CHAPTER 1

1. INTRODUCTION

Today we live in the digital world. The quantity of structured and unstructured data being created and spaced is breaking out with increasing level of digitization. The data is being generated from various sources – social media, transactions, sensors, digital images, videos, audios and clicks for domains including healthcare, retail, energy and utilities. Besides business and organizations, individuals add up to the data volume too. For instance, 32 billion content are being shared on Facebook every month; the photos viewed every 16 seconds in Picasa could cover a football field (Shelley).

Big data is a term for massive data sets having large, more varied and complex structure with the difficulties of storing, analyzing and visualizing for further processes or results (SINANC, SAGIROGLU, & Duygu, 2013). It is increasingly becoming obligatory for organizations to mine this data to stay competitive. Analyzing this data can provide additive competitive benefits for an enterprise. However the current volume of big data sets are too complicated to be managed and processed by conventional relational databases and data warehousing technologies.

Particle Swarm Optimization (PSO) is an evolutionary optimization algorithm that was inspired by experiments with simulated bird flocking. This evolutionary algorithm has become popular because it is simple, requires little tuning, and has been found to be effective for a wide range of problems. Often a function that needs to be optimized takes a long time to evaluate. A problem using web content, commercial transaction information, or bioinformatics data, for example, may involve large amounts of data and require minutes or hours for each function evaluation. To optimize such functions, PSO must be parallelized (McNabb, Monson, & D., 2012).

In our project we have proposed and implemented Particle Swarm Optimization (PSO), an evolutionary algorithm in a parallel fashion using MapReduce architecture. This optimization algorithm is generally used to analyze and extract some meaningful

information from large scale data that is usually available in unstructured manner. The MapReduce Particle Swarm Optimization program performs the same operations faster and more efficient than the sequential code. However, instead of performing PSO iterations internally, it delegates this work to the Hadoop MapReduce system, which makes it run in a parallel fashion.

1.1 Motivation

Big Data has become one of the buzzwords in IT industry during the last couple of years. Initially it was shaped by organizations which had to deal with speedy growth rates of data like web data, data resulting from scientific or business simulations or other data sources. The Google File System and MapReduce Architecture were resulted from the pressure to handle the expanding data amount on the web by Google (Ghemawat, Dean, & Sanjay, 2008). These technologies were tried to rebuild as open source software. This lead to Apache Hadoop and the Hadoop File System and laid the foundation for technologies under the umbrella of 'big data'.

Evolutionary Computing (EC) is a field of Computer science that adapts the theory of Darwinian evolution to optimize Computing problems (Eiben & Smith, 2003)**.** The basis of EC revolves around the process of natural evolution such as competition, random variation, and reproduction. The process of EC is globally applicable for wide range of problems due to the fact that evolution itself is an optimization process. Two of its most popular algorithms, Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) have been used ubiquitously to optimize problems in numerous fields of study.

Despite the robust nature of EC, expensive computing and storing resources are required due to its parallel nature. Thus when EC gets parallely executed it performs better as compared to that of its normal serial execution. Therefore, EC has since become a popular subject of parallelization using various parallel processing techniques and architectures.

Optimization is the process of search for the maximum or minimum of a given objective function. Particle Swarm Optimization (PSO) is a simple and effective evolutionary algorithm, but it may take hours or days to optimize difficult objective functions which are deceptive or expensive. Deceptive functions may be highly multimodal and

multidimensional, and PSO requires extensive exploration to avoid being trapped in local optima. Expensive functions, whose computational complexity may arise from dependence on detailed simulations or large datasets, takes a long time to evaluate. For deceptive or expensive objective functions, PSO must be parallelized to use multiprocessor systems and clusters efficiently.

This thesis investigates the implications of parallelizing PSO and in particular, the details of parallelization and the effects of large swarms. PSO can be expressed naturally in Google's MapReduce framework (Ghemawat, Dean, & Sanjay, 2008) to develop a robust and simple parallel implementation that includes communication, load balancing, and fault tolerance. This flexible implementation makes it easy to apply modifications to the algorithm, such as those that improve optimization of difficult objective functions and improve parallel performance.

1.2 Objective

The objective of the thesis is to propose and realize an approach to implement Particle Swarm Optimization Algorithm in a parallel manner and use it to compute the effort of COCOMO Model. The population of the swarm here are in the form of big data are stored in Hadoop Distributed File System (HDFS) on Hadoop to have better fault tolerance and to store large amount of data in distributed manner.

The machine learning technique used in our model is Particle Swarm Optimization. PSO is a fairly recent addition to the family of non-gradient based, probabilistic search approach that is based on a simple social model and is closely related to swarming theory. Although PSO algorithm presents several attractive properties to the designer, they are plagued by high computational cost as measured by elapsed time (Ludwig, 2014).

Thus, this evolutionary algorithm is implemented in MapReduce architecture on Hadoop to overcome its limitations of large scale serial implementation for computationally intensive functions. MapReduce PSO is simple, flexible, scalable and robust because it is designed in the MapReduce parallel programming model.

The benefits that can be obtained using this model are remarkable: the unique property of Hadoop of replication factor provides us at different level of abstraction with the availability of documentation that is beneficial for anyone who wants to access the records. Hadoop also improves response time and is fault tolerant due to which storing database on it is very advantageous.

Thus, this model overall proves to be beneficial for both data owner as well as end-users. Using this model an optimal value of effort required could be computed for a given project within a small span of time which could save both computation time and resources when there are reasonable numbers of projects or complex computations.

1.3 Thesis Outline

The thesis is divided into 6 chapters:

Chapter 1 is the introduction part. It describes the motivation of the work, objective of the thesis and also the structure of the thesis.

Chapter 2 is the literature survey. It includes the description of the work and contribution of various people in the same field and their findings.

Chapter 3 is basically the research background. This chapter discusses optimization, overview of particle swarm optimization and its application. Also a brief overview of big data, its characteristics, challenges faced by big data and its benefits. Workflow of MapReduce architecture is also discussed in this chapter.

Chapter 4 presents the detailed explanation of Hadoop and its components used in the work and their working.

Chapter 5 presents the implementation or proposed work and the results obtained after running the code on our setup.

Conclusion and future work are presented in **chapter 6**.

CHAPTER 2

2. LITERATURE REVIEW

In the work proposed by Andrew W. McNabb, Christopher K. Monson, and Kevin D. Seppi (McNabb, Monson, & D., 2012), they have stated that a problem using web content, commercial transaction information or bioinformatics data may involve large amounts of data and requires minutes and hours for each function evaluation. And PSO must be parallelized to optimize such functions. They have proposed MapReduce Particle Swarm Optimization (MRPSO) that is intended for computationally intensive functions and have used problem of training a Radial Basis Function (RBF) Network as representative of optimization problems which uses large amount of data. Once a program successfully scales, it must still address the issue of failing nodes. Google too faced the same problem in large scale parallelization. It is demonstrated that MRPSO scales to 256 processors on moderately difficult problems and tolerates node failures.

In the work of Junjun Wang, Dongfeng Yuan and Mingyan Jiang (Wang, Yuan, & Jiang, 2012)**,** the authors have proposed an improved method called parallel K-PSO. As K-Means has limited processing capability because of its time complexity in serial scenario, improving the performance of K-Means has become challenging and significant. The K-Means Clustering Algorithm has following two inherent drawbacks:

i) Excessively depends on the initial cluster centers.

ii) Converges easily to the local optimum.

The idea proposed focuses on enhancing the global search capability of K-Means by taking the improved PSO to optimize its initial clusters, and improving the performance in dealing with massive data by transforming K-Means from serial to parallel using MapReduce.

In the work of Chia-Yu Lin, Yuan-Ming Pai, Kun-Hung Tsai, Charles H.-P. Wen, and Li-Chun Wang (Lin, Pai, Tsai, & Wen, 2013), a MapReduce Modified Cuckoo Search (MRMCS) has been proposed. This is an efficient modified Cuckoo Search implementation on MapReduce architecture. Cuckoo Search is modified to deal with the issue of job partitioning i.e. which job goes to the mappers and which job goes to the reducers. Following four evaluation functions and two engineering design problems are used to conduct the experiment:

- **a)** Evaluations Functions
  - **i)** Griewank Function
  - **ii)** Rastrigrin Function
  - **iii)** Rosenbrock Function
  - **iv)** Sphere Function
- **b)** Engineering Design Problems
  - **i)** Application of Spring Design
  - **ii)** Application of Welded Beam Design

The result shows that the MRMCS outperforms and shows better convergence in obtaining optimality than MRPSO with 2-4 times speedup.

The work proposed by Dino Kečo, Abdulhamit Subasi (Kečo & Subasi, 2012), presents parallel implementation of Genetic Algorithm. This parallel implementation of Genetic Algorithm is compared with its serial implementation in solving One Max (Bit Counting) problem. The comparison criteria used in this work are fitness convergence, quality of final solution, algorithm scalability and cloud resource utilization. The proposed model shows better performance and fitness convergence than serial model, but has lower quality of solution because of species problem.

CHAPTER 3

3. RESEARCH BACKGROUND

3.1 Particle Swarm Optimization

Particle swarm optimization (PSO) is an artificial intelligence (AI) technique that can be used to find approximate optimal solutions to extremely difficult or impossible numeric maximization and minimization problems. Particle Swarm Optimization is an algorithm capable of optimizing a non-linear and multimodal problem which usually reaches good solutions efficiently and quickly while requiring minimal parameterization.

The algorithm and its concept of "Particle Swarm Optimization"(PSO) were introduced by James Kennedy and Russel Eberhart in 1995 (Eberhart, Kennedy, & Russell, 1995). However, its origins go further backwards since the basic principle of optimization by swarm is inspired in previous attempts at reproducing observed behaviors of animals in their natural habitat, such as bird flocking or fish schooling, and thus ultimately its origins are nature itself. These roots in natural processes of swarms lead to the categorization of the algorithm as one of Swarm Intelligence and Artificial Life.

3.1.1 Overview

The basic concept of the algorithm is to create a swarm of particles which move in the space around them (**problem space**) searching for their goal, the place which best suits their needs given by a **fitness function**. A nature analogy with birds is the following: a bird flock flies in its environment looking for the best place to rest (the best place can be a combination of characteristics like space for all the flock, food access, water access or any other relevant characteristic).

The PSO algorithm is population-based: a set of potential solutions evolves to approach a convenient solution (or set of solutions) for a problem. Being an optimization method, the

aim is finding the global optimum of a real-valued function (fitness function) defined in a given space (search space).

The social metaphor that led to this algorithm can be summarized as follows: the individuals that are part of a society hold an opinion that is part of a "belief space" (the search space) shared by every possible individual. Individuals may modify this "opinion state" based on three factors:

- The knowledge of the environment (its fitness value)
- The individual's previous history of states (its memory)
- The previous history of states of the individual's neighborhood

An individual's neighborhood may be defined in several ways, configuring somehow the "social network" of the individual. Several neighborhood topologies exist (full, ring, star, etc.) depending on whether an individual interacts with all, some, or only one of the rest of the population.

Following certain rules of interaction, the individuals in the population adapt their scheme of belief to the ones that are more successful among their social network. Over the time, a culture arises, in which the individuals hold opinions that are closely related.

3.1.2 Optimization

Optimization is the mechanism by which one finds the maximum or minimum value of a function or process. This mechanism is used in fields such as physics, chemistry, economics, and engineering where the goal is to maximize efficiency, production, or some other measure. Optimization can refer to either minimization or maximization; maximization of a function f is equivalent to minimization of the opposite of this function, −f.

