_i->p_j implies that pre(p_j)   post(p_i)**

➢ A set of constrints **C= { C_1,C_2,…,C_m}**

Coordination a state machine model: ZooKeeper

The coordination model depends on the specific task, such as coordination of data storage, orchestration of multiple activities, blocking an activity until an event occurs, reaching consensus for the next action, or recovery after an error.

- The **entities to be coordinated** could be processes running on a set of cloud servers or even running on multiple clouds.

- **Servers running critical tasks are often replicated**, so when one primary server fails, a backup automatically continues the execution.

- Consider now an advertising service that involves **a large number of servers in a cloud**. The advertising service runs on a number of servers specialized for tasks such as

  - **database access, monitoring, accounting, event logging, installers, customer dashboards**

- A solution for the **proxy coordination problem** is to consider a proxy as a deterministic finite state machine that performs the commands sent by clients in some sequence

ZooKeeper

- Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable **distributed coordination.**

- The **ZooKeeper framework was originally built at “Yahoo!”** for accessing their applications in an easy and robust manner. Later, Apache ZooKeeper became a standard for organized service used by **Hadoop, HBase, and other distributed frameworks.**

- ZooKeeper is a centralized service **for maintaining configuration information, naming, providing distributed synchronization, and providing group services.**

    - A distributed application **can run on multiple systems in a network at a given time** (simultaneously) by coordinating among themselves to complete a particular task in a fast and efficient manner.

    - A group of systems in which a distributed application is running is called a **Cluster** and each machine running in a cluster is called a **Node.**

- A distributed application has two parts, Server and Client application.

    - Server applications are actually distributed and have a common interface so that **clients can connect to any server in the cluster and get the same result.** Client applications are the tools to interact with a distributed application.

Why do we need Zookeeper in the Hadoop?

- Distributed applications are **difficult to coordinate and work** with as they are much **more error prone due to huge number of machines** attached to network.
  - As many machines are involved, **race condition and deadlocks are common problems** when implementing distributed applications.
  - Race condition occurs when **a machine tries to perform two or more operations at a time** and this can be taken care by serialization property of ZooKeeper.
- Deadlocks are when **two or more machines try to access same shared resource** at the same time…More precisely they try to access each other's resources which leads to lock of system as none of the system is releasing the resource but waiting for other system to release it. **Synchronization in Zookeeper helps to solve the deadlock.**
- Another major issue with **distributed application can be partial failure of process,** which can lead to inconsistency of data. **Zookeeper handles this through atomicity, which means either whole of the process will finish or nothing will persist after failure.**

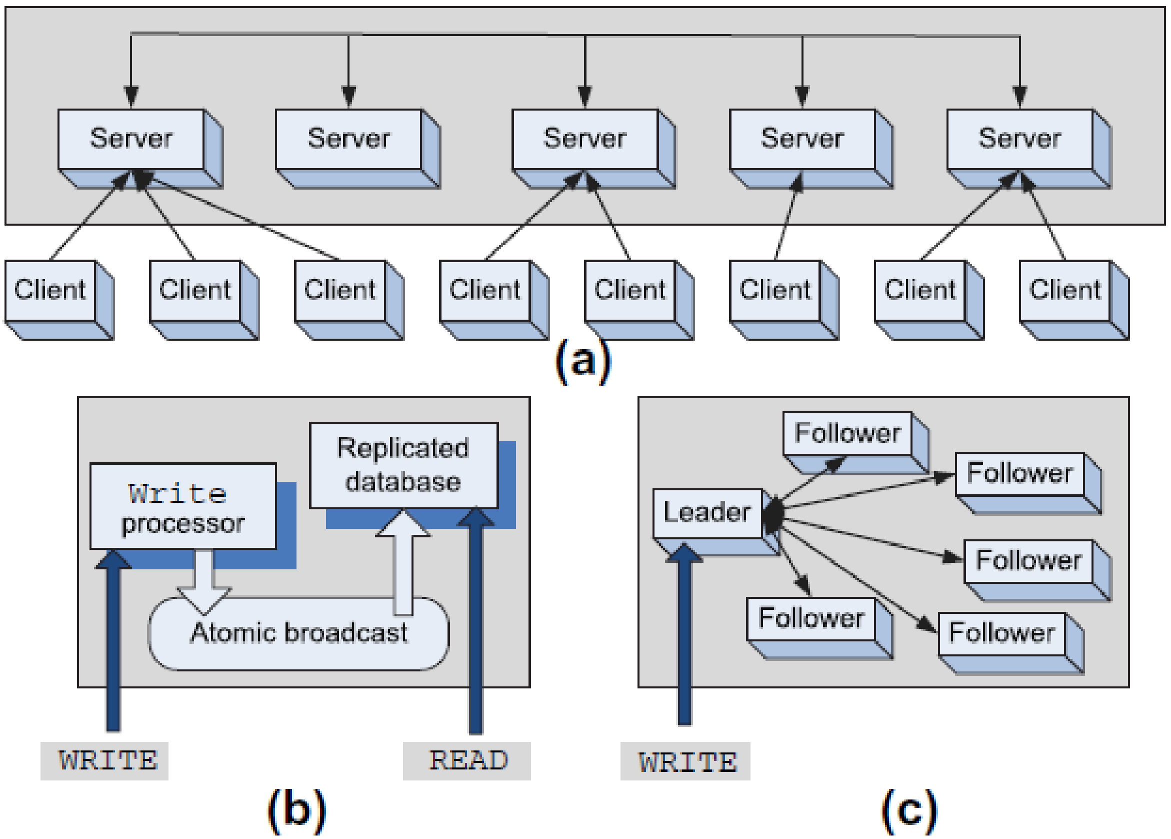


FIGURE 4.4

The *ZooKeeper* coordination service. (a) The service provides a single system image. Clients can connect to any server in the pack. (b) Functional model of the *ZooKeeper* service. The replicated database is accessed directly by `read` commands; `write` commands involve more intricate processing based on atomic broadcast. (c) Processing a `write` command: (1) A server receiving the command from a client forwards the command to the *leader*; (2) the *leader* uses atomic broadcast to reach consensus among all *followers*.

Figures 4.4(b) and (c) show that **a read operation** directed to any server in the pack returns the same result,

- whereas the processing of **a write operation is more involved**;
- the servers elect a leader, and any follower receiving a request from one of the clients connected to it forwards it to the leader.
- The leader uses atomic broadcast to reach consensus. When the leader fails, the servers elect a new leader.

The system is organized as a shared hierarchical namespace similar to the organization of a file system.

- A name is a sequence of path elements separated by a backslash. Every name in Zookeper's namespace is identified by a unique path (see Figure 4.5).

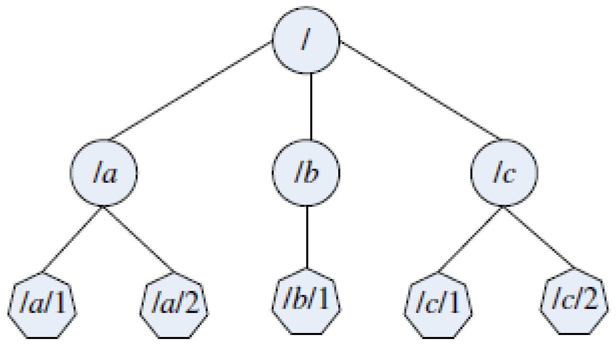


FIGURE 4.5

ZooKeeper is organized as a shared hierarchical namespace in which a name is a sequence of path elements separated by a backslash.

ZooKeeper service guarantees

- **Atomicity -** a transaction either completes or fails.
- **Sequential consistency of updates** - updates are applied strictly in the order they are received.
- **Single system image for the clients** - a client receives the same response regardless of the server it connects to.
- **Persistence of updates** - once applied, an update persists until it is overwritten by a client.
- **Reliability -** the system is guaranteed to function correctly as long as the majority of servers function correctly.

Zookeeper API is simple - consists of seven operations:

- **Create -** add a node at a given location on the tree.
- **Delete -** delete a node.
- **Get data** - read data from a node.
- **Set data** - write data to a node.
- **Get children** - retrieve a list of the children of the node.
- **Synch** - wait for the data to propagate.

Messaging layer → responsible for the election of a new leader when the current leader fails. Messaging protocols use:

> **Packets - sequence of bytes sent through a FIFO channel.**
> **Proposals - units of agreement.**
> **Messages - sequence of bytes atomically broadcast to all servers**.

- A message is included into a proposal and it is agreed upon before it is delivered.
- Proposals are agreed upon by exchanging packets with a quorum of servers, as required by the Paxos algorithm.

Messaging layer guarantees:

- **Reliable delivery**: if a message **m** is delivered to one server, it will be eventually delivered to all servers.
- **Total order**: if message **m** is delivered before message **n** to one server, it will be delivered before **n** to all servers.
- **Causal order**: if message **n** is sent after **m** has been delivered by the sender of **n**, then **m** must be ordered before **n**.

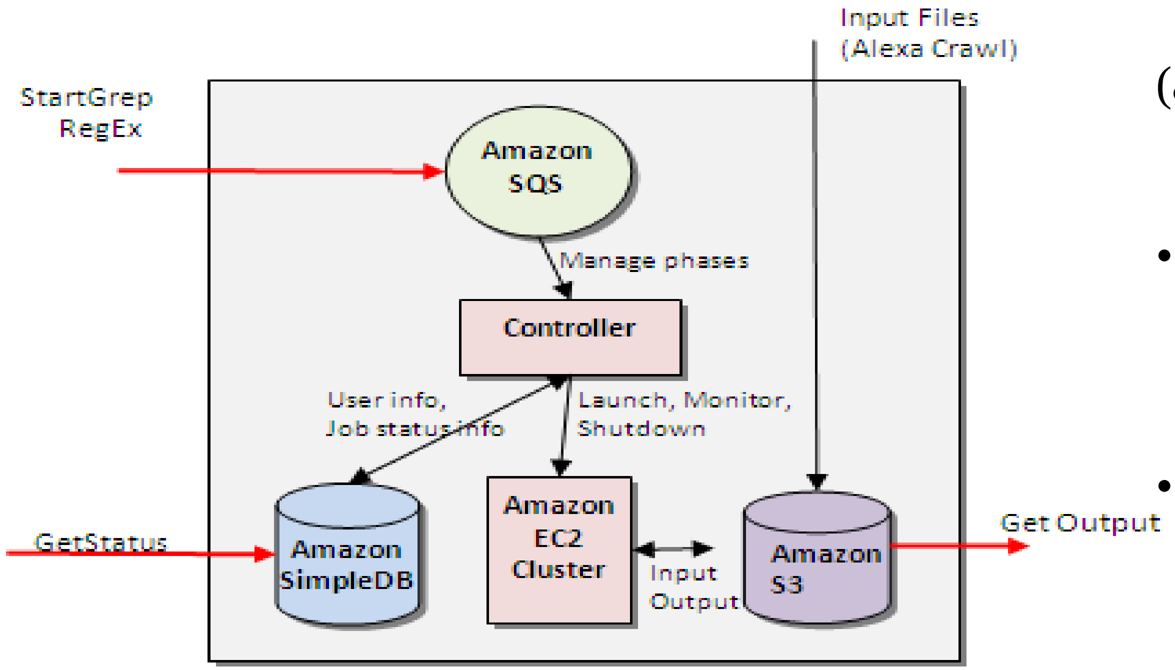
Case study: GrepTheWeb

- An application called **GrepTheWeb**, is now in production at Amazon.
    - The application allows a user to **define a regular expression** and **search the Web** for records that match it. GrepTheWeb is analogous to the **Unix grep command used to search a file** for a given regular expression.
    - This **application performs a search of a very large set of records,** attempting to identify records that satisfy a regular expression.
    - The source of this search is a **collection of document URLs** produced by the **Alexa Web Search,** a software system that crawls the Web every night.
        - The **inputs to the applications are a regular expression** and the **large data set produced by the Web-crawling software**;
        - the **output is the set of records** that satisfy the expression.

GrepTheWeb uses Hadoop MapReduce,

- an **open-source software pa**ckage that splits a large data set into chunks, distributes them across multiple systems, launches the processing, and, when the processing is complete, aggregates the outputs from different systems into a final result.

- Apache Hadoop is a software library for distributed processing of large data sets across clusters of computers using a simple programming model.



(a) The **simplified workflow** showing the two inputs,

- the **regular expression** and the **input records** generated by the Web crawler.

- A third type of input is the user commands to report the **current status** and to terminate the processing.

