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.

Apache Hadoop

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

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 Distributed File System (HDFS): Introduction

- **The Hadoop Distributed File System (HDFS)** is the **primary data storage system** used by Hadoop applications.
- The Hadoop Distributed File System (HDFS) is **designed to store very large data** sets reliably, and to **stream those data sets at high bandwidth** to user applications
- HDFS is simply a distributed file system. This means that **a single large dataset can be stored in several different storage nodes** within a compute cluster.
- HDFS is how Hadoop is able to offer **scalability and reliability** for the storage of large datasets in a distributed fashion.
- **HDFS (storage)** and **YARN (processing)** are the two core components of Apache Hadoop.
- **Hadoop is an open-source framework**. Hadoop is best known for its **fault tolerance and high availability feature.** Hadoop clusters are scalable. The Hadoop framework is easy to use. It ensures fast data processing due to distributed processing.

What is HDFS?

Hadoop Distributed File System is a **fault-tolerant** **data storage file system** that runs on **commodity hardware.**

- a single large dataset can be stored in several different storage nodes within a compute cluster.
- It offer **scalability and reliability** for the storage of large datasets in a **distributed** fashion.

What are the Benefits of HDFS?

- **It is fast.** It can deliver more than 2 GB of data per second
- **It is free.** HDFS is an open-source software
- **It is reliable.** The file system stores multiple copies of data in separate systems to ensure it is always accessible.

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.

Hadoop Architecture Overview

Apache Hadoop offers **a scalable, flexible, and reliable distributed computing big data framework** for a cluster of systems with storage capacity and local computing power leveraging commodity hardware.

Hadoop follows a **Master-Slave architecture** to transform and analyze large datasets using the Hadoop MapReduce paradigm.

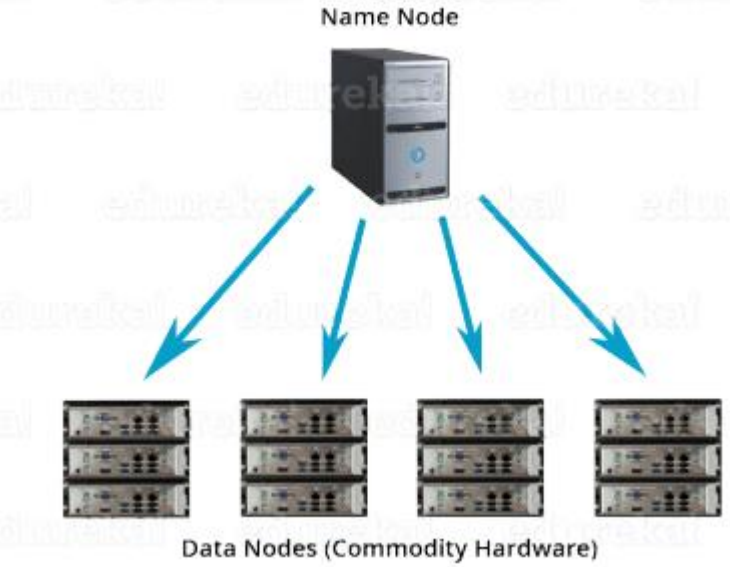
The components that play a vital role in the Hadoop application architecture are -

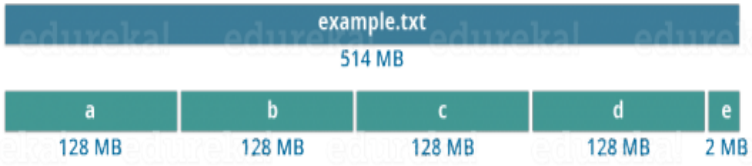
- **Hadoop Distributed File System (HDFS)–** The Hadoop distributed file system stores data on commodity machines, providing high aggregate bandwidth across the cluster
- **Hadoop MapReduce** – This application of the MapReduce programming model is useful for large-scale data processing.
- **Hadoop YARN –** This platform is in charge of managing computing resources in clusters and utilizing them for planning users' applications.

Apache Hadoop : 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.

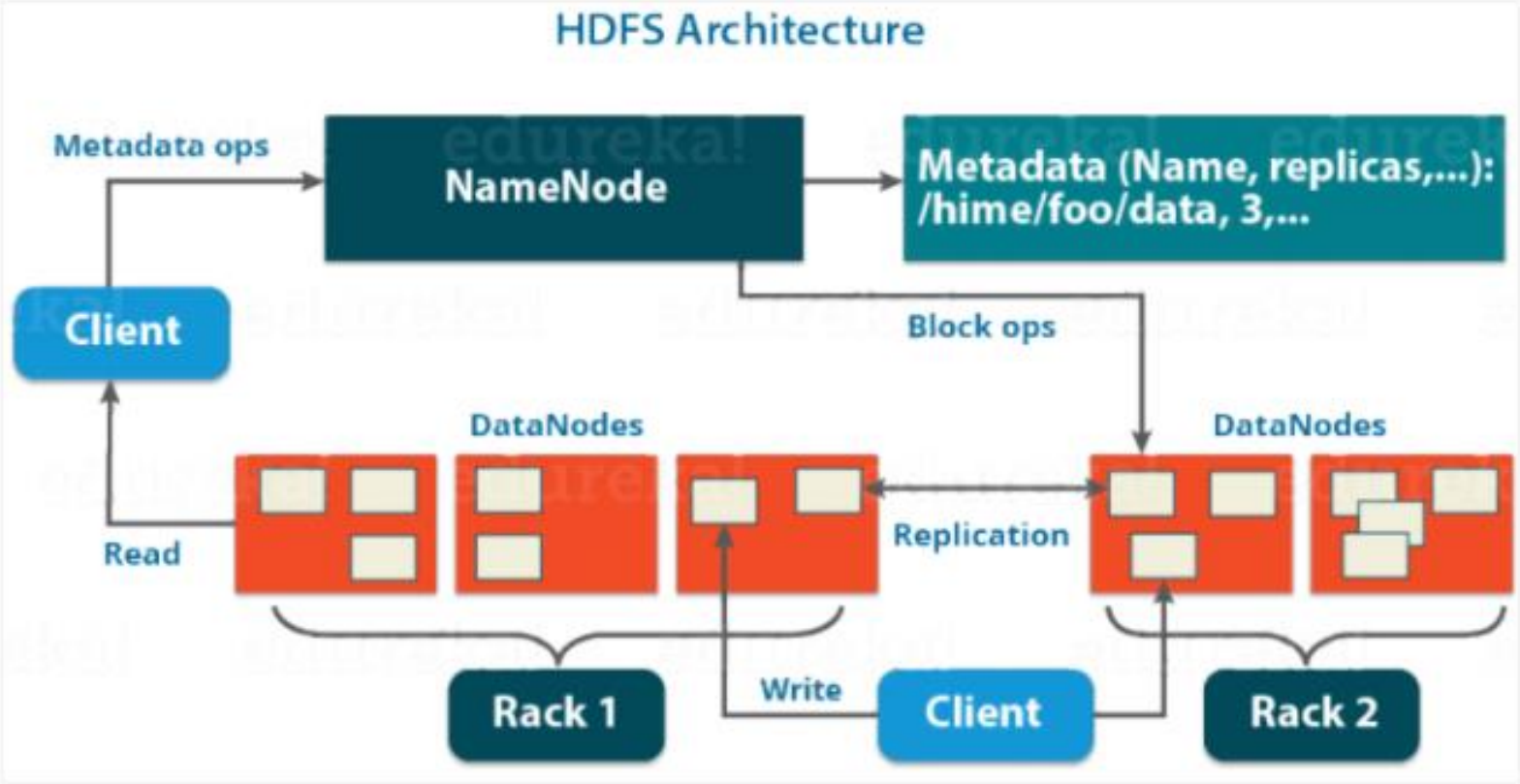
- Apache Hadoop **HDFS Architecture follows a Master/Slave Architecture,**
- where a cluster comprises of a single **NameNode (Master node)** and all the other nodes are **DataNodes (Slave nodes).**
- HDFS can be deployed on a broad spectrum o machines that **support Java**.
- Though one can run several DataNodes on a single machine, but in the practical world, these **DataNodes are spread across various machines.**





Data in HDFS is scattered across the DataNodes as blocks

HDFS Architecture



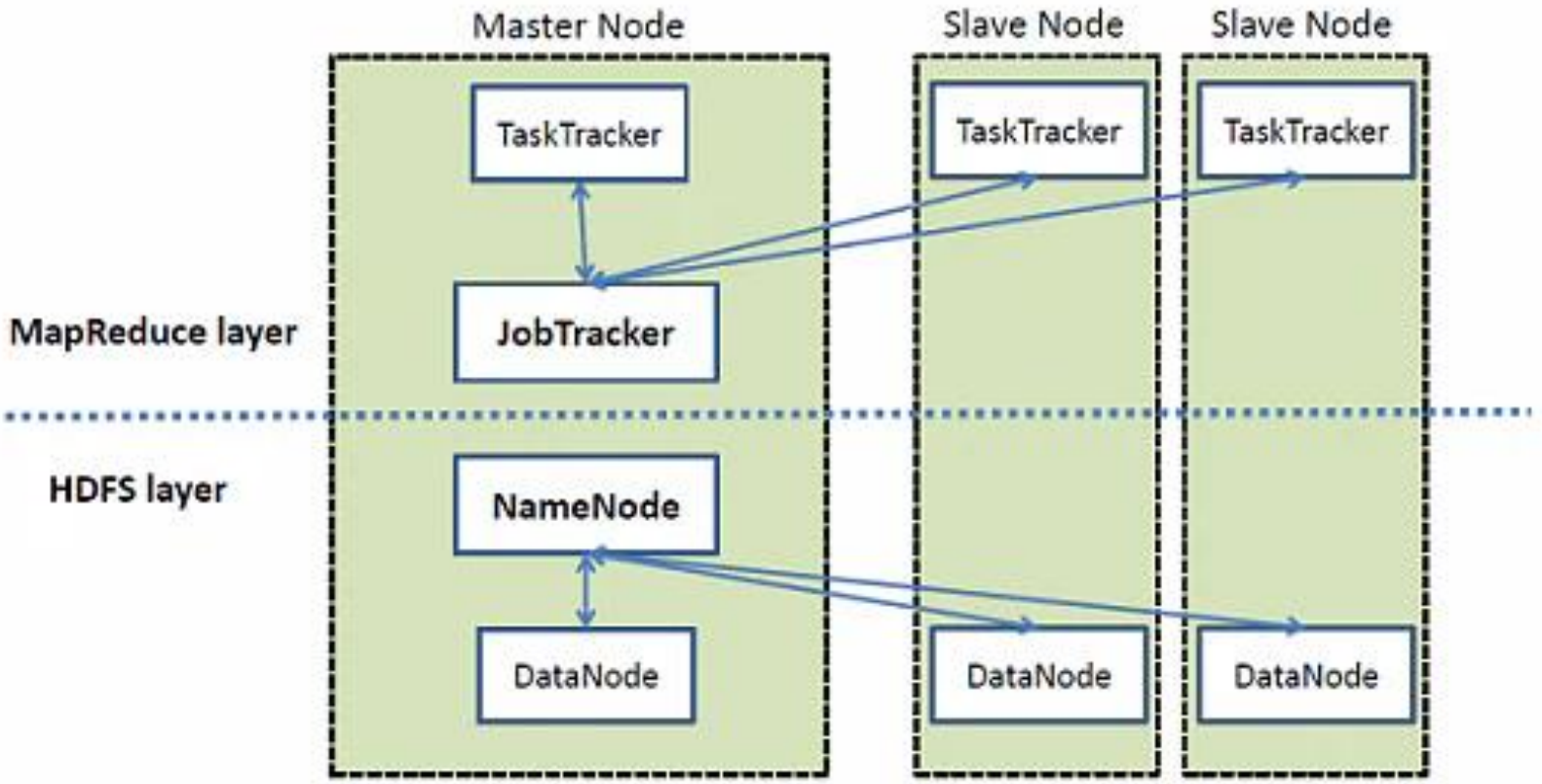
- Hadoop follows a master-slave architecture design for **data storage and distributed processing** using **HDFS** and **MapReduce.**

- The **master node** for data storage is Hadoop HDFS, the **NameNode**, and the master node for parallel data processing using Hadoop MapReduce is the **Job Tracker.**

- The **slave nodes** in the Hadoop physical architecture are the other machines in the Hadoop cluster that store data and **perform complex computations**.

- Every slave node has a Task Tracker daemon and a **DataNode** that synchronizes the processes with the Job Tracker and NameNode.

High Level Architecture of Hadoop



Distributed Storage Layer

- **A Hadoop cluster consists of several nodes**, each having its own disk space, memory, bandwidth, and processing.

- HDFS considers each disk drive and slave node in the cluster as inconsistent, and HDFS holds three copies of each data set throughout the cluster as a backup.

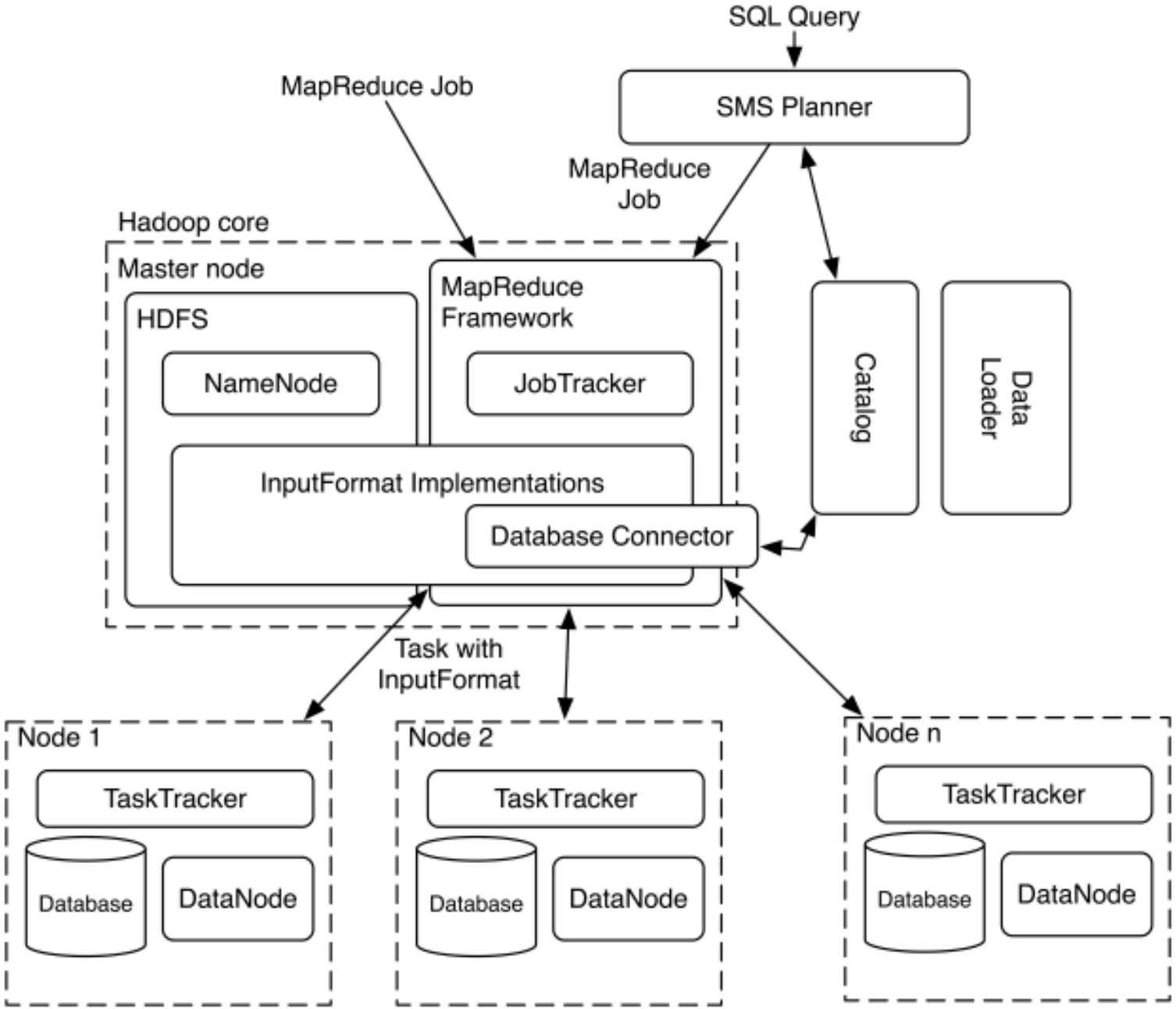
- The HDFS **master node (NameNode)** maintains each data block's information and copies.

Cluster Resource Management

- **MapReduce** was responsible for the **tasks of resource management** as well as data processing in a Hadoop cluster.

Application Programming Interface

- Adding YARN to the Hadoop physical architecture has caused several data processing frameworks and APIs to emerge, such as HBase, Hive, Apache Pig, SQL, etc


SQL Query
SMS Planner
MapReduce Job
MapReduce Job
Hadoop core
Master node
HDFS
NameNode
MapReduce Framework
JobTracker
InputFormat Implementations
Database Connector
Catalog
Data Loader
Task with InputFormat
Node 1
TaskTracker
Database
DataNode
Node 2
TaskTracker
Database
DataNode
Node n
TaskTracker
Database
DataNode

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**

Low-latency data access

- Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS.
- Remember, **HDFS is optimized for delivering a high throughput of data**, and this may be at the expense of latency. Hbase is currently a better choice for low-latency access

Lots of small files

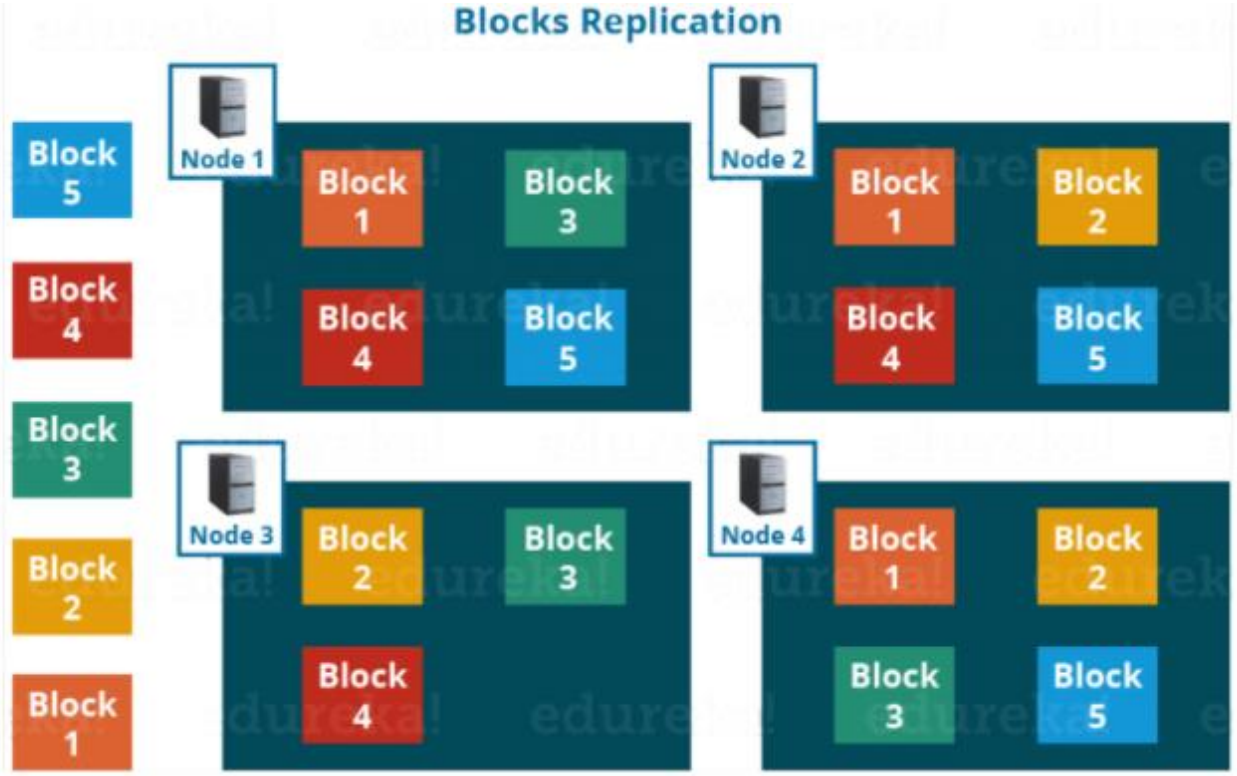
- Since the **namenode holds filesystem metadata in memory**, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode.
- As a rule of thumb, **each file, directory, and block takes about 150 bytes**. So, for example, if you had one million files, each taking one block, you would **need at least 300 MB of memory**

Multiple writers, arbitrary file modifications

- **Files in HDFS may be written to by a single writer. Writes are always made at the end of the file.**
- There is no support for multiple writers, or for modifications at arbitrary offsets in the file. (These might be supported in the future, but they are likely to be relatively inefficient.)

