Introduction Hadoop: Big Data, Apache Hadoop Hadoop Eco System, Moving Data in and out of Hadoop, Understanding inputs and outputs of MapReduce, Data Serialization.

October 16, 2023

1 Apache Hadoop and Hadoop Eco System

Hadoop Ecosystem is neither a programming language nor a service, it is a platform or framework which solves big data problems. You can consider it as a suite which encompasses a number of services (ingesting, storing, analyzing and maintaining) inside it.

Below are the Hadoop components, that together form a Hadoop ecosystem.

- HDFS: Hadoop Distributed File System
- YARN : Yet Another Resource Negotiator
- MapReduce : Data processing using programming.
- Spark : In-memory Data Processing PIG, HIVE: Data Processing Services using Query (SQL -like).
- HBase : NoSQL Database,Mahout, Spark
- MLlib : Machine Learning
- Apache Drill : SQL on Hadoop
- Zookeeper : Managing Cluster
- Oozie : Job Scheduling
- Flume : Data Ingesting Services

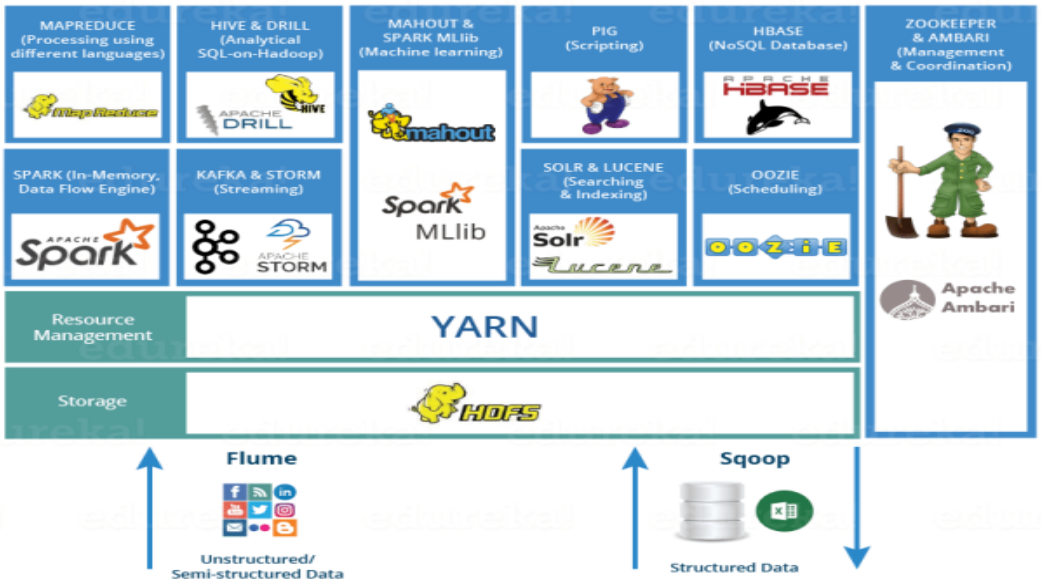

Figure 1: Hadoop ecosystem

1.1 HDFS

- Hadoop Distributed File System is the core component or you can say, the backbone of Hadoop Ecosystem.
- HDFS is the one, which makes it possible to store different types of large data sets(i.e. structured, unstructured and semi structured data).
- HDFS creates a level of abstraction over the resources, from where we can see the whole HDFS as a single unit.
- It helps us in storing our data across various nodes and maintaining the log file about the stored data (metadata).
- HDFS has two core components(NameNode and DataNode)
  - The NameNode is the main node and it does not store the actual data. It contains metadata, just like a log file or you can say as a table of content. Therefore, it requires less storage and high computational resources.
  - all your data is stored on the DataNodes and hence it requires more storage resources. These DataNodes are commodity hardware (like your laptops and desktops) in the distributed environment. That's the reason, why Hadoop solutions are very cost effective.