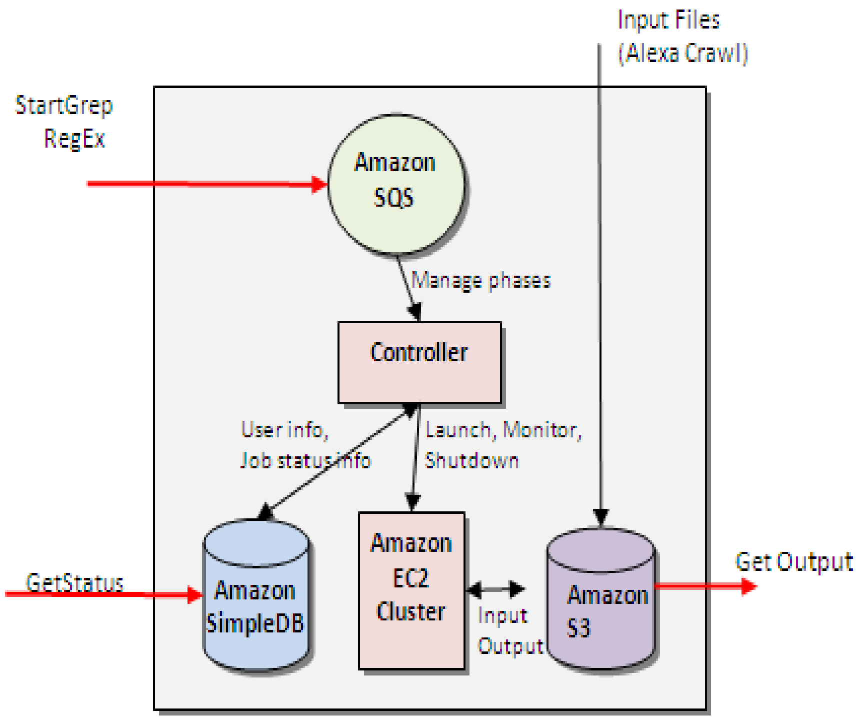
- **Amazon S3(Simple Storage Service)** for retrieving input datasets and for storing the output dataset

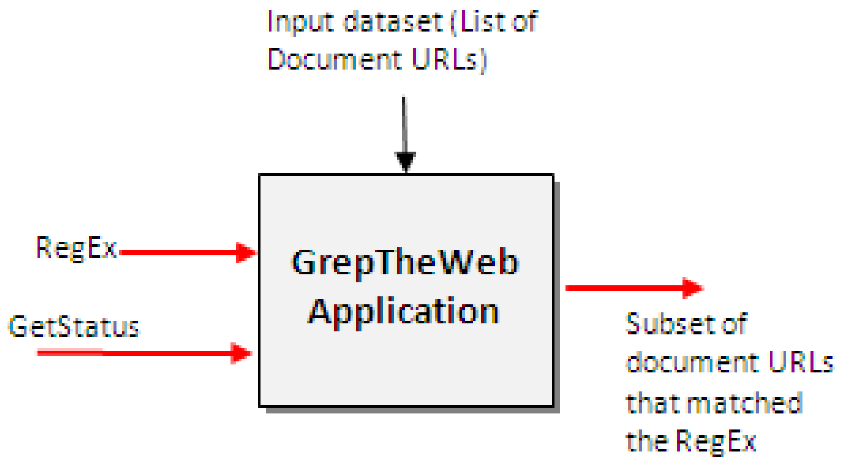
- **Amazon SQS(Simple Queue Service)** for durably buffering requests acting as a "glue" between controllers

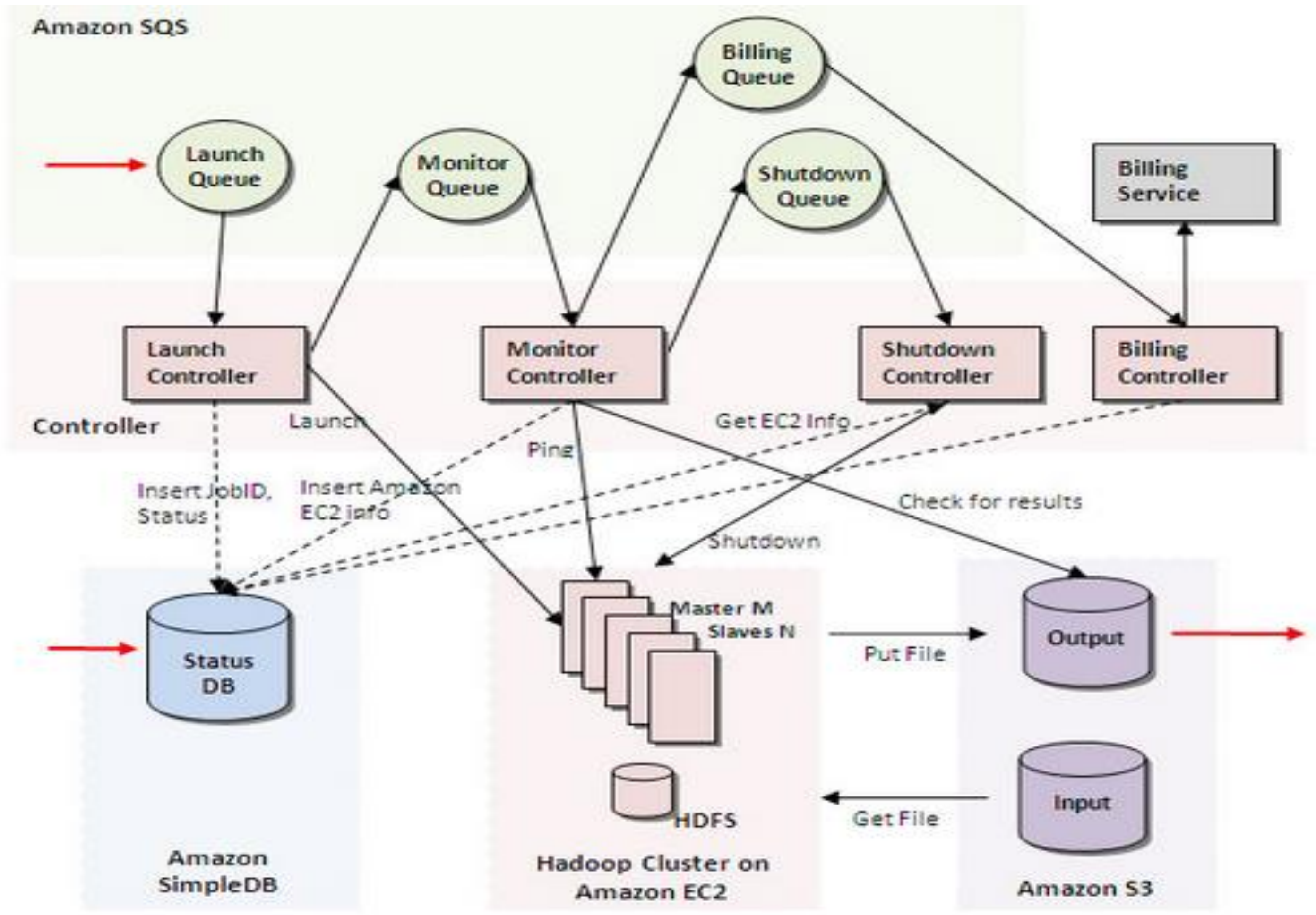
- **Amazon SimpleDB** for storing intermediate status, log, and for user data about tasks

- **Amazon EC2** for running a large distributed processing Hadoop cluster on-demand

- **Hadoop** for distributed processing, automatic parallelization, and job scheduling




Input dataset (List of Document URLs)
RegEx
GetStatus
GrepTheWeb Application
Subset of document URLs that matched the RegEx



The organization of the GrepTheWeb application. The application uses the Hadoop MapReduce software and four Amazon services: EC2, Simple DB, S3, and SQS.

(b) The detailed workflow; the system is based on message passing between several queues; four controller threads periodically poll their associated input queues, retrieve messages, and carry out the required actions.

GrepTheWeb is modular. It does its processing in four phases: Phases of GrepTheWeb Architecture

- **The startup phase.** Creates several queues – **launch, monitor, billing, and shutdown queues.** Starts the corresponding controller threads. Each thread periodically polls its input queue and, when a message is available, retrieves the message, parses it, and takes the required actions.
- **The processing phase.** This phase is triggered by a StartGrep user request; then a launch message is enqueued in the launch queue. The **launch controller thread** picks up the message and executes the launch task; then, it updates the status and time stamps in the Amazon Simple DB domain.

  - **Launch phase** is responsible for **validating and initiating the processing GrepTheWeb request,** instantiating Amazon EC2 instances, launching Hadoop cluster on them and starting all job processes.
  - **Monitor phase** is responsible for **monitoring the EC2 cluster**, maps, reduces, and checking for success and failure.
  - **Shutdown phase** is responsible for **billing and shutting down all Hadoop processes** and Amazon EC2 instances,
  - **Cleanup phase** **deletes Amazon Simple DB** transient data

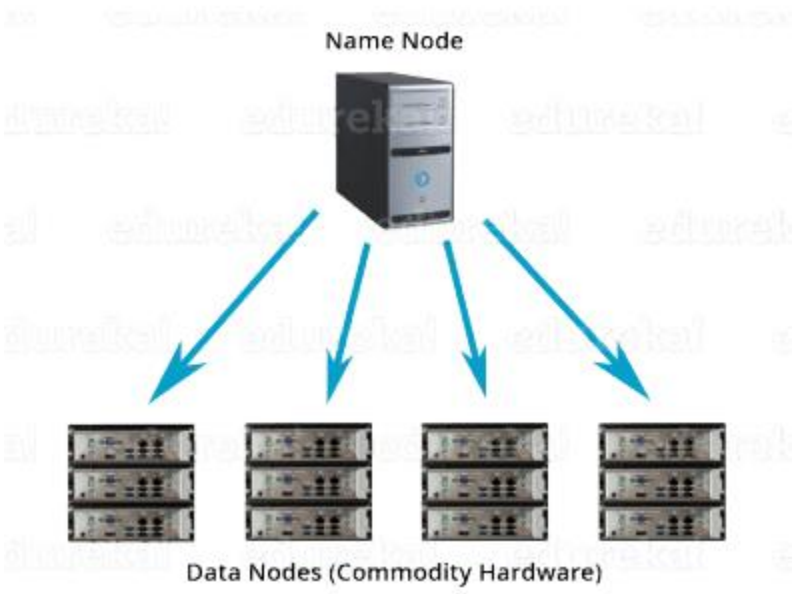
- **Apache Hadoop –** Introduction, Architecture and Components, HDFS,
- The **Map Reduce programming model**, YARN,
- **Interacting with Hadoop ecosystem** – Pig, Hive, HBase, Sqoop

Apache Hadoop HDFS Architecture

Apache HDFS or **Hadoop Distributed File System** is a block-structured file system where each file is divided into blocks of a pre-determined size. These blocks are stored across a cluster of one or several machines.

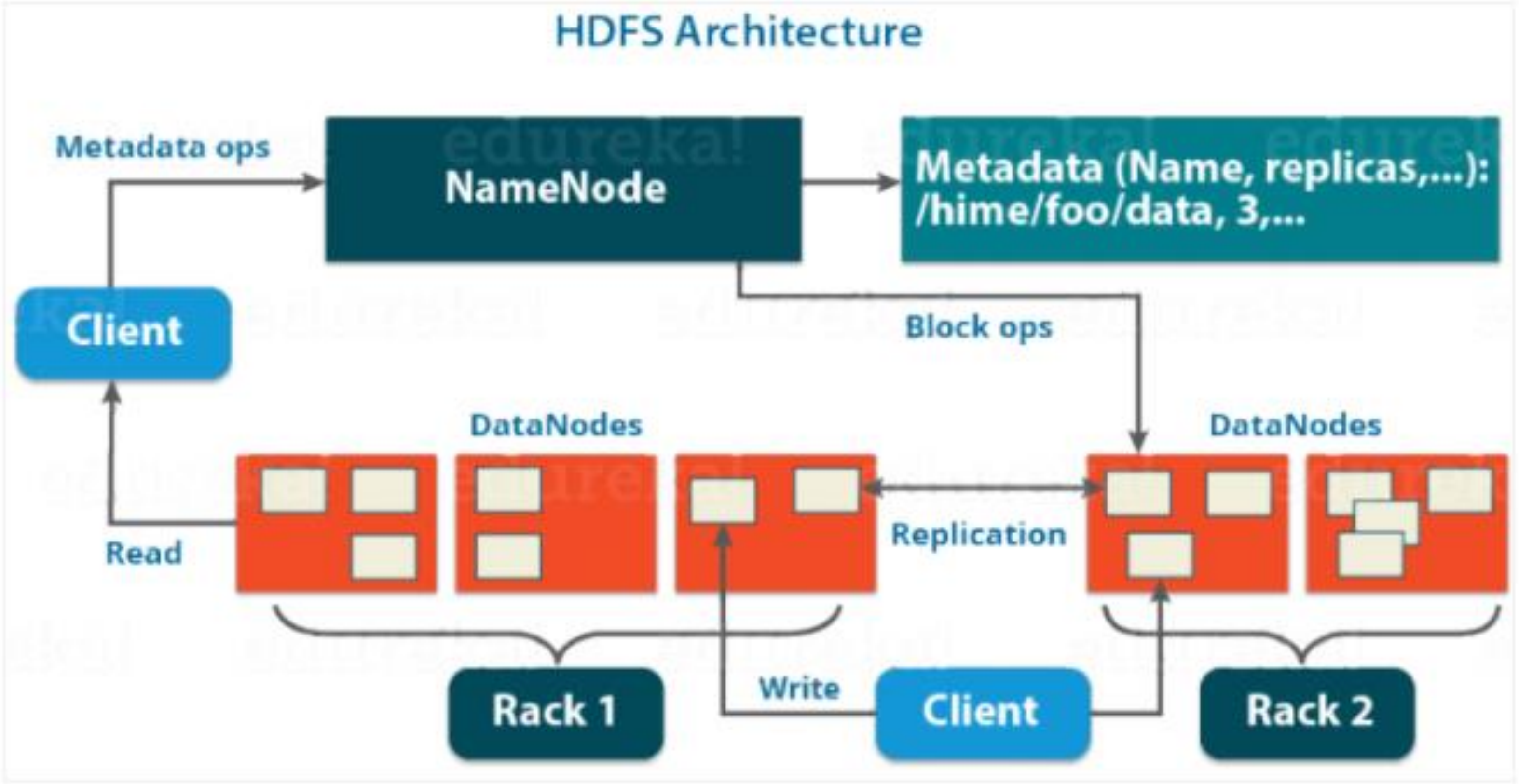
- Apache Hadoop **HDFS Architecture follows a Master/Slave Architecture**, where a cluster comprises of a single **NameNode (Master node)** and all the other nodes are **DataNodes (Slave nodes).**

- HDFS can be deployed on a broad spectrum of machines that support Java. Though one can run several DataNodes on a single machine, but in the practical world, these DataNodes are spread across various machines.