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):



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

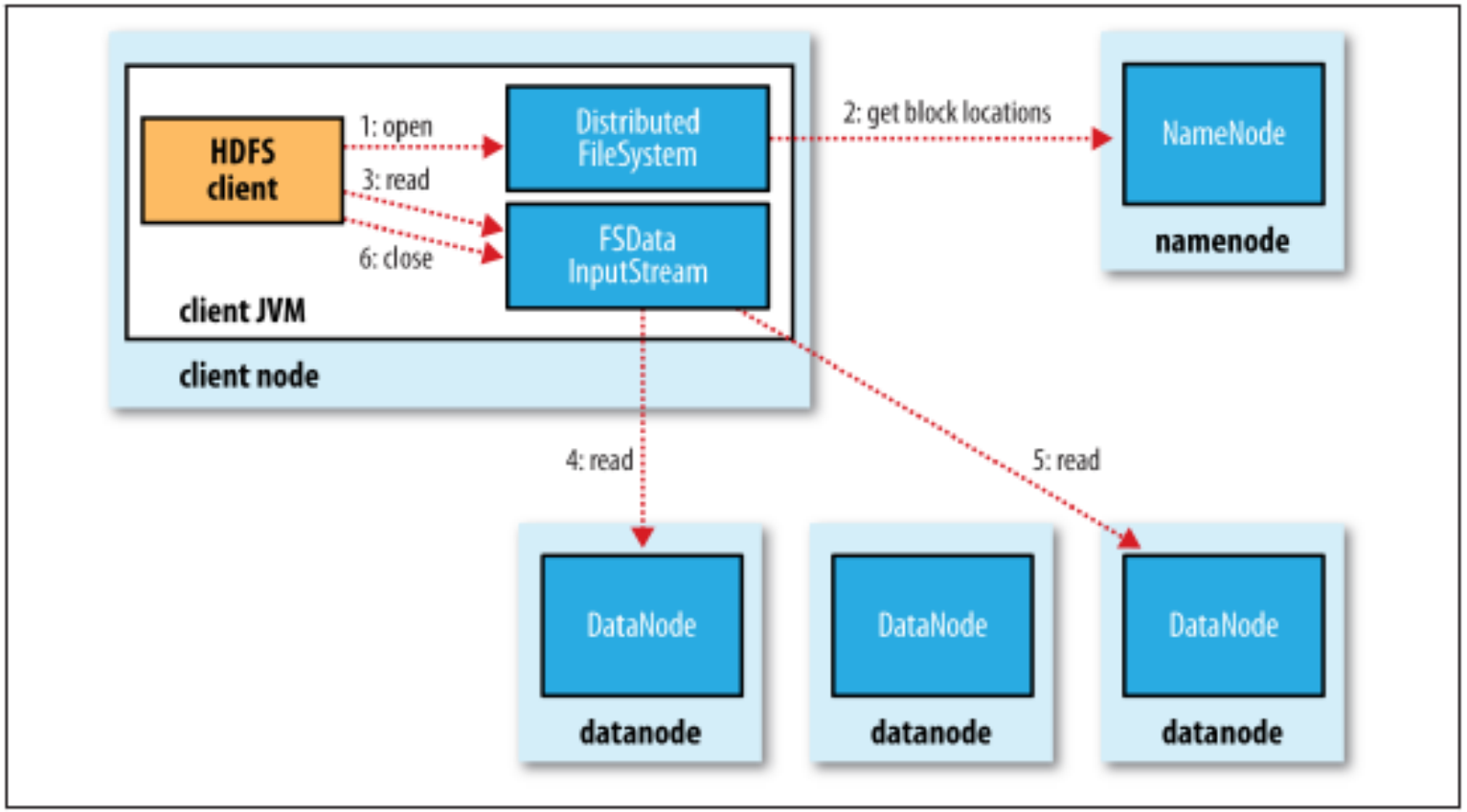


Figure 3-2. A client reading data from HDFS

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

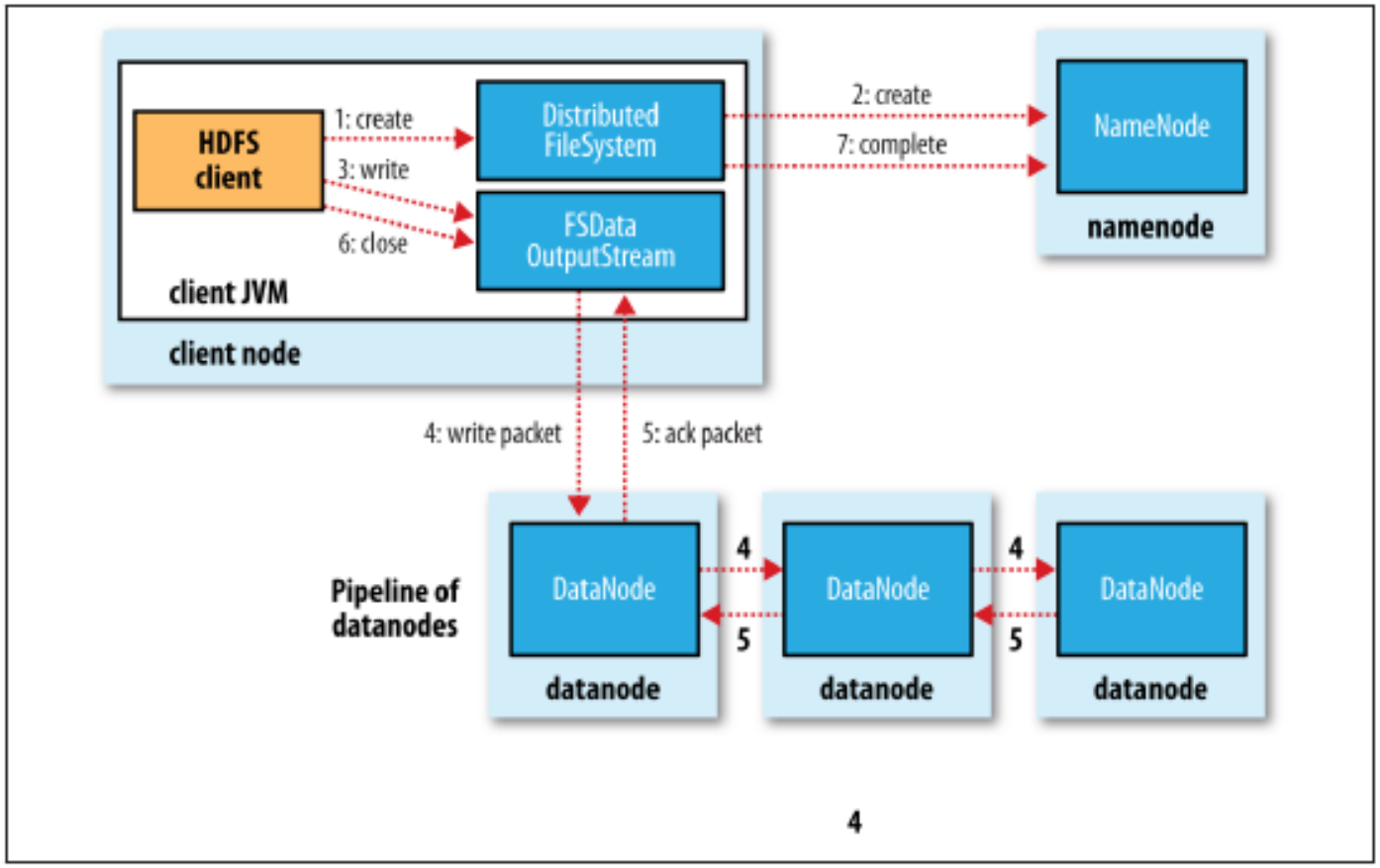


Figure 3-4. A client writing data to HDFS

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).**

What Is MapReduce? : Introduction

MapReduce is a software framework for easily writing applications which **process vast amounts of data (multi-terabyte data-sets) in-parallel** on large clusters (thousands of nodes) of **commodity hardware in a reliable, fault-tolerant manner**.

A **MapReduce job** usually **splits the input data-set into independent chunks** which are processed by the **map tasks** in a completely parallel manner.

The framework **sorts the outputs of the maps**, which are then input to the **reduce tasks**. Typically both the input and the output of the job are stored in a file-system

The MapReduce framework consists of a single master **JobTracker** and one slave **TaskTracker** per cluster-node.

- **The master is responsible for scheduling the jobs' component tasks** on the slaves, monitoring them and re-executing the failed tasks.

- The **slaves execute the tasks** as directed by the master.

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

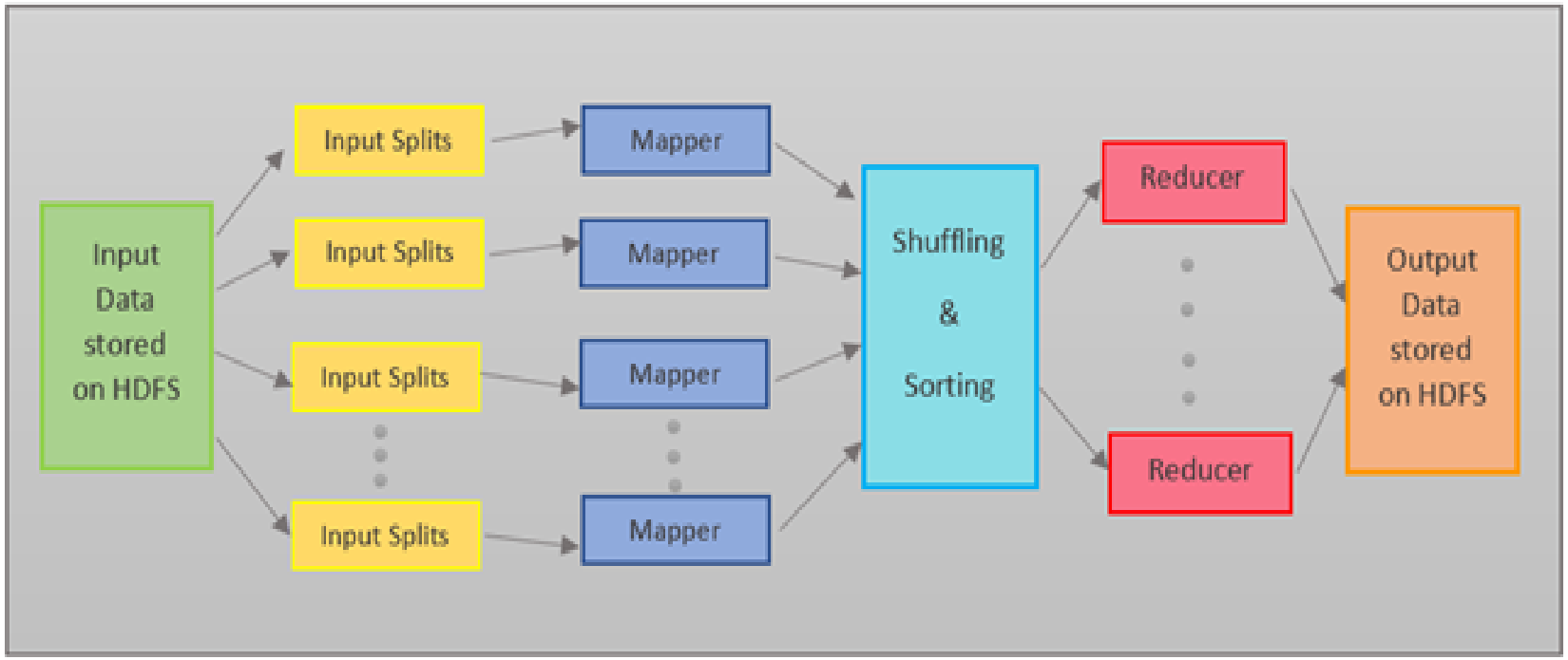
Reducer Class:

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 Architecture



Input Splits:

MapReduce splits the input into **smaller chunks called input splits**, representing a block of work with a single mapper task.

Mapping:

The **input data is processed and divided into smaller segments** in the mapper phase, where the number of mappers is equal to the number of input splits.

Shuffling:

In the shuffling phase, the output of the mapper phase is passed to the reducer phase by **removing duplicate values and grouping the values.** The output remains in the form of keys and values in the mapper phase.

Sorting:

Sorting is performed simultaneously with shuffling. The **Sorting phase involves merging and sorting the output generated by the mapper**. The intermediate key-value pairs are sorted by key before starting the reducer phase, and the values can take any order.

Reducing:

In the reducer phase, the intermediate values from the shuffling phase are reduced to produce **a single output value that summarizes the entire dataset**. HDFS is then used to store the final output.

MapReduce Architecture : Components

Map Phase:

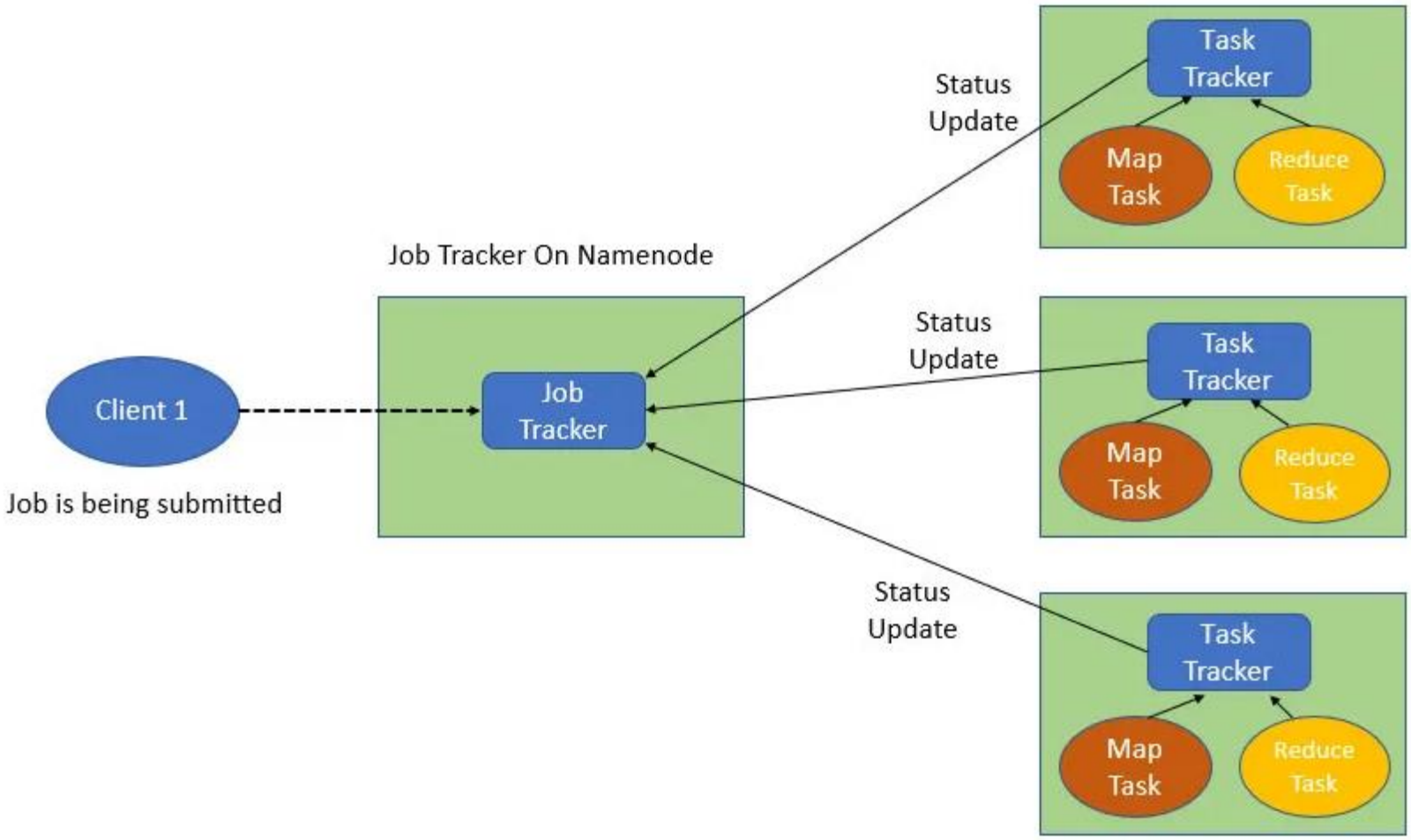
Map phase splits the input data into two parts. They are Keys and Values. Writable and comparable is the key in the processing stage where only in the processing stage, Value is writable.

Processing in Intermediate:

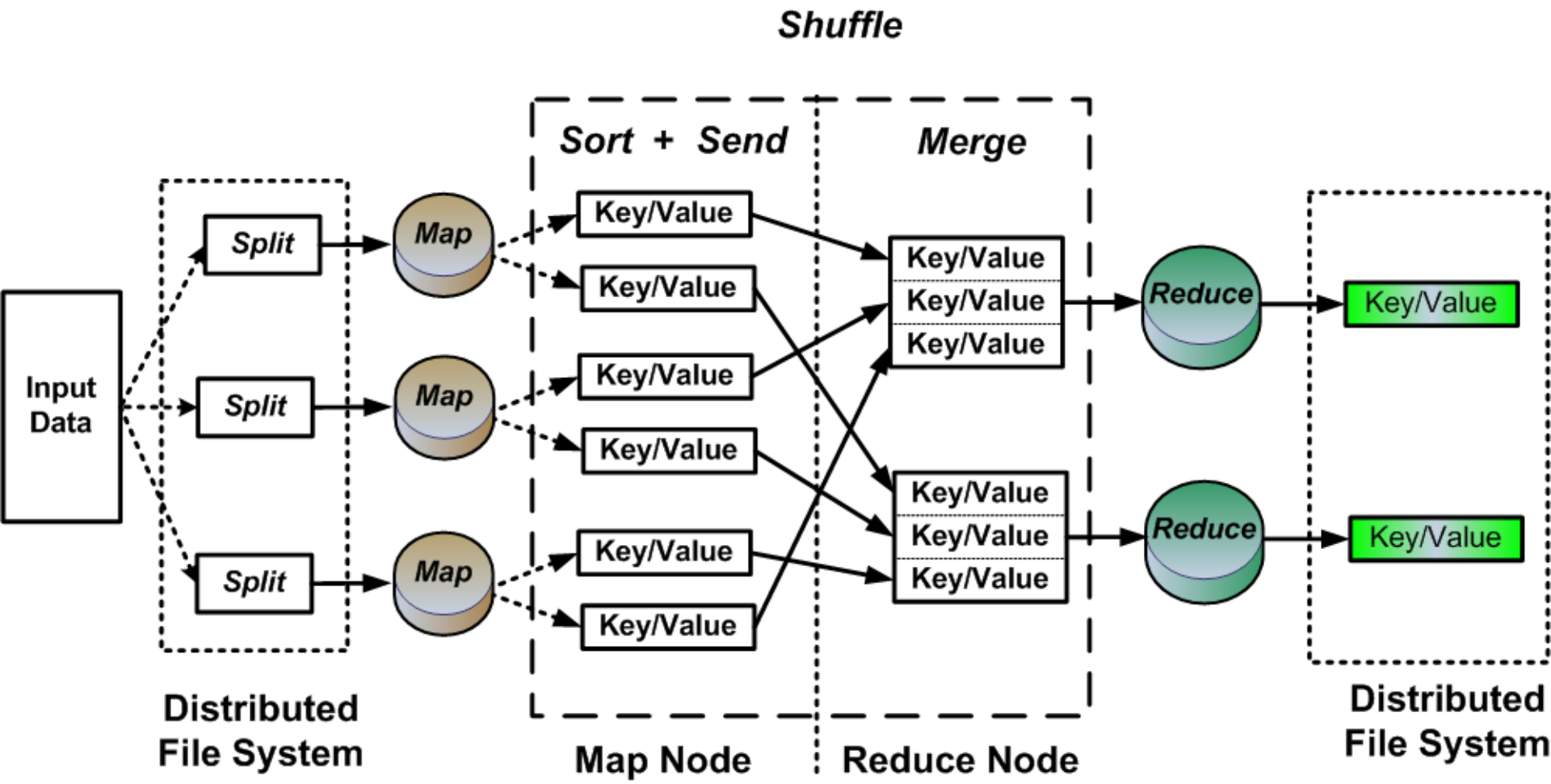
In the intermediate phase, the map input gets into the sort and shuffle phase. Hadoop nodes do not have replications where all the intermediate data is stored in a local file system.

Reducer Phase:

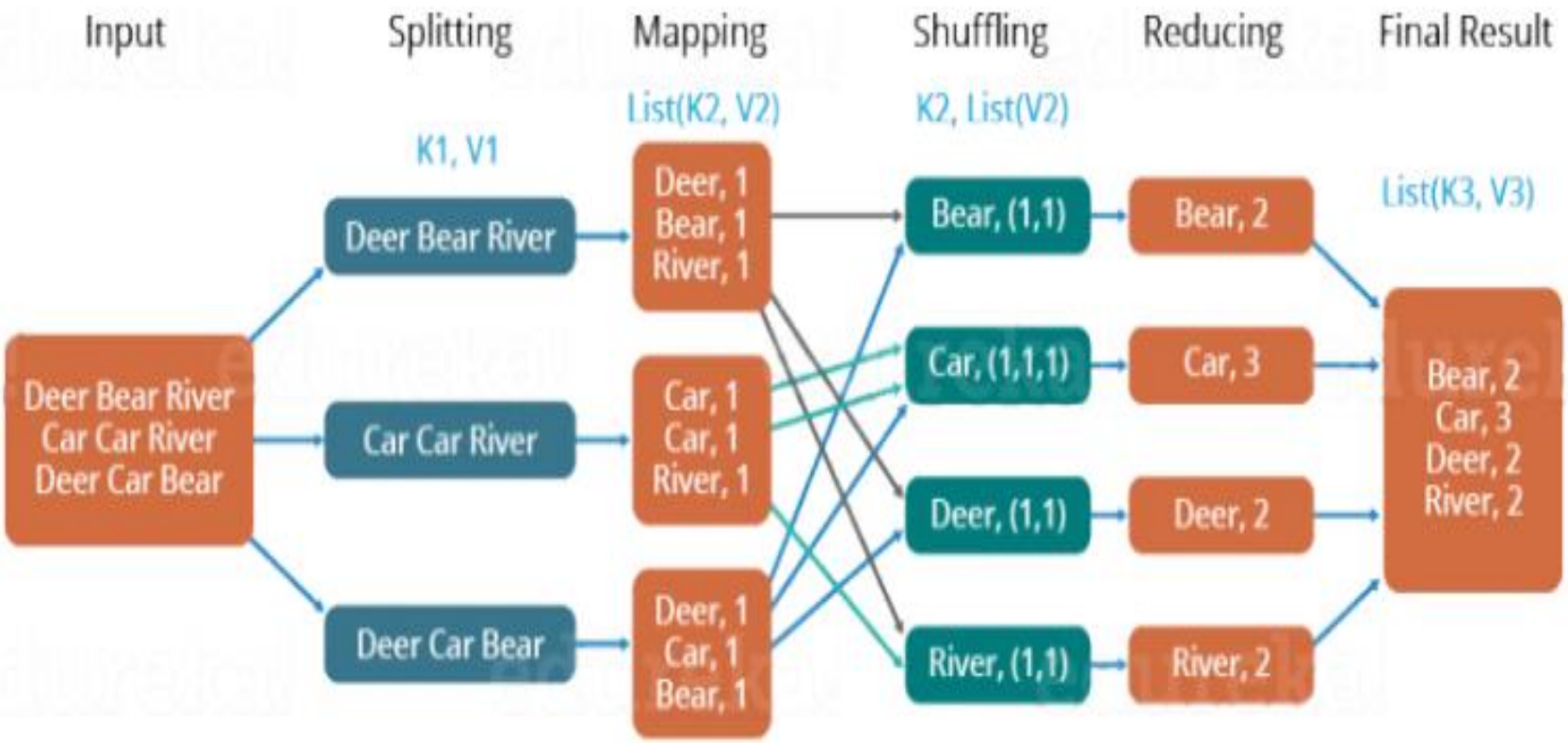
The reducer takes in the data input that is sorted and shuffled. **All the input data will be combined, and similar key-value pairs are to be written to the hdfs system.**



The job is divided into two components: **Map tasks (Splits and mapping)** and **Reduce tasks (Reducing and shuffling).**


Shuffle
Sort + Send
Merge
Input Data
Split
Split
Split
Map
Map
Map
Key/Value
Key/Value
Key/Value
Key/Value
Key/Value
Key/Value
Key/Value
Key/Value
Key/Value
Key/Value
Reduce
Reduce
Key/Value
Key/Value
Distributed File System
Map Node
Reduce Node
Distributed File System

