Big Data Processing Algorithms

VenkataSwamy Martha

Abstract Information has been growing large enough to realize the need to extend traditional algorithms to scale. Since the data cannot fit in memory and is distributed across machines, the algorithms should also comply with the distributed storage. This chapter introduces some of the algorithms to work on such distributed storage and to scale with massive data. The algorithms, called Big Data Processing Algorithms, comprise random walks, distributed hash tables, streaming, bulk synchronous processing (BSP), and MapReduce paradigms. Each of these algorithms is unique in its approach and fits certain problems. The goal of the algorithms is to reduce network communications in the distributed network, minimize the data movements, bring down synchronous delays, and optimize computational resources. Data to be processed where it resides, peer-to-peer-based network communications, computational and aggregation components for synchronization are some of the techniques being used in these algorithms to achieve the goals. MapReduce has been adopted in Big Data problems widely. This chapter demonstrates how MapReduce enables analytics to process massive data with ease. This chapter also provides example applications and codebase for readers to start hands-on with the algorithms.

Keywords Distributed algorithms · Big data algorithms · MapReduce paradigm · Mapper · Reducer · Apache Hadoop · Job tracker · Task tracker · Name node · DataNode · YARN · InputReader · OutputWriter · Multi-outputs · Hadoop example

1 Introduction

Information has been growing at a faster rate, and computing industry has been finding ways to store such massive data. Managing and processing the data from such distributed storage is complex than just archiving for future use. Because the

V. Martha (✉)
@WalmartLabs, Sunnyvale, CA, USA
e-mail: vmartha@walmartlabs.com

H. Mohanty et al. (eds.), *Big Data*, Studies in Big Data 11,
DOI 10.1007/978-81-322-2494-5_3

data carries very valuable knowledge embedded in it and needs to process the data to mine the insights.

1.1 An Example

Here is one example to demonstrate the importance of efficient algorithms to process Big Data. Consider a firm with, say 1000, geo-separated office locations with a million employees altogether and each location collects the number of hours each employee worked everyday at the location. Accounting department of the firm calculates salary of each employee at the end of each pay cycle. One can use a spreadsheet software suite such as Microsoft Excel to do the job here. The software has its limitations on number of rows it can have in a file. It is also possible that a relational database system serves the platform for these computations. It will be a group by operation on 1,000,000 records (1 million employees) and say each pay cycle is 30 days; therefore, 30 days of records adds up to 30 million records. Group by operation on such a large number of rows takes significant amount of time given a single machine database system. Then, the distributed system's platform comes out to rescue from the computation problem. Distributed algorithms are very essential to benefit salient features of distributed systems, in particular when data is big as in the example we discussed here. The benefits from distributed systems for Big Data accompany few challenges that need attention.

1.2 Challenges with Big Data

As mentioned earlier, information has been growing at rapid pace. Industry has been facing several challenges in benefiting from the vast information [1]. Some of the challenges are listed here.

- Since the information is of giant size, it is very important to identify useful information to infer knowledge from it. Finding right data in the collection of data is possible with domain expertise and business-related requirements.
- Big Data is typically archived as a backup, and industry has been struggling to manage the data properly. The data has to be stored in such a way that it can be processed with minimal effort.
- The data storage systems for Big Data did not attempt to connect the data points. Connecting the data points from several sources can help identify new avenues in the information.
- Big Data technology has not been catching with the evolving data. With the fast grown internet connectivity around the world, almost infinite number of data sources are available to generate information at large scale. There is an immediate need for scalable systems that can store and process the growing data.

Researchers have been attempting to address the above-mentioned challenges by advancing Big Data technology. There were times a single computing machine was used to store and process the data. EDVAC is one of the early computer models proposed by Von Neumann [2]. With the advancement in chip fusion technology, a single machine with multiple processors and multiple cores was introduced. Multi-core system is a machine with a set of processors or cores and with facilities to opt computations in parallel. On the other hand, researchers are also focused on connecting several independent machines to run computations in parallel called a distributed system. A distributed system is typically made from an interconnection network of more than one computer or node.

A single machine is proved to be very effective by the generalized bridging model proposed by Von Neumann. Similarly, parallel programming can be effective when a parallel machine is designed as abstract and handy as the Von Neumann sequential machine model. There are two types of parallel systems: One is multi-core system and other is distributed system. With the advancement in distributed systems and multi-core systems, debate started on whether to adopt multi-core system or distributed system.

1.3 Multi-core Versus Distributed Systems

Both multi-core and distributed systems are designed to run computations in parallel. Though their objective is same, there is a clear distinction between multi-core and distributed computing systems that makes them distinguished in their space. In brief, multi-core computing systems are tightly coupled to facilitate shared space to enable communications, whereas distributed systems are loosely coupled that interact over various channels such as MPI and sockets. A distributed system is assembled out of autonomous computers that communicate over network. On the other hand, a multi-core system is made of a set of processors that have direct access to some shared memory. Both multi-core and distributed systems have advantages and disadvantages which are discussed extensively in [3], and the summary of the discussion is tabulated in Table 1.

Given scalable solutions mandated by Big Data problems, industry is inching toward distributed systems for Big Data processing. Moreover, Big Data cannot fit in a memory to be shared among processes, thus to stamp out multi-core system for Big Data processing.

1.4 Distributed Algorithms

Distributed algorithms are developed to perform computation in distributed systems. The algorithms take benefits from multiprocessors in distributed systems and manage computation, communication, and synchronization. There have been

Table 1 Multi-core versus distributed systems [3]

Criteria	Multi-core system	Distributed system
Resources	Dedicated	Utilize idle resources
Resource management	By application	By the system
User per node ratio	Low	High
Processes per node	Few	Many
Node homogeneity	Homogeneous	Heterogeneous
Configuration	Static	Dynamic
Communication paradigm	Message passing	RPC
IPC performance	High	Low
Scalability	Less likely	Yes
Direct access to shared memory	Yes	Now

several attempts to optimize distributed algorithms in generic model, and the following discussion deals with the algorithms.

1.4.1 Random Walks

Let $G = (\vartheta, \varepsilon)$ be an undirected graph representing a distributed system, made of set of nodes ϑ that are connected by links ε. "A random walk is a stochastic process of constructing a course of vertices ϑ visited by a token begins from a vertex i and makes a stopover at other vertices based on the following transition rules: A token reaches a vertex "i" at a time "$t + 1$" from one of the neighbors of the vertex "i" being the token at time "t," and the hop is determined by specific constraints of the algorithm" [4]. Original random walk is designed to address wide range of problems in mobile and sensor networks. The paradigm later adopted for distributed algorithms. A token visits the computing boxes in distributed systems to trigger computations. The token visit is determined by random walk algorithm. A walk of certain length is performed by electing an arbitrary neighbor in every step. The optimization solutions of random walk algorithm attempt to complete the walk in significantly less number of steps. The cover time of the network measures the number of steps needed to all the nodes from a node in the network, in other words steps needed to form a spanning tree of the network. The random walk path is also used to sync the data between nodes on the results from each step.

1.4.2 Distributed Hash Tables

Distributed hash tables (DHTs), in general, are used to perform lookup service in a distributed system [5]. Given that data is distributed among a set of computing/storage machines and each machine is responsible for a slice of information associated with a key. The key to data association is determined by a common hash table. The hash table is pre-distributed for all the machines in the cluster so that

every machine knows where a certain information resides. Looking up a key gives you a node identification metadata that holds the data chunk. Adopting the DHT concept onto computing model, a distributed hash table can be leveraged to find a set of computation machines to perform a task in a distributed system. Computing optimization can be achieved by optimizing the hash table/function for uniform distribution of computing nodes among all possible keys.