Mathematically, a minimization task is defined as:

Given f: $R^n \rightarrow R$

Find $\hat{y} \in R^n$ such that $f(\hat{y}) \leq f(\hat{y}), \forall\ \hat{y} \in R^n$

Similarly, a maximization task is defined as:

Given f: $R^n \rightarrow R$

Find $\hat{y} \in R^n$ such that $f(\hat{y}) \geq f(\hat{y}), \forall\ \hat{y} \in R^n$

The domain R^n of f is referred to as the search space (or parameter space). Each element of R^n is called a candidate solution in the search space, with $\hat{y}$ being the optimal solution. The value n denotes the number of dimensions of the search space, and thus the number of parameters involved in the optimization problem. The function f is called the objective function, which maps the search space to the function space. Since a function has only one output, this function space is usually one-dimensional. The function space is then mapped to the one-dimensional fitness space, providing a single fitness value for each set of parameters. This single fitness value determines the optimality of the set of parameters for the desired task.

In most cases, the function space can be directly mapped to the fitness space. However, the distinction between function space and fitness space is important in cases such as multi-objective optimization tasks, which include several objective functions drawing input from the same parameter space. For a known (differentiable) function f, calculus can fairly easily provide us with the minima and maxima of f. However, in real-life optimization tasks, this objective function f is often not directly known. Instead, the objective function is a "black box" to which we apply parameters (the candidate solution) and receive an output value. The result of this evaluation of a candidate solution becomes the solution's fitness. The final goal of an optimization task is to find the parameters in the search space that maximize or minimize this fitness.

In some optimization tasks, called constrained optimization tasks, the elements in a candidate solution can be subject to certain constraints (such as being greater than or less than zero).

3.1.3 The PSO Algorithm

As stated before, PSO simulates the behaviors of bird flocking. Suppose the following scenario: a group of birds are randomly searching food in an area. There is only one piece of food in the area being searched. All the birds do not know where the food is. But they know how far the food is in each iteration. So what's the best strategy to find the food? The effective one is to follow the bird which is nearest to the food.

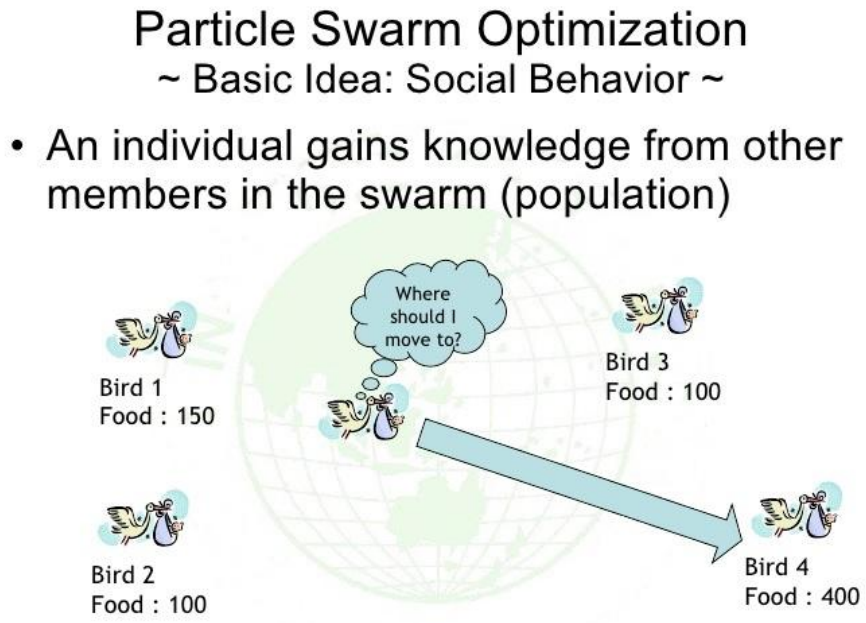


Figure 3.1: Basic Idea- Particle Swarm Optimization

PSO learned from the scenario and used it to solve the optimization problems. In PSO, each single solution is a "bird" in the search space. We call it "particle". All of particles have fitness values which are evaluated by the fitness function to be optimized, and have velocities which direct the flying of the particles. The particles fly through the problem space by following the current optimum particles.

PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. In every iteration, each particle is updated by following two "best" values.

The first one is the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called **pbest**. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the population. This best value is a global best and called **gbest**.

After finding the two best values, the particle updates its velocity and positions with following equation (a) and (b).

v[ ] = v[ ] + c1 * rand() * (pbest[ ] - present[ ]) + c2 * rand() * (gbest[ ] - present[ ])….**(a)**

present[ ] = present[ ] + v[ ]....……..**(b)**

v[ ] is the particle velocity, present[ ] is the current particle (solution). pbest[ ] & gbest[ ] are defined as stated before. rand () is a random number between (0,1). c1, c2 are learning factors, usually c1 = c2 = 2.

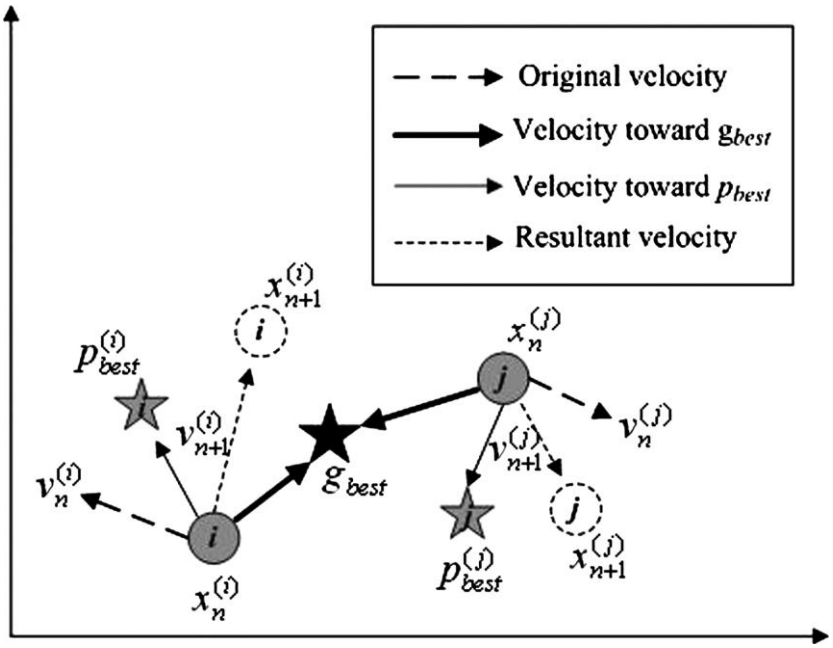


Figure 3.2: Resultant velocity of the particles

The pseudo code of the procedure is as follows:

For each particle
  Initialize particle
END
Do
  For each particle
    Calculate fitness value
    If the fitness value is better than the best fitness value (pBest) in history
      set current value as the new pBest
  End
  Choose the particle with the best fitness value of all the particles as the gBest
  For each particle
    Calculate particle velocity according equation (a)
    Update particle position according equation (b)
  End
While maximum iterations or minimum error criteria is not attained

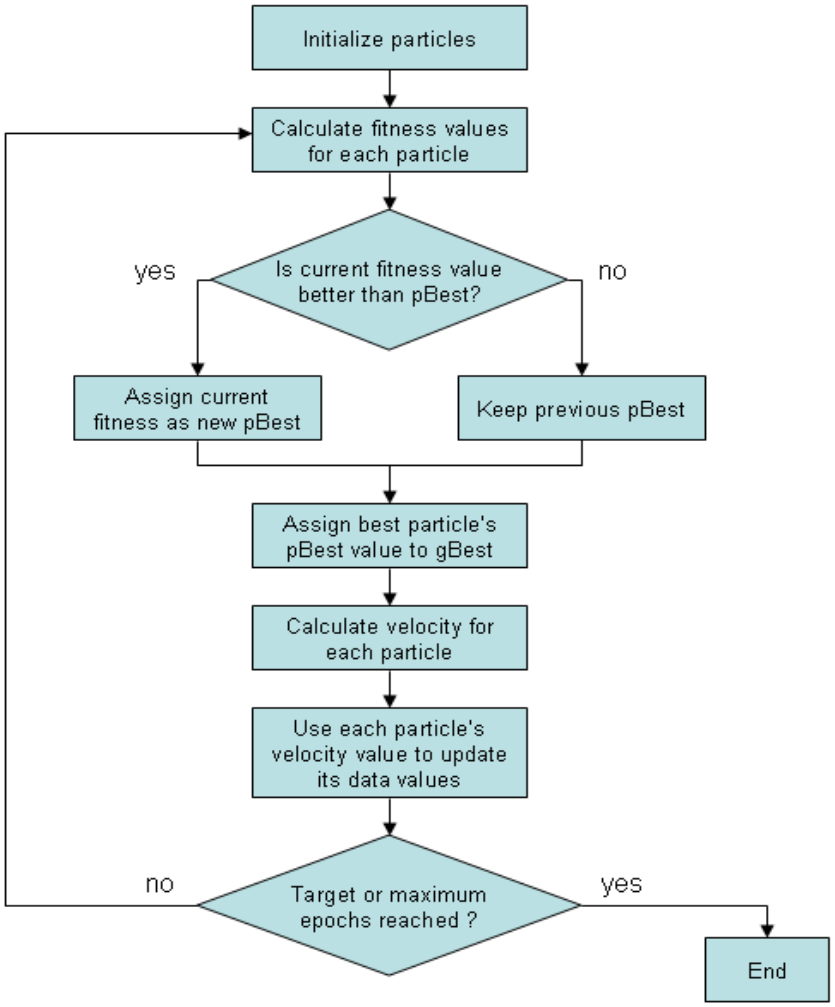


Figure 3.3: Flowchart of PSO

3.1.4 Applications

The applications of the Particle Swarm Optimization Algorithm can be summarized as:

I. Non-convex Search Spaces

Convexity is extremely important in optimization algorithms because it has nice properties involving gradients that can make optimization guaranteed. In a space like the Rastrigin function, particle swarm optimization is able to deal with the local minima and in many cases finds the global optimum.

II. Integer or Discontinuous Space

In a similar vein, integer search spaces are difficult for traditional optimization algorithms. In problems that involve integer variables, the search space is discontinuous and gradient information is rarely effective. Particle swarm optimization does not require the space to be continuous but precautions need to be taken to position particles exactly on specific values.

III. Neural-Networks

One could treat the neural network weight space as a high dimensional particle swarm optimization search space. In this application of PSO, particles could be a swarm of neural networks attempting to find the lowest error on some classification or regression task.

IV. Support Vector Machines (and Regression)

For classification and regression tasks using Support Vector Machines, the user has the ability to choose a few hyper parameters that control the kernel function, the cost associated with failing to correctly classify a training item, the loss function parameters, etc. Since the search space is continuous there is a combinatorial explosion as the number of hyper parameters increases. Particle swarm optimization could be used to find the optimal set of hyper parameters by creating particles that search a space of various values for each of the hyper parameters while attempting to produce the best error on the data.

V. Multi-Objective Optimization

In the spirit of optimization problems, multi-objective programs involve optimizing programs with multiple objective functions where objective functions are potentially in conflict with one another. In these problems, particle swarm optimization can be used to find a good trade-off between the different objective functions.