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**

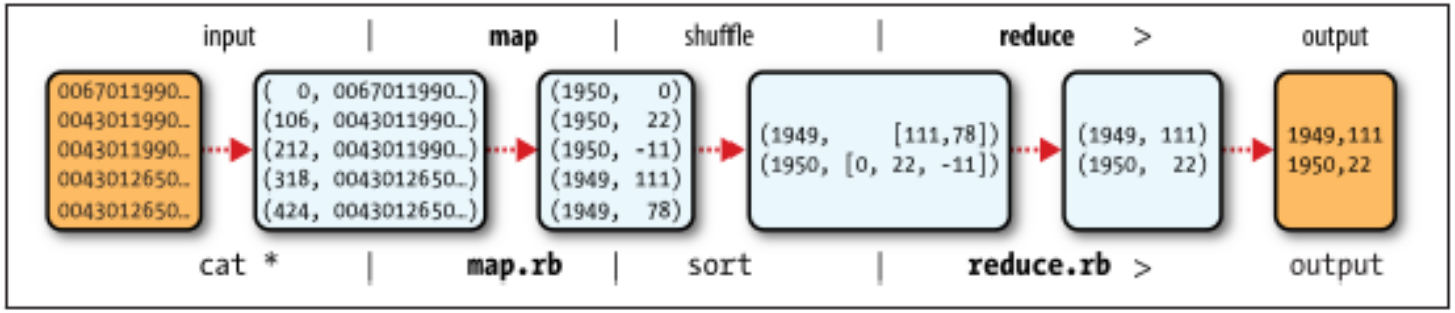


Figure 2-1. MapReduce logical data flow

MapReduce data flow with a single reduce task

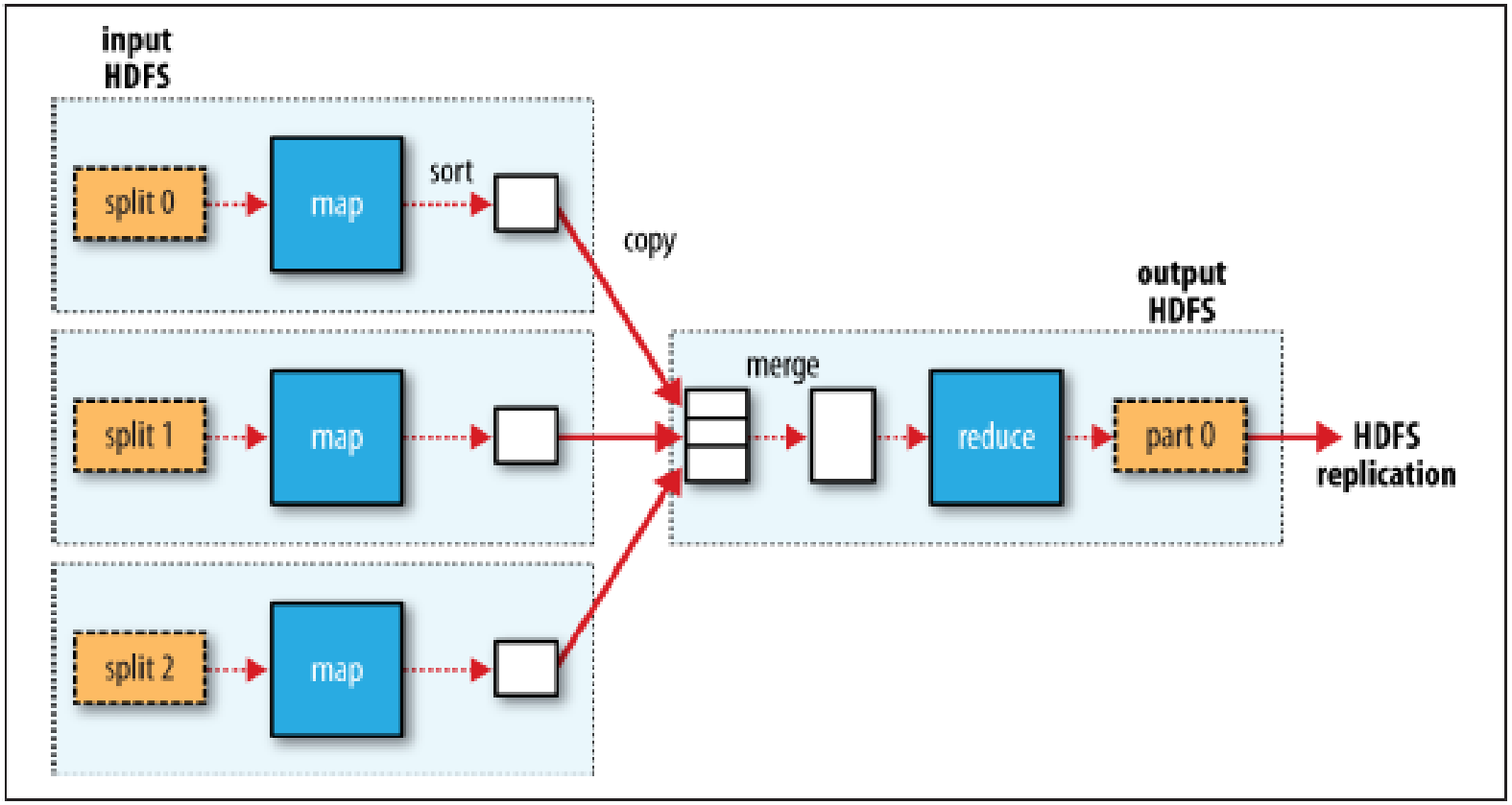


Figure 2-3. MapReduce data flow with a single reduce task

The number of reduce tasks is not governed by the size of the input, but instead is specified independently.

- **When there are multiple reducers**, the map tasks partition their output, each creating **one partition for each reduce task.**

- **There can be many keys (and their associated values) in each partition**, but the records for any given key are all in a single partition.

- The partitioning can be **controlled by a user-defined partitioning function**, but normally the default partitioner—which buckets keys using a hash function—works very well.

- The data flow for the general case of **multiple reduce tasks is illustrated in Figure 2-4**.

  - This diagram makes it clear why **the data flow between map and reduce tasks is colloquially known as "the shuffle,"** as each reduce task is fed by many map tasks.

MapReduce data flow with a Multiple reduce task

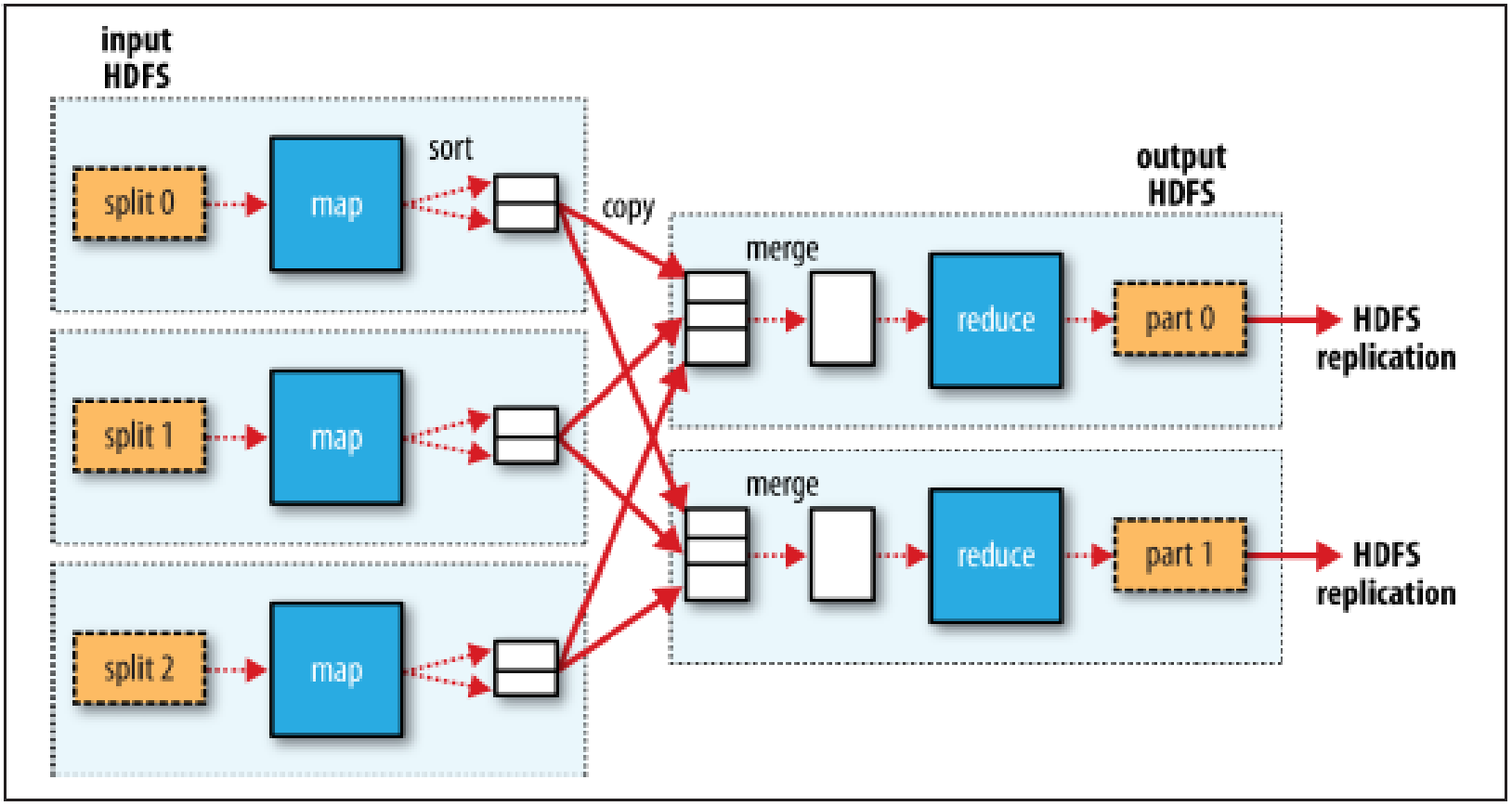


Figure 2-4. MapReduce data flow with multiple reduce tasks

Anatomy of a MapReduce Job Run: How MapReduce Works

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.

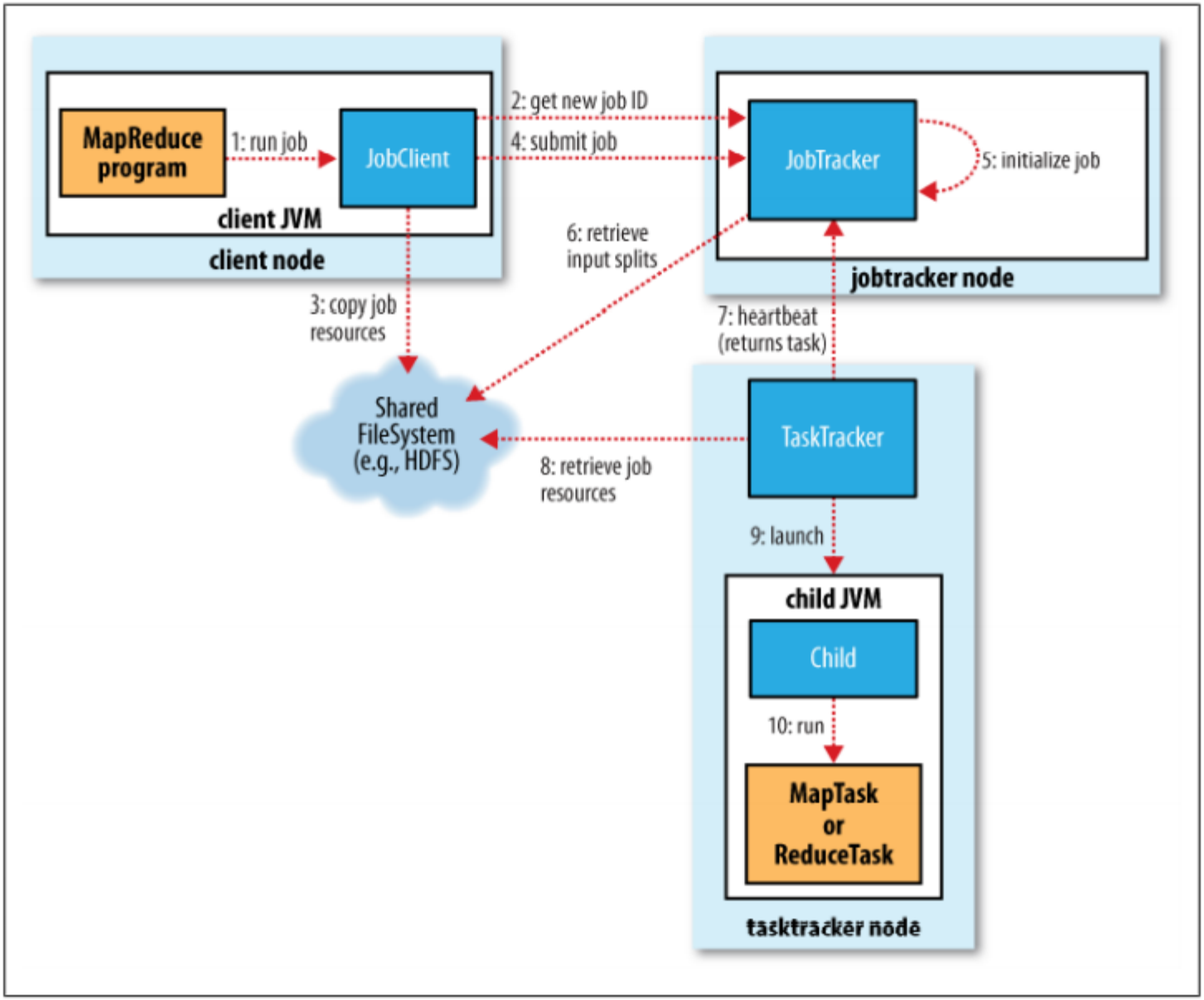


Figure 6-1. How Hadoop runs a MapReduce job using the classic framework

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;

Failures in Classic MapReduce

In the MapReduce 1 runtime there are three failure modes to consider:

- Failure of the running task(MapReduce),
- Failure of the Tastracker, and
- Failure of the Jobtracker.

Task Failure: **Use Codes→ Task**

- Consider first the case of the **child Task Failing :** The most common way that this happens is when user code in the map or reduce task **throws a Runtime Exception.**
  - If this happens, the child JVM reports the error back to its parent tasktracker, before it exits. The error ultimately makes it into the user logs.
  - The **tasktracker marks the task attempt as failed**, freeing up a slot to run another task.
- **For Streaming tasks,** if the Streaming process exits with a nonzero exit code, it is marked as failed.
- Another failure mode is the **Sudden Exit of the Child JVM**—perhaps there is a **JVM bug that causes the JVM to exit** for a particular set of circumstances exposed by the MapReduce user code. In this case, the **tasktracker notices that the process has exited(No Progress)** and marks the attempt as failed.

Tasktracker Failure

- Failure of a tasktracker is another failure mode.
  - **If a tasktracker fails by crashing, or running very slowly,** it will stop sending heartbeats to the jobtracker (or send them very infrequently).
  - The jobtracker will notice **a tasktracker that has stopped sending heartbeats** and remove it from its pool of tasktrackers to schedule tasks on.
  - **A tasktracker can also be blacklisted** by the jobtracker, even if the tasktracker has not failed. **If more than four tasks from the same job fail on a particular tasktracker** ,then the jobtracker records this as a fault.
  - A tasktracker is **blacklisted** if the number of faults is over **some minimum threshold and is significantly higher than the average number of faults** for tasktrackers in the cluster cluster.

Jobtracker Failure

- **Failure of the jobtracker is the most serious failure mode.**
- Hadoop has no mechanism for dealing with failure of the jobtracker—it is a single point of failure—so in this case the job fails.
- However, this failure mode has a low chance of occurring, since the chance of a particular machine failing is low.
- The good news is that **the situation is improved in YARN, since one of its design goals is to eliminate single points of failure in MapReduce**.

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: **Jobtracker**
  - 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. : **Scheduling and Application Manager**
- The **YARN Node Managers,** which **launch and monitor the compute containers on Salve 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**.

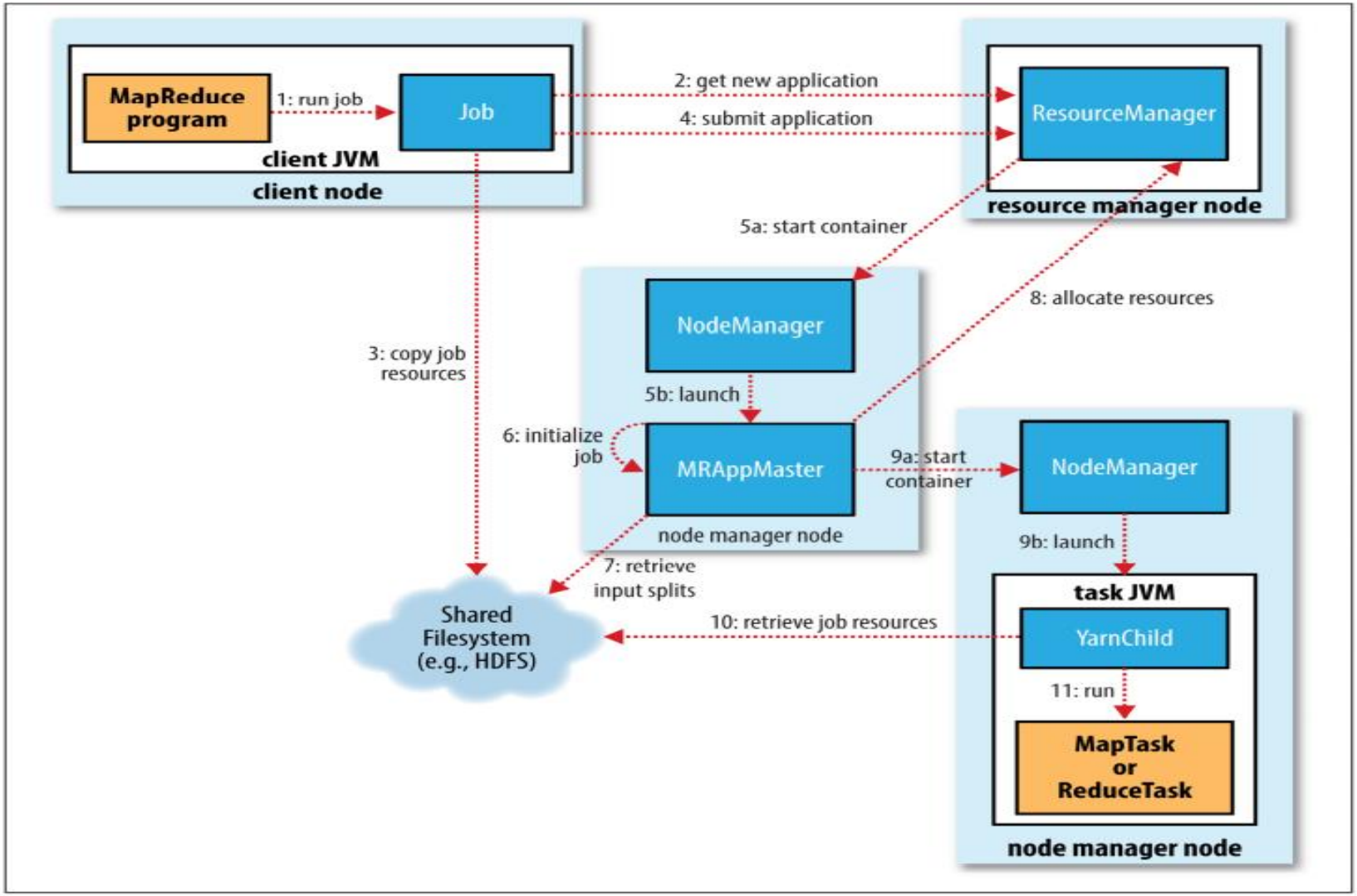


Figure 7-1. How Hadoop runs a MapReduce job

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).

Hadoop 2.0 (YARN)

ResourceManager –

- The ResourceManager component is the **negotiator in a cluster** for all the resources present in that cluster.
- Furthermore, this component is classified into **an application manager which is responsible for managing user jobs.**
- From Hadoop 2.0 any MapReduce job will be considered as an application.

ApplicationMaster –

- This component is the **place in which a job or application exists**.
- It also **manages all the MapReduce jobs** and is concluded after the job processing is complete.

NodeManager –

- The node manager component **acts as the server for job history**.
- It is responsible for securing information of the completed jobs.
- It also **keeps track of the users' jobs along with their workflow** for a particular node.

Advantage of Hadoop 2.0 (YARN)

Better utilization of resources – The YARN framework **does not have any fixed slots for tasks.** It provides a central resource manager which allows you to share multiple applications through a common resource.

Running non-MapReduce applications – In YARN, the **scheduling and resource management capabilities are separated from the data processing component**.

- This allows Hadoop to run **varied types of applications** which do not conform to the programming of the Hadoop framework.
- Hadoop clusters are now capable of **running independent interactive queries** and performing better real-time analysis.

Backward compatibility – YARN comes as a backward-compatible framework, which means any **existing job of MapReduce can be executed in Hadoop 2.0.**

JobTracker no longer exists – The two major roles of **the JobTracker were resource management and job scheduling.** With the introduction of the YARN framework these are now segregated into two separate components, namely:

- **NodeManager**
- **ResourceManager**

Advantage of Hadoop 2.0 (YARN)

Scalability: The scheduler in Resource manager of YARN architecture allows Hadoop to extend and **manage thousands of nodes and clusters.**

Compatibility: YARN **supports the existing map-reduce applications without disruptions** thus making it compatible with Hadoop 1.0 as well.

Cluster Utilization: Since YARN supports **Dynamic utilization of cluster** in Hadoop, which enables optimized Cluster Utilization.

Multi-tenancy: It **allows multiple engine access** thus giving organizations a benefit of multi-tenancy.

Failures in YARN

For MapReduce programs running on YARN, we need to consider the failure of any of the following entities:

- the Task Failure ,
- the Application Master Failure,
- the Node Manager Failure
- the Resource Manager Failure

Task Failure

- Failure of the running task is similar to the classic case.
- **Runtime Exceptions and Sudden Exits of the JVM** are propagated back to the application master and the task attempt is marked as failed.
- Likewise, hanging tasks are noticed by the application master by the absence of a ping over the umbilical channel (**the timeout is set by mapreduce.task.time out**), and again the task attempt is marked as failed.
- The configuration properties for determining when a task is considered to be failed are the same as the classic case: **a task is marked as failed after four attempts**

Application Master Failure

- Just like MapReduce tasks are given several attempts to succeed (in the face of **hardware or network failures**) applications in YARN are tried multiple times in the event of failure.

- By default, applications are marked as failed if they fail once, but this can be increased by setting the property **yarn.resourcemanager.am.max-retries.**

- An application master sends **periodic heartbeats to the resource manager**, and in the event of application master failure, the resource manager will detect the failure and start a new instance of the master running in a new container (managed by a node manager).

Node Manager Failure

- If a node manager fails, then it will **stop sending heartbeats to the resource manager**, and the node manager will be removed from the resource manager's pool of available nodes.

- The property **yarn.resourcemanager.nm.liveness-monitor.expiry-intervalms**, which defaults to 600000 (10 minutes), determines the **minimum time the resource manager waits** before considering a node manager that has sent no heartbeat in that time as failed.

- **Node managers may be blacklisted if the number of failures for the application is high. Blacklisting is done by the application master**, and for MapReduce the application master will reschedule tasks on different nodes if more than three tasks fail on a node manager.