1.4.3 Bulk Synchronous Parallel (BSP)

Bulk Synchronous Parallel (BSP) model runs an abstract computer, called BSP computer. The BSP computer is compiled of a set of processors connected by a communication network. Each processor is facilitated with a local memory needed for the processor. In BSP, a processor, means a computing device, could be several cores of CPUs, that is capable of executing a task [6]. BSP algorithm is a sequence of super steps, and each super step consists of an input phase, computing phase, and output phase. The computing phase processes the data sent from previous super step and generates appropriate data as output to be received by the processors in the next step. The processors in a super step run on data received from previous super step, and the processors are asynchronous within a super step. The processors are synchronized after every super step. The communication channels are used to synchronize the process. Direct communication is possible between each processor and every other processor, given absolute authority on distributing the data among processors in the super step. BSP does not support shared memory or broadcasting. BSP is denoted by

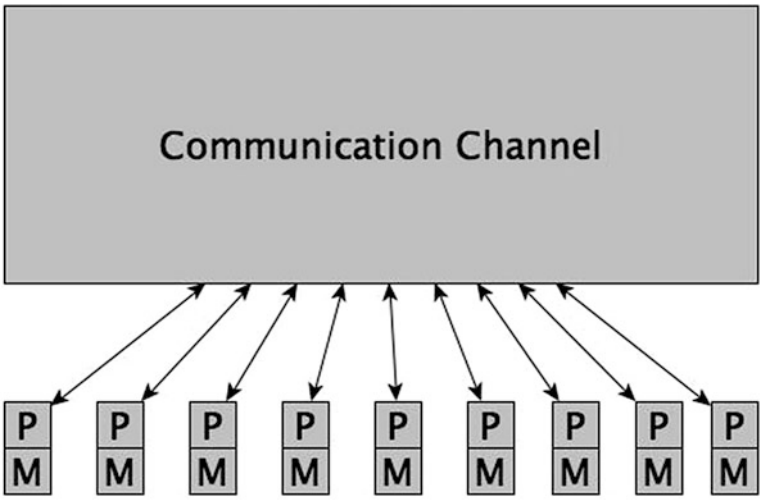

Fig. 1 BSP architecture

(p,g,l) where 'p' is the number of processors, 'g' is communication throughput ratio, and 'l' is the communication latency. The computation cost of BSP algorithm for S steps is $W + H.g + S.l.$ where W is local computation volume $= \sum_{S}^{s=1} w_s$, H is communication cost $= \sum_{S}^{s=1} h_s$, and S is synchronization cost. The BSP algorithm should be designed to optimize computation cost, communication cost, and synchronization cost [7]. The architecture diagram is presented in Fig. 1.

2 MapReduce

2.1 MapReduce Paradigm

Big Data problems typically bid definitive approaches and could sometimes follow non-conventional computing archetypes. All of the approaches have been discussed in the computer science literature for Big Data for decades which follow some kind of out of the box techniques, and MapReduce is certainly not the foremost to drive in this direction. The MapReduce is successful in fusing the several non-conventional computing models together to perform computations on a grand, unimaginable scale. The capability of its design made the MapReduce a synonym for Big Data.

MapReduce paradigm calcifies a distributed system to play authority in processing Big Data with ease. MapReduce, unlike traditional distribution systems, regulates computations on Big Data with least effort. MapReduce exposes specific functions for a developer to implement distributed application and hides internal hardware details. By this, developers can raise their productivity by focusing resources on application without worrying about organizing the tasks and synchronization of tasks.

MapReduce claimed humongous attention from research community as it was with Von Neumann's computing model. Von Neumann proposed 'a model as a bridging model, a conceptual bridge between the physical implementation of a machine and the software that is executed on that machine' [2] for single process machine, long ago. Von Neumann's model served one of the root pillars for computer science for over half a century. Likewise, MapReduce is a bridge to connect distributed system's platform and distributed applications through a design functional patterns in computation. The applications do not need to know the implementation of distributed system such as hardware, operating system, and resource controls.

The engineers at Google first coined the term Map and Reduce as an exclusive functions that are to be called in a specific order. The main idea of MapReduce comes from functional programming languages. There are two primary functions, Map and Reduce. A map function upon receiving a pair of data elements applies its designated instructions. The function Reduce, given a set of data elements, performs its programmed operations, typically aggregations. These two functions, performed

once on a data chunk of input data in a synchronous order coordinated by a synchronous phase called shuffling and sorting, form the basis of MapReduce [8].

There is no formal definition for the MapReduce model. Based on the Hadoop implementation, we can define it as a 'distributed merge-sort engine.' In MapReduce, there are two user-defined functions to process given data. The two functions are Map and Reduce. Given that data is turned into key/value pairs and each map gets a key/value pair. Data is processed in MapReduce as follows:

- Map: The map function takes given appointed key/value pair say (K1, V1) to process accordingly and returns a sequence of key/value pairs say (K2, V2).
- Partition: Among the output key/value pairs from all map functions, all the associated values corresponding to the same key, e.g., K2, are grouped to constitute a list of values. The key and the list of values are then passed to a reduce function.
- Reduce: The reduce function operates on all the values for a given key, e.g. (K2, {V2,…}), to return final result as a sequence of key/value pairs, e.g. (K3, V3).

The reduce functions do not start running until all the map functions have completed the processing of given data. A map or reduce function is independent of others and has no shared memory policy. The map and reduce functions are performed as tasks by Task trackers, also called slaves, and the tasks are controlled by job tracker, or master. The architectural view of MapReduce-based systems is shown in Fig. 2.

Designing an algorithm for MapReduce requires morphing a problem into a distributed sorting problem and fitting an algorithm into the user-defined functions described above. The algorithm should be divided into asynchronous independent smaller tasks.

The pseudo-snippet of MapReduce paradigm can be written as following:

```
Take Input File
Split the input file into List<InputSplit>
For each split S in <InputSplit>
      Convert the content of S to List<key,Value>
      For each Key, Value in List<key, value>
            Intermediate<Key',Value'> <- Map(Key, Value)
      Intermediate<Key', List<Value'>> <-
          Shuffle(Intermediate<Key',Value'>)
      For each Key', List<Value'> in Intermediate<Key',
          List<Value'>>
            Output<Key'',Value''> <- Reduce(Key', List<Value'>)
      OutputWriter(Output<Key'', Value''>)
```

Various implementations of MapReduce are available and Hadoop is one among them. Hadoop not only provides to write MapReduce algorithm but also provides supplementary features that help MapReduce algorithm benefit. The following discussion illustrates the Hadoop tools.