3.2 Big Data

Big Data is the next generation of data warehousing and business analytics and is poised to deliver top line revenues cost efficiently for enterprises. The greatest part about this phenomenon is the rapid pace of innovation and change; where we are today is not where we'll be in just two years and definitely not where we'll be in a decade.

This new age didn't suddenly emerge. It's not an overnight phenomenon. It's been coming for a while. It has many deep roots and many branches. In fact, if you speak with most data industry veterans, Big Data has been around for decades for firms that have been handling tons of transactional data over the years—even dating back to the mainframe era. The reasons for this new age are varied and complex can be summarized as:

i) **Computing perfect storm**. Big Data analytics are the natural result of four major global trends: Moore's Law (which basically says that technology always gets cheaper), mobile computing (that smart phone or mobile tablet in your hand), social networking (Facebook, Foursquare, Pinterest, etc.), and cloud computing (you don't even have to own hardware or software anymore; you can rent or lease someone else's).

ii) **Data perfect storm**. Volumes of transactional data have been around for decades for most big firms, but the flood gates have now opened with more *volume*, and the *velocity* and *variety*—the three Vs—of data that has arrived in unprecedented ways. This perfect storm of the three Vs makes it extremely complex and cumbersome with the current data management and analytics technology and practices.

iii) **Convergence perfect storm**. Another perfect storm is happening, too. Traditional data management and analytics software and hardware technologies, open-source technology, and commodity hardware are merging to create new alternatives for IT and business executives to address Big Data analytics.

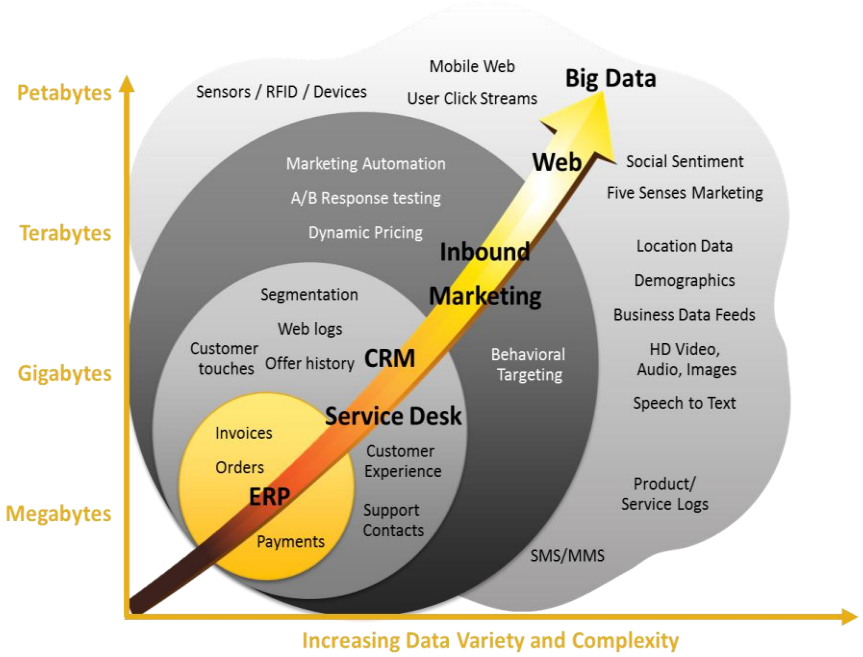


Figure 3.4: Big Data Variety & Complexity

In summary, the Big Data world is being fueled with an abundance mentality; a rising tide lifts all boats. This new mentality is fueled by a gigantic global corkboard that includes data scientists, crowd sourcing, and opens source methodologies.

3.2.1 Definition

Since 2011 interest in an area known as big data has increased exponentially. The term big data has become ubiquitous. Owing to a shared origin between academia, industry and the media there is no single unified definition, and various stakeholders provide diverse and often contradictory definitions.

The McKinsey Global Institute has defined this term as – *"Big Data refers to data sets whose size is beyond the ability of typical database software tools to capture, store,*

manage and analyze." The process of research into massive amounts of data to reveal hidden patterns and secret correlations named as big data analytics (SINANC, SAGIROGLU, & Duygu, 2013).

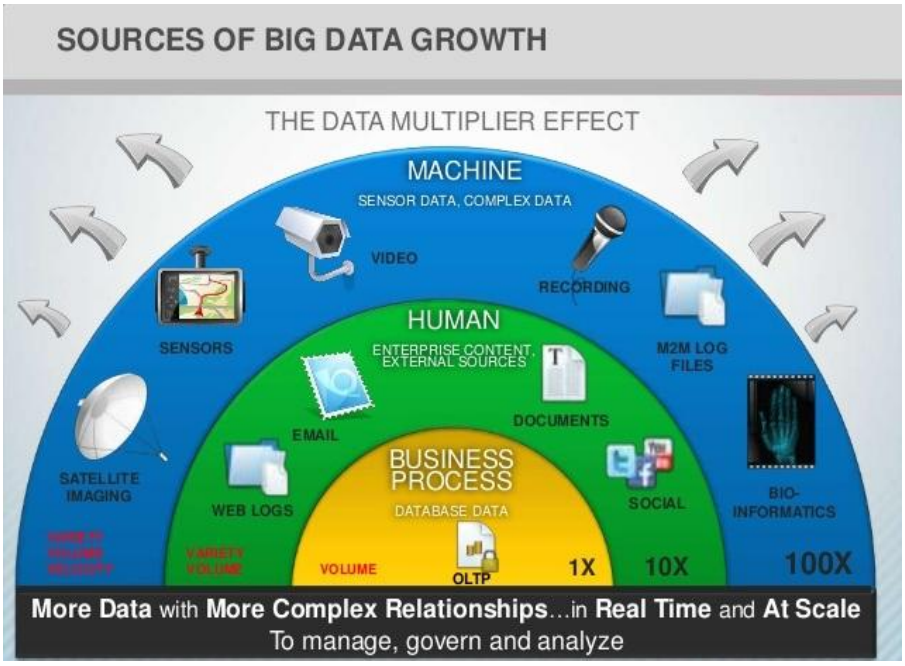


Figure 3.5: Sources of Big Data

3.2.2 Characteristics: The Four V's of Big Data

Big Data is characterized by following four main components:

- **Volume** – The quantity of data that is generated is very significant in this context. It is the size of the data which determines the value and potential of the data under consideration and whether it can actually be considered Big Data or not. The name 'Big Data' itself contains a term which is related to size and hence the characteristic.
- **Variety** - The next aspect of Big Data is its variety. This means that the category to which Big Data belongs to is also a very essential fact that needs to be known by the data analysts. This helps the people, who are closely analyzing the data and are associated with it, to effectively use the data to their advantage and thus upholding the importance of the Big Data. Big Data comes from a great variety of sources and generally is in three types: **structured**, **semi-structured** and

unstructured. Structured data inserts a data warehouse already tagged and easily sorted but unstructured data is random and difficult to analyze. Semi-structured data does not conform to fixed fields but contains tags to separate data elements (SINANC, 2013).

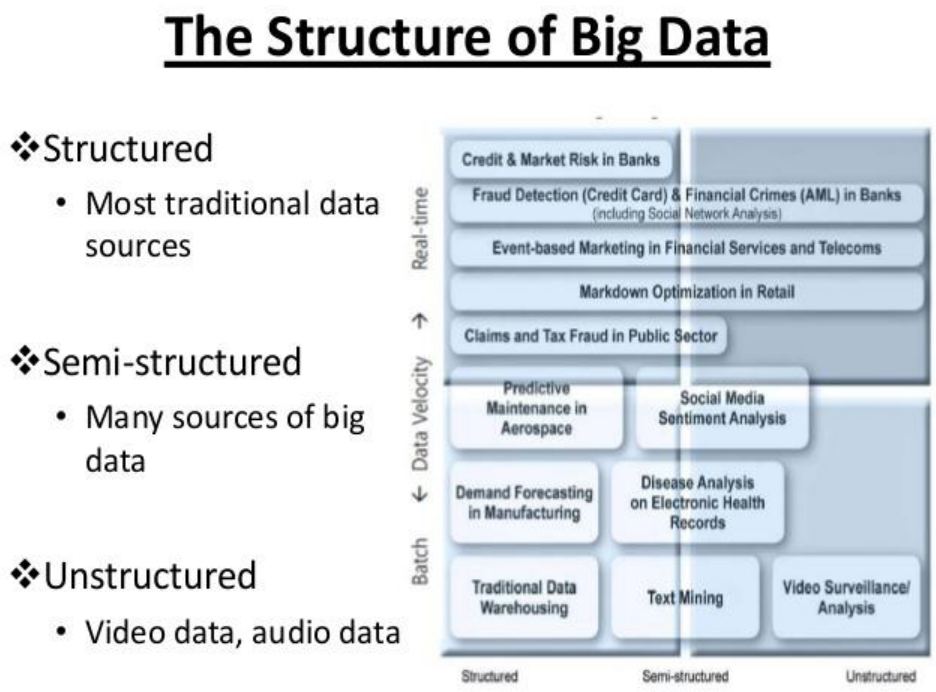


Figure 3.6: Forms of Big Data

- **Velocity** - The term 'velocity' in the context refers to the speed of generation of data or how fast the data is generated and processed to meet the demands and the challenges which lie ahead in the path of growth and development.
- **Veracity** – Big Data 'veracity' refers to the biases, noise and abnormality in data. Is the data is being stored, and mined meaningful to the problem being analyzed. The quality of the data being captured can vary greatly. Accuracy of analysis depends on the veracity of the source data.

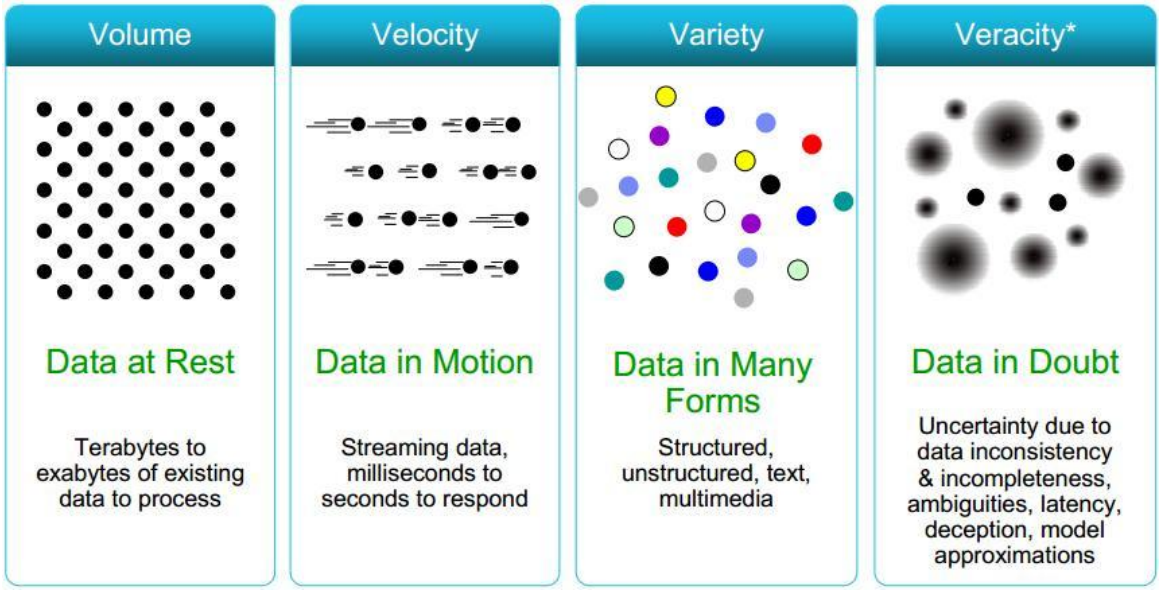


Figure 3.7: 4 V's of Big Data

3.2.3 Big Data Challenges

To fully take advantage of visual analytics, organizations will need to address several challenges related to visualization and big data. Here we've outlined some of those key challenges – and potential solutions (Singh & Ravinder, 2014).

