Exploring Efficient Ways of Distributing Code Quality Metrics in Cloud

by

Dibangku Baruah, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2016

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Dibangku Baruah

August 30, 2016

Permission to Lend and/or Copy

Dibangku Baruah

August 30, 2016

Acknowledgments

I wish to express my sincere gratitude to my supervisor, Prof. Stephen Barrett for his support, guidance, suggestions and feedback which helped guide me throughout the course of my work on the dissertation.

I would like to thank Paul Duggan and John Welsh, for their constant help and supervision throughout the duration of my thesis.

I am also extremely grateful to all my friends for their words of encouragement, and for always being there to discuss any technical issues.
Finally, a big thank you to my wonderful parents, without whom all of this would not have been possible.

Dibangku Baruah

University of Dublin, Trinity College

September 2016

Exploring Efficient Ways of Distributing Code Quality Metrics in Cloud

Dibangku Baruah, M.Sc.
University of Dublin, Trinity College, 2016

Supervisor: Prof. Stephen Barrett

The objective of this thesis, is to explore ways of efficiently distributing calculation of code quality metrics. A thorough study is made in this thesis, to determine whether or not it is possible to distribute calculation of source code complexity metrics and, if possible whether that solution holds against code repositories of all sizes and to explore the best way to distribute out the processing to many nodes.

The report here outlines the theory about software metrics, their purpose and classification, with particular emphasis on the metrics that is used in the study and later for experimentation.

This thesis ends with an in-depth analysis and evaluation of the results, gathered through the running of the chosen complexity metric on different designed solution to the problem of distributing the processing to many machines.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter we will discuss briefly about software quality metrics and why they are so important in the present context. Then we will introduce the research questions and objectives. And finally we will outline the structure of this dissertation.

1.1 Background

IEEE Standard Glossary of Software Engineering Terms define software quality metric as "a quantitative measure of the degree to which a system, component, or process processes a given attribute" [1]. In other words metrics are techniques or formulas that measure some specific properties or characteristics of a software.

One of the most critical issue faced in software engineering is the modularization of source code, so that the modules are easily maintainable. Hence quantitative representations for the modularization of software systems, in the form of mathematical techniques are needed, so that the resulting modules that are hard to test or maintain can be easily identified. "You can't manage what you can't measure" - DeMarco famously said once [10] [4] .

Software metrics can also be used for :

- Estimating the cost and schedule of future projects.
- Evaluating the productivity impacts of new tools and techniques.
- Determining, predicting and improving software quality over time.
- Assisting in quality control and productivity.
- Assessing the quality of technical products.
- Anticipating and reducing future maintenance cost.

1.2 Motivation

Currently a lot of frameworks are available for calculating source code quality metrics, such as total number of duplicated lines, unit tests coverage, lines of code, security ratings, frequency of comments, etc on platforms like Sonarqube [21], but the process of calculation of these metrics are either confined to a single machine, or the process of calculation of the metrics having to be done from scratch each time any new changes comes in. So for code repositories that very huge, tens of thousands of lines of code, the calculation of these metrics are a very time consuming affair, with all the tasks having to be carried out all over again.

1.3 Research Questions

The research questions that this dissertations aims to answer are :

- Amongst all the code quality metrics available today, which would be the most suitable one for efficiently distributing to a number of machines.

- Is there a way to hold intermediate results from past calculations of a particular metrics, so that the whole calculation doesn't have to be repeated all over again from scratch.
- Which design strategy would be the most suitable for calculating the chosen metric in a distributed environment.

1.4 Research Objectives

Based on the research questions mentioned above, following are the objectives of this dissertation :

- Study the software quality metrics currently available and find the most suitable one for the experiment.
- Design solutions on how to best utilize the resources of a distributed computing infrastructure.
- Implement the different design approaches.
- Evaluate results of the implementations and see how the different solutions fair in comparison to the other.
- Provide evidence of the best design strategy to use for the metric in question.

1.5 Dissertation Outline

The dissertation is organized as follows :

- Chapter 1 - An introduction to the research and outlines the motivation, research questions and objectives of this dissertation.
- Chapter 2 - Introduces the different code quality metrics that are available and a brief introduction to each one of them, with their advantages and drawbacks.

- Chapter 3 - Provides a detailed study on the chosen metric.
- Chapter 4 - Introduces the different design approaches used for building the application.
- Chapter 5 - Provides the implementation detail of the application.
- Chapter 6 - Provides an evaluation of the results derived from the experiments.
- Chapter 7 - Provides future scope of the project and conclusion.

Chapter 2

Software Metric

A software metric, is a standard of measure of a degree to which a software system or process processes some property. Even if a metric is not a measurement(metrics are functions, while measurements are the numbers obtained by the application of metrics), often the two terms are used synonymously. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development. The goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments [23].

Software metric provides a quantitative measure of how good a piece of software is, in relation to the effort needed on the part of humans to make that software work. The whole premise behind software metric is that, before anything can be measured, it needs to be converted into numbers. Software metrics have been found to be very useful in areas, such as improving the performance of software, to software planning. Human interaction is extremely important as software alone cannot perform on its own and software metrics is a way of providing a measure for the software a person is using [29].

2.1 Usage of Software Metrics

Out of all the entities of a software that can be measured, software metrics can be applied to different stages in the software development life cycle. In the beginning phases of a project, metrics can be used for estimating the cost and requirement of different resources. Then when the design phase begins, metrics can be used for measuring the software size and function points. For example, one of the most basic metric for measurement of software size, is the lines of code. Further down the line, different metrics from design structure can be used to help in the testing of the software. Also analyzing the source code can help improve the cost of maintenance.

With the help of software metrics, a developer can find areas of the code base that are performing poorly by running the different metrics on it, and can make the necessary changes based on the results from running the metrics.

2.2 Classes of Software Metrics and Attributes

Software metrics can be grouped into different classes depending on which measure was taken during which phase. They are broadly grouped into :

1. **Process metrics** - measure of software related activities.
2. **Product metrics** - measures artifacts, deliverables and documents resulting from processes.
3. **Resource metrics** - measure software resources.

In each class of entity, we distinguish between internal and external attributes :

- Internal attributes of a product, process or resource are those that can be measured purely in terms of the product, process or resource itself. In other words, an internal attribute can be measured by examining the product, process or resource on its own, separate from its behavior. [12]

- External attributes of a product, process or resources are those that can be measured only with respect to how the product, process or resource relates to its environment. Here, the behavior of the process, product or resource is important, rather than the entity itself. [12]

2.2.1 Process Metrics

Process metrics are those metrics that relate to the measurement of software process attributes. They are measured to inform on the cost, time span, usefulness and efficiency of the the software development activities. Some of the measures that are included are effort estimation, quality of the process, estimating the duration and assessment of the cost of the development process. These metrics can be studied at any stage of the software development process depending on the desired measures.

Source Line of Code (SLOC) metrics, which can be used to get an approximation of the effort needed to develop a code base, is a good example of process metrics.

2.2.2 Product Metrics

Product delivered to the customers does not only include product deliverable but also it includes other artifacts such as prototypes and documentation.

All of the process outputs can be measured in terms of quality, size. In other words these metrics measure both of the product's internal as well as external attributes.

External attributes

External product attributes depend on product behavior and environment that influence the measurement. Examples include attributes such as : *integrity, usability, testability, reusability, operability, portability, efficiency* . [12]

Internal attributes

Internal attributes includes size of the software, complexity, the level of testability, bugs and correctness.

2.2.3 Resource Metrics

Resource metrics pertain more to the managers of the projects like estimating the number of developers needed, number of computers or other physical resources. Measuring these metrics help managers understand and control the process better.

2.3 Software Code Metrics

Software code metrics are those metrics that are countable directly from the source code. Most of the code metrics that we know of today fall into this category of code metrics.

Code metrics is a set of software measures that provide developers better insight into the code they are developing. By taking advantage of code metrics, developers can understand which types and/or methods should be reworked or more thoroughly tested. Development teams can identify potential risks, understand the current state of a project, and track progress during software development [34].

Software code metrics can be divided into the following sub-categories based on the different measures that are performed on the code [2] :

- Quantitative metrics
- Metrics for program flow control complexity

- Metrics for data control flow complexity
- Metrics for control flow and data flow complexity
- Object-oriented metrics
- Safety metrics
- Hybrid metrics

2.3.1 Quantitative Metrics

The metrics that fall under this category are those that measures the quantitative characteristics of the code. Attribute such as the total number of functions, the total size of the program, total number of comments, total number of operands fall into this category. Some examples of this metrics are - ABC metrics, Source Lines of Code(SLOC), Hanstead metrics. [2]

2.3.2 Metrics for Program Flow Complexity

Metrics for program flow measures the control structure or the flow graph to find the complexity of the program. Quantitative metrics totally overlook the control flow structure of the program, but using the program flow complexity measure helps guide the number of test cases required for the program.

Thomas McCabes's, cyclomatic complexity was the first metrics in this category and has been till date one of the most commonly used metrics [28]. Some other metrics that fall into this category are Hansen's method, Chen topological measure, Woodward's measure, Boundary value etc. [2]

2.3.3 Metrics for Data Control Flow Complexity

By analyzing the program's data control flow this metric can be calculated. The way for calculating this metric is by inspecting the different modules for variables declared in them, and how information flow between those modules [32].

2.3.4 Metrics for Control Flow and Data Flow Complexity

This class of metrics use both the quantitative and program and data flow methods for measuring complexity. Examples of this category of metrics are : Module cohesion, Test M-measure etc.

2.3.5 Object-Oriented Metrics

Most of the metrics for software quality measure were introduced way before object oriented programming was introduced. So in order to effectively measure code written in the relatively new object oriented languages, object-oriented metrics were introduced. Some of the commonly used metrics in this categories are Chidamber and Kamerer's metric.

2.3.6 Safety Metrics

This class of metrics measure the quantity of error detected during code review or during testing. Examples of this class of metrics include errors per thousand code lines.

2.3.7 Hybrid Metrics

As the name suggests, this metrics are a combination of a number of different metrics and there is a weighting policy in place depending on the context in which the metrics is to be used. Examples in this category are :- Cocol's metris, Zolnovskiy etc. [2]

2.4 Finding a Suitable Metric

The aim of this dissertation is to find ways of efficiently distributing the calculation of source code quality metrics. For that purpose, before we can being our experiments, we need to look into a number of metrics, to find out which one would be the best fit for our experimental setting.

So in this section we would be examining in detail, some of the well know metrics. We will not be looking into safety metrics, as they don't fit into our theme of research. For the other metrics, we would be looking into the advantages and disadvantages of each of them.

2.5 Source Lines of Code(SLOC)

Source Lines of Code, also known as Lines of Code, is one of the oldest, most widely used and simplest software metric that is used to measure the size of a computer program, by counting the number of lines, in the text of the program's source code. It is typically used to predict the amount of effort that will be required to develop a program. [36]

It is represented as :

- KLOC : Thousand Lines of Code
- MLOC : Million Lines of Code
- GLOC : Billion Lines of Code

Effective Lines of Code(ELOC), estimates effective lines of code excluding parenthesis, blanks and comments. Once LOC is calculated, we can measure the following attributes [26] :

- Productivity = LOC/Person months
- Quality = Defects/LOC
- Cost = Dollars/LOC

Some recommendations for LOC [26] :

1. File length should be 4 to 400 program lines.

2. Function length should be 4 to 40 program lines. At least 30 percent and at most 75 percent of a file should be comments.

2.5.1 Advantages

- Since Lines of Code is a physical entity, there is scope for automating the counting process by developing utilities for calculating the LOC in a program.
- SLOC serve as an intuitive metrics since it can be seen and the effect of it can be visualized.

2.5.2 Disadvantages

1. This metric lacks accountability, in that the productivity of the project can't be measured using only results from the coding phase.
2. Difference in syntactic structure between two programming languages means that, even if both implement the same functionality, the LOC may widely differ between the two programs, depending on their syntax and style.
3. LOC is also dependent on the experience of the developer. An experience developer can get a functionality done in far fewer lines, than a novice programmer, even through both of them are using the same language and implementing the same functionality.
4. There is no defined standard on what a line of code means - like whether we should include comments in the count or whether code that spans multiple lines, be counted as single line. It's hard to draw the line, with new languages introduced very often.
5. Another issue, with LOC is that it is not very useful to compare hand written code and auto-generated code.

2.5.3 Takeaways

The disadvantages of SLOC, seems to have out-weighted the advantages gained from this metric. Even through there are undeniable drawbacks to using this metric, still when combined with other metrics it can prove to be quite useful and insightful. If a certain function is huge compared to the average length of the functions in the program, it can serve as an indication, that particular function may need re-factoring.

As for the suitability of LOC, in this dissertation, it is quite trivial to calculate. So SLOC is not considered to be one of those metrics, that fall into the bucket of suitable metrics for efficient distribution.

2.6 Chidamber and Kemerer Object-Oriented Metrics Suite

A Metric Suite for Object Oriented Design, was introducted by Chidamber and Kemerer in 1994 focusing on metrics specifically for object-oriented code [6]. The metric suite proposed by Chidamber and Kemerer is one of the most well known suite of object-oriented metrics. The suite consists of the following six metric :

- Weighted Methods per Class(WMC)
- Depth of Inheritance Tree(DIT)
- Coupling between Object Classes(CBO)
- Number of Children(NOC)
- Response for a class(RFC)
- Lack of Cohesion in Methods(LCOM)

2.6.1 Weighted Methods per Class

WMC is the complexity measure of an individual class. The higher the number of member functions, the more complex that particular class becomes. Also, another side effect of having large number of methods in a class is the higher possibility of impact on the children, since the children will inherit the methods of the class.

$$WMC = \sum_{i=1}^{n} c_i$$

where $c_1, ..., c_n$ is the complexity of the methods [1].

2.6.2 Depth of Inheritance Tree

DIT is a measure of the depth of a class within the inheritance hierarchy, starting from the root all the way down to the lowest child node. The theoretical basis is the measure of how many ancestor classes can potentially affect a class.

A class which is deeper in the class hierarchy, will inherit more methods and hence its behavior is likely to be more complex. Applications can be either top heavy, meaning there are a lot of classes near the root of the tree, or bottom heavy, where there are too many classes near the bottom of the hierarchy.

Deeper trees constitute greater design complexity, as there are comparatively more classes and methods involved. Also a deeper class has more potential of reuse of the inherited methods compared to a class higher in the hierarchy. [6]

2.6.3 Coupling Between Object Classes

Coupling is a measure of the interdependence of two objects. For example, if methods of object A calls methods of object B, then objects A and B are said to be coupled.

[1]It has been left to the developer to decide which complexity measure he/she would like to use.

Some of the key ideas behind CBO are [6] :

- The stronger the coupling between the modules, the harder it is to understand and hence maintain the modules.
- When coupling is high, more testing is required to achieve satisfactory reliability level.
- So, all the above points suggest that low coupling is desirable.

2.6.4 Number of Children

NOC is the number of immediate sub-classes for each class in the class hierarchy. The theoretical basis of this metric, is the measure of how many sub-classes are going to inherit the methods of the parent class. It indicates the breadth of the class.

Some of the key ideas behind NOC are :

- Greater number of children implies greater reuse, since inheritance is a form of reuse.
- If a class has a large number of children, it may require more testing for the methods of the class.
- If a class has a lot of children, then there is a greater likelihood of improper abstraction of the parent class.

2.6.5 Response for a Class

RFC is equivalent to the Response Set, RS for a class. RS of a class is a set of methods that can be executed in response to a message received by an object of that class. As it involved calling methods from outside the class, it is also an indication of the communication taking place between the two classes.

Some of the key ideas behind RFC are [6] :

- If a single message can trigger the calling of a large number of methods, it becomes harder to test and debug the class.
- If a lot of methods can be invoked from a class, the complexity is higher of the class.

2.6.6 Lack of Cohesion in Methods

Cohesion is a measure of the closeness of the operations in a class. It indicates how closely are the local instance variables related to the local methods in a class.

Two different ways of measuring cohesion are :

- Calculate the percentage of methods that use each of the data fields. Lower percentage means greater cohesion of data methods in the class.
- The similarity of methods can be derived from the attributes that they operate on.

Some of the key ideas behind LCOM are [6] :

- Higher cohesion means the class has been properly subdivided.
- Low cohesion increases complexity, which leads to an increase in error during development process.
- Lack of cohesion implies that the class should probably be split into sub-classes.

2.6.7 Summary of OO Metrics

The table below provides a summary of OO metrics that have been discussed so far, and whether a low or high value is more suitable for each of the metrics for good quality of code.

Metric	**Desirable Value**
Weighted Methods Per Class	Lower
Depth of Inheritance Tree	Lower
Coupling Between Object	Lower
Number of Children	Lower
Response for a Class	Lower
Lack of Cohesion in Methods	Lower
Number of Classes	Higher
Lines of Code	Lower

Table 2.1: Summary of the metrics

2.6.8 Disadvantages

- Internal metrics alone can't provide very detailed insight into software quality unless there is proof that they are related to external metrics.
- The validity of some of the metrics suggested by Chidamber and Kemerer, can't be totally verified [7].
- Most of the observations in Chidamber and Kemerer are empirical in nature, meaning those are based solely on observation rather than a theoretical backing. [11].

2.6.9 Takeaways

Most of the metrics of the OO suite consists of calculating the interdependence between different classes. As our motive is to distribute out the processing, the dependence of one entity on the other, would make the distribution very hard.

2.7 ABC Metric

ABC metric provides a measure of software size. This metric was introduced to address the shortcoming of SLOC metric, by Jerry Fitzpatrick.

Most of the imperative languages use data storage(variables) to perform useful work. Such languages have only three fundamental operations: storage, test(conditions), and branching [15].

- (A)Assignment - an explicit transfer of data into a variable.
- (B)Branch - an explicit forward program branch out of scope.
- (C)Condition - a logical/ boolean test.

A fundamental operation is denoted by each of the component, and it is usually written as an ordered triplet of integers. For example, an ABC value of <5,4,6>would mean that the program has 7 assignment count, 4 branching count and 6 condition count.

