Topics

Map Reduce: Map Reduce framework and basics, how Map Reduce works, developing a Map Reduce application, unit tests with MR unit, test data and local tests, anatomy of a Map Reduce job run, failures, job scheduling, shuffle and sort, task execution, Map Reduce types, input formats, output formats, Map Reduce features, Real-world Map Reduce

MAP REDUCE FRAMEWORK & BASICS

Hadoop MapReduce is a programming paradigm at the heart of Apache Hadoop for providing massive scalability across hundreds or thousands of Hadoop clusters on commodity hardware. The MapReduce model processes large unstructured data sets with a distributed algorithm on a Hadoop cluster.

The term MapReduce represents two separate and distinct tasks Hadoop programs perform-Map Job and Reduce Job. Map job scales takes data sets as input and processes them to produce key value pairs. Reduce job takes the output of the Map job i.e. the key value pairs and aggregates them to produce desired results. The input and output of the map and reduce jobs are stored in HDFS.

MapReduce is primarily written in Java, therefore more often than not, it is advisable to learn Java for Hadoop MapReduce. MapReduce libraries have been written in many programming languages. Though it is mainly implemented in Java, there are non-Java interfaces such as Streaming (Scripting Languages), Pipes(C++), Pig, Hive, Cascading. In case of Streaming API, the corresponding jar is included and the mapper and reducer are written in Python/Scripting language. Hadoop which in turn uses MapReduce technique has a lot of use cases. On a general note it is used in scenario of needle in a haystack or for continuous monitoring of a huge system statistics. One such example is monitoring the traffic in a country road network and handling the traffic flow to prevent a jam. One common example is analyzing and storage of twitter data. It is also used in Log analysis which consists of various summations.

What is MapReduce?

A MapReduce is a data processing tool which is used to process the data parallelly in a distributed form. It was developed in 2004, on the basis of paper titled as "MapReduce: Simplified Data Processing on Large Clusters," published by Google.

The MapReduce is a paradigm which has two phases, the mapper phase, and the reducer phase. In the Mapper, the input is given in the form of a key-value pair. The output of the Mapper is fed to the reducer as input. The reducer runs only after the Mapper is over. The reducer too takes input in key-value format, and the output of reducer is the final output.

Steps in Map Reduce

The map takes data in the form of pairs and returns a list of <key, value> pairs. The keys will not be unique in this case.

Using the output of Map, sort and shuffle are applied by the Hadoop architecture. This sort and shuffle acts on these list of <key, value> pairs and sends out unique keys and a list of values associated with this unique key <key, list(values)>.

An output of sort and shuffle sent to the reducer phase. The reducer performs a defined function on a list of values for unique keys, and Final output <key, value> will be stored/displayed.

Sort and Shuffle

The sort and shuffle occur on the output of Mapper and before the reducer. When the Mapper task is complete, the results are sorted by key, partitioned if there are multiple reducers, and then written to disk. Using the input from each Mapper <k2,v2>, we collect all the values for each unique key k2. This output from the shuffle phase in the form of <k2, list(v2)> is sent as input to reducer phase.

Usage of MapReduce

It can be used in various application like document clustering, distributed sorting, and web link-graph reversal.

It can be used for distributed pattern-based searching.

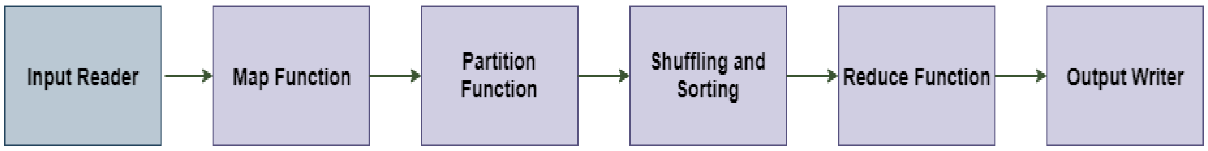
We can also use MapReduce in machine learning.

It was used by Google to regenerate Google's index of the World Wide Web.

It can be used in multiple computing environments such as multi-cluster, multi-core, and mobile environment.

Data Flow In MapReduce

MapReduce is used to compute the huge amount of data . To handle the upcoming data in a parallel and distributed form, the data has to flow from various phases.



Phases of MapReduce data flow

Input reader

The input reader reads the upcoming data and splits it into the data blocks of the appropriate size (64 MB to 128 MB). Each data block is associated with a Map function.

Once input reads the data, it generates the corresponding key-value pairs. The input files reside in HDFS.

Map function

The map function process the upcoming key-value pairs and generated the corresponding output key-value pairs. The map input and output type may be different from each other.

Partition function

The partition function assigns the output of each Map function to the appropriate reducer. The available key and value provide this function. It returns the index of reducers.

Shuffling and Sorting

The data are shuffled between/within nodes so that it moves out from the map and get ready to process for reduce function. Sometimes, the shuffling of data can take much computation time.

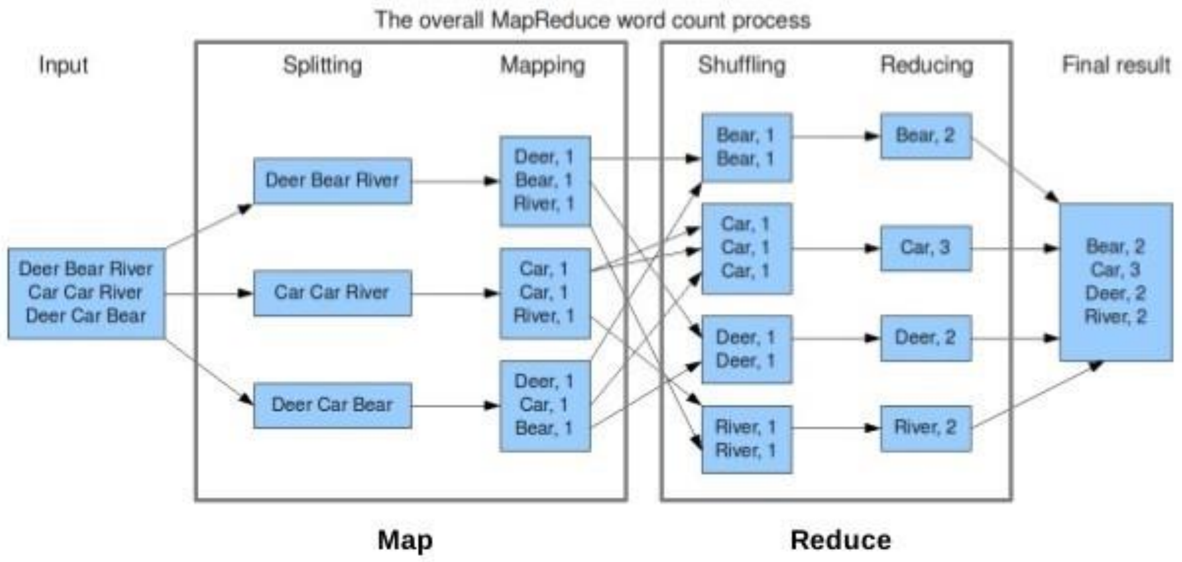
The sorting operation is performed on input data for Reduce function. Here, the data is compared using comparison function and arranged in a sorted form.

Reduce function

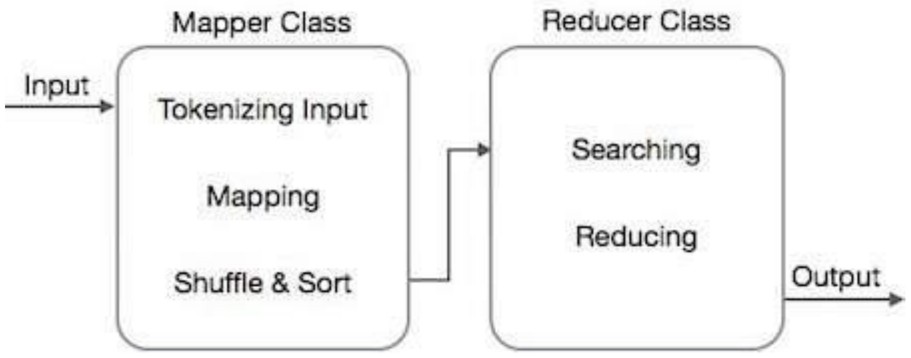
The Reduce function is assigned to each unique key. These keys are already arranged in sorted order. The values associated with the keys can iterate the Reduce and generates the corresponding output.

Output writer

Once the data flow from all the above phases, Output writer executes. The role of Output writer is to write the Reduce output to the stable storage.