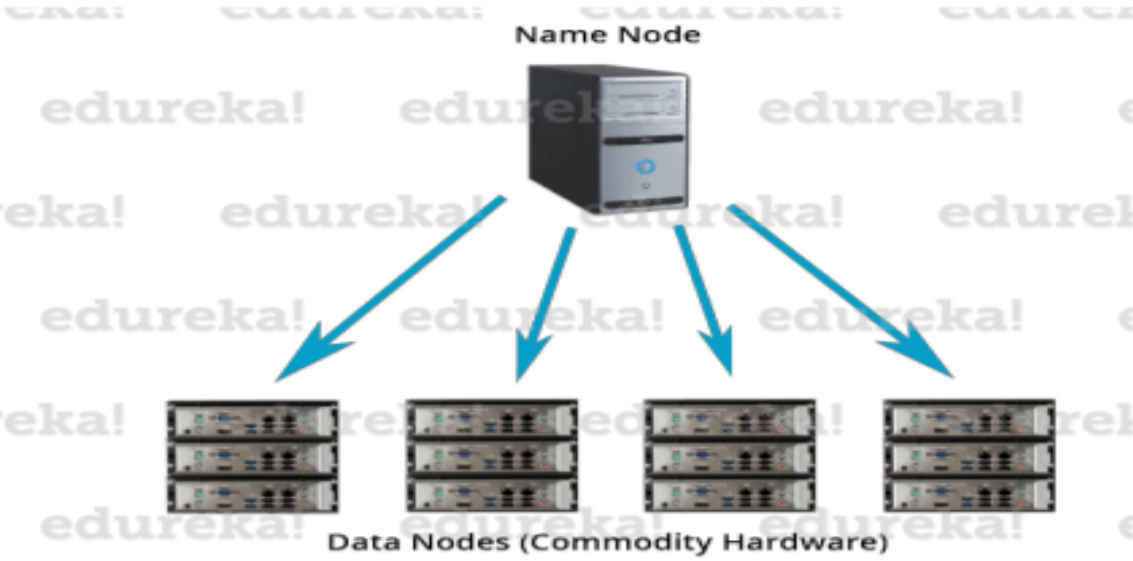

Figure 2: Data Nodes(Commodity Hardware)

1.2 YARN

Consider YARN as the brain of your Hadoop Ecosystem. It performs all your processing activities by allocating resources and scheduling tasks.

- It has two major components,that is Resource Manager and Node Manager.
  1. Resource Manager is again a main node in the processing department.
  2. It receives the processing requests, and then passes the parts of requests to corresponding Node Managers accordingly, where the actual processing takes place.
  3. Node Managers are installed on every Data Node. It is responsible for execution of task on every single Data Node.
- Resource Manager has two components: Schedulers and application manager

1. **Schedulers:** Based on your application resource requirements, Schedulers perform scheduling algorithms and allocates the resources.
2. **Applications Manager:** While Applications Manager accepts the job submission, negotiates to containers (i.e. the Data node environment where process executes) for executing the application specific Application Master and monitoring the progress. Application Masters are the deamons which reside on DataNode and communicates to containers for execution of tasks on each DataNode.

1.3 MAPREDUCE

It is the core component of processing in a Hadoop Ecosystem as it provides the logic of processing. In other words, MapReduce is a software framework which helps in writing applications that processes large data sets using distributed and parallel algorithms inside Hadoop environment.

In a MapReduce program, Map() and Reduce() are two functions.

1. The Map function performs actions like filtering, grouping and sorting.
2. While Reduce function aggregates and summarizes the result produced by map function.
3. The result generated by the Map function is a key value pair (K, V) which acts as the input for Reduce function

1.4 Pig

- PIG has two parts: Pig Latin, the language and the pig runtime, for the execution environment. You can better understand it as Java and JVM.
- It supports pig latin language, which has SQL like command structure.
- 10 line of pig latin = approx. 200 lines of Map-Reduce Java code.
- at the back end of Pig job, a map-reduce job executes.
  - The compiler internally converts pig latin to MapReduce. It produces a sequential set of MapReduce jobs, and that's an abstraction (which works like black box).
  - PIG was initially developed by Yahoo.
  - It gives you a platform for building data flow for ETL (Extract, Transform and Load),processing and analyzing huge data sets.

1.4.1 Differences between Apache MapReduce and PIG

S.No	Apache MapReduce	Apache PIG
1	It is a low-level data processing tool	It is a high-level data flow tool.
2	Here, it is required to develop complex programs using Java or Python.	It is not required to develop complex programs.
3	It's a best practice for enormous data.	It's a best practice for data for future prediction.
4	It is difficult to perform data operations in MapReduce.	It provides built-in operators to perform data operations like union, sorting and ordering.
5	It doesn't allow nested data types.	It provides nested data types like tuple, bag, and map.

1.5 HIVE

- Facebook created HIVE for people who are fluent with SQL. Thus, HIVE makes them feel at home while working in a Hadoop Ecosystem.
- Basically, HIVE is a data warehousing component which performs reading, writing and managing large data sets in a distributed environment using SQL-like interface.

- The query language of Hive is called Hive Query Language(HQL), which is very similar like SQL**(HIVE + SQL = HQL)**.
- It has 2 basic components: Hive Command Line and JDBC/ODBC driver.
- The Hive Command line interface is used to execute HQL commands.
- Java Database Connectivity (JDBC) and Object Database Connectivity(ODBC) is used to establish connection from data storage.
- Hive is highly scalable. As, it can serve both the purposes, i.e. large data set processing (i.e. Batch query processing) and real time processing (i.e. Interactive query processing).
- It supports all primitive data types of SQL.
- predefined functions, or write tailored user defined functions (UDF) can be used to accomplish your specific needs.

1.6 Mahout

Mahout which is renowned for machine learning. Mahout provides an environment for creating machine learning applications which are scalable. Machine learning algorithms allow us to build self-learning machines that evolve by itself without being explicitly programmed. Based on user behaviour, data patterns and past experiences it makes important future decisions.