A single ABC size value can be obtained by finding the magnitude of the ABC vector using the following formula [15] :

$$|ABC| = sqrt((A * A) + (B * B) + (C * C))$$

The value calculated for ABC metric for a single method is the best when it's less than or equal 10. It is still considered good when the value is less than 20, but once it crosses 21, it indicates that the program needs refactoring. It is unacceptable to have a program with value more than 61.

2.7.1 Advantages

1. ABC metric is not very hard to compute.

2. The vector form for the metric retain more information than the scalar value calculated by the equation.

3. The developers style of programming doesn't affect the value derived from this metric.

2.7.2 Disadvantages

1. This metric is not a true reflection of the actual size of the program but it only shows the working size.

2. This metric has not gained widespread use for measurement of program sizes.

2.7.3 Takeaways

Similar to our previous metrics, SLOC, ABC metric is quite trivial and would not be a good fit for the purposes of our dissertation.

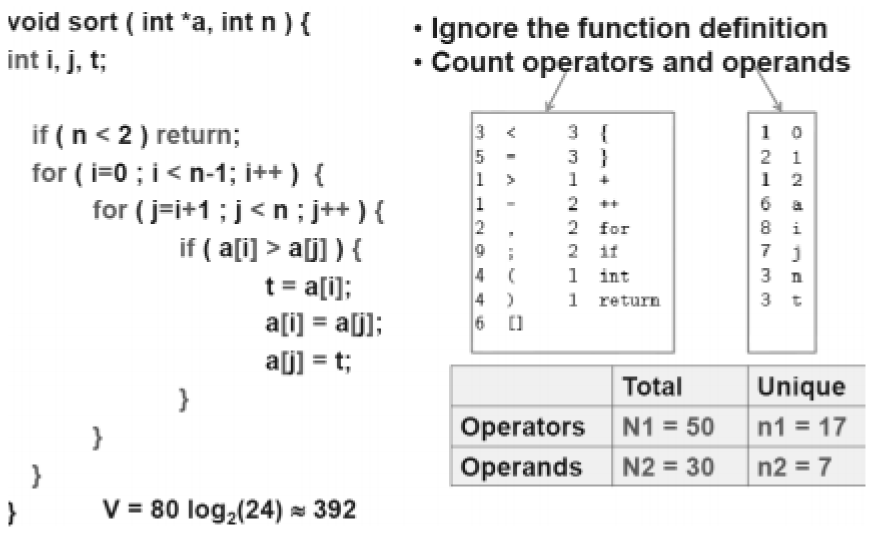
2.8 Halstead Measures of Complexity

This suite of metrics were introduced by Maurice Halstead in 1977 and is also known as Halstead software science or as Halstead metrics [5].

The idea behind this metric, is that a particular metric should be independent of their execution on a specific platform but should be dependent on the implementation or expression of the algorithms. The computation of the metrics is therefore done statically from the code. [20]

Halstead metrics are based on the following indices :

- n1 - distinct number of operators in the program.
- n2 - distinct number of operands in the program.
- N1 - total number of operators in a program.
- N2 - total number of operands in a program



```
void sort ( int *a, int n ) {
int i, j, t;

  if ( n < 2 ) return;
  for ( i=0 ; i < n-1; i++ )  {
       for ( j=i+1 ; j < n ; j++ ) {
              if ( a[i] > a[j] ) {
                     t = a[i];
                     a[i] = a[j];
                     a[j] = t;
              }
       }
  }
}
```

$V = 80 \log_2(24) \approx 392$

- Ignore the function definition
- Count operators and operands

3	<	3	{
5	=	3	}
1	>	1	+
1	-	2	++
2	,	2	for
9	;	2	if
4	(	1	int
4	)	1	return
6	[]		

1	0
2	1
1	2
6	a
8	i
7	j
3	n
3	t

	Total	Unique
Operators	N1 = 50	n1 = 17
Operands	N2 = 30	n2 = 7

Figure 2.1: Example of calculation of Halstead Metric

Based on these notions of operators and operands, Halstead defines a number of attributes of software [30]:

2.8.1 Program Vocabulary

It is the count of the number of unique operators and the number of unique operands.

$$n = n1 + n1$$

where

$n1$ = the number of distinct operators in the program,

$n2$ = the number of distinct operands in the program,

and

n = the total number of distinct tokens or vocabulary of the program.

2.8.2 Program Length

It is the count of total number of operators and total number of operands.

$$N = N1 + N1$$

where

$N1$ = total number of operators in the program,

$N2$ = total number of operands in the program

and

N = program length or size of the program.

2.8.3 Program Volume

It is another measure of program size, defined by :

$$V = N * log_2(n)$$

where

N = program length, calculated from Halstead formula for program length,

n = program vocabulary, calculated from the formula of Halstead formula for program length, and

V = volume of the program.

2.8.4 Difficulty Level

This metric is proportional to the number of distinct operators(n_1) in the program and also depends on the total number of operands(N_2) and the number of distinct operands (n_2). It is denoted as :

$$D = (\frac{n_1}{2} * (\frac{N_2}{n_2}))$$

2.8.5 Program Level

It is the inverse of the level of difficulty. Also a low level program is more prone to errors than a high-level program

$$L = \frac{1}{D}$$

2.8.6 Programmign Effort

Halstead metrics can be used as a model of software development process. The effort required to develop a software is given by the equation :

$$E = \frac{V}{L}$$

2.8.7 Development time

The development time can be obtained from the formula :

$$T = \frac{E}{S}$$

where

E is the total effort and S is the Stround number [2]. Value of S is usually taken as 18 for calculating these metrics.

2.8.8 Advantage

- Halstead is simple to calculate, it doesn't require complex analysis of program structure
- A big advantage is that, it can be used to measure the overall quality of programs.

2.8.9 Disadvantage

- It becomes difficult to distinguish between operators and operands in the model used by Halstead.
- The analogy between complexity increase and volume increase is vague in that the metrics does not specify which values makes the program complex.

[2]Stround number : Is a number ranging from 5 to 20, which is the number of distinct things a human brain can perform at a time.

- The Halstead metrics is not very helpful in measuring the structure of the program or the interactions between the modules.
- Stround's number used by Halstead, which was an assumption maybe incorrect [27] .

2.8.10 Takeaways

Halstead's suite of software metrics provide quite an array of metrics for calculating different facets of a program. Halstead metrics can be used with other quantitative metrics such as SLOC, for calculating program complexity.

2.9 Selection of Metrics

As can be seen from the discussion above, a large number of code quality metrics are there, with each one serving different purposes. The selection of a particular metric, will depend upon who the user is and what is his/her requirement. Metric selection may be controlled by the user type/requirement. If the users are developers, they would mostly need metrics that would help them in refactoring complex code components, help in developing test cases, help in analyzing the quality of code etc. For managers the emphasis would be on measuring maintability, portability, quality etc.

But none of the code quality metrics explained in this chapter, satisfy the research objectives we defined in the beginning of the dissertation. Hence we wouldn't be using any of the metrics defined in this section. Although, we were not able to select an acceptable metrics for our problem, we did learn about the properties of code metrics, that we would like in our acceptable metric to have. In the next chapter, we would dive deeper into the metric, that is an appropriate fit to our problem space.

Chapter 3

McCabe Cyclomatic Complexity

As the name of the chapter suggest, we have chosen McCabe's cyclomatic complexity, as the key metric for the purpose of our dissertation. We will do an in-depth study of the properties of the metric, together with why it would be a suitable metric for the problem area we are trying to address in this thesis.

Complexity in general is defined as "the degree to which a system or component has a design implementation that is difficult to understand and verify". Complexity metric is used to predict critical information about reliability and maintainability of software systems [33].

Thomas J. McCabe, in 1976 put forward cyclomatic complexity, which has been one of the most widely used metrics till date. This metric is independent of language as well as language format and is a quantitative metric used for indicating the complexity of the program using Control Flow Graph(CFG) of the program. In other words, it measures the number of linearly independent paths through the program's source code.

McCabe also stated that, *the complexity of a program is independent of physical size, that is adding and subtracting functional statements leaves the complexity unchanged and the complexity depends on the decision structure of a program* [28].

The control flow graph of a program consists of nodes, which correspond to commands in the program and the directed edges connecting two nodes, if the command corresponding to the second node is executed immediately after the command that the first node corresponds to. There is a wide range of application of cyclomatic complexity starting from functions, classes, all the way upto module and project level.

Cyclomatic complexity can also give an idea about the number of test cases that needs to be written for a program. Alternatively, the cyclomatic complexity of a program is equal to the number of test cases.

3.1 Definition

Cyclomatic complexity is the number of linearly independent paths within a program. If a program does not contain any decision points(conditionals), then the complexity of the program would be 1, since there is just a single control path through the program. For example if the code has a single IF statement, the complexity of the program would be 2, since there would be two distinct paths through the program, one when the expression in the IF clause evaluates to true and another one when it is FALSE.

Therefore, the cyclomatic number V(G) of a graph G, is defined as [28] :

$$v(G) = e - n + 2p$$

where

$v(G)$ is the cyclomatic complexity of any graph G.

e is the number of edges of the graph of a program.

n is the number of vertices/nodes of the graph

p is the number of connected components[1] of the graph.

The above formula maybe thought of as calculating the number of linearly independent cycles[2], that exists in the graph. Also cyclomatic complexity represents the number of decision points at machine-level instructions. So in high level languages, where conditionals involve multiple predicates, then those predicates must be taken into consideration while calculating cyclomatic complexity.

[1]A directed graph is called strongly connected if there is a path in each direction between each pair of vertices of the graph

[2]Linearly independent cycles are those that do not contain other cycles within themselves

Let us look at some examples to see how cyclomatic complexity can be calculated from graphs.

Example #1:

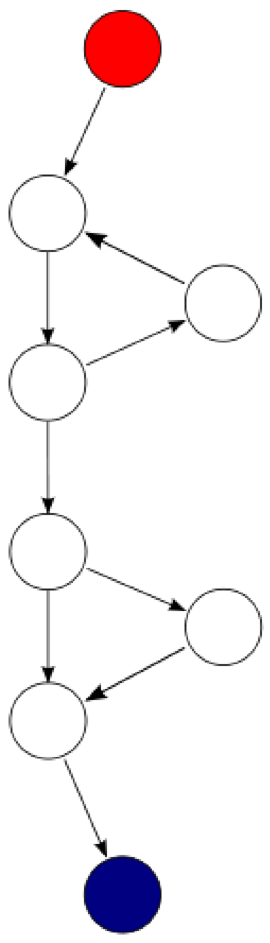
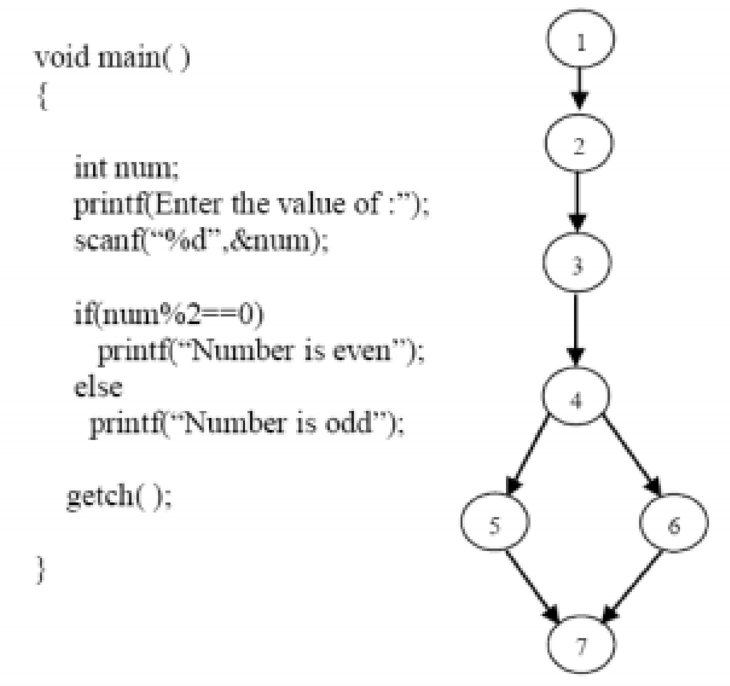


Figure 3.1: Control flow graph for a simple program

In the graph above, red denotes the starting point and blue is the end point. It has 9 edges, 8 nodes and there is one connected component. So the cyclomatic complexity for the program is :

$$v(G) = e - n + 2p = 9 - 8 + 2*1 = 3$$

Example #2:



```
void main( )
{

    int num;
    printf(Enter the value of :”);
    scanf(“%d”,&num);

    if(num%2==0)
      printf(“Number is even”);
    else
      printf(“Number is odd”);

    getch( );

}
```

Figure 3.2: Control flow graph of the main function on the left

The C program on the left maps to the Control Flow Graph on the right. The CFG has 7 edges, 7 nodes and 1 connected component. So the cyclomatic complexity of the program would be:

$$v(G) = e - n + 2P = 7 - 7 + 2*1 = 2$$

3.1.1 Properties

Some of the properties that cyclomatic complexity number holds are :

1. v(G) is always greater than or equal to one. In other words the minimum value of cyclomatic complexity is 1 and can't be negative as there is always a single flow through the graph.

2. Another way to interpret CC, is that it denotes the maximum number of linearly independent paths in a graph G.

3. If there is any deletion or insertion of functional statements in a graph G, it doesn't affect the value of V(G).

4. There is only one path in the graph if v(G) = 1.

5. If there in an insertion of a new extra edge into the graph, the cyclomatic complexity, V(G) increases by one compared to the previous value.

6. The cyclomatic complexity depends just on the decision structure of the graph, G not on any other things.

3.2 Categories of Cyclomatic Complexity

The values derived from the measure of cyclomatic complexity can be used to imply the relationship between the complexity of the code base and complexity value.

Cyclomatic Complexity	**Code Complexity**
1-10	Simple and easy to understand programs.
11-20	More complex, risk is moderate. Testing becomes difficult due to the large number of branches.
21-50	Complex, high risk. Testing can be difficult and requires a lot of effort.
50+	Untestable, very high risk, unmaintainable code.

Table 3.1: Ranges of cyclomatic complexity

3.3 Cyclomatic Complexity Variations

Next let us look at some variations to cyclomatic complexity, and how we can generalize it a bit more.

3.3.1 Connected Components Variation

In McCabe's cyclomatic complexity formula, p denotes the number of connected components. In all the Control Flow Graphs that we have encountered until now, there is a single entry point and a single exit point, with all the nodes reachable from the entry point and the exit node reachable from all the other nodes. However there maybe situations when there is a main program with many subroutines being called from the main program. A main program M, with two sub programs A and B, together with their Control Flow Graphs is shown below.

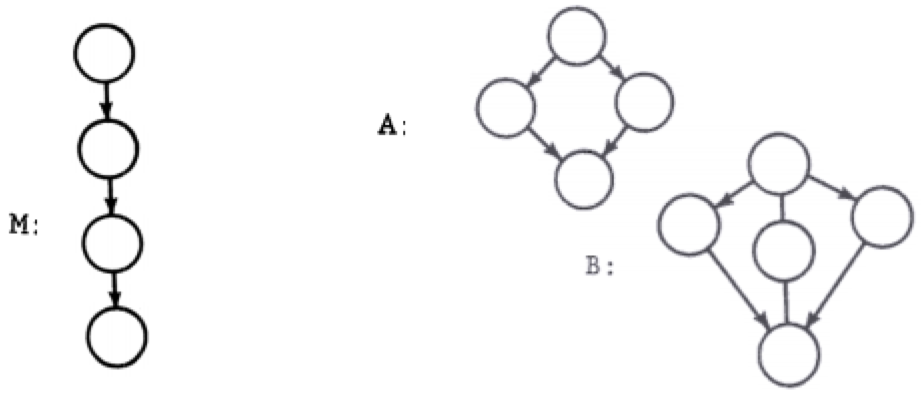


Figure 3.3: Multiple components in a single program

So the graph above consists of 3 connected components. Since p is 3, the cyclomatic complexity of the program with 13 vertices and 13 edges can be calculated to be :

$$v(C) = e - v + 2p = 13 - 13 + 2 * 3 = 6$$

So for programs containing multiple hierarchical subroutines, the method described above can be used to compute the complexity.

If you look closer at the value calculated above, it would become evident that, the result would be exactly the same if we were to calculate the complexity measure on each of the components separately and add them together to derive the final result. Hence the formula can be rewritten as :

$$v(C) = e - n + 2p = \sum_{1}^{k} e_i - \sum_{1}^{k} n_i + 2k$$

$$= \sum_{i}^{k}(e_i - n_i + 2) = \sum_{i}^{k} v(C_i)$$

where,

e_i = number of edges in the i^{th} component

n_i = the number of nodes in the i^{th} component.

C_i = is the complexity of the i^{th} component.

So the complexity over a collection of control graphs, can be defined as a summation of the complexity of the individual components.

3.3.2 Variation with Regards to Program Syntax

The formula that we have seen until now, by taking the connected components of a graph into consideration, can become cumbersome for the programmer sometimes, when the graph becomes very large, or the hierarchy tree get huge. In order to solve the issues associated with those two cases, two simplifications are possible with regards to :

- Number of predicates
- Number of regions

Number of Predicates

Mills in his paper [31], proves that in a structured program[3], if the number of functions is represented as θ, the number of predicates as π and the number of nodes as γ, then the number of edges, e would be :

$$e = 1 + \theta + 3\pi$$

Also the number of nodes can be represented as :

$$n = \theta + 2\pi + 2$$

Assuming that the value of p is 1 and substituting the values above in $v = e - n + 2$, we get

$$v = (1 + \theta + 3\pi) - (\theta + 2\pi + 2) + 2$$

$$= \pi + 1$$

So, it can be shown that the cyclomatic complexity of a program, which is structured, is equal to the number of predicates in the program plus one.