Mapper class takes the input, tokenizes it, maps and sorts it. The output of Mapper class is used as input by Reducer class, which in turn searches matching pairs and reduces them.



MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems. In technical terms, MapReduce algorithm helps in sending the Map & Reduce tasks to appropriate servers in a cluster.

These mathematical algorithms may include the following −

Sorting

Searching

Indexing

TF-IDF

Sorting

Sorting is one of the basic MapReduce algorithms to process and analyze data. MapReduce implements sorting algorithm to automatically sort the output key-value pairs from the mapper by their keys.

Sorting methods are implemented in the mapper class itself.

In the Shuffle and Sort phase, after tokenizing the values in the mapper class, the Context class (user-defined class) collects the matching valued keys as a collection.

To collect similar key-value pairs (intermediate keys), the Mapper class takes the help of Raw Comparator class to sort the key-value pairs.

The set of intermediate key-value pairs for a given Reducer is automatically sorted by Hadoop to form key-values (K2, {V2, V2, …}) before they are presented to the Reducer.

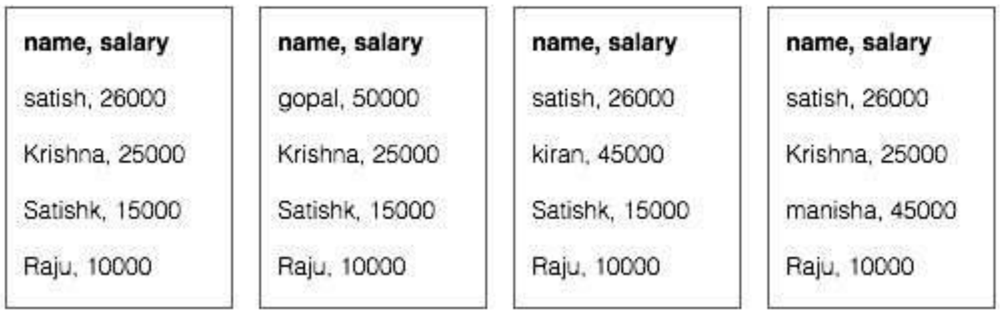
Searching

Searching plays an important role in MapReduce algorithm. It helps in the combiner phase (optional) and in the Reducer phase. Let us try to understand how Searching works with the help of an example.

Example

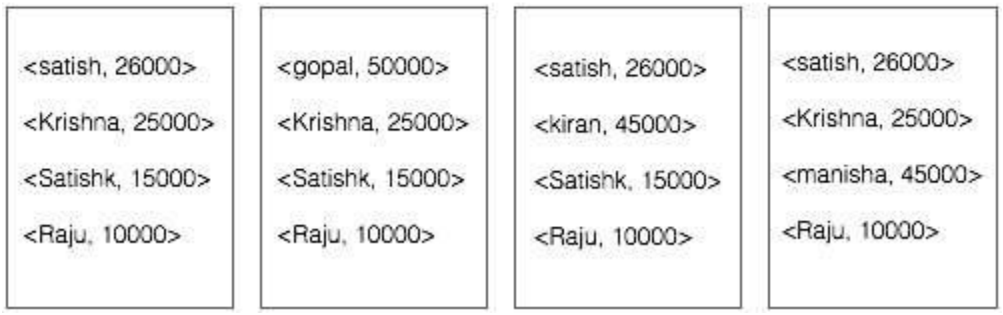
The following example shows how MapReduce employs Searching algorithm to find out the details of the employee who draws the highest salary in a given employee dataset.

Let us assume we have employee data in four different files − A, B, C, and D. Let us also assume there are duplicate employee records in all four files because of importing the employee data from all database tables repeatedly. See the following illustration.



name, salary	**name, salary**	**name, salary**	**name, salary**
satish, 26000	gopal, 50000	satish, 26000	satish, 26000
Krishna, 25000	Krishna, 25000	kiran, 45000	Krishna, 25000
Satishk, 15000	Satishk, 15000	Satishk, 15000	manisha, 45000
Raju, 10000	Raju, 10000	Raju, 10000	Raju, 10000

The Map phase processes each input file and provides the employee data in key-value pairs (<k, v> : <emp name, salary>). See the following illustration.



The combiner phase (searching technique) will accept the input from the Map phase as a key-value pair with employee name and salary. Using searching technique, the combiner will check all the employee salary to find the highest salaried employee in each file.

Reducer phase − Form each file, you will find the highest salaried employee. To avoid redundancy, check all the <k, v> pairs and eliminate duplicate entries, if any. The same algorithm is used in between the four <k, v> pairs, which are coming from four input files. The final output should be as follows −

<gopal, 50000>

Indexing

Normally indexing is used to point to a particular data and its address. It performs batch indexing on the input files for a particular Mapper.

The indexing technique that is normally used in MapReduce is known as inverted index. Search engines like Google and Bing use inverted indexing technique. Let us try to understand how Indexing works with the help of a simple example.

Example

The following text is the input for inverted indexing. Here T[0], T[1], and t[2] are the file names and their content are in double quotes.

T[0] = "it is what it is"

T[1] = "what is it"

T[2] = "it is a banana"

After applying the Indexing algorithm, we get the following output −

"a": {2}

"banana": {2}

"is": {0, 1, 2}

"it": {0, 1, 2}

"what": {0, 1}

Here "a": {2} implies the term "a" appears in the T[2] file. Similarly, "is": {0, 1, 2} implies the term "is" appears in the files T[0], T[1], and T[2].

TF-IDF

TF-IDF is a text processing algorithm which is short for Term Frequency − Inverse Document Frequency. It is one of the common web analysis algorithms. Here, the term 'frequency' refers to the number of times a term appears in a document.

Term Frequency (TF)

It measures how frequently a particular term occurs in a document. It is calculated by the number of times a word appears in a document divided by the total number of words in that document.

Inverse Document Frequency (IDF)

It measures the importance of a term. It is calculated by the number of documents in the text database divided by the number of documents where a specific term appears.

While computing TF, all the terms are considered equally important. That means, TF counts the term frequency for normal words like “is”, “a”, “what”, etc. Thus we need to know the frequent terms while scaling up the rare ones

Terminology

Pay Load − Applications implement the Map and the Reduce functions, and form the core of the job.

Mapper − Mapper maps the input key/value pairs to a set of intermediate key/value pair.

Named Node − Node that manages the Hadoop Distributed File System (HDFS).

Data Node − Node where data is presented in advance before any processing takes place.

Master Node − Node where Job Tracker runs and which accepts job requests from clients.

Slave Node − Node where Map and Reduce program runs.

Job Tracker − Schedules jobs and tracks the assign jobs to Task tracker.

Task Tracker − Tracks the task and reports status to Job Tracker.

Job − A program is an execution of a Mapper and Reducer across a dataset.

Task − An execution of a Mapper or a Reducer on a slice of data.

Task Attempt − A particular instance of an attempt to execute a task on a Slave Node.

Advantages of MapReduce

The two biggest advantages of MapReduce are:

1. Parallel Processing:

In MapReduce, we are dividing the job among multiple nodes and each node works with a part of the job simultaneously. So, MapReduce is based on Divide and Conquer paradigm which helps us to process the data using different machines. As the data is processed by multiple machines instead of a single machine in parallel, the time taken to process the data gets reduced.

2. Data Locality:

Instead of moving data to the processing unit, we are moving the processing unit to the data in the MapReduce Framework. In the traditional system, we used to bring data to the processing unit and process it. But, as the data grew and became very huge, bringing this huge amount of data to the processing unit posed the following issues:

Moving huge data to processing is costly and deteriorates the network performance.

Processing takes time as the data is processed by a single unit which becomes the bottleneck.

The master node can get over-burdened and may fail.

Now, MapReduce allows us to overcome the above issues by bringing the processing unit to the data. So, as you can see in the above image that the data is distributed among multiple nodes where each node processes the part of the data residing on it. This allows us to have the following advantages:

It is very cost-effective to move processing unit to the data.

The processing time is reduced as all the nodes are working with their part of the data in parallel.

Every node gets a part of the data to process and therefore, there is no chance of a node getting overburdened.

