UNIT - 4

Introduction in Microeconomics (Jadavpur University)

UNIT IV: PROGRAMMING MODEL

Open source grid middleware packages – Globus Toolkit (GT4) Architecture , Configuration – Usage of Globus – Main components and Programming model - Introduction to Hadoop Framework - Mapreduce, Input splitting, map and reduce functions, specifying input and output parameters, configuring and running a job – Design of Hadoop file system, HDFS concepts, command line and java interface, dataflow of File read & File write.

Open Source Grid Middleware Packages

- The Open Grid Forum and Object Management are two well- formed organizations behind the standards
- Middleware is the software layer that connects software components. It lies between operating system and the applications.
- Grid middleware is specially designed a layer between hardware and software, enable the sharing of heterogeneous resources and managing virtual organizations created around the grid.
- The popular grid middleware are
    1. BOINC -Berkeley Open Infrastructure for Network Computing.
    2. UNICORE - Middleware developed by the German grid computing community.
    3. Globus (GT4) - A middleware library jointly developed by Argonne National Lab., Univ. of Chicago, and USC Information Science Institute, funded by DARPA, NSF, and NIH.
    4. CGSP in ChinaGrid - The CGSP (ChinaGrid Support Platform) is a middleware library developed by 20 top universities in China as part of the ChinaGrid Project
    5. Condor-G - Originally developed at the Univ. of Wisconsin for general distributed computing, and later extended to Condor-G for grid job management.
    6. Sun Grid Engine (SGE) - Developed by Sun Microsystems for business grid applications. Applied to private grids and local clusters within enterprises or campuses.
    7. gLight -Born from the collaborative efforts of more than 80 people in 12 different academic and industrial research centers as part of the EGEE Project, gLite provided a framework for building grid applications tapping into the power of distributed computing and storage resources across the Internet.

Globus Toolkit Architecture (GT4)

- The Globus Toolkit, is an open middleware library for the grid computing communities. These open source software libraries support many operational grids and their applications on an international basis.
- The toolkit addresses common problems and issues related to grid resource discovery, management, communication, security, fault detection, and portability. The software itself provides a variety of components and capabilities.
- The library includes a rich set of service implementations. The implemented software supports grid infrastructure management, provides tools for building new web services in Java , C, and Python, builds a powerful standard-based security infrastructure and client

API s (in different languages), and offers comprehensive command-line programs for accessing various grid services.

- Security infrastructure and client APIs (in different languages), and offers comprehensive command-line programs for accessing various grid services.
- The Globus Toolkit was initially motivated by a desire to remove obstacles that prevent seamless collaboration, and thus sharing of resources and services, in scientific and engineering applications. The shared resources can be computers, storage, data, services, networks, science instruments (e.g., sensors), and so on. The Globus library version GT4, is conceptually shown in Figure

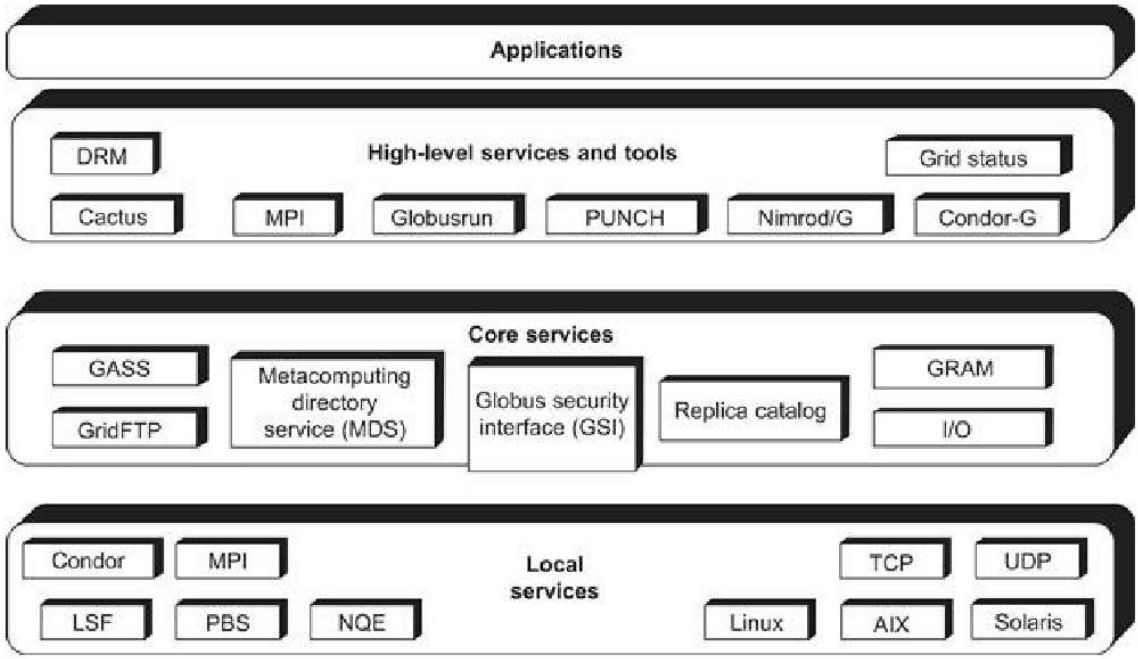


Figure: Globus Toolkit GT4 supports distributed and cluster computing services

The GT4 Library

- GT4 offers the middle-level core services in grid applications.
- The high-level services and tools, such as MPI , Condor-G, and Nirod/G, are developed by third parties for general purpose distributed computing applications.
- The local services, such as LSF, TCP, Linux, and Condor, are at the boom level and are fundamental tools supplied by other developers.
- As a de facto standard in grid middleware, GT4 is based on industry-standard web service technologies.

Functionalities of GT4

- Global Resource Allocation Manager (GRAM) - Grid Resource Access and Management (HTTP-based)

- Communication (Nexus) - Unicast and multicast communication
- Grid Security Infrastructure (GSI) - Authentication and related security services
- Monitory and Discovery Service (MDS) - Distributed access to structure and state information
- Health and Status (HBM) - Heartbeat monitoring of system components
- Global Access of Secondary Storage (GASS) - Grid access of data in remote secondary storage
- Grid File Transfer (GridFTP) Inter-node fast file transfer

Globus Job Workflow

- A typical job execution sequence proceeds as follows: The user delegates his credentials to a delegation service.
- The user submits a job request to GRAM with the delegation identifier as a parameter.
- GRAM parses the request, retrieves the user proxy certificate from the delegation service, and then acts on behalf of the user.
- GRAM sends a transfer request to the RFT (Reliable File Transfer), which applies GridFTP to bring in the necessary files.
- GRAM invokes a local scheduler via a GRAM adapter and the SEG (Scheduler Event
- Generator) initiates a set of user j obs.
- The local scheduler reports the j ob state to the SEG. Once the j ob is complete, GRAM uses RFT and GridFTP to stage out the resultant files. The grid monitors the progress of these operations and sends the user a notification

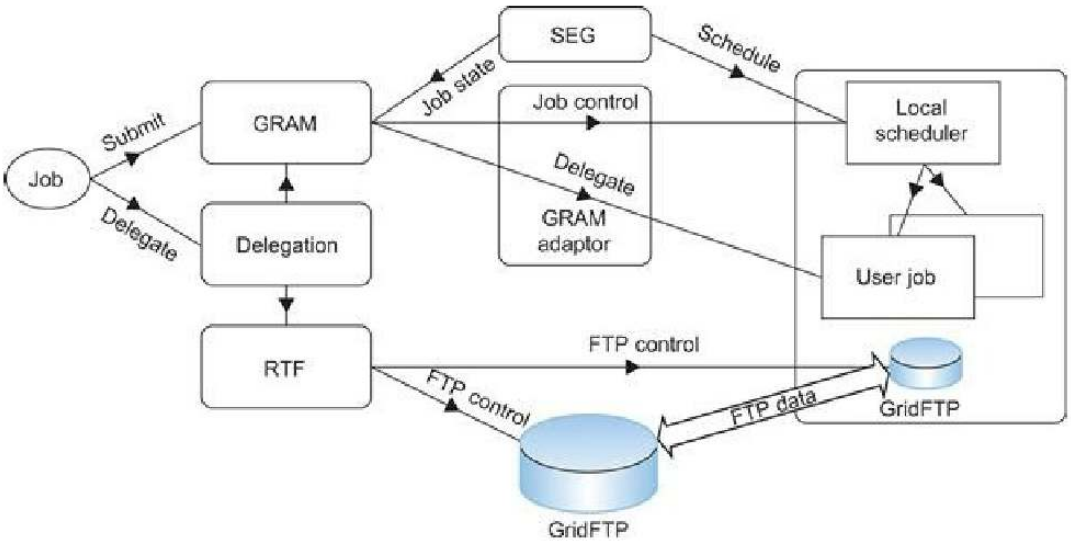


Figure: Globus job workflow among interactive functional modules.