[3]Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures, for and while loops in contrast to using simple tests and jumps such as the goto statement which could lead to "spaghetti code" which is difficult both to follow and to maintain [24].

In the graph below, the cyclomatic complexity is computed to be 8, as there 7 branching points.

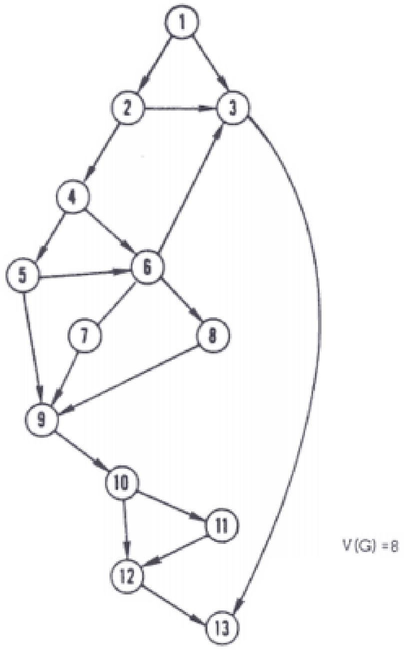


Figure 3.4: Complexity calculation from the number of branching statements

The above corollary works only if there is just a single predicate in the expression of the form `IF C1 THEN C2`. If there are multiple predicates, then it should be counted as contributing two to the cyclomatic complexity measure.

For example a compound predicate of the form `IF C1 AND C2` in machine level instructions would be interpreted as `IF C1 THEN IF C2 THEN`. Hence for multiples conditional, the number increases in proportion to the number of predicates.

Number of Regions

The next simplification to the cyclomatic complexity calculation comes in the form of observing the control graph for number of regions and it makes use of Euler's Formula [22]. Euler's Formula states that if a graph G has e edges, n vertices and r regions, then

$$n - e + r = 2$$

Just by rearranging the term in the above equation, we get r = e - n + 2. If we compare it to the formula for CC, we can conclude that the number of regions in a planer graph is infact equivalent to the CC of that graph.

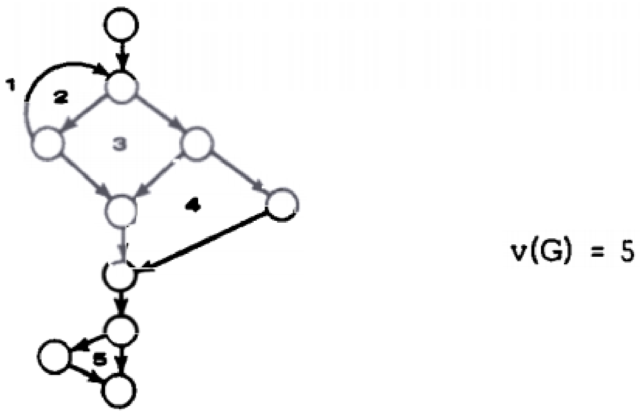


Figure 3.5: Cyclomatic complexity calculation by observing the number of regions

3.4 Reasons for chosing Cyclomatic Complexity

As we have learnt from the discussions in this chapter, the calculation of cyclomatic complexity doesn't have any dependence on any other entities except for the contents in a file. Hence it would be easy to distribute out the processing to several nodes,since the processing is isolated to just a single file. So there would not be any complex distribution to take care of while distributing the files. Also, the actual implementation of cyclomatic complexity is not very computationally intensive. It does involve a bit of work, but it is not very hard on the CPU.

Although cyclomatic complexity has a lot of advantages, it has some disadvantages associated with it as well. Some of it are listed in the next section.

3.4.1 Limitations of Cyclomatic Complexity

Some of the drawbacks associated with measuring code complexity solely on the basis of cyclomatic complexity are :

- It focuses exclusively on control flow as the only source of complexity within a function.
- A expert programmer who has spent a lot of time reading code knows that complexity comes from more than just control flow in the program.
- The cyclomatic complexity number does not properly account for the different numbers of acyclic paths through a linear function as opposed to an exponential one.
- Cyclomatic complexity treats control flow mechanism the same way. But according to Nejmeh[35], some structure are inherently harder to use and properly understand.
- Cyclomatic complexity doesn't account for the level of nesting within a function. For example, three sequential if statements are considered the same as three nested if statements, although for a programmer three nested ifs are much harder to understand and hence should be considered more complex.

3.5 Summary

In this chapter we introducted McCabe's cyclomatic complexity, a very widely used and popular code quality metric. We discussed the many variants for calculating cyclomatic complexity and its advantages and disadvantages.

Chapter 4

Design

In this section we will be taking a look at the design of the different underlying systems used for analyzing code repositories for calculating cyclomatic complexity. But before we begin the section on the design of our system, in the next few sections, we would provide an overview of some of the technologies that have been used in the design of the system, so that when we encounter them later in the chapter, it would be easier to understand the terms and design pattern decisions.

4.1 Overview of MapReduce

MapReduce is a programming model for processing huge volumes of data. Traditionally, most of the systems that were built followed the shared everything architecture whereby, all of the computer resources like CPU, memory etc were shared between many client programs. The problem with this approach was that, even through issues with the system could be immediately isolated, it suffered from the drawback that everything was centralized. So if any computationally intensive task had to be done, all the data would have to be moved where the actual processing was located. This would most often choke the network bandwidth and hence didn't prove to be a very feasible solution as the volume of data got bigger.

Some of the drawbacks of the shared everything architecture are :

- If there is a need for scaling, then the only way out is to scale up, that is put more resources- processing power, memory, hard disk into a single computer.

- All the processors share the same memory address space.
- Sharing of the same system bus among all the processors, may result in the choking of the bandwidth for memory access.
- Because all of the processes are accessing the same data, complex synchronization is required between the processes, for data integrity and durability.

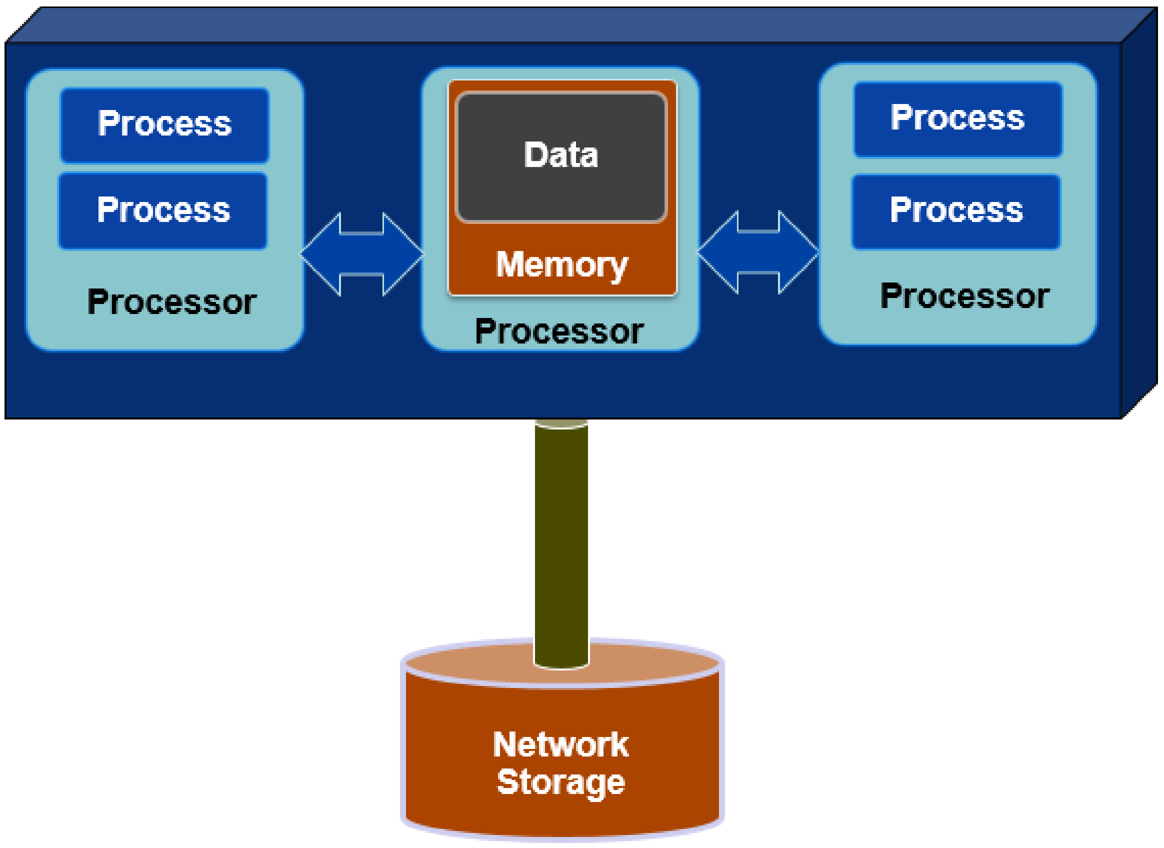


Figure 4.1: Shared everything architecture

- The programming model can get quite complex because of all the complicated synchronization required to keep the data valid.
- Data exchange also need synchronization.

- Since everything is bundled into a single box, failure can be quite expensive. Failures of a single component can have possible ramifications on the whole system, and can bring the system down.
- Since terabytes of data can't possibly be stored in a single computer, data needs to be stored in Storage Area Network(SANs).
- Data needs to be moved across the network from the SAN to the processor nodes, when any processing needs to be done.
- This model doesn't scale for large volumes of data.[38]

MapReduce is a distributed, scalable and fault tolerant system, which follows the shared nothing architecture. Here all the processing entities are physically separate from each other, and data need not to moved around in the network. Instead the processing is moved to the node containing the data. It has made it possible to process terabytes of data, without crippling the network infrastructure.

The core idea behind MapReduce, which was first introduced at Google by Jeff Dean [9], is all the processing takes place on data represented as key-value pairs. The computation is done mostly in two phases, although the second phase is optional :

- Map phase : It involves taking in key-value pairs and outputting intermediate key-value pairs.
- Reduce phase : The reduce phase takes as input the intermediate key-value pairs produced by the map phase, performs some aggregation and produces results also in the form of key value pairs.

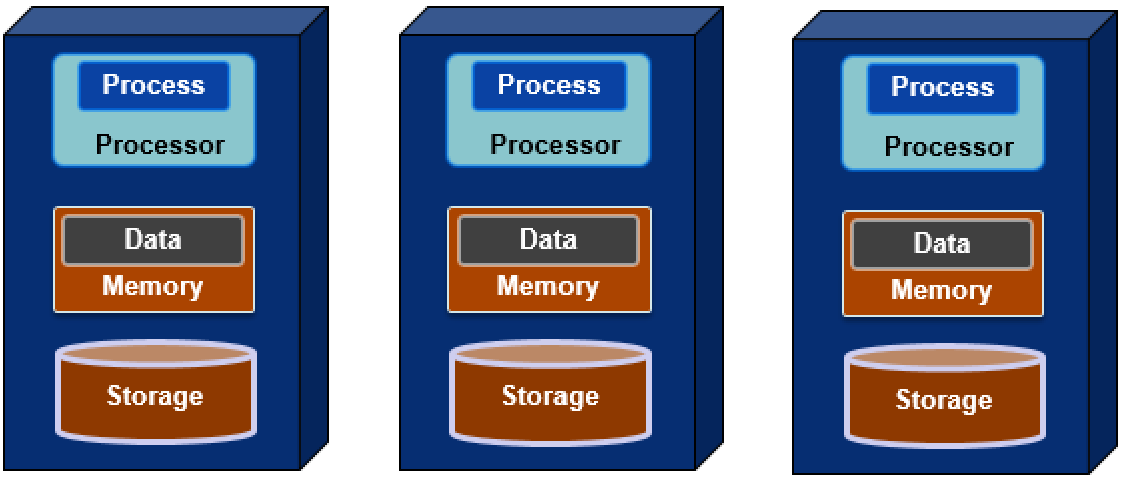


Figure 4.2: Shared nothing architecture

Some of the benefits that this model of data processing brings to the table are :

- It follows the shared nothing architecture meaning that data is local and doesn't have to move across the network.
- There is no requirement for complex synchronization among the nodes in this model.
- The framework has been built with keeping failure in mind. Redundancy has inherently been built into the system, with the capability of storing multiple copies of data.
- There exists no single data storage point of failure since, the processing has been distributed out to all the nodes in the system, and even if a single node fails, it does not hamper the working of the other nodes in the system.
- The data is sharded(partitioned) among the nodes of the system.

4.1.1 MapReduce Architecture

As the name suggests, jobs in map-reduce are primarily composed of two steps - the map side and the reduce side. The high level view of a work-flow looks something like this:

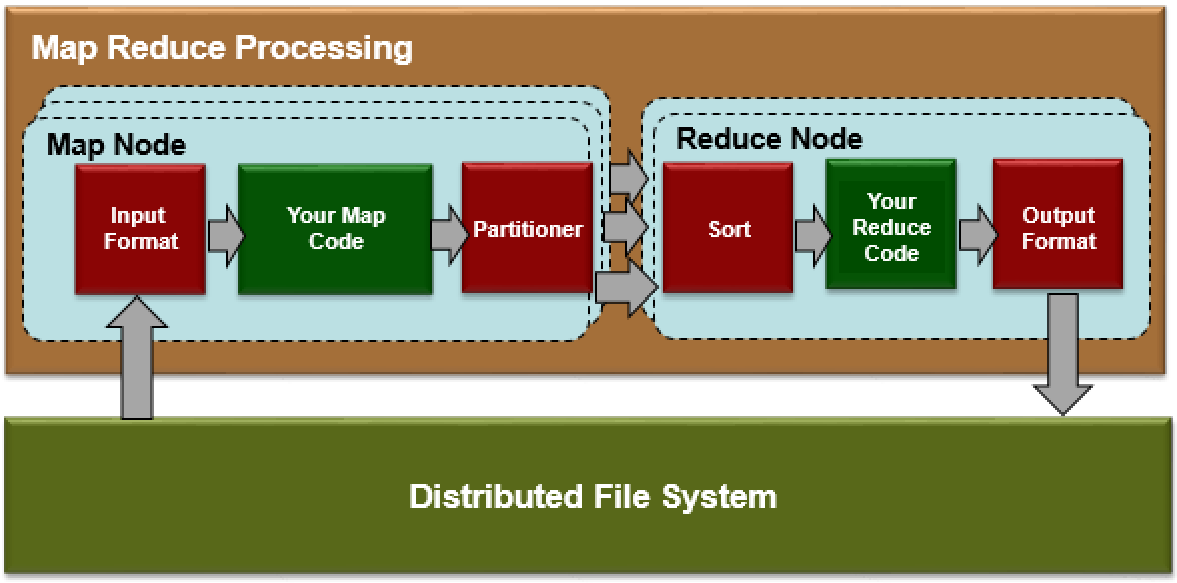


Figure 4.3: Overview of a MapReduce job

4.1.2 Map Side Task

Whenever a job is executed the following steps occur on the mapper side of the job :

1. The map side fires off a number of mappers in parallel, across all the nodes that contains the data on which the job is to be run.
2. Each mapper, then acts on the data in a block and splits each block into individual lines of text whenever the newline character (\n) is encountered.
3. Inside each mapper program, there is a map function, which then takes each line as input in the form of key-value pairs.
4. The map function then produces the result also in the form of key-value pairs.

There maybe cases when duplicate key-value pairs are emitted from the map side, as we will soon see in an example, but that is perfectly fine. Once all the mappers are finished with their jobs, the key-value pairs are sorted according to their key and are passed on to the reducers.

4.1.3 Reduce Side Task

Once the map side of the MapReduce job is done with all the processing, the reduce side of the job starts. By default there is just one reduce task, but the configuration file can be changed to accommodate any number of reducers as required.

After the sorting phase, in the mapper, there is another step called the shuffle phase where, the sorted data are applied to a Partitioner, which is basically a hash function of sort. The Partitioner would take the hash of the key and the number of reducers to determine which reducer the key should be forwarded to. Also, since the hash of a particular key will always be the same, it guarantees that all the key-value pairs that have the same key, will always be forwarded to the same reducer.

After a reducer gets the key-value pairs that were forwarded to it by the mapper, the reducers all sort their keys so that a single loop will parse all the values of a single key, before moving to the next key. The reduce phase will run the logic defined in the reduce task on each key-value pairs and then the final result is produced. Alternatively, it can be said that the reducer transforms all the key-value pairs that share a common key, into a single key value pair.

Below we show the most basic MapReduce program, that counts the frequency of occurrences of words in a document.

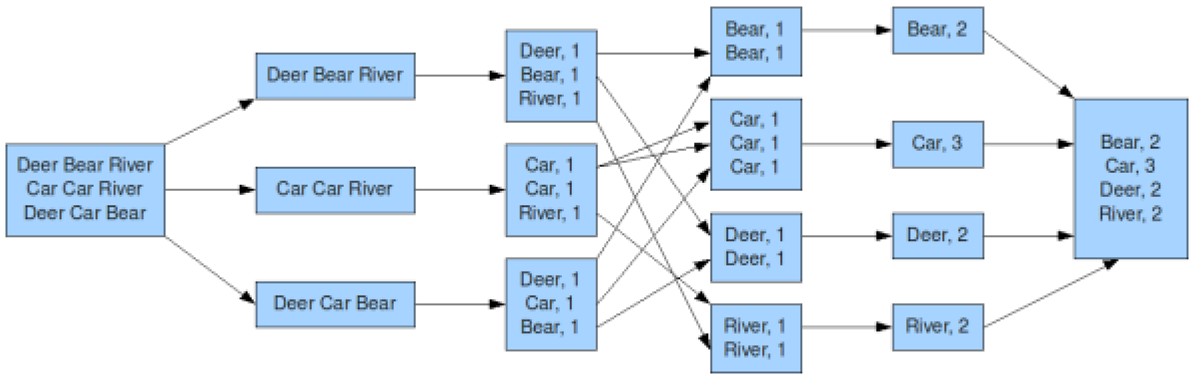


Figure 4.4: The overall MapReduce word count process

4.1.4 Distributed File System

Hadoop Distributed File System(HDFS) is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware [37]. It has been built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern and has been built for sequential access instead of random access. It has been built to store modest number of large files.