What Mahout does

It performs collaborative filtering, clustering and classification. Some people also consider frequent item set missing as Mahout's function.

1. **Collaborative filtering:** Mahout mines user behaviors, their patterns and their characteristics and based on that it predicts and make recommendations to the users.The typical use case is E-commerce website.
2. **Clustering:** It organizes a similar group of data together like articles can contain blogs, news, research papers etc.
3. **Classification:** It means classifying and categorizing data into various sub departments like articles can be categorized into blogs, news, essay, research papers and other categories.
4. **Frequent item set missing:** Here Mahout checks, which objects are likely to be appearing together and make suggestions, if they are missing. For example, cell phone and cover are brought together in general. So, if you search for a cell phone, it will also recommend you the cover and cases.

Mahout provides a command line to invoke various algorithms. It has a predefined set of library which already contains different inbuilt algorithms for different use cases

1.7 Apache Spark

- Apache Spark is a framework for real time data analytics in a distributed computing environment.
- The Spark is written in Scala and was originally developed at the University of California, Berkeley.
- It executes inmemory computations to increase speed of data processing over MapReduce.
- It is 100x faster than Hadoop for large scale data processing by exploiting inmemory computations and other optimizations. Therefore, it requires high processing power than MapReduce.
- Spark comes packed with high level libraries, including support for R, SQL,Python, Scala, Java etc.
- Apache Spark best fits for real time processing, whereas Hadoop was designed to store unstructured data and execute batch processing over it. When we combine, Apache Spark ability, i.e. high processing speed, advance analytics and multiple integration support with Hadoop's low cost operation on commodity hardware, it gives the best results.

Figure 3: Apache Spark

1.8 Hbase

- HBase is an open source, non-relational distributed database. In other words, it is a NoSQL database.
- It supports all types of data and that is why, it's capable of handling anything and everything inside a Hadoop ecosystem.
- It is modelled after Google's BigTable, which is a distributed storage system designed to cope up with large data sets.
- The HBase was designed to run on top of HDFS and provides BigTable like capabilities.
- It gives us a fault tolerant way of storing sparse data, which is common in most Big Data use cases.
- The HBase is written in Java, whereas HBase applications can be written in REST,Avro and Thrift APIs.

1.9 Apache Drill

Apache Drill is used to drill into any kind of data. It's an open source application which works with distributed environment to analyze large data sets.

The main aim behind Apache Drill is to provide scalability so that we can process petabytes and exabytes of data efficiently (or you can say in minutes).

- It is a replica of Google Dremel.
- t supports different kinds NoSQL databases and file systems, which is a powerful feature of Drill. For example: Azure Blob Storage, Google Cloud Storage, HBase,MongoDB, MapR-DB HDFS, MapR-FS, Amazon S3, Swift, NAS and local files.
- The main power of Apache Drill lies in combining a variety of data stores just by using a single query.
- Apache Drill basically follows the ANSI SQL.
- It has a powerful scalability factor in supporting millions of users and serve their query requests over large scale data.

1.10 Apache Zookeeper

- Apache Zookeeper is the coordinator of any Hadoop job which includes a combination of various services in a Hadoop Ecosystem.
- Apache Zookeeper coordinates with various services in a distributed environment.

Before Zookeeper, it was very difficult and time consuming to coordinate between different services in Hadoop Ecosystem. The services earlier had many problems with interactions like common configuration while synchronizing data. Even if the services are configured, changes in the configurations of the services make it complex and difficult to handle. The grouping and naming was also a time-consuming factor.

Due to the above problems, Zookeeper was introduced. It saves a lot of time by performing synchronization, configuration maintenance, grouping and naming.

1.11 Apache Oozie

Consider Apache Oozie as a clock and alarm service inside Hadoop Ecosystem. For Apache jobs, Oozie has been just like a scheduler. It schedules Hadoop jobs and binds them together as one logical work.

There are two kinds of Oozie jobs:

1. **Oozie workflow:** These are sequential set of actions to be executed. You can assume it as a relay race. Where each athlete waits for the last one to complete his part.
2. **Oozie Coordinator:**These are the Oozie jobs which are triggered when the data is made available to it. Think of this as the response-stimuli system in our body. In the same manner as we respond to an external stimulus, an Oozie coordinator responds to the availability of data and it rests otherwise.

1.12 Apache Flume

Ingesting data is an important part of our Hadoop Ecosystem.

- The Flume is a service which helps in ingesting unstructured and semi structured data into HDFS.
- It gives us a solution which is reliable and distributed and helps us in collecting, aggregating and moving large amount of data sets.
- It helps us to ingest online streaming data from various sources like network traffic, social media, email messages, log files etc. in HDFS.