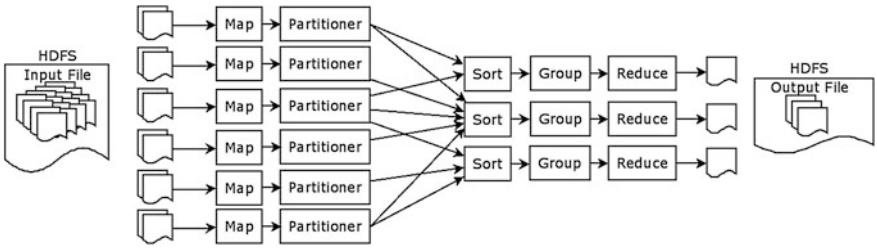

Fig. 2 MapReduce-based systems architecture

2.2 Introduction to Hadoop Tools

Though MapReduce was designed at Google, it has been evolved over time with open-source community. Apache developed several advanced features around the MapReduce technology and released to community to name 'Hadoop.' Sometimes, the words 'Hadoop' and 'MapReduce' are used interchangeably. To draw clear distinction between the two, Hadoop is one of the platforms to implement algorithms of MapReduce paradigm. Hadoop is a collection of several tools closely knit together to make it nearly complete venue to pursue MapReduce. We discuss some of the tools here that are required to develop MapReduce algorithms.

2.2.1 HDFS

MapReduce framework highly depends on the file system upon which the input and output data sits on. To enable various beneficial features, Hadoop constitutes a file system called Hadoop Distributed File System (HDFS) [9]. Similar to many distributed file systems out there, HDFS is also shaped by adopting most of the fundamental concepts from master/slave architecture. HDFS is hoisted from scratch to support very large data. The HDFS system, in succinct terms, constitutes a central server and a set of machines where the data is stored. The central server called Name node (NN) stores metadata while a set of machines are labeled the data nodes (DN) that administer the data comes in. Clients communicate with HDFS Name node to store files. HDFS Name node splits each file into one or more blocks, and then, the blocks are stored in data nodes. HDFS assumes that each file is a sequence of blocks where a block is a sequence of data elements; all blocks except the last one in a file are kept at a fixed size. The size of the blocks in a file is configurable and typically set at 64 MB. If content is less than the specified block size, the rest of the block is left vacant which happens frequently for last block of a file. If the content of the file is less than the block size, the space in lonely block for the file is left empty after filling the content driving the wastage of disk. For the reason, HDFS is unfit for small files, and the more smaller files, the faster they enervates the HDFS. The blocks of a file, are then, are replicated over one or more

Name nodes in the HDFS. The replication facilitates the HDFS to furnish fault tolerance amenity. Analogous to traditional file system, HDFS follows traditional file system in naming files in the system by adopting the tried-and-true directory hierarchy approach. File system operations such as create, rename, and remove files are possible in HDFS as in traditional file systems with an exception for edit operation. One of the most discussed limitations of HDFS from other file systems is that HDFS does not comply with file editing. HDFS is architected to enable the following features to name a few.

1. Big Data: HDFS is designed to support large files and to deal with applications that handle very large datasets. The large files are supposed to split into blocks. By this, HDFS is not recommended for files with small size in particular when one or more files are smaller than a block.
2. High availability (Replication): Applications that read data from HDFS need streaming access to their datasets. High throughput and availability are core features of HDFS. The features are achieved by data replication. The replication factor is configurable per file. An application can specify the number of replicas of a file. Each block of a file is replicated on the number of data nodes as the replication factor. The replication process led HDFS for slower writes but benefits with faster reads. When a file is opened for reading, for each block of the file, one of the data nodes that contain a replica of the block is constituted to serve the data read request. Since there is more than one data node available to serve the data read request, the reads are faster to achieve high availability. To minimize read latency, HDFS tries to satisfy a read request from a replica that is closest to the reader. Once the blocks are written and replicated across data nodes, the blocks cannot be modified leading the HDFS to be termed as write-once-read-many file system.
3. Robustness: HDFS can recover if there is data loss because of data node failure, or disk failure. When data on a data node is corrupted or not available, HDFS administrates the data nodes that replicate the data on the failed node to other nodes to satisfy the replication factor configuration. The re-replication is possible, and each data block is stored on more than one data node. HDFS listens to heartbeats and audits the data blocks for their availability periodically. HDFS constantly regulates all data blocks in the cluster to fulfill the system's configuration and triggers replication process whenever necessary.

HDFS is built on top of two primary components that are responsible in harvesting the above-discussed features. The following discussion presents specifics of the components.

- *Name node*. A machine that manages the HDFS is called 'Name node.' Name node does not store the data itself but administers where to store the data. The Name node executes file system namespace operations such as opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The responsibilities of Name node are given as follows:

  - It maintains the directory tree of the files in the file system.
  - It audits the physical location of the data.

- *Data Nodes* There are a set of DataNodes, one per node in the cluster, to store files. The Name node performs a split on a file into several blocks and directs the file system client to send the file contents in the given split sizes to data nodes. The data nodes store the file splits in blocks as per instructions from Name node. The DataNodes also perform block creation, deletion, and replication upon instruction from the Name node. When there is a request for read or write from the file system clients, the data nodes serve the request.

The data flow in a write operation is presented in Fig. 3. HDFS client who wants to write some data file onto HDFS contacts Name node for a list of data nodes that the client can connect to and write the contents of the file. The Name node updates its metadata of the request and responds with a block id and a data node details. The clients upload the content of the file to the data node, while data node copies the received content into the block specified by the Name node. Name node then finds another data node to comply with the replication factor. The Name node instructs the data node to copy the bock to other data node. The replication continues among the data nodes until the system satisfies the replication factor.

The similar and reverse approach is involved in reading the contents of a file from HDFS. The data flow is presented in Fig. 4. Client node, who wants to read a file, contacts Name node for a list of data nodes where the contents of the file reside. Name node responds with a list of data nodes, three data nodes when replication factor is three. Client node chooses one from the list received and contacts the data nodes to get served by the requested data.