1. Meeting the need for speed

In today's hypercompetitive business environment, companies not only have to find and analyze the relevant data they need, they must find it quickly. Visualization helps organizations perform analyses and make decisions much more rapidly, but the challenge is going through the sheer volumes of data and accessing the level of detail needed, all at a high speed. The challenge only grows as the degree of granularity increases. One possible solution is hardware. Some vendors are using increased memory and powerful parallel processing to crunch large volumes of data extremely quickly. Another method is putting data in-memory but using a grid computing approach, where many machines are used to solve a problem. Both approaches allow organizations to explore huge data volumes and gain business insights in near-real time.

2. Understanding the data

It takes a lot of understanding to get data in the right shape so that you can use visualization as part of data analysis. For example, if the data comes from social media content, you need to know who the user is in a general sense – such as a customer using a particular set of products – and understand what it is you're trying to visualize out of the data. Without some sort of context, visualization tools are likely to be of less value to the user.

One solution to this challenge is to have the proper domain expertise in place. Make sure the people analyzing the data have a deep understanding of where the data comes from, what audience will be consuming the data and how that audience will interpret the information.

3. Addressing data quality

Even if you can find and analyze data quickly and put it in the proper context for the audience that will be consuming the information, the value of data for decision-making purposes will be jeopardized if the data is not accurate or timely. This is a challenge with any data analysis, but when considering the volumes of information involved in big data projects, it becomes even more pronounced. Again, data visualization will only prove to be a valuable tool if the data quality is assured. To address this issue, companies need to have a data governance or information management process in place to ensure the data is clean. It's always best to have a proactive method to address data quality issues so problems won't arise later.

4. Displaying meaningful results

Plotting points on a graph for analysis becomes difficult when dealing with extremely large amounts of information or a variety of categories of information. For example, imagine you have 10 billion rows of retail SKU data that you're trying to compare. The user trying to view 10 billion plots on the screen will have a hard time seeing so many data points. One way to resolve this is to cluster data into a higher-level view where smaller groups of data become visible. By grouping the data together, or "binning," you can more effectively visualize the data.

5. Dealing with outliers

The graphical representations of data made possible by visualization can communicate trends and outliers much faster than tables containing numbers and text. Users can easily spot issues that need attention simply by glancing at a chart. Outliers typically represent about 1 to 5 percent of data, but when you're working with massive amounts of data, viewing 1 to 5 percent of the data is rather difficult. How do you represent those points without getting into plotting issues? Possible solutions are to remove the outliers from the data (and therefore from the chart) or to create a separate chart for the outliers. You can also bin the results to both view the distribution of data and see the outliers. While outliers may not be representative of the data, they may also reveal previously unseen and potentially valuable insights.

3.2.4 Benefits of Big Data Analytics

Google, eBay and LinkedIn were among the first to experiment with big data. They developed proof of concept and small-scale projects to learn if their analytical models could be improved with new data sources. In many cases, the results of these experiments were positive.

Today, big data analytics is no longer just an experimental tool. Many companies have begun to achieve real results with the approach, and are expanding their efforts to encompass more data and models. Three major benefits of big data analytics are:

1. Cost reduction

Big data technologies like Hadoop and cloud-based analytics can provide substantial cost advantages. While comparisons between big data technology and traditional architectures (data warehouses and marts in particular) are difficult because of differences in functionality, a price comparison alone can suggest order-of-magnitude improvements.

Virtually every large company, however, is employing big data technologies not to replace existing architectures, but to augment them. Rather than processing and storing vast quantities of new data in a data warehouse, for example, companies are using

Hadoop clusters for that purpose, and moving data to enterprise warehouses as needed for production analytical applications.

Well-established firms like Citi, Wells Fargo and USAA all have substantial Hadoop projects underway that exist alongside existing storage and processing capabilities for analytics. While the long-term role of these technologies in enterprise architecture is unclear, it's likely that they will play a permanent and important role in helping companies manage big data.

2. Faster, better decision making

Analytics has always involved attempts to improve decision making, and big data doesn't change that. Large organizations are seeking both faster and better decisions with big data, and they're finding them. Driven by the speed of Hadoop and in-memory analytics, several companies focus on speeding up existing decisions.

For example, Caesars, a leading gaming company that has long embraced analytics, is now embracing big data analytics for faster decisions. The company has data about its customers from its Total Rewards loyalty program, web click streams, and real-time play in slot machines. It has traditionally used all those data sources to understand customers, but it has been difficult to integrate and act on them in real time, while the customer is still playing at a slot machine or in the resort.

Caesars has found that if a new customer to its loyalty program has a run of bad luck at the slots; it's likely that customer will never come back. But if it can present, say, a free meal coupon to that customer while he's still at the slot machine, he is much more likely to return to the casino later. The key, however, is to do the necessary analysis in real time and present the offer before the customer turns away in disgust with his luck and the machines at which he's been playing.

In pursuit of this objective, Caesars has acquired Hadoop clusters and commercial analytics software. It has also added some data scientists to its analytics group.

Some firms are more focused on making better decisions analyzing new sources of data. For example, health insurance giant United Healthcare is using "natural language processing" tools from SAS to better understand customer satisfaction and when to intervene to improve it. It starts by converting records of customer voice calls to its call center into text and searching for indications that the customer is dissatisfied. The company has already found that the text analysis improves its predictive capability for customer attrition models.

3. New products and services

Perhaps the most interesting use of big data analytics is to create new products and services for customers. Online companies have done this for a decade or so, but now predominantly offline firms are doing it too. GE, for example, has made a major investment in new service models for its industrial products using big data analytics.

Verizon Wireless is also pursuing new offerings based on its extensive mobile device data. In a business unit called Precision Market Insights, Verizon is selling information about how often mobile phone users are in certain locations, their activities and backgrounds. Customers thus far have included malls, stadium owners and billboard firms.

3.3 Map Reduce Architecture

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster. Conceptually similar approaches have been very well known since 1995 with the Message Passing Interface standard having reduce and scatter operations.

A MapReduce program is composed of a **Map( )** procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce( )** procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications

and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

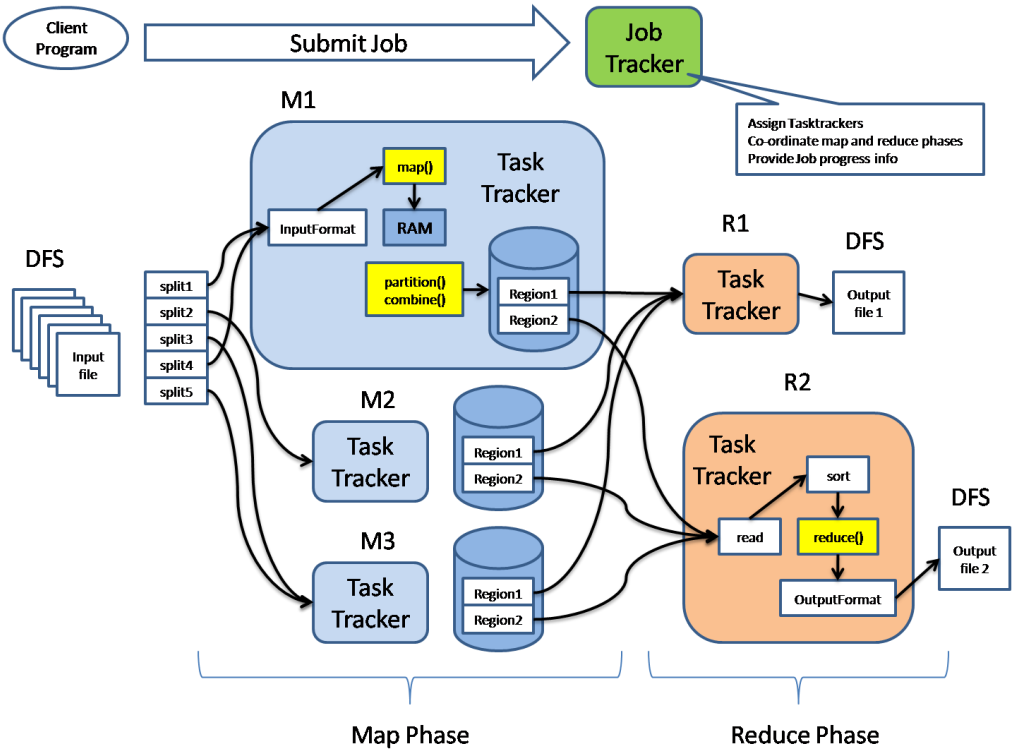


Figure 3.8: Map Reduce Architecture

The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as in their original forms. The key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine once. As such, a single-threaded implementation of MapReduce (such as MongoDB) will usually not be faster than a traditional (non-MapReduce) implementation; any gains are usually only seen with multi-threaded implementations.

The use of this model is beneficial only when the optimized distributed shuffle operation (which reduces network communication cost) and fault tolerance features of the MapReduce framework come into play. Optimizing the communication cost is essential to a good MapReduce algorithm.

MapReduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation that has support for distributed shuffles is part of Apache Hadoop. The name MapReduce originally referred

to the proprietary Google technology, but has since been genericized. MapReduce as a big data processing model is considered dead by many domain experts, as development has moved on to more capable and less disk-oriented mechanism that incorporate full map and reduce capabilities.

3.3.1 Inputs and Outputs

The MapReduce framework operates exclusively on **<key, value>** pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

The key and value classes have to be serializable by the framework and hence need to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.

Input and Output types of a MapReduce job:

Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

Map (k1,v1) → list (k2,v2).....**(c)**

The ***Reduce*** function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

Reduce (k2, list (v2)) → list (v3)……**(d)**

3.3.2 Workflow

MapReduce is implemented in a master/worker configuration, with one master serving as the coordinator of many workers. A worker may be assigned a role of either a *map worker* or a *reduce worker*.

Step 1. Split input

The first step, and the key to massive parallelization in the next step, is to split the input into multiple pieces. Each piece is called a split, or shard. For *M* map workers, we want to

have *M* shards, so that each worker will have something to work on. The number of workers is mostly a function of the amount of machines we have at our disposal.

The MapReduce library of the user program performs this split. The actual form of the split may be specific to the location and form of the data. MapReduce allows the use of custom readers to split a collection of inputs into shards, based on specific format of the files.