A disk has a block size, which is the minimum amount of data that it can read or write. File systems for single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. HDFS, too has the concept of block, but it is a much larger unit - by default it is set to 64 MB. But unlike a file system on a single disk, a file in HDFS that is smaller than the block size does not occupy a full block's worth of underlying storage.

In MapReduce by making the block size large enough, the time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block [3]. The default value is 64MB, but the block size can be increased to accommodate blocks of sizes in the multiples of 64, example 128MB. Some of the benefit of having this block abstraction are :

- A file can be larger than any single disk in the network. So there is no requirement for the blocks from a file to be stored on the same disk space, so they can take advantage of any number of disks in the cluster.
- Blocks fit well for providing fault tolerance and availability. Each block is replicated to a small number of physically separate machines, so if a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client.

4.1.5 Namenode and Datanode

MapReduce model follows the master-slave paradigm, where the master is in charge of all the other nodes in the cluster [37]. In HDFS terminology, the master node is called the `namenode` and the workers are called `datanodes`. The namenode manages the file system namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree.

The datanodes are also where all the data blocks are stored. The namenode maintains the information about the datanode on which the blocks relating to a particular file are stored, so that the information can be quickly serviced to the querying client.

The datanodes are the workhorses of the file system. They store and retrieve blocks when they are told to by clients and namenode and they send occasional heart beat messages to the namenode, notifying that they are still running.

One drawback of this architecture is that, the namenode serves as a single point of failure. If the namenode goes down, all the information related to the cluster is lost and none of the worker nodes would be reachable. In order to deal with possible namenode failures, there is an option for keeping a secondary namenode running on standby at all times. The secondary namenode maintains an exact replica of the information that is stored in the primary namenode, although there exists a lag between the two namenodes and they are never perfectly in sync.

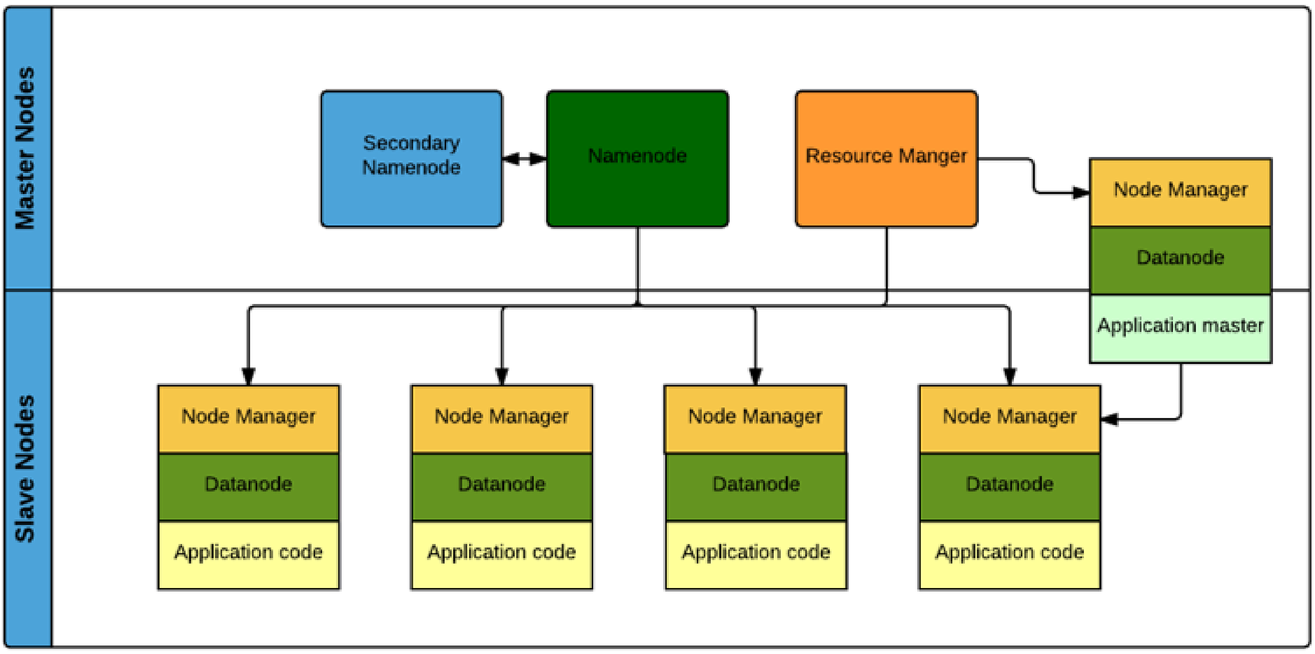


Figure 4.5: The topology of a MapReduce cluster

4.2 Overview of Git

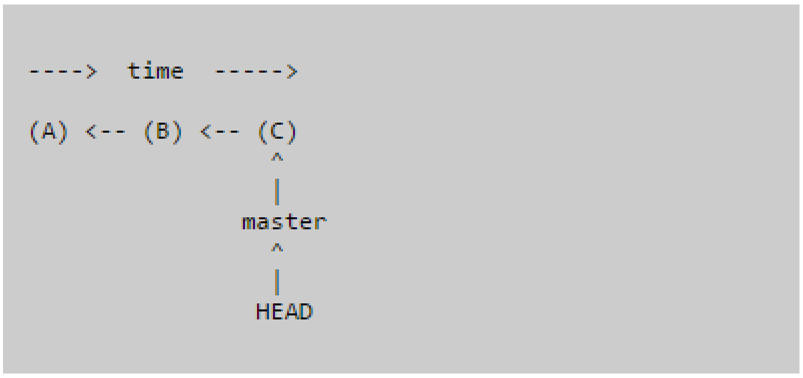
For processing online code repositories, we will be using Github, for pulling in codebases of different sizes and codes written in different languages . Hence in this section we will drill a little bit into the internal structure of Git, how it maintains all the metadata about the code. The purpose of this discussion is to give an overview of the Git, as most of them would be used while performing the experiments.

Git[17] is a distributed version control system, meaning it is used in the management of changes to documents, computer programs or other information. The changes that are made are usually identified by a revision number assigned to them. Unlike classical client-server versioning system, in Git every directory on a computer, is a self contained repository. The computer need not be connected to any central server for accessing the information related to that repository.

The purpose of Git is to manage a project, as changes are made to it over time. The Git repository is stored in the same directory as the project in a sub directory with the .git extension. There can only be one .git directory in the root directory of the project. There is no central repository and the repository is stored in files alongside the project. In Git all the information is stored in data structures called a repository [18], which contains the following :-

- A set of commit objects, which in turn contains.
  - A set of files, that reflect the state of the project at a given point.
  - References to parent commits objects.
  - A SHA1[1] name, that uniquely identifies the commit object.
- A set of references to commit object, called head.

[1]SHA stands for Secure Hash Algorithm



```
---->  time  ----->

(A) <-- (B) <-- (C)
                 ^
                 |
               master
                 ^
                 |
                HEAD
```

Figure 4.6: Git repository with three commits

The figure above gives an illustration of how a git repository would look like with three commits made to it. Here (A), (B) and (C) are the first, second and thirds commits made to the repository. Since each commit has an unique SHA1 ID associated with it, we can refer to it using the SHA1 id. Throughout the project we would be making extensive use of these git fundamentals for carrying out most of the experimentation with different code bases.

4.3 Overview of REST architecture

The Representational State Transfer(REST) style is an abstraction of the architectural elements within a distributed hypermedia system [14] . It is an architectural style where different sets of coordinated components are present, and the components talk to each other over the network only through data exchange, without any of the components having to know about the internal implementation of the other components it is interacting with. This form of design implementation introduces scalability, simplicity and reliability into the system. Systems that conform to the constraints of REST are called RESTFul systems.

REST operates over HTTP[13], which is a request-response protocol. Since REST is implemented over HTTP, many of the design patterns that REST follows, comes from HTTP. Broadly speaking, there is a client, which is a browser most of the time, and there is a web server listening for connections, and for valid requests, it sends back a response.

All resources that are available to be invoked, must be uniquely identified using Uniform Resource Identifiers. Resources also share a uniform interface for transferring the state between the client and server consisting of a constrained set of well defined operations, a constrained set of content type and a protocol that is stateless, cacheable and layered.

The main characteristics that identify a REST-style architecture are :

- The different states and functionalities of the system are distributed among resources that are dispersed.
- There is support for HTTP commands like GET, PUT, POST and DELETE.

4.4 Basic Architecture of the System

The system that we are going to introduce next, is used in the calculation of cyclomatic complexity of code repositories. The source of data for our system is Github [19], which is a web based hosting service for git repositories and supports all the functionalities supported by git and builds some more on top of those. At a very high level, our systems consists of components that pull data from remote Github repositories, performs necessary computations of data and spits out the results derived from the computation.

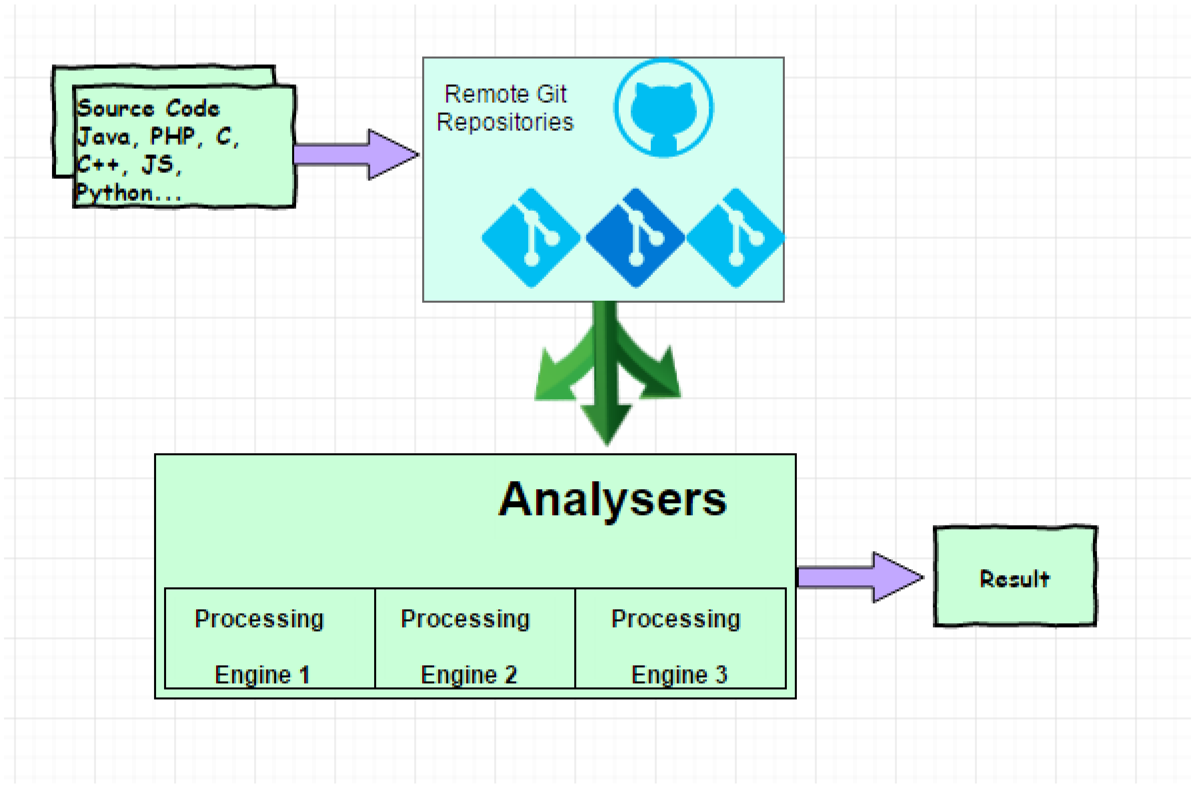


Figure 4.7: Basic overview of the system design

As seen in the picture above, the repositories that are fetched from Github may be written in any high level language, for which the system has been designed to handle. The languages that have support built into the system are Java, PHP, C++, JavaScript, Python and Shell Scripts. After the repositories have been fetched, they are analyzed using different processing engines that are part of the analyser module and after that the results are displayed.

The analyzer module consist of different processing engines, that form the heart of the system and the engines are implemented using the technologies that have been introduced at the beginning of the chapter.

Lets now dive deeper into the different designs of the processing engines that form part of the Analyser module.

4.5 Design 1 : Baseline System

For the baseline design of our system, we have created a single machine application that pulls data from Github repositories and calculates cyclomatic complexity for the files in the code base. This is a single threaded, non-concurrent design, that is used and most of the existing systems that are built to calculate cyclomatic complexity are of this design pattern. We have selected this design, as we would be needing a reference system against which we have to compare how good or bad our other solutions are.

4.6 Design 2 : Distributed Solution, RESTFul

For the next design to our problem, we have created a distributed systems design based on the RESTful architecture pattern.

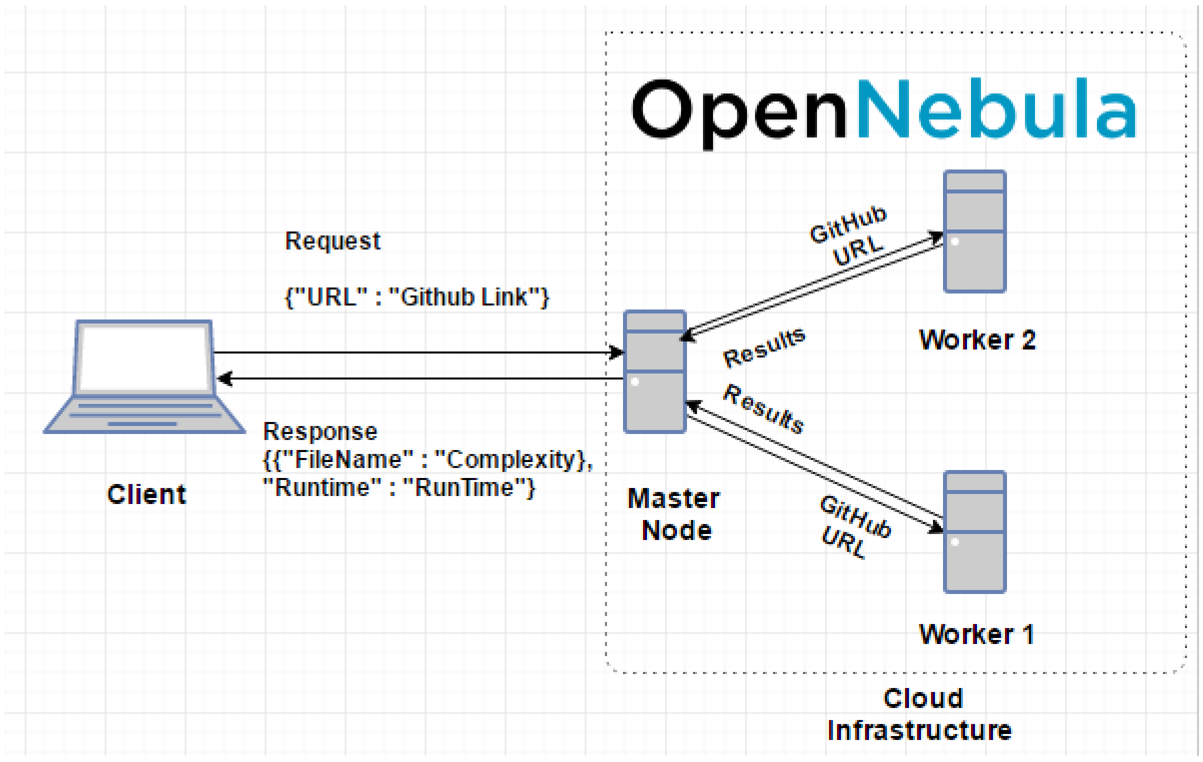


Figure 4.8: Distributed solution implementation using OpenNebula

As seen in the diagram above, we have set up a three machine cluster on OpenNebula[25].Of the three nodes, one node acts as the master and the other two machines work as slaves. Since the system is built to handle REST request, any client on the network can invoke the services of the system. The client sends a request to the master node, with the URL of the Github repository, for which he/she wants to calculate the cyclomatic complexity for. The master node parses the request sent by the client, extracts the Github URL from the message, and passes on the URL to one of the worker machines. Once the worker machine receives a request from the master, it takes the URL and fetches directories associated with the remote repo. It then performs the necessary computation on the downloaded code base, creates the result file and sends it back to the master. The master then forwards the same result to the client, who make the request in the first place.

The decision as to which worker node would be selected is done by the master node based on the work load already allocated to the two worker nodes. The master maintains two queues, one queue is for the requests that comes in and another queue for the work that has been allocated to the workers. So while deciding which worker node to forward the client request to, the master checks the worker queue to see which was the last worker that it had sent the request to. The master sends the request to the other worker, provided the last job that was issued to that worker had been completed. If the last job had not been completed yet, the master en-queues the job into the job queue and waits for the workers to finish the tasks given to them.

The reason for deciding to allocate the processing of a particular repository to one machine instead of distributing out the processing to several machines, is because in open source world, most of the source code, except of open source Linux distributions, which are millions of lines of code, are not greater than a few hundred thousand lines of code. With CPUs being able to perform more than billion instructions per second, and with each computer containing multiple cores, the processing of hundred thousand lines of code would take a very short duration. And if we introduce multiple threading into the picture, its going to bring the running time further down. The problem is not the processing of a single code repository, but the processing of multiple code repositories at the same time, where master doesn't have to wait, for the workers to finish the processing for a single repository. Instead, now the master node is able to assign the processing to repository to either of the two nodes.

4.6.1 Optimization to the Distributed Solution

In this part, we provide an introduction to some of the optimization techniques that have been included in the design of the distributed solution.