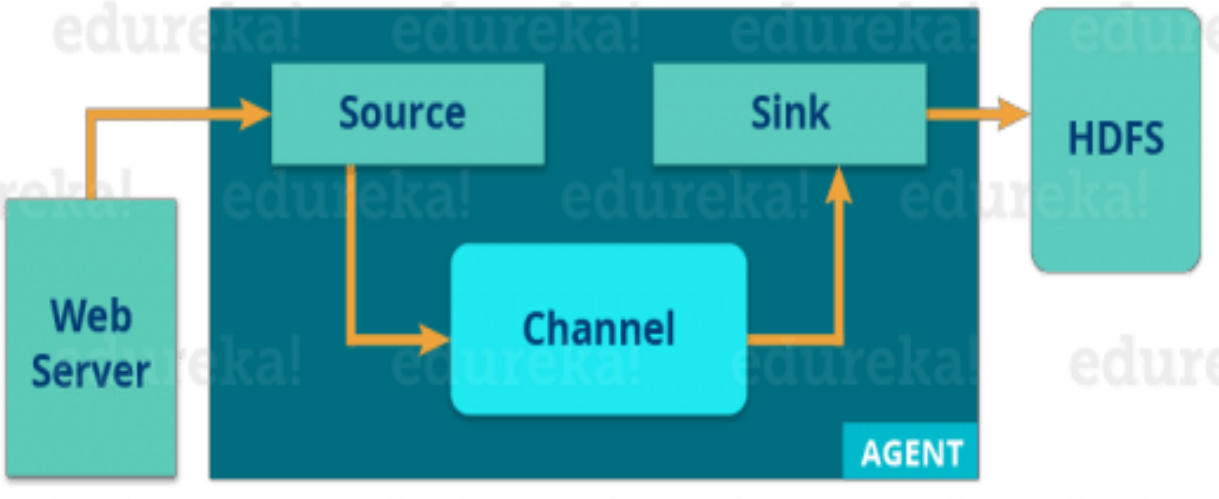

Figure 4: Flume architecture

There is a Flume agent which ingests the streaming data from various data sources to HDFS. From the diagram,the web server indicates the data source. Twitter is among one of the famous sources for streaming data.

The flume agent has 3 components: source, sink and channel.

1. **Source:** it accepts the data from the incoming streamline and stores the data in the channel.
2. **Channel:** it acts as the local storage or the primary storage. A Channel is a temporary storage between the source of data and persistent data in the HDFS.
3. **Sink:** Then, our last component i.e. Sink, collects the data from the channel and commits or writes the data in the HDFS permanently.

2 Moving Data in and out of Hadoop

Data exists in various forms and locations throughout your environments complicates the process of ingress and egress.

1. How do you bring in data that's sitting in an OLTP (online transaction processing) database?
2. How do you ingress log data that's being produced by tens of thousands of production servers?
3. how do you work with binary data sitting behind a firewall?
4. how do you automate your data ingress and egress process so that your data is moved at regular intervals?

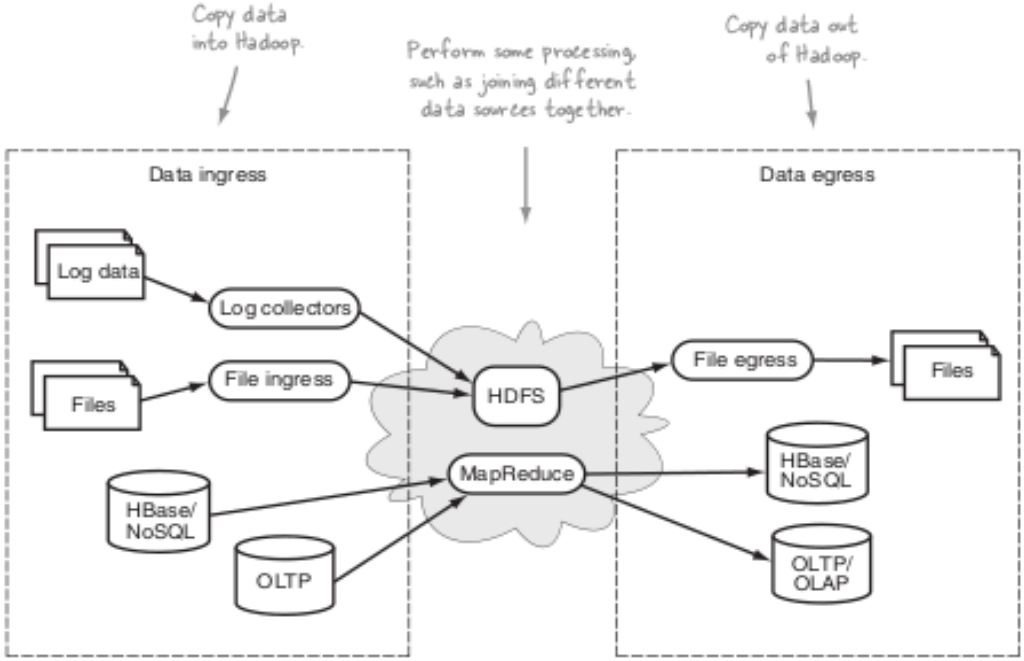

Figure 5: Hadoop data ingress and egress transports data to and from an external system to an internal one.