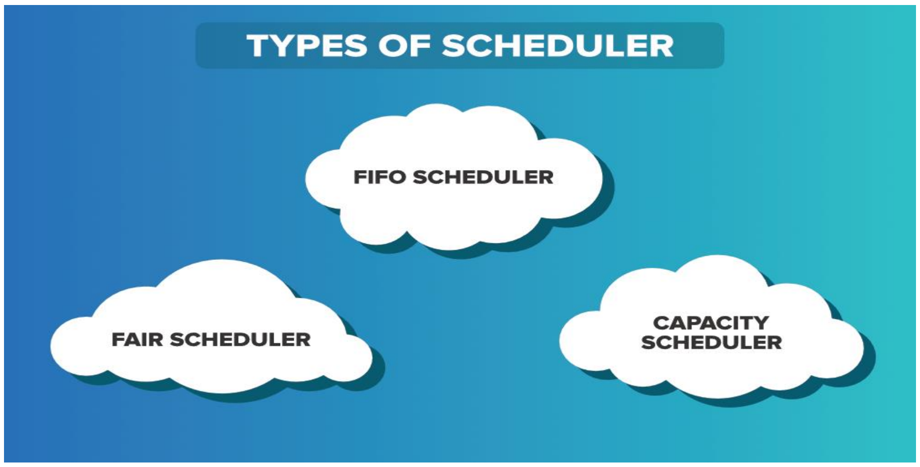
JOB SCHEDULING

In Hadoop, we can receive multiple jobs from different clients to perform. The Map-Reduce framework is used to perform multiple tasks in parallel in a typical Hadoop cluster to process large size datasets at a fast rate. This Map-Reduce Framework is responsible for scheduling and monitoring the tasks given by different clients in a Hadoop cluster. But this method of scheduling jobs is used prior to **Hadoop 2**.

Now in Hadoop 2, we have YARN (Yet Another Resource Negotiator). In YARN we have separate Daemons for performing Job scheduling, Monitoring, and Resource Management as Application Master, Node Manager, and Resource Manager respectively.

Here, Resource Manager is the Master Daemon responsible for tracking or providing the resources required by any application within the cluster, and Node Manager is the slave Daemon which monitors and keeps track of the resources used by an application and sends the feedback to Resource Manager.

Schedulers and **Applications Manager** are the 2 major components of resource Manager. The Scheduler in YARN is totally dedicated to scheduling the jobs, it can not track the status of the application. On the basis of required resources, the schedular performs or we can say schedule the Jobs.



There are mainly 3 types of Schedulers in Hadoop:

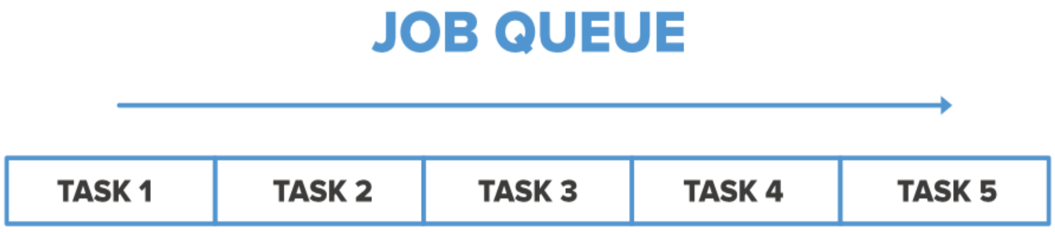
FIFO (First In First Out) Scheduler.

Capacity Scheduler.

Fair Scheduler.

These Schedulers are actually a kind of algorithm that we use to schedule tasks in a Hadoop cluster when we receive requests from different-different clients.

A **Job queue** is nothing but the collection of various tasks that we have received from our various clients. The tasks are available in the queue and we need to schedule this task on the basis of our requirements.



1. FIFO Scheduler

As the name suggests FIFO i.e. First In First Out, so the tasks or application that comes first will be served first. This is the default Schedular we use in Hadoop. The tasks are placed in a queue and the tasks are performed in their submission order. In this method, once the job is scheduled, no intervention is allowed. So sometimes the high priority process has to wait for a long time since the priority of the task does not matter in this method.

Advantage:

No need for configuration

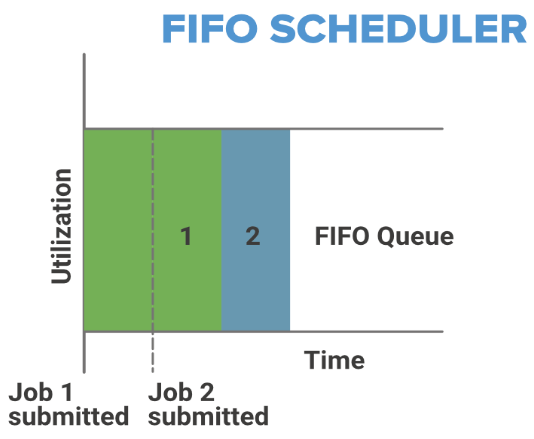
First Come First Serve

simple to execute

Disadvantage:

Priority of task doesn’t matte, so high priority jobs needs to wait

Not suitable for shared cluster



2. Capacity Scheduler

In Capacity Scheduler we have multiple job queues for scheduling our tasks. The Capacity Scheduler allows multiple occupants to share a large size Hadoop cluster. In Capacity Schedular corresponding for each job queue, we provide some slots or cluster resources for performing job operation. Each job queue has it’s own slots to perform its task. In case we have tasks to perform in only one queue then the tasks of that queue can access the slots of other queues also as they are free to use, and when the new task enters to some other queue then jobs in running in its own slots of the cluster are replaced with its own job.

Capacity Scheduler also provides a level of abstraction to know which occupant is utilizing the more cluster resource or slots, so that the single user or application doesn’t take disappropriate or unnecessary slots in the cluster. The capacity Schedular mainly contains 3 types of the queue that are root, parent, and leaf which are used to represent cluster, organization, or any subgroup, application submission respectively.

Advantage:

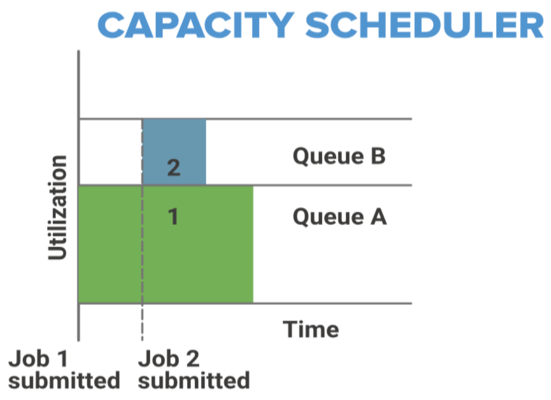
Best for working with Multiple clients or priority jobs in a Hadoop cluster

Maximizes throughput in the Hadoop cluster

Disadvantage:

More complex

Not easy to configure for everyone



3. Fair Scheduler

The Fair Scheduler is very much similar to that of the capacity scheduler. The priority of the job is kept in consideration. With the help of Fair Scheduler, the YARN applications can share the resources in the large Hadoop Cluster and these resources are maintained dynamically so no need for prior capacity. The resources are distributed in such a manner that all applications within a cluster get an equal amount of time. Fair Scheduler takes Scheduling decision on the basis of memory, we can configure it to work with CPU also.

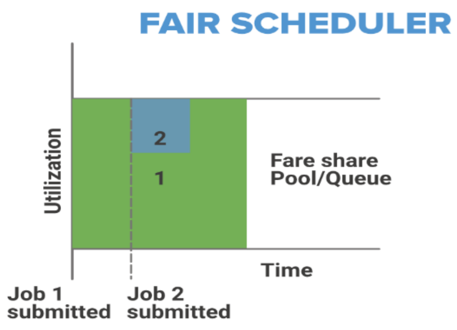
As we told you it is similar to Capacity Scheduler but the major thing to notice is that in Fair Scheduler whenever any high priority job arises in the same queue, the task is processed in parallel by replacing some portion from the already dedicated slots.

Advantages:

Resources assigned to each application depends upon its priority.

it can limit the concurrent running task in a perticular pool or queue.

Disadvantages: The configuration is required.



Failures in MapReduce

In the MapReduce 1 runtime there are three failure modes to consider: failure of the running task, failure of the tastracker, and failure of the jobtracker. Let's look at each in turn.

Task Failure

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. This behavior is governed by the stream.non.zero.exit.is.failure property (the default is true).

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 and marks the attempt as failed.

When the jobtracker is notified of a task attempt that has failed (by the tasktracker's heartbeat call), it will reschedule execution of the task. The jobtracker will try to avoid rescheduling the task on a tasktracker where it has previously failed. Furthermore, if a task fails four times (or more), it will not be retried further. This value is configurable: the maximum number of attempts to run a task is controlled by the mapred.map.max.attempts property for map tasks and mapred.reduce.max.attempts for reduce tasks. By default, if any task fails four times (or whatever the maximum number of attempts is configured to), the whole job fails.