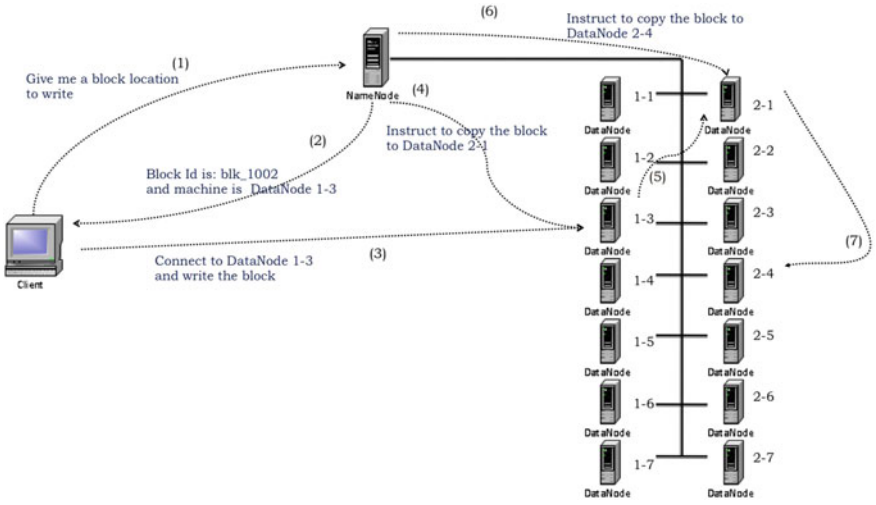

Fig. 3 Data flow diagram in a write operation in HDFS

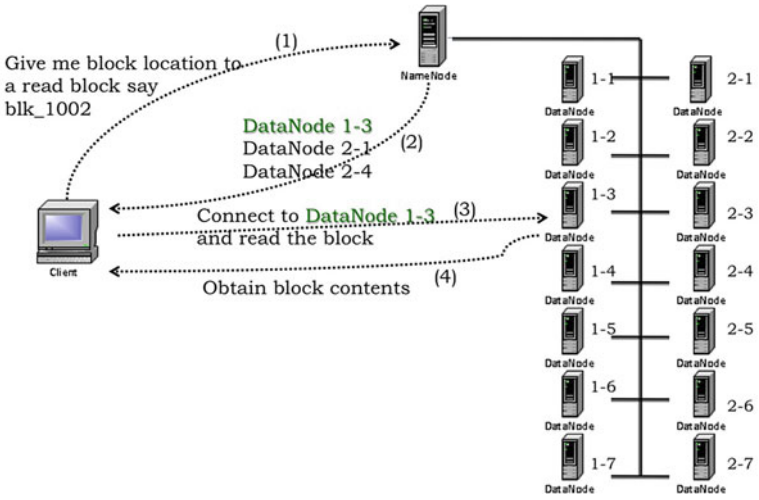

Fig. 4 Data flow diagram in a read operation in HDFS

HDFS is highly available and fault tolerant as client has an option to choose other data node when a data node is down while reading a file. When the client is one of the data nodes, it reads from local file system if the block resides in itself. Such scenarios commonly encounter when an application run in the data nodes. By this, the data does move to application, but application moves to data. Taking this advantage, MapReduce runs its tasks on data nodes. Hadoop implemented a set of tools to achieve MapReduce computing with the advantages taken from HDFS.

2.3 MapReduce Computing Platform

The Hadoop's MapReduce computing platform is depicted in Fig. 5. Hadoop's MapReduce computing platform constitutes two components: One is JobTracker

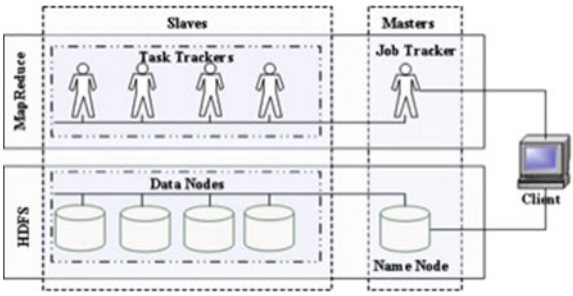

Fig. 5 Hadoop's MapReduce computing platform [15]

acts as a master in the Hadoop cluster while the other is called Task trackers. The Task trackers can be treated as workers.

A typical (simplified) activity flow in Hadoop is as follows [10].

1. A server, called Job Tracker, accepts jobs (MapReduce programs) from clients. A job is associated with a mapper method, a reducer method, and a set of input files and output files.
2. The Job Tracker contacts Name node to get the location of the input files. The Job Tracker applies appropriate scheduling algorithm to assign tasks to a set of computing nodes, called Task trackers. The scheduling algorithm takes data locality into account to optimize data movements.
3. The Job Tracker distributes the mapper and reducer methods among the scheduled Task trackers. In addition to the mapper and reducer methods, the job tracker also distributes the job configuration so that the mapper and reducer methods run based on the provided configuration.
4. The Task tracker performs the assigned mapper task. The Task tracker reads input from data nodes and applies given method on the input.
5. The map task creates and writes to an intermediate key-value pairs to a file on the local file system of the Task tracker.
6. Partitioner reads the results from the map task and finds appropriate Task tracker to run reduce task. The intermediate results emitted by map tasks are then propagated to reduce tasks.
7. The Task trackers that are scheduled to run reduce tasks apply operations programmed in reduce function on the data elements streamed from map tasks and emit a key-value pairs as output. The output data elements are then written to HDFS.

The control flow discussed here is evident enough to assert that the tasks come down to data location to perform designated operations. If there are no computing resources available at the machine where the data located, the computation is carried out in a nearest machine to the data. Such behavior is called data locality, in MapReduce context. Data locality improves the distributed computations by reducing the data movements in the network within the cluster.

2.3.1 Job Tracker

Job Tracker is a server that has the implementation for necessary user interfaces needed to submit and run a map reduce job. Once a map reduce job is submitted to Hadoop cluster, the JobTracker of the Hadoop cluster engineers a scheme to run the submitted job. The scheme involves identifying Task trackers in the cluster to perform map operations, triggering the mappers on Task trackers, monitoring the task while running, etc. Job Tracker runs in listening mode to take requests from clients. When a client submits a job, the job tracker communicates with Name node to obtain a list of machines that carry the input data for the job. The list is used in an

optimization algorithm, and the job tracker comes up with an optimized scheme to run the job on Task trackers. As mentioned earlier, the scheme attempts to reduce network bandwidth utilization within the cluster by adopting data locality feature. By data locality, the preference to run a task to run on a data chunk is (1) the machine where the data is located, (2) the rack where the data is located, and (3) a computing machine in the cluster. It could not always possible to find a node for a map task to run on local data; then, it is highly possible to find a node in the same rack. If there is no node available to take the task, then whatsoever machine in the cluster can perform the task, though not optimal but can let the job advance. One of the reasons a Task tracker not available to take a task is that the Task tracker could have been running the tasks up to its maximum capacity. Whenever the job tracker identifies a Task tracker to run a task, it monitors the task until it terminates. The job tracker finds another optimal Task tracker in case if a task failed on a Task tracker. By restarting a task on another Task tracker, the job tracker ensures that the job does not terminate if there is a task failed once, but attempts to run several times. At the same time, a job cannot be turned into successful states unless all the tasks of the job complete without errors.

The responsibilities of Job Tracker can be summarized as follows:

- Manage Task trackers and its resources as jobs being submitted.
- Schedule tasks among available resources.
- Monitor the tasks of jobs, and restart the failed tasks for configured number of attempts.

Job tracker is the heart of MapReduce computing platform as it is the entry and exit points for clients to submit a job. On the other hand, there is only one Job Tracker process runs on a Hadoop cluster. The JobTracker poses single point of failure for the Hadoop MapReduce service because there is no alternate when the only job tracker shuts down. When the job tracker falls down, all the jobs running on the cluster are deemed to be halted for client.

2.3.2 Task Trackers

Task tracker is a daemon runs on computing nodes of Hadoop cluster. The Task tracker receives instructions to perform map and/or reduce tasks from the Job Tracker. The Task tracker posts the available resources in the node and a heartbeat message at every specific interval of time to the job tracker. The job tracker performs bookkeeping of the Task tracker and corresponding resource and exploits the information in job scheduling. The Task tracker performs the assigned tasks (map, reduce) on given data. The job tracker tracks the Task trackers for the tasks that are running and instructs the Task trackers to where to send the output data.

A Task tracker performs a mapper method or a reduce method. The methods are illustrated in detail as follows.

- *Mapper* Mapper is a function that reads input files of given job in terms of <*Key, Value*> pairs and emits some other <*Key, Value*> pairs. The emitted key-value pairs are to be consumed by reducers and called intermediate key-value pairs. The skeleton of the mapper implementation is presented here.

```
public class MyMapper
   extends Mapper<Object, Object, Object, Object>{

   // Class members definitions goes here

   public void map(Object key, Object value, Context context)
       throws IOException, InterruptedException{
     // process key value pair
     context.write(OutputKey, OutputValue);
    }
}
```

A Mapper, say MyMapper, can be implemented by extending the Mapper interface which is in turn extends MapReduceBase class. A custom mapper is realized by overriding 'map' function in the MyMapper class. The map function takes input key and value as objects and a context that represents the context of the task. The context is then used to read task-specific variable and pass on counters to client.