Firstly, code repositories are always constantly changing with the addition, deletion and modification to files in the code base. And it is no different for code hosted on Github. What it entails is that with every change that is made to the repo, there might be a corresponding change to cyclomatic complexity of the file.

But changes made to one file is just confined to that particular file, other files in the repository are not affected. But while cloning repositories from Github for a second time , all the files that are associated with a repository gets pulled down, not just the ones that got changed since the last cloning.

This is a huge overhead, in that the cyclomatic complexity would have to be re-calculated for the entire project all over again, and if the repository is millions of lines of code, its going to be painfully slow to calculate the metrics again. So can we do better?

The solution to this problem, is a two pronged approach. The first approach is associated with storing intermediate results. As mentioned at the beginning of the chapter, all the commits in git are uniquely identified by a SHA1 value. So the idea here is to, incur the cost of calculating cyclomatic complexity for the entire project just once - the very first time the project is run and and all subsequent runs are executed only for the files, that have been changed since the last commit ID of the repo.

The second optimization comes in the form of spawning multiple threads for the calculation of cyclomatic complexity. This solution banks on the concurrency provided by the underlying hardware of the machine, to speed up the processing of the files. For each of the files,a separate thread handles the processing.

More details on the implementation of the two optimization techniques will be provided in the next chapter, where the implementation part is explained.

4.7 Design 3 : Distributed Solution using Hadoop MapReduce

In the previous section, we talked about how we implemented the distributed solution using a master- slave REST approach. In that solution, the handling of the distribution of the files to the worker nodes, is done by the master based on the work load of the worker nodes.

In this part, we will look into a different approach, which is Apache's Hadoop MapReduce [16] (version 2.7.) . As discussed in the beginning of the chapter, Hadoop operates under the namenode and the datanode model. So for this part, in the same three node OpenNebula cluster that we had used previously, we installed Hadoop, to do the processing for us. The difference from the previous implementations is that, Hadoop automatically does the distribution of the files to the data nodes without any specific instructions from the user. However it is not built for supporting interactive applications, so there wouldn't be any REST APIs for invoking the operations on the cluster.

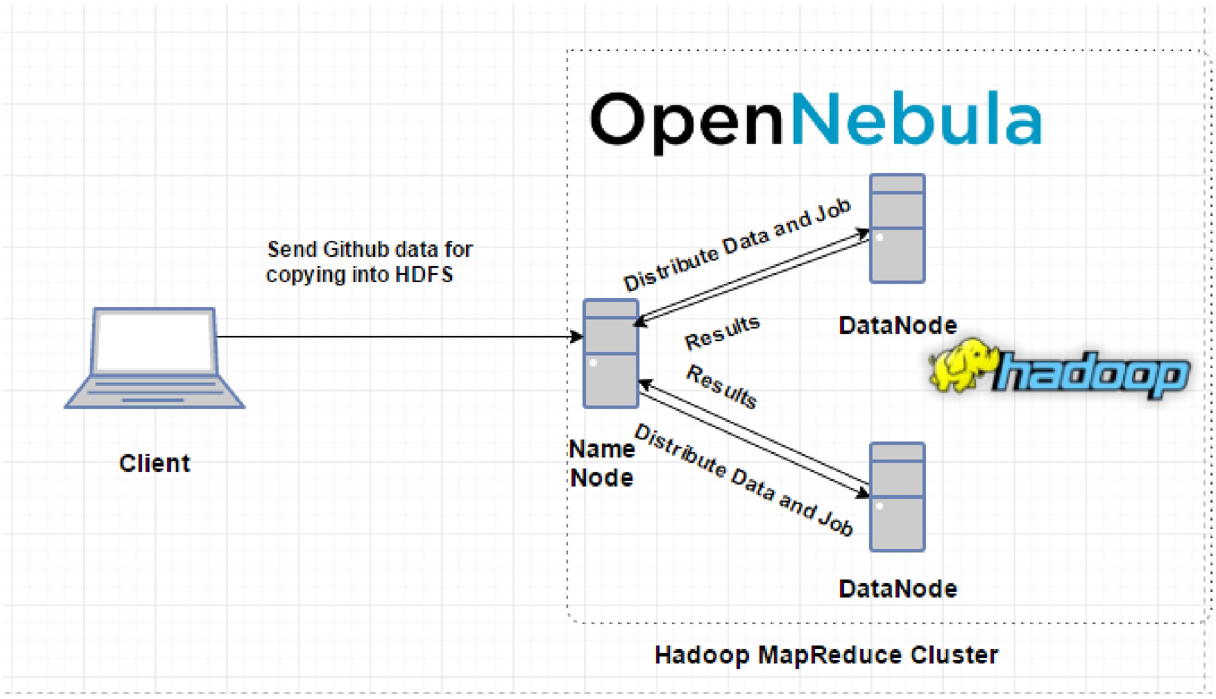


Figure 4.9: Distributing data using Hadoop MapReduce

As we can see from the figure above, this approach is quite different from our previous approach, in that, at a time both of the nodes are involved the computation of cyclomatic complexity for a particular Github project, thus making use of the extra resources that is available to us in both the worker machines. Once the dataNodes are done with the calculation of the complexity, they send the results back to the nameNode, which the user can then view.

4.8 Summary

In this chapter we started with a high level detail about the various technologies that we were going to use in our implementation. Then we discussed the different design strategies that we will be using for solving the problem, starting with the baseline single machine solution, all the way upto Hadoop MapReduce solution.

Chapter 5

Implementation

In the last chapter we outlined the design for our application. In this chapter we will be delving deeper into the implementation detail of the designs previously discussed. We will present the technologies used, together with the completed application that have been developed for experimenting with the topic of the dissertation. Finally we will discuss how the implementation satisfies the research objectives we had set out in the first chapter.

5.1 Technologies used

Our application was implemented in the Java programming language. The reason for choosing Java is, it is very feature rich, mature language with cross platform compatibility and looks native on all platforms, be it Windows or Linux. It is type safe, with very good exception handling capabilities, which is very necessary for a distributed application, otherwise it becomes very hard to debug without proper exception handling. There are a lot of free tools available, starting with excellent IDEs like Eclipse, Netbeans, to very mature web servers - Apache Tomcat and application server like JBoss, Glassfish. Also there is built in support for HTML, SQL, JSON, XML etc and other third party libraries are freely available, for almost any utility that we need.

We will also be using the JGit[8] third party library in Java the implements the core Git version control system. There were other libraries for interacting with Github, but JGit proved to be the most mature and well documented library that has a large user base and provides a lot of the core git functionalities.

Next, as described in the distributed model in the previous chapter, we are using a RESTFul architecture for our application, so we would be needing a web server to deploy our applications to. For that we would be using Apache Tomcat. The reason we decided to go with Tomcat is because it is incredibly lightweight. It has very low memory footprint compared to its peers and is relatively quick to redeploy and start. It is open source and that provides higher flexibility to it that we can tweak the internals to suit our needs. It is extremely secure and it provides a high level of security.

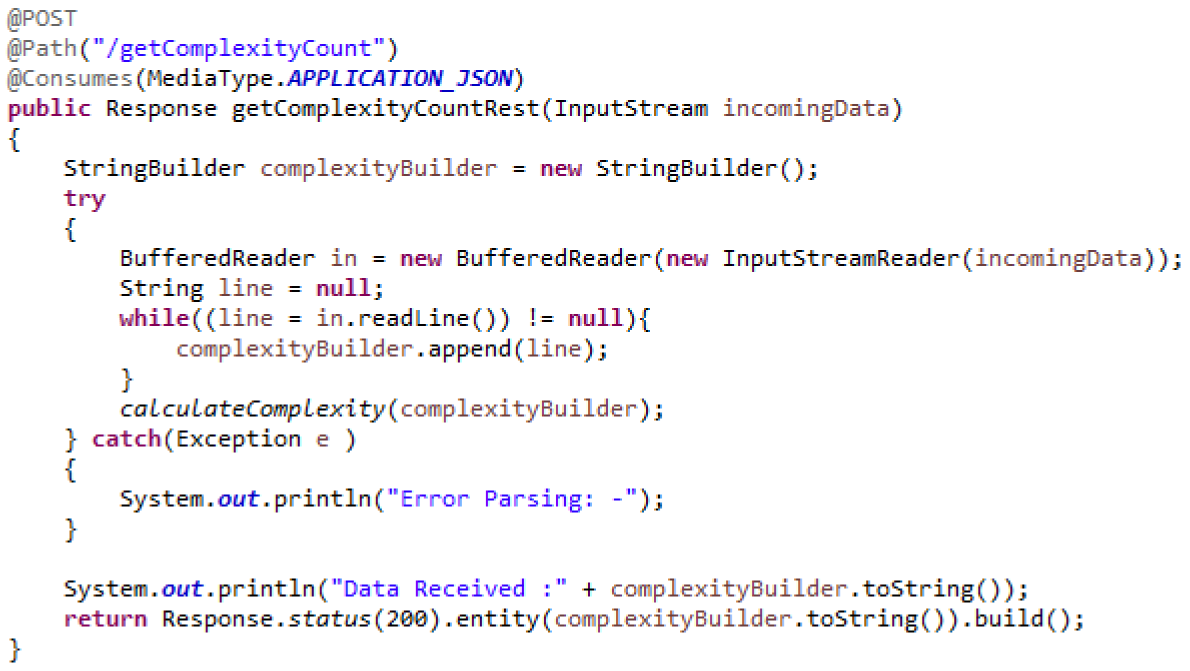
For document exchange, in our design model, we would be using JavaScript Object Notation(JSON), as it is very flexible, compact and easy to use. The message size is very small and allows for convenient serialization and de-serialization of data, which is very useful in a web application that deals with information exchange between the machines in the cluster.

Lastly, we used Hadoop MapReduce, as another implementation strategy for experimenting with our idea. The reason being, MapReduce distribute the data and computation, and since the computation is local to the data, there is very low network overload. All the individual tasks that are part of the job, are independent of one another, so if one fails , the other jobs are not affected. The programming model used is very simple, with the user just having to write the map and reduce tasks, rest all the other aspects are handled internally by the framework.

For implementing our different models, we are using the school of Computer Science and Statistic's cloud platform Opennebula. There we have set up three Linux machines with the following configurations - Debian Wheezy - 64 bit CPU, 4 GB RAM, with 8 GB of storage space.

5.2 REST Implementation

As mentioned earlier, we have a REST server that listens for incoming requests from clients. After deploying the REST server code on Apache Tomcat, the listener method looks something like shown in the next page.



```
@POST
@Path("/getComplexityCount")
@Consumes(MediaType.APPLICATION_JSON)
public Response getComplexityCountRest(InputStream incomingData)
{
    StringBuilder complexityBuilder = new StringBuilder();
    try
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(incomingData));
        String line = null;
        while((line = in.readLine()) != null){
            complexityBuilder.append(line);
        }
        calculateComplexity(complexityBuilder);
    } catch(Exception e )
    {
        System.out.println("Error Parsing: -");
    }

    System.out.println("Data Received :" + complexityBuilder.toString());
    return Response.status(200).entity(complexityBuilder.toString()).build();
}
```

Figure 5.1: REST listener class

As we can see from the snapshot of the code above, there is a listener service deployed on the following URL :

<IP Address of the Server >: <PortNumber >\getComplexityCount

In our case, Tomcat has been started on part 8080 and the IP address of the machine is 10.62.0.3. So the URL of the server through which the service can be accessed is : http:\\10.62.0.3:8080\getComplexityCount.

The JSON file that is sent along with the request would be of the following form.

```
{
"GithubURL" : "https://github.com/arsduo/koala.git",
"Rerun" : "false"
}
```

Listing 1: JSON request format

In the example, we are passing the URL of the Koala repository on Github, and specifying that it is the first run on that particular code base. In this way any valid Github URL can be passed in the JSON file.

After the URL is passed to the worker nodes, those nodes then pull the source code from Github, and measures cyclomatic complexity for each of the files in the project and the returned result to the client is also a JSON file, where several information pertaining to that particular project like total commits made to that repo, total running time, last commit id etc are sent back to the client machine.

```
{"ProjectURL":"https://github.com/arsduo/koala.git",
"Commiter":"Dan <dan@facebook.com>",
"Total project commit :":"995",
"FileInfo":[
{"FileName ":"discover.rb","#Commits":1,"Complexity":1},
{"FileName ":"batch_operation.rb","#Commits":4,"Complexity":24},
{"FileName ":"graph_api.rb","#Commits":47,"Complexity":127},
{"FileName ":"graph_batch_api.rb","#Commits":13,"Complexity":25},
{"FileName ":"graph_collection.rb","#Commits":11,"Complexity":30},
{"FileName ":"rest_api.rb","#Commits":7,"Complexity":13},
{"FileName ":"api.rb","#Commits":30,"Complexity":45},
{"FileName ":"errors.rb","#Commits":8,"Complexity":17},
{"FileName ":"multipart_request.rb","#Commits":2,"Complexity":12},
{"FileName ":"response.rb","#Commits":1,"Complexity":3}],
"Start time :  ":"2016-08-22T17:45:26.669",
"TotalRunTime":"20.977",
"LocalProjectDir :":"koala5672490344333596029",
"Last commit id":" fdf7949363086a3504568137db7f1ae609f28dc9",
"End time  ":"2016-08-22T17:45:47.646"}
```

Listing 2: Result returned back to the client

Above we have shown an example of a truncated version of a JSON reply that is sent back to the client.

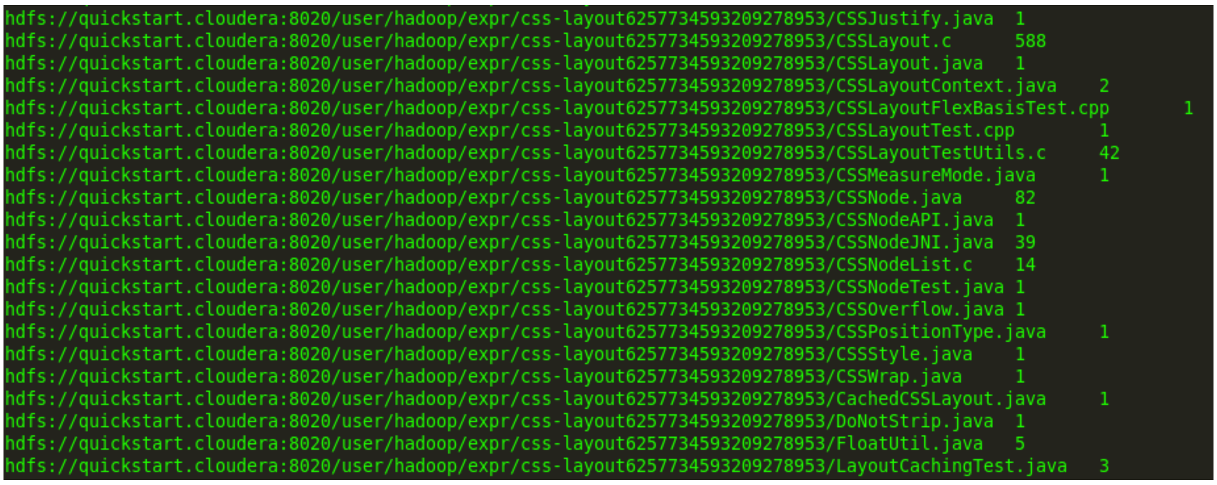
As can be seen from the JSON file above, each run of cyclomatic complexity on a repository stores the last commit id and the local directory name under which the files of the repository are stored. So the next time the client sends in a request, with the URL of the same repository, the program would pick up the last commit id from the stored JSON file, find all files that have changed since the last commit and run complexity metric only on those changed files, which is a huge saving in terms of processing.

5.3 MapReduce Implementation

Any processing done on a file in MapReduce is done in key-value pairs, so for our map class we will extend from the Mapper class, which would look something like : **Mapper <LongWritable, Text, Text, IntWritable >**. So for all the input files, we will treat their line number as key, which would be mapped to LongWritable and all the text corresponding to a particular line number would be treated as value and that would map to Text in the Mapper class. The data type of the output of the mapper class is Text and IntWritable, where Text would be the name of the file and Intwritable would be the cyclomatic complexity corresponding to that file.

Similarly for the reduce class, we would extend from the Reducer class, which would be : **Reducer<Text, IntWritable, Text, IntWritable >**. The input to the reducer would be the output from the mapper; it would just iterate over all the values that are sent by the output of the map task, and produce the output as a unique value for each of key, in our case , it would the cyclomatic complexity of that particular file. And the reduce task would do that for all the files that are part of the repository.