A task attempt may also be *killed*, which is different from it failing. A task attempt may be killed because it is a speculative duplicate, or because the tasktracker it was running on failed, and the jobtracker marked all the task attempts running on it as killed. Killed task attempts do not count against the number of attempts to run the task (as set
by mapred.map.max.attempts and mapred.reduce.max.attempts), since it wasn't the task's fault that an attempt was killed.

Users may also kill or fail task attempts using the web UI or the command line.

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 (if it hasn't received one for 10 minutes, configured via the mapred.task tracker.expiry.interval property, in milliseconds) and remove it from its pool of tasktrackers to schedule tasks on. The jobtracker arranges for map tasks that were run and completed successfully on that tasktracker to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed tasktracker's local filesystem may not be accessible to the reduce task. Any tasks in progress are also rescheduled.

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 (set by (mapred.max.tracker.failures), then the jobtracker records this as a fault. A tasktracker is blacklisted if the number of faults is over some minimum threshold (four, set by mapred.max.tracker.blacklists) and is significantly higher than the average number of faults for tasktrackers in the cluster cluster.

Blacklisted tasktrackers are not assigned tasks, but they continue to communicate with the jobtracker. Faults expire over time (at the rate of one per day), so tasktrackers get the chance to run jobs again simply by leaving them running. Alternatively, if there is an underlying fault that can be fixed (by replacing hardware, for example), the tasktracker will be removed from the jobtracker's blacklist after it restarts and rejoins the 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.

After restarting a jobtracker, any jobs that were running at the time it was stopped will need to be re-submitted. There is a configuration option that attempts to recover any running jobs (mapred.jobtracker.restart.recover, turned off by default), however it is known not to work reliably, so should not be used.

SHUFFLE & SORT IN MAP REDUCE

Shuffle phase in Hadoop transfers the map output from Mapper to a Reducer in MapReduce. **Sort phase** in MapReduce covers the merging and sorting of map outputs. Data from the mapper are grouped by the key, split among reducers and sorted by the key. Every reducer obtains all values associated with the same key. Shuffle and sort phase in Hadoop occur simultaneously and are done by the MapReduce framework.

Let us now understand both these processes in details below:

Shuffling in MapReduce

The process of transferring data from the mappers to reducers is known as shuffling i.e. the process by which the system performs the sort and transfers the map output to the reducer as input. So, MapReduce shuffle phase is necessary for the reducers, otherwise, they would not have any input (or input from every mapper). As shuffling can start even before the map phase has finished so this saves some time and completes the tasks in lesser time.

Sorting in MapReduce

The keys generated by the mapper are automatically sorted by MapReduce Framework, i.e. Before starting of reducer, all intermediate **key-value pairs** in MapReduce that are generated by mapper get sorted by key and not by value. Values passed to each reducer are not sorted; they can be in any order.

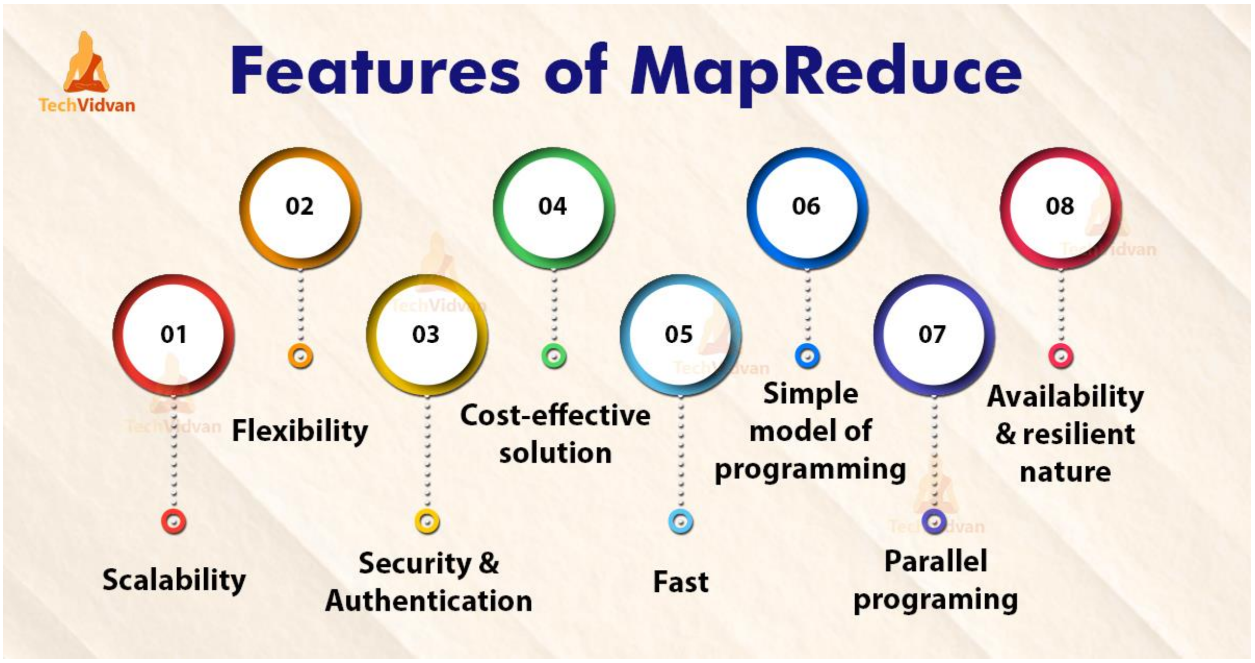
Sorting in Hadoop helps reducer to easily distinguish when a new reduce task should start. This saves time for the reducer. Reducer starts a new reduce task when the next key in the sorted input data is different than the previous. Each reduce task takes key-value pairs as input and generates key-value pair as output.

Note that shuffling and sorting in Hadoop MapReduce is not performed at all if you specify zero reducers (setNumReduceTasks(0)). Then, the MapReduce job stops at the map phase, and the map phase does not include any kind of sorting (so even the map phase is faster).

Secondary Sorting in MapReduce

If we want to sort reducer's values, then the secondary sorting technique is used as it enables us to sort the values (in ascending or descending order) passed to each reducer.

FEATURES OF MAP REDUCE



1. Scalability

Apache Hadoop is a highly scalable framework. This is because of its ability to store and distribute huge data across plenty of servers. All these servers were inexpensive and can operate in parallel. We can easily scale the storage and computation power by adding servers to the cluster.

Hadoop MapReduce programming enables organizations to run applications from large sets of nodes which could involve the use of thousands of terabytes of data. Hadoop MapReduce programming enables business organizations to run applications from large sets of nodes. This can use thousands of terabytes of data.

2. Flexibility

MapReduce programming enables companies to access new sources of data. It enables companies to operate on different types of data. It allows enterprises to access structured as well as unstructured data, and derive significant value by gaining insights from the multiple sources of data. Additionally, the MapReduce framework also provides support for the multiple languages and data from sources ranging from email, social media, to clickstream. The MapReduce processes data in simple key-value pairs thus supports data type including meta-data, images, and large files. Hence, MapReduce is flexible to deal with data rather than traditional DBMS.

Security and Authentication

The MapReduce programming model uses HBase and HDFS security platform that allows access only to the authenticated users to operate on the data. Thus, it protects unauthorized access to system data and enhances system security.

4. Cost-effective solution

Hadoop's scalable architecture with the MapReduce programming framework allows the storage and processing of large data sets in a very affordable manner.

5. Fast

Hadoop uses a distributed storage method called as a Hadoop Distributed File System that basically implements a mapping system for locating data in a cluster. The tools that are used for data processing, such as MapReduce programming, are generally located on the very same servers that allow for the faster processing of data.

So, Even if we are dealing with large volumes of unstructured data, Hadoop MapReduce just takes minutes to process terabytes of data. It can process petabytes of data in just an hour.

6. Simple model of programming

Amongst the various features of Hadoop MapReduce, one of the most important features is that it is based on a simple programming model. Basically, this allows programmers to develop the MapReduce programs which can handle tasks easily and efficiently.

The MapReduce programs can be written in Java, which is not very hard to pick up and is also used widely. So, anyone can easily learn and write MapReduce programs and meet their data processing needs.

7. Parallel Programming

One of the major aspects of the working of MapReduce programming is its parallel processing. It divides the tasks in a manner that allows their execution in parallel.
The parallel processing allows multiple processors to execute these divided tasks. So the entire program is run in less time.

