Western Kentucky University
TopSCHOLAR®

Masters Theses & Specialist Projects | Graduate School

Summer 2015

An Apache Hadoop Framework for Large-Scale Peptide Identification

Harinivesh Donepudi
Western Kentucky University, harinivesh.donepudi110@topper.wku.edu

Follow this and additional works at: http://digitalcommons.wku.edu/theses

Part of the Biochemistry, Biophysics, and Structural Biology Commons, and the OS and Networks Commons

Recommended Citation
Donepudi, Harinivesh, "An Apache Hadoop Framework for Large-Scale Peptide Identification" (2015). *Masters Theses & Specialist Projects.* Paper 1527.
http://digitalcommons.wku.edu/theses/1527

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Masters Theses & Specialist Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact connie.foster@wku.edu.

AN APACHE HADOOP FRAMEWORK FOR LARGE-SCALE PEPTIDE IDENTIFICATION

A Thesis
Presented to
The Faculty of the Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Harinivesh Donepudi

August 2015

AN APACHE HADOOP FRAMEWORK FOR LARGE-SCALE PEPTIDE IDENTIFICATION

Date Recommended 07/23/15

Dr. Zhonghang Xia, Director of Thesis

Dr. James Gary

Dr. Michael Galloway

Dean, Graduate Studies and Research Date 8-2-15

DEDICATION

To my Mother and God in whom I trust.

&

To all the computer scientists that are working around the globe for better society who believe "You can have data without information, but you cannot have information without data".

ACKNOWLEDGMENTS

I wish to extend my gratitude to Dr. Zhonghang Xia for offering his advice and guidance throughout my masters. It has been a great pleasure to work alongside him both a graduate assistant and a student. I am truly grateful for his instrumental support throughout my research. I am thankful for the opportunities he provided me to use the latest technologies such as Hadoop.

I would also like to acknowledge Dr. Michael Galloway. I am thankful for the way that he motivated me. I enjoyed our discussions. I appreciate him taking the time to review this document. His insight was imperative in completing this research. He also inspired me to extend my current research towards cloud computing.

I would like to thank Dr. James Gary who accepted my request to be part of my thesis committee. He and the rest of the Computer Science faculty have never ceased to be supportive in my endeavors. I would also like to extend my gratitude to Western Kentucky University for allowing me to pursue my masters here.

I would also like to thank my friend Travis Brummett for helping me throughout my course work and for aiding me in revising this document. I would like to extend my appreciation to my friend Bindu Priya Bhavineni for supporting me during my tough times. I would like to Thank all my classmates for their extended support.

Finally, I would like to acknowledge my family. I want to extend my gratitude to my mother who has continuously supported me.

CONTENTS

DEDICATION . . . iii

ACKNOWLEDGMENTS . . . iv

LIST OF TABLES . . . x

LIST OF FIGURES . . . xi

ABSTRACT . . . 1

1 INTRODUCTION . . . 2

1.1 Big Data Processing . . . 2

1.2 Methods of Analyzing Big Data . . . 3

1.2.1 Apache Hadoop . . . 3

1.3 Bioinformatics . . . 4

1.4 Protein and Peptide Identification . . . 5

1.4.1 Peptide . . . 5

1.4.2 PSM Data as a Big Data . . . 6

1.4.3 C-Ranker, Peptide, and Protein Identification . . . 7

1.5 Prior Research . . . 8

1.5.1 Problems with the Current Bioinformatics Algorithm(C-Ranker) . . . 9

1.6 Proposed Solution . . . 9

1.6.1 Thesis Statement . . . 10

1.6.2 Research Question . . . 10

1.6.3 Area of Research 10
1.6.4 Scope of Thesis 11
1.6.5 Methodology 11
1.6.6 Advantages 11
1.7 Organization of Thesis 12
2 BACKGROUND 14
2.1 Terms and Definitions 14
2.1.1 Protein Identification Using Mass Spectrometry 14
2.1.2 Protein Search Engines Using Databases 16
2.1.3 Post Database Searching 17
2.2 PeptideProphet 18
2.2.1 Benifits and Issues of PeptideProphet 18
2.3 Percolator 19
2.3.1 Benefits and Issues of Percolator 20
2.4 CRanker 20
2.4.1 Reasons for Choosing CRanker 21
2.5 Why Apache®Hadoop™Framework 23
2.5.1 Top Reasons to Consider Hadoop 23
2.5.2 Apache®Hadoop™ 23
2.6 Apache Hadoop Distributed File System 26
2.6.1 HDFS Architecture 26
2.7 Apache Hadoop MapReduce 31

2.7.1 What is MapReduce? 31

3 EXISTING PROBLEMS AND PLANNED IMPROVEMENTS 36

3.0.2 Existing Solution to Execute CRanker 36

3.0.3 Existing Issues with CRanker 39

3.1 Why Only Use Apache Hadoop for CRanker Execution 41

3.1.1 Case Studies Which Motivated the Choice of Apache Hadoop . . . 41

4 PROPOSALS,ENVIRONMENT SETUP, DESIGN, AND IMPLEMENTATION . 43

4.1 Proposals Made 43

4.1.1 Observations Made on Increasing the Computation Power 43

4.1.2 Observations Made on Using High-Performance Computing 45

4.1.3 Observations Made on Using GPU Computing 47

4.1.4 What About Re-Implementing the CRanker Algorithm 49

4.2 Parallelizing the CRanker Application 51

4.2.1 Finalized Idea for CRanker Execution and Approved Concept . . . 51

4.3 Setting of Environment and Other Essentials 51

4.3.1 Setting Up the Hardware Infrastructure 52

4.3.2 Setting Up the Software Infrastructure 54

4.4 Proposed Framework Architecture, Design, and Implementation 54

4.4.1 Idea Execution and Design 54

4.4.2 Architecture 57

4.4.3 Implementation 61

5 DEVELOPMENT AND EXECUTION 64

5.1 Setting up of Resources 64

5.2 Determining the Approach for CRanker Execution 65

5.2.1 Outline of the Algorithm 66

5.2.2 Execution of the Algorithm 67

5.3 Determining the Joins for File Comparison 69

5.3.1 Outline of MapReduce Join Algorithm 70

6 EVALUATIONS AND ANALYSIS 72

6.1 Evaluating the Distributed Execution of CRanker Algorithm Using Apache Hadoop Approach 74

6.1.1 Evaluating the Memory Considerations and its Analysis 74

6.1.2 Evaluating the CRanker Execution Time 78

6.2 Evaluation and Analysis of File Comparison Algorithm 82

7 CONCLUSION AND FUTURE WORK 85

7.1 Conclusion 85

7.2 Future Work 86

REFERENCES 88

APPENDICES 91

A MAPREDUCE CODE FOR CRANKER DISTRIBUTED EXECUTION 92

A.A The driver for MapReduce CRanker distributed execution 92

A.B The MapReduce code for CRanker distributed execution 96

A.B.1 CRanker Execution Command 101

B MAPREDUCE CODE FOR FILE COMPARISON 102

B.A MapReduce Code for File Comparison Using Joins 102

C APACHE HADOOP CONFIGURATION FILES 106
C.A Apache Hadoop Master Node configuration Files 106
C.A.1 MapReduce Configuration 106
C.A.2 HDFS Configuration 108
C.A.3 Core Site Configuration 111
C.A.4 Apache Hadoop Yarn Configuration 112
C.B Apache Hadoop Slave Nodes configuration Files 114
C.B.1 MapReduce Configuration 114
C.B.2 HDFS Configuration 116
C.B.3 Core Site Configuration 119
C.B.4 Apache Hadoop Yarn Configuration 120
LIST OF ABBREVIATIONS 122

LIST OF TABLES

2.1 Target PSMS Output by PeptideProphet, Percolator, and CRanker. 22

2.2 Overlap of PeptideProphet and CRanker. 22

3.1 PBMC data execution on CRanker 40

5.1 Hardware used for development and execution 64

6.1 Test Beds Used for CRanker Distributed Execution Using Amazon EC2 . . 72

6.2 Test Bed for Apache Hadoop CRanker Execution on Localhost 73

6.3 CRanker Execution Times on Cluster 1 81

6.4 Matched Percentage of the CRanker Output 84

LIST OF FIGURES

1.1 Big Data Analysis Pipeline 4

2.1 Protein Arrangement and Protein Gathering 15

2.2 Seeking a Spectrum Library and de Novo sequencing 15

2.3 Hadoop execution architecture. 24

2.4 Hadoop architecture. 25

2.5 HDFS Architecture (Source Apache Hadoop). 28

2.6 HDFS Block Replication (Source from Apache Hadoop) 29

2.7 Hadoop Client Creates a File. 30

2.8 HighLevel MapReduce Pipeline 34

2.9 MapReduce Data Flow 35

3.1 CRanker Read Flow 37

3.2 CRanker Solve Process 38

3.3 CRanker Write Process 38

4.1 GPU Processing 48

4.2 CRanker DataFlow 49

4.3 CRanker Input Split DataFlow 50

4.4 Amazon EC2 instance access 53

4.5 CRanker Input File Split 56

4.6 Hadoop Resource Management Architecture 58

4.7 Proposed Framework Architecture 60

4.8 CRanker with MapReduce Dissected 61

5.1 MapReduce Join 70

6.1 CRanker Execution Burden Comparison 73

6.2 CRanker Normal Memory Usage Vs Apache Hadoop Jobs Memory Consumption on Localhost 75

6.3 CRanker Normal Memory Usage Vs Apache Hadoop Jobs Memory Consumption on Cluster 1 76

6.4 CRanker Normal Memory Usage vs Apache Hadoop Jobs Memory Consumption on Cluster 2 77

6.5 Summary of CRanker Memory Utilization 78

6.6 CRanker Execution Time:Normal Execution Vs Single Node Execution on Localhost 79

6.7 CRanker Execution Time:Normal Execution Vs Cluster 1 80

6.8 CRanker Execution Time:Normal Execution Vs Cluster 2 82

6.9 CRanker Execution Summary 83

6.10 Matched Percentage of CRanker Output 84

AN APACHE HADOOP FRAMEWORK FOR LARGE-SCALE PEPTIDE IDENTIFICATION

Harinivesh Donepudi August 2015 124 Pages

Directed by: Dr. Zhonghang Xia, Dr. James Gary, Dr. Michael Galloway

Department of Computer Science Western Kentucky University

Peptide identification is an essential step in protein identification, and Peptide Spectrum Match (PSM) data set is huge, which is a time consuming process to work on a single machine. In a typical run of the peptide identification method, PSMs are positioned by a cross correlation, a statistical score, or a likelihood that the match between the trial and hypothetical is correct and unique. This process takes a long time to execute, and there is a demand for an increase in performance to handle large peptide data sets. Development of distributed frameworks are needed to reduce the processing time, but this comes at the price of complexity in developing and executing them. In distributed computing, the program may divide into multiple parts to be executed.

The work in this thesis describes the implementation of Apache Hadoop framework for large-scale peptide identification using C-Ranker. The Apache Hadoop data processing software is immersed in a complex environment composed of massive machine clusters, large data sets, and several processing jobs. The framework uses Apache Hadoop Distributed File System (HDFS) and Apache Mapreduce to store and process the peptide data respectively.The proposed framework uses a peptide processing algorithm named C-Ranker which takes peptide data as an input and identifies the correct PSMs. The framework has two steps: Execute the C-Ranker algorithm on Hadoop cluster and compare the correct PSMs data generated via Hadoop approach with the normal execution approach of C-Ranker.

The goal of this framework is to process large peptide datasets using Apache Hadoop distributed approach.

Chapter 1

INTRODUCTION

There is great potential for end users in numerous fields of science to routinely lead big scale computations on distributed resources by utilizing a blend of the accompanying emerging technologies. Distributed computing contains the distributed middleware for connecting data or cluster of computing centers, this including resource scheduling or reservation, remote job submission and data management

1.1 Big Data Processing

Even from its earlier days, Google had to manage the issues in the usage and operation of a web index. In the time of "big data", no single machine can be relied upon to handle the volume of data and preparation needed to satisfy Google's central goal. That goal is: "to compose the world's data and make it universally available and useful". Today, we are flooded with a surge of data. In a broad scope of utilization areas, information is being gathered at an exceptional scale. Choices that were previously made in shrouded in mystery, or on meticulously developed models of reality, can now be made in light of the information itself. Such big data examination now drives every part of our present day society including retail, financial services, mobile services, manufacturing, physical sciences, and life sciences. In fact, there is a whole sequence of bioinformatics that to a great extent dedicated to the curation and examination of such information. As innovation advances,

especially with the appearance of Next Generation Sequencing, the size and number of trial data sets that are accessible is expanding exponentially.