Client-Globus Interactions

- There are strong interactions between provider programs and user code. GT4 makes heavy use of industry-standard web service protocols and mechanisms in service Description, discovery, access, authentication, authorization, and the like.
- GT4 makes extensive use of Java, C, and Python to write user code.
- Web service mechanisms define specific interfaces for grid computing. Web services provide flexible, extensible, and widely adopted XML-based interfaces.
- These demand computational, communication, data, and storage resources. We must enable a range of end-user tools that provide the higher-level capabilities needed in

specific user applications. Developers can use these services and libraries to build simple and complex systems quickly.

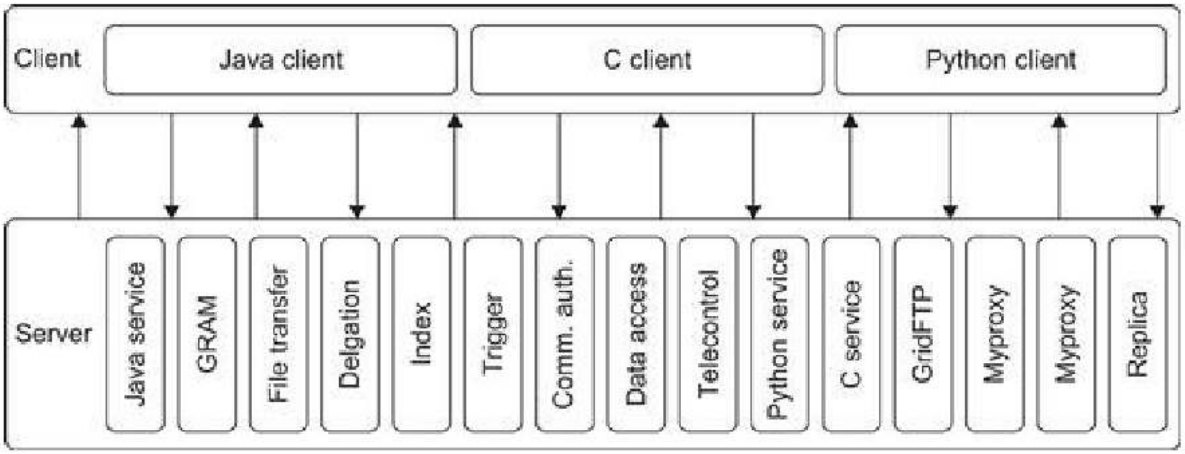
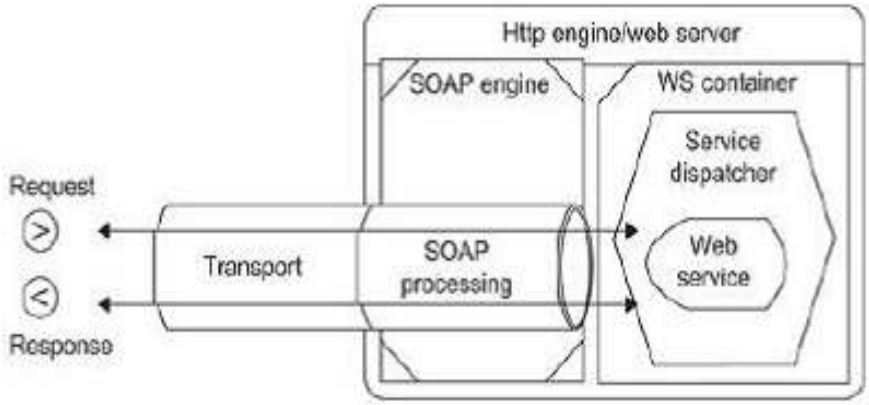


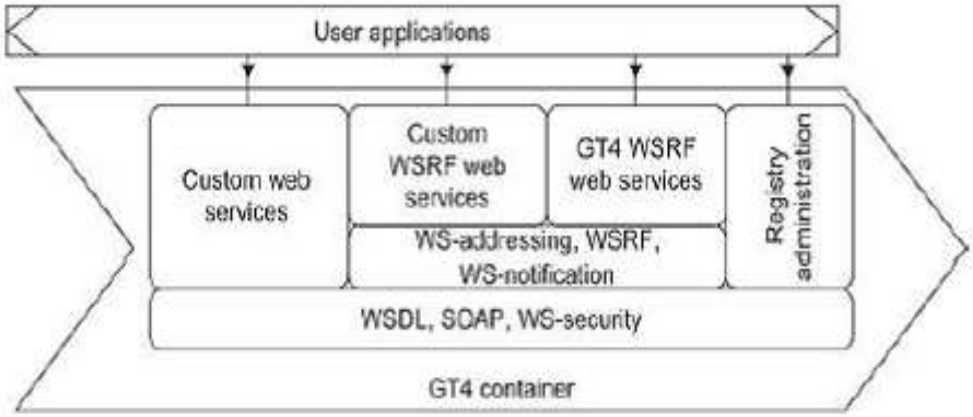
Figure: Client and GT4 server interactions; vertical boxes correspond to service programs and horizontal boxes represent the user codes.

The horizontal boxes in the client domain denote custom applications and/or third-party tools that access GT4 services. The toolkit programs provide a set of useful infrastructure services.

Three containers are used to host user-developed services written in Java, Python, and C, respectively. These containers provide implementations of security, management, discovery, state management, and other mechanisms frequently required when building services.



(a) The globus container



Data Management Using GT4

- Grid applications one need to provide access to and/or integrate large quantities of data at multiple sites. The GT4 tools can be used individually or in conj unction with other tools to develop interesting solutions to efficient data access. The following list briefly introduces these GT4 tools:

  1. Grid FTP supports reliable, secure, and fast memory-to-memory and disk-to-disk data movement over high-bandwidth WANs. Based on the popular FTP protocol for internet file transfer, Grid FTP adds additional features such as parallel data transfer, third-party data transfer, and striped data transfer. I n addition, Grid FTP benefits from using the strong Globus Security Infra structure for securing data channels with authentication and reusability. It has been reported that the grid has achieved 27 Gbit/second end-to-end transfer speeds over some WANs.
  2. RFT provides reliable management of multiple Grid FTP transfers. I t has been used to orchestrate the transfer of millions of files among many sites simultaneously.
  3. RLS (Replica Location Service) is a scalable system for maintaining and providing access to information about the location of replicated files and data sets.
  4. OGSA-DAI (Globus Data Access and Integration) tools were developed by the UK eScience program and provide access to relational and XML databases.