2.1 Key elements of ingress and egress

Moving large quantities of data in and out of Hadoop has logistical challenges that include consistency guarantees and resource impacts on data sources and destinations. Before we dive into the techniques, however, we need to discuss the design elements to be aware of when working with data ingress and egress.

2.1.1 Idempotent

An idempotent operation produces the same result no matter how many times it's executed. In a relational database the inserts typically aren't idempotent, because executing them multiple times doesn't produce the same resulting database state. Alternatively, updates often are idempotent, because they'll produce the same end result.

Any time data is being written idempotence should be a consideration, and data ingress and egress in Hadoop is no different.

- How well do distributed log collection frameworks deal with data retransmissions?
- How do you ensure idempotent behavior in a MapReduce job where multiple tasks are inserting into a database in parallel?

2.1.2 Aggregation

The data aggregation process combines multiple data elements. In the context of data ingress this can be useful because moving large quantities of small files into HDFS potentially translates into NameNode memory woes, as well as slow MapReduce execution times. Having the ability to aggregate files or data together mitigates this problem.

2.1.3 Data Format Transformation

The data format transformation process converts one data format into another. Often your source data isn't in a format that's ideal for processing in tools such as MapReduce. If your source data is multiline XML or JSON form, for example, you may want to consider a preprocessing step. This would convert the data into a form that can be split, such as a JSON or an XML element per line, or convert it into a format such as Avro.

2.1.4 Recoverability

Recoverability allows an ingress or egress tool to retry in the event of a failed operation. Because it's unlikely that any data source, sink, or Hadoop itself can be 100 percent available, it's important that an ingress or egress action be retried in the event of failure.

2.1.5 Correctness

In the context of data transportation, checking for correctness is how you verify that no data corruption occurred as the data was in transit. When you work with heterogeneous systems such as Hadoop data ingress and egress tools, the fact that data is being transported across different hosts, networks, and protocols only increases the potential for problems during data transfer. Common methods for checking correctness of raw data such as storage devices include Cyclic Redundancy Checks (CRC), what HDFS uses internally to maintain block-level integrity.

2.1.6 Resource Consumption and Performance

Resource consumption and performance are measures of system resource utilization and system efficiency, respectively. Ingress and egress tools don't typically incur significant load (resource consumption) on a system, unless you have appreciable data volumes. For performance, the questions to ask include whether the tool performs ingress and egress activities in parallel, and if so, what mechanisms it provides to tune the amount of parallelism. For example, if your data source is a production database, don't use a large number of concurrent map tasks to import data.

2.1.7 Monitoring

Monitoring ensures that functions are performing as expected in automated systems. For data ingress and egress, monitoring breaks down into two elements: ensuring that the process(es) involved in ingress and egress are alive, and validating that source and destination data are being produced as expected.

On to the techniques. Let's start with how you can leverage Hadoop's builtin ingress and egress mechanisms.

2.2 Moving Data into Hadoop

there are two primary methods that can be used for moving data into Hadoop:

- writing external data at the HDFS level (a data push).
- reading external data at the MapReduce level (more like a pull).

Reading data in MapReduce has advantages in the ease with which the operation can be parallelized and fault tolerant. Not all data is accessible from MapReduce, however, such as in the case of log files, which is where other systems need to be relied upon for transportation, including HDFS for the final data hop.

Hadoop provides a set of command line utilities that work similarly to the Linux file commands, and serve as your primary interface with HDFS. We"re going to have a look into HDFS by interacting with it from https://www.overleaf.com/project/ command line. We will take a look at the most common file management tasks in Hadoop, which include

1. Adding files and directories to HDFS.
2. Retrieving files from HDFS to local filesystem.
3. Deleting files from HDFS.

Hadoop file commands take the following form:**$hadoop fs -cmd**

Where **cmd** is the specific file command and **args** is a variable number of arguments. The command **cmd** is usually named after the corresponding Unix equivalent. For example, the command for listing files is ls as in Unix

2.3 Listing Files in HDFS

After loading the information in the server, we can find the list of files in a directory, status of a file, using "ls". Given below is the syntax of ls that you can pass to a directory or a filename as an argument.

$hadoop fs -ls $<$ **args** $>$

for example **$hadoop fs –ls** lists all directories and files in current directory.

2.4 Adding Files and Directories to HDFS

Before you can run Hadoop programs on data stored in HDFS, you"ll need to put the data into HDFS first. Let"s create a directory and put a file in it. HDFS has a default working directory of **/user/$USER**, where **$USER** is your login user name. This directory is not automatically created for you, though, so let us create it with the **mkdir** command. For the purpose of illustration, we use chuck. You should substitute your user name in the example commands.

$ hadoop fs - mkdir path_hadoop/directory_name

eg : hadoop fs –mkdir /lab

Hadoop's **mkdir** command automatically creates parent directories if they do not already exist. Now that we have a working directory, we can put a file into it. Create some text file on your local filesystem called **example.txt**.