8. Availability and resilient nature

Whenever the data is sent to an individual node, the same set of data is forwarded to some other nodes in a cluster. So, if any particular node suffers from a failure, then there are always other copies present on other nodes that can still be accessed whenever needed. This assures high availability of data.

One of the major features offered by Apache Hadoop is its fault tolerance. The Hadoop MapReduce framework has the ability to quickly recognizing faults that occur. It then applies a quick and automatic recovery solution. This feature makes it a game-changer in the world of big data processing.

Task Execution:

Below steps defines how the task is executed:

1. It copies the jar file from the shared file system (HDFS).

2. It creates a local working directory and un-jars the jar file into local file system.

3. It creates an instance of task runner.

4. It starts Task Runner in a new JVM to run the map or reduce task.

5. The child process communicates the progress to parent process.

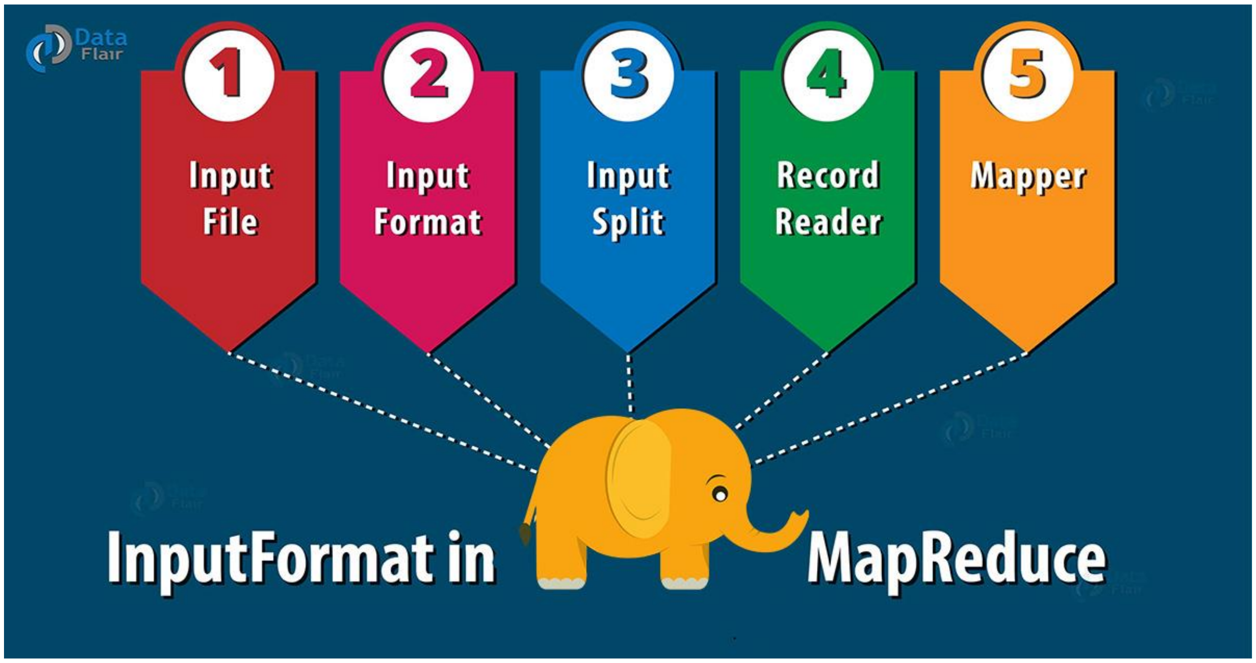
6. Each task can perform setup and clean up actions based on OutputComitter.

7. The input provided via stdin and get output via stdout from the running process even if the map or reduce tasks ran via pipes or socket in case of streaming.

Hadoop InputFormat, Types of InputFormat in MapReduce

1. Objective

Hadoop InputFormat checks the Input-Specification of the job. InputFormat split the Input file into InputSplit and assign to individual Mapper. In this Hadoop InputFormat Tutorial, we will learn what is InputFormat in Hadoop **MapReduce**, different methods to get the data to the mapper and different types of InputFormat in Hadoop like FileInputFormat in Hadoop, TextInputFormat, KeyValueTextInputFormat, etc



What is Hadoop InputFormat?

How the input files are split up and read in Hadoop is defined by the InputFormat. An Hadoop InputFormat is the first component in Map-Reduce, it is responsible for creating the input splits and dividing them into records. If you are not familiar with MapReduce Job Flow, so follow our **Hadoop MapReduce Data flow tutorial** for more understanding.

Initially, the data for a MapReduce task is stored in input files, and input files typically reside in **HDFS**. Although these files format is arbitrary, line-based log files and binary format can be used. Using InputFormat we define how these input files are split and read. The InputFormat class is one of the fundamental classes in the Hadoop MapReduce framework which provides the following functionality:

The files or other objects that should be used for input is selected by the InputFormat.

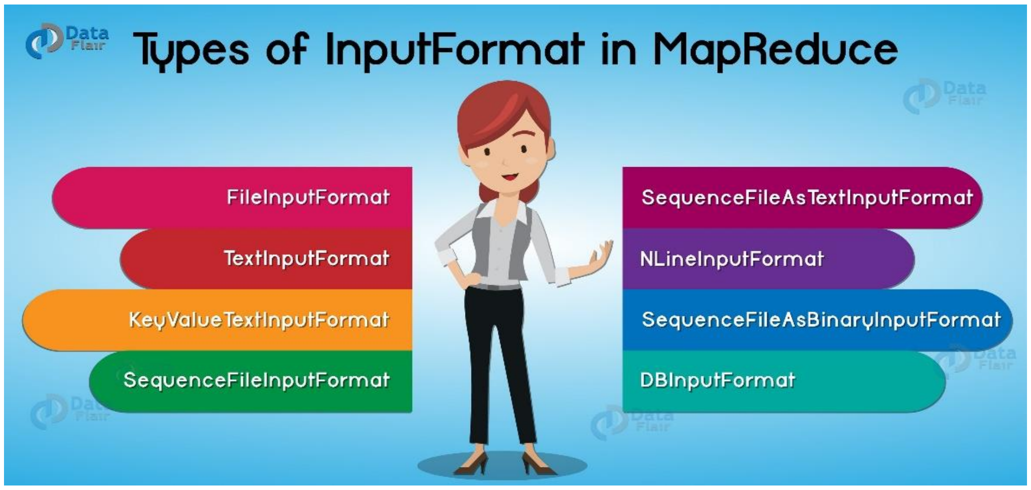
InputFormat defines the Data splits, which defines both the size of individual **Map tasks** and its potential execution server.

InputFormat defines the **RecordReader**, which is responsible for reading actual records from the input files.

3. How we get the data to mapper?

We have 2 methods to get the data to **mapper** in MapReduce: getsplits() and createRecordReader() as shown below:

```
[php]public abstract class InputFormat<K, V>
{
public abstract List<InputSplit> getSplits(JobContext context)
throws IOException, InterruptedException;
public abstract RecordReader<K, V>
createRecordReader(InputSplit split,
TaskAttemptContext context) throws IOException,
InterruptedException;
}[/php]
```



Types of InputFormat

4.1. FileInputFormat in Hadoop

It is the base class for all file-based InputFormats. Hadoop FileInputFormat specifies input directory where data files are located. When we start a Hadoop job, FileInputFormat is provided with a path containing files to read. FileInputFormat will read all files and divides these files into one or more InputSplits.

4.2. TextInputFormat

It is the default InputFormat of MapReduce. TextInputFormat treats each line of each input file as a separate record and performs no parsing. This is useful for unformatted data or line-based records like log files.

Key – It is the byte offset of the beginning of the line within the file (not whole file just one split), so it will be unique if combined with the file name.

Value – It is the contents of the line, excluding line terminators.

4.3. KeyValueTextInputFormat

It is similar to TextInputFormat as it also treats each line of input as a separate record. While TextInputFormat treats entire line as the value, but the KeyValueTextInputFormat breaks the line itself into key and value by a tab character ('/t'). Here Key is everything up to the tab character while the value is the remaining part of the line after tab character.

4.4. SequenceFileInputFormat

Hadoop **SequenceFileInputForma**t is an InputFormat which reads sequence files. Sequence files are binary files that stores sequences of binary **key-value pairs**. Sequence files block-compress and provide direct serialization and deserialization of several arbitrary data types (not just text). Here Key & Value both are user-defined.