MapReduce Model

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster

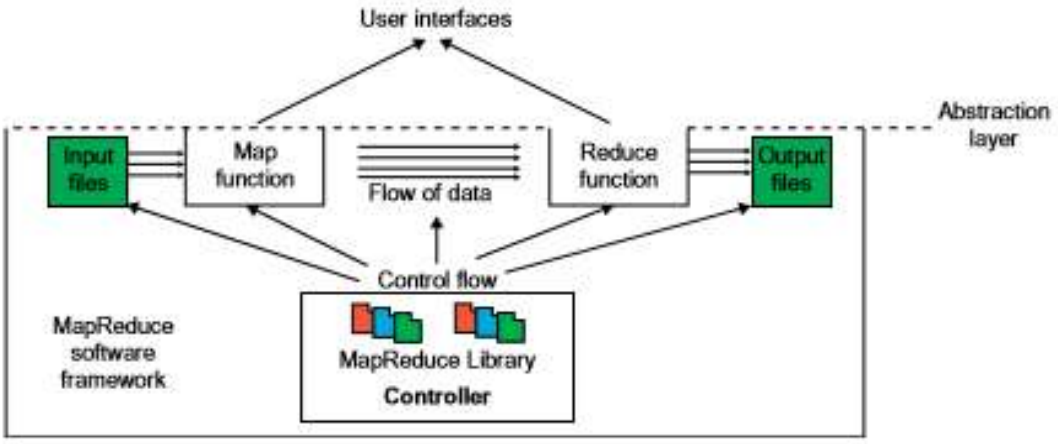
The model is based on two distinct steps for an application:

• ***Map***: An initial ingestion and transformation step, in which individual input records can be processed in parallel.

• ***Reduce***: An aggregation or summarization step, in which all associated records must be processed together by a single entity.

The core concept of MapReduce in Hadoop is that input may be split into logical chunks, and each chunk may be initially processed independently, by a map task. The results of these individual processing chunks can be physically partitioned into distinct sets, which are then sorted. Each sorted chunk is passed to a reduce task.

A map task may run on any compute node in the cluster, and multiple map tasks may be running in parallel across the cluster. The map task is responsible for transforming the input records into key/value pairs. The output of all of the maps will be partitioned, and each partition will be sorted. There will be one partition for each reduce task. Each partition's sorted keys and the values associated with the keys are then processed by the reduce task. There may be multiple reduce tasks running in parallel on the cluster.



FIGURE

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.

Formal Definition of MapReduce

The MapReduce software framework provides an abstraction layer with the data flow and flow of control to users, and hides the implementation of all data flow steps such as data partitioning, mapping, synchronization, communication, and scheduling. Here, although the data flow in such frameworks is predefined, the abstraction layer provides two well-defined interfaces in the form of two functions: Map and Reduce .

These two main functions can be overridden by the user to achieve specific objectives. Figure 6.1 shows the MapReduce framework with data flow and control flow. Therefore, the user overrides the Map and Reduce functions first and then invokes the provided MapReduce (Spec, & Results) function from the library to start the flow of data.

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

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

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

Formal Notation of MapReduce Data Flow

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

$$(key_1, val_1) \xrightarrow{Map\ Function} \text{List}\,(key_2, val_2)$$

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

$$(key_2, \text{List}\,(val_2)) \xrightarrow{Reduce\ Function} \text{List}\,(val_2)$$

MapReduce Logical Data Flow

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

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

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

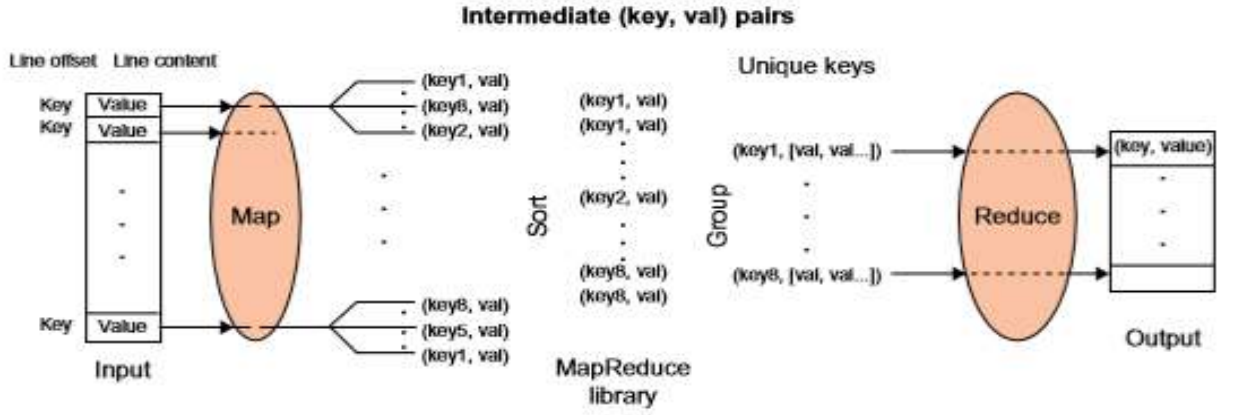


FIGURE 6.2

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

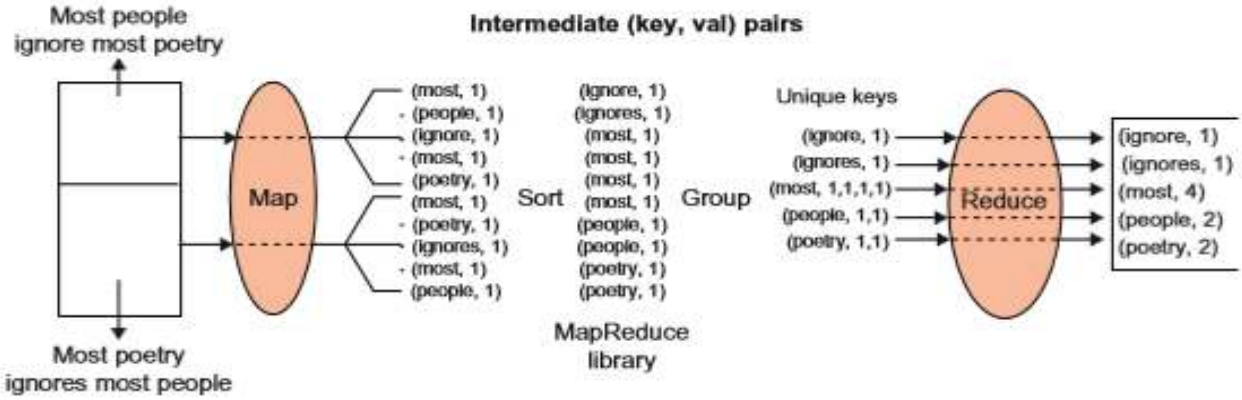


FIGURE 6.3

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

MapReduce Actual Data and Control Flow

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

1. Data partitioning

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

2. Computation partitioning

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

3. Determining the master and workers

The MapReduce architecture is based on a master- worker model. Therefore, one of the copies of the user program becomes the master and the rest become workers. The master picks idle workers, and assigns the map and reduce tasks to them. A map/reduce worker is typically a

computation engine such as a cluster node to run map/ reduce tasks by executing Map/Reduce functions. Steps 4–7 describe the map workers.