For example, As demonstrated in the Figure 1.1, the analysis of Big Data includes different particular stages each of which presents challenges. Numerous individuals shockingly concentrate just on the investigation/demonstration stage. While that stage is essential, it is of little use without alternate periods of the information examination pipeline. In the investigation phase , which has gotten much consideration, there are inadequately comprehended complexities in the setting of multi-tenanted clusters where a few clients projects run concurrently. Numerous critical difficulties develop past the investigation stage. Case in point, big data must be overseen in connection, which may be uproarious, heterogeneous, and exclude a forthright model. Doing so raises the need to track provenance and to handle instability and failure: points that are vital to achievement, but sometimes simultaneously as big data. Also, the inquiries to the information examination pipeline will normally not all be laid out ahead of time. We may need to make sense of proper inquiries given the information. Doing this will require more brilliant frameworks providing better backing for client connection with the examination pipeline. We presently have a noteworthy bottleneck in the amount of individuals enabled to investigate the inquiries of the data and dissect it. Users can radically expand this number by supporting three levels of engagement with the data, not all obliging profound database aptitude.

1.2 Methods of Analyzing Big Data

1.2.1 Apache Hadoop

Apache Hadoop is an open-source programming system. It is written in Java for distributed storage and handling of vast information sets on PC clusters manufactured from

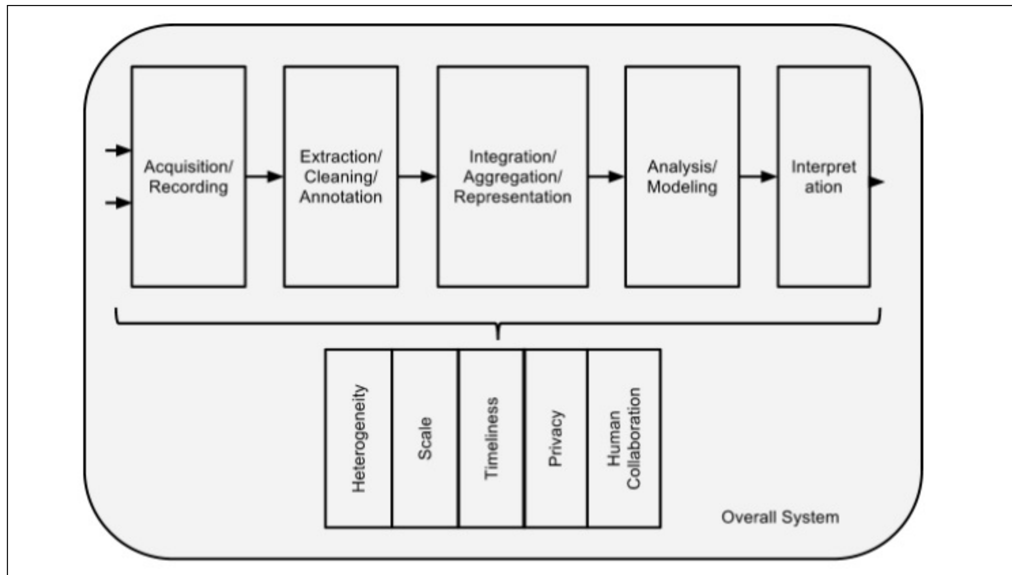


Figure 1.1: Big Data Analysis Pipeline

commodity hardware. All the modules in Hadoop are composed with a major presumption that equipment failure (of individual machines, or racks of machines) is ordinary.In the case of failures it is natural to handle through the programming. The center of Apache Hadoop comprised of a storage part (Hadoop Distributed File System (HDFS)) and a processing part (MapReduce).

1.3 Bioinformatics

Bioinformatics is an interdisciplinary field that creates systems and programming methods for For the comprehension of biological information. As an interdisciplinary field of science, bioinformatics joins software engineering, measurements, math, and architecture to study and procedure natural information.

Bioinformatics is an umbrella term for the assortment of organic studies that utilizes computer programming as a component of technique. It is also a reference to particular investigation "pipelines" that are utilized over and over, especially in the fields of

hereditary qualities and genomics. Normal employments of bioinformatics incorporate the ID of competitor qualities and nucleotides (SNPs). Regularly, such recognizable proof is improved to the point of comprehension of the hereditary premise of infection, exceptional adjustments, attractive properties or contrasts between populaces. In a less formal manner, bioinformatics additionally tries to comprehend the authoritative standards inside nucleic acid and protein sequences.

1.4 Protein and Peptide Identification

Proteins are huge natural particles, or macromolecules, comprising of one or more long chains of amino acid buildups. Proteins perform an inconceivable exhibit of capacities inside of living life forms, including catalyzing metabolic responses, recreating DNA, reacting to boosts, and transporting atoms between the source and destination. Proteins vary from each other fundamentally in their succession of amino acids, managed by the nucleotide grouping of their qualities. That normally brings about collapsing of the protein into a particular three-dimensional structure that decides its action.[Lehninger, Nelson, and Cox, 2005].

1.4.1 Peptide

Peptides are functioning organic particles. They are short chains of corrosive amino monomers connected by peptide (amide) bonds. Peptides differs from proteins on the premise of size, and as a self-assertive benchmark can be comprehended to contain roughly fifty or less amino acids. Proteins comprise of one or more polypeptides orchestrated in a naturally practical manner, regularly bound to ligands. The size limits that differentiate peptides from polypeptides and proteins are not definitive. Long peptides, for example, or

amyloid beta have been alluded to as proteins, and littler proteins like insulin have been mistaken as peptides [Zealandpharma.com, 2015].

1.4.2 PSM Data as a Big Data

Liquid chromatography combined with tandem mass spectrometry (LC/MS/MS) offers the guarantee to exhaustively recognize and evaluate the proteome of complexes, cells, and tissues.

The extensive quantities of peptide spectra created by LC/MS/MS investigations are routinely sought to utilize a search engine against hypothetical fracture spectra received from target databases containing either protein or interpreted nucleic acid sequences. It is regularly expected that a peptide spectrum match (PSM) for every MS/MS range is contained in the sequence database. In a run of the peptide identification technique, PSMs are positioned by a cross correlation, a measurable score, or a likelihood that the match between the experimental and hypothetical is correct and unique. Just those PSMs with the most noteworthy scores or most significant probabilities are accounted for as correct. On the other hand, this methodology dishonestly distinguishes the peptides frequently. In all actuality, more than half of PSMs initially doled out by database search engines, for example, SEQUEST[Fields.scripps.edu, 2015], MASCOT [Matrixscience.com, 2015], and X!TANDEM [Thegpm.org, 2015] are erroneous. Accordingly, the exactness of database list items is frequently assessed via seeking a decoy protein database to recognize the false discovery rate (FDR). Decoy databases contain either modified or arbitrarily rearranged protein successions received from the right or target protein database. The database search engine doles out an observed spectrum range to either an objective or a decoy sequence. The objective decoy database search additionally shows the quality or reliability of the tar-

get PSMs. In any case, the objective PSMs are not all correct because of the low quality of the exploratory MS/MS data, the arrangement not in the database, or unforeseen amino acid alterations. As a result, a small amount of the objective PSMs from the search engine is false positives. Thus, manual or computational methodologies are critical to approving target PSMs after a protein database examination of the LC/MS/MS data.

In the course of recent years, the number and size of proteomic datasets made out of mass spectrometry-inferred protein identifications reported in the literature have become drastic. Tandem mass spectra are frequently checked against immense protein databases produced from genomes or RNA-Seq data for peptide identification. Most existing tools for mass spectrometry-based peptide identification consider a pair mass range against all peptides in a database. The atomic masses are like the precursor mass of the spectrum, making mass spectral data analysis moderate for large databases.

During this process, tremendous amounts of data is being delivered utilizing cutting edge innovations like Next Generation Sequencing Machines and high-throughput Mass Spectrometers. The generated big data make issues regarding storage, networking, and calculations. Keep in mind the end goal to process such data in a convenient way. High-performance computing, distributed computing is turning into a vital segment in biological science, bioinformatics , and computational biology. So, to process these huge data sets there is a need for big data analysis methods. The demand for big data analytic frameworks has been growing in the field of peptide identification over the recent years.

1.4.3 C-Ranker, Peptide, and Protein Identification

In a typical binary classification of the correct and incorrect PSMs, target PSMs are labeled as correct, or +1 and decoy PSMs are labeled as inaccurate or -1. The classi-

fier learns from the training dataset to assign either +1 or -1 class labels to PSMs. However, in peptide identification, the target PSMs are not trustworthy [Jian, Niu, Xia, Samir, Sumanasekera, Mu, Jennings, Hoek, Allos, Howard, et al., 2013]. Although some algorithms have been proposed for identifying high-quality PSMs, parameter selection remains a big challenge. C-Ranker aims to overcome this problem automatically.

Sequence database searching (for the large-scale dataset) [Matrixscience, Matrixscience] and de novo sequencing (new protein discovery) [Seidler, Zinn, Boehm, and Lehmann, 2010] are two standard approaches for peptide identification. The mass spectrometry (MS) based strategy coupled with sequence database searching has become the dominant method for peptide identification in large-scale proteomics studies. A variety of statistical and machine learning algorithms have been described to select these true PSMs in efficient manners among them in the C-Ranker.

In sequence database searching, an expansive number of PSMs are routinely produced, then again, just a small amount of them are correct. The undertaking of peptide identification is to pick correct ones from the database search yields. C-Ranker is the algorithm with scoring approach to rank all PSMs, and users can choose those top-scored PSMs as indicated by FDRs. The C-Ranker technique has been approved on various PSM datasets created from the SEQUEST database search tool. C-Ranker utilizes the primal SVM system and adapts to the weight of each PSM as a variable.C-Ranker is developed in Matlab; it comes with Windows and Linux distribution packages.

1.5 Prior Research

Prior to this thesis, C-Ranker Linux distribution or Windows installation is used for validating Shotgun Proteomics Datasets. The algorithm is designed to execute only

on a single machine. There are certain steps involved in the execution that includes read, solve, and write. C-Ranker with MATLAB runtime is a memory consuming algorithm that requires a high computing machine. In general, for a dataset having about 400,000 PSM records, it may take about five hours on a PC with CPU Intel Core i5 3.10GHz of 4 cores and Memory 8GB. Increases the data set size, i.e., PSM records, would increase the execution time of the C-Ranker. So, an idea was developed to decrease the execution time without compromising the reliability.

1.5.1 Problems with the Current Bioinformatics Algorithm(C-Ranker)

Many bioinformatics algorithms are parallelizable. Parallelism is not readily available in the original code. The user who needs to make the algorithm parallelizable may have to rewrite the entire code. C-Ranker also comes under this category where the parallelism is not readily available, Thus there is a need of framework to overcome this problem.

1.6 Proposed Solution

This thesis attempts to move from a theoretical approach to more practical approach by incorporating the data provided. Hadoop MapReduce and HDFS concepts are used to do the distributed processing. An algorithm was designed to use the distributed concepts of Hadoop and execute the C-Ranker on a cluster of nodes. This approach will reduce the execution time of C-Ranker without compromising the actual behavior of it. In this framework, C-Ranker input files are divided and distributed across all the nodes that are registered with the Hadoop node manager. Each mapper on the node in a cluster will consume the local data and execute the C-Ranker steps.

The main steps in the proposed solution include:

1. Create an algorithm/Framework to execute C-Ranker in distributed mode.

2. The framework is designed in such a way that it may work with other post database searching algorithms like C-Ranker with minimal changes.

3. Compare the generated distributed output of C-Ranker with the actual output of the C-Ranker, i.e., executed on a single node.

4. Make sure the algorithm is well executed on the set of predefined nodes.

1.6.1 Thesis Statement

This segment portrays the reasoning, extension and technique utilized as a part of finishing this research. It portrays why the research was led, and additionally some fundamental points of implementation; it also depicts what was and was not excluded in the research.

1.6.2 Research Question

How can the execution time of the C-Ranker algorithm be reduced without changing the execution behavior and its implementation? How to is it possible to have a better resource management while executing C-Ranker? Does the distributive or parallel execution fit here?