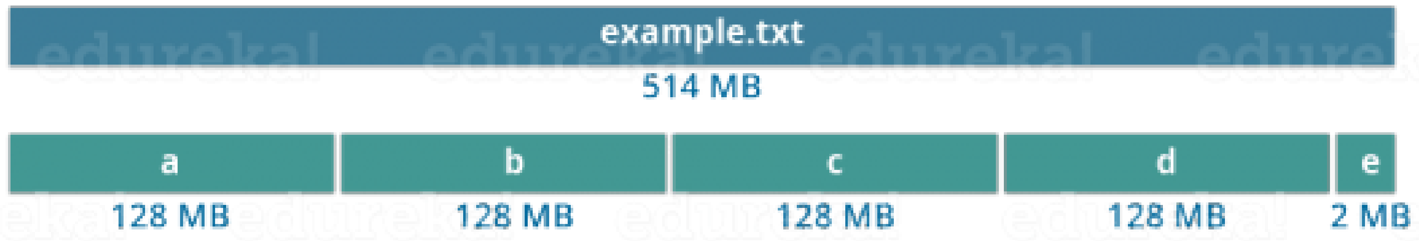


Key Aspects of Hadoop

- **Open Source Software:** It is free to download ,use and Contribute
- **Framework:** Everything that you will need to develop and execute and applications is provided-Programs and Tools etc.
- **Distributed:** Divided and stores data across multiple computers. Computation/Processing is done in parallel across multiple connected nodes.
- **Massive Storage:** Stores massive amount of data across nodes of low cost commodity hardware.
- **Fast Processing:** Large amounts of data is processed in parallel, and having quick response.

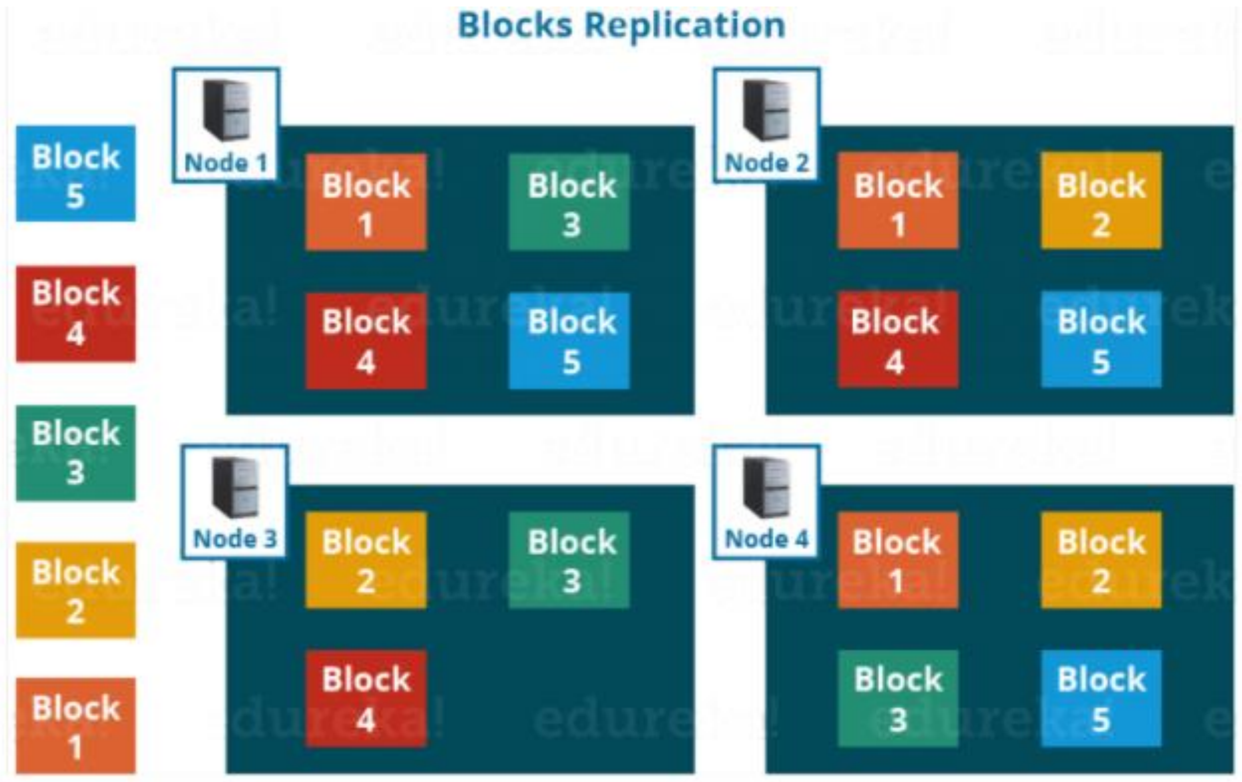


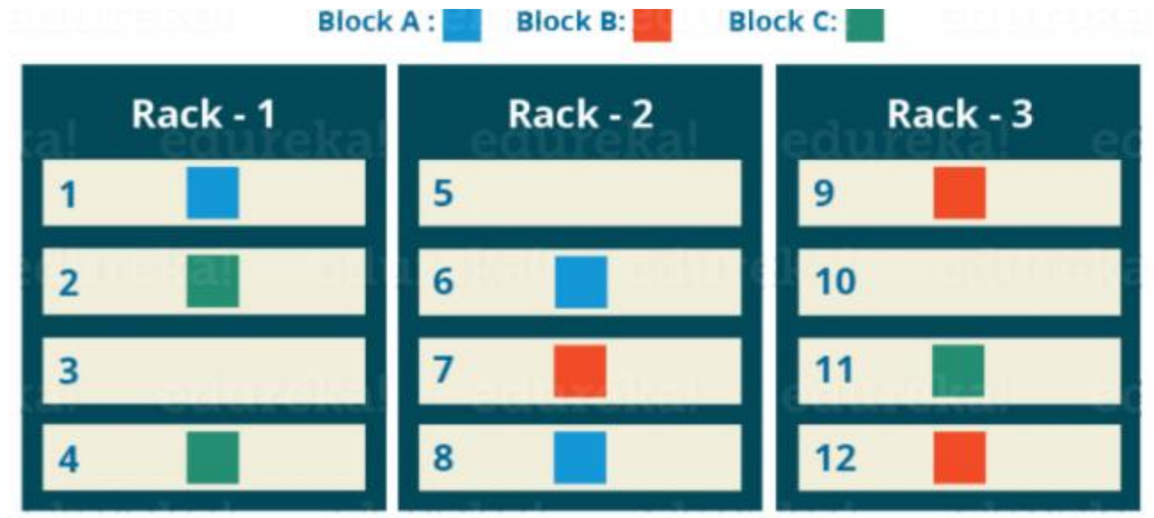
Data in HDFS is scattered across the DataNodes as blocks

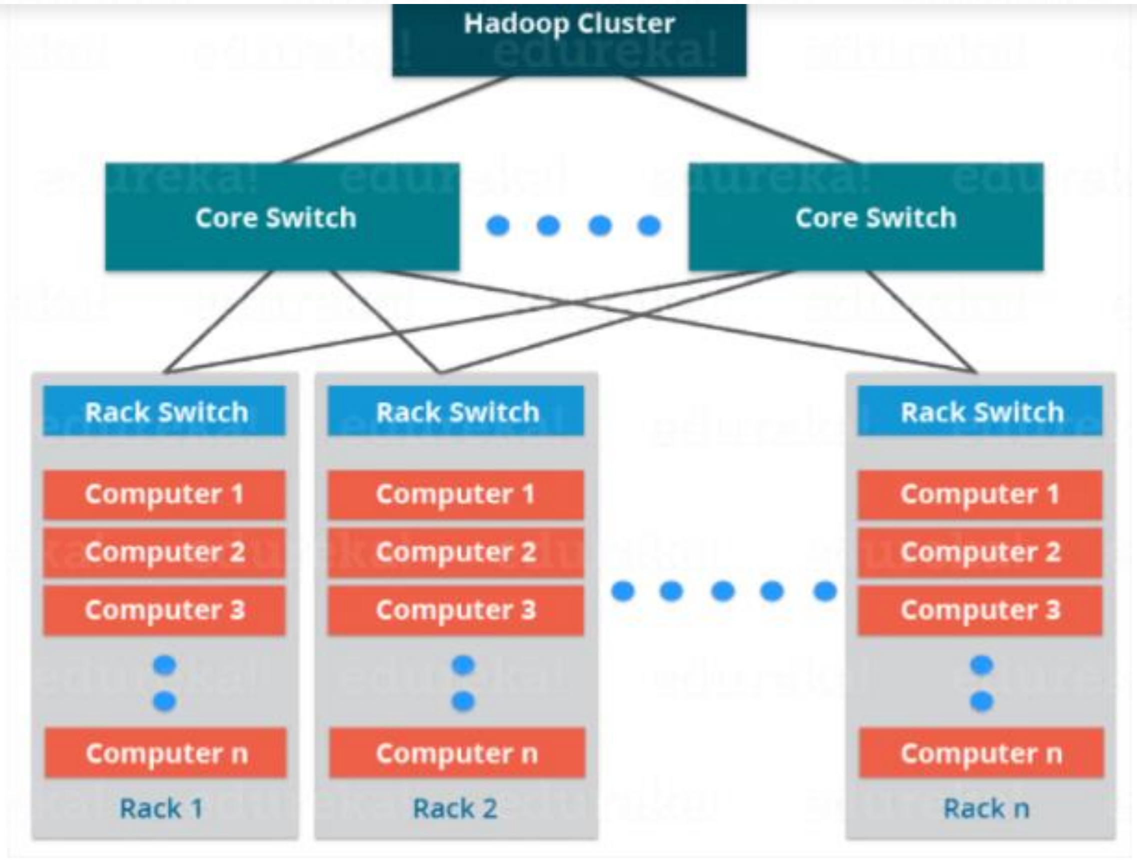


HDFS provides a **reliable way to store huge data in a distributed environment** as data blocks. The blocks are also replicated to provide **fault tolerance.**

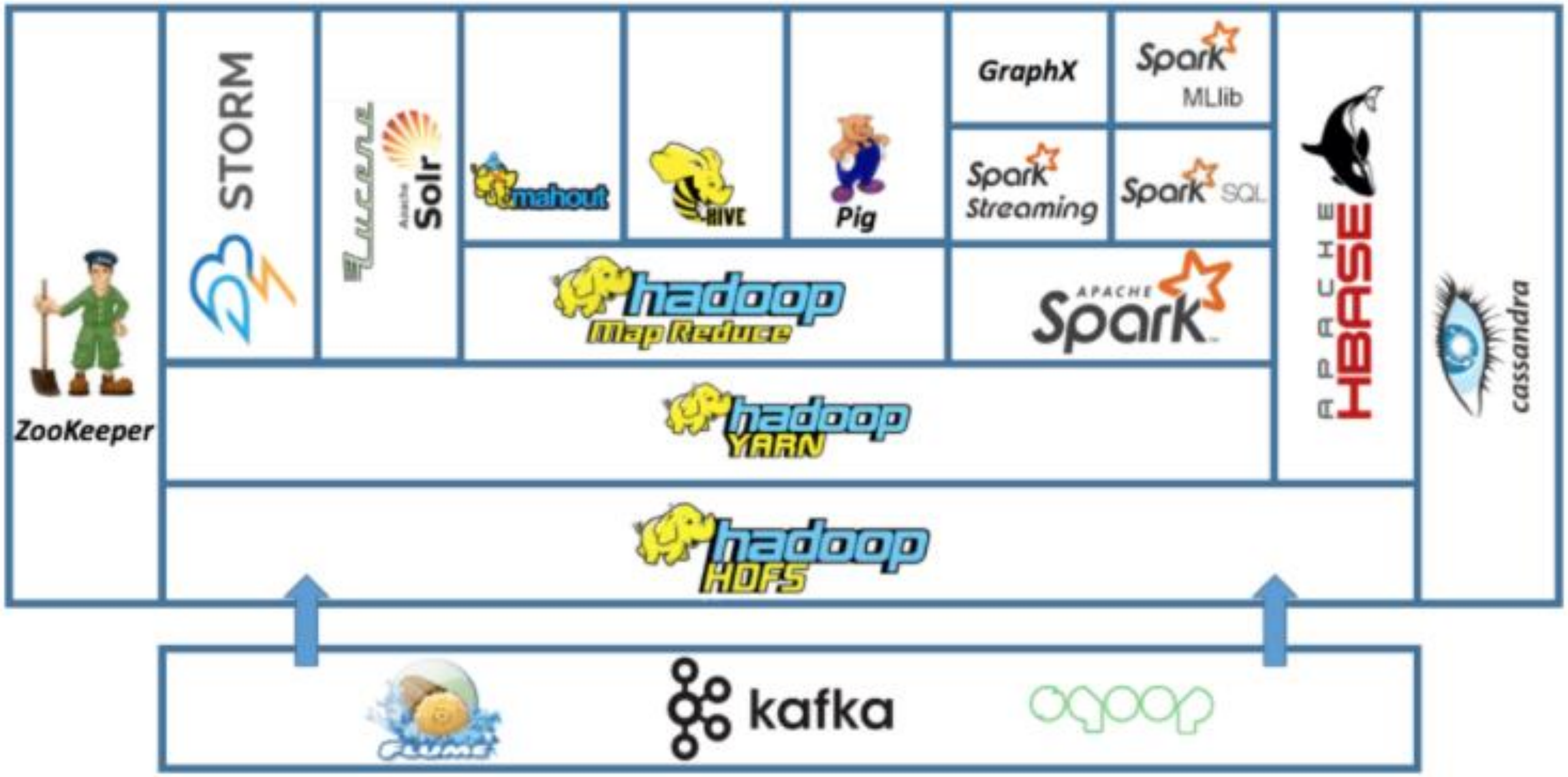
The **default replication factor is 3** which is again configurable. So, as you can see in the figure below where **each block is replicated three times** and stored on different DataNodes (considering the default replication factor):




Block A : 🟦
Block B: 🟥
Block C: 🟩
Rack - 1
1
2
3
4
Rack - 2
5
6
7
8
Rack - 3
9
10
11
12


Hadoop Cluster
Core Switch
Core Switch
Rack Switch
Computer 1
Computer 2
Computer 3
Computer n
Rack 1
Rack Switch
Computer 1
Computer 2
Computer 3
Computer n
Rack 2
Rack Switch
Computer 1
Computer 2
Computer 3
Computer n
Rack n

Hadoop Components



The Hadoop Distributed File system

When a dataset outgrows the storage capacity of a single physical machine, it becomes **necessary to partition it across a number of separate machines**.