Resource Manager Failure

- **Failure of the resource manager is serious**, since **without it neither jobs nor task containers can be launched.**

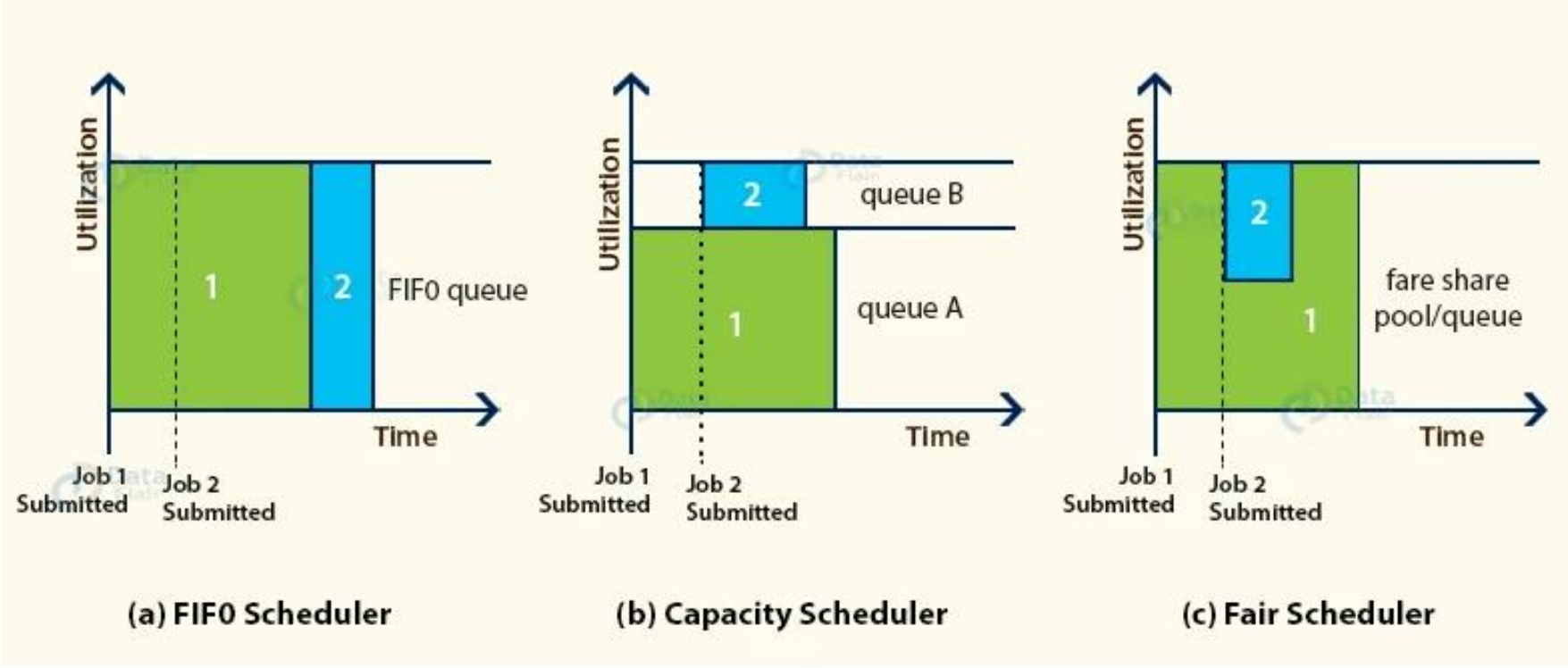
- **The resource manager was designed from the outset to be able to recover from crashes**, by using a checkpointing mechanism to save its state to persistent storage, although at the time of writing the latest release did not have a complete implementation.

- After a crash, **a new resource manager instance is brought up (by an adminstrator)** and **it recovers from the saved state**. The state consists of the node managers in the system as well as the running applications.

Job Scheduling

- Schedulers in YARN Resource Manager is a pure scheduler which is responsible for **allocating resources to the various running applications**.

- The scheduler performs scheduling **based on the resource requirements of the applications.**



Job Scheduling

MapReduce in Hadoop comes with a choice of schedulers. The default in MapReduce 1 is the original **FIFO queue-based scheduler,** and there are also multiuser schedulers called the **Fair Scheduler** and the **Capacity Scheduler.**

The Fair Scheduler :

- The Fair Scheduler **aims to give every user a fair share of the cluster capacity** over time.
    - If a single job is running, it gets all of the cluster.
    - As more jobs are submitted, free task slots are given to the jobs in such a way as to give each user a fair share of the cluster.
    - A short job belonging to one user will complete in a reasonable time even while another user's long job is running, and the long job will still make progress.
    - Jobs are placed in pools, and by default, each user gets their own pool. A user who submits more jobs than a second user will not get any more cluster resources than the second, on average.
- The **Fair Scheduler supports preemption**, so if a pool has not received its fair share for a certain period of time, then the scheduler will kill tasks in pools running over capacity in order to give the slots to the pool running under capacity.

The Capacity Scheduler

- The Capacity Scheduler takes a slightly different approach to multiuser scheduling.

- A cluster is made up of a number of queues (like the Fair Scheduler's pools), which may be hierarchical (so a queue may be the child of another queue), and each queue has an allocated capacity.

- **This is like the Fair Scheduler, except that within each queue, jobs are scheduled using FIFO scheduling (with priorities).**

- In effect, the Capacity Scheduler allows users or organizations (defined using queues) to simulate a separate MapReduce cluster with FIFO scheduling for each user or organization.

- The Fair Scheduler, by contrast, (which actually also supports FIFO job scheduling within pools as an option, making it like the Capacity Scheduler) enforces fair sharing within each pool, so running jobs share the pool's resources.

Shuffle and Sort

- MapReduce makes the guarantee that the **input to every reducer is sorted by key.** The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the **shuffle**

    - The shuffle is an area of the codebase where **refinements and improvements are continually being made**
    - The shuffle is the heart of MapReduce and is where the "magic" happens.

The Map Side

- When the map function starts producing output, it is not simply written to disk. The process is more involved, and **takes advantage of buffering writes in memory** and doing some presorting for efficiency reasons. Figure 6-6 shows what happens.

- Each map task has a circular **memory buffer that it writes the output** to.

- The **buffer is 100 MB by default**, When the contents of the **buffer reaches a certain threshold size (io.sort.spill.per cent, default 0.80, or 80%), a background thread will start to spill the contents to disk.**

- Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete.

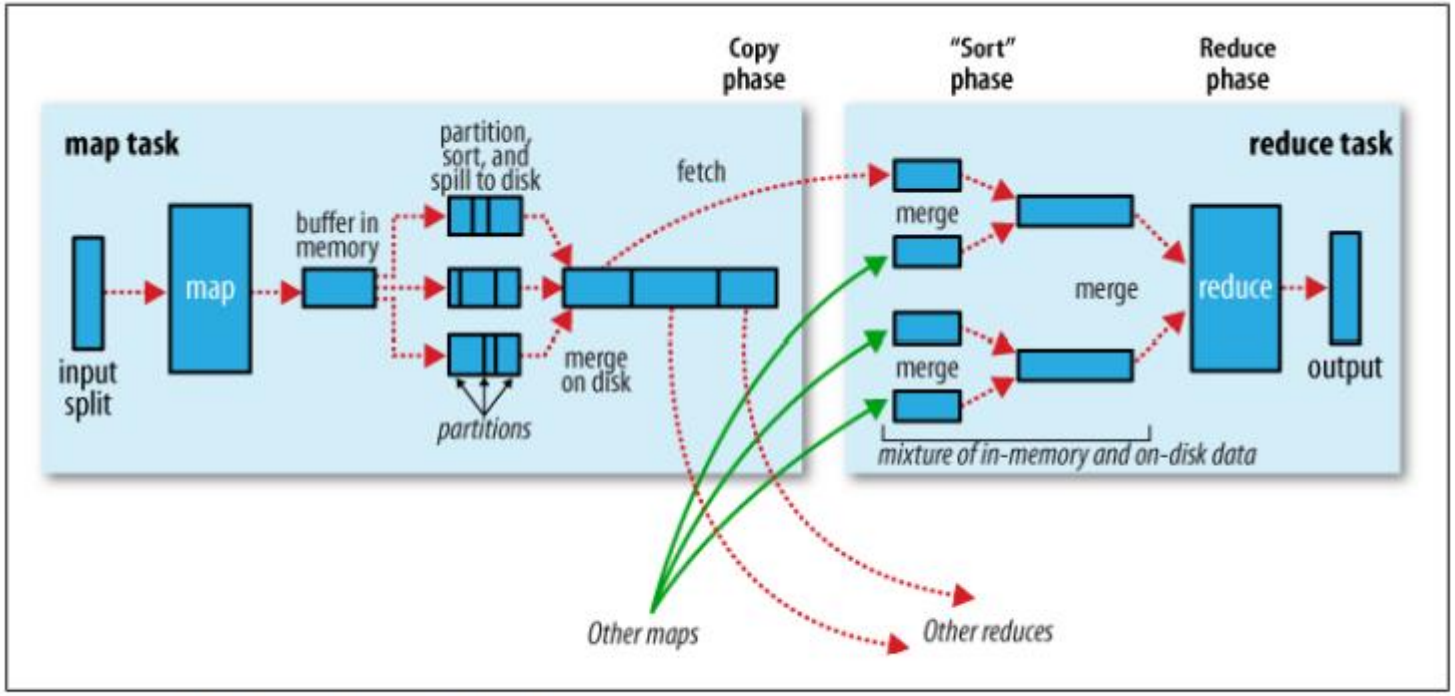


Figure 6-6. Shuffle and sort in MapReduce

Spills are written in round-robin fashion to the directories specified by the **mapred.local.dir property**, in a job-specific subdirectory.

- Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to.

- **Within each partition, the background thread performs an in-memory sort by key**, and if there is a combiner function, it is run on the output of the sort.

- Running the combiner function makes for a more compact map output, so there is less data to write to local disk and to transfer to the reducer.

- Each time the memory buffer reaches the spill threshold, a new spill file is created, so after the map task has written its last output record there could be several spill files.

- Before the task is finished, the **spill files are merged into a single partitioned and sorted output file.**

- **It is often a good idea to compress the map output as it is written to disk**, since doing so **makes it faster to write to disk, saves disk space**, and reduces the amount of data to transfer to the reducer.

- The output file's partitions are made available to the reducers over HTTP. The maximum number of worker threads used to serve the file partitions is controlled by the tasktracker.http.threads property—this setting is per tasktracker, not per map task slot.

The Reduce Side

- Let's turn now to the reduce part of the process.

- **The map output file is sitting on the local disk of the machine that ran the map task** (note that although map outputs always get written to local disk, reduce outputs may not be), but now it is needed by the machine that is about to run the reduce task for the partition.

- Furthermore, the **reduce task needs the map output for its particular partition from several map tasks across the cluster**.

- The **map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes.** This is known as the copy phase of the reduce task.

- The reduce task has a small number of copier threads so that it can fetch map outputs in parallel. **The default is five threads**, but this number can be changed by setting the **mapred.reduce.parallel.copies property**.

- The map outputs are copied to the reduce task JVM's memory if they are small enough; otherwise, they are copied to disk.

- **When the in-memory buffer reaches a threshold size or reaches a threshold number of map outputs ,it is merged and spilled to disk**. If a combiner is specified it will be run during the merge to reduce the amount of data written to disk

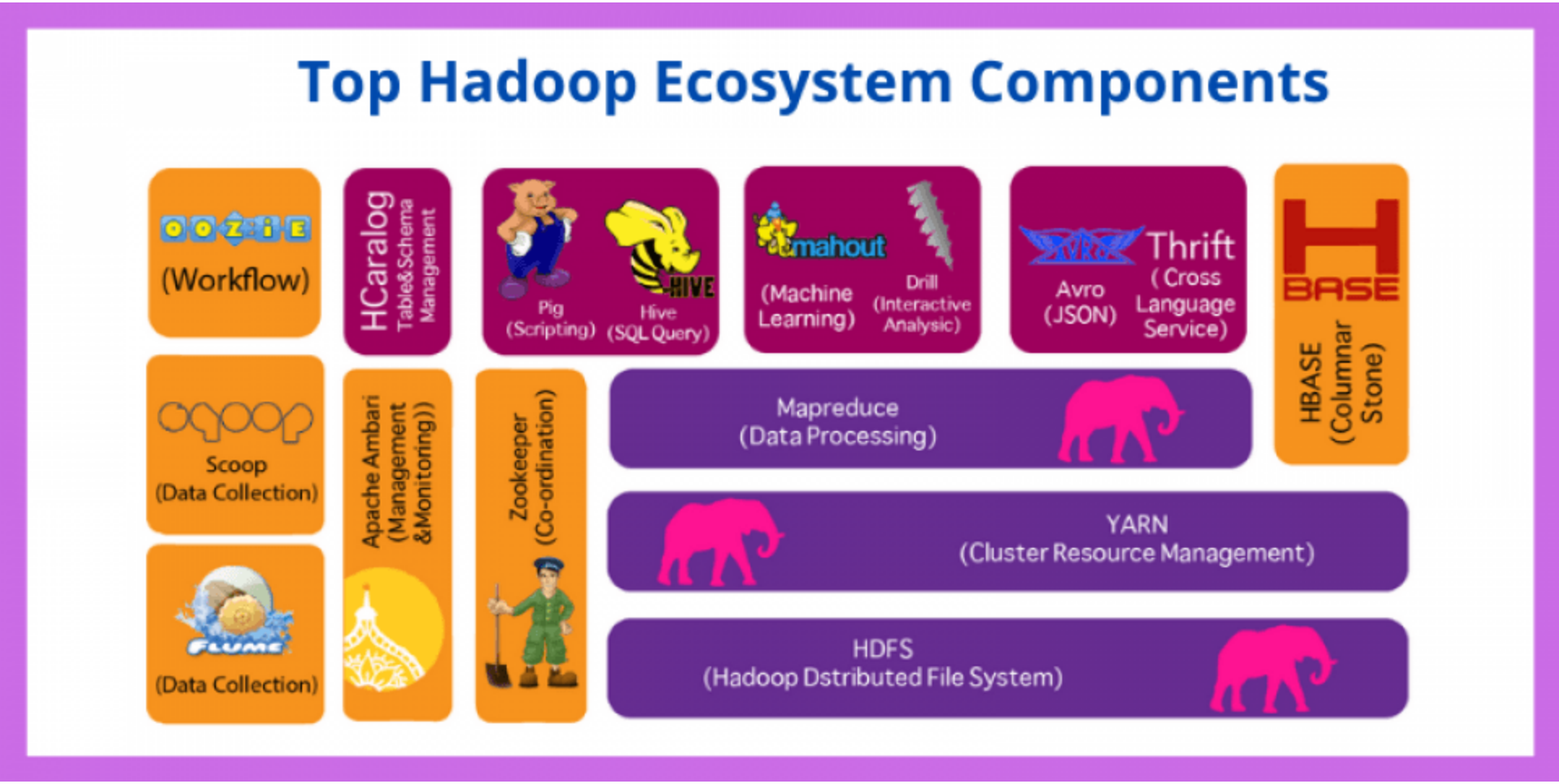
- As the copies accumulate on disk, a background thread merges them into larger, sorted files. This saves some time merging later on. Note that any map outputs that were compressed (by the map task) have to be decompressed in memory in order to perform a merge on them.

- **When all the map outputs have been copied, the reduce task moves into the sort phase** (which should properly be called the merge phase, as the sorting was carried out on the map side), which merges the map outputs, maintaining their sort ordering. This is done in rounds.

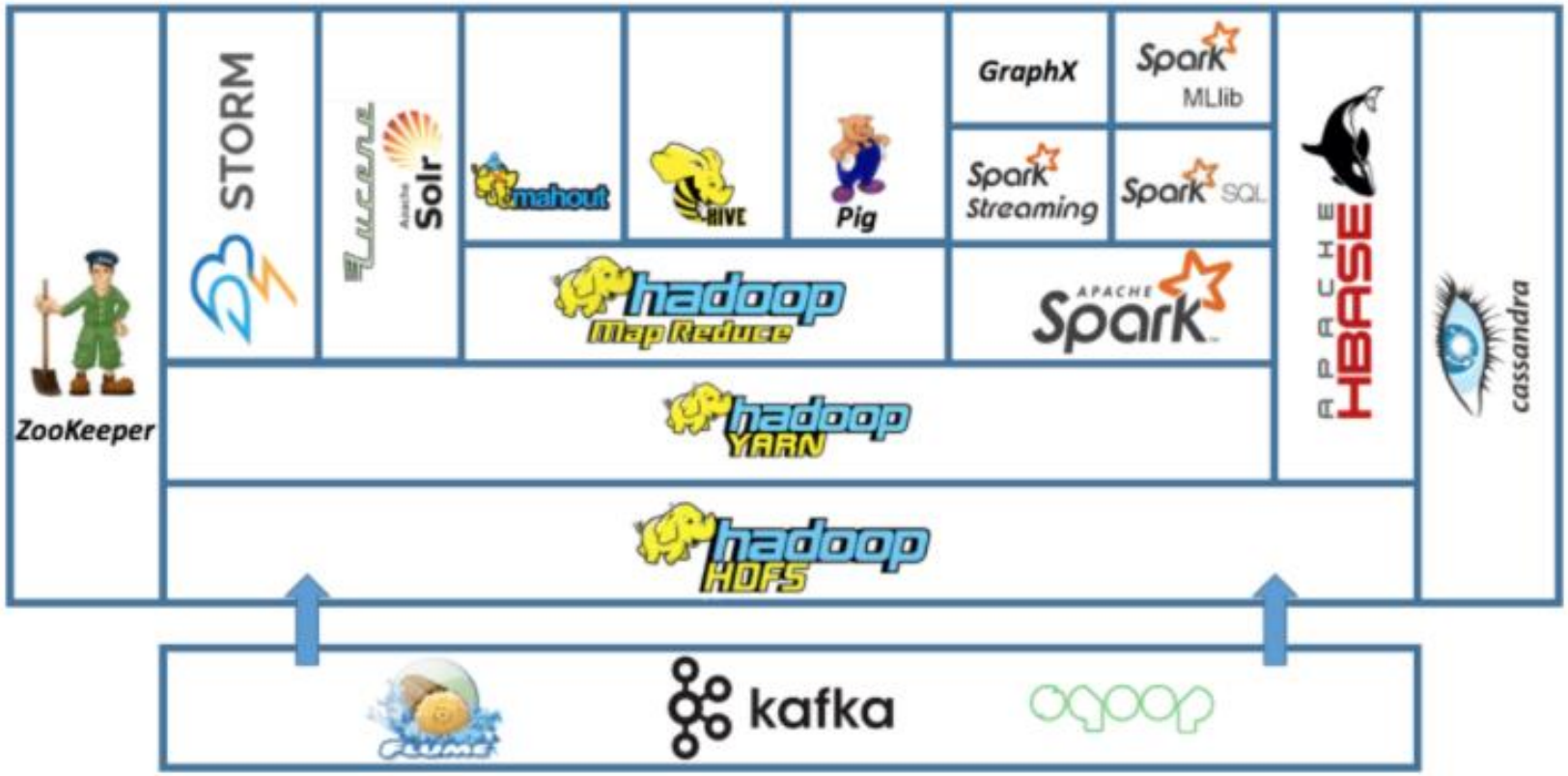
- Rather than have a final round that merges these five files into a single sorted file, the merge saves a trip to disk by directly feeding the reduce function in what is the last phase: the reduce phase.

- During the reduce phase, **the reduce function is invoked for each key in the sorted output**. The output of this phase is written directly to the output filesystem, typically HDFS.

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



Hadoop Components



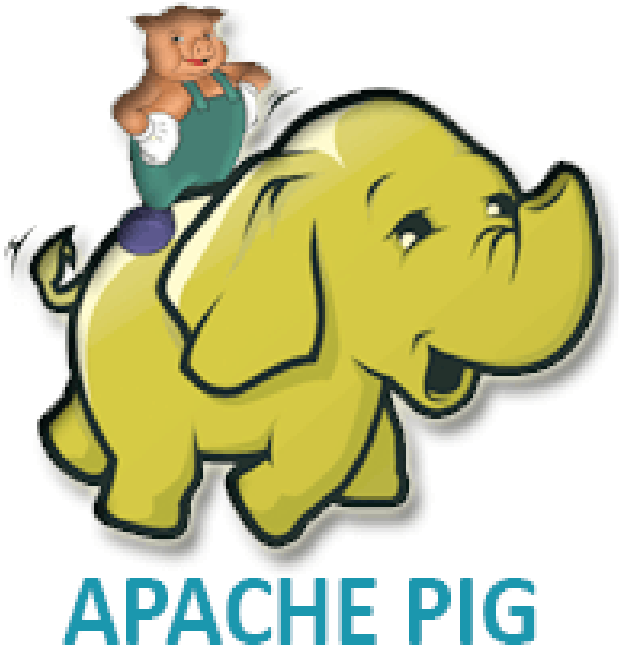
Apache Pig: Introduction

Apache Pig is **a platform for analyzing large data sets** that consists of a high-level language for expressing data analysis programs.

Pig is a high-level platform or tool which is used to process the large datasets**. It provides a high-level of abstraction for processing over the MapReduce**.

- It provides a high-level scripting language, known as **Pig Latin** which is used to develop the data analysis codes

Pig Hadoop is basically **a high-level programming language that is helpful for the analysis of huge datasets.** Pig Hadoop was **developed by Yahoo!**



Pig-Data Model: Introduction

- Pig Hadoop is basically a high-level programming language that is helpful for the analysis of huge datasets. Pig Hadoop was developed by Yahoo! and is generally used with Hadoop to perform a lot of data administration operations.

- **Pig is made up of two pieces:**

  • The language used to express data flows, called **Pig Latin.**

  • The execution environment to run Pig Latin programs. There are currently two environments:
    - local execution in **a single JVM** and
    - distributed execution on **a Hadoop cluster.**

- **Pig is a scripting language for exploring large datasets.**

- Pig is very supportive of a programmer writing a query, since it provides several commands for introspecting the data structures in your program as it is written.

https://pig.apache.org/

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.**

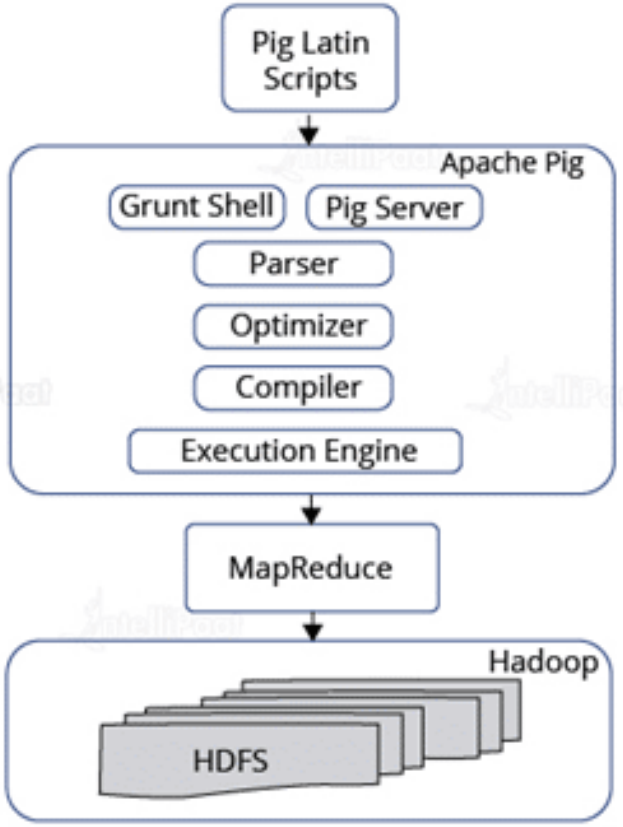
A Runtime Engine: Translate Pig Latin into MapReduce Programming

- 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 Architecture

There are several features of Apache Pig:

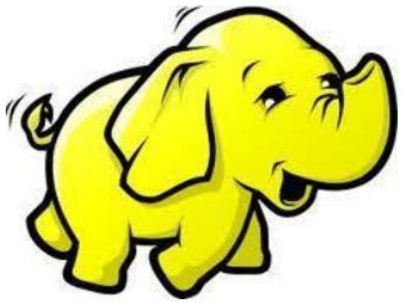
- **In-built operators**: Apache Pig provides a very good set of operators for performing several data operations like sort, join, filter, etc.
- **Ease of programming:** Since Pig Latin has similarities with SQL, it is very easy to write a Pig script.
- **Automatic optimization:** The tasks in Apache Pig are automatically optimized. This makes the programmers concentrate only on the semantics of the language.
- **Handles all kinds of data**: Apache Pig can analyze both structured and unstructured data and store the results in HDFS.