1.6.3 Area of Research

The purpose of the study is to implement the distributive framework to execute the C-Ranker in a distributive environment, including the preserving of the current execution behavior of C-Ranker. For this to be accomplished an algorithm is designed using Apache Hadoop [Hadoop, 2011] MapReduce and HDFS. The second part of this study

includes how the files are compared using MapReduce joins. The research also evaluates the overhead introduced by virtualization and Hadoop by comparing the performance of the C-Ranker application using physical and virtual machines, on LAN clusters, and with Hadoop. The results will be recorded in the form of tables, graphs and are utilized to analyze various parameters.

1.6.4 Scope of Thesis

This thesis focuses on quantifying and analyzing the results produced by C-Ranker using the distributive/parallel execution approach. Currently, C-Ranker is taking a long time to process the large data sets. So, an attempt has made to reduce this execution time. The research focuses only on executing the C-Ranker on multiple nodes at a single point in time. This thesis will not discuss changing the C-Ranker‘s actual implementation (code) to make it work on distributed computing.

1.6.5 Methodology

Initially, a domain and data flow diagrams were developed; then an architecture was designed. In the next stage, MapReduce classes and the driver classes were created. The last step was building up a system to present the outcomes. The outcomes will be most useful by including all the observations and results in Microsoft Excel spreadsheets. From these worksheets, different charts and tables can be delivered alongside access to the raw data.

1.6.6 Advantages

There are advantages to using Hadoop for distributed computing that will be explained in detail in the coming chapters. In short, the resource management is much easier when using Hadoop as the distributed platform. The C-Ranker distribution approach will

significantly reduce the current execution time for large data sets with minimal cost in the hardware and software infrastructure. In the case that there is an increase in the PSM data in the future, the framework that is designed can work with huge data sets and process them efficiently. The proposed framework also increases the reliability of storing and processing the PSM data using its distributive approach.

1.7 Organization of Thesis

This thesis reports results of C-Ranker generation using the distributive approach that is designed and developed in Chapter 4 with Apache Hadoop HDFS and MapReduce concepts. The rest of this thesis is organized as follows :

Chapter 2 *Characteristics Definitions Explained*

- Formally introduces about bioinformatics and its role, differences of post database searching algorithms like peptide prophet, percolator, and C-Ranker. The reason for choosing C-Ranker to execute it in distributed mode. Introduction of technology details like Hadoop framework, its advantages, and the reasons for selecting the Hadoop for this thesis.

Chapter 3 *Existing Problems and Planned Improvements*

- Formally describes the current solution that C-Ranker has and problems facing that. This chapter also summarizes the solution planned to solve the existing problem.

Chapter 4 *Proposals,Environment Setup, Design and Implementation*

- Formally describes all the proposals made, reasons for selecting the current proposal, the design, and the plans to implement the proposal and its implementation details.

Chapter 5 *Development Execution*

- Describes development, observations during the execution, what is being achieved? and the ways the problem has been solved.

Chapter 6 *Results and Summary*

- Show case results and summarize the total research.

Chapter 7 *Conclusions and Future Directions*

- Summarizes the entire research and concludes the work. The look at the different directions future research might follow.

Chapter 2

BACKGROUND

This chapter focuses on why we need peptide identification techniques, database search techniques for peptide identification, and post database technologies. It also review the reason for choosing only CRanker among other post database search algorithms. Lastly, it will outline why Hadoop and its components were shortlisted for this research work and details about the Apache Hadoop Components that are used in this thesis.

2.1 Terms and Definitions

It is bioinformatics that connects genomes, proteomes, and biological processes and permits us to study and concentrate on information from this data. Bioinformatics plays a significant role in identifying the proteins based on the peptide information.

2.1.1 Protein Identification Using Mass Spectrometry

Peptide identification is the key stride in protein identification and, more so, quantification. Various businesses and non-commercial database search instruments have been created to rank the PSMs given scoring functions and report the top-scored as target PSMs. Protein identification by mass spectrometry is widely used in biological research.

Protein identification by mass spectrometry (MS) is an imperative system in proteomics. Via seeking an MS range against a given protein database, the most coordinated proteins are sorted utilizing a scoring capacity, and the main one is frequently viewed as the accurately recognized protein. Mass spectrometry-based protein identification has turned

into a precious tool for explaining protein capacity. A few routines have been created for protein ID including, arrangement accumulation with masses of peptides or their parts, phantom library seeking, and de novo sequencing.

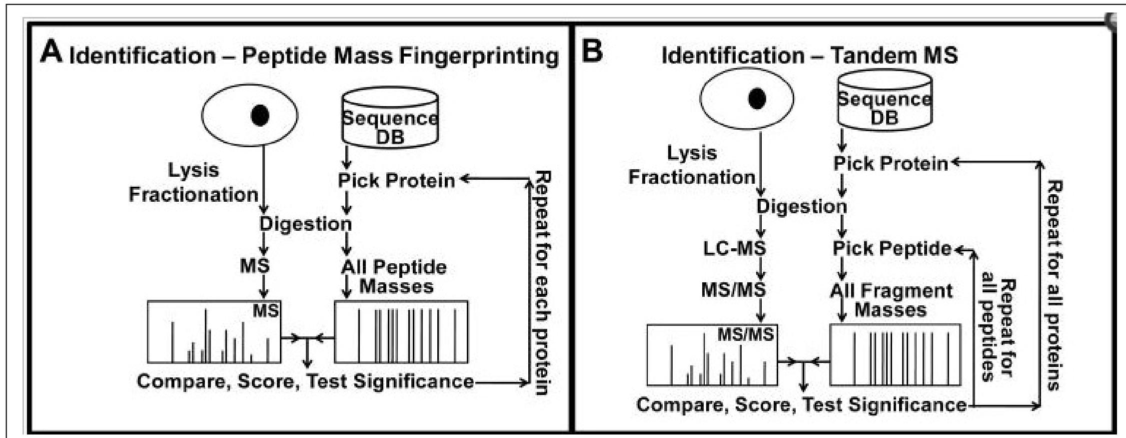


Figure 2.1: Protein Arrangement and Protein Gathering

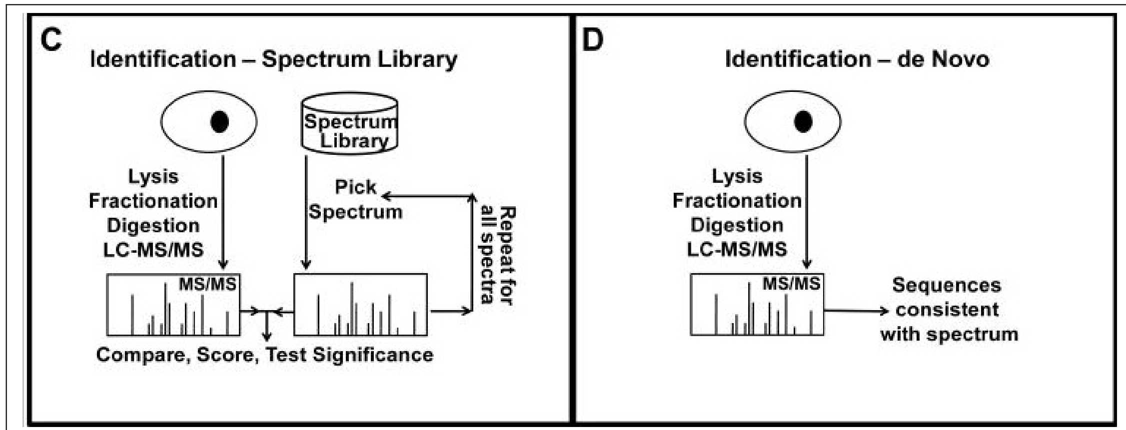


Figure 2.2: Seeking a Spectrum Library and de Novo sequencing

Peptide and protein identifications made in many mass spectrometry-based proteomic work processes first include gaining an arrangement of tandem mass (MS/MS) spectra. They then consists of cross-examining every spectrum against spectra anticipation from a rundown of protein groupings via search engines is performed. SEQUEST, Mascot, OMSSA, and X!Tandem are examples. The yield of these projects demonstrates

the best hypothetical peptide matches to the information spectra, which are then used to deduce the source protein that was available in the natural specimen.

2.1.2 Protein Search Engines Using Databases

2.1.2.1 SEQUEST

SEQUEST [Keller, Nesvizhskii, Kolker, and Aebersold, 2002] changes over the character-based representation of amino acid sequences in a protein database to fracture designs that are compared against the MS/MS range produced on the objective peptide. The calculation initially recognizes amino acid successions in the database that match the deliberate mass of the peptide. It then looks at fragmented particles against the MS/MS range and creates a preparatory score for every amino acid succession. A cross-relationship analysis is then performed on the main 500 preparatory scoring peptides by associating hypothetical recreated spectra against the experimental spectrum. Yield results are shown appropriately. To put it plainly, SEQUEST performs robotized peptide/protein sequencing through database searching of MS/MS spectra without the requirement for any manual succession interpretation. However, it can make use of translated grouping data if accessible.

2.1.2.2 Mascot

Mascot [Matrixscience.com, 2015] is a software search engine that uses mass spectrometry information to recognize proteins from peptide succession databases. It is broadly utilized via research facilities around the globe. It utilizes a probabilistic scoring calculation for protein identification that was adjusted from the MOWSE (for MOlecular Weight SEarch) [Pappin, Hojrup, and Bleasby, 1993] calculation. Mascot is openly accessible to use on the Matrix Science website.

2.1.3 Post Database Searching

In the course of recent years, MS/MS with database search has been utilized progressively for high-throughput investigation of complex protein tests. It has been made conceivable via computerized database search programming, such as SEQUEST. These applications compare every spectrum against those normal for every conceivable peptide acquired from a grouping database that have masses inside of a slip resilience of the antecedent particle mass. Every spectrum is then doled out to the database peptide with the most elevated general score, or set of scores, that reflects different parts of the fit in the middle of range and peptide. These scores help segregate in the midst of correct and incorrect peptide assignments to spectra and thus encourage discovery of false recognizable pieces of identifications.

There is a need for powerful and precise statistical models to evaluate the legitimacy of peptide identifications made by MS/MS and database search. Every peptide task to a spectrum is assessed for every single other task in the dataset, including some incorrect assignments. The technique applies machine learning strategies utilizing database search scores and the quantity of tryptic ends of the doled out peptides to recognize from the inaccurately appointed peptides in the dataset effectively. In this manner, it calculates the likelihood of being correct for every peptide tasked to a spectrum. The calculation applies a strategy to SEQUEST database search results for ESI-MS/MS spectra produced from a set of sample purified proteins. The algorithm show the registered probabilities are exact using this dataset with peptide assignments of known legitimacy and have high energy to separate in the middle of efficiently and inaccurately appointed peptides.

These statistical analysis algorithms guarantee to be of an incredible quality to high-throughput proteomics. Examples of the post database searching algorithms are Peptide-Prophet, Percolator, and C-Ranker.

These algorithms utilize semi-directed machines learning how to enhance the discrimination in the middle of the right and erroneous spectrum identifications. The matches from searching a decoy database give the negative illustrations to the classifier, and a subset of the high-scoring matches from the objective database present the positive cases.

2.2 PeptideProphet

PeptideProphet [Ma, Vitek, and Nesvizhskii, 2012] consequently approves peptide assignments to MS/MS spectra made by database search projects like SEQUEST. From each dataset, PeptideProphet learns distributions of search scores and peptide properties among right and wrong peptides and uses those distributions to process for every likely outcome that it is right. Applicable peptide properties incorporate the quantity of ends good with enzymatic cleavage (for unconstrained searches) and the amount of missed compound cleavages. It also incorporates the mass distinction of the antecedent ion, the vicinity of light or substantial cysteine (for ICAT tests), and the vicinity of an N-glycosylation theme (for N-glycosylation catch tests). PeptideProphet can be utilized as a second stride following the examination of MS/MS spectra created from any mass spectrometer and relegated peptides utilizing any number of database search programs. Normally, PeptideProphet analysis is trailed by ProteinProphet, which assembles peptides by their relating protein(s) to process probabilities that those proteins were available in the original sample.

2.2.1 Benifits and Issues of PeptideProphet

2.2.1.1 Benifits of PeptideProphet

PeptideProphet is widely used for Peptide identification. There are certain advantages:

- The PeptideProphet model is accurate.
- The PeptideProphet model is more sensitive than threshold model.
- The PeptideProphet model allows user to choose an error rate.

2.2.1.2 Issues with PeptideProphet

- It is impossible to compare results from different search algorithms and multiple tools.