4. Reading the input data (data distribution)

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

5. Map function

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

6. Combiner function

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

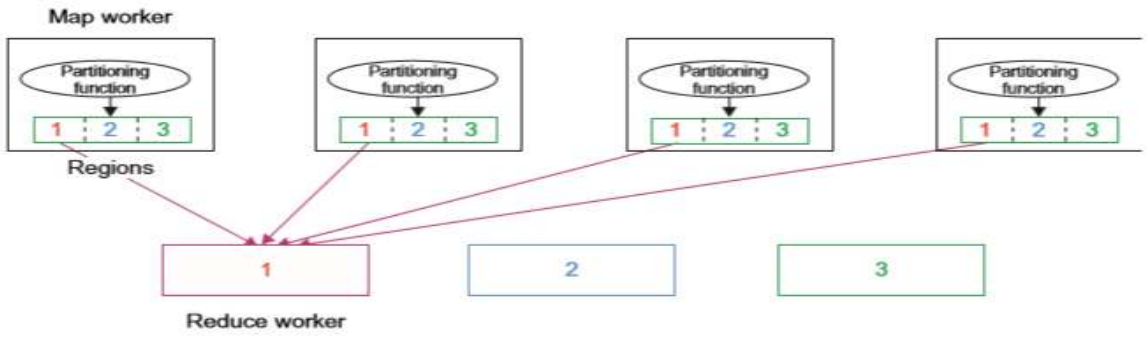


FIGURE 6.4

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

7. Partitioning function

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

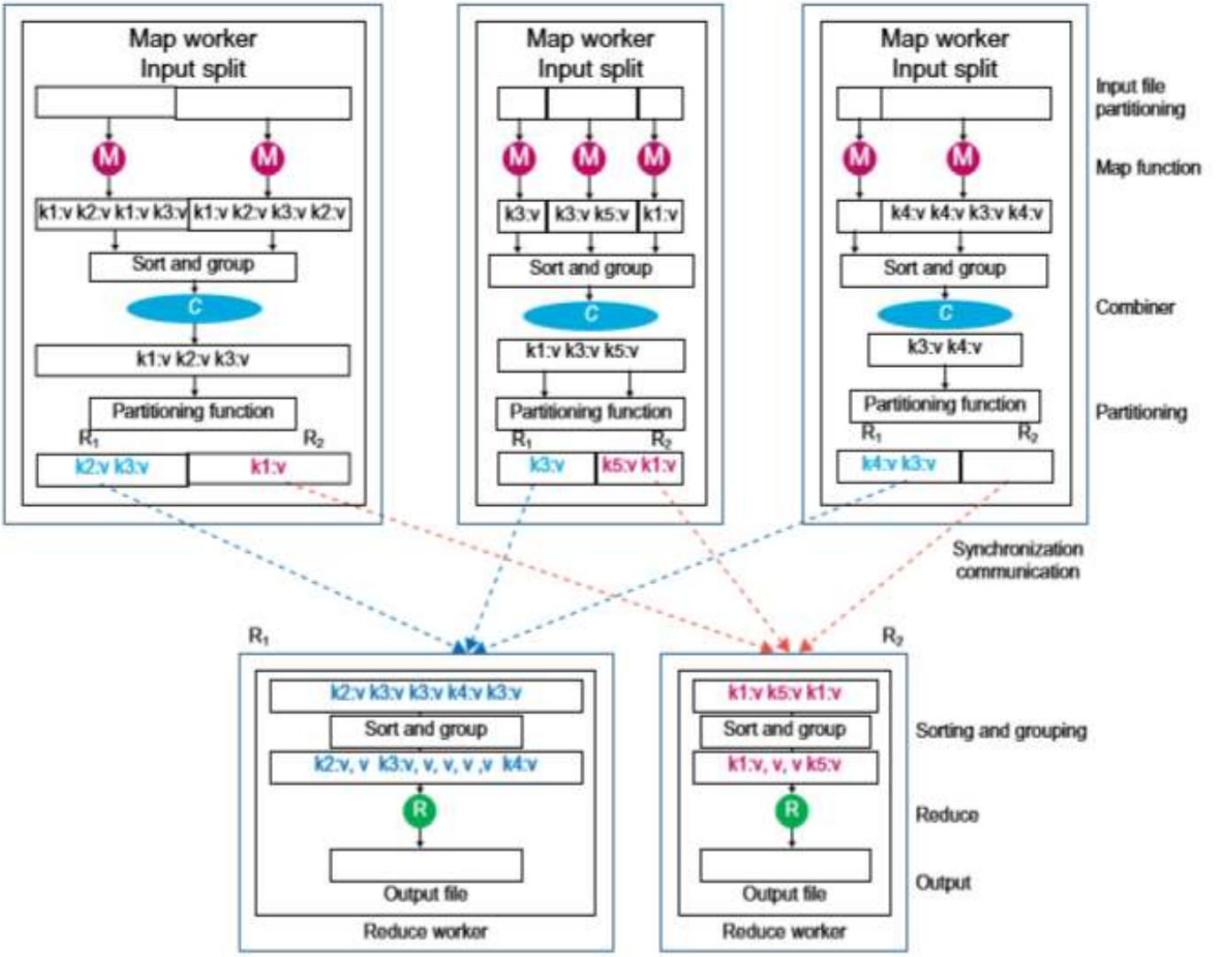


FIGURE 6.5

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.

8. Synchronization

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

9. Communication

Reduce worker i, already notified of the location of region i of all map workers, uses a remote procedure call to read the data from the respective region of all map workers. Since all reduce workers read the data from all map workers, all-to-all communication among all map and reduce workers, which incurs network congestion, occurs in the network. This issue is one of the major bottlenecks in increasing the performance of such systems. A data transfer module was proposed to schedule data transfers independently. Steps 10 and 11 correspond to the reduce worker domain:

10. Sorting and Grouping

When the process of reading the input data is finalized by a reduce worker, the data is initially buffered in the local disk of the reduce worker. Then the reduce worker groups

intermediate (key, value) pairs by sorting the data based on their keys, followed by grouping all occurrences of identical keys. Note that the buffered data is sorted and grouped because the number of unique keys produced by a map worker may be more than R regions in which more than one key exists in each region of a map worker (see Figure 6.4).

11. Reduce function

The reduce worker iterates over the grouped (key, value) pairs, and for each unique key, it sends the key and corresponding values to the Reduce function. Then this function processes its input data and stores the output results in predetermined files in the user's program

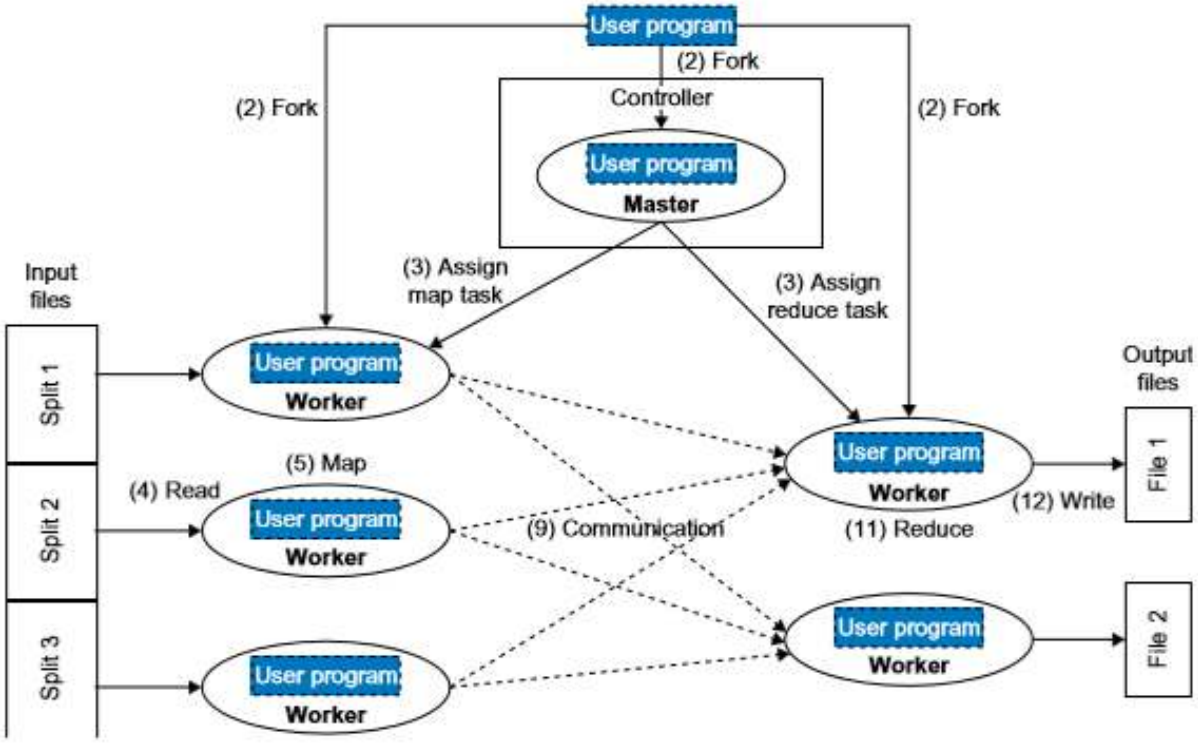


FIGURE 6.6

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.