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**

PIG vs MapReduce



MapReduce

- Powerful model for parallelism.
- Based on a rigid procedural structure.
- Provides a good opportunity to parallelize algorithm.
- Must think in terms of map and reduce functions
- More than likely will require Java programmers

PIG

- Higher level procedural data flow language
- Similar to SQL query where the user specifies the what and leaves the “how” to the underlying processing engine.

There are three ways of executing Pig programs, all of which work in both local and MapReduce mode:

Script

- **Pig can run a script file that contains Pig commands**. For example, pig script.pig runs the commands in the local file script.pig. Alternatively, for very short scripts, you can use the -e option to run a script specified as a string on the command line.

Grunt

- **Grunt is an interactive shell for running Pig commands**. Grunt is started when no file is specified for Pig to run and the -e option is not used. It is also possible to run Pig scripts from within Grunt using run and exec.

Embedded

- **You can run Pig programs from Java using the PigServer class**, much like you can use JDBC to run SQL programs from Java. For programmatic access to Grunt, use PigRunner.

Pig Latin:Developing and Testing Pig Latin Scripts.

- The **syntax and semantics of the Pig Latin** programming language.

- Structure A Pig Latin program consists of a collection of statements. A statement can be thought of as an operation or a command. For example, a GROUP operation is a type of statement:

```
grouped_records = GROUP records BY year;
```

The command to list the files in a Hadoop filesystem is another example of a statement:

```
ls /
```

- **Pig Latin has two forms of comments**. Double hyphens are used for single-line comments. Everything from the first hyphen to the end of the line is ignored by the Pig Latin interpreter:

```
-- My program
DUMP A; -- What's in A?
```

- C-style comments are more flexible since they delimit the beginning and end of the comment block with /* and */ markers. They can span lines or be embedded in a single line:

```
/*
 * Description of my program spanning
 * multiple lines.
 */
A = LOAD 'input/pig/join/A';
B = LOAD 'input/pig/join/B';
C = JOIN A BY $0, /* ignored */ B BY $1;
DUMP C;
```

Table 16-4. Pig Latin commands

Category	Command	Description
Hadoop filesystem	`cat`	Prints the contents of one or more files
	`cd`	Changes the current directory
	`copyFromLocal`	Copies a local file or directory to a Hadoop filesystem
	`copyToLocal`	Copies a file or directory on a Hadoop filesystem to the local filesystem
	`cp`	Copies a file or directory to another directory
	`fs`	Accesses Hadoop's filesystem shell
	`ls`	Lists files
	`mkdir`	Creates a new directory
	`mv`	Moves a file or directory to another directory
	`pwd`	Prints the path of the current working directory
	`rm`	Deletes a file or directory
	`rmf`	Forcibly deletes a file or directory (does not fail if the file or directory does not exist)
Hadoop MapReduce	`kill`	Kills a MapReduce job

Developing and testing Pig Latin scripts

Table 16-1. Pig Latin relational operators

Category	Operator	Description
Loading and storing	LOAD	Loads data from the filesystem or other storage into a relation
	STORE	Saves a relation to the filesystem or other storage
	DUMP (\d)	Prints a relation to the console
Filtering	FILTER	Removes unwanted rows from a relation
	DISTINCT	Removes duplicate rows from a relation
	FOREACH...GENERATE	Adds or removes fields to or from a relation
	MAPREDUCE	Runs a MapReduce job using a relation as input
	STREAM	Transforms a relation using an external program
	SAMPLE	Selects a random sample of a relation
	ASSERT	Ensures a condition is true for all rows in a relation; otherwise, fails
Grouping and joining	JOIN	Joins two or more relations
	COGROUP	Groups the data in two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates the cross product of two or more relations
	CUBE	Creates aggregations for all combinations of specified columns in a relation
Sorting	ORDER	Sorts a relation by one or more fields
	RANK	Assign a rank to each tuple in a relation, optionally sorting by fields first
	LIMIT	Limits the size of a relation to a maximum number of tuples
Combining and splitting	UNION	Combines two or more relations into one
	SPLIT	Splits a relation into two or more relations

Category	Command	Description
Utility	`clear`	Clears the screen in Grunt
	`exec`	Runs a script in a new Grunt shell in batch mode
	`help`	Shows the available commands and options
	`history`	Prints the query statements run in the current Grunt session
	`quit` (`\q`)	Exits the interpreter
	`run`	Runs a script within the existing Grunt shell
	`set`	Sets Pig options and MapReduce job properties
	`sh`	Runs a shell command from within Grunt

Table 16-2. Pig Latin diagnostic operators

Operator (Shortcut)	Description
`DESCRIBE` (`\de`)	Prints a relation's schema
`EXPLAIN` (`\e`)	Prints the logical and physical plans
`ILLUSTRATE` (`\i`)	Shows a sample execution of the logical plan, using a generated subset of the input

Table 16-5. Pig Latin expressions

Category	Expressions	Description	Examples
Constant	Literal	Constant value (see also the "Literal example" column in Table 16-6)	`1.0`, `'a'`
Field (by position)	`$n`	Field in position *n* (zero-based)	`$0`
Field (by name)	*f*	Field named *f*	`year`
Field (disambiguate)	*r*`::`*f*	Field named *f* from relation *r* after grouping or joining	`A::year`
Projection	*c*`.$`*n*, *c*`.`*f*	Field in container *c* (relation, bag, or tuple) by position, by name	`records.$0`, `records.year`
Map lookup	*m*`#`*k*	Value associated with key *k* in map *m*	`items#'Coat'`
Cast	`(`*t*`)` *f*	Cast of field *f* to type *t*	`(int) year`
Arithmetic	*x* `+` *y*, *x* `-` *y*	Addition, subtraction	`$1 + $2`, `$1 - $2`
	x `*` *y*, *x* `/` *y*	Multiplication, division	`$1 * $2`, `$1 / $2`
	x `%` *y*	Modulo, the remainder of *x* divided by *y*	`$1 % $2`
	`+`*x*, `-`*x*	Unary positive, negation	`+1`, `-1`
Conditional	*x* `?` *y* `:` *z*	Bincond/ternary; *y* if *x* evaluates to true, *z* otherwise	`quality == 0 ? 0 : 1`
	CASE	Multi-case conditional	`CASE q WHEN 0 THEN 'good' ELSE 'bad' END`

Category	Expressions	Description	Examples
Comparison	*x* == *y*, *x* != *y*	Equals, does not equal	quality == 0, tempera ture != 9999
	x > *y*, *x* < *y*	Greater than, less than	quality > 0, quality < 10
	x >= *y*, *x* <= *y*	Greater than or equal to, less than or equal to	quality >= 1, quality <= 9
	x matches *y*	Pattern matching with regular expression	quality matches '[01459]'
	x is null	Is null	temperature is null
	x is not null	Is not null	temperature is not null
Boolean	*x* OR *y*	Logical OR	q == 0 OR q == 1
	x AND *y*	Logical AND	q == 0 AND r == 0
	NOT *x*	Logical negation	NOT q matches '[01459]'
	IN *x*	Set membership	q IN (0, 1, 4, 5, 9)
Functional	*fn*(*f1*,*f2*,...)	Invocation of function *fn* on fields *f1*, *f2*, etc.	isGood(quality)
Flatten	FLATTEN(*f*)	Removal of a level of nesting from bags and tuples	FLATTEN(group)

Table 16-6. Pig Latin types

Category	Type	Description	Literal example
Boolean	`boolean`	True/false value	`true`
Numeric	`int`	32-bit signed integer	`1`
	`long`	64-bit signed integer	`1L`
	`float`	32-bit floating-point number	`1.0F`
	`double`	64-bit floating-point number	`1.0`
	`biginteger`	Arbitrary-precision integer	`'10000000000'`
	`bigdecimal`	Arbitrary-precision signed decimal number	`'0.11000100000000000000000001'`
Text	`chararray`	Character array in UTF-16 format	`'a'`
Binary	`bytearray`	Byte array	Not supported
Temporal	`datetime`	Date and time with time zone	Not supported, use `ToDate` built-in function
Complex	`tuple`	Sequence of fields of any type	`(1,'pomegranate')`
	`bag`	Unordered collection of tuples, possibly with duplicates	`{(1,'pomegranate'),(2)}`
	`map`	Set of key-value pairs; keys must be character arrays, but values may be any type	`['a'#'pomegranate']`

Table 16-7. A selection of Pig's built-in functions

Category	Function	Description
Eval	AVG	Calculates the average (mean) value of entries in a bag.
	CONCAT	Concatenates byte arrays or character arrays together.
	COUNT	Calculates the number of non-`null` entries in a bag.
	COUNT_STAR	Calculates the number of entries in a bag, including those that are `null`.
	DIFF	Calculates the set difference of two bags. If the two arguments are not bags, returns a bag containing both if they are equal; otherwise, returns an empty bag.
	MAX	Calculates the maximum value of entries in a bag.
	MIN	Calculates the minimum value of entries in a bag.
	SIZE	Calculates the size of a type. The size of numeric types is always 1; for character arrays, it is the number of characters; for byte arrays, the number of bytes; and for containers (tuple, bag, map), it is the number of entries.
	SUM	Calculates the sum of the values of entries in a bag.
	TOBAG	Converts one or more expressions to individual tuples, which are then put in a bag. A synonym for `()`.
	TOKENIZE	Tokenizes a character array into a bag of its constituent words.
	TOMAP	Converts an even number of expressions to a map of key-value pairs. A synonym for `[]`.
	TOP	Calculates the top *n* tuples in a bag.
	TOTUPLE	Converts one or more expressions to a tuple. A synonym for `{}`.
Filter	IsEmpty	Tests whether a bag or map is empty.
Load/Store	PigStorage	Loads or stores relations using a field-delimited text format. Each line is broken into fields using a configurable field delimiter (defaults to a tab character) to be stored in the tuple's fields. It is the default storage when none is specified.[a]

Installing and Running Pig

Downloading Pig

Download the latest version of Pig by visiting the following link
https://dlcdn.apache.org/pig/pig-0.17.0/pig-0.17.0.tar.gz .

1.Create a new Folder inside Desktop , name the Folder as your USN <1ms18cs017>.

2.Move the Downloaded Pig File to USN <1ms18cs017> Folder.

3.Right Click on that File and Extract inside the USN <1ms18cs017> Folder.

Open Terminal

Navigate to Extracted Hadoop Folder cd ~/Desktop/1MS18CS017/pig-0.17.0

Create a New File named Bash.sh

Copy the Below code and Paste inside Bash.sh and save that File.

```
export JAVA_HOME=$(readlink -f $(which javac) | awk 'BEGIN {FS="/bin"} {print $1}')
if ! command -v pig &> /dev/null
then
    export PATH=$(echo $PATH):$(pwd)/bin
fi
```

Installing and Running Pig

```
root@kali:~/Desktop/1MS18CS017/pig-0.17.0# vi Bash.sh
root@kali:~/Desktop/1MS18CS017/pig-0.17.0# cat Bash.sh
export JAVA_HOME=$(readlink -f $(which javac) | awk 'BEGIN {FS="/bin"} {print $1}')
if ! command -v pig  &> /dev/null
then
    export PATH=$(echo $PATH):$(pwd)/bin
fi

root@kali:~/Desktop/1MS18CS017/pig-0.17.0# source Bash.sh
root@kali:~/Desktop/1MS18CS017/pig-0.17.0#
```

To launch pig hadoop should be configured.

```
root@kali:~/Desktop/1MS18CS017/pig-0.17.0# echo $PATH
/root/.nvm/versions/node/v14.17.0/bin:/root/anaconda3/condabin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin:/root/Desktop/1MS18CS0
17/pig-0.17.0/bin
root@kali:~/Desktop/1MS18CS017/pig-0.17.0# pig
2022-07-03 20:31:32,276 INFO  [main] pig.ExecTypeProvider (ExecTypeProvider.java:selectExecType(41)) - Trying ExecType : LOCAL
2022-07-03 20:31:32,277 INFO  [main] pig.ExecTypeProvider (ExecTypeProvider.java:selectExecType(41)) - Trying ExecType : MAPREDUCE
2022-07-03 20:31:32,277 INFO  [main] pig.ExecTypeProvider (ExecTypeProvider.java:selectExecType(43)) - Picked MAPREDUCE as the ExecType
2022-07-03 20:31:32,300 [main] INFO  org.apache.pig.Main - Apache Pig version 0.17.0 (r1797386) compiled Jun 02 2017, 15:41:58
2022-07-03 20:31:32,300 [main] INFO  org.apache.pig.Main - Logging error messages to: /root/Desktop/1MS18CS017/pig-0.17.0/pig_1656660492296.log
2022-07-03 20:31:32,310 [main] INFO  org.apache.pig.impl.util.Utils - Default bootup file /root/.pigbootup not found
2022-07-03 20:31:32,331 [main] ERROR org.apache.pig.Main - ERROR 4010: Cannot find hadoop configurations in classpath (neither hadoop-site.xml nor core-site
.xml was found in the classpath). If you plan to use local mode, please put -x local option in command line
Details at logfile: /root/Desktop/1MS18CS017/pig-0.17.0/pig_1656660492296.log
2022-07-03 20:31:32,340 [main] INFO  org.apache.pig.Main - Pig script completed in 103 milliseconds (103 ms)
root@kali:~/Desktop/1MS18CS017/pig-0.17.0#
```

Installing and Running Pig

set up Hadoop

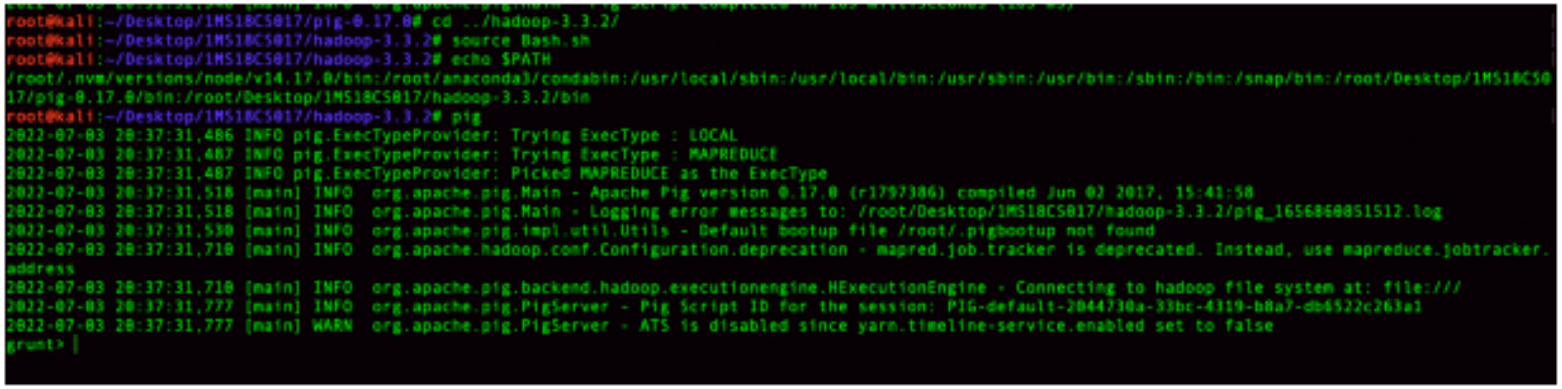
1.Navigate to Hadoop Folder to set up Hadoop.

```
cd ~/Desktop/<1ms18cs017>/hadoop-3.3.2
```

2.Execute Bash.sh inside hadoop Folder

```
source Bash.sh
```

After this u can see the Hadoop Folder inside PATH Variable



Basic Pig Commands

- **Fs: This will list all the file in the HDFS**

grunt> fs –ls

- **Clear: This will clear the interactive Grunt shell.**

grunt> clear

- **Reading Data: Assuming the data resides in HDFS, and we need to read data to Pig.**

grunt> **college_students** = **LOAD** ‘hdfs://localhost:9000/pig_data/college_data.txt’

USING PigStorage(‘,’)

as (**id**:int, **firstname**:chararray, **lastname**:chararray, **phone**:chararray, **city**:chararray);

PigStorage() : is the function that loads and stores data as structured text files.

Basic Pig Commands

- **Storing Data: Store operator is used to storing the processed/loaded data.**

grunt> **STORE college_students INTO** ‘ hdfs://localhost:9000/pig_Output/ ‘ USING PigStorage (‘,’);

Here, “/pig_Output/” is the directory where relation needs to be stored.

- **Dump Operator: This command is used to display the results on screen. It usually helps in debugging.**

grunt> **Dump college_students;**

- **Describe Operator: It helps the programmer to view the schema of the relation.**

grunt> **describe** college_students;

Reference: https://www.educba.com/pig-commands/

Basic Pig Commands

- **Group: This command works towards grouping data with the same key.**

grunt> **group_data = GROUP** college_students **by** first name;

- **Join: This is used to combine two or more relations.**

Example: In order to perform self-join, let's say relation "customer" is loaded from HDFS tp pig commands in two relations customers1 & customers2.

grunt> customers3 = **JOIN** customers1 **BY** id, customers2 **BY** id;

Join could be self-join, Inner-join, Outer-join.

- **Union: It merges two relations. The condition for merging is that both the relation's columns and domains must be identical.**

grunt> **student = UNION student1, student2;**

Basic Pig Commands

- **Filter: This helps in filtering out the tuples out of relation, based on certain conditions.**

filter_data **= FILTER college_student**s **BY** city == ‘Chennai’;

- **Distinct: This helps in the removal of redundant tuples from the relation.**

grunt> distinct_data = **DISTINCT** college_students;

This filtering will create a new relation name “distinct_data”

- **Foreach: This helps in generating data transformation based on column data.**

grunt> foreach_data = **FOREACH** student_details **GENERATE** id,age,city;

This will get the id, age, and city values of each student from the relation student_details and hence will store it into another relation named foreach_data.

Basic Pig Commands

Order by: This command displays the result in a sorted order based on one or more fields.

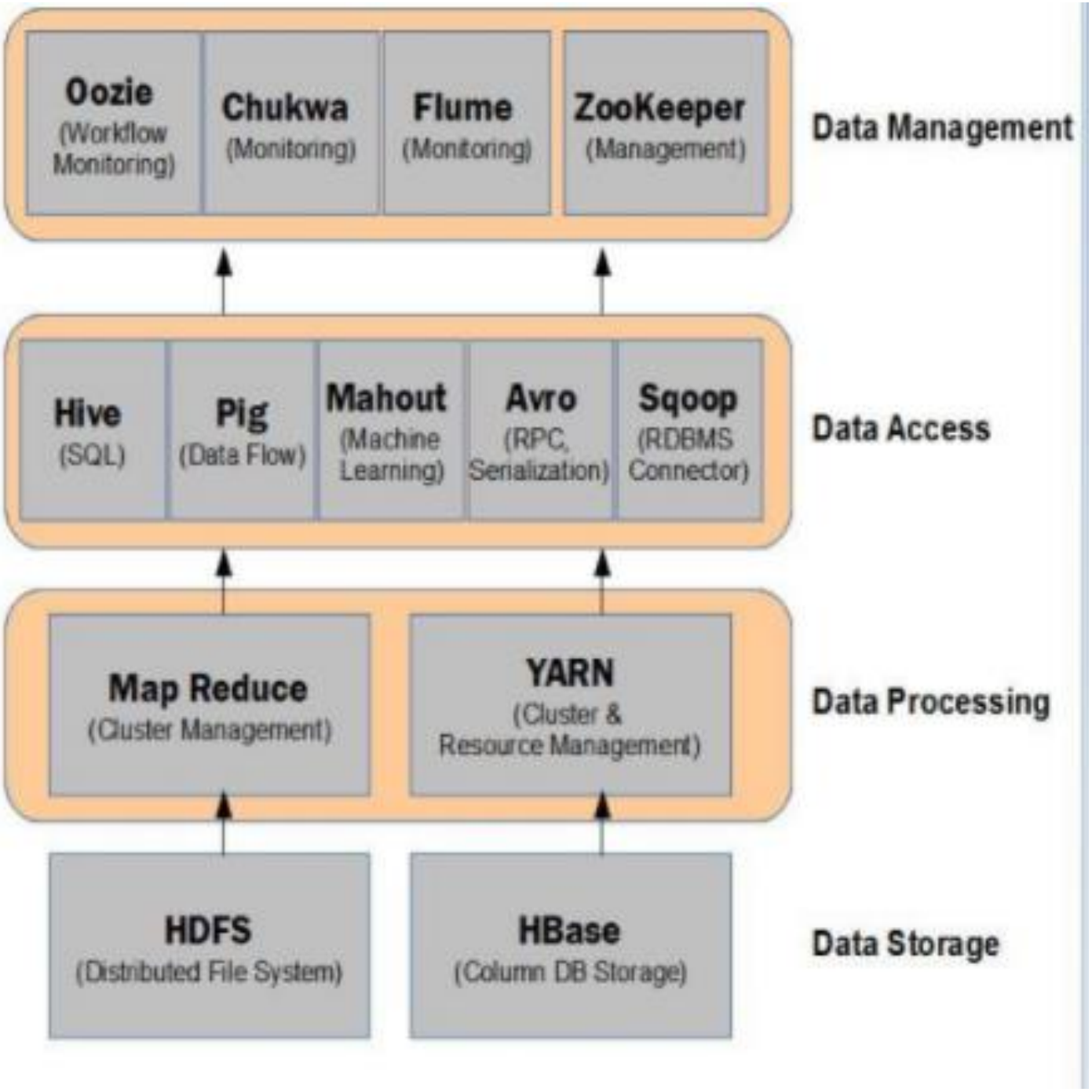
grunt> order_by_data = **ORDER** college_students **BY** age **DESC**;

https://pig.apache.org/docs/latest/start.html

Hadoop Basic PIG Commands with Examples

1. **Run PIG command from console (cluster mode)**
2. Data Input using pig: **Load data** from hdfs to Pig
3. **Dump Command:** This command is used to display all data loaded.
4. **Foreach:** This command is used to generate data transformation based on columns of data
5. **Filter:** Select particular tuples from a relation based on a condition.
6. **Order By:** Sort a relation based on one or more fields
7. **Store:** Save results to the local file system or HDFS
8. **Cogroup:** This operator is used to group two databases using a particular column.
9. **Join:** This operator is used to join two or more table.
10. **Cross:** The CROSS operator computes the cross-product of two or more relations.
11. **Union:** The UNION operator of Pig Latin is used to merge the content of two relations.