- File systems that manage the storage across a network of machines are called **distributed file systems.** Since they are network based, all the complications of network programming kick in, thus making distributed file systems more complex than regular disk file systems.
    - For example, one of **the biggest challenges is making the file system tolerate node failure without suffering data loss.**
    - Hadoop comes with **a distributed file system called HDFS,** which stands for **Hadoop Distributed File system.**
    - Hadoop actually has a general purpose filesystem abstraction, so we'll see along the way how Hadoop integrates with **other storage systems** **(such as the local file system and Amazon S3).**

The Design of HDFS

HDFS is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware

Very large files

- "Very large" in this context means files that are **hundreds of megabytes, gigabytes**, or terabytes in size. There are **Hadoop clusters running today that store petabytes of data**

Streaming data access

- HDFS is built around the idea that the most efficient data processing pattern is a **write-once, read-many-times pattern.**
- A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so **the time to read the whole dataset is more important than the latency in reading the first record.**

- **Commodity hardware**
  - Hadoop doesn't require expensive, highly reliable hardware. It's designed to run on clusters of commodity hardware
  - **HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure**

Namenodes and Datanodes

An HDFS cluster has two types of nodes operating in a master–worker pattern:

- **a Namenode (the master)** and **a number of Datanodes (workers)**.
  - Namenode **manages the filesystem namespace**.
    - It maintains the **filesystem tree** and the **metadata for all the files** and **directories in the tree**.
    - This information is stored persistently on the local disk in the form of **two files: the namespace image and the edit log.**
  - Datanodes **are the workhorses of the filesystem.**
    - They store and retrieve blocks when they are told to (by clients or the namenode), and they **report back to the namenode periodically** with lists of blocks that they are storing.

Functions of NameNode:

- It is the master computer that maintains and manages the DataNodes (slave nodes)

- It records the metadata of all the files stored in the cluster, e.g. **The location of blocks stored, the size of the files, permissions, hierarchy, etc**.

- It **records each change that takes place to the file system metadata**. For example, if a file is deleted in HDFS, the NameNode will immediately record this in the EditLog

    - **FsImage:** It contains the **complete state of the file system namespace** since the start of the NameNode.

    - **EditLogs:** It contains all the **recent modifications made to the file system** with respect to the most recent FsImage.

- It regularly receives a **Heartbeat and a block report** from all the DataNodes in the cluster to ensure that the DataNodes are live.

- In case of the DataNode failure, the NameNode **chooses new DataNodes** for new replicas, balance disk usage and manages the communication traffic to the DataNodes.

Functions of DataNode:

- These are slave computer or process which runs on each slave machine.
- The actual data is stored on DataNodes.
- The DataNodes perform the low-level **read and write requests from the file system's clients.**
- They send **heartbeats to the NameNode periodically** to report the overall health of HDFS, by default, this frequency is set to 3 seconds.

Anatomy of a File Read

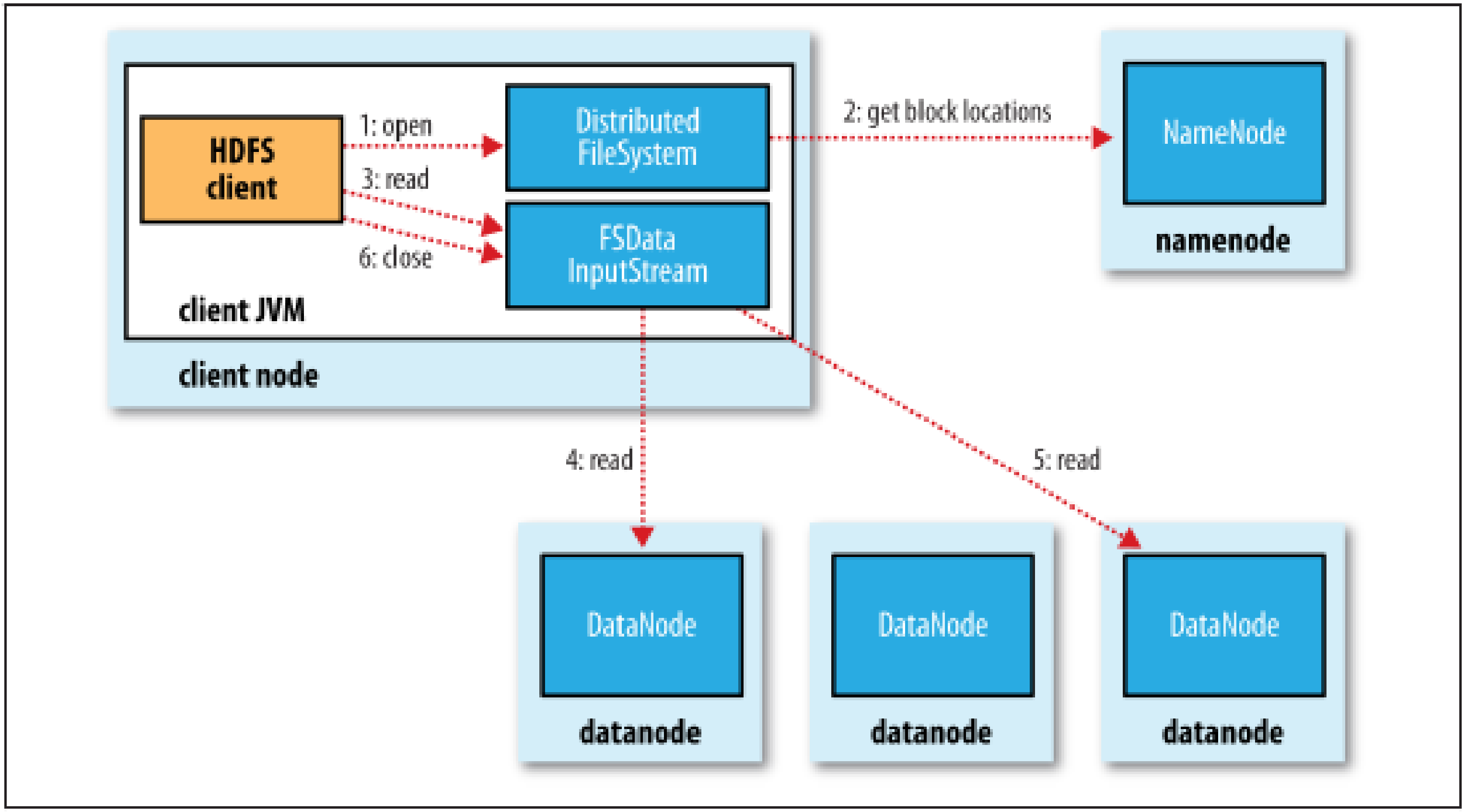


Figure 3-2. A client reading data from HDFS

To get an idea of how data flows between the client interacting with HDFS, the namenode, and the datanodes, consider Figure 3-2, which shows **the main sequence of events when reading a file.**

- The client **opens the file** it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem **(step 1)**

- DistributedFileSystem calls the namenode, using remote procedure calls (RPCs), to **determine the locations of the first few blocks** in the file **(step 2).**

    - For each block, the **namenode returns the addresses of the datanodes that have a copy of that block**. Furthermore, the datanodes are sorted according to their proximity to the client (according to the topology of the cluster's network

    - The DistributedFileSystem returns an FSDataInputStream (an input stream that **supports file seeks**) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

- The **client then calls read() on the stream** **(step 3).** DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file.

- Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream **(step 4).**

- When the **end of the block is reached, DFSInputStream will close the connection** to the datanode, then find the best datanode for the next block **(step 5).**

- Blocks are read in order, with the **DFSInputStream opening new connections to datanodes as the client reads through the stream**.

- It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. **When the client has finished reading, it calls close() on the FSDataInputStream (step 6).**

    - During reading, **if the DFSInputStream encounters an error** while communicating with a datanode, **it will try the next closest one for that block**. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks.

    - The DFSInput Stream also verifies checksums for the data transferred to it from the datanode**. If a corrupted block is found,** the DFSInputStream attempts to read a replica of the block from another datanode; **it also reports the corrupted block to the namenode.**

Anatomy of a File Write

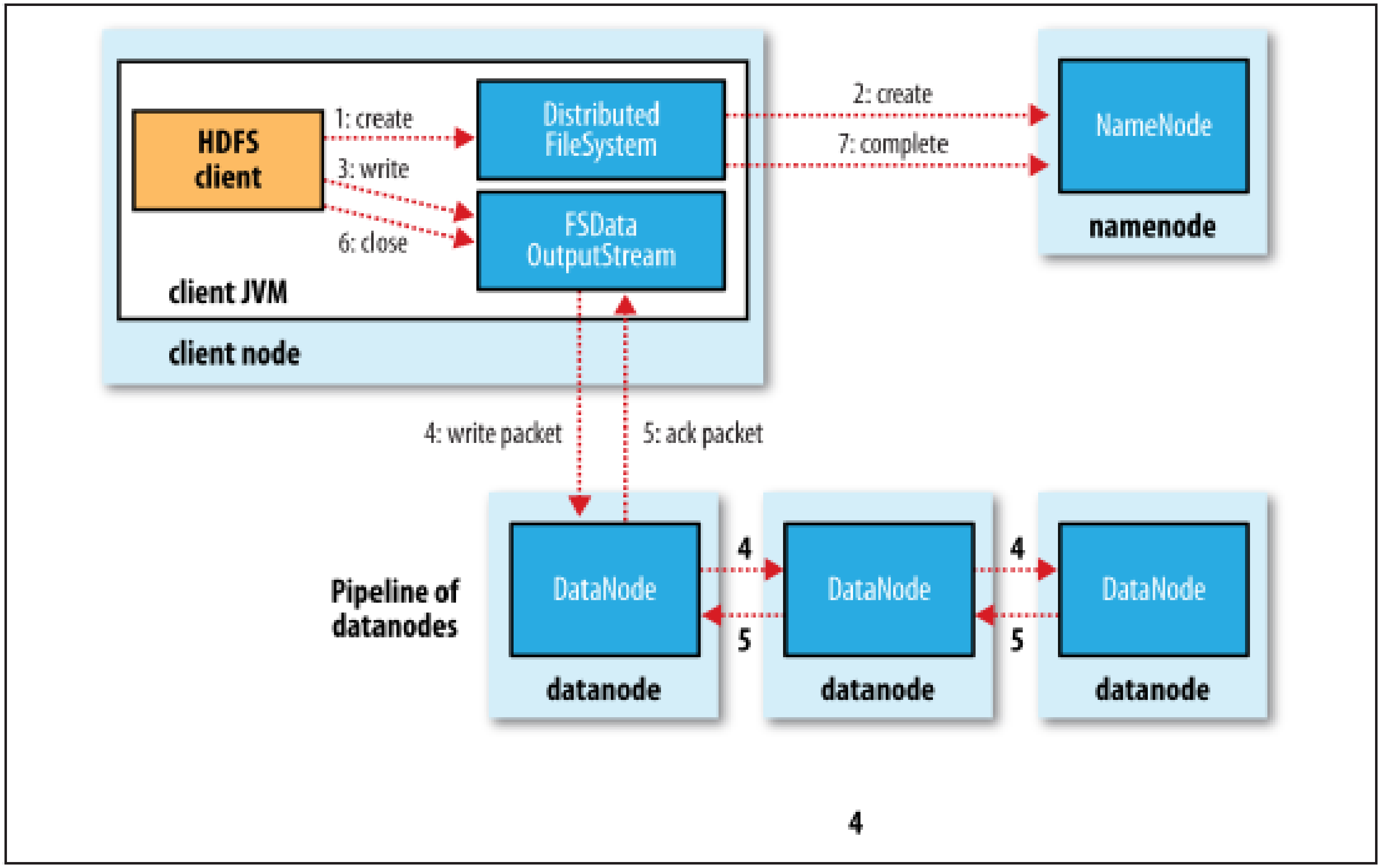


Figure 3-4. A client writing data to HDFS

Consider the case of creating a new file, writing data to it, then closing the file. This is illustrated in Figure 3-4.

- The **client creates the file by calling create()** on DistributedFileSystem (**step 1** Figure 3-4).

- DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem's namespace, **with no blocks associated with it (step 2).**

  - The **namenode performs various checks to make sure the file doesn't already exist** and that the **client has the right permissions to create the file.** If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException.

- As the client writes data **(step 3),** the **DFSOutputStream splits it into packets**, which it writes to **an internal queue called the data queue**. The data queue is consumed by the DataStreamer, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas.