Introducing Hadoop

Hadoop is the Apache Software Foundation top-level project that holds the various Hadoop subprojects that graduated from the Apache Incubator. The Hadoop project provides and supports the development of open source software that supplies a framework for the development of highly scalable distributed computing applications. The Hadoop framework handles the processing details, leaving developers free to focus on application logic.

How Hadoop Works:

- Data is split into small blocks of 64 or 128MB and stored onto a minimum of 3 machines at a time to ensure data availability & reliability
- Many machines are connected in a cluster work in parallel for faster crunching of data
- If any one machine fails, the work is assigned to another automatically
- MapReduce breaks complex tasks into smaller chunks to be executed in parallel
- Hive – SQL like interface
- Pig – data management language, like commercial tools AbInitio, Informatica,
- Hbase – column oriented database on top of HDFS
- Flume – real time data streaming such as credit card transaction, videos

- Sqoop – SQL interface to RDBMS and HDFS
- Zookeeper – a DBA management for Hadoop

a. Hadoop Distributed File System

HDFS is the primary storage system of Hadoop. Hadoop distributed file system (HDFS) is java based file system that provides scalable, fault tolerance, reliable and cost efficient data storage for big data. HDFS is a distributed filesystem that runson commodity hardware. HDFS is already configured with default configuration for many installations. Most of the time for large clusters configuration is needed. Hadoop interact directly with HDFS by shell-like commands.

Components of HDFS:

i. NameNode

It is also known as Master node. NameNode does not store actual data or dataset. NameNode stores Metadata i.e. number of blocks, their location, on which Rack, which Datanode the data is stored and other details. It consists of files and directories.

Tasks of NameNode

- Manage file system namespace.
- Regulates client's access to files.
- Executes file system execution such as naming, closing, opening files and directories.

ii. DataNode

It is also known as Slave. HDFS Datanode is responsible for storing actual data in HDFS. Datanode performs read and write operation as per the request of the clients. Replica block of Datanode consists of 2 files on the file system. The first file is for data and second file is for recording the block's metadata. HDFS Metadata includes checksums for data. At startup, each Datanode connects to its corresponding Namenode and does handshaking. Verification of namespace ID and software version of DataNode take place by handshaking. At the time of mismatch found, DataNode goes down automatically.

Tasks of DataNode

- DataNode performs operations like block replica creation, deletion and replication according to the instruction of NameNode.
- DataNode manages data storage of the system.

b. MapReduce

Hadoop MapReduce is the core component of hadoop which provides data processing. MapReduce is a software framework for easily writing applications that process the vast amount of structured and unstructured data stored in the Hadoop Distributed File system.

Hadoop MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster. By this parallel processing, speed and reliability of cluster is improved.

Working of MapReduce

MapReduce works by breaking the processing into two phases:

- Map phase
- Reduce phase

Each phase has key-value pairs as input and output. In addition, programmer also specifies two functions: **map function** and**reduce function**

Map function takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs).

Reduce function takes the output from the Map as an input and combines those data tuples based on the key and accordingly modifies the value of the key.

Features of MapReduce

i. Simplicity

MapReduce jobs are easy to run. Applications can be written in any language such as java, C++, and python.

ii. Scalability

MapReduce can process petabytes of data.

iii. Speed

By means of parallel processing problems that take days to solve, it is solved in hours and minutes by MapReduce.

iv. Fault tolerance

MapReduce takes care of failures. If one copy of data is unavailable, another machine has a copy of the same key pair which can be used for solving the same subtask.

c. YARN

YARN provides the resource management. YARN is called as the operating system of hadoop as it is responsible for managing and monitoring workloads. It allows multiple data processing engines such as real-time streaming and batch processing to handle data stored on a single platform.

YARN has been projected as a data operating system for Hadoop2. Main features of YARN are:

- **Flexibility:** Enables other purpose-built data processing models beyond MapReduce (batch), such as interactive and streaming. Due to this feature of YARN, other applications can also be run along with Map Reduce programs in hadoop2.

- **Efficiency –** As many applications can be run on same cluster, efficiency of Hadoop increases without much effect on quality of service.
- **Shared –** Provides a stable, reliable, secure foundation and shared operational services across multiple workloads. Additional programming models such as graph processing and iterative modeling are now possible for data processing.

Refer YARN Comprehensive Guide for more details.

d. Hive

Hive is an open source data warehouse system for querying and analyzing large datasets stored in hadoop files. Hive do three main functions: data summarization, query, and analysis.

Hive use language called **HiveQL** (HQL), which is similar to SQL. HiveQL automatically translates SQL-like queries into MapReduce jobs which will execute on hadoop.

Main parts of Hive are:

- **Metastore**: metadata is stored in
- **Driver**: manage the lifecycle of a HiveQL statement.
- **Query complier**: compiles HiveQL into directed acyclic graph.
- **Hive server**: provide a thrift interface and JDBC/ODBC server.

Refer Hive Comprehensive Guide for more details.

e. Pig

Pig is a high- level language platform for analyzing and querying huge dataset that are stored in HDFS. Language used in Pig is called PigLatin. It is very similar to SQL. It is used to load the data, apply the required filters and dump the data in the required format. For Programs execution, pig requires Java runtime environment.

Features of Apache Pig:

- **Extensibility: F**or carrying out special purpose processing, users are allowed to create their own function.
- **Optimization opportunities**: Pig allows the system to optimize automatic execution. This allows the user to pay attention to semantics instead of efficiency.
- **Handles all kinds of data:** Pig analyzes both structured as well as unstructured.

f. Hbase

It is distributed database that was designed to store structured data in tables that could have billions of row and millions of columns. Hbase is scalable, distributed, and Nosql database that is built on top of HDFS. Hbase provide real time access to read or write data in HDFS.

Components of Hbase

i. Hbase master

It is not part of the actual data storage but negotiates load balancing across all RegionServer.

- Maintain and monitor the hadoop cluster.
- Performs administration (interface for creating, updating and deleting tables.)
- Controls the failover.
- HMaster handles DDL operation.

ii. RegionServer

It is the worker node which handle read, write, update and delete requests from clients. Region server process runs on every node in hadoop cluster. Region server runs on HDFS DateNode.

g. HCatalog

HCatalog is a table and storage management layer for hadoop. HCatalog supports different components available in hadoop like MapReduce, hive and pig to easily read and write data from the cluster. HCatalog is a key component of Hive that enables the user to store their data in any format and structure.

By default, HCatalog supports RCFile, CSV, JSON, sequenceFile and ORC file formats.

Benefits of HCatalog:

- Enables notifications of data availability.
- With the table abstraction, HCatalog frees the user from overhead of data storage.
- Provide visibility for data cleaning and archiving tools.

h. Avro

Avro is a most popular Data serialization system. Avro is an open source project that provides data serialization and data exchange services for hadoop. These services can be used together or independently. Big data can exchange programs written in different languages using Avro.

Using serialization service programs can serialize data into files or messages. Avro stores data definition and data together in one message or file making it easy for programs to dynamically understand information stored in Avro file or message.

Avro schema: Avro relies on schemas for serialization/deserialization. Avro requires schema when data is written or read. When Avro data is stored in a file its schema is stored with it, so that files may be processed later by any program.

Dynamic typing: It refers to serialization and deserialization without code generation. It complements the code generation which is available in Avro for statically typed language as an optional optimization.

Avro provides:

- Rich data structures.
- Remote procedure call.
- Compact, fast, binary data format.
- Container file, to store persistent data.

i. Thrift

It is a software framework for scalable cross-language services development. Thrift is an interface definition language used for RPC communication. Hadoop does a lot of RPC calls so there is a possibility of using Apache Thrift for performance or other reasons.

j. Apache Drill

The main purpose of the drill is large-scale data processing including structured and semi-structured data. It is a low latency distributed query engine that is designed to scale to several thousands of nodes and query petabytes of data. The drill is the first distributed SQL query engine that has a schema-free model.