2.3 Percolator

Percolator [Käll, Canterbury, Weston, Noble, and MacCoss, 2007] is an algorithm that uses a semi-directed machine calculating out how to enhance the discrimination in the middle of correct and incorrect spectrum identifications. The matches from searching a decoy database give the negative cases to the classifier and a subset of the high-scoring matches from the objective database give the positive illustrations. Percolator prepares a machine learning calculation called a support vector machine (SVM) to separate between the positive and negative matches by relegating weights to various components. Cases of elements incorporate Mascot score, antecedent mass blunder, part mass mistake, the number of variable modifications etc. The vector of components with their ideal weights is then utilized to re-rank matches from all queries frequently prompting enhanced affectability. The necessities for utilizing Percolator to re-rank the matches from a Mascot search are:

1. MS/MS search.

2. The search must incorporate the outcomes from an automatic decoy database search.

3. The search must contain no less than 100 questions

4. More than 100 databases are searched.

2.3.1 Benefits and Issues of Percolator

2.3.1.1 Benefits

- Percolator will normally give an advantageous change in sensitivity.

2.3.1.2 Issues

- In the event that there are different high scoring matches to a solitary query, the present methodology is to submit just the first rank match to Percolator.

- In the other way three main matches had Mascot scores of 60, 50, and 40, and the Percolator re-scored the rank one match to 54, the rank two and three matches would be re-scored to 45 and 36. That would maintain a strategic distance from peculiarities but it is not perfect.

2.4 CRanker

Although some algorithms have been proposed for identifying high-quality PSMs, parameter selection remains a big challenge. C-Ranker, which from now onward will be called as CRanker in this document, aims to overcome this problem automatically. It is a post-database searching software for identification of peptides. The target of CRanker is to identify correct PSMs output from the database searching tool SEQUEST. It was developed in Matlab and C.

By default CRanker uses nine attributes for representing a PSM data point, of which 5 comes from original sequest output file:

xcorr, deltacn, sprank, ions, hit mass;

The other four are calculated by CRanker:

enzN, enzC, numProt, xcorrR

In addition to the nine attributes, three other attributes, i.e., spectrum, protein, and peptide, are employed by CRanker to distinguished PSM data and calculate the appended features.

In the research conducted by Dr. Zhonghang Xia, peptide identification by CRanker [Liang, Xia, Niu, and Link, 2014] is proved as the most efficient algorithm for post database search. In sequence database searching an extensive number of PSMs are routinely produced but only a fraction of them are correct. The errand of peptide identification is to pick the correct ones from database search yields. In the binary classification "great" PSMs are allocated to the class of "right" or "+1" and "terrible" PSMs to the class of "wrong" or "-1". Distinctive from normal classification issue, the objective PSMs are not reliable i.e., "+1" marks (relating to target PSMs) are not dependable.

2.4.1 Reasons for Choosing CRanker

Based on the below research results in Table 2.1, CRanker is probably most efficient compared to PeptideProphet and Percolator.

Performance of CRanker is evaluated by comparing the algorithm with PeptideProphet and Percolator based on PSMs generated from the SEQUEST search engine.

Table 2.1 demonstrates that the aggregate quantities of PSMs distinguished by CRanker, PeptideProphet, and Percolator over all datasets (preparing and test) at FDR =

Data	Method	Total	TP	FP
ups1	PeptideProphet	582	566	16
	Percolator	450	438	12
	CRanker	601	585	16
yeast	PeptideProphet	1481	1443	38
	Percolator	1429	1394	35
	CRanker	1491	1455	36
orbit-mips	PeptideProphet	34035	33233	802
	Percolator	33846	33053	793
	CRanker	35006	34123	880
orbit-nomips	PeptideProphet	36542	35673	869
	Percolator	36096	35230	866
	CRanker	37337	36416	921

Table 2.1: Target PSMS Output by PeptideProphet, Percolator, and CRanker.

Data	PeptideProphet	CRanker	Overlap
ups1	582	576	509
orbit-mips	34035	34273	32243

Table 2.2: Overlap of PeptideProphet and CRanker.

0.05. Obviously, CRanker can recognize more PSMs than the two algorithms. By considering the recognized PSMs among the three algorithms CRanker and PeptideProphet has the same overlapping.

Table 2.2 demonstrates the overlapping of aggregate PSMs distinguished by PeptideProphet and CRanker as 88.4 % and 94.8% on UPS1, and "orbitmips", respectively. The outcome shows the covering degree is around 90% and the larger part of PSMs accepted by CRanker were also approved by PeptideProphet.

CRanker utilizes the primal SVM system and adapts to the weight of each PSM as a variable. The execution of CRanker outperformed the benchmarked calculations of PeptideProphet and Percolator for a greater variety of PSM datasets. The exploratory studies show CRanker outperforms the other two by recognizing more targets at the same FDRs.

Based on the above reason, CRanker is chosen to execute on the distributed framework using Hadoop Framework. A Linux version of CRanker is used to make it run on the Apache Hadoop framework.

2.5 Why Apache®Hadoop™Framework

This section summarizes the reasons for choosing Hadoop as the distributed framework for this thesis and also what Apache Hadoop is.

2.5.1 Top Reasons to Consider Hadoop

Parallelizing the applications that can run on multiple resources, each of which executes the sequential application of a subset of the inputs, requires additional software to manage and monitor job distribution and should possibly offer fault tolerance. There should be an automated process of transferring or sharing large data sets and selecting appropriate application binaries for a variety of environments or existing services. Managing the creation and submission of a large number of jobs to be executed in parallel and recovering from possible failures are the important points to be considered while choosing the distributive framework. Hadoop offers all these services very efficiently, and it also customizes these services as per user needs using MapReduce programming. Based on several case studies, it has been proven that big data analysis has been much easier and effective using Hadoop Framework. This is why Hadoop has been chosen to make the CRanker execution distributed.

2.5.2 Apache®Hadoop™

Apache®Hadoop™[White, 2012] is an open source software framework that enables distributed processing of large data sets across clusters of commodity servers. It is intended to scale up from a solitary node to a large number of nodes, with a high level of

adaptation to internal failure (fault tolerance). Instead of depending on the top of the line hardware, the resiliency of these clusters comes from the software's ability to detect and handle failures at the application layer.

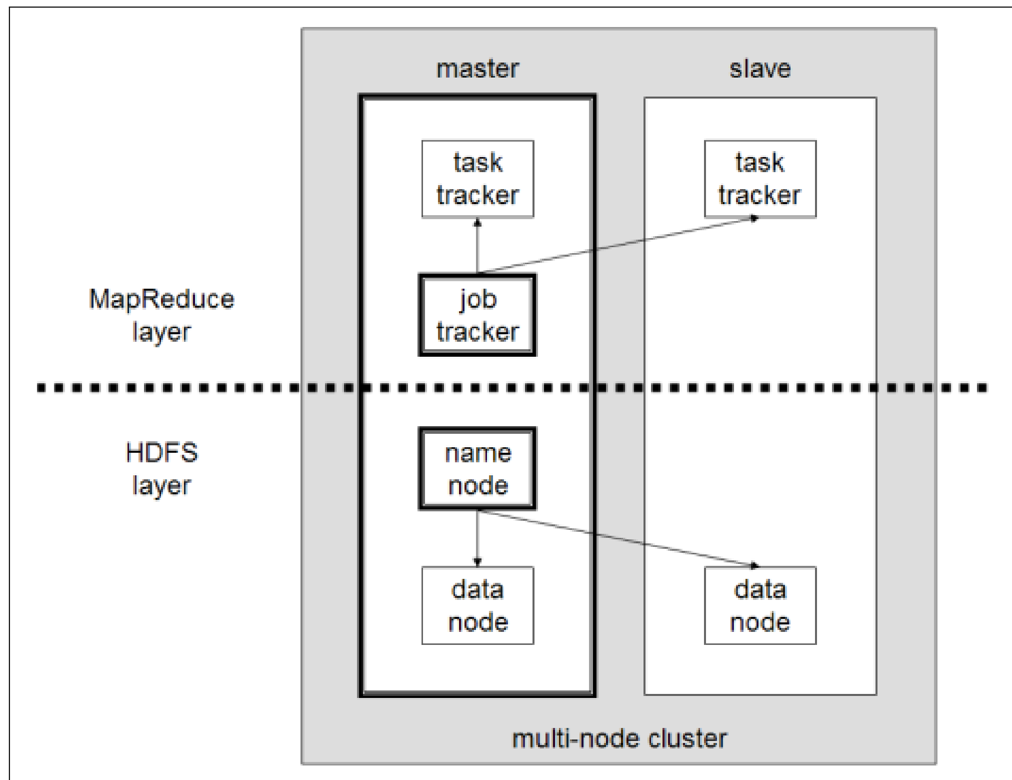


Figure 2.3: Hadoop execution architecture.

The project incorporates these modules:

- Hadoop Common: The typical utilities that are backing the other Hadoop modules
- Hadoop Distributed File System (HDFS™): A distributed file system that provides high-throughput access to application data.
- Hadoop YARN: A framework for scheduling the jobs and cluster node management.
- Hadoop MapReduce: A YARN-based framework for parallel processing of vast data sets.

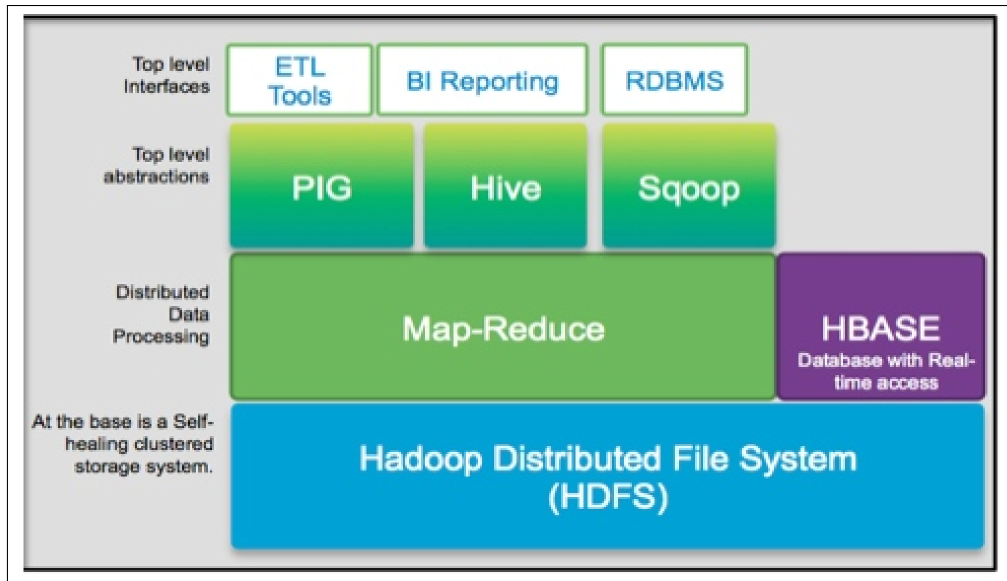


Figure 2.4: Hadoop architecture.

The term "Hadoop" now alludes to the base modules above, as well as to the "environment", or gathering of extra programming bundles that can be introduced on top of or nearby Hadoop; for example, Apache Pig, Apache Hive, Apache HBase, Apache Spark, and others.Apache Hadoop's HDFS and MapReduce components were motivated by Google papers on their Google File System [Ghemawat, Gobioff, and Leung, 2003] and Mapreduce. The Hadoop system itself is for the most part written in the Java programming language, with some part of a native code in C and command line utilities composed as Shell script. For end-clients, however, MapReduce Java code is basic. Any programming language can be utilized with "Hadoop Streaming" to implement the "Map" and "Reduce" parts of the client's program. Other related projects uncover other higher-level user interfaces and APIs. This thesis uses HDFS and MapReduce as its platform for distributed processing. The HDFS and MapReduce will be discussed in the following sections.

2.6 Apache Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is intended to store substantial data sets accurately, and to stream those information sets at high data transmission to client applications. In a large cluster, a large number of servers both host straightforwardly connected capacity and execute client application undertakings. By distributing storage and computation crosswise over numerous servers, the resource can develop with interest while staying prudent at each size.

HDFS stores file system metadata and application information independently. Distributed file system metadata, such as PVFS [Zhu and Jiang, 2006] and GFS [Ghemawat et al., 2003], HDFS stores metadata on a dedicated server node, called the NameNode. Application data is put away on different servers called DataNodes. All servers are completely associated and correspond with one another utilizing TCP-based protocols. Unlike PVFS, the DataNodes in HDFS do not depend on data security mechanisms such as RAID to make the information tough. As GFS, the file content is reproduced on numerous DataNodes for dependability. While guaranteeing data durability, this method has the included point of interest that data transfer bandwidth capacity is multiplied and there are more open doors for finding computing close to the required data.