  - The list of **datanodes forms a pipeline**, and here we'll assume the replication level is three, so there are three nodes in the pipeline

- The DataStreamer streams the packets to the **first datanode in the pipeline, which stores each packet and forwards it to the second datanode in the pipeline**. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline **(step 4).**

If any datanode fails while data is being written to it, then the following actions are taken:

- **First, the pipeline is closed,** and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets.
- The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on.
- **The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes.**

The DFSOutputStream also maintains **an internal queue of packets** that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline **(step 5).**

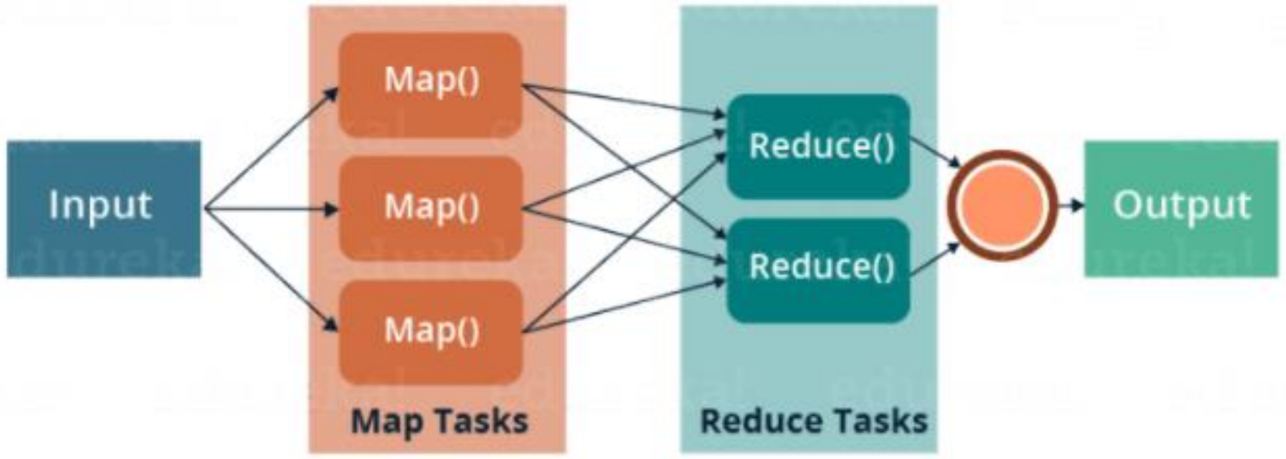
When the client has finished writing data, it calls close() on the stream **(step 6).**

This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete **(step7).**

The Map Reduce programming model

MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.

- MapReduce consists of two distinct tasks – Map and Reduce.
- As the name MapReduce suggests, the reducer phase takes place after the mapper phase has been completed.So, the first is the map job, where **a block of data is read and processed to produce key-value pairs as intermediate outputs**.
- The output of a Mapper or map job (key-value pairs) is input to the Reducer.
- **The reducer receives the key-value pair from multiple map jobs.**
- Then, the **reducer aggregates those intermediate data tuples** (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.



Mapper Class:

The first stage in Data Processing using MapReduce is the Mapper Class. Here, Record Reader processes **each Input record and generates the respective key-value pair**. Hadoop's Mapper store saves this intermediate data into the local disk.

Input Split

It is the logical representation of data. It represents a block of work that contains a single map task in the MapReduce Program.

RecordReader

It interacts with the Input split and converts the obtained data in the form of Key-Value Pairs.

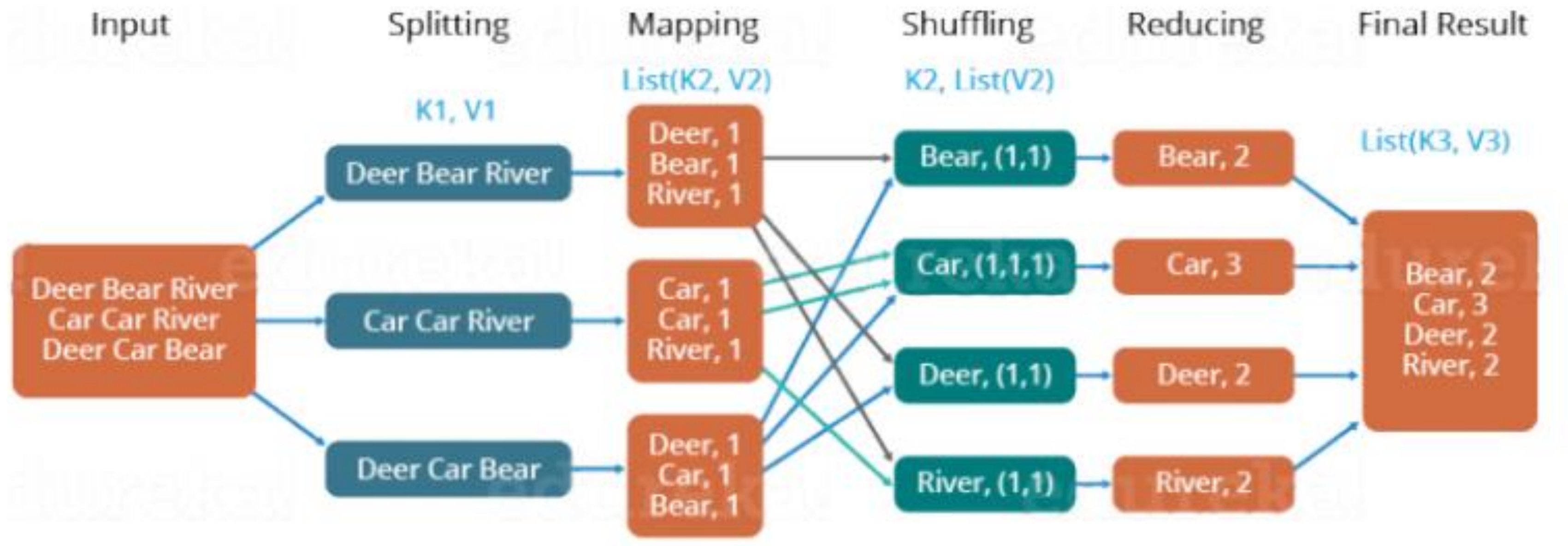
Reducer Class:

The Intermediate output generated from the mapper is fed to the reducer which processes it and generates the final output which is then saved in the HDFS.

Driver Class :

The major component in a MapReduce job is a Driver Class. It is responsible for setting up a MapReduce Job to run-in Hadoop. We specify the names of Mapper and Reducer Classes long with data types and their respective job names.

MapReduce : A Word Count Example



Map and Reduce

- MapReduce works by breaking the **processing into two phases**:
    - **the map phase and the reduce phase.**
- Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer.
- The programmer also specifies two functions: **the map function and the reduce function**