Reference: https://prwatech.in/blog/hadoop/hadoop-basic-pig-commands-with-examples/


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

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**

Features of Apache Hive

- **Open-source:** Apache Hive is an open-source tool. We can use it free of cost.
- **Query large datasets**: Hive can query and manage huge datasets stored in Hadoop Distributed File System.
- **Hive Query Language:** Hive uses Hive Query Language which is similar to SQL. We do not require any knowledge of programming languages to work with Hive. Only the knowledge of basic SQL query is enough to work with Hive
- **Ad-hoc queries:** Hive allows us to run Ad-hoc queries which are the loosely typed command or query whose value depends on some variable for the data analysis.
- **User-Defined Functions:** It also provides support for User-Defined Functions for the tasks like data cleansing and filtering. We can define UDFs according to our requirements

Apache Hive: Architecture

The Apache Hive is **data warehouse software facilitates reading, writing, and managing large datasets** residing in distributed storage using SQL.

Hive is built on top of Apache Hadoop, which is an open-source framework used to efficiently store and process large datasets

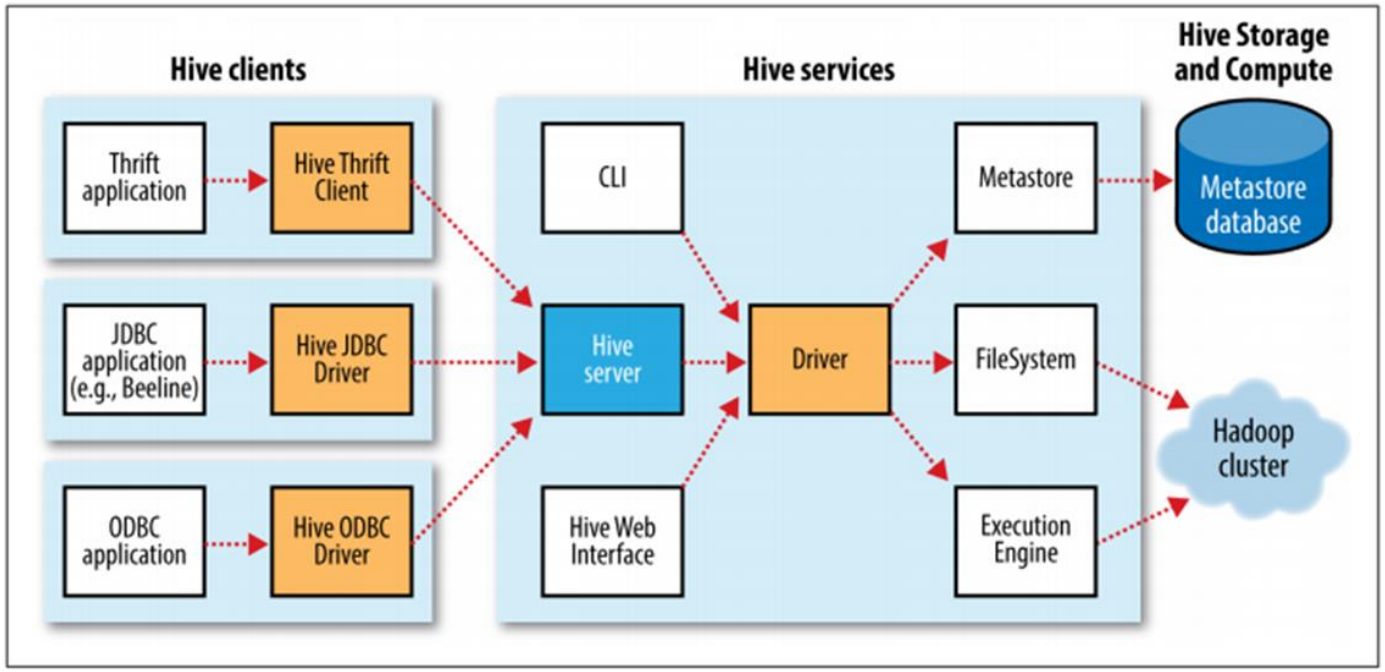
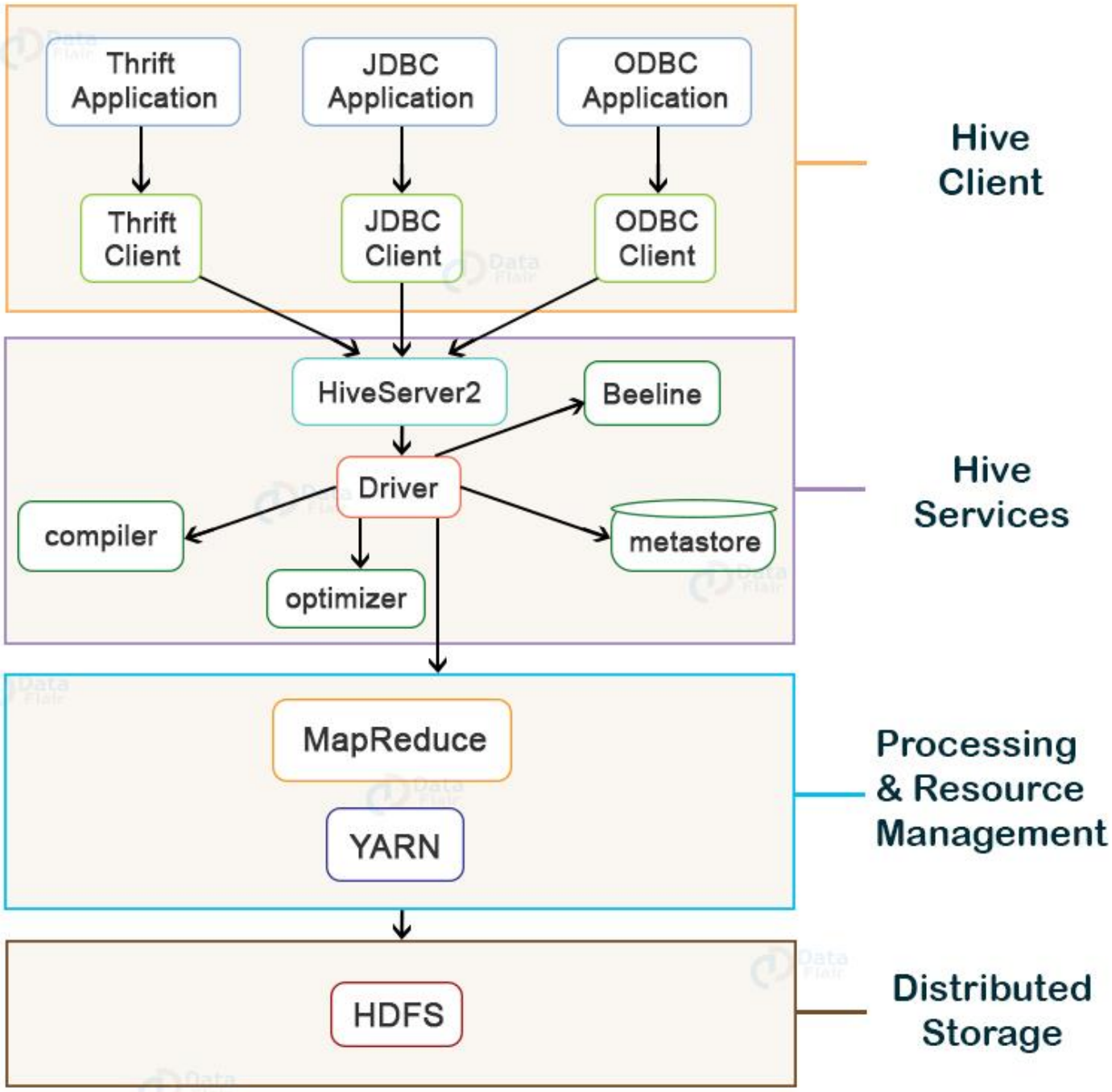


Figure 17-1. Hive architecture

Core Components: Architecture

Hive consists of three core parts:

- **Hive Clients**: Hive offers a variety of drivers designed for communication with different applications.
    - For example, Hive provides Thrift clients for **Thrift-based applications**. These clients and drivers then communicate with the Hive server, which falls under Hive services.
- **Hive Services:** Hive services perform client interactions with Hive. For example, if a client wants to perform a query, it must talk with Hive services.
- **Hive Storage and Computing:** Hive services such as file system, job client, and meta store then communicates with Hive storage and stores things like metadata table information and query results.


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

HiveQL Data Manipulation and HiveQL Queries.

- **Hive supports both primitive and complex data types**. Primitives include numeric, Boolean, string, and timestamp types. The complex data types include arrays, maps, and structs

Table 17-3. Hive data types

Category	Type	Description	Literal examples
Primitive	BOOLEAN	True/false value.	TRUE
	TINYINT	1-byte (8-bit) signed integer, from –128 to 127.	1Y
	SMALLINT	2-byte (16-bit) signed integer, from –32,768 to 32,767.	1S
	INT	4-byte (32-bit) signed integer, from –2,147,483,648 to 2,147,483,647.	1
	BIGINT	8-byte (64-bit) signed integer, from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	1L
	FLOAT	4-byte (32-bit) single-precision floating-point number.	1.0
	DOUBLE	8-byte (64-bit) double-precision floating-point number.	1.0
	DECIMAL	Arbitrary-precision signed decimal number.	1.0
	STRING	Unbounded variable-length character string.	'a', "a"
	VARCHAR	Variable-length character string.	'a', "a"
	CHAR	Fixed-length character string.	'a', "a"
	BINARY	Byte array.	Not supported
	TIMESTAMP	Timestamp with nanosecond precision.	1325502245000, '2012-01-02 03:04:05.123456789'
	DATE	Date.	'2012-01-02'

Category	Type	Description	Literal examples
Complex	ARRAY	An ordered collection of fields. The fields must all be of the same type.	`array(1, 2)`[a]
	MAP	An unordered collection of key-value pairs. Keys must be primitives; values may be any type. For a particular map, the keys must be the same type, and the values must be the same type.	`map('a', 1, 'b', 2)`
	STRUCT	A collection of named fields. The fields may be of different types.	`struct('a', 1, 1.0)`,[b] `named_struct('col1', 'a', 'col2', 1, 'col3', 1.0)`
	UNION	A value that may be one of a number of defined data types. The value is tagged with an integer (zero-indexed) representing its data type in the union.	`create_union(1, 'a', 63)`

Table 17-2. A high-level comparison of SQL and HiveQL

Feature	SQL	HiveQL	References
Updates	`UPDATE`, `INSERT`, `DELETE`	`UPDATE`, `INSERT`, `DELETE`	"Inserts" on page 500; "Updates, Transactions, and Indexes" on page 483
Transactions	Supported	Limited support	
Indexes	Supported	Supported	
Data types	Integral, floating-point, fixed-point, text and binary strings, temporal	Boolean, integral, floating-point, fixed-point, text and binary strings, temporal, array, map, struct	"Data Types" on page 486
Functions	Hundreds of built-in functions	Hundreds of built-in functions	"Operators and Functions" on page 488
Multitable inserts	Not supported	Supported	"Multitable insert" on page 501
`CREATE TABLE...AS SELECT`	Not valid SQL-92, but found in some databases	Supported	"CREATE TABLE...AS SELECT" on page 501
`SELECT`	SQL-92	SQL-92. `SORT BY` for partial ordering, `LIMIT` to limit number of rows returned	"Querying Data" on page 503
Joins	SQL-92, or variants (join tables in the `FROM` clause, join condition in the `WHERE` clause)	Inner joins, outer joins, semi joins, map joins, cross joins	"Joins" on page 505
Subqueries	In any clause (correlated or noncorrelated)	In the `FROM`, `WHERE`, or `HAVING` clauses (uncorrelated subqueries not supported)	"Subqueries" on page 508
Views	Updatable (materialized or nonmaterialized)	Read-only (materialized views not supported)	"Views" on page 509

Hive Data Manipulation : Load, Insert, Update, Delete

Examples

```
CREATE TABLE students (name VARCHAR(64), age INT, gpa DECIMAL(3, 2))
  CLUSTERED BY (age) INTO 2 BUCKETS STORED AS ORC;

INSERT INTO TABLE students
  VALUES ('fred flintstone', 35, 1.28), ('barney rubble', 32, 2.32);


CREATE TABLE pageviews (userid VARCHAR(64), link STRING, came_from STRING)
  PARTITIONED BY (datestamp STRING) CLUSTERED BY (userid) INTO 256 BUCKETS STORED AS ORC;

INSERT INTO TABLE pageviews PARTITION (datestamp = '2014-09-23')
  VALUES ('jsmith', 'mail.com', 'sports.com'), ('jdoe', 'mail.com', null);

INSERT INTO TABLE pageviews PARTITION (datestamp)
  VALUES ('tjohnson', 'sports.com', 'finance.com', '2014-09-23'), ('tlee', 'finance.com', null, '2014-09-21');

INSERT INTO TABLE pageviews
  VALUES ('tjohnson', 'sports.com', 'finance.com', '2014-09-23'), ('tlee', 'finance.com', null, '2014-09-21');
```

https://cwiki.apache.org/confluence/display/Hive/LanguageManual

Importing Data

- We've already seen how to use the LOAD DATA operation to import data into a Hive table (or partition) by copying or moving files to the table's directory.

- You can also populate a table with data from another Hive table using an INSERT statement, or at creation time using the CTAS construct, which is an abbreviation used to refer to CREATE TABLE...AS SELECT

INSERT OVERWRITE TABLE

Here's an example of an `INSERT` statement:

```
INSERT OVERWRITE TABLE target
SELECT col1, col2
  FROM source;
```

For partitioned tables, you can specify the partition to insert into by supplying a `PARTITION` clause:

```
INSERT OVERWRITE TABLE target
PARTITION (dt='2010-01-01')
SELECT col1, col2
  FROM source;
```

Multitable insert

In HiveQL, you can turn the `INSERT` statement around and start with the `FROM` clause, for the same effect:

```
FROM source
INSERT OVERWRITE TABLE target
  SELECT col1, col2;
```

The reason for this syntax becomes clear when you see that it's possible to have multiple `INSERT` clauses in the same query. This so-called *multitable insert* is more efficient than multiple `INSERT` statements, since the source table need only be scanned once to produce the multiple, disjoint outputs.

Here's an example that computes various statistics over the weather dataset:

```
FROM records2
INSERT OVERWRITE TABLE stations_by_year
  SELECT year, COUNT(DISTINCT station)
  GROUP BY year
INSERT OVERWRITE TABLE records_by_year
  SELECT year, COUNT(1)
  GROUP BY year
INSERT OVERWRITE TABLE good_records_by_year
  SELECT year, COUNT(1)
  WHERE temperature != 9999
    AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
  GROUP BY year;
```

Altering Tables

Since Hive uses the schema on read approach, it's flexible in permitting a table's definition to change after the table has been created. The general caveat, however, is that it is up to you, in many cases, to ensure that the data is changed to reflect the new structure.

You can rename a table using the `ALTER TABLE` statement:

```
ALTER TABLE source RENAME TO target;
```

In addition to updating the table metadata, `ALTER TABLE` moves the underlying table directory so that it reflects the new name. In the current example, */user/hive/warehouse/source* is renamed to */user/hive/warehouse/target*. (An external table's underlying directory is not moved; only the metadata is updated.)

Hive allows you to change the definition for columns, add new columns, or even replace all existing columns in a table with a new set.

For example, consider adding a new column:

```
ALTER TABLE target ADD COLUMNS (col3 STRING);
```

Querying Data

Sorting and Aggregating

```
hive> FROM records2
    > SELECT year, temperature
    > DISTRIBUTE BY year
    > SORT BY year ASC, temperature DESC;
1949    111
1949    78
1950    22
1950    0
1950    -11
```

MapReduce Scripts

- Using an approach like Hadoop Streaming, the TRANSFORM, MAP, and REDUCE clauses make it possible to invoke an external script or program from Hive.

- Suppose we want to use a script to filter out rows that don't meet some condition, such as the script in Example 12-1, which removes poor quality readings.

Example 12-1. Python script to filter out poor quality weather records

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  (year, temp, q) = line.strip().split()
  if (temp != "9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

We can use the script as follows:

```
hive> ADD FILE /path/to/is_good_quality.py;
hive> FROM records2
    > SELECT TRANSFORM(year, temperature, quality)
    > USING 'is_good_quality.py'
    > AS year, temperature;
1949    111
1949    78
1950    0
1950    22
1950    -11
```

Joins

One of the nice things about using Hive, rather than raw MapReduce, is that it makes performing commonly used operations very simple. Join operations are a case in point, given how involved they are to implement in MapReduce ("Joins" on page 281).

Inner joins

The simplest kind of join is the inner join, where each match in the input tables results in a row in the output. Consider two small demonstration tables: `sales`, which lists the names of people and the ID of the item they bought; and `things`, which lists the item ID and its name:

```
hive> SELECT * FROM sales;
Joe     2
Hank    4
Ali     0
Eve     3
Hank    2
hive> SELECT * FROM things;
2    Tie
4    Coat
3    Hat
1    Scarf
```

We can perform an inner join on the two tables as follows:

```
hive> SELECT sales.*, things.*
    > FROM sales JOIN things ON (sales.id = things.id);
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

Outer joins :Outer joins allow you to find nonmatches in the tables being joined.

```
hive> SELECT sales.*, things.*
    > FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);
Ali     0    NULL NULL
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

Notice that the row for Ali is now returned, and the columns from the `things` table are `NULL`, since there is no match.

Hive supports right outer joins, which reverses the roles of the tables relative to the left join. In this case, all items from the `things` table are included, even those that weren't purchased by anyone (a scarf):

```
hive> SELECT sales.*, things.*
    > FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);
NULL    NULL 1    Scarf
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

Finally, there is a full outer join, where the output has a row for each row from both tables in the join:

```
hive> SELECT sales.*, things.*
    > FROM sales FULL OUTER JOIN things ON (sales.id = things.id);
Ali     0    NULL NULL
NULL    NULL 1    Scarf
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

Subqueries

- A subquery is a SELECT statement that is embedded in another SQL statement. Hive has limited support for subqueries, only permitting a subquery in the FROM clause of a SELECT statement.
- The following query finds the mean maximum temperature for every year and weather station:

```
SELECT station, year, AVG(max_temperature)
FROM (
  SELECT station, year, MAX(temperature) AS max_temperature
  FROM records2
  WHERE temperature != 9999
    AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
  GROUP BY station, year

 ) mt
 GROUP BY station, year;
```

The subquery is used to find the maximum temperature for each station/date combination, then the outer query uses the `AVG` aggregate function to find the average of the maximum temperature readings for each station/date combination.

The outer query accesses the results of the subquery like it does a table, which is why the subquery must be given an alias (`mt`). The columns of the subquery have to be given unique names so that the outer query can refer to them.

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

Hbase - Data Model and Implementation.

HBase is a **column-oriented non-relational database management system** that runs on top of Hadoop Distributed File System (HDFS)

HBase has a number of features like:

- **Scalable:** HBase allows data to be scaled across various nodes as it is stored in HDFS.
- **Automatic failure support:** Write ahead Log across clusters are present that provides automatic support against failure.
- **Consistent read and write:** HBase provides consistent read and write of data.
- **JAVA API for client access**: HBase provides easy to use JAVA API for clients.
- **Block cache and Bloom filters**: It supports block cache and bloom filters for high volume query optimization.

Hbase – Data Model and Implementations

- **Applications store data into labeled tables. Tables are made of rows and columns.**
    - Table cells—the intersection of row and column coordinates—are versioned.
    - By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion.
    - A cell's content is an uninterpreted array of bytes
- **Row columns are grouped into column families.**
    - All column family members have a common prefix,
    - so, for example, the **columns temperature:air and temperature:dew_point** are both members of the temperature column family
    - whereas **station:identifier** belongs to the station family
- **Physically, all column family members are stored together on the filesystem.**
    - So, though earlier we described HBase as a column-oriented store, it would be more accurate if it were described as a column-family-oriented store

An example HBase table for storing photos is shown in Figure 20-1.

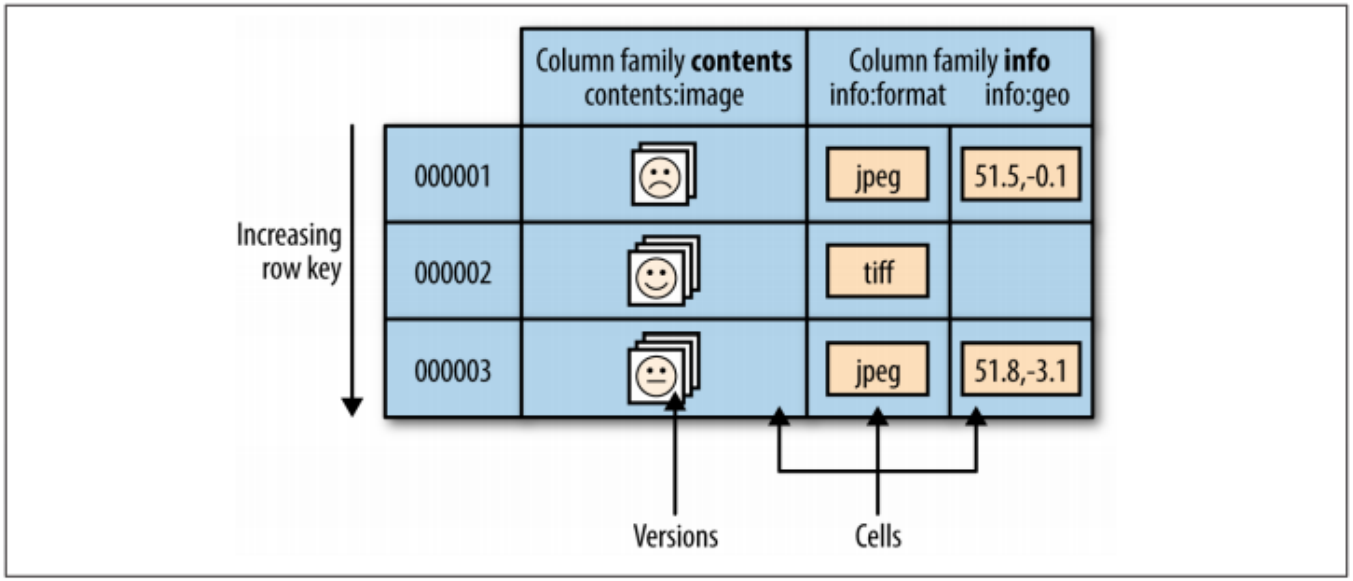


Figure 20-1. The HBase data model, illustrated for a table storing photos

- Row columns are grouped into column families. All column family members have a common prefix, so,

- for example, the columns **info:format and info:geo** are both members of the info column family, whereas **contents:image** belongs to the contents family

- The column family prefix must be composed of printable characters. The qual- ifying tail, the column family qualifier, can be made of any arbitrary bytes. The column family and the qualifier are always separated by a colon character (:)

Hbase –Implementations

- Just as HDFS and MapReduce are built of clients, slaves, and a coordinating master—**namenode and datanodes in HDFS** and **jobtracker and tasktrackers in MapReduce**

- HBase modeled with an **HBase master node** orchestrating a cluster of **one or more regionserver slaves** (see Figure 13-1).