2.6.1 HDFS Architecture

This section discusses the HDFS architecture.

2.6.1.1 NameNode

The NameNode is the center point of the HDFS file system. It keeps the catalog tree of all files in the file system and tracks where the file data is kept. It does not store

the information of these files itself. Client applications converse with the NameNode at whatever point they wish to find a file, or when they need to include/duplicate/move/erase a record. The NameNode reacts the effective demands by giving back a list of pertinent DataNode servers where the data lives.

The NameNode is a single point of failure for the HDFS cluster. HDFS is not presently a high availability system. At the point when the NameNode goes down, the record framework goes down. There is a discretionary secondary NameNode that can be facilitated on a different machine. It just makes checkpoints of the namespace by blending the altered files into the image record and does not give any genuine redundancy. Hadoop 0.21+ has a BackupNameNode that is a piece of an arrangement to have an HA name service, yet it needs dynamic commitments from the individuals who need it (i.e. user) to make it highly available. So, it is the user‘s responsibility to use the high configuration machines to host the NameNode.

2.6.1.2 Data Node

A DataNode stores data in the Hadoop Distributed File System. A working file system has more than one DataNode, with data replicated crosswise over them. On startup, a DataNode interfaces with the NameNode; turning until that service comes up. It then will ask the NameNode for file system operations. Client applications can talk straightforwardly to a DataNode once the NameNode has given the location of the data. Correspondingly, MapReduce operations cultivated out to TaskTracker occasions close to a DataNode talk specifically to the DataNode to get to the files. TaskTracker occasions can undoubtedly be conveyed on the same servers that host DataNode examples so that MapReduce operations are performed near the data. DataNode occurrences can converse with one another,

which is their main purpose when they are replicating data. There is usually no compelling reason to utilize RAID stockpiling for DataNode data. In fact, that data is intended to be recreated over different servers, instead of numerous circles on the same server. A perfect arrangement is for a server to have a DataNode plus a TaskTracker. That will permit each TaskTracker 100 % of a CPU and separate disks to write and read data.

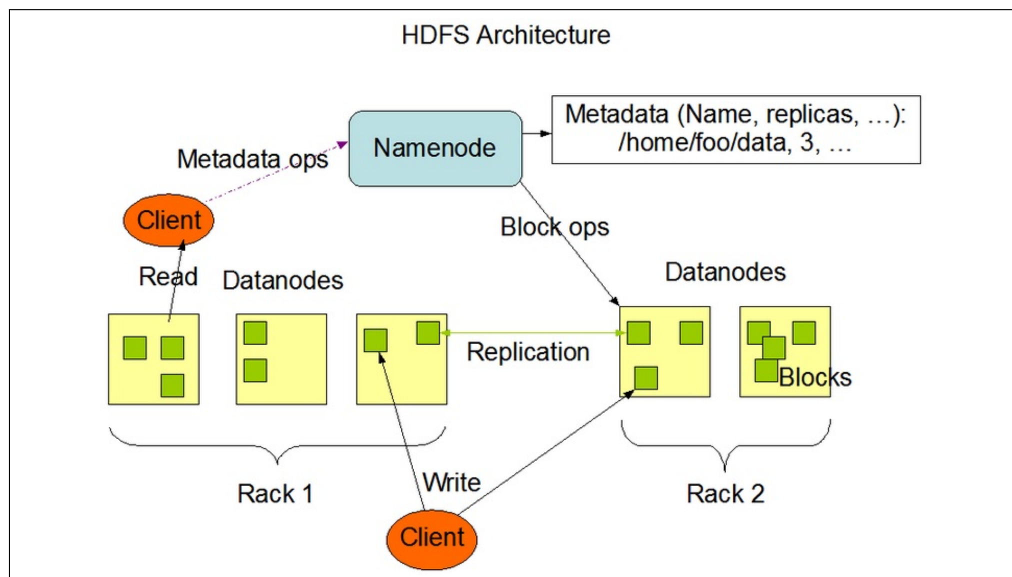


Figure 2.5: HDFS Architecture (Source Apache Hadoop).

2.6.1.3 Block Replication

HDFS is intended to store huge files crosswise over machines in a vast cluster dependably. It stores every file as a grouping of blocks; all blocks in a file aside from the last block are the same size. The blocks of a file are reproduced for adaptation to fault tolerance. The block size and replication component are configurable per file. An application can indicate the quantity of copies of a file. The replication factor can be determined at file creation time and can be changed later. Files in HDFS are written once and have entirely one writer at any time. The NameNode settles on all choices in regards to replication of

blocks. It occasionally gets a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Reception of a Heartbeat infers that the DataNode is working legitimately. A Blockreport contains a list of all blocks on a DataNode.

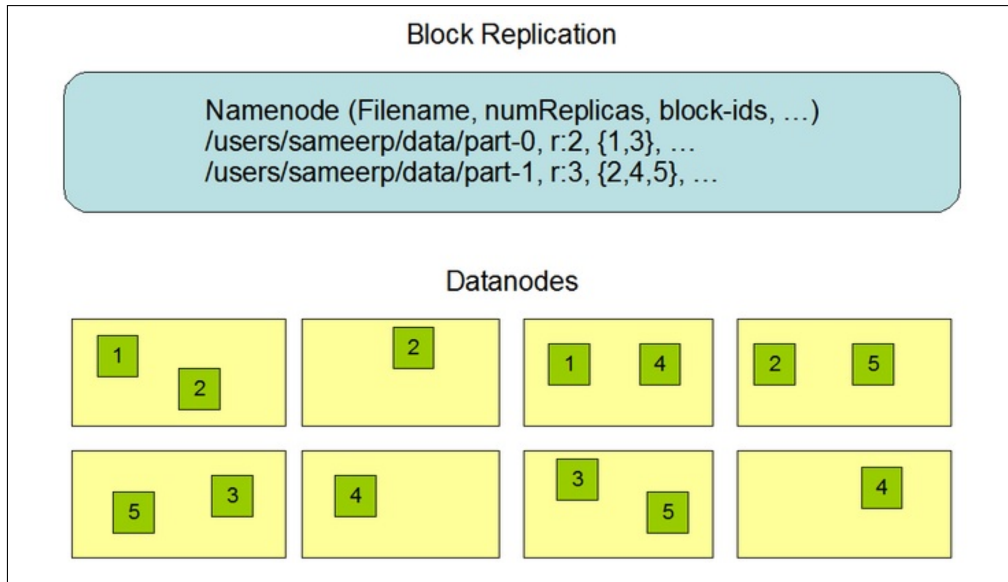


Figure 2.6: HDFS Block Replication (Source from Apache Hadoop) .

2.6.1.4 HDFS Client

Client applications get to the file system utilizing the HDFS client, a library that fares the HDFS file system interface. Like most ordinary file systems, HDFS bolsters operations to read, write, and delete files, and operations to delete and create directories. The client references files and indexes by way of the namespace. The client application does not have to realize that file system metadata and storage are on diverse servers, or that blocks have various replicas. At the point when an application reads a record, the HDFS client first approaches the NameNode for the rundown of DataNodes that host replicas of the blocks of the file. The list is sorted by the topology of network separation from the client. The client contacts a DataNode specifically and demands the exchange of the

desired block. At the time a client can first request the NameNode pick DataNodes to host replicas of the first block of the record. The client sorts out a pipeline from node-to-node and sends the data. When the first block is filled, the customer demands new DataNodes to be chosen to host copies of the following block. Another pipeline is sorted out, and the client sends the further bytes of the file. The decision of DataNodes for every piece is likely to appear as something else. The communications among the customer, the NameNode, and the DataNodes are represented in the Figure 2.7.

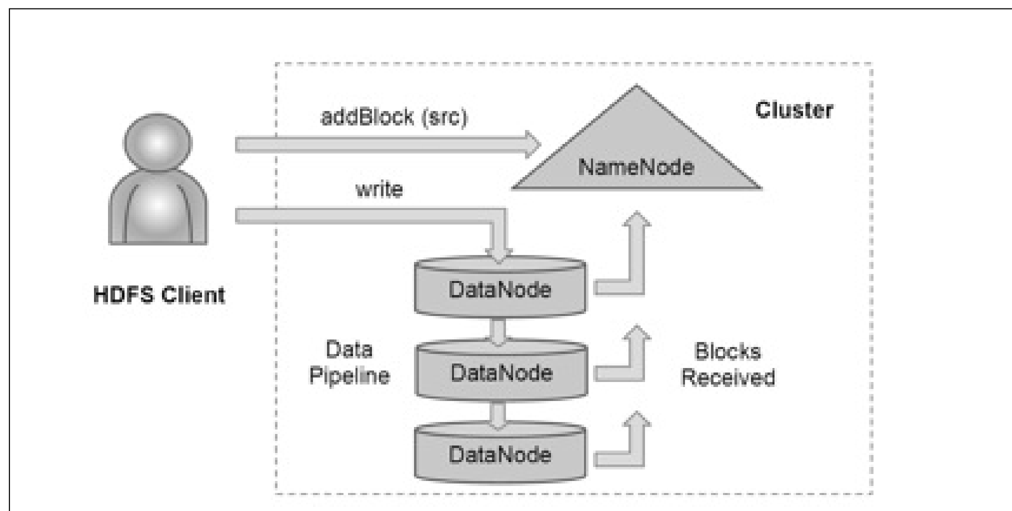


Figure 2.7: Hadoop Client Creates a File.

2.6.1.5 Secondary NameNode

Secondary NameNode is a deceptive name that some may inaccurately translate as NameNode and it is used when the essential NameNode gets disconnected from the picture. The Secondary NameNode routinely joins with the essential NameNode and assembles previews of the essential NameNode‘s registry data, which the framework then moves to local or remote indexes. These check-pointed pictures can be utilized to restart a failed essential NameNode without needing to replay the entire journal of the file system actions,

then to alter the log to make an up and coming index structure. Since the NameNode is the single point for storage and administration of metadata, it create a bottleneck for supporting an enormous number of files, particularly an expansive number of small records or files. HDFS Federation, another expansion, intends to handle this issue to a certain degree by permitting numerous namespaces served by separate NameNodes.

2.7 Apache Hadoop MapReduce

This section summarize the MapReduce concepts of Apache Hadoop

2.7.1 What is MapReduce?

MapReduce is a basic programming model for handling huge data sets in parallel. The essential thought of MapReduce is to gap an undertaking into subtasks, handle the subtasks in parallel, and total the after effects of the subtasks to shape the last output. Programs written in MapReduce are naturally parallelized; software engineers should not be worried about the execution points of interest of parallel handling. Instead, software engineers compose two capacities: Map and Reduce. The mapping stage reads the information (in parallel) and distributes the data to the reducers. Assistant stages, for example, sorting, partitioning, comparison and consolidating values can likewise occur between the Map and Reduce stages.

MapReduce programs are, for the most part, used to process extensively large files. The input and output for the map and reduce functions are communicated as key-value pairs. The utilization and subtle elements of key-value sets are talked about in sections on the map and reduce areas underneath.

A Hadoop MapReduce program likewise has a part called the Driver. The driver handles initializing the job with its subtle setup elements and indicating the mapper and

the reducer classes for the job. It also advises the Hadoop stage to execute the code on the predetermined input file(s) and to control the location of the output files.

MapReduce can exploit locality of data, preparing it on or close to the storage assets so as to reduce the separation over which it must be transmitted. MapReduce programs are called jobs in Hadoop.

2.7.1.1 InputReader or RecordReader

The InputSplit is characterized as a slice of work, yet does not portray how to get to it. The RecordReader class stacks the data from its source and converts it into (key, value) sets suitable for reading by the Mapper. The RecordReader instance is specified by the InputFormat. The default TextInputFormat and InputFormat gives a LineRecordReader, which treats every line of the info file as a new value. The key connected with every line is its byte offset in the file. The RecordReader is invoked over and over on the input until the whole InputSplit has been expended. Every initiation of the RecordReader prompts another call to the map system for the Mapper.

2.7.1.2 Mapper

MapReduce operates exclusively on <key, value> pairs.

The input output structure is as follows:

- Job Input: <key, value> pairs
- Job Output: <key, value> pairs

The motivation behind the map stage is to sort out the data in an arrangement for the processing done in the reduce stage. The data to the map capacity is as key-value pairs, despite the fact that the information to a MapReduce program is a file or file(s). Naturally,

the value is a data record and the key is the balance of the data record from the earliest starting point of the data file.