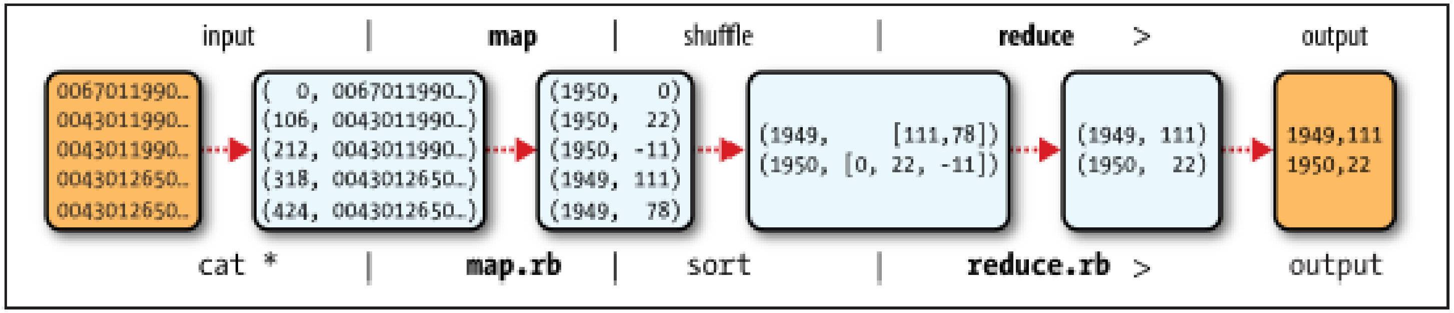


Figure 2-1. MapReduce logical data flow

MapReduce data flow with a single reduce task

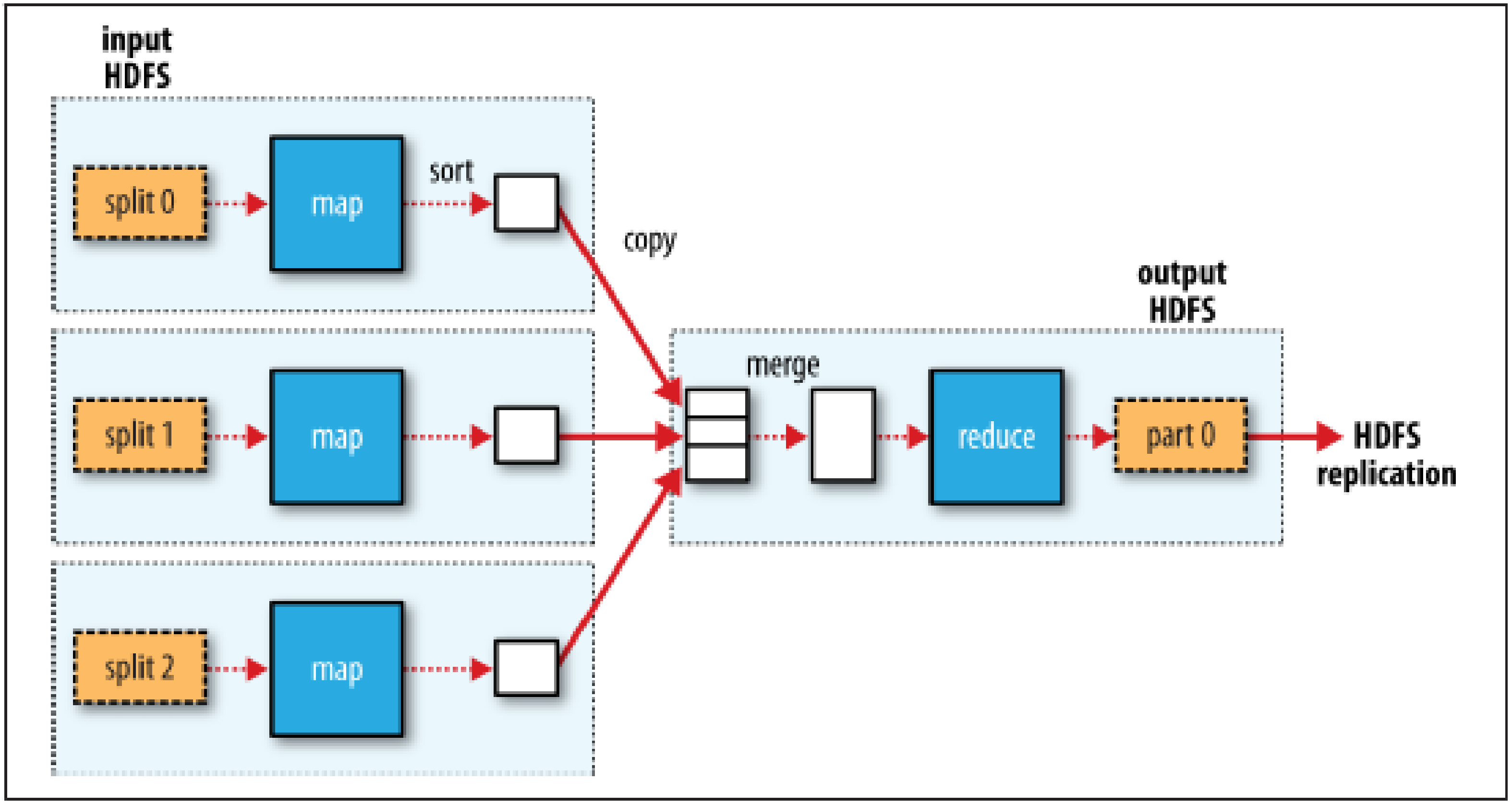


Figure 2-3. MapReduce data flow with a single reduce task

MapReduce data flow with a Multiple reduce task

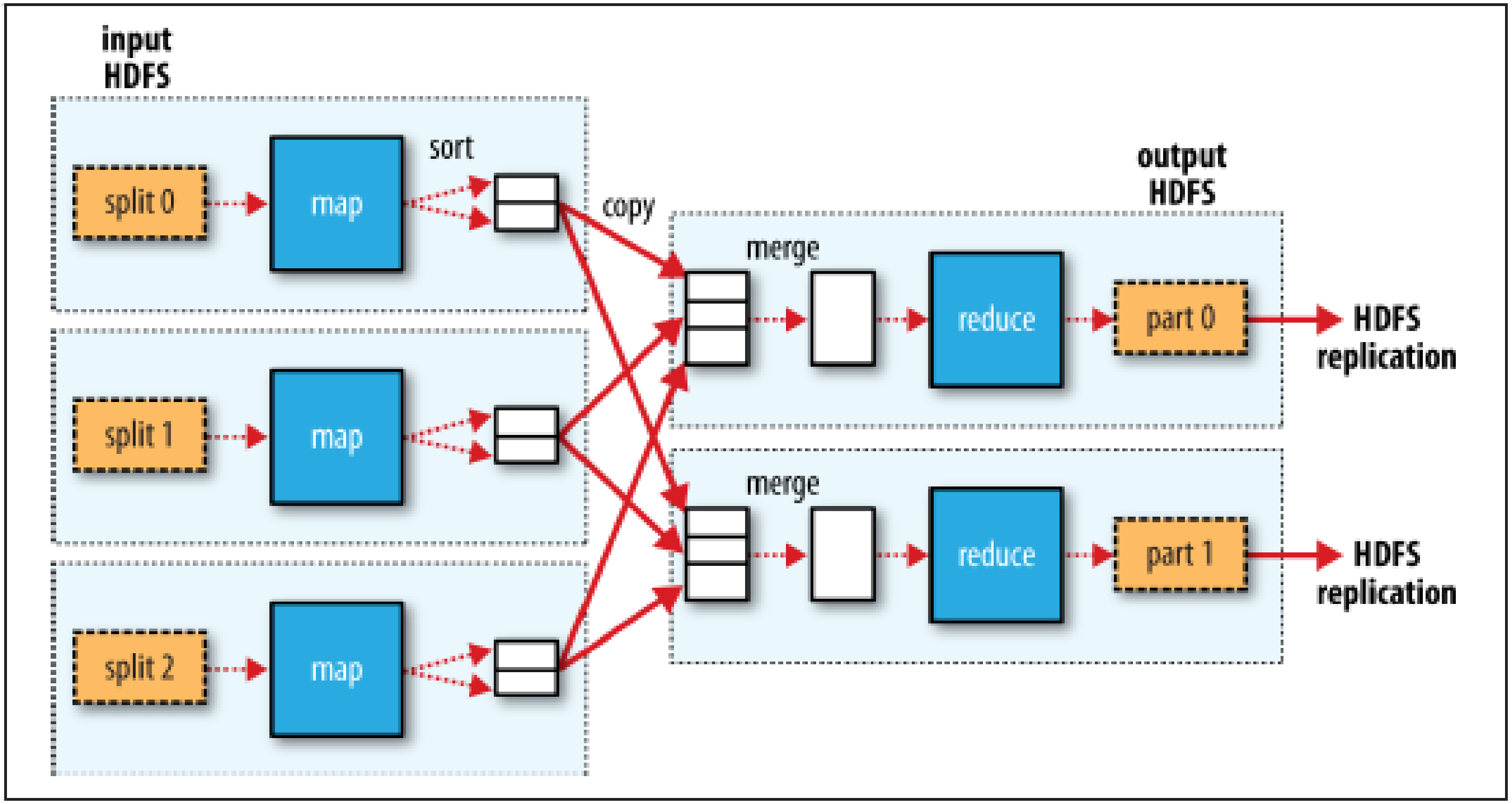


Figure 2-4. MapReduce data flow with multiple reduce tasks

Anatomy of a MapReduce Job Run

Classic MapReduce (MapReduce 1)

A job run in classic MapReduce is illustrated in Figure 6-1. At the highest level, there are four independent entities:

• The **client**, which submits the MapReduce job.

• The **Jobtracker,** which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.

• The **Tasktrackers,** which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.

• The **Distributed filesystem** (normally HDFS, covered in Chapter 3), which is used for sharing job files between the other entities.

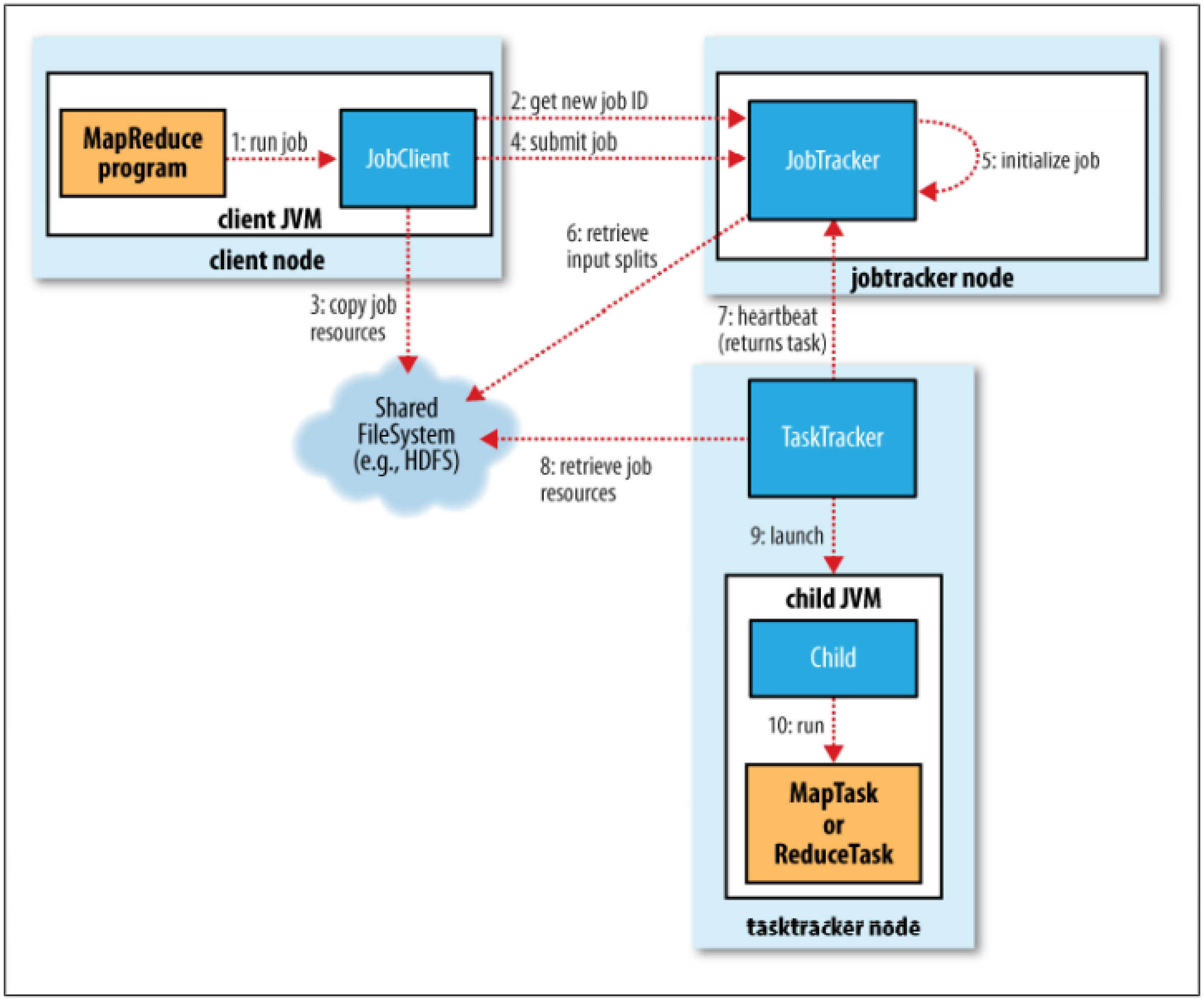


Figure 6-1. How Hadoop runs a MapReduce job using the classic framework

Job Submission:

- The submit() method on Job creates an internal **JobSummitter** instance and calls sub mitJobInternal() on it (step 1 in Figure 6-1). The job submission process implemented by JobSummitter does the following:

- **Asks the jobtracker for a new job ID** (by calling getNewJobId() on JobTracker) (step 2).

    - **Checks the output specification of the job.** For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.

    - **Computes the input splits for the job.** If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.

    - **Copies the resources needed to run the job**, including the job JAR file, the configuration file, and the computed

- The job JAR is copied with a high replication factor ,so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job (step 3).

- Tells the jobtracker that the job is ready for execution (by calling submitJob() on JobTracker) (step 4).

Job Initialization :

- When the JobTracker receives a call to its submitJob() method, it puts it into **an internal queue from where the job scheduler will pick it up and initialize it.**

- Initialization involves **creating an object to represent the job being run**, which encapsulates its tasks, and bookkeeping information to keep track of the tasks' status and progress (step 5).

- To create the list of tasks to run, the **job scheduler first retrieves the input splits computed by the client** from the shared filesystem (step 6).

- It then creates **one map task for each split**. The number of reduce tasks to create is determined by the mapred.reduce.tasks property in the Job, which is set by the setNumReduceTasks() method, and the **scheduler simply creates this number of reduce tasks to be run**. Tasks are given IDs at this point

Task Assignment :

- Tasktrackers run a simple loop that **periodically sends heartbeat method calls to the jobtracker.**

- **Heartbeats tell the jobtracker that a tasktracker is alive,** but they also double as a channel for messages.

- As a part of the heartbeat, **a tasktracker will indicate whether it is ready to run a new task,** and if it is, the **jobtracker will allocate it a task,** which it communicates to the tasktracker using the heartbeat return value (step 7).

Task Execution :

- Now that the tasktracker has been assigned a task, the next step is for it to run the task.
  - **First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker's filesystem. I**t also copies any files needed from the distributed cache by the application to the local disk; see "Distributed Cache" on page 288 (step 8).
  - **Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory.**
  - **Third, it creates an instance of TaskRunner to run the task. TaskRunner launches a new Java Virtual Machine (step 9) to run each task in (step 10),** so that any bugs in the user-defined map and reduce functions don't affect the tasktracker (by causing it to crash or hang, for example).
  - It is, however, possible to reuse the JVM between tasks;

YARN (MapReduce 2)

For very large clusters in the region of **4000 nodes and higher, the MapReduce system described in the previous section begins to hit scalability bottlenecks**, so in 2010 a group at Yahoo! began to design the next generation of MapReduce.

The result was **YARN,** short for **Yet Another Resource Negotiator** (or if you prefer recursive ancronyms, **YARN Application Resource Negotiator**)

YARN meets the **scalability shortcomings of "classic" MapReduce** by splitting the responsibilities of the jobtracker into separate entities.

- The jobtracker takes care of both **job scheduling (matching tasks with tasktrackers) and task progress monitoring (keeping track of tasks and restarting failed or slow tasks, and doing task bookkeeping such as maintaining counter totals).**

- YARN separates these two roles into two independent daemons:

  - a **resource manager** to manage the use of resources across the cluster, and
  - **an application master** to manage the lifecycle of applications running on the cluster.

The steps Hadoop takes to run a job: The whole process is illustrated in Figure 7-1. At the highest level, there are five independent entities

- The **client,** which submits the MapReduce job.
- The **YARN resource manager,** which **coordinates the allocation of compute resources** on the cluster.
- The **YARN node managers,** which **launch and monitor the compute containers on machines** in the cluster.
- The **MapReduce application master**, which coordinates the tasks running the Map- Reduce job. The application master and the MapReduce tasks run in containers that are **scheduled by the resource manager and managed by the node managers.**
- The **distributed filesystem** (HDFS, covered in Chapter 3), which is used for **sharing job files between the other entities**.

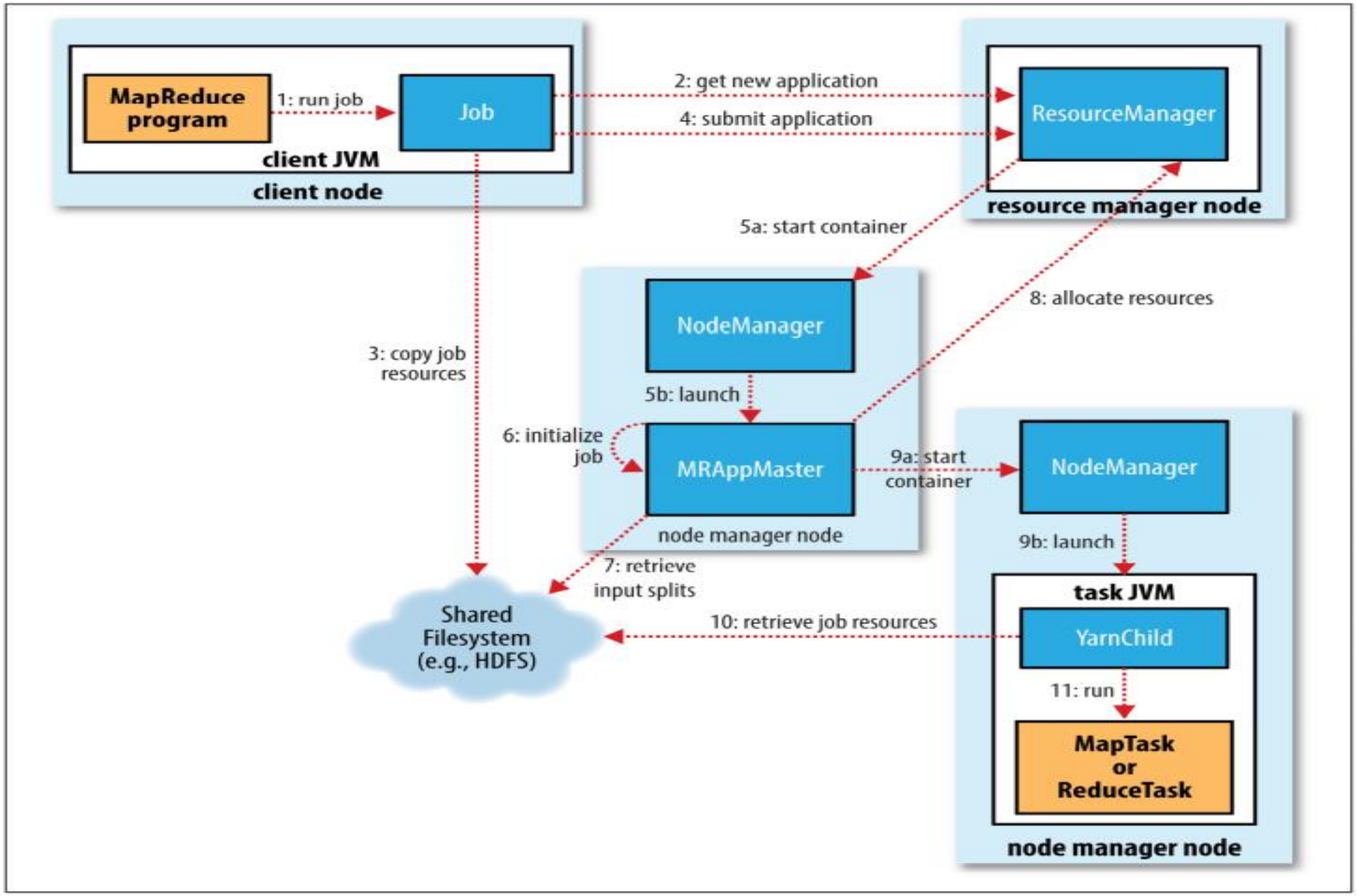


Figure 7-1. How Hadoop runs a MapReduce job

Job Submission :

- The **submit() method** on Job **creates an internal Job Submitter instance and calls submit Job Internal()** on it (step 1 in Figure 7-1).