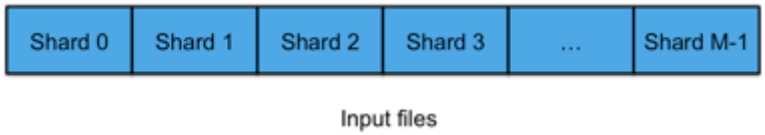


Figure 3.9: Split input into shards

Step 2. Fork processes

The next step is to create the master and the workers. The master is responsible for dispatching jobs to workers, keeping track of progress, and returning results. The master picks idle workers and assigns them either a map task or a reduce task. A map task works on a single shard of the original data. A reduce task works on intermediate data generated by the map tasks. In all, there will be *M* map tasks and *R* reduce tasks. The number of reduce tasks is the number of partitions defined by the user. A worker is sent a message by the master identifying the program (map or reduce) it has to load and the data it has to read.

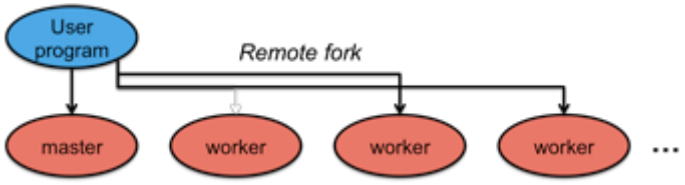


Figure 3.10: Remotely execute worker processes

Step 3. Map

Each *map* task reads from the input shard that is assigned to it. It parses the data and generates *(key, value)* pairs for data of interest. In parsing the input, the *map* function is likely to get rid of a lot of data that is of no interest. By having many map workers do this in parallel, we can linearly scale the performance of the task of extracting data.

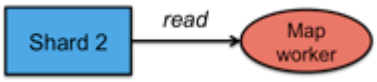


Figure 3.11: Map task

Step 4: Map worker: Partition

The stream of *(key, value)* pairs that each worker generates is buffered in memory and periodically stored on the local disk of the map worker. This data is partitioned into *R* regions by a partitioning function.

The partitioning function is responsible for deciding which of the *R* reduce workers will work on a specific key. The default partitioning function is simply a hash of *key* modulo *R* but a user can replace this with a custom partition function if there is a need to have certain keys processed by a specific reduce worker.

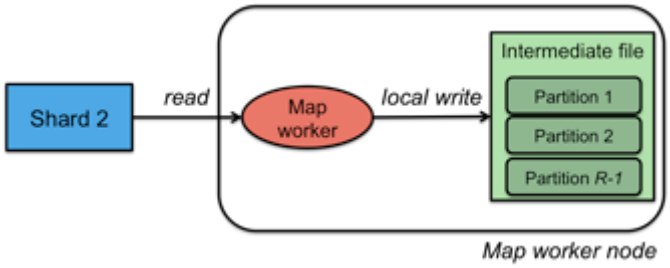


Figure 3.12: Create intermediate files

Step 5: Reduce: Sort (Shuffle)

When all the map workers have completed their work, the master notifies the reduce workers to start working. The first thing a reduce worker needs to is to get the data that it

needs to present to the user's *reduce* function. The reduce worker contacts every map worker via remote procedure calls to get the *(key, value)* data that was targeted for its partition. This data is then sorted by the keys. Sorting is needed since it will usually be the case that there are many occurrences of the same key and many keys will map to the same reduce worker (same partition). After sorting, all occurrences of the same key are grouped together so that it is easy to grab all the data that is associated with a single key.

This phase is sometimes called the shuffle phase.

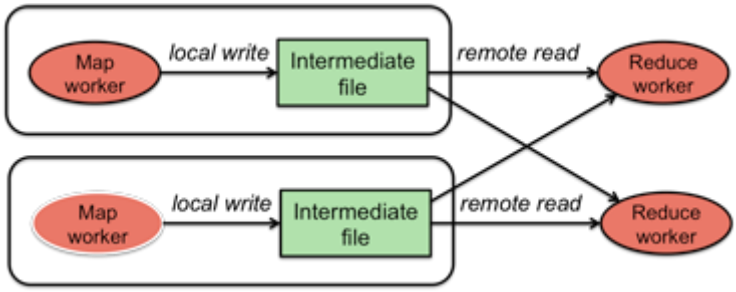


Figure 3.13: Sort and merge partitioned data

Step 6: Reduce function

With data sorted by keys, the user's *Reduce* function can now be called. The reduce worker calls the *Reduce* function once for each unique key. The function is passed two parameters: the key and the list of intermediate values that are associated with the key.

The *Reduce* function writes output sent to file.

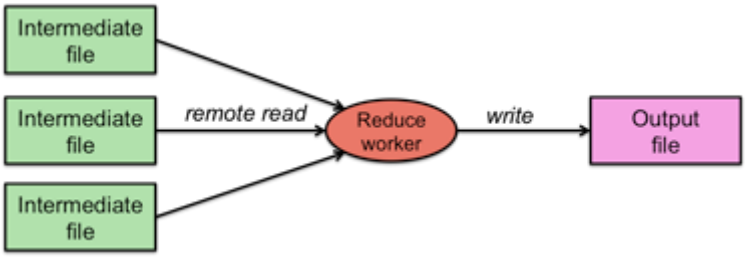


Figure 3.14: Reduce function writes output

Step 7: Done!

When all the reduce workers have completed execution, the master passes control back to the user program. Output of MapReduce is stored in the *R* output files that the *R* reduce workers created.

The big picture

Figure 7 illustrates the entire MapReduce process. The client library initializes the shards and creates map workers, reduce workers, and a master. Map workers are assigned a shard to process. If there are more shards than map workers, a map worker will be assigned another shard when it is done. Map workers invoke the user's *Map* function to parse the data and write intermediate *(key, value)* results onto their local disks. This intermediate data is partitioned into *R* partitions according to a partioning function. Each of *R* reduce workers contacts all of the map workers and gets the set of *(key, value)* intermediate data that was targeted to its partition. It then calls the user's *Reduce* function once for each unique key and gives it a list of all values that were generated for that key. The *Reduce* function writes its final output to a file that the user's program can access once MapReduce has completed.

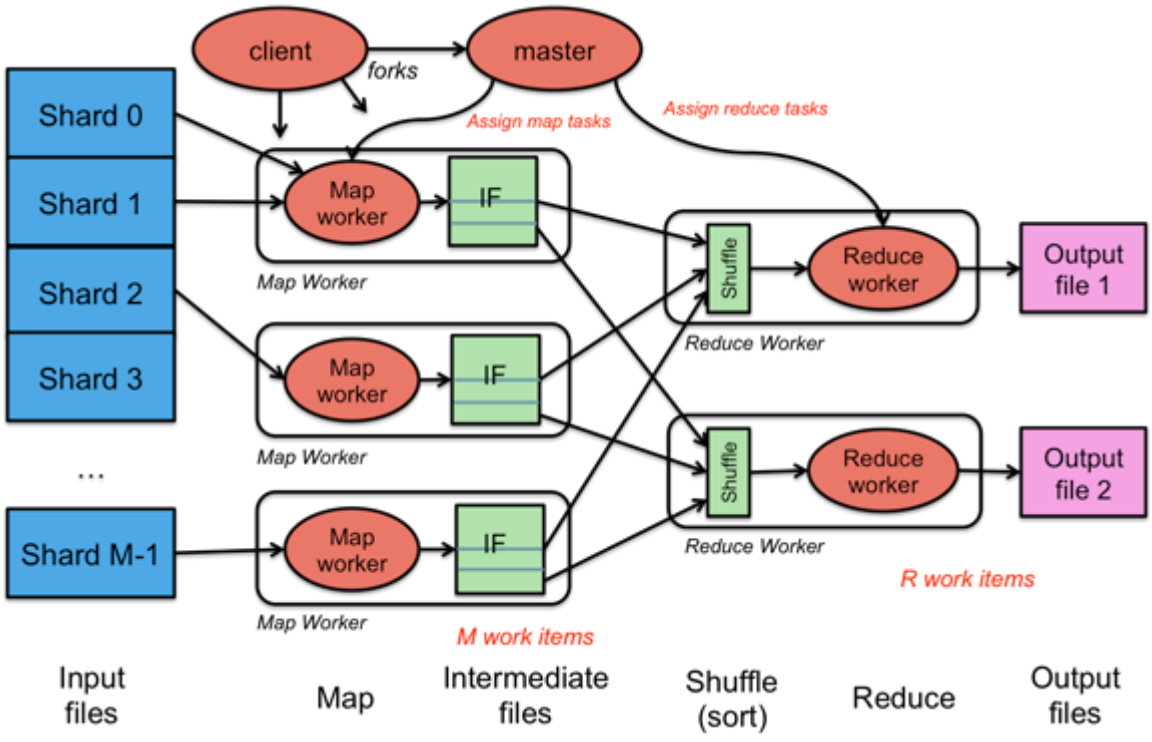


Figure 3.15: MapReduce

3.3.3 Benefits

The following table describes some of MapReduce's key benefits:

Table 1: Benefits of MapReduce Architecture

Benefit	Description
Simplicity	Developers can write applications in their language of choice, such as Java, C++ or Python, and MapReduce jobs are easy to run
Scalability	MapReduce can process petabytes of data, stored in HDFS on one cluster
Speed	Parallel processing means that MapReduce can take problems that used to take days to solve and solve them in hours or minutes
Recovery	MapReduce takes care of failures. If a machine with one copy of the data is unavailable, another machine has a copy of the same key/value pair, which can be used to solve the same sub-task. The JobTracker keeps track of it all.
Minimal data motion	MapReduce moves compute processes to the data on HDFS and not the other way around. Processing tasks can occur on the physical node where the data resides. This significantly reduces the network I/O patterns and contributes to Hadoop's processing speed.

CHAPTER 4

4. APACHE HADOOP

Apache Hadoop is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures (of individual machines or racks of machines) are commonplace and thus should be automatically handled in software by the framework.

The core of Apache Hadoop consists of a storage part (Hadoop Distributed File System (HDFS)) and a processing part (MapReduce). Hadoop splits files into large blocks and distributes them amongst the nodes in the cluster. To process the data, Hadoop MapReduce transfers packaged code for nodes to process in parallel, based on the data each node needs to process. This approach takes advantage of data locality—nodes manipulating the data that they have on hand—to allow the data to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are connected via high-speed networking.

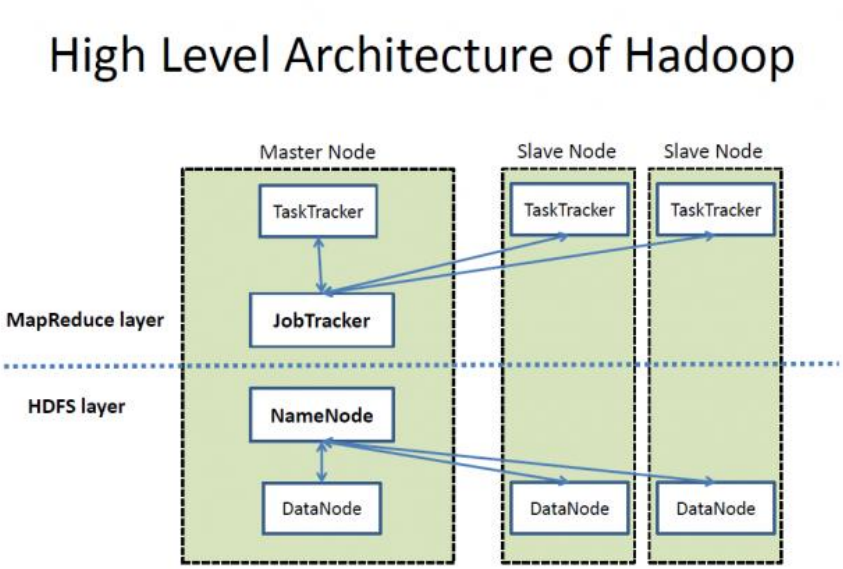


Figure 4.1: Architecture of Hadoop

The base Apache Hadoop framework is composed of the following modules:

- *Hadoop Common* – contains libraries and utilities needed by other Hadoop modules;
- *Hadoop Distributed File System (HDFS)* – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- *Hadoop YARN* – a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications; and
- *Hadoop MapReduce* – a programming model for large scale data processing.