- *Reducer* The key-value pairs emitted by all mappers are shuffled to accumulate all the intermediate records with a key will see same reducer. The reducer receives set of values for a given key and iterates over all the values for aggregation.

Here is the Reducer implementation, bare-bones code.

```
public class MyReducer
    extends Reducer<Object, Object, Object, Object>{

   // Class members definitions goes here

  public void reduce(Object key, Iterable<Object> values,
                Context context) throws IOException,
                    InterruptedException{
     // process values list for given key

     context.write(OutputKey, OutputValue);
  }
}
```

As for Mapper, a Reducer is developed by extending Reducer interface which in turn extends MapReduceBase class and a custom reduce function can be written by overriding reduce function in the class. Unlike from Mapper, Reducer's second argument is 'Iterable' which splits the values for the given key upon iteration. Context in reducer is leveraged as in mapper.

2.3.3 YARN

The recent version of Hadoop scrapped Job Tracker and Task trackers for a new job processing framework called YARN [11]. YARN stands for Yet Another Resource Negotiator. YARN constitutes Resource manager and Node Manager.

1. *Resource manager* Resource manager is the daemon which governs all jobs in the system. Resource manager regulates the resources in Task trackers. Resource manager schedules tasks based on the resources available on Task trackers. The Resource manager constitutes two primary tools: Scheduler and Applications Manager.

   - The Scheduler is solely designed to apportion the available resources on computing nodes of the Hadoop cluster. The allocation attempts to satisfy the capacity, queue, SLA, etc. There is no guarantee that the scheduler restarts the failed tasks but attempts to reschedule the task on the same node or other node for several attempts according to the scheduling configuration. A scheduler can employ a sophisticated algorithm to allocate resources among jobs submitted; by default, a resource container administers resources such as CPU, memory, disk, and network, which is the basis for scheduling algorithm. Recent version of YARN release supports memory as resource in scheduling. More such resource information is used more in optimal scheduling. YARN allows to use custom-designed schedulers in addition to the FIFO scheduler. Capacity Scheduler and Fair Scheduler are two such schedulers.
     The Capacity Scheduler is developed targeting a goal to share a large cluster among several organizations with minimum capacity guarantee. It is designed for supporting seamless computing flow in a shared cluster being utilized by more than one customer and to optimize resource utilization. Capacity Scheduler allows to maintain more than one queue, and each queue follows its configured scheduling scheme. Each queue complies with the configuration provided for this queue. Clients need to submit a job to the respective queue. The queue is processed in FIFO strategy. The jobs in a queue can be run on the machines that fall under the queue.
     Other scheduler is Fair Scheduler that the jobs equally receive their quota of resources to make progress in the job. There are pools, and each pool is allocated a portion of resources. When a job is submitted to a pool, all the

jobs in the pool share the resources allocated to the pool. In each pool, there is a fair distribution of resources among the jobs running on the cluster. A job never starves to terminate because it is FIFO and time-out waiting in getting resources. The scheduler optionally supports preemption of jobs in other pools in case to satisfy fair share policy.

- The Applications Manager stays open to receive requests to run jobs. When needed, Applications Manager negotiates with clients when the requested resources are not viable or resource allocations are changed to run the job. SLAs exchanged between Applications Manager and client, and when consensus is reached, the job is put on to execution pool. Scheduler takes the job to schedule it to run on the cluster.

2. *Node Manager* Node Manager does the similar job as Task trackers. Node Manager constitutes resource containers and performs operations such as monitoring resource usage and posting the resource status to the Resource manager/Scheduler in the cluster. Node Manager resides on a computing machine, coordinates the tasks within the machine, and informs Resource manager and Applications Manager on the status of the resources, tasks running on the machine.

Irrespective of YARN or Hadoop's early release, Hadoop's underlying design pattern, and client application development, APIs stay nearly unchanged.

There might be several mappers running as the map tasks get completed. As map tasks complete, the intermediate key-value pairs start reaching corresponding reducers to perform reduce operation. The nodes that run map tasks in a Hadoop cluster begin forwarding the intermediate key-value pairs to reducers based on intermediate key.

2.3.4 Partitioners and Combiners

Partitioner: Partitioning, as mentioned earlier, is the process of determining which reducer to work on which intermediate keys and values. There is a partitioner for each node. The partitioner of each node determines for all of mappers output (key, value) pairs running on the node which reducer will receive them. The intermediate key-value pairs of certain key are destined to one reducer despite the key-value pairs generated from different mappers. The partitioning algorithm should be in such a way that there should never be a need to move data from one node to another node. Default partitioning algorithm is hash algorithm. The key-value pairs of keys having same hash go to one reducer. There is also an option to define customized partitioner for a given job. The partitioner should decide which reducer a key-value pair should send to. The number of partitions the algorithm distributes the intermediate key-values pairs is equal to the number of reducers. The job configuration should specify the number of reducers, in other words the number of partitions.

```
public static class CustomPartitioner extends
    Partitioner<Text, Text> {

  @Override
  public int getPartition(Text key, Text value, int
      numReduceTasks) {
      // Find the right partition for the key
      return partition_number;

  }
}
```

Combiner: The objective of the combiner is to optimize the data movements among nodes. The combiner runs on each mapper and performs same task as reducer so that the data is reduced at the mapper node itself. The combiner step is optional in a job. Combiner receives key-value pairs from the mappers in a node and aggregates using the combiner method passed by the job configuration. With the interception of combiner in the pipeline, the output of the combiner is called intermediate key-value pairs and passed to partitioner for determining the appropriate reducer. To summarize, the combiner is a mini-reduce process which operates only on data generated by one machine.

```
public static class MyCombiner extends
      Reducer<Text, Text, Text, Text> {

  public void reduce(Text key, Iterable<Text> values,
      Context context)

        throws IOException, InterruptedException {
        // Aggregate the values for new key value pairs
        context.write(key, new Text(concatenatedData));
  }
}
```

2.3.5 Input Reader and Output Writers

We have been discussing how a job processes given data. Now let us discuss how a job reads the data and writes the output. Each job is associated with a set of input files and a set of output files. A method to define how to read the input file, to generate key-value pairs, is called input reader. Similarly, there is a method to define what format the output should be written, from key-value pairs emitted from reducers, which is called Output Writer.

Input Reader: Input of a MapReduce job is a complete file or directory. When a map is running the task, it receives only a chunk of data from the complete file or directory. A chunk called input split is passed to a mapper. The size of the input split, a mapper should work on, supposed to be defined by client with domain expertise. Default input split size is 64 MB or the same size of a block in HDFS. An input split could be the combination of one or more or partial files. Job scheduler also follows the same input split concept to achieve data locality feature.

Given that the input file of a job is split into several chunks called InputSplits. Each input split constitutes a sequence of bytes, and the bytes need interpretation. A toolkit to interpret the input split is to be developed for each map reduce. RecordReader serves the purpose that translates a given input split into a series of (key, value) pairs. The key-value pairs are in a consumable format when mappers read them. LineRecordReader is one of the record readers out there, and LineRecordReader assumes each line as a record and turns each line into a key-value pair by splitting the line by a delimiter. TextInputFormat is available by default in Hadoop package and can be used to read text files out of the box in a map reducer job. There is few other complex record reader that is capable of decompressing the input splits, deserializing objects, etc. Clients can develop a custom record reader to parse input file data following a specific format.