The output comprises of a collection of key-value pairs which are input for the reducer function. The content of the key-value pairs relies on upon the particular usage.

For instance, a typical beginning program implemented in MapReduce is to count words in a file. The input to the mapper is every line of the file, while the output from every mapper is a situated of key-value pairs where single word is the key and the number one is the value.

To improve the processing limit of the map stage, MapReduce can run a few indistinguishable mappers in parallel. Since each mapper is the same, they create the same result presently Map capacity.

2.7.1.3 Reducer

Each reduce function forms the intermediate values for a particular key created by the map function and creates the output. Essentially there exists a one-one mapping in the middle of keys and reducers. A few reducers can keep running in parallel since they are independent of each other. The quantity of these reducers is chosen by the client. Of course, it is one by default

2.7.1.4 MapReduce Data Flow

At the point when the mapping stage has finished, the intermediate (key, value) pairs must be exchanged between nodes to send all qualities with the same key to a single reducer. The Reduce tasks distributed across the nodes where the mapper executed. That is the main communication step in MapReduce. Individual map undertakings do not exchange data with each other, nor are they mindful of each other‘s presence. Correspondingly,

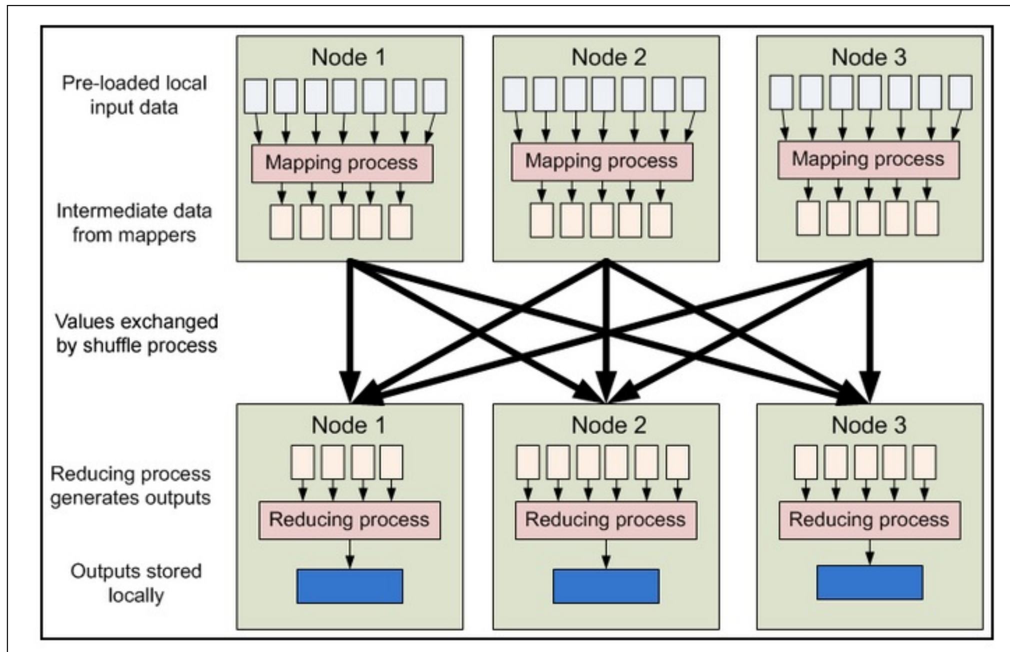


Figure 2.8: HighLevel MapReduce Pipeline

distinctive reduce tasks do not communicate with each other. The user never expressly marshals data starting with one machine then onto the next; all data exchange is taken care of by the Hadoop MapReduce stage itself, guided certainly by the diverse keys connected with values. That is a crucial component of Hadoop MapReduce‘s reliability. In the event that nodes in the cluster falter, tasks must have the capacity to be restarted. In the event that they have been performing indications, e.g., speaking with the outside world, then the mutual state must be restored in a restarted assignment. By taking out correspondence and indications, restarts can be taken care of all the more smoothly. The data flow is explained in the Figure 2.9

.

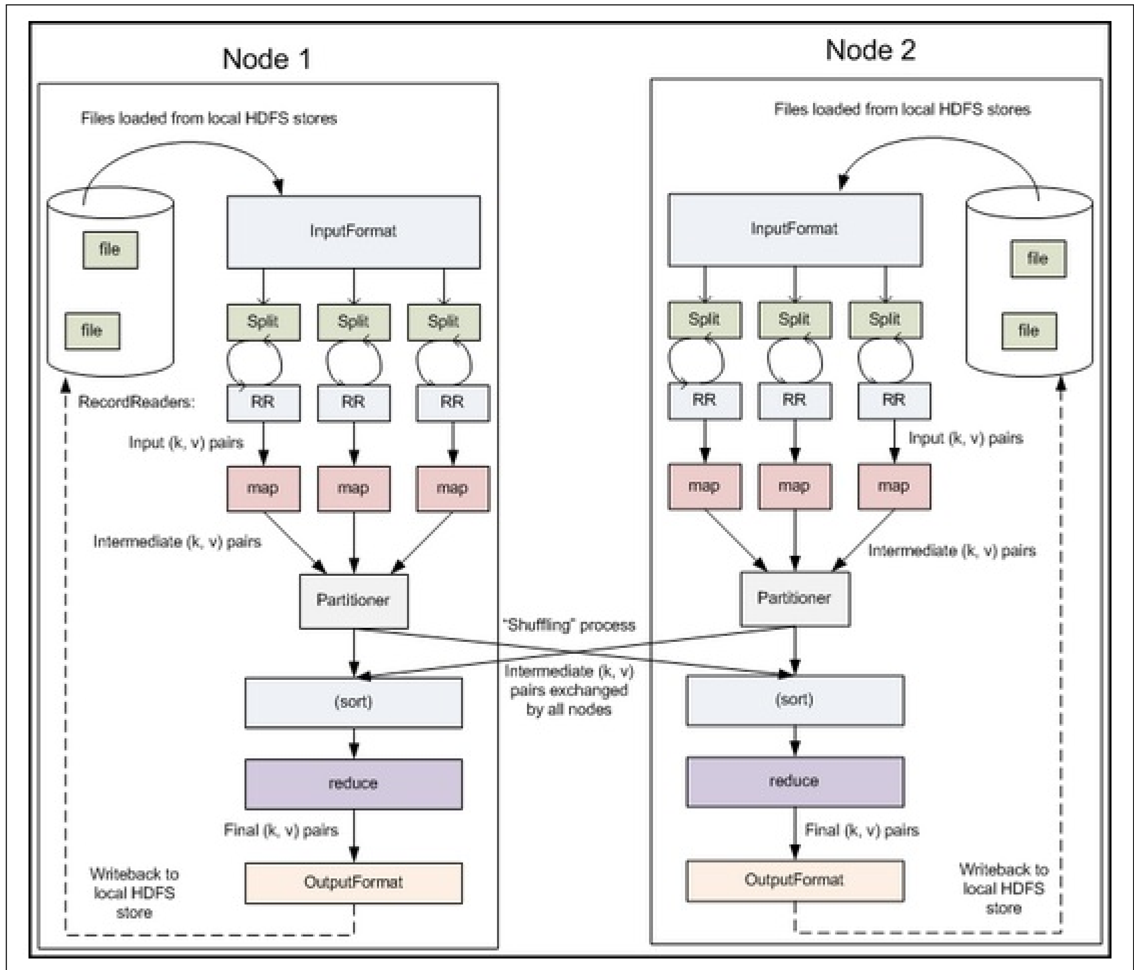


Figure 2.9: MapReduce Data Flow

Chapter 3

EXISTING PROBLEMS AND PLANNED IMPROVEMENTS

Chapter 3 summarizes the current solution available and the issues faced in using process. It will also explain the reason for choosing Apache Hadoop over other Hadoop distributions and the motivation behind the architecture that was proposed.

3.0.2 Existing Solution to Execute CRanker

The following paragraphs describe the CRanker execution from the memory perspective to give an idea of the time and memory consuming activities. CRanker is a post-database searching software for peptide identification. The goal of CRanker is to identify correct PSMs output from the database searching tool SEQUEST. It was developed in Matlab and C. There are certain steps involved in the execution and they are as follows:

1. "cranker read.exe" Read data of PSM records.

2. "cranker solve.exe" Calculate scores for each PSM.

3. "cranker write.exe" Put out the results.

All of the above steps have to be execute manually in a terminal window in the Linux operating system by issuing the appropriate command. The execution of the above steps must be sequential and should be done one after the other as the output of the initial step would be the input for the next step.

In step 1, CRanker reads and load the data in the text file i.e. "inputFileName.txt" during this process all the data in the txt file is read and loaded into the main memory. For the smaller data sets it may not be a problem as the entire data would fit into the main memory at one go. In the main memory the data that was read from the "inputFileName.txt" contains the raw PSM data. The raw PSM data that was consumed by the CRanker instance would be used to produce the "inputFileName.mat" that is stored in the current directory.

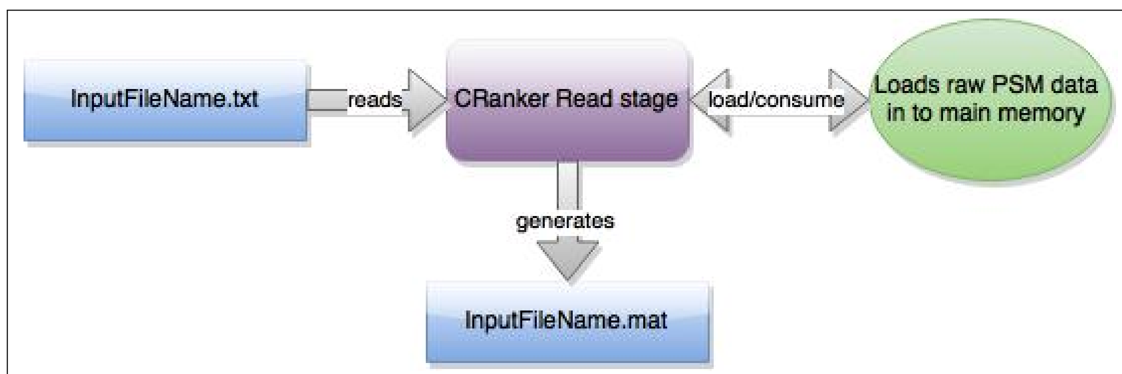


Figure 3.1: CRanker Read Flow

In step 2 of CRanker, solve C-Ranker trains a classification model and calculates the score for each PSM record, trained model, and calculated scores are stored in a file “inputFileName _score.mat”. The values of scores follow in the interval [–1, 1]. A PSM with higher score indicates that it is more likely to be correct. During this process the "inputFileName.mat" is again read and loaded into the main memory and where it will get processed to calculate the scores. The time taken to read, load and process the "inputFileName.mat" depends again on the size of the "inputFileName _score.mat" file. For the larger data sets the .mat file would be in the large size as well and this may cause delay in execution of CRanker. In fact, this is the most time consuming step in the overall CRanker execution even for the smaller PSM data sets with 2 MB.

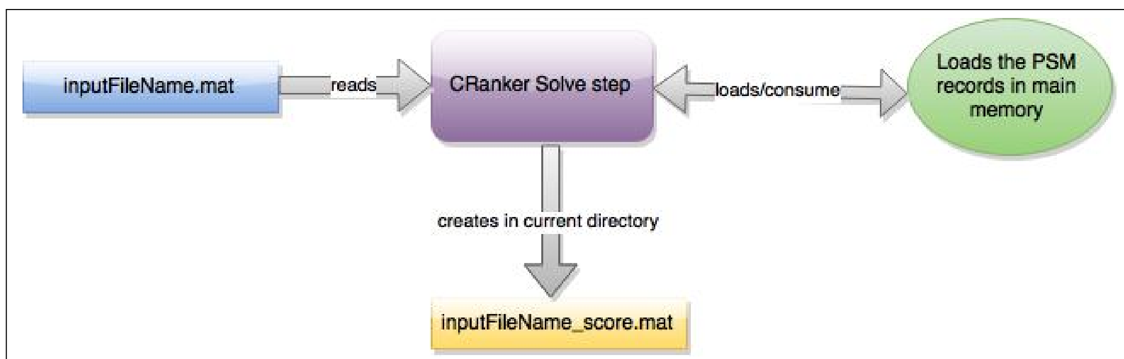


Figure 3.2: CRanker Solve Process

In step 3 CRanker, output identified reliable PSMs to a text file named "inputFileresult dd –mm –yyyy.txt", where dd –mm –yyyy indicates the current date. The output file is again stored in the current directory. During this process, two files named "inputFileName.mat" and "inputFileName _score.mat"are read and loaded in to the main memory, then the correct PSM is written to the output file, the read and processing depends on the size of the input files that are read by CRanker, the larger the file size the greater the burden on the memory and the longer it takes to complete the process.