Application of Apache drill

The drill has become an invaluable tool at cardlytics, a company that provides consumer purchase data for mobile and internet banking. Cardlytics is using a drill to quickly process trillions of record and execute queries.

Features of Apache Drill:

The drill has specialized memory management system to eliminates garbage collection and optimize memory allocation and usage. Drill plays well with Hive by allowing developers to reuse their existing Hive deployment.

- **Extensibility:** Drill provides an extensible architecture at all layers, including query layer, query optimization, and client API. We can extend any layer for the specific need of an organization.
- **Flexibility:** Drill provides a hierarchical columnar data model that can represent complex, highly dynamic data and allow efficient processing.
- **Dynamic schema discovery:** Apache drill does not require schema or type specification for data in order to start the query execution process. Instead, drill starts processing the data in units called record batches and discover schema on the fly during processing.

- **Drill decentralized metadata:** unlike other SQL Hadoop technologies, the drill does not have centralized metadata requirement. Drill users do not need to create and manage tables in metadata in order to query data.

k. Apache Mahout

Mahout is open source framework that is primarily used for creating scalable machine learning algorithm and data mining library. Once data is stored in Hadoop HDFS, mahout provides the data science tools to automatically find meaningful patterns in those big data sets.

Algorithms of Mahout are:

- **Clustering**: Here it takes the item in particular class and organizes them into naturally occurring groups, such that item belonging to the same group are similar to each other.
- **Collaborative filtering**: It mines user behavior and makes product recommendations (e.g. amazon recommendations)
- **Classifications**: It learns from existing categorization and then assigns unclassified items to the best category.
- **Frequent pattern mining**: It analyzes items in a group (e.g. items in a shopping cart or terms in query session) and then identifies which items typically appear together.

l. Apache Sqoop

Sqoop is used for importing data from external sources into related hadoop components like HDFS, Hbase or Hive. It is also used for exporting data from hadoop to other external sources. Sqoop works with relational databases such as teradata, Netezza, oracle, MySQL.

Features of Apache Sqoop:

- **Import sequential datasets from mainframe**: Sqoop satisfies the growing need to move data from the mainframe to HDFS.
- **Import direct to ORC files**: Improves compression and light weight indexing and improve query performance.
- **Parallel data transfer**: For faster performance and optimal system utilization.
- **Efficient data analysis**: Improve efficiency of data analysis by combining structured data and unstructured data on a schema on reading data lake.
- **Fast data copies**: from an external system into hadoop.

m. Apache Flume

Flume is used for efficiently collecting, aggregating and moving a large amount of data from its origin and sending it back to HDFS. Flume is fault tolerant and reliable mechanism. Flume was created to allow flow data from the source into Hadoop environment. It uses a simple extensible data model that allows for the online analytic application. Using Flume, we can get the data from multiple servers immediately into hadoop.

n. Ambari

Ambari is a management platform for provisioning, managing, monitoring and securing apache Hadoop cluster. Hadoop management gets simpler as Ambari provide consistent, secure platform for operational control.

Features of Ambari:

- **Simplified installation, configuration, and management:** Ambari easily and efficiently create and manage clusters at scale.
- **Centralized security setup**: Ambari reduce the complexity to administer and configure cluster security across the entire platform.
- **Highly extensible and customizable**: Ambari is highly extensible for bringing custom services under management.
- **Full visibility into cluster health**: Ambari ensures that the cluster is healthy and available with a holistic approach to monitoring.

o. Zookeeper

Apache Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. Zookeeper is used to manage and coordinate a large cluster of machines.

Features of zookeeper:

- **Fast**: zookeeper is fast with workloads where reads to data are more common than writes. The ideal read/write ratio is 10:1.
- **Ordered**: zookeeper maintains a record of all transactions, which can be used for high-level

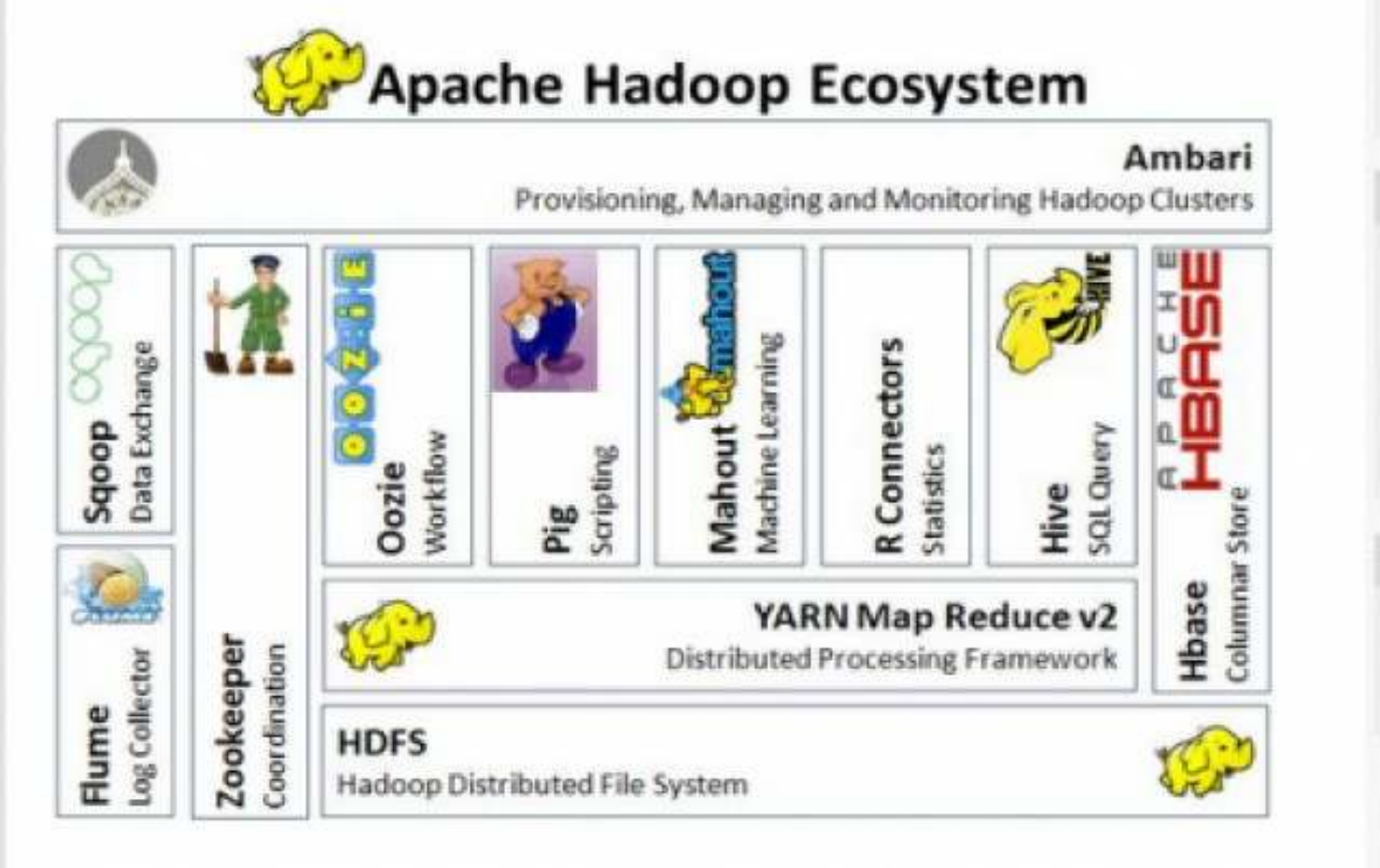
p. Oozie

It is a workflow scheduler system for managing apache hadoop jobs. Oozie combines multiple jobs sequentially into one logical unit of work. Oozie framework is fully integrated with apache hadoop stack, YARN as an architecture center and supports hadoop jobs for apache MapReduce, pig, Hive, and Sqoop.