Output Writer: As the reduce task runs, they generate final output key-value pairs. Output Writer defines how the final key-value pairs should be written to a file. There is an Output Writer for a reducer task. The Output Writer takes the key-value pairs from the reducer and writes to a file on HDFS. The way the key-value pairs are written is governed by the OutputFormat. Therefore, there is a separate file and a common output directory for each reducer; a Hadoop job generates the number of output files as the number of reducers. The output files generated by reducers usually follow the patterns such as part-nnnnn, where nnnnn is the partition identifier associated with a certain reduce task. The common output directory is defined by the user in job configuration. The default output format is the TextOutputFormat which writes a line with key-value pairs in text format separated by tab for each key-value pair emitted by a reducer. In addition, there are two other output formats that are packaged in Hadoop implementation. The following table (Table 2) summarizes the

Table 2 Standard output format implementation

Output format	Description
TextOutputFormat	Given key-value pairs are written to output file in text format with a tab space as delimiter
SequenceFileOutputFormat	Given key-value pairs are written to output file in binary format. This format is useful for jobs reading in another map reduce job
NullOutputFormat	No output is written for this format

output formats. An user-defined Output Writer also allowed to specify job configuration to generate user-specific output format from final output key pairs.

2.4 Putting All Together

Summing up the artifacts learned so far in this chapter is sufficient enough to develop a Hadoop job and make it ready for submission on a Hadoop cluster. Let us go through the process of implementing a Hadoop job in this section to exercise the practices we learned.

Though there are many problems that can be solved by MapReduce paradigm, a Big Data problem yet simple is taken into consideration here to showcase the benefits of the MapReduce phenomenon. Assume that you are given a task to build a search engine on World Wide Web pages similar to Google or Bing. Disregard worrisome features of a search engine and the search engine returns you a list of pages and likely the places where the given keyword is occurred in the World Wide Web. A list of possible keywords in the World Wide Web alongside with corresponding pages and places the keyword occurred can enable the search possible without walking through all the World Wide Web every time there is a search request. Such list is called an index, widely accepted approach. Preparing an index is an easy task when the documents are handful and as well keywords. The real challenge arises when the same task to be done over a manifold of pages which is the case with World Wide Web. The number of documents to be indexed is at high scale and cannot be done on a single machine. MapReduce paradigm lands into overcome the challenges.

It is a typical routine to solve a bigger problem a piece by piece to come up with a viable solution. Endorsing the tradition, let us find a solution for indexing smaller dataset first followed by extending the solution to make it feasible on Big Data. Since MapReduce is partly a composition of map and reduce functions, it is easy to port the solution on smaller dataset onto MapReduce paradigm. The workout on the smaller dataset runs as in the following discussion.

Each page in World Wide Web is a sequence of text character disregarding the non-printing material of the pages. A page, interchangeably called document here, **D** is nothing but a set of text lines. Each text line is a set of keywords. Therefore, **D** = Set{Lines} where Line = Set{Keywords}. Here, each keyword is associated with a line. Now trace back the keyword position. A keyword position is a pair of document id and line number. For each line in the document, generate a pair of the keywords in the line and corresponding positions. The pairs can be aggregated in a method after reading all the contents in the document. The stitching of the above discussion can be written as an algorithm for a document D and is given as follows.

```
public void indexDocument(Document d){
     Map<Position, String> inputPairs = splitDocument(d);

     for( Entry<Position, String> onePair : inputPairs ){
        intermediatePairs.append(indexLine(onePair.Key(),
            onePair.Value()))
     }

     for( String oneKeyword : intermediatePairs.keySet() ) {
          aggregateIndexes( oneKeyword,
              intermediatePairs.get(oneKeyword) );
     }
}
public Map<Position, Text> splitDocument(Document d){
     String[] lines = split(d, newLineCharacter);
     Map<Position, String> inputPairs = new Map();
     for( int i=0; i<size(lines); i++ ){
          inputPairs.append(newPosition(d,i), lines[i])
      }
      return inputPairs
}
public MultipleMapping<String, Position> indexLine(Position
    pos, Line value) {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         emittingMap.append(word, pos);
      }
}
public List<Pair<String, String>> aggregateIndexes(word,
     Collection<Position> positionList) {
      emittingList.append(new Pair(word,
          StringUtils.join(positionList,',')));
}
```

As pointed out earlier, this algorithm works effectively on one document without encountering performance issues. When we carry forward the same application onto millions of documents, it fails to claim its outcomes as cogently as with one document. Harvesting the knowledge gained in this chapter, let us revamp the above algorithm into a MapReduce program. One can simply map the functions in the above algorithm into functions in MapReduce by knowing the spirit of MapReduce paradigm. Table 3 show cases the linear mapping to demonstrate how easy it is to develop an algorithm in MapReduce paradigm.

Now migrating the methods into MapReduce tools such as Map and Reduce methods, though obvious, is presented here. The source code for the MapReduce algorithm is illustrated in the following order: (1) Necessary classes for the MapReduce tools to function, (2) Mapper method, (3) Reducer method, and (4) InputReader implementations to understand the document content.

Table 3 Typical algorithm to MapReduce

Typical method	Method in MapReduce
splitDocument	InputReader
indexLine	Mapper
aggregateIndexes	Reducer

The mapper and reducer classes need to be supported by several supplementary classes, in particular Input Reader and Position. The Position class is defined to standardize on the offset/location of a text in a file. The position is a combination of 'File Name' and 'Offset in the File.' The position class has to be writable to be passed across mappers and reducers. The position class can be implemented as follows.

```
class Position implements Writable {
   String filename;
   Long offset;

   public Position() {
   }

   Position(String filename, long pos) {
      this.filename = filename;
      this.offset = pos;
   }

   @Override
   public void write(DataOutput d) throws IOException {
      d.writeChars(filename + "\\n");
      d.writeLong(offset);
   }

   @Override
   public void readFields(DataInput di) throws IOException {
      filename = di.readLine();
      offset = di.readLong();
   }

   @Override

   public String toString() {
      return filename + ":" + offset;
   }
}
```

A mapper takes a line in the given input with its corresponding offset to generate key-value pairs of tokens and the offset. The mapper class implementation is as follows.

```
public class DI_Mapper extends Mapper<Position, Text, Text,
    Position> {
   Text word = new Text();
   @Override
   public void map(Position key, Text value, Context context)
       throws IOException, InterruptedException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         context.write(word, key);
      }
   }
}
```

A reducer method takes all offsets emitted by more than one mapper for a given token, and to split aggregate of offsets in a string format. All the offsets of a token are distributed by partitioner. The implementation of reducer class is as follows.

```
public class DI_Reducer extends Reducer<Text, Position, Text,
    Text> {

   @Override
   public void reduce(Text key, Iterable<Position> values,
       Context context) throws IOException, InterruptedException
       {
      String positionsString = new String();
      for (Position onePosition : values) {
         positionsString += onePosition.toString() + ",";
      }
      context.write(key, new Text(positionsString));
   }
}
```