The result of the MapReduce program is written back to HDFS. It can be accessed by navigating to the directory containing the results. The output of a job would look something like this - the location of a file in HDFS followed by the cyclomatic complexity of that file

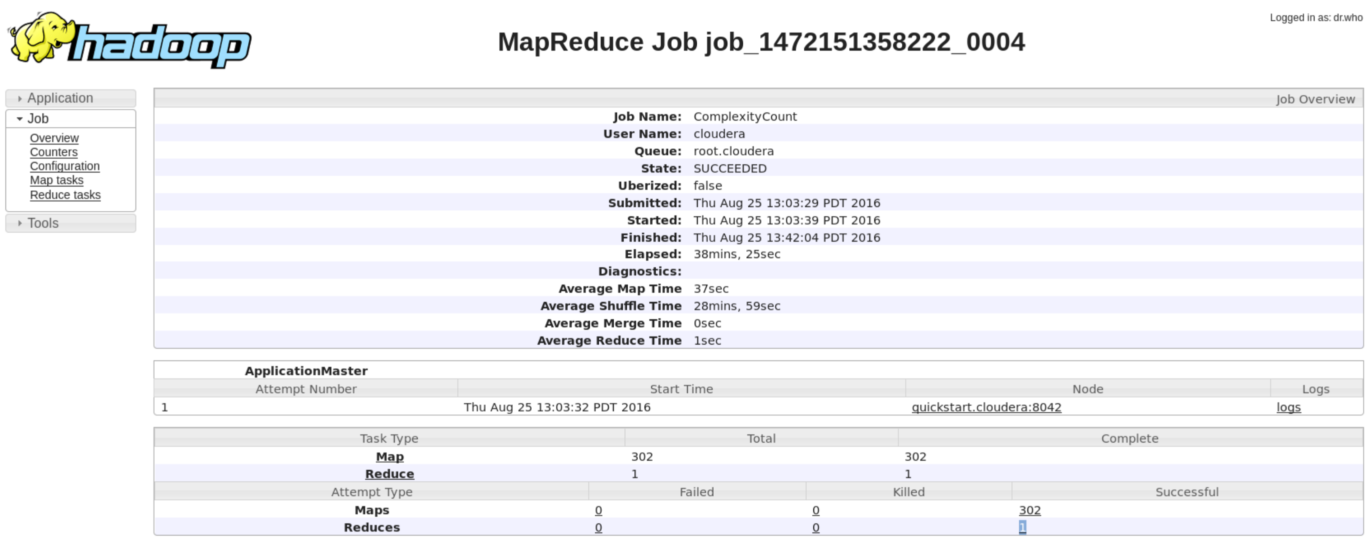


```
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSJustify.java  1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSLayout.c      588
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSLayout.java   1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSLayoutContext.java    2
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSLayoutFlexBasisTest.cpp       1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSLayoutTest.cpp        1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSLayoutTestUtils.c     42
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSMeasureMode.java      1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSNode.java     82
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSNodeAPI.java  1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSNodeJNI.java  39
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSNodeList.c    14
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSNodeTest.java 1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSOverflow.java 1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSPositionType.java     1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSStyle.java    1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CSSWrap.java     1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/CachedCSSLayout.java     1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/DoNotStrip.java  1
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/FloatUtil.java   5
hdfs://quickstart.cloudera:8020/user/hadoop/expr/css-layout6257734593209278953/LayoutCachingTest.java   3
```

Figure 5.2: Format of the output in HDFS

For retrieving information related to the MapReduce job, Hadoop provides a Web Interface service, which by default runs on port 19888, and can be viewed any web browser. There are separate information starting from information about the MapReduce job as a whole, to information about each map jobs and each reduce job down to the minutest detail.

As can be seen from the figure in the next page, it would give us a high level overview about the average time it takes for map task, shuffle, sort and reduce task for the whole job and the total number of map and reduce jobs that were spawned in the duration of the execution of the MapReduce program.



Logged in as: dr.who

MapReduce Job job_1472151358222_0004

Application
Job
Overview
Counters
Configuration
Map tasks
Reduce tasks
Tools

Job Overview

Job Name:	ComplexityCount
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Thu Aug 25 13:03:29 PDT 2016
Started:	Thu Aug 25 13:03:39 PDT 2016
Finished:	Thu Aug 25 13:42:04 PDT 2016
Elapsed:	38mins, 25sec
Diagnostics:	
Average Map Time	37sec
Average Shuffle Time	28mins, 59sec
Average Merge Time	0sec
Average Reduce Time	1sec

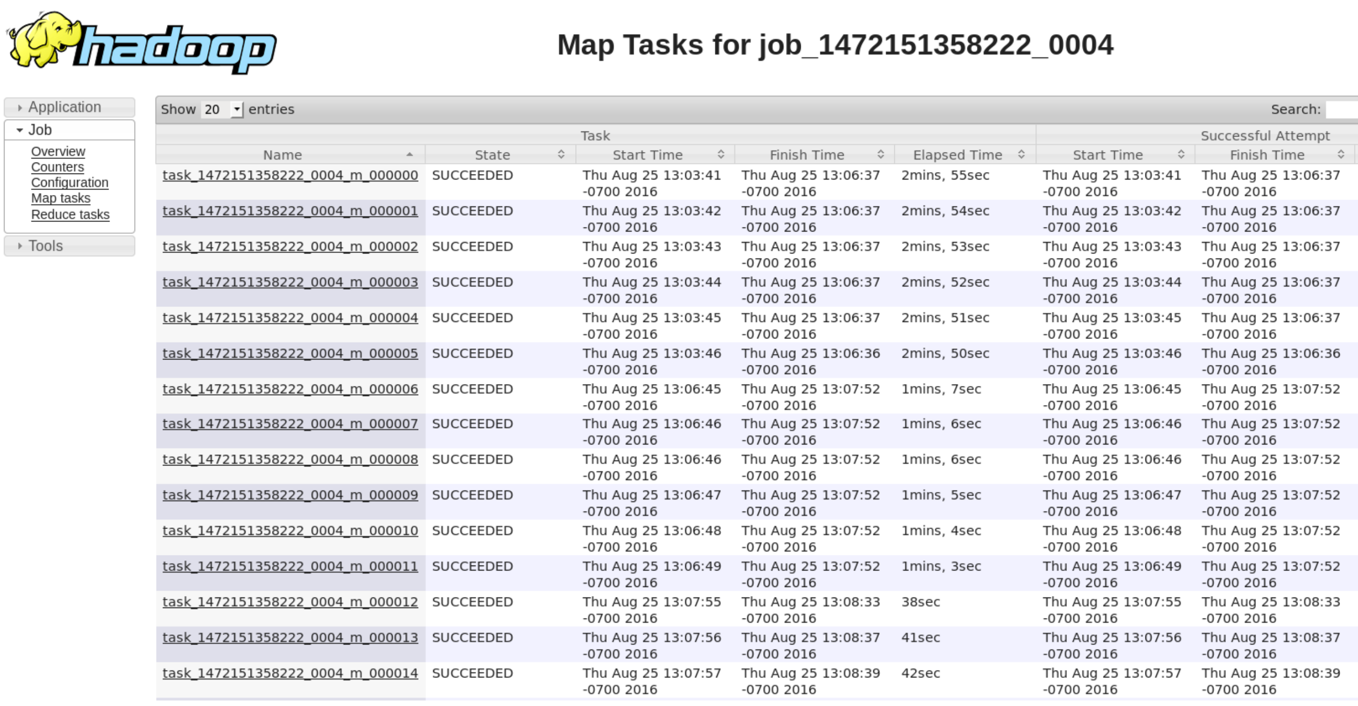
ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Thu Aug 25 13:03:32 PDT 2016	quickstart.cloudera:8042	logs

Task Type	Total	Complete
Map	302	302
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	302
Reduces	0	0	1

Figure 5.3: High level detail about a MapReduce job.



Map Tasks for job_1472151358222_0004

Application
Job
Overview
Counters
Configuration
Map tasks
Reduce tasks
Tools

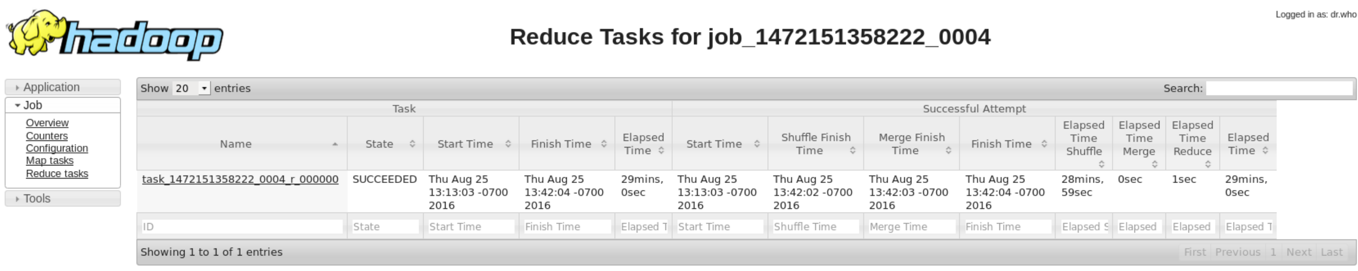
Show 20 entries

Search:

Task					Successful Attempt	
Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time
task_1472151358222_0004_m_000000	SUCCEEDED	Thu Aug 25 13:03:41 -0700 2016	Thu Aug 25 13:06:37 -0700 2016	2mins, 55sec	Thu Aug 25 13:03:41 -0700 2016	Thu Aug 25 13:06:37 -0700 2016
task_1472151358222_0004_m_000001	SUCCEEDED	Thu Aug 25 13:03:42 -0700 2016	Thu Aug 25 13:06:37 -0700 2016	2mins, 54sec	Thu Aug 25 13:03:42 -0700 2016	Thu Aug 25 13:06:37 -0700 2016
task_1472151358222_0004_m_000002	SUCCEEDED	Thu Aug 25 13:03:43 -0700 2016	Thu Aug 25 13:06:37 -0700 2016	2mins, 53sec	Thu Aug 25 13:03:43 -0700 2016	Thu Aug 25 13:06:37 -0700 2016
task_1472151358222_0004_m_000003	SUCCEEDED	Thu Aug 25 13:03:44 -0700 2016	Thu Aug 25 13:06:37 -0700 2016	2mins, 52sec	Thu Aug 25 13:03:44 -0700 2016	Thu Aug 25 13:06:37 -0700 2016
task_1472151358222_0004_m_000004	SUCCEEDED	Thu Aug 25 13:03:45 -0700 2016	Thu Aug 25 13:06:37 -0700 2016	2mins, 51sec	Thu Aug 25 13:03:45 -0700 2016	Thu Aug 25 13:06:37 -0700 2016
task_1472151358222_0004_m_000005	SUCCEEDED	Thu Aug 25 13:03:46 -0700 2016	Thu Aug 25 13:06:36 -0700 2016	2mins, 50sec	Thu Aug 25 13:03:46 -0700 2016	Thu Aug 25 13:06:36 -0700 2016
task_1472151358222_0004_m_000006	SUCCEEDED	Thu Aug 25 13:06:45 -0700 2016	Thu Aug 25 13:07:52 -0700 2016	1mins, 7sec	Thu Aug 25 13:06:45 -0700 2016	Thu Aug 25 13:07:52 -0700 2016
task_1472151358222_0004_m_000007	SUCCEEDED	Thu Aug 25 13:06:46 -0700 2016	Thu Aug 25 13:07:52 -0700 2016	1mins, 6sec	Thu Aug 25 13:06:46 -0700 2016	Thu Aug 25 13:07:52 -0700 2016
task_1472151358222_0004_m_000008	SUCCEEDED	Thu Aug 25 13:06:46 -0700 2016	Thu Aug 25 13:07:52 -0700 2016	1mins, 6sec	Thu Aug 25 13:06:46 -0700 2016	Thu Aug 25 13:07:52 -0700 2016
task_1472151358222_0004_m_000009	SUCCEEDED	Thu Aug 25 13:06:47 -0700 2016	Thu Aug 25 13:07:52 -0700 2016	1mins, 5sec	Thu Aug 25 13:06:47 -0700 2016	Thu Aug 25 13:07:52 -0700 2016
task_1472151358222_0004_m_000010	SUCCEEDED	Thu Aug 25 13:06:48 -0700 2016	Thu Aug 25 13:07:52 -0700 2016	1mins, 4sec	Thu Aug 25 13:06:48 -0700 2016	Thu Aug 25 13:07:52 -0700 2016
task_1472151358222_0004_m_000011	SUCCEEDED	Thu Aug 25 13:06:49 -0700 2016	Thu Aug 25 13:07:52 -0700 2016	1mins, 3sec	Thu Aug 25 13:06:49 -0700 2016	Thu Aug 25 13:07:52 -0700 2016
task_1472151358222_0004_m_000012	SUCCEEDED	Thu Aug 25 13:07:55 -0700 2016	Thu Aug 25 13:08:33 -0700 2016	38sec	Thu Aug 25 13:07:55 -0700 2016	Thu Aug 25 13:08:33 -0700 2016
task_1472151358222_0004_m_000013	SUCCEEDED	Thu Aug 25 13:07:56 -0700 2016	Thu Aug 25 13:08:37 -0700 2016	41sec	Thu Aug 25 13:07:56 -0700 2016	Thu Aug 25 13:08:37 -0700 2016
task_1472151358222_0004_m_000014	SUCCEEDED	Thu Aug 25 13:07:57 -0700 2016	Thu Aug 25 13:08:39 -0700 2016	42sec	Thu Aug 25 13:07:57 -0700 2016	Thu Aug 25 13:08:39 -0700 2016

Figure 5.4: Details about individual map task

Again Figure 5.4, shows information about each of the map jobs that were run, but with details at a much more granular level, like the time it took to complete each map task, the part of the file, that a particular map task handled and likewise. Each of the map tasks are numbered starting at 0 and can go on to indicate the total number of map task in the program.



Reduce Tasks for job_1472151358222_0004

Task					Successful Attempt							
Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Shuffle Finish Time	Merge Finish Time	Finish Time	Elapsed Time Shuffle	Elapsed Time Merge	Elapsed Time Reduce	Elapsed Time
task_1472151358222_0004_r_000000	SUCCEEDED	Thu Aug 25 13:13:03 -0700 2016	Thu Aug 25 13:42:04 -0700 2016	29mins, 0sec	Thu Aug 25 13:13:03 -0700 2016	Thu Aug 25 13:42:02 -0700 2016	Thu Aug 25 13:42:03 -0700 2016	Thu Aug 25 13:42:04 -0700 2016	28mins, 59sec	0sec	1sec	29mins, 0sec
ID	State	Start Time	Finish Time	Elapsed T	Start Time	Shuffle Time	Merge Time	Finish Time	Elapsed S	Elapsed	Elapsed	Elapsed T

Showing 1 to 1 of 1 entries

Figure 5.5: Details about reduce task

Similarly the above figure shows the detail of the reduce task for a MapReduce job that was shown before. As can be seen from the figure, there was just one reduce task for the program in the example and like the map tasks, its contains information about the run time for that particular reduce task and a log file where more details about the reduce task can be found.

5.4 Implementation Summary

In this chapter we have outlined the technologies that we have chosen and why we chose to go with those said technologies. We introduced Java, Tomcat, MapReduce, Jgit to name a few.

Next we explained the implementation of the RESTFul web service that listens for connections from clients, and once it gets a request, it passes it on to the worker nodes for the necessary computation to be done. We touched upon the format of the JSON request and response messages that are handled by our application.

We also described the MapReduce implementation version of our program, where we showed how the data type of the Mapper and Reducer classes were chosen and also how the end results about a completed MapReduce job can be retrieved from the Hadoop service running on port 19888. We are satisfied that both the implemented versions of our application, satisfy the requirement that we had set out in out previous chapter. Next we will be evaluating the results gathered from performing experiments on different data sets.

Chapter 6

Evaluation

In the last chapter we discussed the different implementation strategies used for the various design solutions to our problem. In this chapter we will be evaluating the results derived from the running of our implemented solutions on open source code repositories on Github. The links to the repositories used for experimentation purposes, are listed in Appendix 2. Next we will discuss if we have any issues with the data that was collected and provide explanation for any anomaly that we might have encountered in the result data set.

6.1 Comparison between the Baseline and Distributed Model

For this experiment, we executed both the baseline and distributed implementation of cyclomatic complexity on the code repositories that we had mentioned at the beginning of the chapter. For the baseline solution, we didn't have any additional third party library for interacting with Github and cloning the repositories locally. So for running the experiments, we cloned the repositories manually and then passed on the processing to the baseline implementation.

While for the distributed mode, everything was taken care of by the application, from cloning to checking the git system variables for changed files. The results that we gathered, were the time it took each of the implementation to run, starting from the time the first file was passed in for processing, up to the time the last file in the repository was processed.

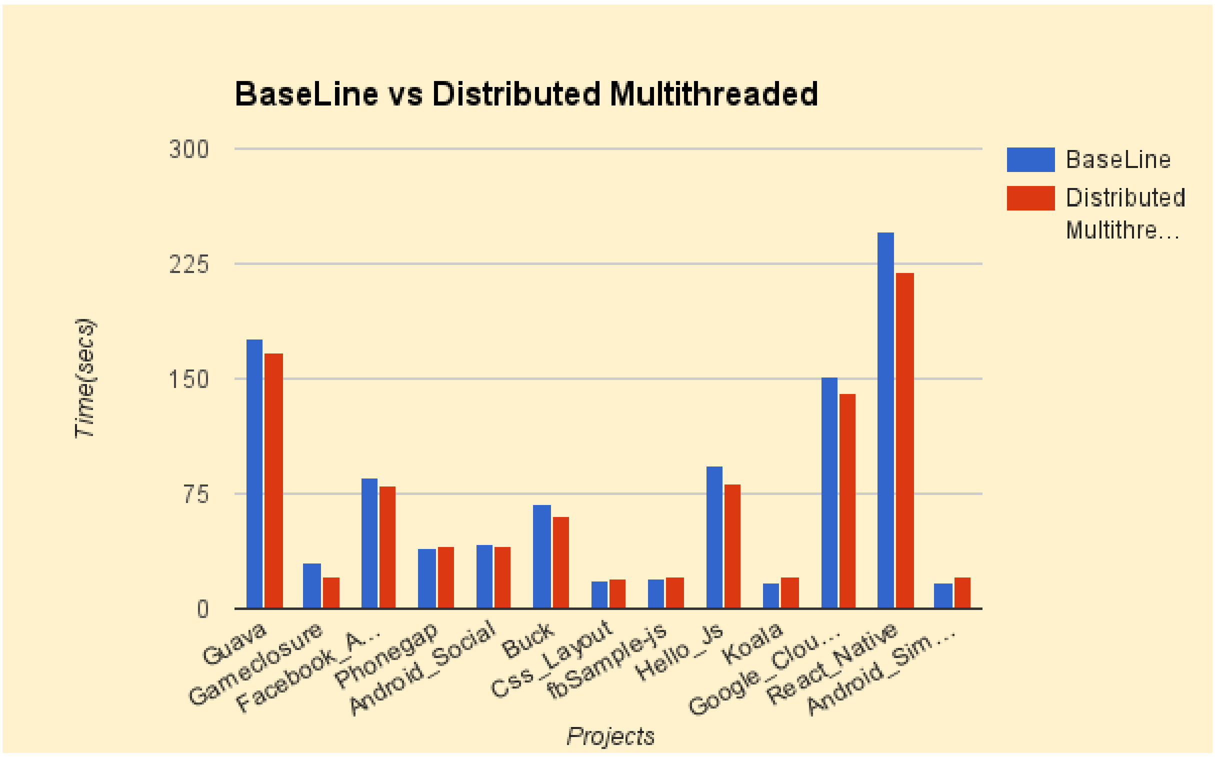


Figure 6.1: Results comparing the runtime between the baseline and distributed implementation

The figure above shows the comparison between the baseline and distributed model runtime. The blue bars represent baseline and red represents the distributed model run time. As can be seen from the graphs, the distributed multi-threaded model almost always get the calculation of cyclomatic complexity done sooner than the baseline implementation except when the size of the projects are relatively small. For small sized projects the baseline solution seems to perform better than the distributed model. The reason for that being, since in the distributed model, even though the computation can be done faster, there is a need for synchronization among the threads that are performing the calculation. So for projects with small number of files, even through the files are processed in parallel, there is an extra overhead for synchronization between the threads. In the baseline solution, there is just a single thread of execution, and there is no need for synchronization in that model. Hence even if the processing might be a bit slower, but in the end the model not involving distribution works better for projects with small number of files.