The Hadoop command **put** is used to copy files from the local system into HDFS.

$hadoop fs –put path/filename.txt

for example **$hadoop fs -put example.txt.** [here period (.) as the last argument in the command above means that we are putting the file into the default working directory] is equivalent to:**$hadoop fs -put example.txt/lab/**

2.5 Retrieving Files from HDFS

The Hadoop command get copies files from HDFS back to the local filesystem. To retrieve example.txt, we can run the following command:

$ hadoop fs -get example.txt

Another way to access the data is to display it. The Hadoop cat command allows us to do that:

$ hadoop fs -cat example.txt

2.6 Deleting Files from HDFS

The Hadoop command for removing files is **rm**.The **rm** command can also be used to delete empty directories.

$ hadoop fs -rm example.txt

- **ls <path>:** Lists the contents of the directory specified by path, showing the names, permissions, owner, size and modification date for each entry.
- **lsr <path>:** Behaves like -ls, but recursively displays entries in all subdirectories of path.
- **du <path>:** Shows disk usage, in bytes, for all the files which match path, filenames are reported with the full HDFS protocol prefix
- **dus <path>:** Like -du, but prints a summary of disk usage of all files/directories in the path.
- **mv <src> <dest>:** Moves the file or directory indicated by src to dest, within HDFS.
- **cp <src> <dest>:** Copies the file or directory identified by src to dest, within HDFS.
- **rm <path>:** Removes the file or empty directory identified by path.
- **rmr <path>:** Removes the file or directory identified by path. Recursively deletes any child entries (i.e., files or sub directories of path).
- **put <localSrc> <dest>:** Copies the file or directory from the local file system identified by localSrc to dest within the DFS.
- **copyFromLocal <localSrc> <dest>:** Identical to -put
- **moveFromLocal <localSrc> <dest>:** Copies the file or directory from the local file system identified by localSrc to dest within HDFS, and then deletes the local copy on success.
- **get [-crc] <src> <localDest>:** Copies the file or directory in HDFS identified by src to the local file system path identified by localDest.
- **getmerge <src> <localDest>:** Retrieves all files that match the path src in HDFS, and copies them to a single, merged file in the local file system identified by localDest.
- **cat <filen-ame>:** Displays the contents of filename on stdout.
- **copyToLocal <src> <localDest>:** Identical to -get
- **moveToLocal <src> <localDest>:** Works like -get, but deletes the HDFS copy on success.
- **mkdir <path>:** Creates a directory named path in HDFS, Creates any parent directories in path that are missing (e.g., mkdir -p inLinux).
- **setrep [-R] [-w] rep <path>:** Sets the target replication factor for files identified by path to rep. (The actual replication factor will move toward the target over time)
- **touchz <path>:** Creates a file at path containing the current time as a timestamp. Fails if a file already exists at path, unless the file is already size 0.
- **test -[ezd] <path>:** Returns 1 if path exists; has zero length; or is a directory or 0 otherwise.

- **stat [format] <path>:** Prints information about path. Format is a string which accepts file sizein blocks (%b),filename (%n), block size (%o), replication (%r), andmodification date (%y, %Y).
- **tail [-f] <file2name>:** Shows the last 1KB of file on stdout.
- **chmod [-R] mode,mode,... <path>:** Changes the file permissions associated with one or more objects identified by path, Performs changes recursively with R. mode is a 3-digit octal mode, or augo+/- rwxX. Assumes if no scope is specified and does not apply an unmask.
- **chown [-R] [owner][:[group]] <path>...** Sets the owning user and/or group for files or directories identified by path
- **chgrp [-R] group <path>...** Sets the owning group for files or directories identified by path
- **help <cmd-name>:** Returns usage information for one of the commands listed above. You must omit the leading '-' character in cmd.

3 Understanding Inputs and Outputs of MapReduce

MapReduce is a programming model and processing framework used in Hadoop for processing and generating large datasets in a distributed computing environment. In MapReduce, the input and output data are typically stored in the Hadoop Distributed File System (HDFS), and the processing is divided into two main phases: the Map phase and the Reduce phase.

Example: Word Count using MapReduce

Suppose you have a large text document, and you want to count the frequency of each word in that document using MapReduce. Here's how you would do it:

Input Data: The input data is a large text document stored in HDFS, such as:

Hello, world This is a simple example of MapReduce. Hello, Hadoop

The input data is divided into smaller chunks called "splits" by Hadoop, and each split is processed by a separate Mapper task.

3.1 Map Phase

The Map phase takes each split of the input data and processes it line by line, emitting key-value pairs where the key is a word and the value is the count of that word in the line. For example, for the input line "Hello, world", the Mapper emits the following key-value pairs:

Key: "Hello", Value: 1
Key: "world", Value: 1

3.2 Shuffle and Sort:

Hadoop automatically groups and sorts the key-value pairs generated by the Mapper tasks based on the keys. This step ensures that all occurrences of the same word are grouped together and ready for the Reduce phase.