The term "Hadoop" has come to refer not just to the base modules above, but also to the "ecosystem", or collection of additional software packages that can be installed on top of or alongside Hadoop, such as Apache Pig, Apache Hive, Apache HBase, Apache Spark, and others.

Apache Hadoop's MapReduce and HDFS components were inspired by Google papers on their MapReduce and Google File System. Hadoop is supported by its own list of operating systems - Red Hat Enterprise, CentOS, Oracle Linux, Ubuntu, SUSE Linux Enterprise Server.

4.1 Characteristics of Hadoop

1. Scale-Out rather than Scale-Up means Hadoop requires more machines or nodes to be added to the existing distributed system which is easier instead of adding more RAM or CPU for scaling up which is more difficult.

2. It brings code to data, in data to code data is loaded to the processor from storage device located remotely and results are sent back to storage device as done traditionally whereas to bring code to data means both processor and storage are located on same machine and processors run code and access underlying database.

3. **Deal with failures –** they are common while working with large number of machines but Hadoop is designed to cope up with failures as data is replicated on various nodes and tasks are retired.

4. **Abstract complexity** of distributed and concurrent applications, allows developers to focus on application development and business logic and frees developer from worrying about processing Big Data on clusters of commodity hardware and system level challenges

5. Vibrant open-source community

6. Many tools and products reside on top of Hadoop

7. Hadoop consists of the Hadoop Common, which provides access to the file systems supported by Hadoop.

8. Hadoop has published APIs

4.2 Hadoop Cluster

Hadoop Cluster is a set of "cheap" commodity hardware networked together which resides in the same location i.e. set of servers resides in set of racks which are in data centre. “Cheap” Commodity Server Hardware means that there is no need for super-computers, and can use commodity unreliable hardware. The hardware used are not desktops but servers. Hadoop Cluster is a collection of Hadoop nodes where each node consists of a Processor and Storage as shown in figure 4.2. In Hadoop cluster, processors access underlying local storage and execute code.

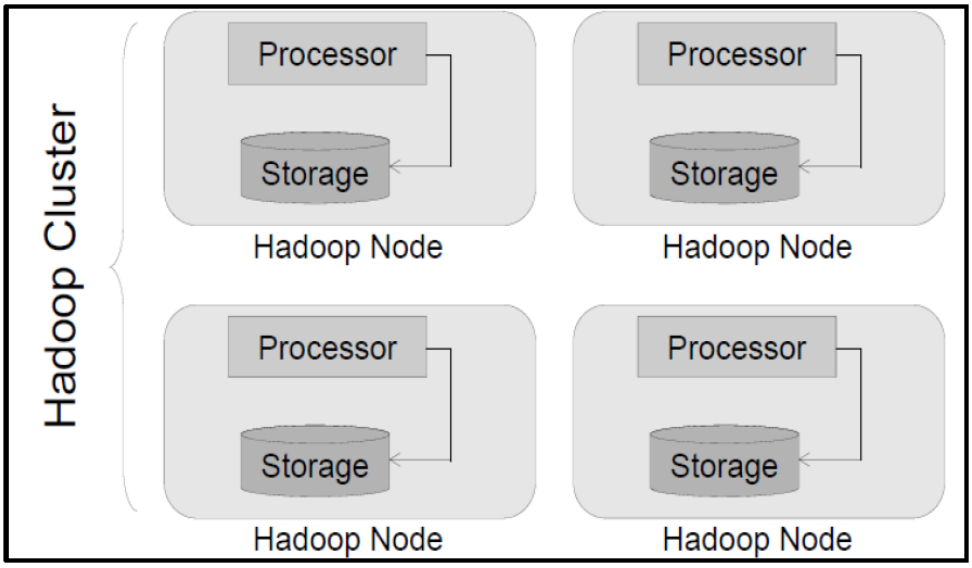


Figure 4.2: Hadoop Cluster

4.3 Hadoop Ecosystem

1. Hadoop Distributed File System

2. MapReduce: a distributed data processing framework

3. HBase: Hadoop column database; supports random reads and limited queries and batch

4. Zookeeper: Highly-Available Coordination Service

5. Oozie: Hadoop workflow scheduler and manager

6. Pig: Data processing language and execution environment

7. Hive: Data warehouse with SQL interface

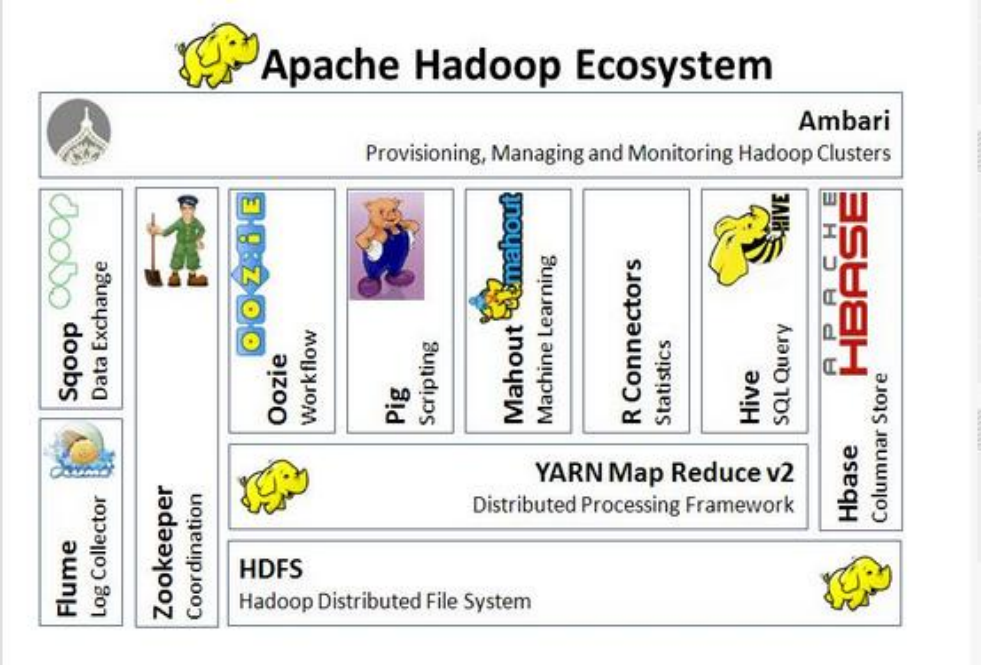


Figure 4.3: Apache Hadoop Ecosystem

4.4 Comparison between Hadoop and Distributed Databases

Table 2: Comparison between Hadoop and Distributed Databases

S.No.	Hadoop	Distributed Databases
1.	Used with relational database for batch processing	Until recently used for batch processing in various applications
2.	Scale out using more machines	Scale up using CPUs and RAM
3.	Cheap commodity is used to scale out	Expensive to scale for larger installations
4.	Works best with unstructured or semi-structured data	Works well with structured data tables that conform to a specified schema
5.	For offline batch processing	For Online Transactions and low-latency queries
6.	Is designed to stream large amounts of data and large files	It works best with small records
7.	Supports JSON, XML, images, etc.	Does not have support for JSON, images, XML etc.

4.5 Benefits

While large Web 2.0 companies such as Google and Facebook use Hadoop to store and manage their huge data sets, Hadoop has also proven valuable for many other more traditional enterprises based on its five big advantages.

1. Scalable

Hadoop is a highly scalable storage platform, because it can store and distribute very large data sets across hundreds of inexpensive servers that operate in parallel. Unlike traditional relational database systems (RDBMS) that can't scale to process large amounts of data, Hadoop enables businesses to run applications on thousands of nodes involving thousands of terabytes of data.

2. Cost effective

Hadoop also offers a cost effective storage solution for businesses' exploding data sets. The problem with traditional relational database management systems is that it is extremely cost prohibitive to scale to such a degree in order to process such massive volumes of data. In an effort to reduce costs, many companies in the past would have had to down-sample data and classify it based on certain assumptions as to which data was the most valuable. The raw data would be deleted, as it would be too cost-prohibitive to keep. While this approach may have worked in the short term, this meant that when business priorities changed, the complete raw data set was not available, as it was too expensive to store. Hadoop, on the other hand, is designed as a scale-out architecture that can affordably store all of a company's data for later use. The cost savings are staggering: instead of costing thousands to tens of thousands of pounds per terabyte, Hadoop offers computing and storage capabilities for hundreds of pounds per terabyte.

3. Flexible

Hadoop enables businesses to easily access new data sources and tap into different types of data (both structured and unstructured) to generate value from that data. This means businesses can use Hadoop to derive valuable business insights from data sources such as social media, email conversations or clickstream data. In addition, Hadoop can be used for a wide variety of purposes, such as log processing, recommendation systems, data warehousing, market campaign analysis and fraud detection.

4. Fast

Hadoop's unique storage method is based on a distributed file system that basically 'maps' data wherever it is located on a cluster. The tools for data processing are often on

the same servers where the data is located, resulting in much faster data processing. If you're dealing with large volumes of unstructured data, Hadoop is able to efficiently process terabytes of data in just minutes, and petabytes in hours.

5. Resilient to failure

A key advantage of using Hadoop is its fault tolerance. When data is sent to an individual node, that data is also replicated to other nodes in the cluster, which means that in the event of failure, there is another copy available for use.

The MapR distribution goes beyond that by eliminating the NameNode and replacing it with a distributed No NameNode architecture that provides true high availability. Our architecture provides protection from both single and multiple failures.

When it comes to handling large data sets in a safe and cost-effective manner, Hadoop has the advantage over relational database management systems, and its value for any size business will continue to increase as unstructured data continues to grow.