4.5. SequenceFileAsTextInputFormat

Hadoop **SequenceFileAsTextInputFormat** is another form of SequenceFileInputFormat which converts the sequence file key values to Text objects. By calling **'tostring()'** conversion is performed on the keys and values. This InputFormat makes sequence files suitable input for streaming.

4.6. SequenceFileAsBinaryInputFormat

Hadoop **SequenceFileAsBinaryInputFormat** is a SequenceFileInputFormat using which we can extract the sequence file's keys and values as an opaque binary object.

4.7. NLineInputFormat

Hadoop **NLineInputFormat** is another form of TextInputFormat where the keys are byte offset of the line and values are contents of the line. Each mapper receives a variable number of lines of input with TextInputFormat and KeyValueTextInputFormat and the number depends on the size of the split and the length of the lines. And if we want our mapper to receive a fixed number of lines of input, then we use

NLineInputFormat.
N is the number of lines of input that each mapper receives. By default (N=1), each mapper receives exactly one line of input. If N=2, then each split contains two lines. One mapper will receive the first two Key-Value pairs and another mapper will receive the second two key-value pairs.

4.8. DBInputFormat

Hadoop **DBInputFormat** is an InputFormat that reads data from a relational database, using JDBC. As it doesn't have portioning capabilities, so we need to careful not to swamp the database from which we are reading too many mappers. So it is best for loading relatively small datasets, perhaps for joining with large datasets from HDFS using MultipleInputs. Here Key is LongWritables while Value is DBWritables.

What is Hadoop Output Format?

Before we start with Hadoop Output Format in **MapReduce**, let us first see what is a RecordWriter in MapReduce and what is its role in MapReduce?

i. Hadoop RecordWriter

As we know, **Reducer** takes as input a set of an intermediate **key-value pair** produced by the **mapper** and runs a reducer function on them to generate output that is again zero or more key-value pairs.

RecordWriter writes these output key-value pairs from the Reducer phase to output files.

ii. Hadoop Output Format

As we saw above, Hadoop RecordWriter takes output data from Reducer and writes this data to output files. The way these output key-value pairs are written in output files by RecordWriter is determined by the Output Format. The Output Format and **InputFormat** functions are alike. OutputFormat instances provided by Hadoop are used to write to files on the **HDFS** or local disk. OutputFormat describes the output-specification for a **Map-Reduce job**. On the basis of output specification;

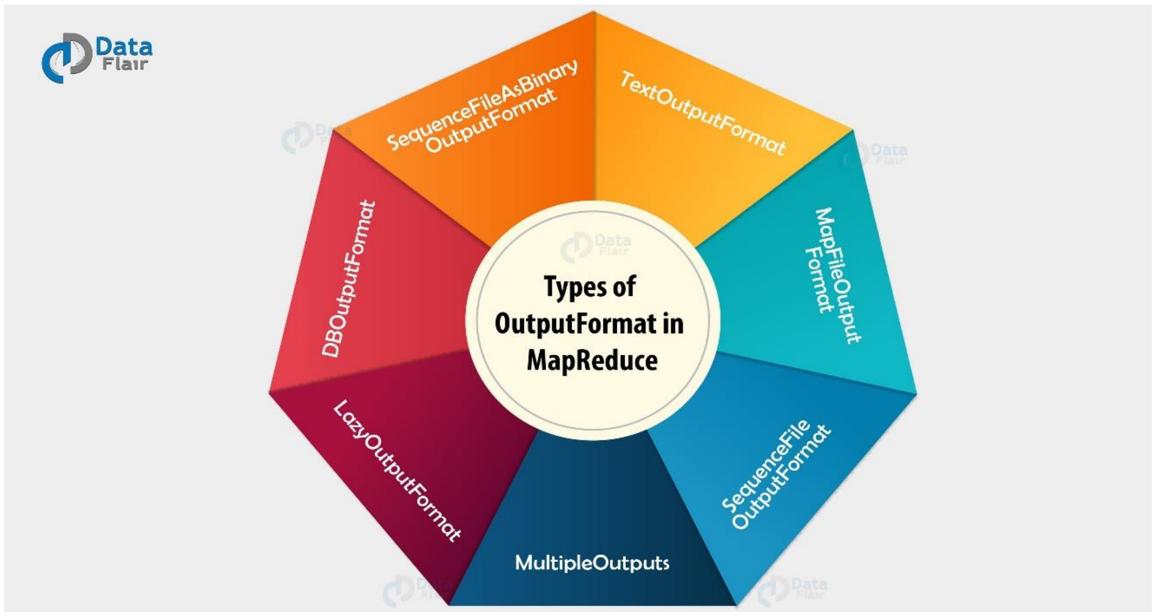
MapReduce job checks that the output directory does not already exist.

OutputFormat provides the RecordWriter implementation to be used to write the output files of the job. Output files are stored in a FileSystem.

FileOutputFormat.setOutputPath() method is used to set the output directory. Every Reducer writes a separate file in a common output directory.

3. Types of Hadoop Output Formats

There are various types of Hadoop OutputFormat. Let us see some of them below:


Types of OutputFormat in MapReduce
TextOutputFormat
MapFileOutput Format
SequenceFile OutputFormat
MultipleOutputs
LazyOutputFormat
DBOutputFormat
SequenceFileAsBinary OutputFormat

Types of Hadoop Output Formats

TextOutputFormat

MapReduce default Hadoop reducer Output Format is **TextOutputFormat**, which writes (key, value) pairs on individual lines of text files and its keys and values can be of any type since TextOutputFormat turns them to string by calling toString() on them. Each key-value pair is separated by a tab character, which can be changed using ***MapReduce.output.textoutputformat.separator*** property. KeyValueTextOutputFormat is used for reading these output text files since it breaks lines into key-value pairs based on a configurable separator.

ii. SequenceFileOutputFormat

It is an Output Format which writes sequences files for its output and it is intermediate format use between MapReduce jobs, which rapidly serialize arbitrary data types to the file; and the corresponding SequenceFileInputFormat will deserialize the file into the same types and presents the data to the next mapper in the same manner as it was emitted by the previous reducer, since these are compact and readily compressible. Compression is controlled by the static methods on SequenceFileOutputFormat.

iii. SequenceFileAsBinaryOutputFormat

It is another form of SequenceFileInputFormat which writes keys and values to sequence file in binary format.

iv. MapFileOutputFormat

It is another form of FileOutputFormat in Hadoop Output Format, which is used to write output as map files. The key in a MapFile must be added in order, so we need to ensure that reducer emits keys in sorted order.
Any doubt yet in Hadoop Oputput Format? Please Ask.

v. MultipleOutputs

It allows writing data to files whose names are derived from the output keys and values, or in fact from an arbitrary string.

vi. LazyOutputFormat

Sometimes FileOutputFormat will create output files, even if they are empty. LazyOutputFormat is a wrapper OutputFormat which ensures that the output file will be created only when the record is emitted for a given partition.

vii. DBOutputFormat

DBOutputFormat in Hadoop is an Output Format for writing to relational databases and **HBase**. It sends the reduce output to a SQL table. It accepts key-value pairs, where the key has a type extending DBwritable. Returned RecordWriter writes only the key to the database with a batch SQL query.

REAL WORLD MAP REDUCE

Let us understand it with a real-time example, and the example helps you understand Mapreduce Programming Model in a story manner:

Suppose the Indian government has assigned you the task to count the population of India. You can demand all the resources you want, but you have to do this task in 4 months. Calculating the population of such a large country is not an easy task for a single person(you). So what will be your approach?.

One of the ways to solve this problem is to divide the country by states and assign individual in-charge to each state to count the population of that state.

Task Of Each Individual: Each Individual has to visit every home present in the state and need to keep a record of each house members as:

State_Name Member_House1

State_Name Member_House2

State_Name Member_House3

.

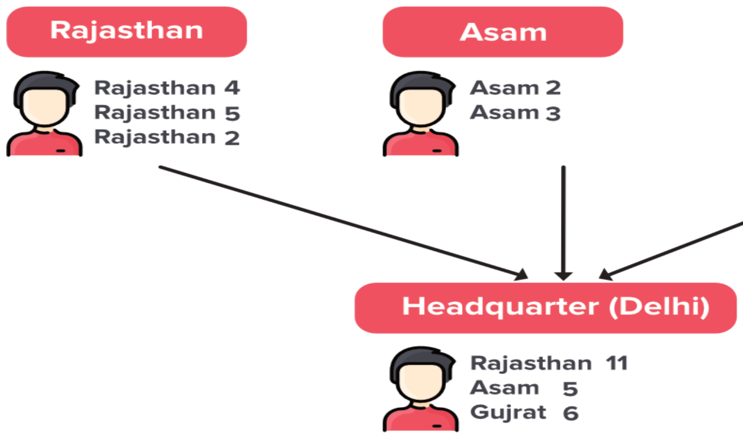
.