The input to the MapReduce job is a set of text files, and the files should be read to transform the content to appropriate key-value pairs for mapper class. The input reader class does the service for the purpose. The record reader also called input reader takes the input files and generates key-value pairs that would be taken by mappers. The record reader implementation for the mapper here is presented in the following code.

```
public class DI_InputReader extends RecordReader<Position, Text>
    {

   private long start;
   private long pos;
   private long end;
   private LineReader in;
   private int maxLineLength;
   private String filename;
   private Position key;
   private Text value = new Text();

   @Override
   public void initialize(InputSplit genericSplit,
       TaskAttemptContext context) throws IOException,
       InterruptedException {
      FileSplit split = (FileSplit) genericSplit;
      filename = split.getPath().toString();

      Configuration job = context.getConfiguration();
      this.maxLineLength =
          job.getInt("mapred.linerecordreader.maxlength",
            Integer.MAX_VALUE);

      start = split.getStart();
      end = start + split.getLength();

      final Path file = split.getPath();
      FileSystem fs = file.getFileSystem(job);
      FSDataInputStream fileIn = fs.open(split.getPath());

      boolean skipFirstLine = false;
      if (start != 0) {
         skipFirstLine = true;
         --start;
         fileIn.seek(start);
      }

      in = new LineReader(fileIn, job);

      if (skipFirstLine) {
         Text dummy = new Text();
         start += in.readLine(dummy, 0,
               (int) Math.min(
                     (long) Integer.MAX_VALUE,
                     end - start));
      }
      this.pos = start;
   }

   @Override
   public boolean nextKeyValue() throws IOException,
       InterruptedException {
```

```
      // Filename is the key
      key = new Position(filename, pos);

      int newSize = 0;
      while (pos < end) {
         newSize = in.readLine(value, maxLineLength,
               Math.max((int) Math.min(
                           Integer.MAX_VALUE, end - pos),
                     maxLineLength));
         if (newSize == 0)
            break;
         pos += newSize;
         if (newSize < maxLineLength)
            break;
      }
      if (newSize == 0) {
         key = null;
         value = null;
         return false;
      } else {
         return true;
      }
   }

   @Override
   public Position getCurrentKey() throws IOException,
       InterruptedException {
      return key;
   }

   @Override
   public Text getCurrentValue() throws IOException,
       InterruptedException {
      return value;
   }

   @Override
   public float getProgress() throws IOException,
       InterruptedException {
      return start==end?0.0f:Math.min(1.0f, (pos - start) /
          (float) (end - start));
   }

   @Override
   public void close() throws IOException {
      if (in != null) {
         in.close();
      }
   }
}
```

Now, we have every tool needed to run a map reduce job. The map reduce job can be triggered from a method by importing the above-stated implementation. One such method to submit a map reduce job is called Driver. The driver method is named to submit map reduce with given map and reduce implementation to run on specified input files to generate designated output files. Hadoop provides the 'Tool' class to develop such driver classes. The following class implements the Tool class to submit a map reduce job with the given configuration. The main method here calls a method named 'run' with the command line arguments. The command line arguments are input directory that constitutes input files and output directory where the output files to be stored. The code for the driver class is as follows.

```
public class DocumentIndexing extends Configured implements Tool
    {

   public static void main(String[] args) throws Exception {
      int res = ToolRunner.run(new Configuration(), new
          DocumentIndexing(), args);
      System.exit(res);
   }

   @Override
   public int run(String[] args) throws Exception {

      // When implementing tool
      Configuration conf = this.getConf();

      Job job = new Job(conf);
      job.setJarByClass(DocumentIndexing.class);
      job.setJobName("DocumnetIndexing");

      // Assuming key and value of both mapper and reducer are
          text format
      job.setMapOutputKeyClass(Text.class);
      job.setMapOutputValueClass(Position.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(Text.class);

      job.setMapperClass(DI_Mapper.class);
      job.setReducerClass(DI_Reducer.class);

      // Input
      FileInputFormat.addInputPath(job, new Path(args[0]));
      job.setInputFormatClass(DI_InputFormat.class);
      // Output
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
      job.setOutputFormatClass(TextOutputFormat.class);

      // Execute job and return status
      return job.waitForCompletion(true) ? 0 : 1;
   }
}
```

To run the above MapReduce application, one needs to compile the Java code and to compress into a jar package. The built jar can be used to submit the job with input and output paths as arguments. The command to submit a job is as follows.

```
Shell> hadoop jar DocumentIndexing-exe-jar-with-dependencies.jar
    documentindexing.DocumentIndexing -D mapred.reduce.tasks=5
    inputPath outputPath
```

The skills learned from this exercise are enough to model most of the other algorithms in MapReduce paradigm. In addition to the basic artifacts in MapReduce paradigm discussed so far, there are several other advanced concepts that make the MapReduce suitable for many domains. Some of the techniques are discussed in the following section.

3 Advanced MapReduce Techniques

3.1 Hadoop Streaming

Hadoop streaming allows one to create and run Map/Reduce jobs with any executable or script as a mapper and/or a reducer. Standard input and standard output (stdin, stdout) serve the channels to pass data among mappers and reducers. Let us discuss how it works with mapper, and then reducer will be in the similar approach. Mapper application receives input key-value pairs as input from standard input (stdin), and the mapper application has to parse the line came from stdin to extract key-value pairs. Upon completion of map instructions, the output key-value pairs can be emitted by writing to standard output (stdout).

A streaming application splits each line of given text file at the first tab character to recover key and value from the line. A mapper or reducer in streaming applications writes their output to stdout in the same format: key value.

As a mapper task in a streaming application runs, Hadoop converts the content of the given input files into lines and feeds the lines to the stdin of the given mapper process. The mapper program should split the input on the first tab character on each given feed of line to recover key-value pair. Mapper runs the programmed instructions on the given key-value pair and writes the output key-value pair to stdout separated by tab character.

The output of the mapper task is partitioned by a partitioner to feed to reducers. The inputs of a reducer are sorted so that while each line contains only a single (key, value) pair, all the values for the same key are adjacent to one another. Reducer program reads the given lines and recovers list of values for each key. Reducer generates output key-value pairs that are written to output file in HDFS.

Provided a Mapper and Reducer can handle given input in the text format, the mapper or reducer program can be written in any language as long as the nodes in the Hadoop cluster know how to interpret the language.

The following command is a typical way to run a Hadoop streaming job. The command triggers a map reduce job to use 'IdentityMapper' for mapper and shell script-based word count application as reducer.

```
hadoop jar $HADOOP_HOME/hadoop-streaming.jar
   -input myInputDirs
   -output myOutputDir
   -mapper org.apache.hadoop.mapred.lib.IdentityMapper
   -reducer /bin/wc
```

3.2 Distributed Cache

There are plenty of cases where every mapper needs some data in addition to the data to be processed say a configuration file or dictionary file. In the example MapReduce program we discussed, we extended the application where the tokenizer needs to read stop words list to ignore stop words from input text. The stop words file is to be available to every mapper that uses a tokenizer before the mapper start processing given input key-value pairs. Hadoop's distributed cache addresses the problem by providing tools to copy the given data to every node that runs mapper or reducer. The data is only copied once per job and the ability to cache archives which are unarchived on the slaves. The data in the form of files which could be text, archives, jars, etc. Hadoop assumes that the files are in HDFS. The data copying occurs at the time of job creation, and the framework makes the cached files available to the cluster nodes at their computational time.

3.3 Multiple Outputs

As discussed earlier, there are number of output files in a directory for a MapReduce job as the number of reducers in the job. Each reducer writes to a file. Hadoop allows a reducer in a job to generate more than one file and more than one output format. The job that needs to generate multiple outputs takes the output names as configuration. Reducer writes the output key-value pairs to appropriate output named file. Each output, or named output, may be configured with its own OutputFormat, with its own key class and with its own value class.