4.6 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System is a file system that runs on top of native file system like Ext3, Ext4 and others, and is based on Google file system. It gives user appearance of a single disk. It is highly fault tolerant in a way that it can handle disk crashes, machine crashes, etc. It is built upon cheap commodity hardware which reduces the overall cost of installation of Hadoop

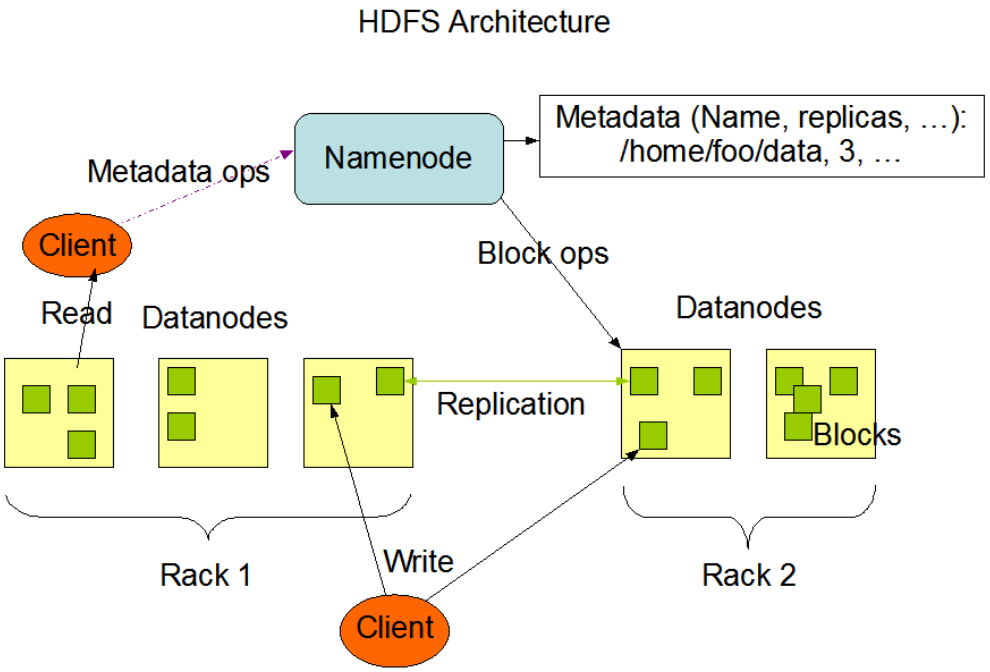


Figure 4.4: HDFS Architecture

4.6.1 HDFS Daemons

File system cluster is being managed by three types of processes namely, NameNode, DataNode and Secondary NameNode.

a. **NameNode:** It manages the file systems namespace, meta-data and file blocks. It runs on one machine and manages several machines. All DataNodes report to NameNode about their presence and according to the number of available DataNodes it manages degree of replication as decided by the Administrator. For fast access NameNode keeps all block meta-data in memory. The other role is to serve the client queries, it allows clients to add/copy/move/delete a file, it will records the actions into a transaction log. For the performance, it save the whole file structure tree in RAM and hard drive. A HDFS only allow one running NameNode, that's why it is a single point of failure, if the NameNode failed or goes down, the whole file system will goes offline too. So, for the NameNode machine, we need to take special cares on it, such as adding more RAM to it, this will increase the file system capacity, and do not make it as DataNode, JobTracker and other optional roles.

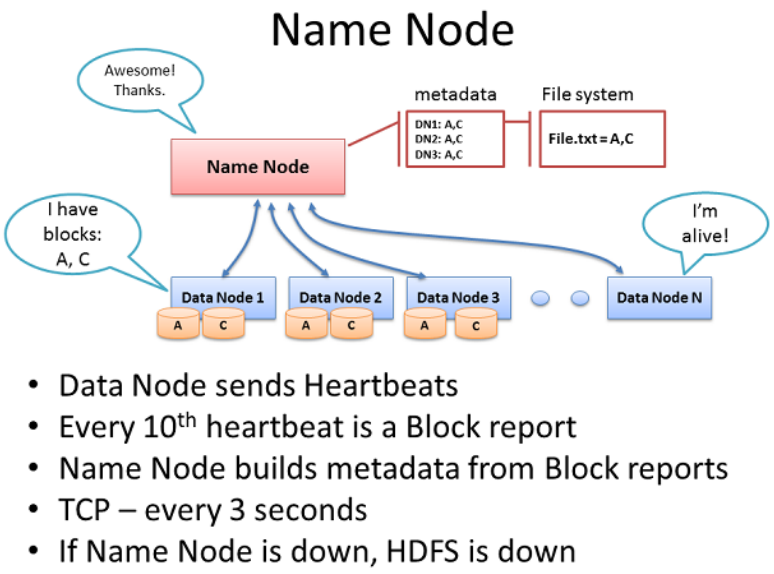


Figure 4.5: NameNode

b. **DataNode:** It stores and retrieves data blocks according to the request after it has reported to NameNode about its health. It runs on many machines and forms the cluster. On startup, DataNode will connect to the NameNode and get ready to respond to the operations from NameNode. After the NameNode telling the position of a file to the client, the client will directly talk to the DataNode to access the files. DataNodes could also talk to each other when they replicating data. The DataNode will also periodically send a report of all existing blocks to the NameNode and validates the data block checksums

c. **Secondary NameNode:** It performs the house keeping work so that NameNode doesn't have to do it and reduces the load of NameNode. It requires similar hardware as NameNode machine and is not used for high-availability – not a backup for NameNode. Its works is to back-up the metadata and store it to the hard disk, this may helping to reduce the restarting time of NameNode. In HDFS, the recent actions on HDFS will be stored in to a file called EditLog on the NameNode, after restarting HDFS; the NameNode will replay according to the Editlog. Secondary NameNode will periodically combines the content of EditLog into a checkpoint and clear the EditLog File, after that, the NameNode will replay start from the latest checkpoint, the restarting time of NameNode will be reduced.

4.6.2 HDFS File Read and Write

In the Hadoop Cluster, NameNode accepts the request but does not directly read or write data to HDFS which is one of the reasons for HDFS‟s scalability. Initially, client interacts with the NameNode to update the HDFS namespace of NameNode and client retrieves block locations for reading and writing then it directly interacts with Datanode to read/write data. The Read and Write operations on the file are explained below.

4.6.2.1 HDFS Write

The write operation in HDFS is done in seven steps as shown in figure 4.6.

1. Create new file in the NameNode's Namespace and calculate block topology
2. Stream data to the first DataNode
3. Stream data to the second DataNode in the pipeline
4. Stream data to the third DataNode
5. Success/Failure acknowledgement
6. Success/Failure acknowledgement
7. Success/Failure acknowledgement

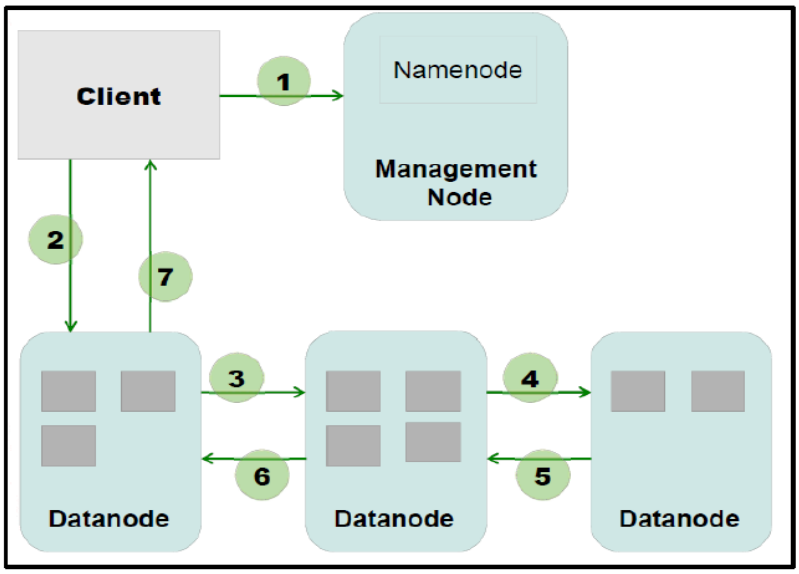


Figure 4.6: HDFS Write

4.6.2.2 HDFS Read

The read operation in HDFS is done in three steps as shown in figure 4.7.

1. Client retrieves block location from NameNode
2. Client read blocks to re-assemble the file
3. Client read blocks to re-assemble the file

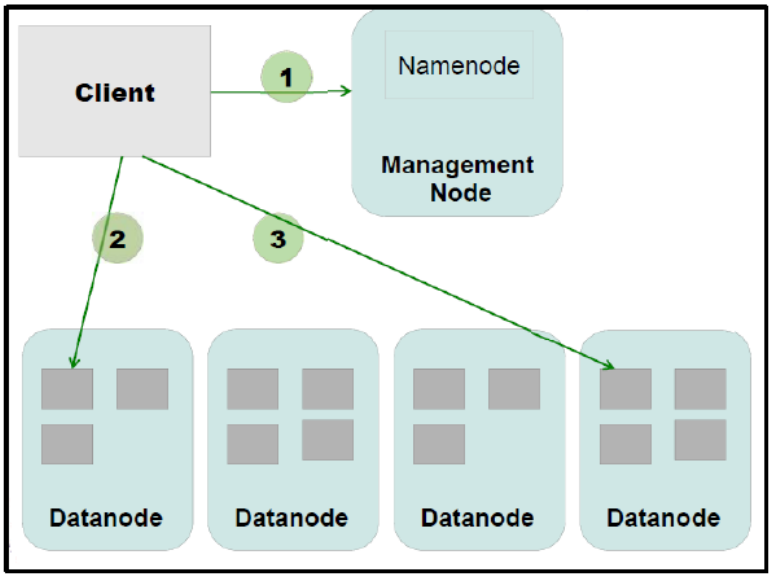


Figure 4.7: HDFS Read

CHAPTER 5

5. IMPLEMENTATION AND RESULTS

This section demonstrates the parallel implementation of Particle Swarm Optimization algorithm with its application in finding the effort of a project using COCOMO Model. As described in previous section, the particles in PSO update their velocity and position in each iteration so as to merge to a global optimal solution. PSO has a bottleneck for mathematically expensive functions and thus needs to be parallelized for more efficiency.

The steps of the algorithm that can be parallelized are:

i) Each iteration of PSO algorithm can be executed in parallel
ii) Each particle can update its position parallely

But the iterations in PSO can't be parallelized as there is a issue of message passing among the particles in each iteration. The results of previous iteration must be preserved and utilized in next iteration. Thus to transform PSO in parallel fashion following issues needs to be considered:

i) Determine the input to MapReduce architecture
ii) Determine the jobs of mapper and reducer tasks
iii) Exchanging information between mapper tasks

A large number of initial random population is provided as input to the MapReduce framework, which is then splitted into chunks and distributed across various mappers. Each swarm (particle) is represented as a key-value pair as:

Key K1 : swarm_id

Value V1 : set of attributes representing the swarm like position, velocity, personal best, etc.

❖ Creating initial population

A large number of initial random population is created, which is provided as input to the MapReduce framework. The population created is in the form of tuples containing following information: swarm id, position, velocity, personal best, and position at personal best.

Here, position of the particle is in the form of (a,b), where a and b are the coefficients of the COCOMO model used to compute effort as:

$$\textbf{Effort (E)} = a\,(LOC)^{b} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \textbf{(e)}$$

Initial velocities of the particle have been assigned to zero. The particles update their velocity and position at each iteration using equation (a) and (b).

❖ Map Function

In Parallel Particle Swarm Optimization Algorithm, the map function is called once for each particle. The key is the offset of the tuple which represents the swarm and the value is the state string representation of the swarm (particle).

In mapper, fitness of each particle is being computed and then personal best of each particle is evaluated. This updated state string representation of the swarm containing personal best and the position at which personal best is obtained is sent to the reducer. Here we have taken variable block size so as to test the efficiency of our model. Thus, for each block one mapper task is executed. After all mapper tasks have been executed, the reducer is initiated.

map (key, value, context)

1. particle = Particle (value)
2. # evaluate fitness of the particle
3. fitness = calculate_fitness (particle.position)
4. # calculate personal best of the particle
5. if (fitness<pBest)
6. pBest = fitness
7. end if
8. emit (key, repr (particle))

❖ **Reduce Function**

The reduce function in this model receives a key and a list of all associated values. Here, we have explicitly defined the key as 1 so as to run a single reducer to reduce overhead. In reduce phase the global best of all the particles is calculated and the position at which global best is obtained is stored.