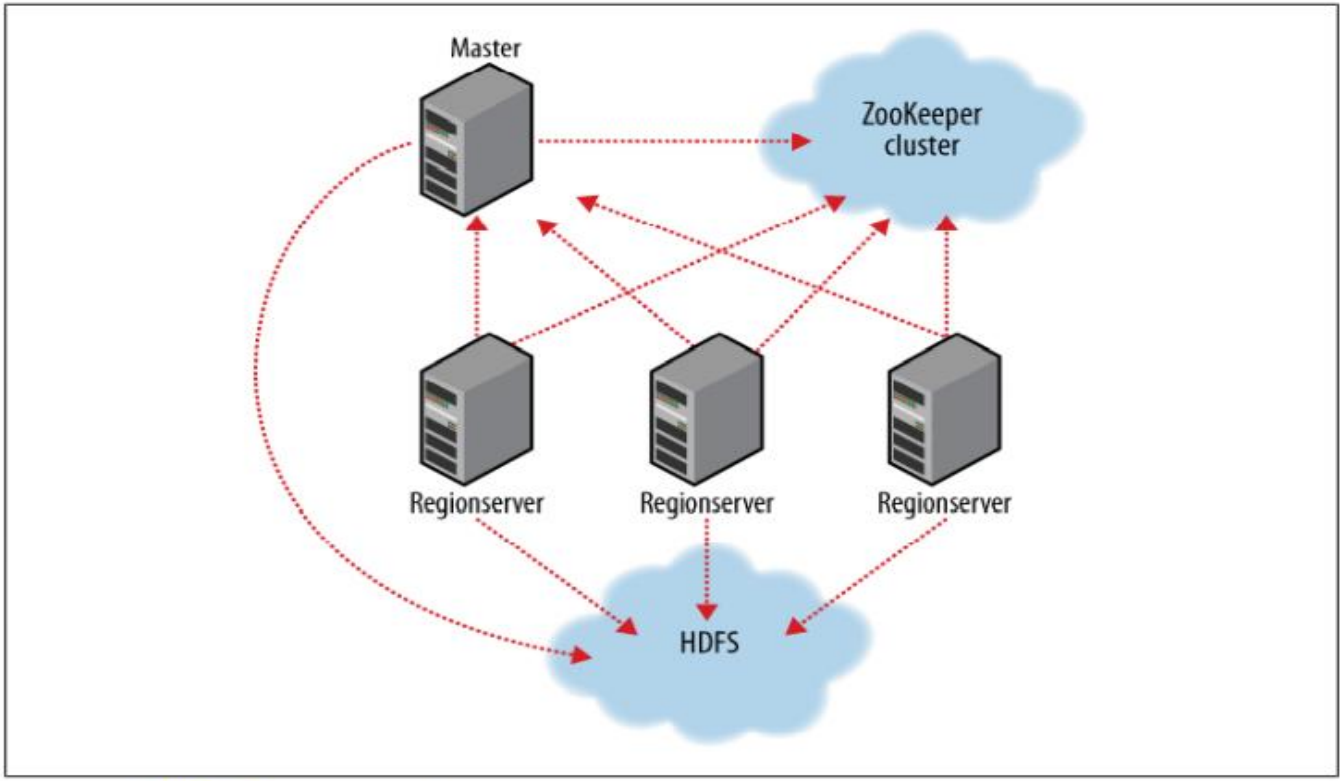


Figure 13-1. HBase cluster members

- The **HBase master** is responsible for bootstrapping a virgin install,
    - **for assigning regions to registered regionservers,** and
    - **for recovering regionserver failures**.
- The master node is lightly loaded. The **regionservers** **carry zero or more regions** and field client read/write requests.
- They also **manage region splits** informing the HBase master about the new daughter regions for it
    - **to manage the offlining of parent region** and
    - **assignment of the replacement daughters**

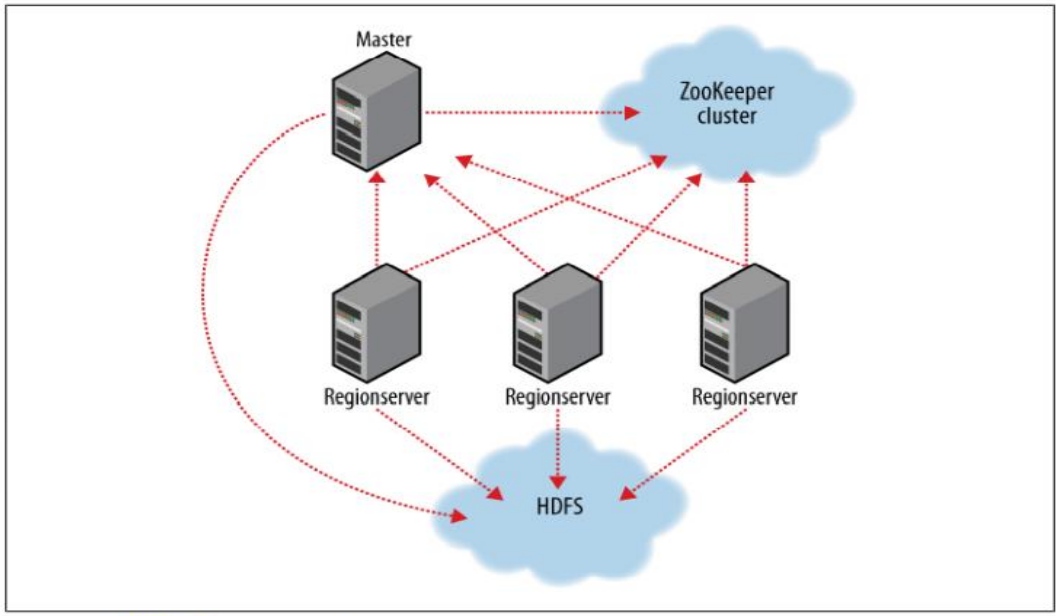


Figure 13-1. HBase cluster members

- **HBase depends on ZooKeeper** and by default it manages a ZooKeeper instance as the authority on cluster state.

    - HBase hosts vitals such as the location of the root catalog table and the address of the current cluster Master.
    - Assignment of regions is mediated via ZooKeeper in case participating servers crash mid-assignment.
    - Hosting the assignment transaction state in ZooKeeper makes it so recovery can pick up on the assignment at where the crashed server left off.

- **HBase persists data via the Hadoop filesystem API**.

- Since **there are multiple implementations of the filesystem interface**—HBase can persist to any of these implementations

    - one for the local filesystem,
    - one for the KFS filesystem,
    - Amazon's S3, and
    - HDFS (the Hadoop Distributed Filesystem)

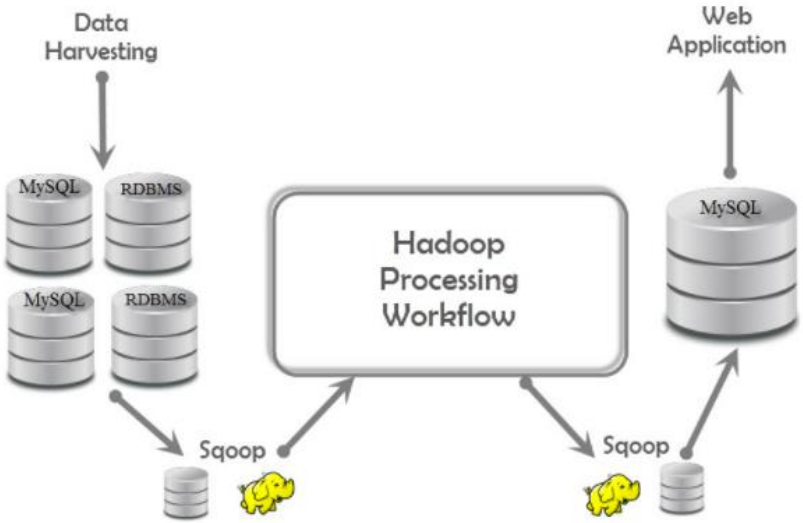
- Most experience though has been had using HDFS, though by default, unless told otherwise, HBase writes to the local filesystem.

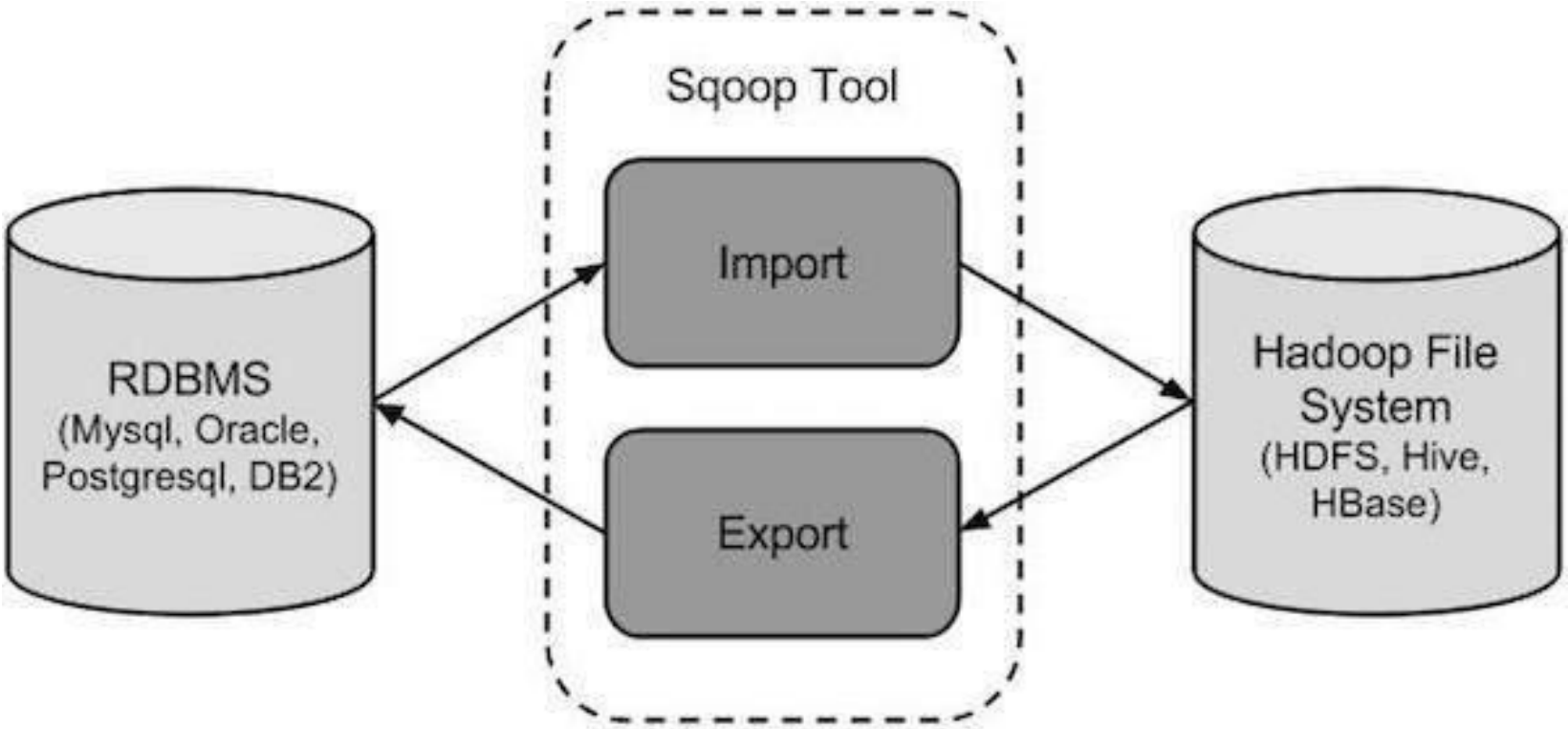
Interacting with Hadoop ecosystem –Sqoop

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.




Sqoop Tool
Import
Export
RDBMS
(Mysql, Oracle,
Postgresql, DB2)
Hadoop File
System
(HDFS, Hive,
HBase)

Apache Sqoop: Introduction

Apache Sqoop is a data ingestion tool designed for efficiently transferring bulk data between Apache Hadoop and structured data-stores such as relational databases, and vice-versa.

Important Features of Apache Sqoop

1. Sqoop uses the YARN framework to import and export data. Parallelism is enhanced by fault tolerance in this way.

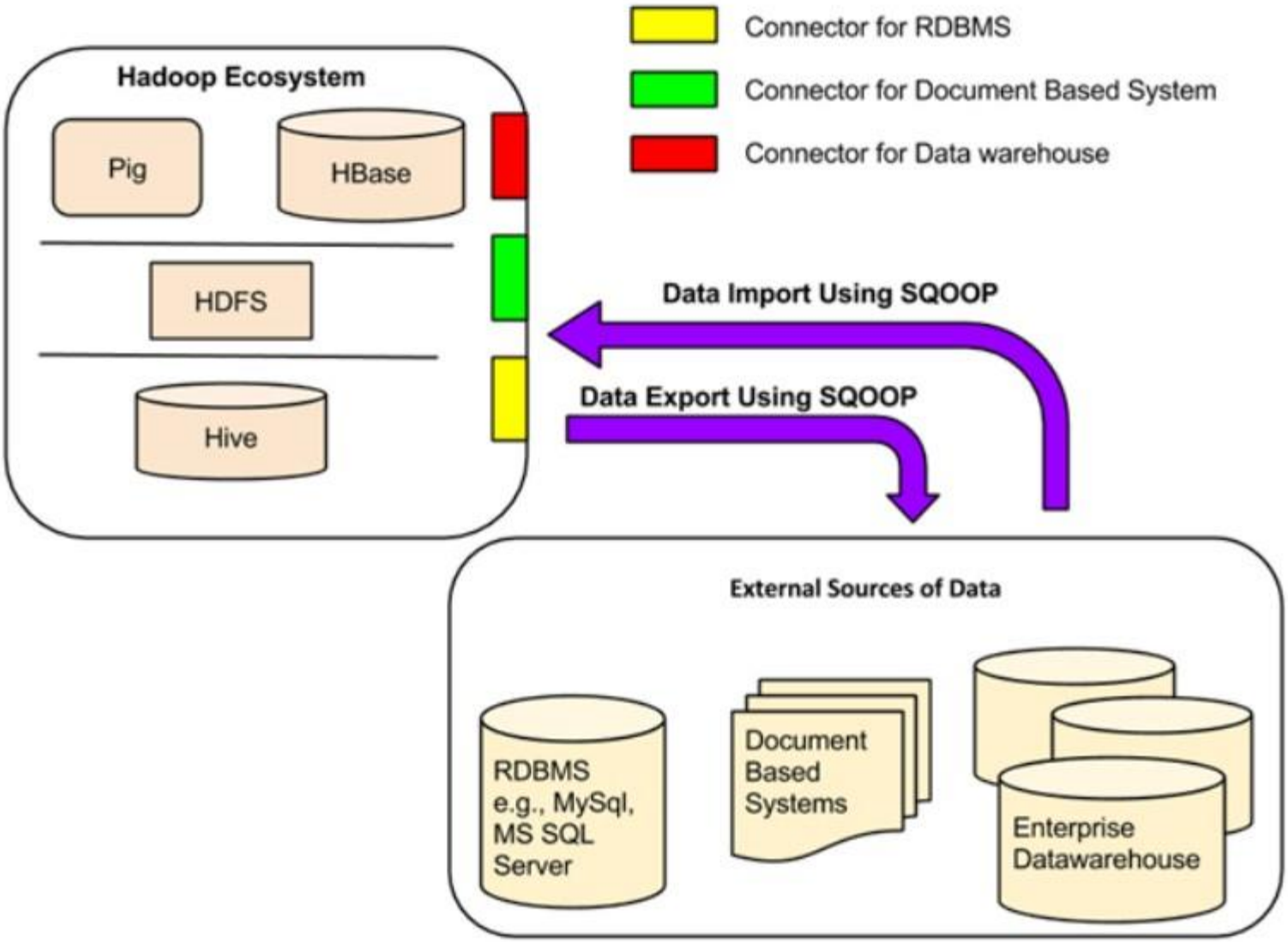
2. We may import the outcomes of a SQL query into HDFS using Sqoop.

3. For several RDBMSs, including the MySQL and Microsoft SQL servers, Sqoop offers connectors.

4. Sqoop supports the Kerberos computer network authentication protocol, allowing nodes to authenticate users while securely communicating across an unsafe network.

5. With a single command, Sqoop can load the full table or specific sections of the table.

Sqoop Architecture



Advantages of using Sqoop

1. It entails data transfer from numerous structured sources, like Oracle, Postgres, etc.

2. Due to the parallel data transport, it is quick and efficient.

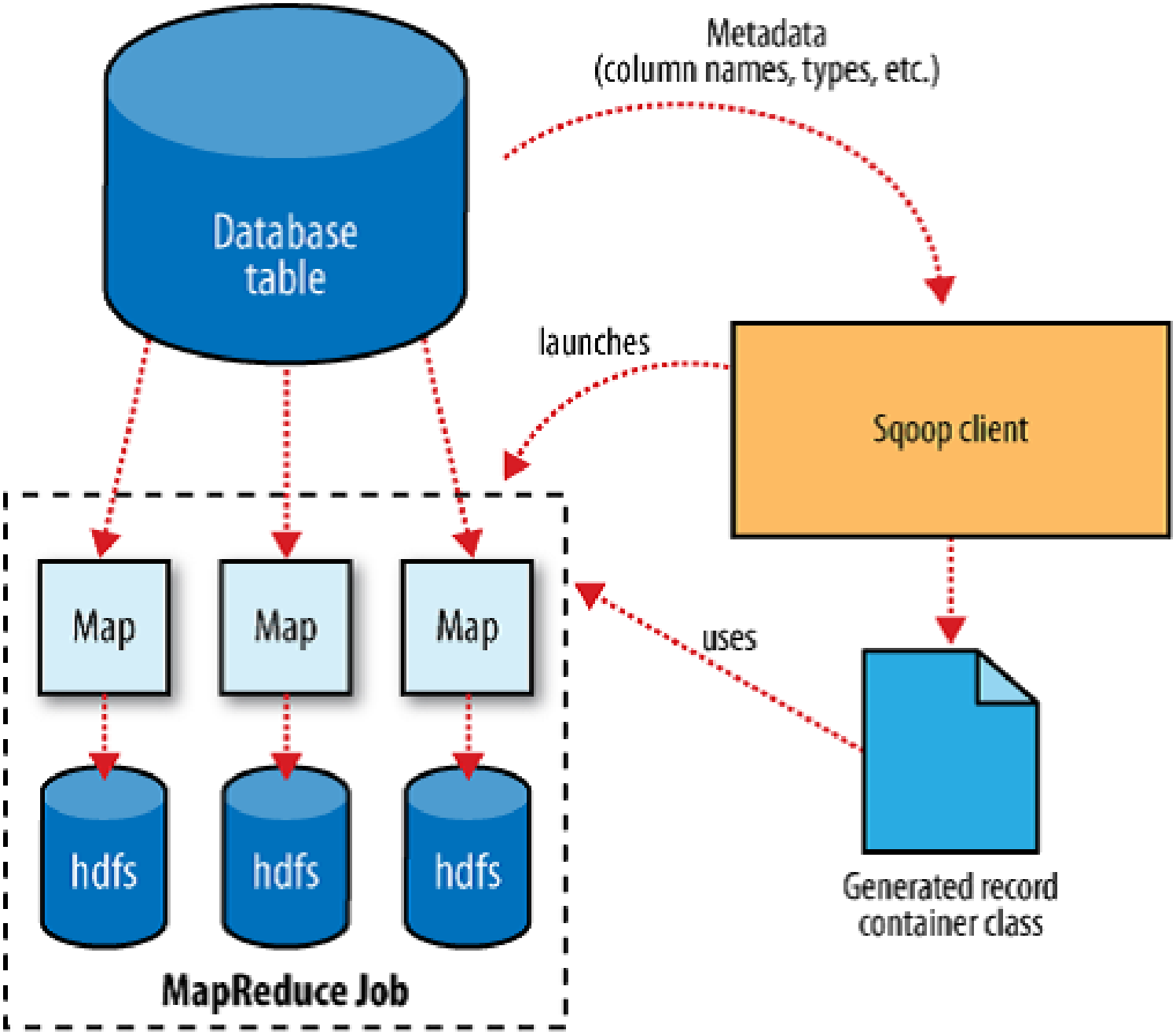
3. Many procedures can be automated, which increases efficiency.