3.4 Iterative MapReduce

It is very likely that an application cannot be implemented as one single map reduce job. It is also distinctly possible that same map reduce job needs to be repeated

several times to generate expected output. Given the need for more such repetitive scenarios, there have been several implementations to support iterative calls to a map reduce job. iMapReduce is one of them and is discussed here.

Iterative iMapReduce runs mapper and reducer methods iteratively. The iterative calls to mapper and reducer methods improve performance by reducing the overhead of creating jobs repeatedly, eliminating the data movements, and allowing asynchronous execution of mappers [12]. Even though it is not true that every iterative algorithm can benefit from iMapReduce, many machine learning algorithms are quite suitable in iMapReduce.

Following section introduces some of the machine learning algorithms in MapReduce paradigm.

4 Machine Learning with MapReduce

The MapReduce architecture in Hadoop does not support iterative mapper and reducers directly, but there are several implementations that extended Hadoop to support iterative map reduce jobs. Machine learning algorithms take advantage of the iterative map reduce tool to train features on Big Data [13]. Clustering is one of classical machine learning algorithms and is discussed here to illustrate the sense of machine learning in Hadoop.

4.1 Clustering

A cluster is said to be a group of observations with similar interests. Clustering algorithm attempts to identify the groups in the given observations data. Two data points are grouped together if they have similar interests. The interests could be distance, concept, pattern, homogeneous property, etc. The clustering algorithms optimize clustering quality. There are no generic criteria to measure quality of clustering, but different problems follow different methods. K-means clustering is one of the clustering algorithms which optimize the clustering quality by minimizing the distance among observations within clusters.

4.1.1 K-Means Clustering

K-means clustering algorithm takes a set of vectors which we call training data. There are k-vectors to start with as k-cluster representatives. The algorithm distributes the vectors to k-clusters where the clusters are vectors too. The distribution tries to allocate each vector to its nearest cluster. The distribution process repeats iteratively; after every iteration, the new cluster representative is computed. The iteration stops when the cluster representative converges. There is no guarantee that it converges. The iterations stop if it does not converge for configured limit.

Since the same procedure is iteratively performed in the algorithm, now we can do extend the algorithm onto MapReduce. The MapReduce version of the algorithm also iterative so uses iMapReduce. In each iteration, a map reduce job allocates given observations to clusters from previous iteration and computes new cluster representatives. The mapper of the map reduce job distributes the observations into clusters, while the reducer computes new cluster representatives and emits the new cluster. The output clusters from reducer will be consumed by the mappers in the next iteration [14].

The pseudocode of the application is presented here.

```
public void map(Text Key, Text Value){
    previousClusterCenters =
        readPreviousIterationClusterCenters(); // From
        DistributedCache
    int minDistance = 10000; // some maximum value
    nearestCenter = previousClusterCenters[0];

    for (oneCenter : previousClusterCenters){
       distanceFromThisCenter = computeDistance( Value,
           oneCenter );
       if( minDistance > distanceFromThisCenter ){
            minDistance = distanceFromThisCenter;
            nearestCenter =oneCenter
       }
    }
    emit( nearestCenter, Value )
}

public void reduce( Text Key, Iterator<Text> Values ) {
       newCenter = computeCenter( Values );
       if( newCenter != Key )
              IncrementCounterBy1( NumberOfClustersChanged )
       emit ( newCenter, NullWritable );
}

public static void main(){
    iteration = 0
    uploadInitialCentersToHDFS();
    while true {
      ConfigureCentersFileToDistributedCache();
      submitJob()
      ObtainOutputFileAsNewCentersFile();
      currentChanges = getCounters(NumberOfClustersChanged);

      if( currentChanges ==0 || iteration > threshold)
          break;
      iteration++;
    }
}
```

5 Conclusion

MapReduce is one of the well-known distributed paradigms. MapReduce can handle massive amount of data by distributing the tasks of a given job among several machines. By adopting functional programming concept, MapReduce can run two functions, called map and reduce, on Big Data to emit the outcomes. The mappers process given input data independent of each other and sync the emitted results in reducer phase where the reducers process the data emitted by mappers. MapReduce reaps benefits from a suitable distributed file system, and HDFS is one of such file systems. HDFS serves a feature to move a map/reduce task to data location when performing a map reduce job in Hadoop. MapReduce blended with HDFS can tackle very large data with least effort in an efficient way. A problem can be ported to MapReduce paradigm by transforming the problem into asynchronous smaller tasks, map and reduce tasks. If synchronous is needed, it can be done once in a job through partitioning by a partitioner.

This chapter illustrated the primary components of Hadoop and its usage. An example scenario is also discussed to demonstrate how to write a 'HelloWorld' program in a MapReduce paradigm. An advanced MapReduce algorithm, k-means clustering, is presented to show the capability of MapReduce in various domains.

6 Review Questions and Exercises

1. Develop a MapReduce application to prepare a dictionary of keywords from a given text document collection.
2. Discuss the data flow in a MapReduce job with a neat diagram.
3. Develop a MapReduce application to find popular keywords in a given text document collection.
4. How HDFS recovers data when a node in the cluster goes down. Assume that the replication factor in the cluster is Review Question '3.'
5. Design a MapReduce job for matrix multiplication. The input is two file and each contains data for a matrix.

References

1. Tole, A.A.: Big data challenges. Database Syst. J. **4**(3), 31–40 (2013)
2. Von Neumann, J.: First draft of a report on the EDVAC. IEEE Ann. Hist. Comput. **15**(4), 27–75 (1993)
3. Riesen, R., Brightwell, R., Maccabe, A.B.: Differences between distributed and parallel systems. In: SAND98-2221, Unlimited Release, Printed October 1998. Available via http://www.cs.sandia.gov/rbbrigh/papers/distpar.pdf. (1998)
4. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self-stabilizing mutual exclusion. In: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing (PODC '90), pp. 119–131. ACM, New York (1990)

5. Gribble, S.D., et al.: Scalable, distributed data structures for internet service construction. In: Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation, vol. 4. USENIX Association (2000)
6. Gerbessiotis, Alexandros V., Valiant, Leslie G.: Direct bulk-synchronous parallel algorithms. J. Parallel Distrib. Comput. **22**(2), 251–267 (1994)
7. Leslie, G.V.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
9. Borthakur, D.: The hadoop distributed file system: architecture and design. Hadoop Project Website (2007). Available via https://svn.eu.apache.org/repos/asf/hadoop/common/tags/release-0.16.3/docs/hdfs_design.pdf
10. Hadoop' MapReduce Tutorial, Last updated on 08/04/2013. Available at http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
11. Vavilapalli, V.K., et al.: Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing. ACM (2013)
12. Zhang, Y., et al.: iMAPreduce: a distributed computing framework for iterative computation. J. Grid Comput. **10**(1), 47–68 (2012)
13. Chu, C., et al.: Map-reduce for machine learning on multicore. Adv. Neural Inf. Process. Syst. **19**, 281 (2007)
14. Zhao, W., Huifang, M., Qing, H.: Parallel k-means clustering based on mapreduce. Cloud Computing, pp. 674–679. Springer, Berlin (2009)
15. Martha, V.S.: GraphStore: a distributed graph storage system for big data networks. Dissertaion, University of Arkansas at Little Rock (2013). Available at http://gradworks.umi.com/35/87/3587625.html