So we can conclude that for small projects with very few files in it, ,the baseline solution performs better, while for all other scenarios the distributed implementation is the preferred model.

6.2 Comparison between First Run and Subsequent Runs

As explained in the objectives of the research, once we run calculation of cyclomatic complexity baseline implementation on a project, the next time any changes are made to the project, we have to run cyclomatic complexity all over again from scratch. This is a very cumbersome problem and a huge pain point, so we tried to address it in the distributed implementation.

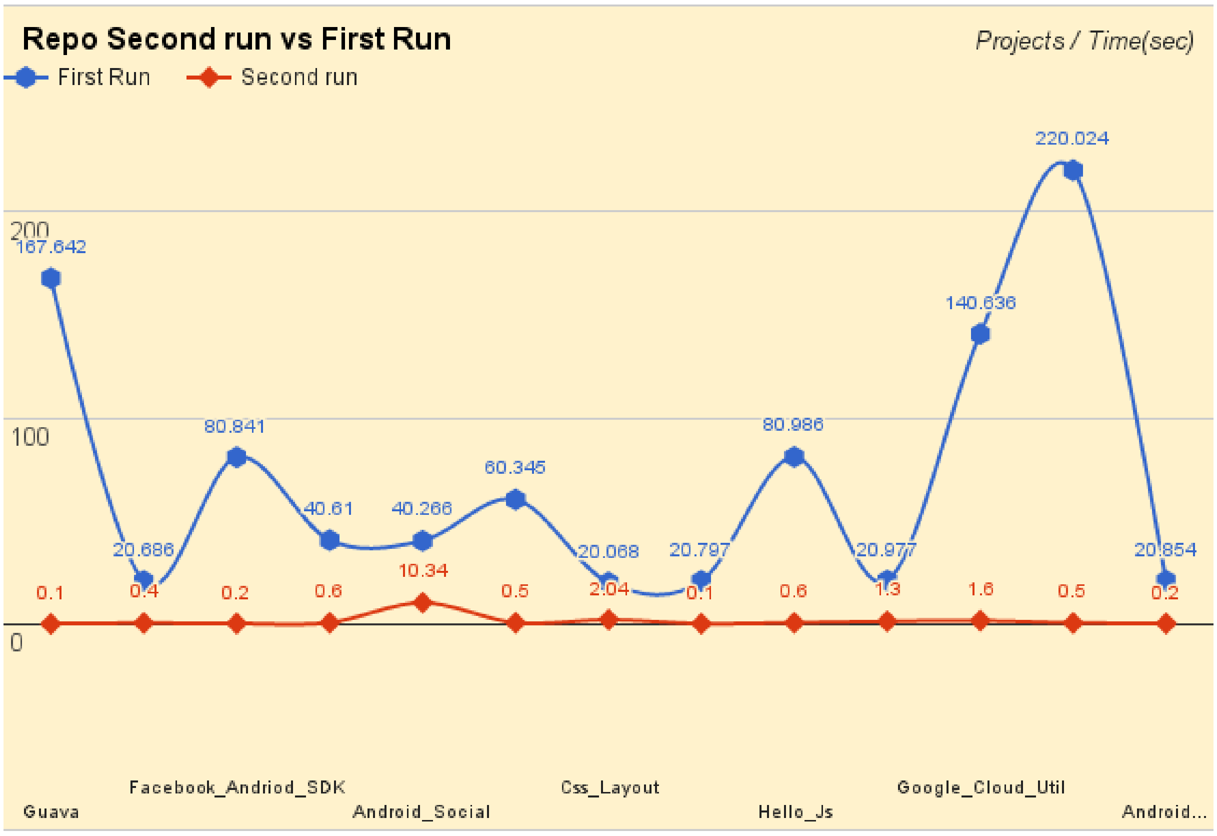


Figure 6.2: Comparison between the first and second runs in a distributed model

The blue graph represents the first run of the distributed version of CC on the project and the red graph the second run on the same project with additional commits than the first run. As explained in the implementation chapter, we hold intermediate data from the first run on a repository and use that information for subsequent runs. As we ran our experiments within a duration of a few days, there weren't many commits made to the repository that were under study. So we would just compare the first run with the second run.

As is evident from the graph above, the second run is extremely faster compared to the first run and finishes under a second for almost all the repository in the study except one. The running time for the second run of CC is dependent on the amount of changes that have been made since the last commit id was saved in the intermediate result file. But the changes are no where near the number of files that needs to be touched when a repository is first brought in for processing. Hence storing the intermediate results brings down the processing time considerable.

6.3 Comparision between Distributed and MapReduce Model

The setting for the distributed model remains the same, as it was for the comparison with the baseline implementation. For the MapReduce implementation we need to make a new adjustments through. First, we will be comparing only the first run in both implementations, as in the MapReduce model there is no provision of holding intermediate results. Secondly, since the namenode has to distribute out the data first, before the processing can begin, there is an extra cost added to the distribution of data and that would have to be added on top of the cost of processing the data.

In the figure below, the red graph represents the MapReduce running time graph for each of the Github repositories, while the blue line represents the running time on the same repositories for the distributed design. The processing time for MapReduce, is the sum of the time required to distribute the code out to the machines plus the time needed to process the data.

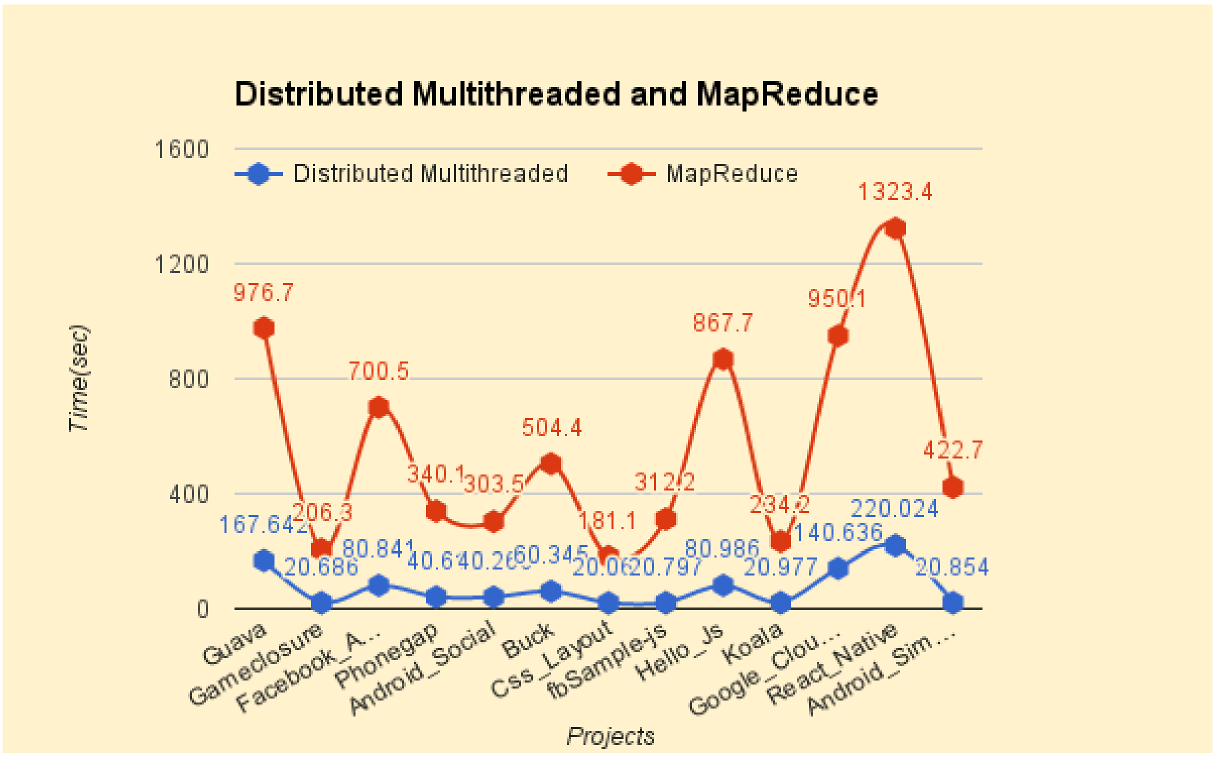


Figure 6.3: Comparison between distributed and MapReduce running time

As can be seen from the figure, the processing time in MapReduce is higher than the time required to process the same data in our other distributed solution. The difference between the run times are further apart, so this would need a bit more in depth analysis. There is an anomaly here, since the run times shouldn't have been very spread apart from each other, as MapReduce is using more hardware than our distributed design for solving the same problem.

So next, as a starting point we would look at average size of the files constituting each of the repositories.

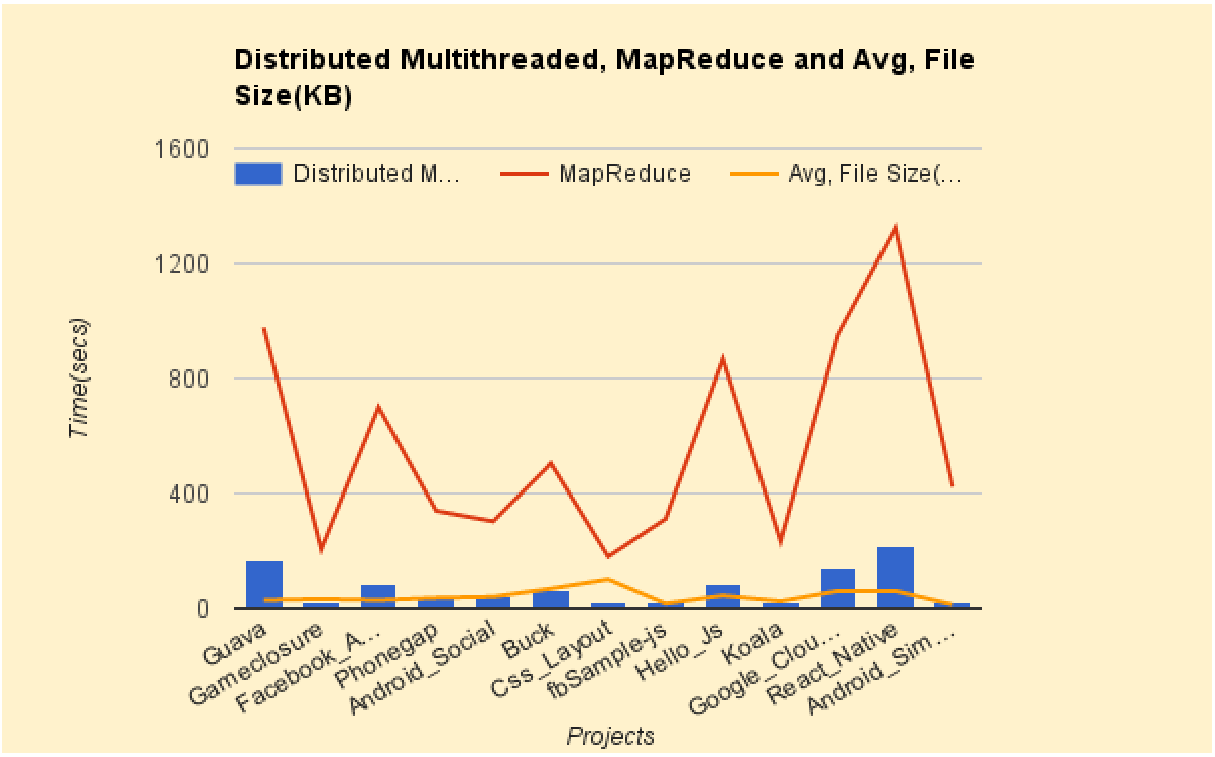


Figure 6.4: Distributed, MapReduce implementation vs average file size

The yellow line shows the average size of the files for each of the individual projects. Although it doesn't convey a lot of information it is a good starting point to go deeper into the cause of the problem. The figure in the next page, shows projects that have very similar running time in our distributed model and compares the average size of the files for these projects and the MapReduce running time. It starts to show some relation in that, as the avg size of the files in the project increases, the MapReduce run time is gets lower compared to the other projects. But it can't be specifically said that it is the file size that is causing the program to run very slow. We need to study a bit about the file storage in HDFS and if it is possible that the size of files impact the running of MapReduce jobs.

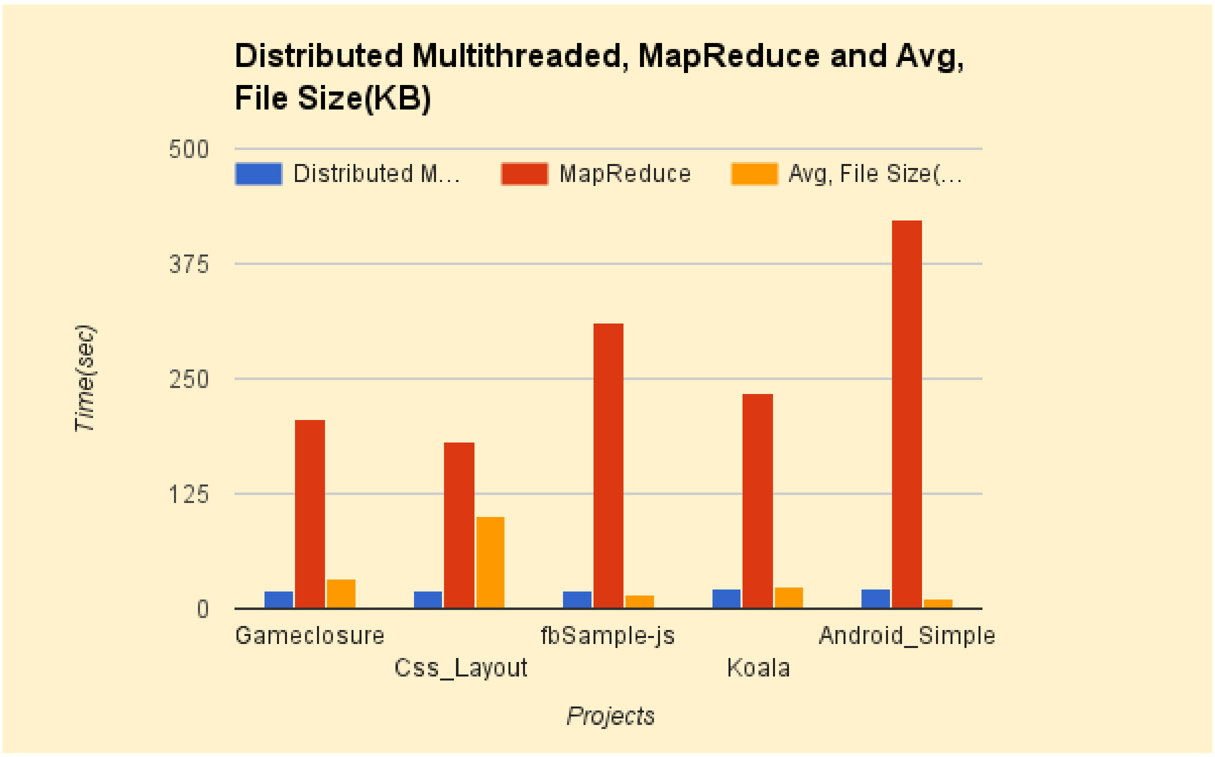


Figure 6.5: Looking at the avg file size for similar projects

6.4 Effects of File Size on MapReduce Jobs

As we have discussed earlier in the chapter on implementation, the name node of an HDFS cluster stores all the information related to the files stored in HDFS, in its memory. Every file or directory in HDFS is represented as an object in the name node's memory. If there are a lot of small files, then name nodes memory becomes overwhelmed by the amount of information that needs to be stored. In normal operations, the name node must constantly track and check where every block of data is stored in the cluster. This is done by listening to data nodes to report on all of their blocks of data. The more blocks a data node must report, the more network bandwidth it will consume. So for optimization it is clear that if we reduce the number of small files, we can reduce the name node memory footprint and network impact.

In Hadoop architecture there is a concept of blocks and they are typically of 64MB size. It can be configured to stored files in other sizes as well, but it has to be multiples of 64 always. Hence 64 MB is the smallest block size that can be stored. The reason for splitting a file and storing it into different blocks is for parallel processing of data.

But there is a logical storage unit as well called input split which is only a logical chunk of data and points to the start and end locations within a block. The number of mappers that are started is equivalent to the number of input splits that the data is divided into. When a MapReduce job launches, it schedules one map task per block of data being processed. So if the file is very small and there are a lot of them, then each map task processes very little input and there are a lot more map tasks, each of which imposes extra bookkeeping overhead. Usually Hadoop is configured so that each map task runs in its own JVM. So for a project with thousand files, it require spinning up and tearing down thousands of JVM. A cluster only has so many resources, so there is provision for limiting the maximum number of concurrent mappers to 25. And for a job with say one thousand files to process, only 25 of those can be run at a time, causing the others to be queued. The queue will become quite large and the processing will therefore take a long time. Furthermore, HDFS is not geared up to efficiently access small files, it is primarily designed for streaming access of large files. Reading through small files normally causes lots of seeks and lots of hopping from data node to data node to retrieve each small file, all of which is an inefficient data access pattern. One large sequential read will always outperform reading the same amount of data via several random reads.