4. Integration with Kerberos security authentication is feasible.

5. Direct data loading is possible from HBase and Hive.

6. It is a powerful tool with a sizable support network.

7. As a result of its ongoing development and contributions, it is frequently updated.


Metadata
(column names, types, etc.)
Database table
launches
Sqoop client
Map
Map
Map
uses
hdfs
hdfs
hdfs
Generated record container class
MapReduce Job

Workflows: coordination of multiple activities

- Coordination based on a state machine model -the Zoo Keeper
- 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:

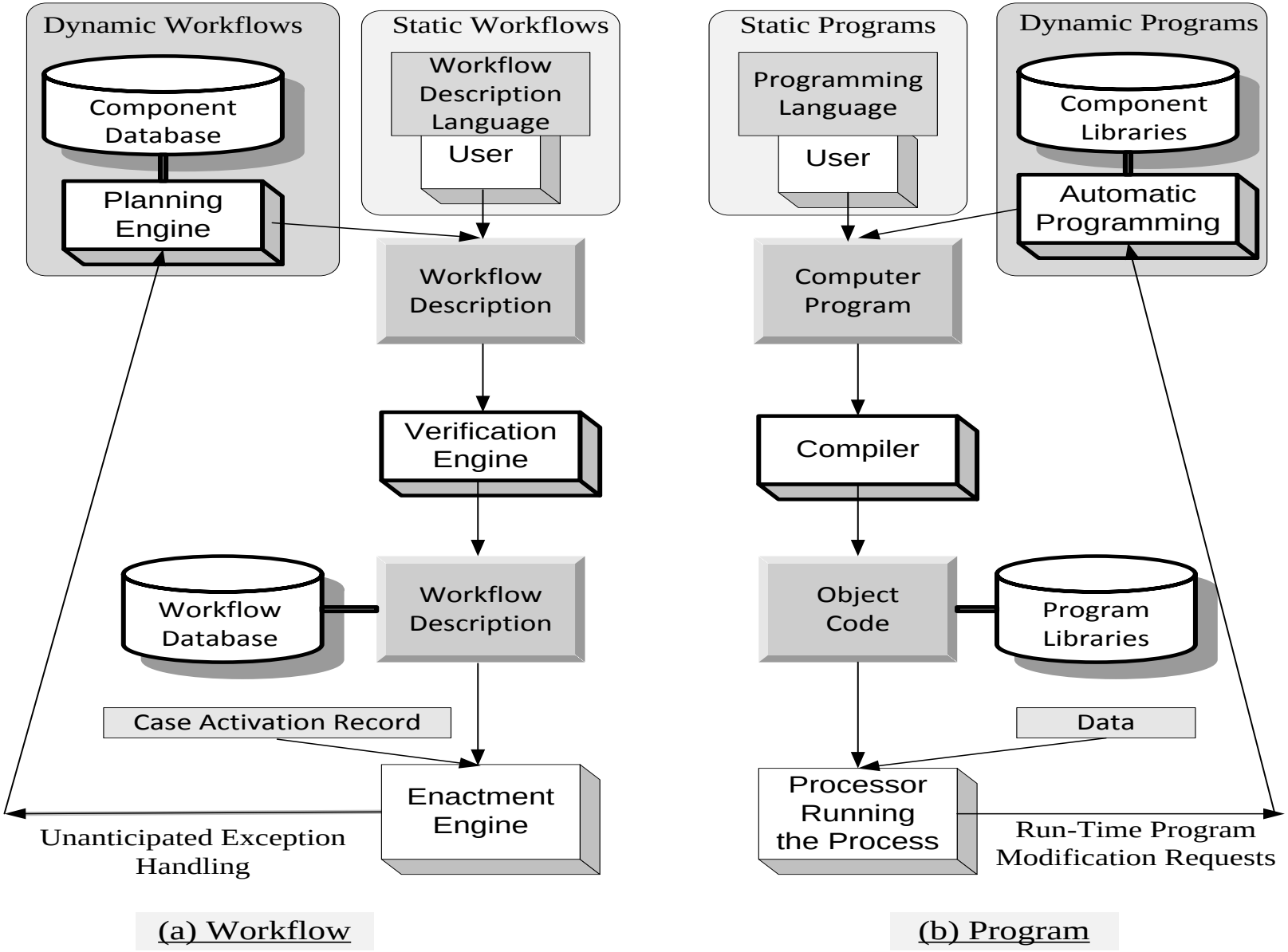
* The successor is enabled after all predecessors end.
* The successors is enabled after k out of n>k predecessors end.
* Iterative: The tasks between the fork and the join are executed repeatedly.

- 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.



(a) Workflow

(b) Program

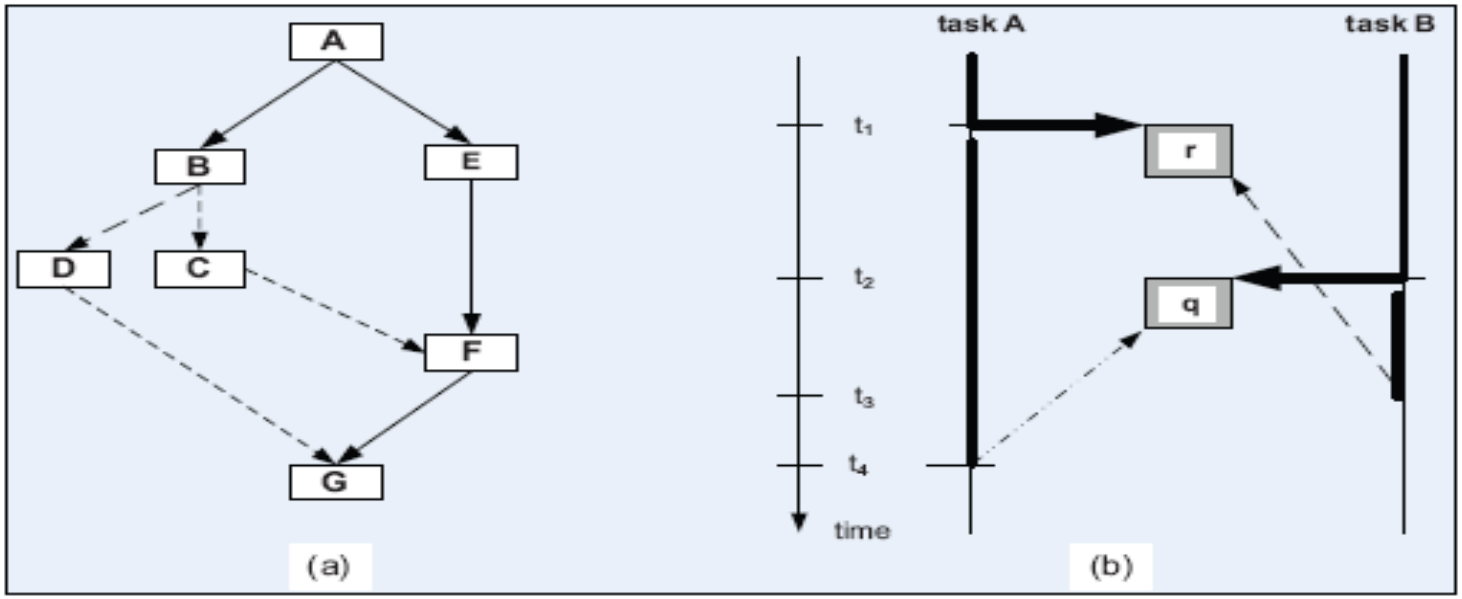


Figure 35: (a) A process description which violates the liveness requirement; if task C is chosen after completion of B, the process will terminate after executing task G; if D is chosen, then F will never be instantiated because it requires the completion of both C and E. The process will never terminate 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:

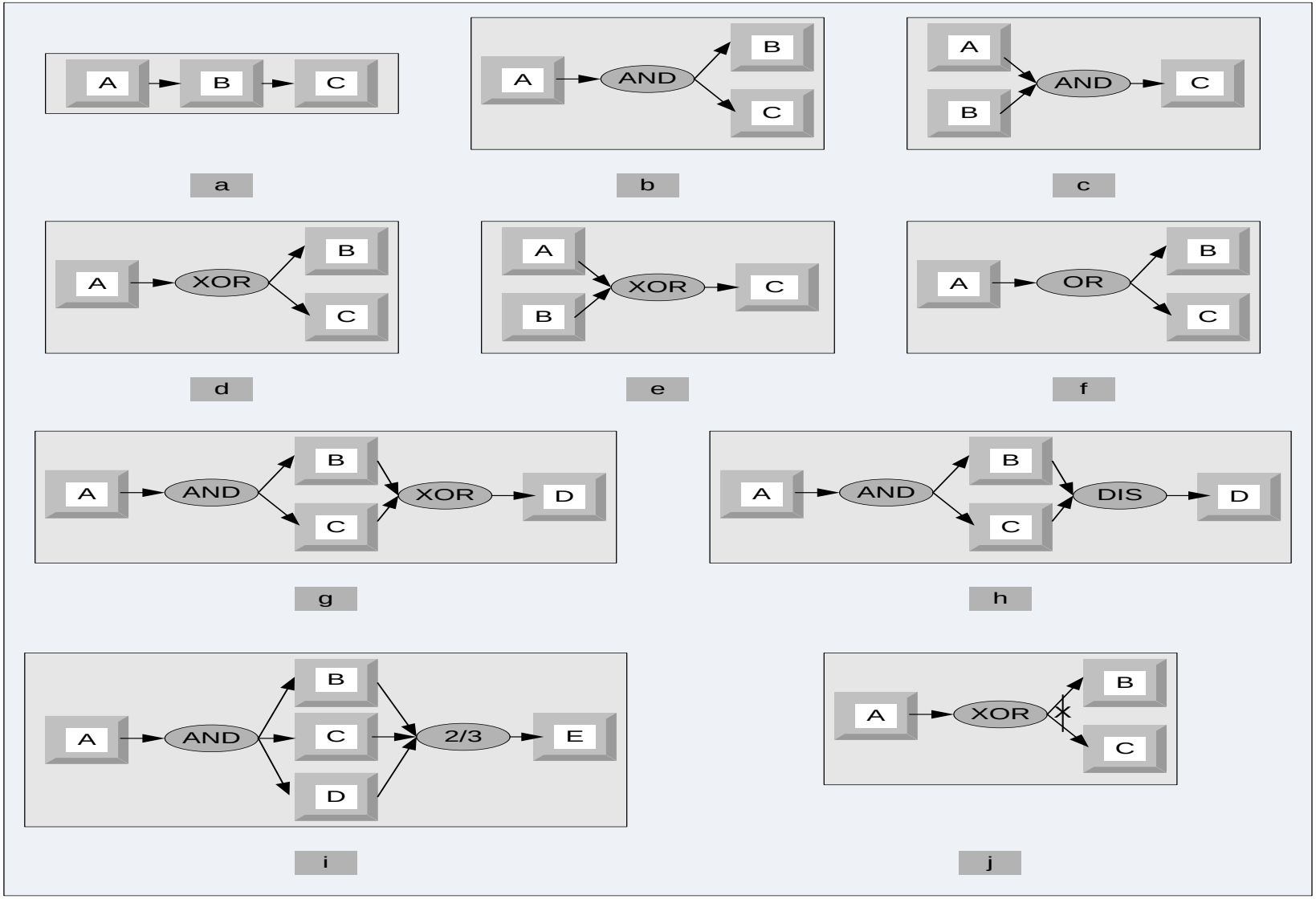
- **Sequence -** several tasks have to be **scheduled one after the completion of the other.**
- **AND split -** both **tasks B and C are activated when task A terminates**.
- **Synchronization -** task **C can only start after tasks A and B terminate**.
- **XOR split -** after **completion of task A, 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
2/3
D
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.**

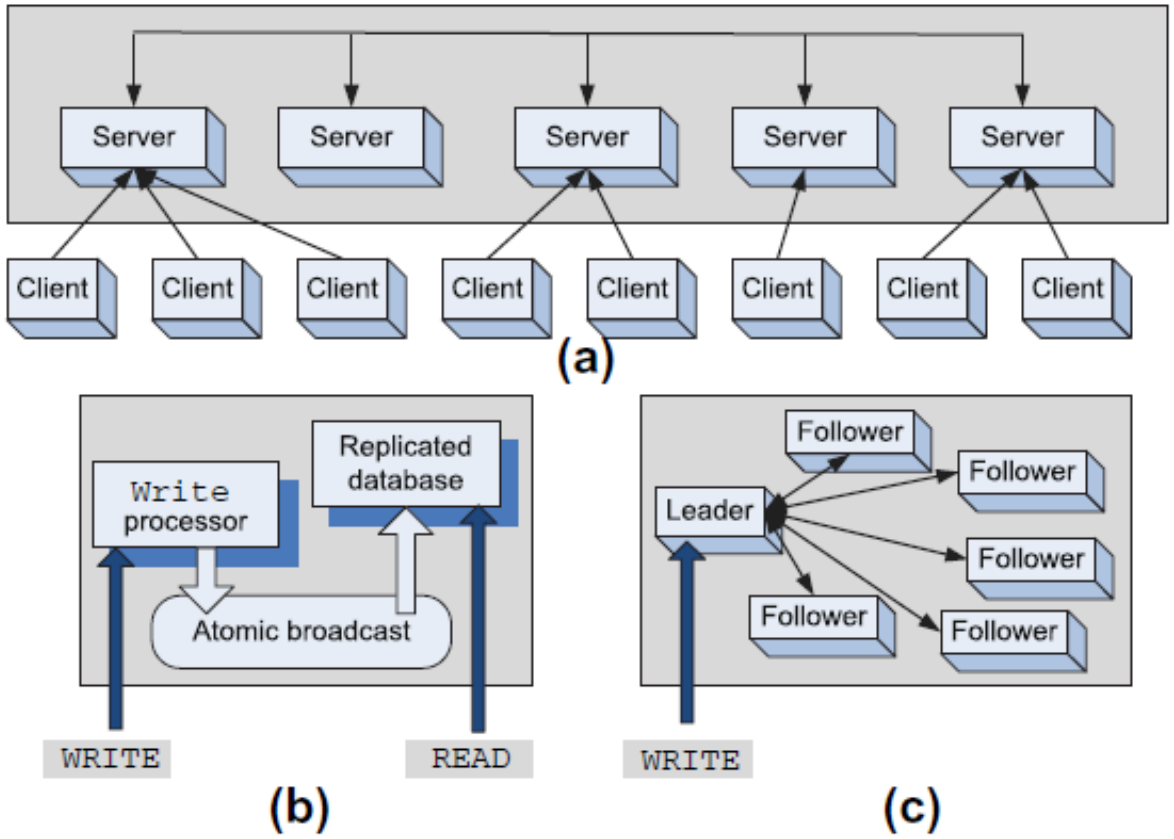


FIGURE 4.4

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).

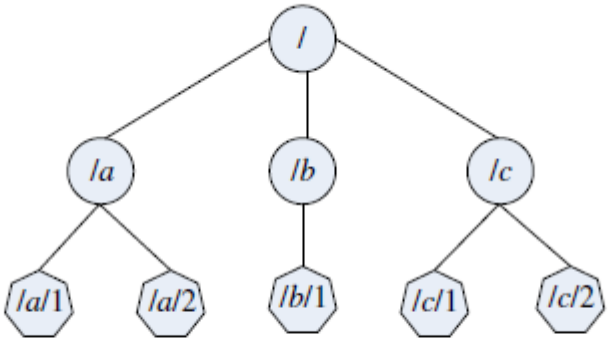


FIGURE 4.5

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**.

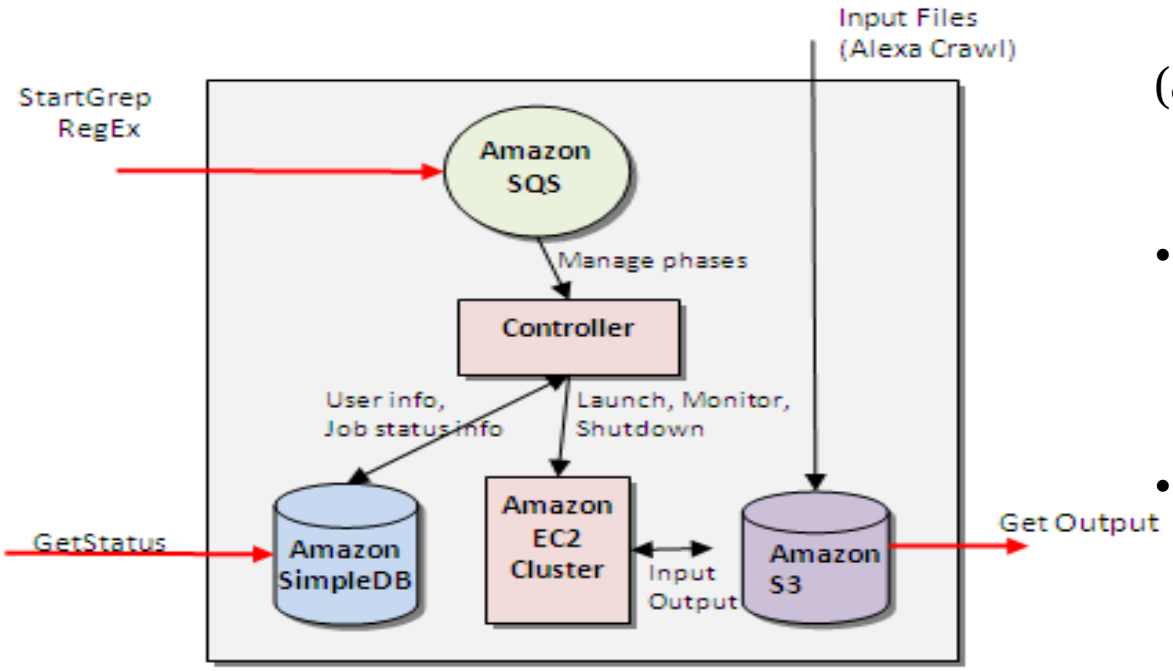
Case study: GrepTheWeb

- 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.

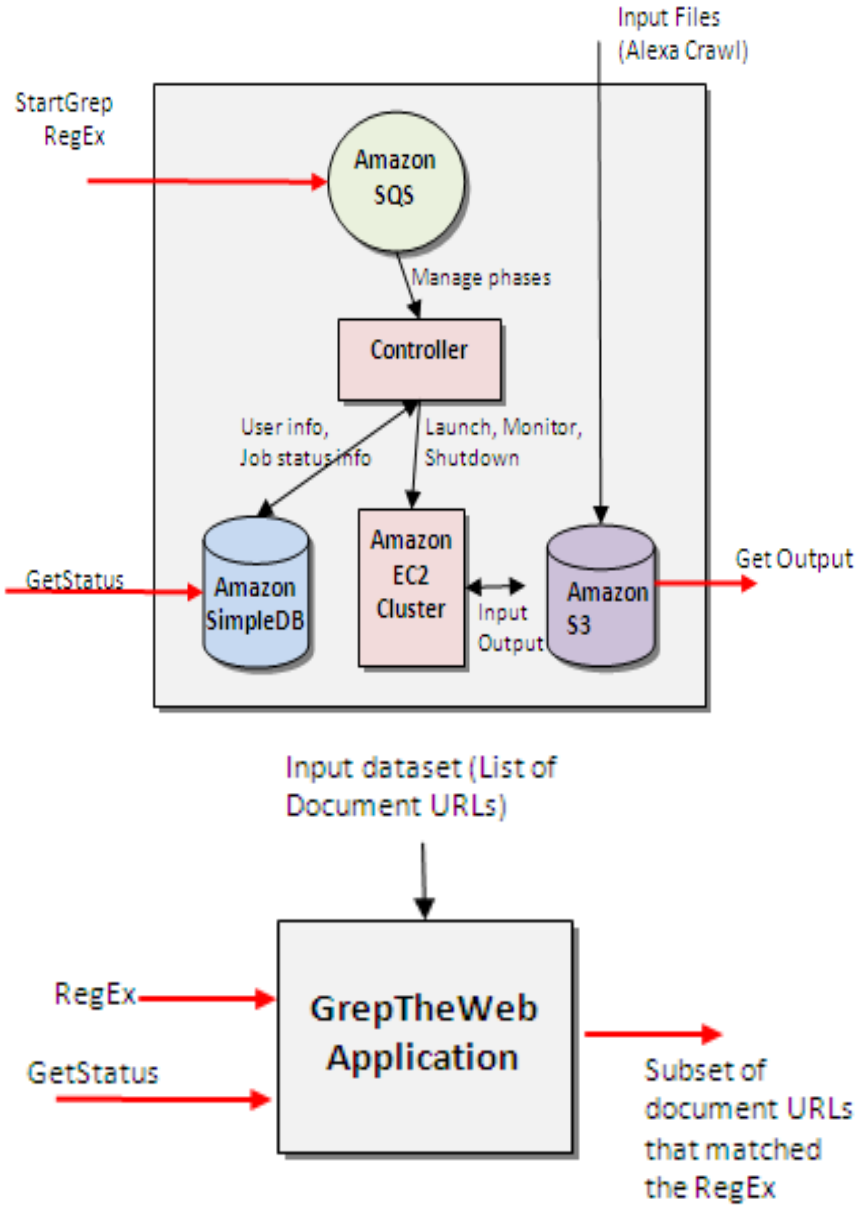


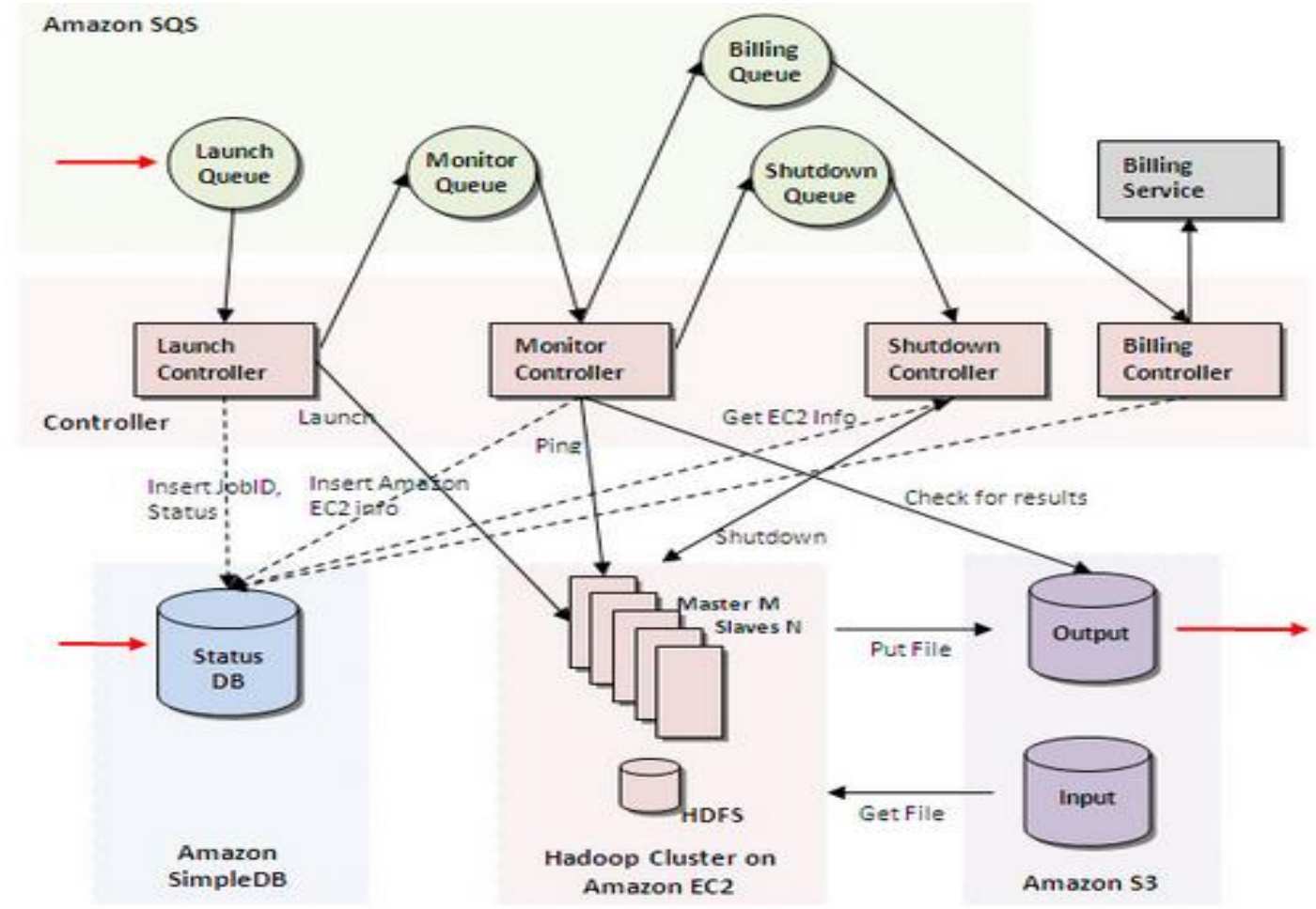
(a) The **simplified workflow** showing the two inputs,

- 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





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