- The job submission process implemented by Job Submitter does the following:

    - **Asks the resource manager for a new application ID**, used for the MapReduce job ID (step 2).

    - **Checks the output specification of the job.** For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.

    - **Computes the input splits for the job.** If the splits cannot be computed (because the input paths don't exist, for example), the job is not submitted and an error is thrown to the MapReduce program.

    - **Copies the resources needed to run the job, including the job JAR file**, the configuration file, and the **computed input splits, to the shared filesystem in a directory named after the job ID** (step 3).

    - **Submits the job** by calling submit Application() on the resource manager (step 4).

Job Initialization:

- When the resource manager receives a call to its **submitApplication() method**, it hands off the request to the YARN scheduler.
    - **The scheduler allocates a container, and the resource manager then launches the application master's process there**, under the node manager's management (steps 5a and 5b).
- The application master for MapReduce jobs is a Java application whose main class is MRAppMaster.
    - It **initializes the job by creating a number of bookkeeping objects to keep track of the job's progress**, as it will receive progress and completion reports from the tasks (step 6).
- Next, it retrieves the input splits computed in the client from the shared filesystem (step 7).

Task Assignment:

- If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the resource manager (step 8).

    - **Requests for map tasks** are made first and with a higher priority than those for reduce tasks, since all the **map tasks must complete before the sort phase of the reduce** can start (see "Shuffle and Sort" on page 197).

    - **Requests for reduce tasks** are not made until 5% of map tasks have completed (see "Reduce slow start" on page 308).

Task Execution:

- Once a **task has been assigned resources for a container on a particular node** by the resource manager's scheduler, the application master starts the container by contacting the node manager (steps 9a and 9b).

- **The task is executed by a Java application whose main class is YarnChild**. Before it can run the task, it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache (step 10; see "Distributed Cache" on page 274).

- Finally, it runs the map or reduce task (step 11).

Interacting with Hadoop ecosystem – Pig, Hive, HBase, Sqoop

Interacting with Hadoop ecosystem – Pig

Pig is a scripting platform that runs on Hadoop clusters designed to process and analyze large datasets.

- **Pig is a data flow system for Hadoop**. It uses Pig Latin to specify data flow. Pig is an alternative to MapReduce Programming.
- For Big Data Analytics, Pig gives a simple data flow language known as **Pig Latin** which has functionalities similar to SQL like join, filter, limit etc.

There are two major components of the Pig:

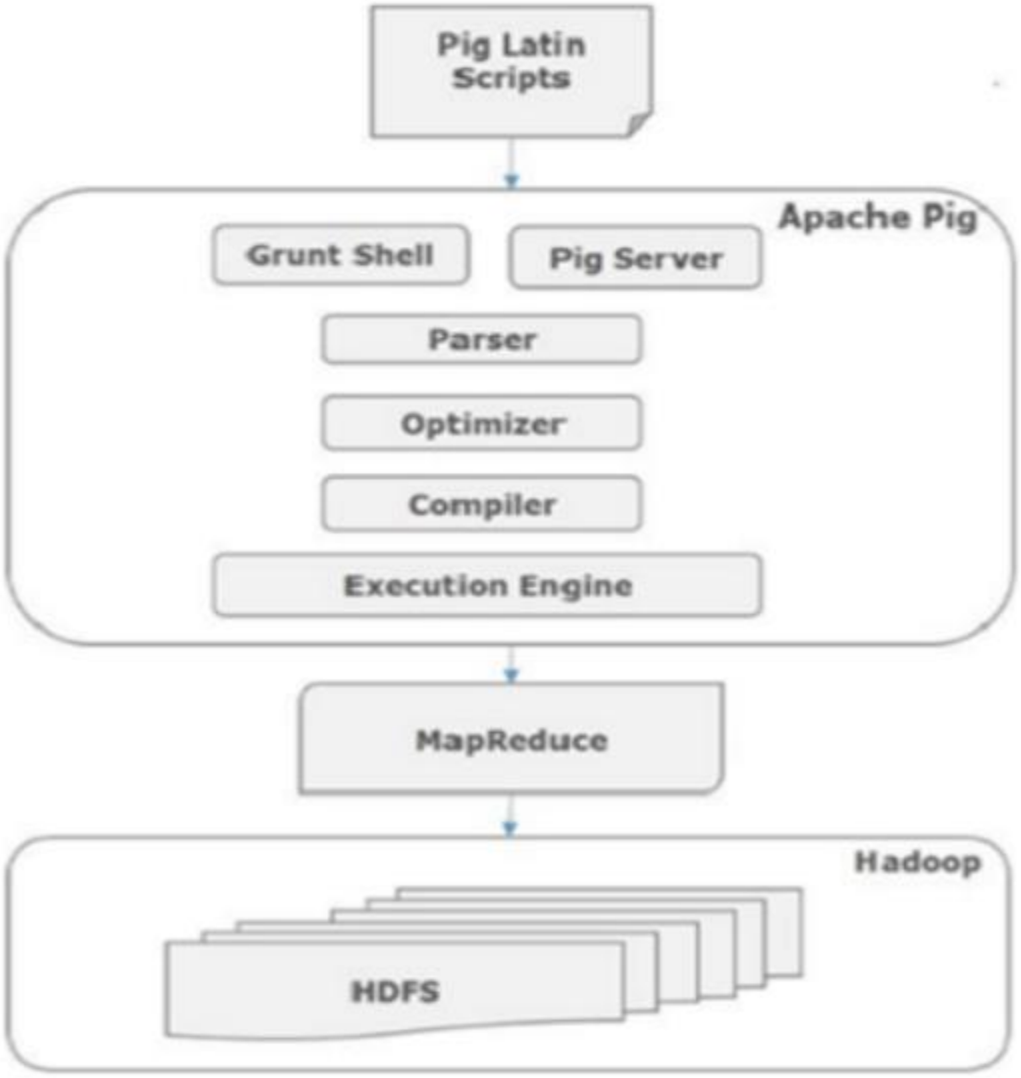
Pig Latin script language: Data Processing Language

- The Pig Latin script is a procedural data flow language. It contains syntax and commands that can be applied to implement business logic. Examples of Pig Latin are LOAD and STORE.

A runtime engine: Translate Pig Latin into MapReduce Programming

- The runtime engine is a compiler that produces sequences of MapReduce programs. It uses HDFS to store and retrieve data. It is also used to interact with the Hadoop system (HDFS and MapReduce).

Interacting with Hadoop ecosystem – Pig



Apache Pig vs MapReduce

•**Pig Latin is a high-level data flow language**, whereas MapReduce is a low-level data processing paradigm.

•Without writing complex Java implementations in MapReduce, programmers can achieve the same implementations very **easily using Pig Latin**.

•**Pig provides many built-in operators to support data operations** like joins, filters, ordering, sorting etc. Whereas to perform the same function in MapReduce is a humongous task.

•**Performing a Join operation in Apache Pig is simple.** Whereas it is difficult in MapReduce to perform a Join operation between the data sets, as it requires multiple MapReduce tasks to be executed sequentially to fulfill the job

Interacting with Hadoop ecosystem –Hive

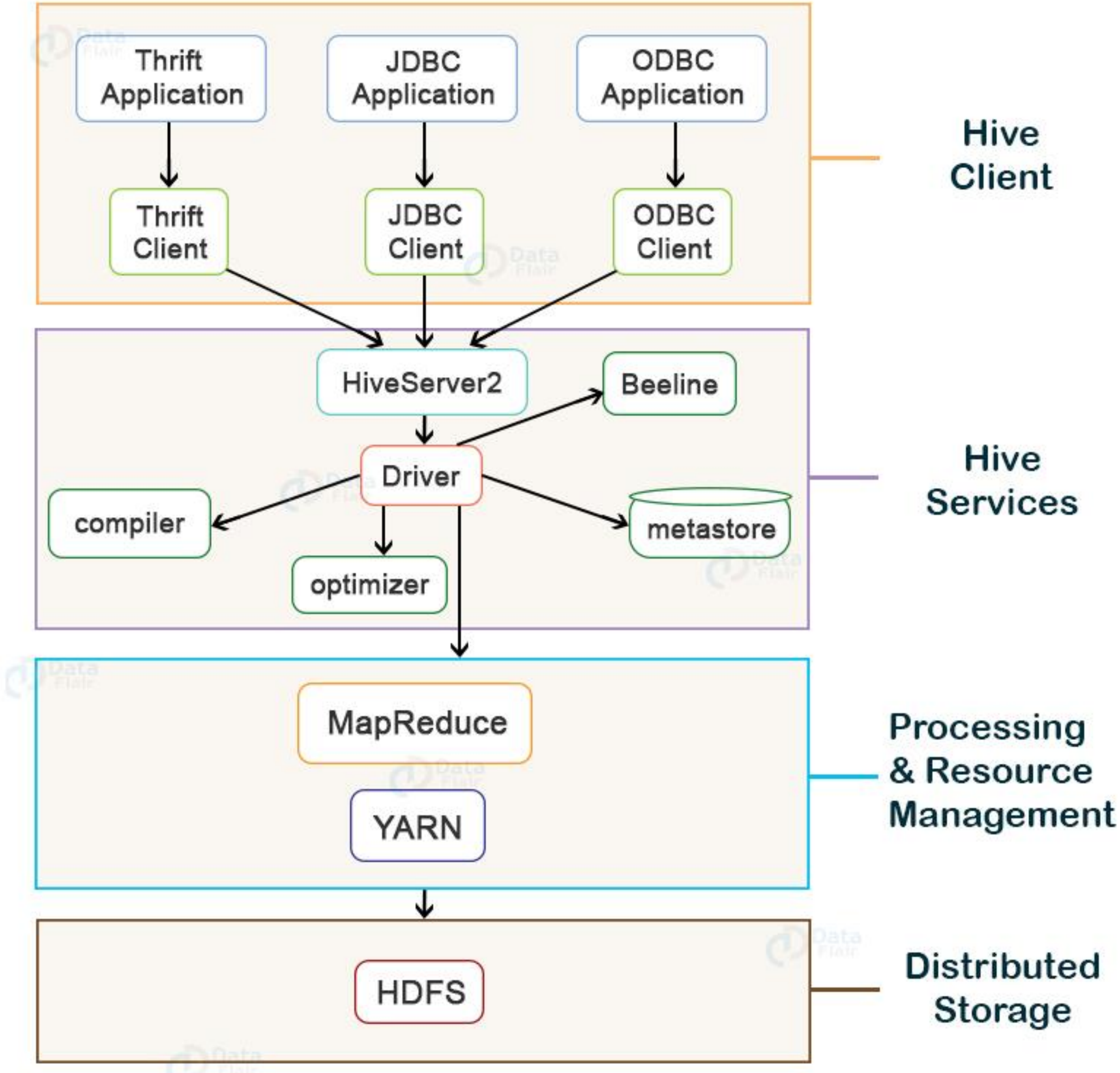
Apache Hive is an open-source **data warehousing tool** for performing distributed processing and data analysis. It was developed by Facebook to reduce the work of writing the Java MapReduce program.

Apache Hive uses a **Hive Query language**, which is a declarative language similar to SQL. Hive translates the hive queries into MapReduce programs.

It supports developers to perform **processing and analyses on structured and semi-structured data** by replacing complex java MapReduce programs with hive queries.