6.5 Modifications to Files copied to HDFS

In the last section, we looked at the possible side effects that small files can have on MapReduce performance. So if we rerun MapReduce jobs on say the Guava library with more than 1000 files, here is what the the MR job summary list the information as :

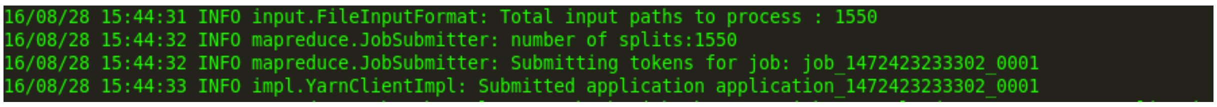
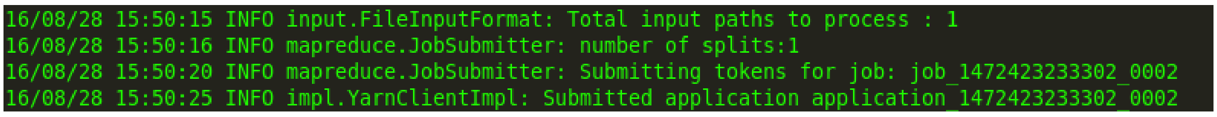

```
16/08/28 15:44:31 INFO input.FileInputFormat: Total input paths to process : 1550
16/08/28 15:44:32 INFO mapreduce.JobSubmitter: number of splits:1550
16/08/28 15:44:32 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1472423233302_0001
16/08/28 15:44:33 INFO impl.YarnClientImpl: Submitted application application_1472423233302_0001
```

Figure 6.6: Job status when Guava was run previously

As can be seen from the second line in the figure above, the MR job is having to process 1550 splits, and as a result it would be creating 1550 individual map task for the 1550 files, distributed between two machines. Previously we didn't have any idea what was causing the problem. As we know now on a project with 1500 files, the MapReduce job will spawn 1500 mappers, which can be a huge overhead for just two machines. Even with a project with 100 files, its going to be a huge overhead, not to mention the overhead when we are dealing with 1500 files. The cost of queuing and contention for getting CPU for the hundreds of mappers is just too high for a cluster of just 3 machines.

So in order to test, how the cluster fairs when big files are given, we combined the contents of 1550 individual files that made up the previous project, into a single large file and here is what we observed :



```
16/08/28 15:50:15 INFO input.FileInputFormat: Total input paths to process : 1
16/08/28 15:50:16 INFO mapreduce.JobSubmitter: number of splits:1
16/08/28 15:50:20 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1472423233302_0002
16/08/28 15:50:25 INFO impl.YarnClientImpl: Submitted application application_1472423233302_0002
```

Figure 6.7: Job status when files were combined into a single large file

As can be seen from the snapshot above, although the contents of the files are still the same, but now in one file, instead of hundreds of small files, MR job has got just one input split to process as seen from the second line. Hence the MR job will create just a single map task for processing the file. Although it is not a very efficient way of handling a file, from a concurrent processing point of view, but now there is no requirement for the name node to keep track of the hundreds of mappers and their queuing and execution details.

Now here is what we did. We took all the repositories that we had tested on previously and combined the contents of the files that made up a repository into a single file for each repository. Then we copied the single file that made up a repository, into HDFS and ran the MR job on that merged file. And here is what the results looked like :

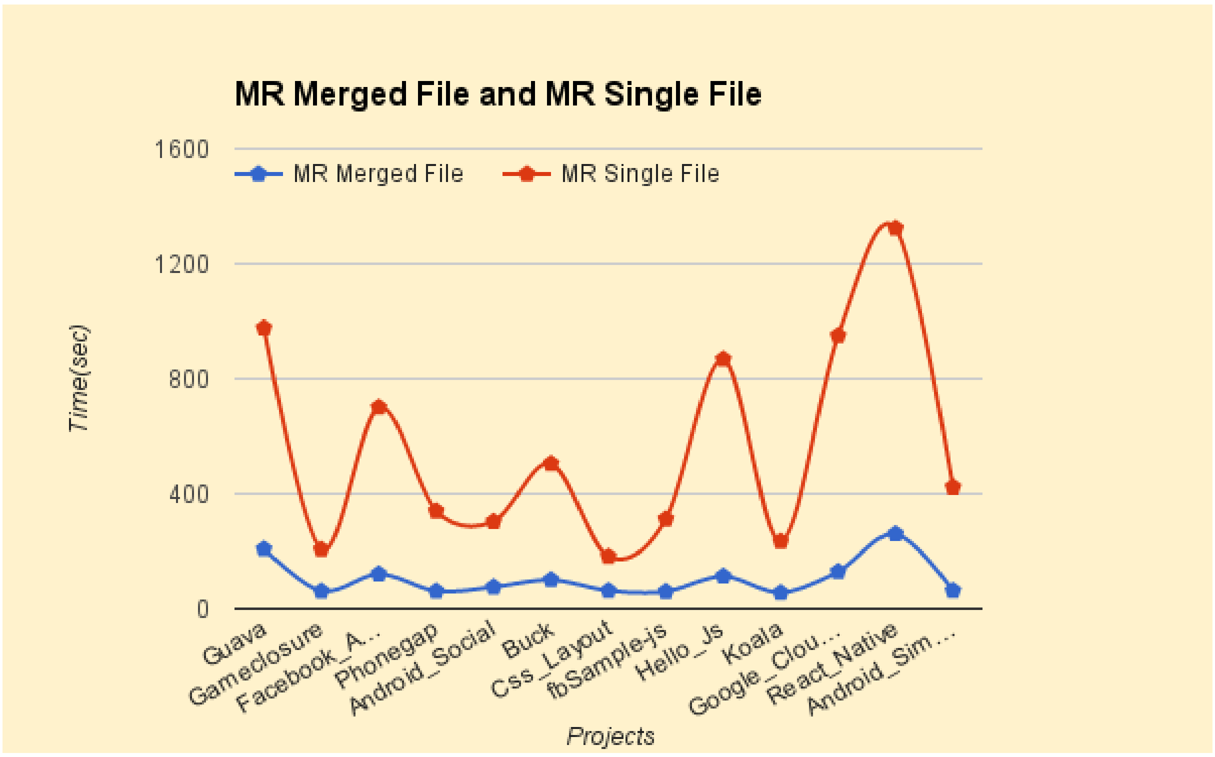


Figure 6.8: Comparison between runtime for individual files and merged file

While running the MR job for the merged files, we see a considerable speed up of the processing for each job. So merging the file does lead to an improvement in the running time of the job. However the timings are still not near to the running time for our REST implementation. The files are merged into one huge file, and the merged file size is no where close to the 64MB block size that Hadoop uses to split files. The reason for the higher running time might be because the size of a file on most occasion is less than 64MB, the whole file is directed just to a single machine since it would fit into one block. So even through we have got two workers, we are able to utilize the processing power of just a single machine. But further study needs to be made to pin point the exact reason and we will leave it for future work.

6.6 Discussion

So as we can see, file size plays an important role, on how the processing happens in Hadoop. It is not very helpful if we have got a large number of small files, or just a single large file that is smaller than the block size of 64MB. For the former case, a large number of map tasks are spawned which chocks the memory of the name node , while in the latter case, there is just a single map task and the data can easily fit into a single machine, there isn't a lot of option to parallelize the processing of data. But what we usually observe in code repositories is that there are hardly any very large files and a repo usually consists of large number of small sized files. So if we need to take the help of the processing that Hadoop provides, we would have to find ways to combine everything into a file, so that the file is not very small and the file is greater than 64MB block size.

6.7 Evaluation Summary

In this chapter we evaluated the data which we gathered by running the different solutions that we implemented for cyclomatic complexity calculation, on open source code repositories on Github.

First we compared the results gathered from the running of the baseline solution against the distributed design and we found that distributed design almost always performs better. The only exception being when there are few files in the project. Once the number of files become huge, our distributed solution always has a better performance than the baseline solution.

Secondly, we contrasted the results gathered through the optimization step, where by we saved intermediate results while running the projects for the first time and used the intermediate results to perform calculation when the project is submitted for the second time. What we observed was, that storing intermediate results brought the running time down significantly for all the project and was a huge performance optimization for large project.

Next, we compared the distributed implementation with the MapReduce implementation of cyclomatic complexity, and we observed that the distributed implementation is a better solution if we take the running time on a repository into consideration. Also we found that MapReduce is not very geared towards processing on very small sized files, due to memory footprint issues and the large number of map tasks that are created for each of the files. Lastly we experimented with combining the small files into a large file by copying all the contents of the small files into a single large file. This saw the run time come down significantly compared to the MapReduce job which was running for single files, but still the running time was nowhere closer to our distributed model which performed seamlessly with all sizes of files. We concluded the section by discussing the use of the Hadoop platform for performing calculation on code repositories.

Chapter 7

Conclusions

In the previous chapter we evaluated the data sets collected from performing our experiments on the applications that we had implemented. In this chapter we will summarize our project and discuss on future works.

7.1 Project Overview

The aim of this dissertation was to explore different ways of distributing calculation of code quality metrics and to come up with solutions to this problem. We also asked if it might be possible to hold intermediate results to speed up the processing for subsequent runs of the projects.

We started by going through the different code quality metrics that are available along with their merits and checking if they would fit into our research model. After studying a lot a metrics, we decided to go with cyclomatic complexity, which measures the total number of linear independent paths through a program as our metrics of choice. Next we introduced different implementation, that could be used for solving the problem. We discussed the web services model, which allowed for efficient processing of repositories of all sizes, regardless of whether the files that constituted the repository were small or big. The MapReduce model had problems processing large number of small files and alternatives had to worked out to get the processing done in lesser time. Also we were able to bring the running time down for multiple runs of the same repository my storing intermediate results from the previous runs.

7.2 Future Work

While this project sets a good foundation for calculating code quality metrics using cyclomatic complexity, still there are a lot of room for improvement. As we know by now, there is no one silver bullet that caters to all the issues regarding the quality of code. A lot of code quality metrics are available and each one servers a separate purpose. Depending on the context different code metrics can be used. So for future studies it would be worthwhile to look into some other metrics that are of a different category than cyclomatic complexity and if possible find ways of integrating other metrics with cyclomatic complexity and provide an aggregated number from both the metrics combined together.

Also in the current implementation, we made the assumption that any time new changes are made to a file, the cyclomatic complexity might change and so we run CC computation on the file again. But it is not always the case. There might be scenarios where, a lot of changes have been made to a repository, but CC remains unchanged. In the future, there is this exciting area of research on how cyclomatic complexity changes on a commit to commit basis for a file, so that whenever any new changes comes in the future, it becomes possible to predict whether there would be any changes to cyclomatic complexity. Even if the predicted value is not very accurate, if we are even able to provide a rough estimate that would be a huge achievement as we would be able to produce results without even reading the contents of the file.

The problem with small sized files that we encountered while processing files, through MapReduce needs to be studied in more details. More specifically we said, Hadoop is not very well suited for running large number of small files. Future work in this domain would involve looking at what other have done when they have encountered the same problem with having a lot of small files and then if possible port those ideas into out application. We also need to investigate what is the value for the size of a file, where the processing in MapReduce will outperform the processing for the distributed solution and whether or not it's possible to do that.

Appendix A

Abbreviations

Short Term	Expanded Term
SLOC	Source Lines of Code
LOC	Lines of Code
ELOC	Effective Lines of Code
CK	Chidamber and Kemerer
WMC	Weighted Methods Class
CBO	Coupling Between Objects
NOC	Number of Children
RFC	Response for a Class
LCOM	Lack of Cohesion of methods
CFG	Control Flow Graph
CC	Cyclomatic Complexity
SAN	Storage Area Network
MR	Map Reduce
HDFS	Hadoop Distributed File System
DFS	Distributed File System
SHA-1	Secure Hash Algorithm
REST	Representation State Transfer
HTTP	Hypertext Transfer Protocol
URI	Uniform Resource Identifiers
Repo	Repository
CPU	Central Processing Unit

Short Term	**Expanded Term**
API	Applicaition Programming Interface
OS	Operating System
IDE	Integrated Development Environment
HTML	Hypertext Markup Language
SQL	Structured Query Language
XML	Extensible Markup Language
JSON	JavaScript Object Notation
HDFS	Hadoop Distributed File System
UI	User Interface

Appendix B

Github Repositories used in Experiments

Repository Name	**Github URL**
Guava	https://github.com/google/guava.git
Gameclosure	https://github.com/gameclosure/ facebook.git
FacebookAndroidSDK	https://github.com/facebook/ facebook-android-sdk.git
Phonegap	https://github.com/Wizcorp/ phonegap-facebook-plugin.git
AndroidSocial	https://github.com/antonkrasov/ AndroidSocialNetworks.git
Buck	https://github.com/facebook/buck.git
CSSLayout	https://github.com/facebook/ css-layout.git
FBSampleJS	https://github.com/fbsamples/f8app.git
HelloJS	https://github.com/MrSwitch/hello.js.git
Koala	https://github.com/arsduo/koala.git
GoogleCloud	https://github.com/GoogleCloudPlatform/ gsutil.git
Armour	https://github.com/theo-armour/threo.js.git
ReactNative	https://github.com/facebook/react-native.git
AndroidSimple	https://github.com/sromku/android-simple-facebook.git

Table B.1: Repositories used in the experiments

Bibliography

[1] IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990.

[2] Hassan Raza Bhatti. *Automatic Measurement of Source Code Complexity.* PhD thesis, Masters Thesis, Lulea University of Technology, Lulea, Sweden, 2011.

[3] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[4] Luke Campbell and Brian Koster. Software metrics: Adding engineering rigor to a currently ephemeral process. *briefing presented to the McGrummwell F/A-24 CDR course*, 1995.

[5] David N Card and William W Agresti. Measuring software design complexity. *Journal of Systems and Software*, 8(3):185–197, 1988.

[6] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[7] Neville I. Churcher, Martin J. Shepperd, S Chidamber, and CF Kemerer. Comments on" a metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 21(3):263–265, 1995.

[8] JGit commiters. JGit homepage. `https://eclipse.org/jgit/`, 2016. [Online; accessed 25-August-2016].

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] T. DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[11] Khaled El Emam and Norman F Schneidewind. *Methodology for Validating Software Product Metrics.* Wiley Online Library, 2000.

[12] N. FENTON and S. L. Pfleeger. In *Software Metrics - A Rigorous and Practical Approach, 2 ed*, chapter 3, pages 74–75. International Thomson Computer Press, London, 1996.

[13] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[14] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[15] Jerry Fitzpatrick. Applying the abc metrics to c, c++ and java. *C++ Report*, 1997.

[16] Apache Foundation. HadoopMapReduceVersion2.7. `https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`, 2016. [Online; accessed 22-August-2016].

[17] git. Git Version Control. `https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control`, 2016. [Online; accessed 19-August-2016].

[18] Git-scm. Git Internal Definitions. `https://www.sbf5.com/~cduan/technical/git/git-1.shtml`, 2016. [Online; accessed 19-August-2016].

[19] GitHub. Github Home. `https://github.com/`, 2016. [Online; accessed 22-August-2016].

[20] Maurice Howard Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.

[21] SonarQube Inc. Sonarqube Description. `http://www.sonarqube.org/Description`, 2016. [Online; accessed 10-August-2016].

[22] Wikipedia Inc. Euler's Formula. `https://en.wikipedia.org/wiki/Planar_graph#Euler.27s_formula`, 2016. [Online; accessed 16-August-2016].

[23] Wikipedia Inc. Software Metrics. https://en.wikipedia.org/wiki/Software_metric, 2016. [Online; accessed 11-August-2016].

[24] Wikipedia Inc. Structure Programming. https://en.wikipedia.org/wiki/Structured_programming, 2016. [Online; accessed 16-August-2016].

[25] Trinity College IT. SCSS OpenNebula Home. https://support.scss.tcd.ie/index.php/SCSSnebula_Virtualisation_Platform, 2016. [Online; accessed 22-August-2016].

[26] Ayman Madi, Oussama Kassem Zein, and Seifedine Kadry. On the improvement of cyclomatic complexity metric. volume 7, pages 67–82. 2013.

[27] Lou Marco. Measuring software complexity. *Enterprise Systems Journal*, 1997.

[28] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[29] Sam Miller. Area of Use of Software Metrics. http://www.articlesbase.com/management-articles/areas-of-use-for-software-metrics-306127.html, 2008. [Online; accessed 12-August-2016].

[30] Everald E Mills. Software metrics. Technical report, DTIC Document, 1988.

[31] Harlan D Mills. Mathematical foundations for structured programming. 1972.

[32] Anton Milutin. Software code metrics. http://www.viva64.com/content/articles/codeanalyzers/?f=Metrics.html&lang=en&content=code-analyzers, 2009. [Online; accessed 13-August-2016].

[33] Sanjay Misra. An object oriented complexity metric based on cognitive weights. In *6th IEEE International Conference on Cognitive Informatics*, pages 134–139. IEEE, 2007.

[34] Microsoft Inc. MSDN. Code Metrics Values. http://msdn.microsoft.com/en-us/library/bb385914.aspx, 2015. [Online; accessed 13-August-2016].

[35] Brian A Nejmeh. Npath: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, 1988.

[36] Steven D Sheetz, David Henderson, and Linda Wallace. Understanding developer and manager perceptions of function points and source lines of code. *Journal of Systems and Software*, 82(9):1540–1549, 2009.

[37] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[38] Jennifer Widom. Research problems in data warehousing. In *Proceedings of the fourth international conference on Information and knowledge management*, pages 25–30. ACM, 1995.