In Oozie, users are permitted to create Directed Acyclic Graph of workflow, which can run in parallel and sequentially in hadoop. Oozie is scalable and can manage timely execution of thousands of workflow in a hadoop cluster. Oozie is very much flexible as well. One can easily start, stop, suspend and rerun jobs. It is even possible to skip a specific failed node or rerun it in Oozie.

There are two basic types of Oozie jobs:

- **Oozie workflow:** It is to store and run workflows composed of hadoop jobs e.g., MapReduce, pig, Hive.
- **Oozie coordinator**: It runs workflow jobs based on predefined schedules and availability of data.



Map & Reduce function

A Simple Map Function: IdentityMapper

The Hadoop framework provides a very simple map function, called IdentityMapper. It is used in jobs that only need to reduce the input, and not transform the raw input. All map functions must implement the Mapper interface, which guarantees that the map function will always be called with a key. The key is an instance of a WritableComparable object, a value that is an instance of a Writable object, an output object, and a reporter.

IdentityMapper.java

```
package org.apache.hadoop.mapred.lib;
import java.io.IOException;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;
/** Implements the identity function, mapping inputs directly to outputs. */
public class IdentityMapper<K, V>
extends MapReduceBase implements Mapper<K, V, K, V> {
/** The identify function. Input key/value pair is written directly to
* output.*/
public void map(K key, V val,
OutputCollector<K, V> output, Reporter reporter)
throws IOException {
output.collect(key, val);
}
}
```

A Simple Reduce Function: IdentityReducer

The Hadoop framework calls the reduce function one time for each unique key. The framework provides the key and the set of values that share that key.

IdentityReducer.java

```
package org.apache.hadoop.mapred.lib;
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;
/** Performs no reduction, writing all input values directly to the output. */
public class IdentityReducer<K, V>
extends MapReduceBase implements Reducer<K, V, K, V> {
```

■ THE BASICS OF A MAPREDUCE JOB

```
/** Writes all keys and values directly to output. */
public void reduce(K key, Iterator<V> values,
OutputCollector<K, V> output, Reporter reporter)
throws IOException {
while (values.hasNext()) {
output.collect(key, values.next());
}
}
```

If you require the output of your job to be sorted, the reducer function must pass the key objects to the output.collect() method unchanged. The reduce phase is, however, free to output any number of records, including zero records, with the same key and different values.

HDFS Concepts

Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes.

HDFShas the concept of a block, but it is a much larger unit—64 MB by default.File0sin HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

Simplicity is something to strive for all in all systems, but is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management and eliminating metadata concerns

Namenodes and Datanodes

An HDFS cluster has two types of node operating in a master-worker pattern: a *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree.

The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. Datanodes are the workhorses of the filesystem. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.

It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.

HDFS Federation

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling.

HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under */user*, say, and a second

Namenode might handle files under */share*.Under federation, each namenode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

Block pool storage is *not* partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

HDFS High-Availability

The combination of replicating namenode metadata on multiple filesystems, and using

the secondary namenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem. The namenode is still a*single point of failure* (SPOF), since if it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online. In the event of the failure of the active namenode, the

standby takes over its duties to continue servicing client requests without a significant interruption.

A few architectural changes are needed to allow this to happen:

• The namenodes must use highly-available shared storage to share the edit log.

When a standby namenode comes up it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.

• Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.

• Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

If the active namenode fails, then the standby can take over very quickly since it has the latest state available in memory: both the latest edit log entries, and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), since the system needs to be conservative in deciding that the active namenode has failed.

Failover and fencing

The transition from the active namenode to the standby is managed by a new entity in the system called the *failover controller*. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active. Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures and trigger a failover should a namenode fail.

Failover may also be initiated manually by an administrator, in the case of routine maintenance, for example.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as *fencing*. The system employs a range of fencing mechanisms, including killing the namenode's process, revoking its access to the shared storage directory, and disabling its network port via a remote management command. As a last resort, the previously active namenode can be fenced with a technique rather graphically known as *STONITH*, or "shoot the other node in the head", which uses a specialized power distribution unit to forcibly power down the host machine. Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname which is mapped to a pair of namenode addresses, and the client library tries each namenode address until the operation succeeds.

Anatomy of a File Read

The client opens the file it wishes to read by calling open () on theFileSystem object, which for HDFS is an instance of DistributedFileSystem.DistributedFileSystem calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file. The namenode returns the addresses of the datanodes that have a copy of that block.

If the client is itself a datanode ,then it will read from the local datanode, if it hosts a copy of the block .The DistributedFileSystem returns an FSDataInputStream to the client for it to read data from.FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

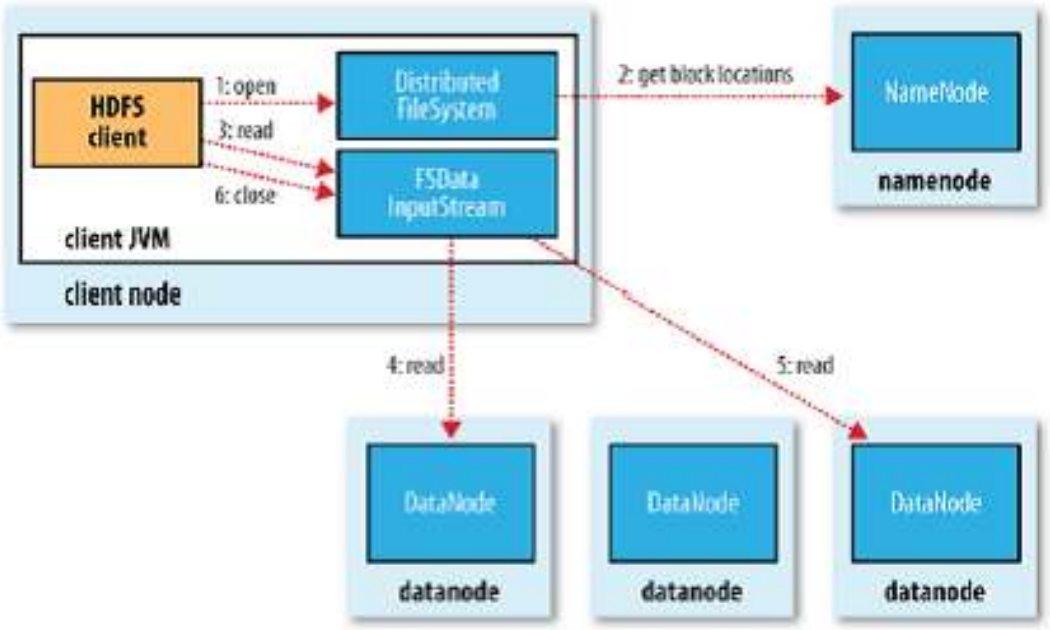


Figure: *A client reading data from HDFS*

The client then calls read () on the stream. 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. 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. This happens transparently to the client, which from its point of view is just reading a continuous stream.

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 theFSDataInputStream .