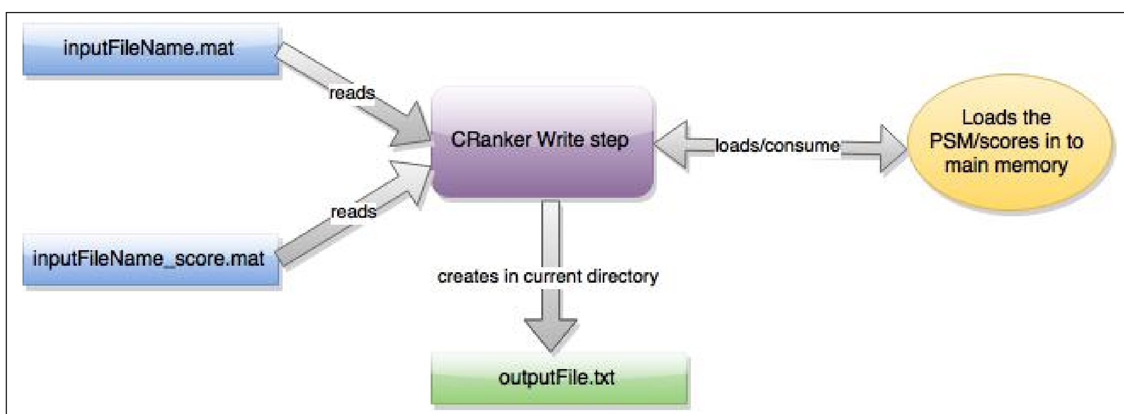


Figure 3.3: CRanker Write Process

During the execution of CRanker an interesting observation was made about MATLAB Compiler Runtime (MCR) which is used to enable the execution of compiled MAT-

LAB applications or components(CRanker in this case). This consumes the more memory than the actual CRanker algorithm. There is an exponential increase in the memory consumption based on the size of the input PSM data sets.

3.0.3 Existing Issues with CRanker

The main problem in the current execution of CRanker is the time it takes to process to large data sets. The other problems include memory exceptions while running on low hardware configuration machines and incomplete execution due to failure in one of the steps in CRanker.

The entire execution of CRanker is confined to a single machine in the current implementation. Failure in any one step of the CRanker halts the entire execution flow. There are also some missing characteristics observed during the study of the CRanker for this thesis. They are as follows:

- **Resource sharing**: CRanker doesn't talk about the hardware and software resource sharing to reduce the storage and execution costs.

- **Openess**: There is no information about extending the CRanker or how it can be coupled with respect to the software and hardware changes.

- **Concurrency**: There is no concept of multiprogramming and multiprocessing to handle the large PSM data sets. The multiprocessing is the current industry buzz word to reduce the execution costs.

- **Scalability**: No information about the how the CRanker handles growth. How to handle the execution timing using caching and data replication.

PSM Data Set	Size (KB)	Memory Used (%)	Time to Execute (hrs rounded)
pbmc_ orbit_ mips.txt	11221	60.5	7
pbmc_ orbit_ nomips.txt	12816	67	9
pbmc_ velos_ mips.txt	31422	76	11
pbmc_ velos_ nomips.txt	48486	81	14

Table 3.1: PBMC data execution on CRanker

- **Fault tolerance**: If computers fail during the processing, does CRanker has any mechanism to handle the failure?

For example, it takes 12 hours to process a PSM data set of size 50 MB and during the processing at 11th hour the system crashed, is there any way to handle these kind of scenarios? As per the observation, there is no such feature in the CRanker as it is currently aligned only to a single machine.

As a reliable and dependable application CRanker must adhere to all the above specified properties which any application can do. However, this thesis is not about re-implementing the CRanker so this research tries to address most of the above discussed concerns by designing a framework.

Coming to the main focus of the CRanker which is reducing the execution time and memory management, while running the CRanker on Human Peripheral Blood Mononuclear Cells (PBMC) data sets the CRanker memory and execution time are recorded and shown in the Table 3.1.

The recorded results are based on the CRanker execution on a computer with Intel i-5 processor which has 4 GB RAM and Linux Ubuntu as a operating system.

During execution, the processor and RAM are being shared by the other applications related to operating system that is running on the machine, so execution of CRanker

maybe delayed. However, based on the multiple times of execution under idle condition the values have been averaged. The aim is to design a framework to have a better memory management, reduce execution time and adhere to the principle characteristics that are specified above.

3.1 Why Only Use Apache Hadoop for CRanker Execution

The following section specifies why Apache Hadoop distribution was chosen for the implementation

Apache Software‘s open source data storage and processing framework are an alluring alternative. Not only does the platform offer both distributed processing and computational capacities at a moderate ease of use, but it is also ready to scale to meet the foreseen exponential increment in data. The data is produced by versatile innovation, online networking, the Internet of Things, and other rising advances. These focal points, alongside solid informal and prominent usage by organizations, for example, Facebook, Yahoo, and various Fortune 500 giants is driving the selection of Hadoop.

There are other Hadoop distributions from Hortonworks, Cloudera etc. However, the reason for choosing the Apache Hadoop is, it comes under General public license. This is open source and available only with Linux distribution.

Not only the above statements but the following case studies strongly motivated us to choose Hadoop as a distributed platform for this thesis.

3.1.1 Case Studies Which Motivated the Choice of Apache Hadoop

When thinking about the execution of CRanker on the distributed platform I got a chance to explore various case studies of Hadoop and the following are the ones which impressed me the most.

3.1.1.1 RNA-sequencing Analysis with Myrna

Sequencing output approaches many gigabytes every day, there is a developing requirement for effective programming for investigation of transcriptome sequencing (RNA-Seq) data. Myrna is a distributed cloud computing pipeline for figuring differential quality expression in huge RNA-Seq datasets. The detailed usage of Myrna can be referred at [Langmead, Hansen, Leek, et al., 2010]

3.1.1.2 Newyork Times

The New York Times has chosen to make all general space articles from 1851–1922 publicly accessible for nothing out of pocket. These articles are all pictures checked from the first paper. Indeed from 1851–1980, each one of the 11 million articles are accessible presently in PDF design.A great deal of work is required to produce a PDF adaptation of an article. Every article is made up of various smaller TIFF pictures that should be scaled and stuck together in a sound fashion. The New York Times rented 100 EC2 virtual machines for a day to convert 11 million scanned articles to PDF [Zaharia, Konwinski, Joseph, Katz, and Stoica, 2008].

3.1.1.3 A Bioinformatics Case Study by DDP Option Comparison for Sequence Mapping

A prevalent bioinformatics instrument for group mapping, called CloudBurst, shows how distinctive DDP design mixes could be utilized for the same tool and think about their representations. The more information about this can be referred in the article [Wang, Crawl, Altintas, Tzoumas, and Markl, 2013].

These successful case studies inspired the use of Apache Hadoop as the distributed execution platform for the CRanker.

Chapter 4

PROPOSALS,ENVIRONMENT SETUP, DESIGN, AND IMPLEMENTATION

Chapter 4 discusses the details of the various proposals made to reduce the CRanker execution time, better memory management, efficiency in terms of money, and resource utilization. It also discusses the idea of parallelizing CRanker application execution, terms needed in setting up the required environment, description of architecture, design, and implementation.

4.1 Proposals Made

The main goal of this thesis is to reduce the execution time of the CRanker. It is necessary to identify the possible solutions that are available to make this possible. After some brainstorming, the solution was discovered to increase the computing power of the machine where CRanker actually executes using High-Performance Computing Center for execution and GPU computing. The elaboration of these ideas are as follows:

4.1.1 Observations Made on Increasing the Computation Power

There is unquestionably some linear computational power increase [Claasen, 1999] in central processing unit (CPU) innovation that can be normal later on. On the other hand, the greater part of today's rate increment is as of now in view of multi-core CPU structural architecture. Certain applications, for example, the distinguished identification of peptides utilizing CRanker, will require altogether more computational force than the flow change in CPU technology can offer. In a few regions future applications might be conceivable

if the computational force can be expanded by no less than two orders of magnitude. An increment in computational force is subsequently crucial in order to remain aware of current scientific advancements.

For some standard applications, for a drawn out stretch of time, software engineers did not need to stress over execution. Present day CPU producers have enhanced equipment speed adequately. For a long time the only legitimate way to deal slow equipment was to wait for CPUs to become faster. Moore's Law [Bondyopadhyay, 1998], which states that processing power doubles every 18 months, characterized the whole decade of the 1990s. This was a consequence of changes in the entryways per-bite the dust check or transistors per territory (the fundamental characteristic of CPUs that Moore based his law in light of), the quantity of instructions executed per time unit (clock speed) and the alleged instruction level parallelism (ILP), fundamentally importance the likelihood of performing more than only one single operation inside of the same clock cycle (for instance, summing up two registers and replicating the outcome to another register).

Today, this unnecessary increase in clock speed execution is over. Lately CPU producers have begun offering CPUs with more computational cores rather than quicker CPUs. As of 2003, the laws of material science put an end to the practice of incrementing clock speed. One basic explanation behind this is that multiplying the clock speed means dividing the electrical sign per clock cycle, which requires the physical size of the CPU to be twice what it is right now. On the other hand, diminishing the physical measurements of CPUs is restricted by the diffraction furthest reaches of the lithographic techniques utilized for chip producing.

There are different techniques that are utilized to an build execution that can, in

any event, partially make up for the constrained increment in clock speed. These are, for instance, refined ILP plans, theoretical execution, and branch expectation, which are the main remaining principles for execution change separated from the gate count. These systems are what producers concentrate on today, bringing about feature rich CPUs that are outfitted with an expanding number of computational cores. While an expanded clock cycle naturally accelerates a current application, this is not the case with extra CPUs or cores. The degree that the application can benefit by extra cores relies on the computational issue, the algorithm used to fix it, and the application architecture. The performance improvement is then absolutely subject to the developer, who needs to create enhanced code with a particular end goal to get the greatest conceivable speedup.

For the following decade, the constraining factors on execution will be the capacity to compose and revamp applications to scale at a rate that stays aware of the speed of the core count. Laying out applications for concurrency may be the 'new area'of adaptability in multi-core systems.

It is always a better approach to use the cores on the CPU instead of using high-power computational resources. Based on the current programming methodologies and implementations, it is not easy to execute the program on all the available cores of the CPU. Not to mention, CRanker uses only a single core while executing on a normal computer, so this option is ruled out. High-Performance Computing is the other option that was considered during the early stages of this thesis, and those observations are below.

4.1.2 Observations Made on Using High-Performance Computing

High-Performance Computing [Dowd, 1993] refers to the practice of aggregating computation power in a way that delivers much higher throughput than one could get out

of a normal desktop computer or workstation. This computation is generally used to solve complex problems in science, technology, engineering, and business.

These are exceptionally intriguing machines by ideals of the advances inside them, and the scale at which they are manufactured; sometimes an enormous number of processors make up a solitary machine. Therefore, supercomputers are extravagant, with the main 100 (or somewhere in that vicinity) machines on the planet costing upwards of $20M each.

The computer performance is determined by the hardware components used inside it. All the components that can be found inside a personal computer can also be found inside high-performance computers, but there will be a greater amount of them. The ones that can be found in small and medium-sized organizations today are truly clusters of computers. Every individual PC in a little cluster has somewhere between one and four processors, and today‘s processors normally have between two to four cores. HPC individuals regularly refer to the individual computers in a cluster as nodes. A cluster of enthusiasm to a little business could have presently four nodes, or 16 centers. A typical cluster estimate in numerous organizations is somewhere around 16 and 64 hubs, or from 64 to 256 cores.

The purpose of having an HPC is so that the individual nodes can cooperate to take care of an issue bigger than any one computer can undoubtedly solve. Furthermore, much the same as individuals, the nodes should have the capacity to converse with each other to work together. Obviously computers converse with one another over systems, and there is an assortment of PC system (or interconnection) alternatives accessible for a business cluster.