One who is familiar with SQL commands can easily write the hive queries. Hive makes the job easy for performing operations like

- Analysis of huge datasets
- Ad-hoc queries, Summerization
- Data encapsulation


Thrift Application
JDBC Application
ODBC Application
Thrift Client
JDBC Client
ODBC Client
Hive Client
HiveServer2
Beeline
Driver
compiler
optimizer
metastore
Hive Services
MapReduce
YARN
Processing & Resource Management
HDFS
Distributed Storage

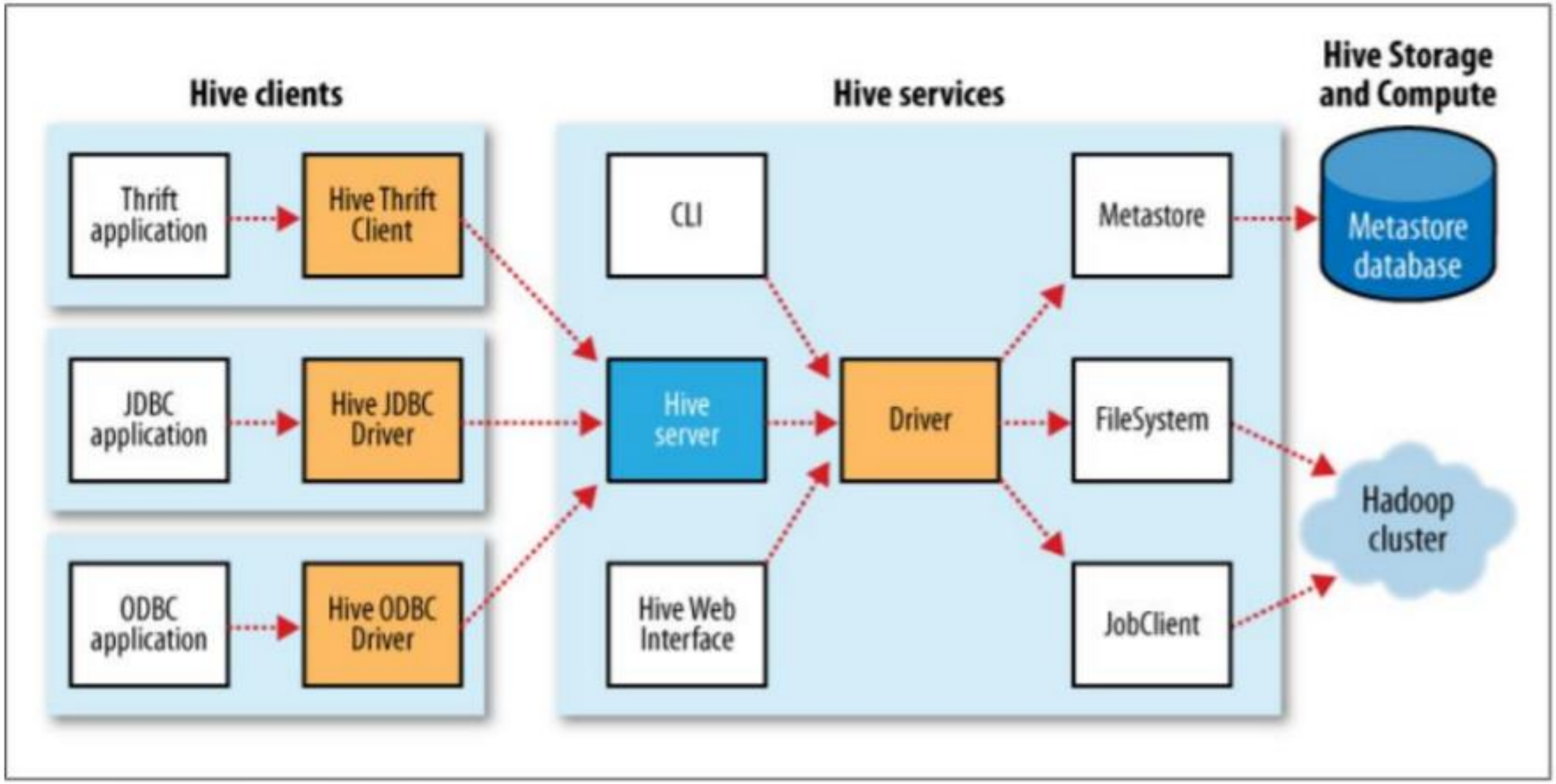
Interacting with Hadoop ecosystem –HBase

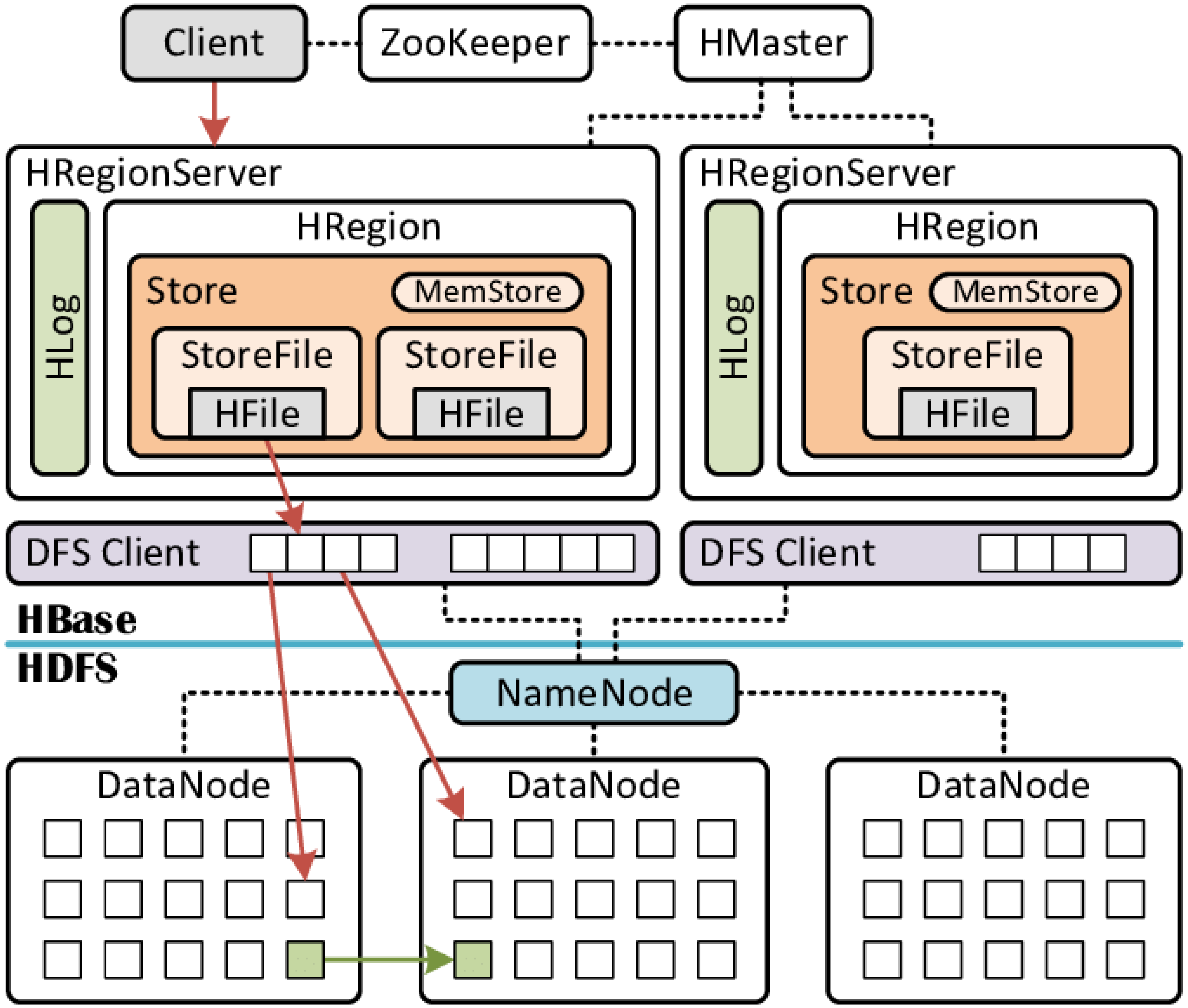
HBase is a **NoSQL database for Hadoop.** It is used to store billions of rows and millions of columns. HBase provides read/write operations

HBase is a **distributed column-oriented database** built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random-access to very large datasets.

HBase is a high-reliability, high-performance, **column-oriented, scalable distributed storage system** that uses HBase technology to build large-scale structured storage clusters on inexpensive PC Servers.

The goal of HBase is to **store and process large amounts of data**, specifically to handle large amounts of data consisting of thousands of rows and columns


Hive clients
Thrift application
Hive Thrift Client
JDBC application
Hive JDBC Driver
ODBC application
Hive ODBC Driver
Hive services
CLI
Hive server
Hive Web Interface
Driver
Metastore
FileSystem
JobClient
Hive Storage and Compute
Metastore database
Hadoop cluster

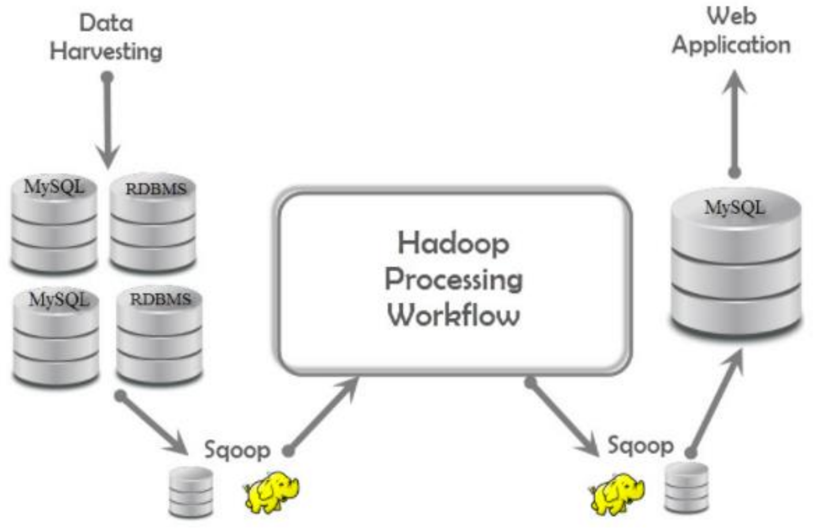

Client
ZooKeeper
HMaster
HRegionServer
HLog
HRegion
Store
MemStore
StoreFile
HFile
StoreFile
HFile
HRegionServer
HLog
HRegion
Store
MemStore
StoreFile
HFile
DFS Client
DFS Client
HBase
HDFS
NameNode
DataNode
DataNode
DataNode

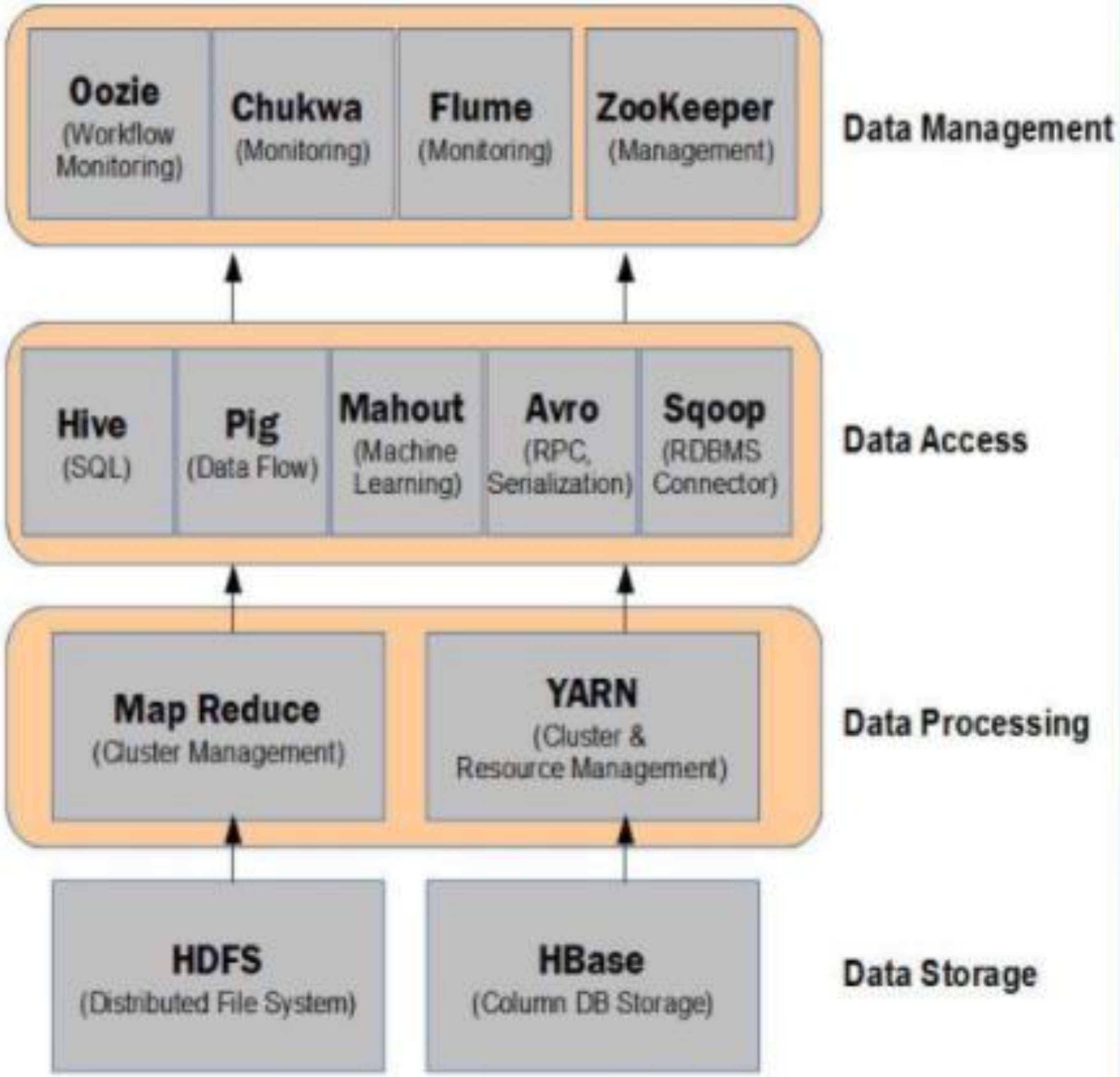
Interacting with Hadoop ecosystem –Sqoop

Sqoop is a tool designed to **transfer data between Hadoop and relational databases** or mainframes.

- You can use Sqoop to import data from a relational database management system (RDBMS) such as MySQL or Oracle or a mainframe into the Hadoop Distributed File System (HDFS), transform the data in Hadoop MapReduce, and then export the data back into an RDBMS.

- Sqoop automates most of this process, relying on the database to describe the schema for the data to be imported. Sqoop uses MapReduce to import and export the data, which provides parallel operation as well as fault tolerance.




Oozie (Workflow Monitoring)
Chukwa (Monitoring)
Flume (Monitoring)
ZooKeeper (Management)
Data Management
Hive (SQL)
Pig (Data Flow)
Mahout (Machine Learning)
Avro (RPC, Serialization)
Sqoop (RDBMS Connector)
Data Access
Map Reduce (Cluster Management)
YARN (Cluster & Resource Management)
Data Processing
HDFS (Distributed File System)
HBase (Column DB Storage)
Data Storage