3.3 Reduce Phase:

The Reduce phase takes the sorted key-value pairs and performs aggregation based on the keys. For each unique word, a Reducer task sums up the counts from all the Mapper outputs with the same key.

- During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.

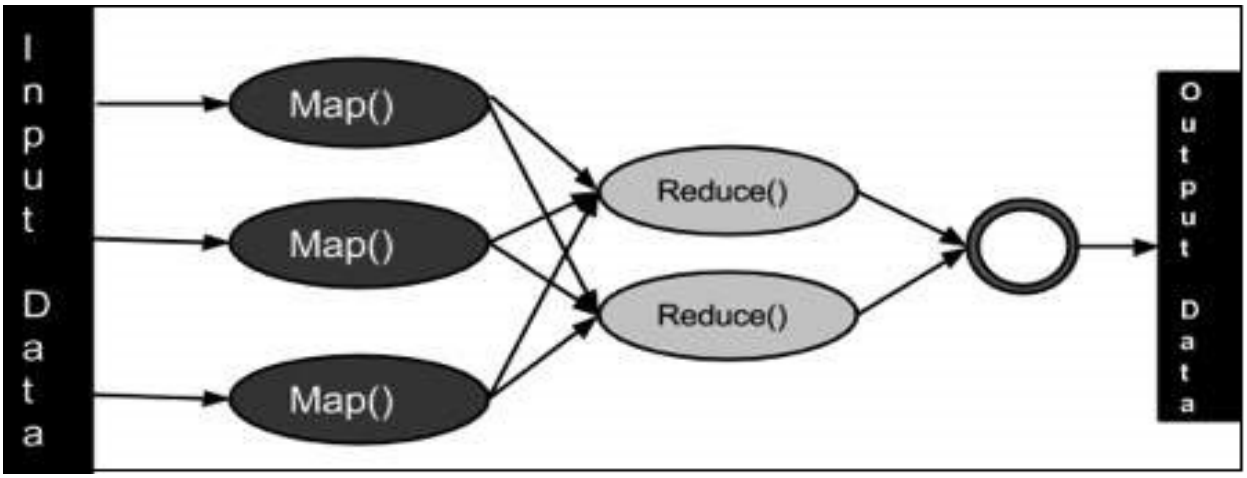

Figure 6: Inputs and Outputs (Java Perspective)

4 Understanding a simple hadoop program

4.1 Dataset(Weather Dataset)

Weather sensors collecting data every hour at many locations across the globe gather a large volume of log data, which is a good candidate for analysis with MapReduce, since it is semi structured and record- oriented.

4.1.1 Data Format

- The data we will use is from the National Climatic Data Center (NCDC, http://www .ncdc.noaa.gov/).
- The data is stored using a line-oriented ASCII format, in which each line is a record.

4.2 Driver code

- A Job object forms the specification of the job. It gives you control over how the job is run.
- When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster).
- Rather than explicitly specify the name of the JAR file, we can pass a class in the Job's **setJarByClass()** method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.
- Having constructed a Job object, we specify the input and output paths.
- An input path is specified by calling the static **addInputPath()** method on **FileInputFormat**, and it can be a single file, a directory(in which case, the input forms all the files in that directory), or a file pattern.
- The output path (of which there is only one) is specified by the static **setOutputPath()** method on **FileOutputFormat**. It specifies a directory where the output files from the reducer functions are written. Next, we specify the map and reduce types to use via the **setMapperClass()** and **setReducerClass()** methods.
- The **setOutputKeyClass()** and **setOutputValueClass()** methods control the output types for the map and the reduce functions, which are often the same, as they are in our case.
- If they are different, then the map output types can be set using the methods **setMapOutputKeyClass()** and **setMapOutputValueClass()**.
- The input types are controlled via the input format, which we have not explicitly set since we are using the default TextInputFormat.
- After setting the classes that define the map and reduce functions, we are ready to run the job. The **waitFor-**

Completion() method on Job submits the job and waits for it to finish.

- The return value of the waitForCompletion() method is a boolean indicating success (true) or failure(false), which we translate into the program's exit code of 0 or 1.

The driver code for weather program is specified below.

```
import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature

{

public static void main(String[] args) throws Exception

{

if (args.length != 2)

System.err.println(&quot;Usage: MaxTemperature &lt;input path&gt; &lt;output path&gt;&quot;);

System.exit(-1);

}

Job job = new Job();

job.setJarByClass(MaxTemperature.class);

job.setJobName(&quot;Max temperature&quot;);

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.setMapperClass(MaxTemperatureMapper.cla ss);

job.setReducerClass(MaxTemperatureReducer.cl ass); job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

System.exit(job.waitForCompletion(true)  0 : 1);

}
```

}

4.3 Mapper code

The Mapper class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function.

For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature (an integer). The following example shows the implementation of our map method.