Next, the particle's velocity and position are updated and this new generation with updated state string representation of the particles is sent to the mapper for the next iteration.

reduce (key, value_list, context)

```
1. particle = none
2. gBest = none
3. # finding gBest i.e. global best
4. for value in value_list
5. record = Particle ( value )
6. if ( record.pBest<=gBest )
7. gBest = record.pBest
8. end if
9. # update the particle
10. particle.update ( new_position, new_velocity )
11. end for
12. # emit the updated particle for next generation
13. emit ( key, repr ( particle ) )
```

This complete process in iterated for a fixed number of iterations or until some terminating criteria is met. Thus, the obtained gBest is the final output and the position at which gBest is obtained is the required value of a and b.

❖ Results

Here, we have optimized Basic COCOMO model parameters (a,b), such that calculated effort approximates to the actual effort for NASA 63 project dataset.

Formally, this problem can be described as finding parameter X= *{x*1, *x*2*}*, with *xi* $\in$ *{*0,5*}* that minimize the following equation:

MMRE = [Actual – x1(KLOC)^{x2}] /Actual……..**(f)**

Here MMRE - Mean Magnitude of Relative Error used as evaluation criteria for assessment of optimized parameters. And since parameter's x1 and x2 are specific to project mode therefore we execute program for each mode separately.

A. Environment

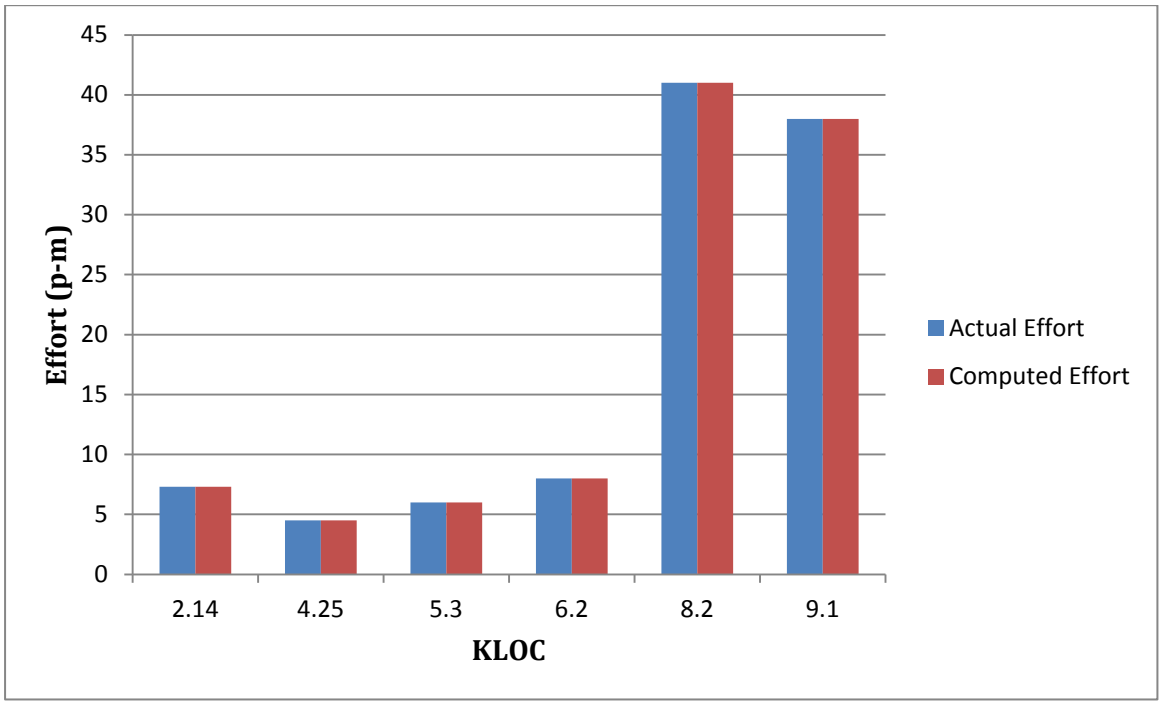
We have implemented the model on Hadoop (2.6) and ran it on our Hadoop cluster with three nodes. Each node runs a two dual Intel Quad cores, 4GB RAM and 250 GB hard disks. The nodes are integrated with Hadoop Distributed File System (HDFS) yielding a potential single image storage space of 2 *52/3 = 34.6*TB* (since the replication factor of HDFS is set to 3). Each node can run 5 mappers and 3 reducers in parallel.

B. Tests

We have performed three tests and obtained the following results:

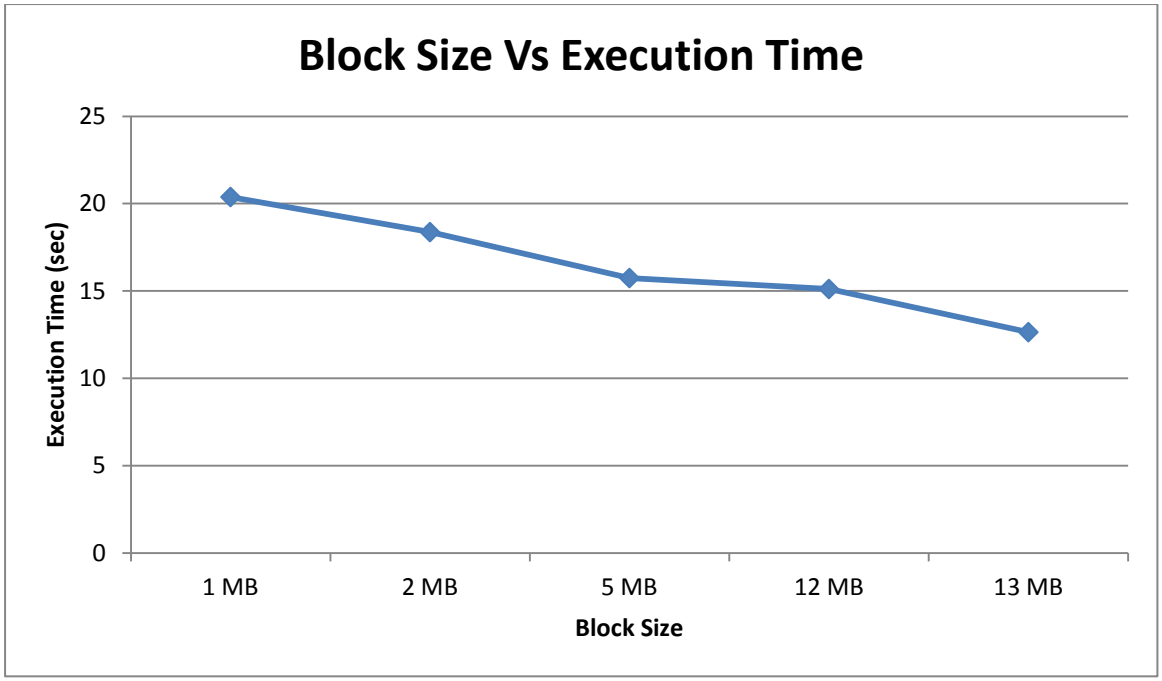
I. Comparison between actual effort and computed effort

In this experiment we have taken the population size of 2 lakhs and performed 3 iterations to obtain the result in each case.



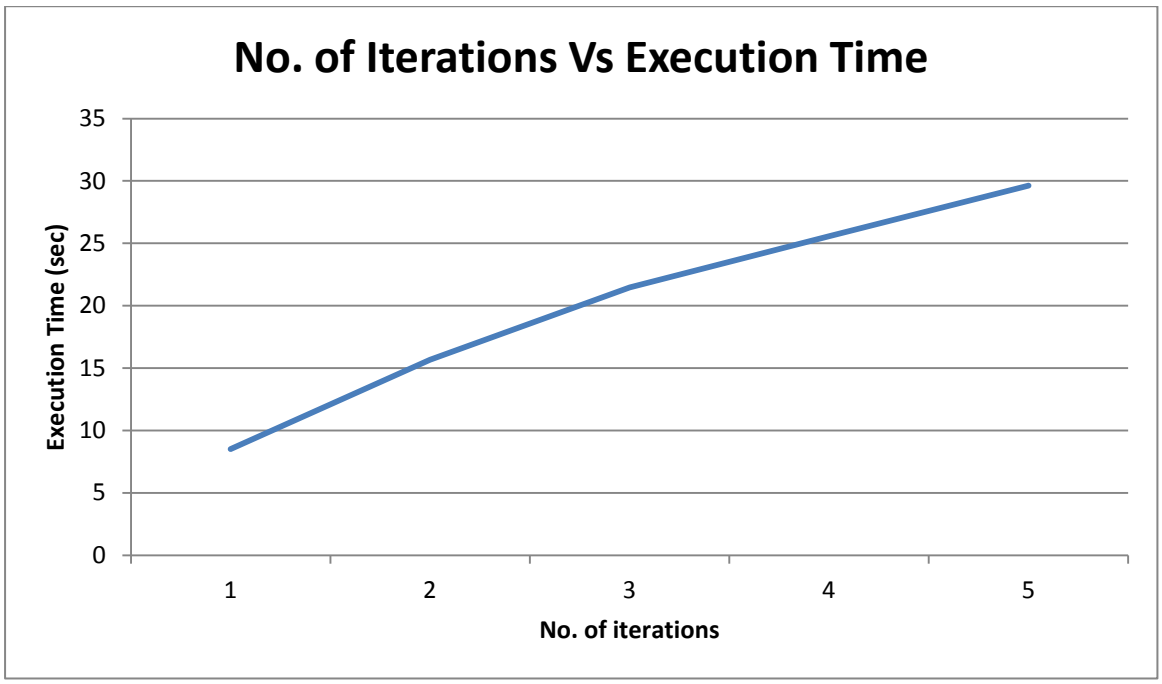
II. Variation of execution time with different block size

Taking the population size of 2 lakhs and performing 3 iterations we observed a measurable variation in execution time while changing the block size of Hadoop resulting in different number of mapper tasks in each iteration.



III. Variation of execution time with number of iterations

The execution time also varies greatly with different number of iteration. Again while performing this experiment we have kept population size of 2 lakhs and block size of 3MB.



CHAPTER 6

6. CONCLUSION AND FUTURE WORK

The Particle Swarm Optimization algorithm can easily be parallelized using MapReduce architecture to solve the optimization problems involving large search space. The problem of large search space could be easily tackled by generating a large number of populations so that each particle in the population needs to search a comparatively smaller search space and can thus find the solution more efficiently in less time.

It has been seen that lots of communication, task start up overhead is associated with Hadoop Map Reduce Architecture thus is not suitable for problems having small search space with less computation.

Experimental results shows that the proposed model can have better convergence than its serial implementation for intensively expensive computation functions. This model would be really efficient if we deal with solving the problem involving a large number of dimensions. In such case it would be beneficial to use this parallel implementation for faster convergence to an optimal solution.

The proposed model for parallel PSO can use a large population but could not be applied to a big dataset due to the fact that the particles keep on updating themselves in each iteration. So, a further detailed study is required to modify the algorithm in such a way so that it could be applied to big data to conclude with some meaningful information out of it.

In future work the proposed model should be modified such that it could work upon a big dataset involving large number of dimensions. Also some work could be done in order to improve the execution time of the algorithm examining other features of MapReduce architecture like partitioner, combiner etc. which may reduce the processing.