State_Name Member_House n

.

.

For Simplicity, we have taken only three states.



This is a simple Divide and Conquer approach and will be followed by each individual to count people in his/her state.

Once they have counted each house member in their respective state. Now they need to sum up their results and need to send it to the Head-quarter at New Delhi.

We have a trained officer at the Head-quarter to receive all the results from each state and aggregate them by each state to get the population of that entire state. and Now, with this approach, you are easily able to count the population of India by summing up the results obtained at Head-quarter.

The Indian Govt. is happy with your work and the next year they asked you to do the same job in 2 months instead of 4 months. Again you will be provided with all the resources you want.

Since the Govt. has provided you with all the resources, you will simply double the number of assigned individual in-charge for each state from one to two. For that divide each state in 2 division and assigned different in-charge for these two divisions as:

State_Name_Incharge_division1

State_Name_Incharge_division2

Similarly, each individual in charge of its division will gather the information about members from each house and keep its record.

We can also do the same thing at the Head-quarters, so let’s also divide the Head-quarter in two division as:

Head-qurter_Division1

Head-qurter_Division2

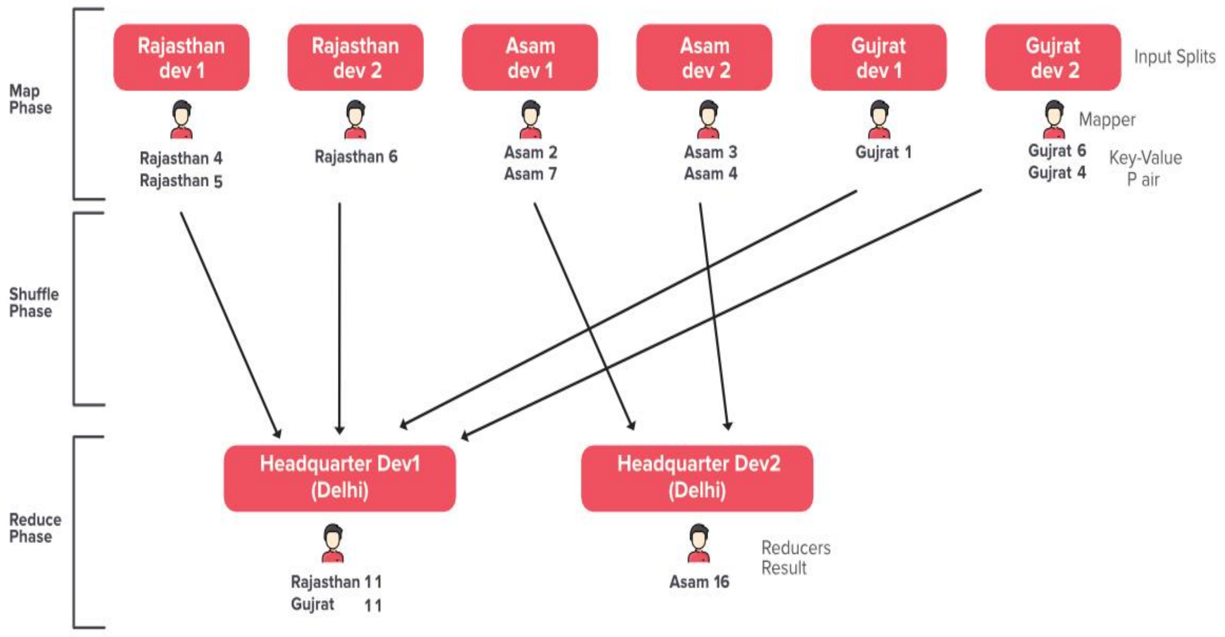
Now with this approach, you can find the population of India in two months. But there is a small problem with this, we never want the divisions of the same state to send their result at different Head-quarters then, in that case, we have the partial population of that state in Head-quarter_Division1 and Head-quarter_Division2 which is inconsistent because we want consolidated population by the state, not the partial counting.

One easy way to solve is that we can instruct all individuals of a state to either send there result to Head-quarter_Division1 or Head-quarter_Division2. Similarly, for all the states.

Our problem has been solved, and you successfully did it in two months.

Now, if they ask you to do this process in a month, you know how to approach the solution.

Great, now we have a good scalable model that works so well. The model we have seen in this example is like the MapReduce Programming model. so now you must be aware that MapReduce is a programming model, not a programming language.



Now let’s discuss the phases and important things involved in our model.

1. Map Phase: The Phase where the individual in-charges are collecting the population of each house in their division is Map Phase.

Mapper: Involved individual in-charge for calculating population

Input Splits: The state or the division of the state

Key-Value Pair: Output from each individual Mapper like the key is Rajasthan and value is 2

2. Reduce Phase: The Phase where you are aggregating your result

Reducers: Individuals who are aggregating the actual result. Here in our example, the trained-officers. Each Reducer produce the output as a key-value pair

3. Shuffle Phase: The Phase where the data is copied from Mappers to Reducers is Shuffler's Phase. It comes in between Map and Reduces phase. Now the Map Phase, Reduce Phase, and Shuffler Phase our the three main Phases of our Mapreduce.

ANATOMY OFA MAP REDUCE JOB RUN

You can run a MapReduce job with a single line of code: JobClient.runJob(conf). It's very short, but it conceals a great deal of processing behind the scenes. This section uncovers the steps Hadoop takes to run a job.

The whole process is illustrated in belowFigure . At the highest level, there are four independent entities:

The client, which submits the MapReduce job.

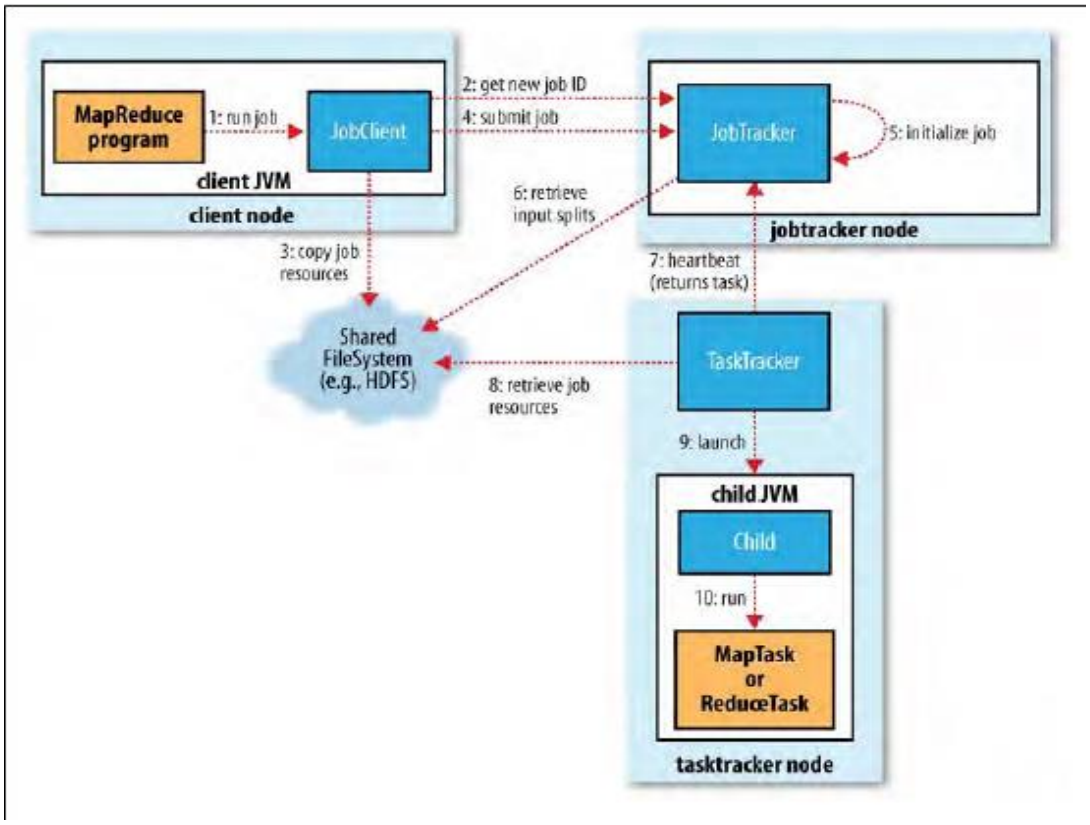
The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.