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper extends Mapper&lt;LongWritable, Text, Text, IntWritable&gt;

{

private static final int MISSING= 9999;

@Override

public void map(LongWritable key, Text value, Context context) throws IOException,InterruptedException

{

String line = value.toString();

String year = line.substring(15,19);

int airTemperature;

if (line.charAt(87) == ”+”)

{

airTemperature = Integer.parseInt(line.substring(88, 92));

}

Else

{

airTemperature = Integer.parseInt(line.substring(87, 92));

}
```

String quality = line.substring(92, 93);

if (airTemperature = MISSING && quality.matches(”[01459]”))

{

context.write(new Text(year), new IntWritable(airTemperature));

}

}

}

- The **map()** method is passed a key and a value.
- We convert the Text value containing the line of input into a Java String, then use its **substring()** method to extract the columns we are interested in.
- The **map()** method also provides an instance of Context to write the output to.
- In this case, we write the year as a Text object (since we are just using it as a key), and the temperature is wrapped in an **IntWritable**.
- We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

4.4 Reducer code

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function. The input types of the reduce function must match the output types of the map function: **Text** and **IntWritable**. And in this case, the output types of the reduce function are **Text** and **IntWritable**, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer extends Reducer<Text, IntWritable, Text, IntWritable>

{

@Override

public void reduce(Text key, Iterable<IntWritable>values,Context context) throws IOException, InterruptedException

{

int maxValue = Integer.MIN$_V$ALUE;

for(IntWritable value : values)

```
{

maxValue = Math.max(maxValue, value.get());

}

context.write(key, new IntWritable(maxValue));

}

}
```

5 Data Serialization in Hadoop

- **Serialization** is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage.
- **Deserialization** is the reverse process of turning a byte stream back into a series of structured objects.
- Serialization appears in two quite distinct areas of distributed data processing: for **interprocess communication** and for **persistent storage**.

In Hadoop, **Interprocess communication** between nodes in the system is implemented using **remote procedure calls (RPCs)**. The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message. In general, it is desirable that an RPC serialization format is:

1. **Compact:** A compact format makes the best use of network bandwidth, which is the most scarce resource in a data center.
2. **Fast:** Interprocess communication forms the backbone for a distributed system, so it is essential that there is as little performance overhead as possible for the serialization and deserialization process.
3. **Extensible:** Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers. For example, it should be possible to add a new argument to a method call, and have the new servers accept messages in the old format (without the new argument) from old clients.
4. **Interoperable:** For some systems, it is desirable to be able to support clients that are written in different languages to the server, so the format needs to be designed to make this possible.

the lifespan of an RPC is less than a second, whereas persistent data may be read years after it was written. As it turns out, the four desirable properties of an RPC's serialization format are also crucial for a persistent storage format. We want the storage format to be compact (to make efficient use of storage space), fast (so the overhead in reading or writing terabytes of data is minimal), extensible (so we can transparently read data written in an older format), and interoperable (so we can read or write persistent data using different languages).

5.1 Writable Interface

The Writable interface defines two methods: one for writing its state to a DataOutput binary stream, and one for reading its state from a DataInput binary stream:

We will use **IntWritable**, a wrapper for a Java int. We can create one and set its value using the **set()** method:

To examine the serialized form of the IntWritable,we write a small helper method that wraps a **java.io.ByteArrayOutputStrea** in a **java.io.DataOutputStream** (an implemen- tation of **java.io.DataOutput**) to capture the bytes in the serialized stream:

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
  void write(DataOutput out) throws IOException;
  void readFields(DataInput in) throws IOException;
}
```

Figure 7: Writable Interface

```
IntWritable writable = new IntWritable();
writable.set(163);
```

Equivalently, we can use the constructor that takes the integer value:

```
IntWritable writable = new IntWritable(163);
```

```
public static byte[] serialize(Writable writable) throws IOException {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  DataOutputStream dataOut = new DataOutputStream(out);
  writable.write(dataOut);
  dataOut.close();
  return out.toByteArray();
}
```

An integer is written using four bytes (as we see using JUnit 4 assertions):

```
byte[] bytes = serialize(writable);
assertThat(bytes.length, is(4));
```

The bytes are written in **big-endian order** (so the most significant byte is written to the stream first, this is dictated by the **java.io.DataOutput** interface), and we can see their hexadecimal representation by using a method on Hadoop's StringUtils:

assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));

Let's try deserialization. Again, we create a helper method to read a Writable object from a byte array:

```
public static byte[] deserialize(Writable writable, byte[] bytes)
    throws IOException {
  ByteArrayInputStream in = new ByteArrayInputStream(bytes);
  DataInputStream dataIn = new DataInputStream(in);
  writable.readFields(dataIn);
  dataIn.close();
  return bytes;
}
```

We construct a new, value-less, IntWritable, then call **deserialize()** to read from the output data that we just wrote. Then we check that its value, retrieved using the **get()** method, is the original value, 163