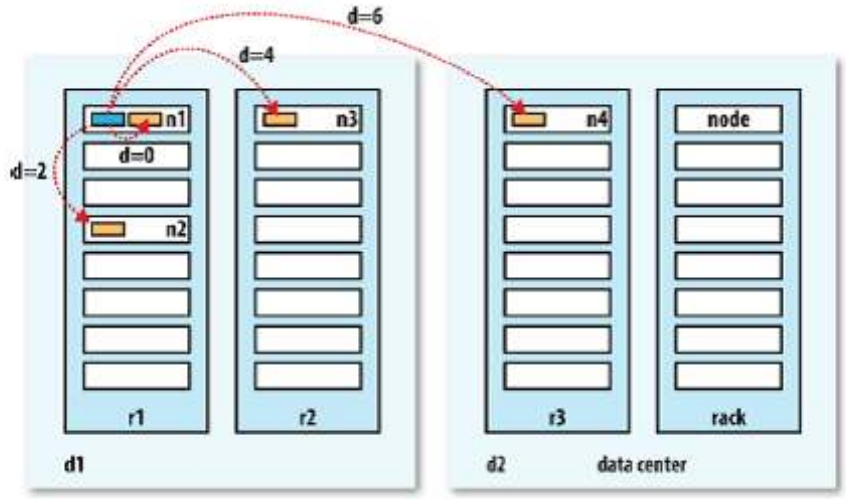


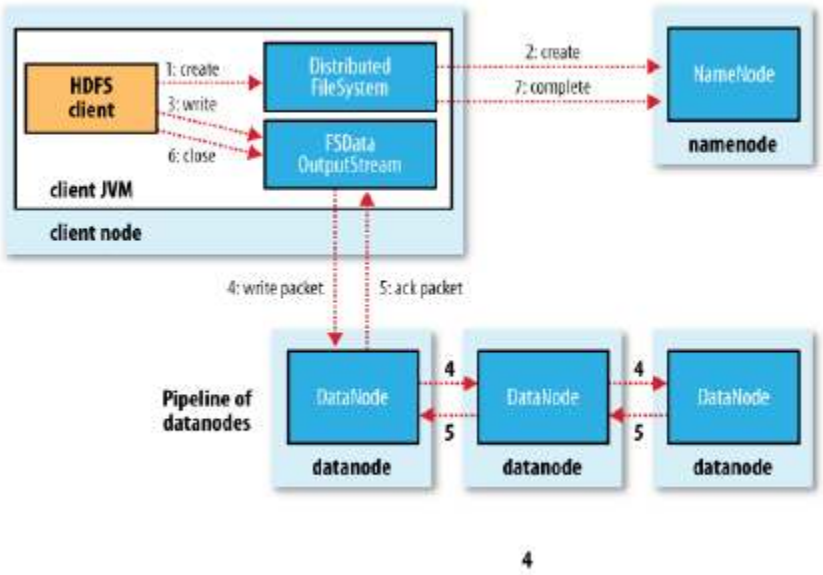
Figure: *Network distance in Hadoop*

During reading, if the DFSInputStream encounters an error while communicating with a datanode, then 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 DFSInputStream also verifies checksums for the data transferred to it from the datanode.
If a corrupted block is found, it is reported to the namenode before theDFSInput Stream attempts to read a replica of the block from another datanode.One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients, since the data traffic is spread across all the datanodes in the cluster.

Anatomy of a File write

The client creates the file by calling create () onDistributedFileSystem. 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.



4

Figure: *A client writing data to HDFS*

The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps aDFSOutput Stream, which handles communication with the datanodes and namenode.As the client writes data (step 3), DFSOutputStream splits it into packets, which it writes to an internal queue, called the *data queue*. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline—we'll assume the replication level is three, so there are three nodes in the pipeline. TheDataStreamer streams the packets to the first datanode in the pipeline, which stores the 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).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).If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data.

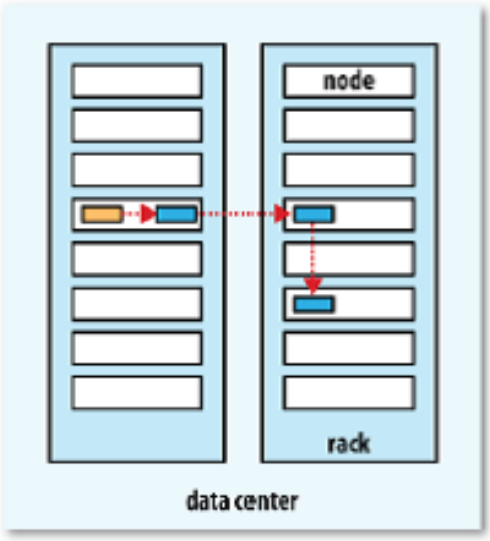


Figure: *A typical replica pipeline*

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 0on 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 the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal. It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long asdfs.replication.min replicas (default one) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached.

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 (step 7). The namenode already knows which blocks the file is made up.

Command line interface in HDFS:

There are many other interfaces to HDFS, but the command line is one of the simplest, and to many developers the most familiar. We are going to run HDFS on one machine, so first follow the instructions for setting up Hadoop in pseudo-distributed mode.Later you'll see how to run on a cluster of machines to give us scalability and fault tolerance.

Creating single node Hadoop cluster

Installing Java:

>sudo apt-get install default-jdk

create hadoop group and hadoop user (hsuser)

>sudo addgroup hadoop

>sudo adduser - -ingroup hadoop hduser

Add user ad sudoers/admin

>sudo adduser hduser sudo

Install openssh server:

>sudo apt-get install openssh-server

Now logout as admin and login with hduser and generate a key for hduser and add the key to the authorized user.

>ssh-keygen -t rsa

this cmd will generate public/private rsa key

>cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys

login to local host:

>ssh localhost

>exit

Install Hadoop:

Install hadoop directly or download it from apache.org and install

>wget http://mirrors.sonic.net/apache/hadoop/common/hadoop-2.7.1/hadoop-2.7.1.tar.gz

then extract using

>tar xvzf hadoop-2.7.1.tar.gz

move hadoop.2.7.1 to a directory of our choice -> /usr/local/hadoop

>sudo mv hadoop-2.7.1 /usr/local/hadoop

verify the directory location

Let give the directory to hduser as owner

>sudo chown -R hduser /usr/local

now edit the bashrc file and append to the end of the file; path to hadoop;

>sudo nano ~/.bashrc

In the last line enter the following comments

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64

export HADOOP_HOME=/usr/local/hadoop

export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export PATH=$PATH:$HADOOP_HOME/bin

export PATH=$PATH:$HADOOP_HOME/sbin
```

save and quit (cntrlO + cntrlX)

>source ~/.bashrc

Now let give java path to run hadoop

>sudo nano /usr.local/hadoop/etc/hadoop/hadoop-env.sh

edit the following line

Change the line export JAVA_HOME=~~${JAVA_HOME}~~ as

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Now let configure the following xml files core-site.xml,hdfs-site.xml,yarn-site.xmland mapred-site.xml.

>sudo nano /usr/local/hadoop/etc/hadoop/core-site.xml

go to last and add the following tags

```
<configuration>
        <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:9000</value>
        </property>
</configuration>
```

>sudo nano /usr/local/hadoop/etc/hadoop/hdfs-site.xml

go to last and add the following tags

```
<configuration>

        <property>

                <name>dfs.replication</name>

                <value>1</value>

        </property>

        <property>

                <name>dfs.namenode.name.dir</name>

                <value>file:/usr/local/hadoop_tmp/hdfs/namenode</value>

        </property>

        <property>

                <name>dfs.datanode.name.dir</name>

                <value>file:/usr/local/hadoop_tmp/hdfs/datanode</value>

        </property>

</configuration>
```

>sudo nano /usr/local/hadoop/etc/hadoop/yarn-site.xml

```
<configuration>
        <property>
                <name>yarn.nodemanager.aux-services</name>
                <value>mapreduce_shuffle</value>
```

```
</property>
<property>
<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
</configuration>
```

Let copy the mapred.xml template and then edit the file

>cp /usr/local/hadoop/etc/hadoop/mapred-site.xml.template /usr/local/hadoop/etc/hadoop/mapred-site.xml

>sudo nano /usr/local/hadoop/etc/hadoop/mapred-site.xml

```
<configuration>

    <property>

        <name>mapreduce.framework.name</name>

        <value>yarn</value>

    </property>

</configuration>
```

Now let create folders where hadoop will process the hdfs jobs

>sudo mkdir -p /usr/local/hadoop_tmp

>sudo mkdir -p /usr/local/hadoop_tmp/hdfs/namenode

>sudo mkdir -p /usr/local/hadoop_tmp/hdfs/datanode

Assign hduser the ownership of the folder

>sudo chown -R hduser /usr/local/hadoop_tmp

>hdfs namenode -format

>start-dfs.sh

>start-yarn.sh

>jps

Sample output:

```
12448 NodeManager
```

12688 Jps
12088 SecondaryNameNode
11677 NameNode
12189 ResourceManager
11823 DataNode