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 , which is used for sharing job files between the other entities.

Job Submission

The is runJob() method on JobClient a convenience method that creates a new JobClient instance and calls submitJob() on it . Having submitted the job, runJob() polls the job's progress once a second and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.



The job submission process implemented by JobClient's submitJob() method does the following:

1. 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 input splits, to the jobtracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job(step3)

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 JobClient 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 JobConf, which is set by the setNumReduce Tasks() 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).

Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. There are various scheduling algorithms as explained later in this chapter (see “Job Scheduling”), but the default one simply maintains a priority list of jobs. Having chosen a job, the jobtracker now chooses a task for the job.

Tasktrackers have a fixed number of slots for map tasks and for reduce tasks: for example, a tasktracker may be able to run two map tasks and two reduce tasks simultaneously. (The precise number depends on the number of cores and the amount of memory on the tasktracker; see “Memory”) The default scheduler fills empty map task slots before reduce task slots, so if the tasktracker has at least one empty map task slot, the jobtracker will select a map task; otherwise, it will select a reduce task.

To choose a reduce task, the jobtracker simply takes the next in its list of yet-to-be-run reduce tasks, since there are no data locality considerations. For a map task, however, it takes account of the tasktracker’s network location and picks a task whose input split is as close as possible to the tasktracker. In the optimal case, the task is data-local, that is, running on the same node that the split resides on. Alternatively, the task may be rack-local: on the same rack, but not the same node, as the split. Some tasks are neither data-local nor rack-local and retrieve their data from a different rack from the one theyare running on. You can tell the proportion of each type of task by looking at a job’s counters .

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. It also copies any files needed from the distributed cache by the application to the local disk; see “Distributed Cache” (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; see “Task JVM Reuse”

The child process communicates with its parent through the umbilical interface. This way it informs the parent of the task’s progress every few seconds until the task is complete.

Streaming and Pipes

Both Streaming and Pipes run special map and reduce tasks for the purpose of launching the user-supplied executable and communicating with it (Figure).

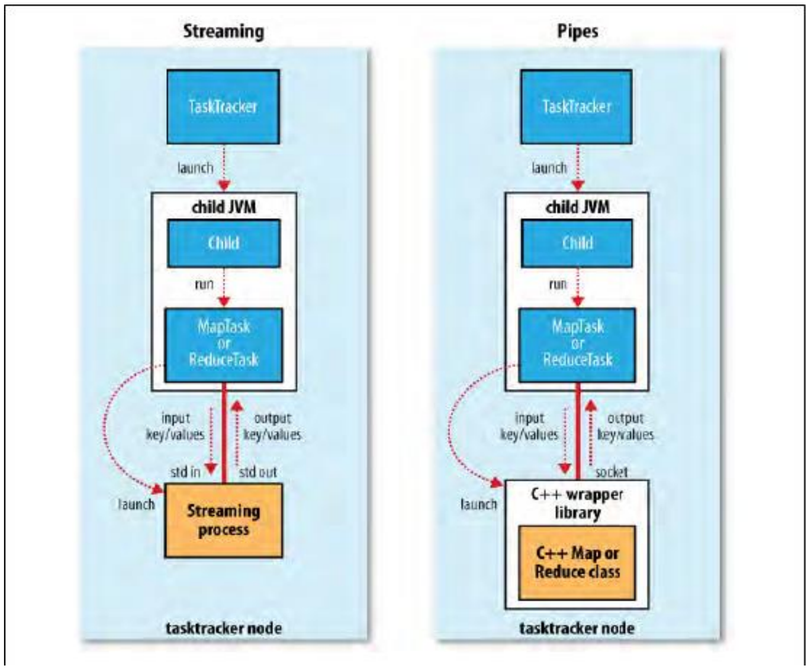
In the case of Streaming, the Streaming task communicates with the process (which may be written in any language) using standard input and output streams. The Pipes task, on the other hand, listens on a socket and passes the C++ process a port number in its environment, so that on startup, the C++ process can establish a persistent socket connection back to the parent Java Pipes task.

In both cases, during execution of the task, the Java process passes input key-value pairs to the external process, which runs it through the user-defined map or reduce function and passes the output key-value pairs back to the Java process. From the tasktracker's point of view, it is as if the tasktracker child process ran the map or reduce code itself.

Progress and Status Updates

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run. Because this is a significant length of time, it's important for the user to get feedback on how the job is progressing. A job and each of its tasks have a status, which includes such things as the state of the job or task (e.g., running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a statusmessage or description (which may be set by user code). These statuses change over the course of the job, so how do they get communicated back to the client?When a task is running, it keeps track of its progress, that is, the proportion of the task completed. For map tasks, this is the proportion of the input that has been processed.

For reduce tasks, it's a little more complex, but the system can still estimate the proportion of the reduce input processed. It does this by dividing the total progress into three parts, corresponding to the three phases of the shuffle (see "Shuffle and Sort"). For example, if the task has run the reducer on half its input, then the task's progress is ⅚, since it has completed the copy and sort phases (⅓ each) and is halfway through the reduce phase (⅙).



What Constitutes Progress in MapReduce?

Progress is not always measurable, but nevertheless it tells Hadoop that a task is doing something. For example, a task writing output records is making progress, even though it cannot be expressed as a percentage of the total number that will be written, since the latter figure may not be known, even by the task producing the output.

Progress reporting is important, as it means Hadoop will not fail a task that's making progress. All of the following operations constitute progress:

Reading an input record (in a mapper or reducer)

Writing an output record (in a mapper or reducer)

Setting the status description on a reporter (using Reporter's setStatus() method)

Incrementing a counter (using Reporter's incrCounter() method)

Calling Reporter's progress() method

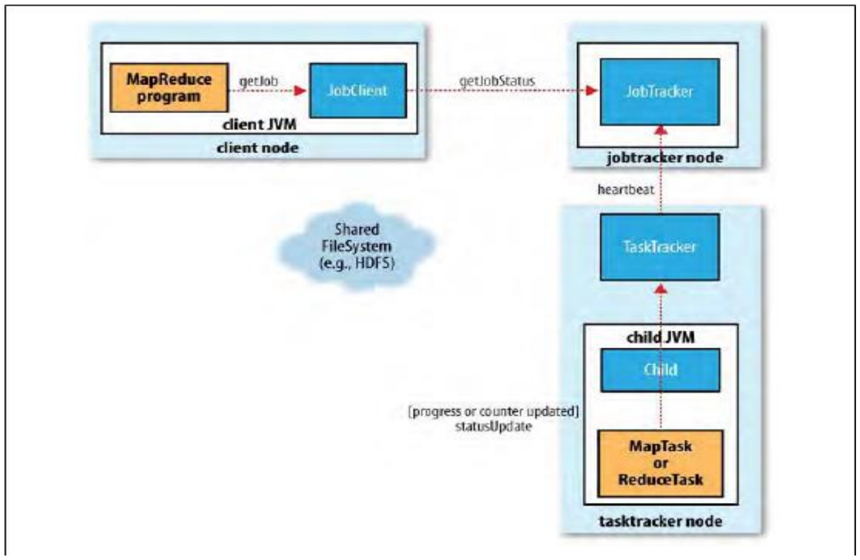
Tasks also have a set of counters that count various events as the task runs (we saw an example in "A test run"), either those built into the framework, such as the number of map output records written, or ones defined by users.

If a task reports progress, it sets a flag to indicate that the status change should be sent to the tasktracker. The flag is checked in a separate thread every three seconds, and if set it notifies the tasktracker of the current task status. Meanwhile, the tasktracker is sending heartbeats to the jobtracker every five seconds (this is a minimum, as the heartbeat interval is actually dependent on the size of the cluster: for larger clusters, the interval is longer), and the status of all the tasks being run by the tasktracker is sent in the call. Counters are sent less frequently than every five seconds, because they can be relatively high-bandwidth.

The jobtracker combines these updates to produce a global view of the status of all the jobs being run and their constituent tasks. Finally, as mentioned earlier, the JobClient receives the latest status by polling the jobtracker every second. Clients can also use JobClient's getJob() method to obtain a RunningJob instance, which contains all of the status information for the job.The method calls are illustrated si given below

Job Completion

When the jobtracker receives a notification that the last task for a job is complete, it changes the status for the job to "successful." Then, when the JobClient polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the runJob() method.



The jobtracker also sends an HTTP job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the job.end.notifica tion.url property.Last, the jobtracker cleans up its working state for the job and instructs tasktrackers to do the same (so intermediate output is deleted, for example)