4.1.2.1 Software that Makes the Cluster Work

Much the same as our desktop or tablet, the High Performance Computing (HPC) group will not keep running without programming. Two of the most famous operating system choices in HPC are Linux (in all the different varieties) and Windows. Linux presently rules HPC establishments, yet this partially because of HPC's legacy in supercomputing, vast scale machines, and UNIX. Deciding which framework to use should depend on what the HPC will be expected to accomplish. The parallel architectures of supercomputers regularly direct the utilization of exceptional programming methods to exploit their speed. Programming tools for distributed processing incorporate standard APIs, for example, Message Passing Interface and Parallel Virtual Machine. Based on this High-Performance Computing observation, the CRanker algorithm has to be re-implemented using parallel programming techniques. Doing this could disturb the actual behavior of CRanker and may result in inaccuracy of the final output. The higher costs involved in using High-Performance computing centers also ruled out this option.

4.1.3 Observations Made on Using GPU Computing

GPU-accelerated computing [Owens, Houston, Luebke, Green, Stone, and Phillips, 2008] is the utilization of a graphics processing unit (GPU) together with a CPU to quicken scientific, analytics, designing, consumer, and enterprise applications. The hardware architecture of the GPU is designed to eliminate the von Neumann bottleneck by devoting more transistors to data processing. Spearheaded in 2007 by NVIDIA, GPU accelerators now power effective datacenters in government labs, colleges, enterprises, and small-and-medium organizations around the globe. GPU-accelerated computing offers phenomenal

application execution by offloading computer escalated segments of the application to the GPU while the rest of the code still keeps running on the CPU. From a client's point of view, applications essentially run faster. The Figure 4.1 (source from NVIDIA) shows how it works.

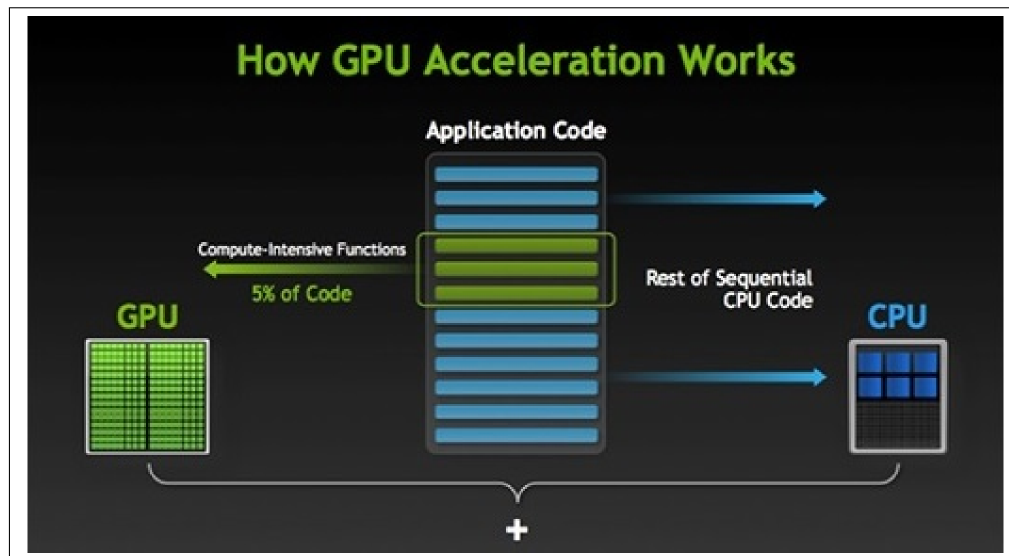


Figure 4.1: GPU Processing

4.1.3.1 CPU vs. GPU Processing

An essential approach to comprehending the contrast between a CPU and GPU is to think about how they process tasks. A CPU comprises of a couple cores improved for successive serial processing while a GPU has a greatly parallel architecture comprising of a vast number of smaller, more efficient cores intended for taking care of different assignments at the same time. Each stream processor has an individual memory interface. Memory access latency can be further covered up by calculations. The same program can, along these lines, execute on numerous data elements in parallel, obstructed by a single memory interface. The GPU is particularly suited for issues that can be expressed as data parallel computations in which the same project is executed on numerous data elements in

parallel with a high proportion of arithmetic operations to common memory operations. On account of the parallel execution of numerous data elements, there is a low necessity for flow control. Algorithms that procure substantial data elements, which can be dealt with in parallel, can be accelerated. Algorithms that can‘t be communicated in a data-parallel manner, particularly those that depend on refined flow control, are not useful for GPU processing.

4.1.4 What About Re-Implementing the CRanker Algorithm

An idea came to re-implement the CRanker in such a way by including the parallelism concepts but the actual workflow of CRanker is complex and the implementation is again a new research area. This thesis didn‘t focus much on re-implementing the CRanker with parallelism concepts.

As CRanker has the data-driven work flow, i.e., CRanker execution is data-driven. The input file containing the peptide data is parsed by CRanker which will perform actions defined in it. Figure 4.2 shows the flow of data.

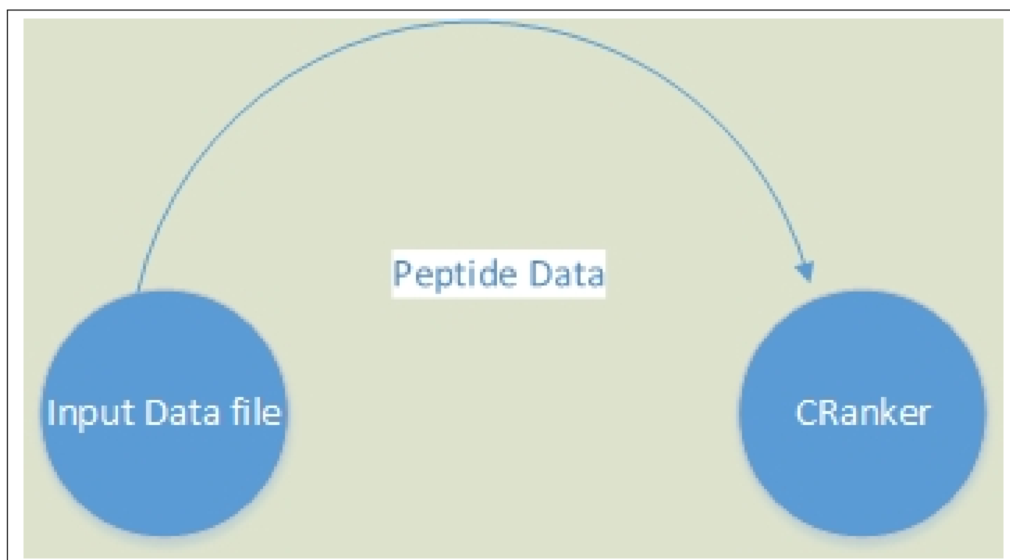


Figure 4.2: CRanker DataFlow

.

The noticeable thing in the CRanker input files is that they are available in ".txt" or ".xls" format. This gives us the confidence that the data available in the rows are not dependent on each other. CRanker can process each row in the input file individually by dividing the input file into small parts. Dividing the input file will not be a problem for CRanker and it can process that data easily. This was tested against the data that split against the rows of each split individual row that was processed by CRanker. The process is displayed in the Figure 4.3

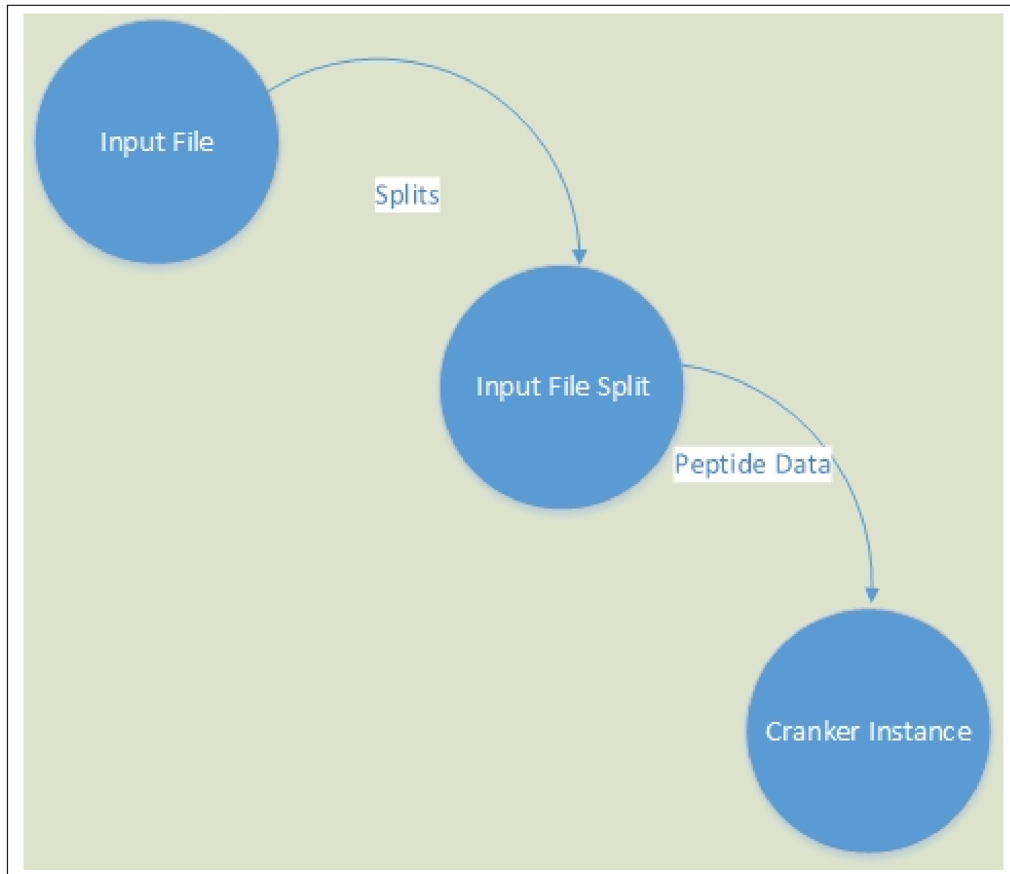


Figure 4.3: CRanker Input Split DataFlow

.

4.2 Parallelizing the CRanker Application

Section 4.2 discusses ideas in parallelizing the CRanker application setting up the required environment:

4.2.1 Finalized Idea for CRanker Execution and Approved Concept

As per our observations, CRanker can process each row in a given input file without depending on the consequent rows, this exactly fits with the Hadoop execution framework. As in a Hadoop cluster, each node processes the data located on its local storage on HDFS. The data that is processed at the local node is independent of the data that is stored at the other node. The CRanker instances that are installed at each of the Hadoop nodes will consume and process the peptide data. During this work, all the nodes with CRanker instances will execute the intermediate steps involved in the data processing at all the local nodes. The detailed explanation will be available in the section for implementation and execution.

4.3 Setting of Environment and Other Essentials

Section 4.3 discusses the methods for setting up the required hardware and software to implement and execute this thesis.

A total of three nodes are used to set up the Apache Hadoop cluster. The three nodes run with the Linux Ubuntu **Ubuntu 14.04.2 LTS** (Trusty Tahr) version operating system which does not have the GUI. Interactions with the operating system are made through the terminal only. The main reason for choosing a Linux operating system is because of Apache Hadoop, which only comes with the Linux distribution, and Ubuntu is the friendliest version of the Linux OS.

4.3.1 Setting Up the Hardware Infrastructure

As Hadoop is intended to execute on a cluster of nodes, there is a need to configure the cluster for Apache Hadoop and execute CRanker. For this thesis, numerous approaches have been identified and finally Amazon Elastic Cloud Computing (EC2) [Amazon, 2010] was chosen as a Infrastructure as a Service (IaaS). The main reason for choosing Amazon EC2 services are: they are reliable, easy to maintain, efficient, and cost effective. They can also be hired on a per hour basis. Utilizing Amazon EC2 eliminates the user need to put resources into the equipment in advance, so with it, it is easy to create and deploy applications quicker. The user can utilize Amazon EC2 to dispatch the same number or a couple of virtual servers as the user needs, to arrange security and organization as well as manage storage. Amazon Elastic Cloud Compute (EC2) empowers the user to scale up or down to handle changes in prerequisites or spikes in popularity, lessening user needs to estimate activity.

Some Features of Amazon EC2

- Virtual computing environments, known as instances.
- Preconfigured templates for user instances, known as Amazon Machine Images (AMIs), that package the needed for the user server (including the operating system and additional software).
- Various configurations of CPU, memory, storage, and networking capacity for different instances, known as instance types.

The Figure 4.4 (Sourced from Amazon Web Services user guide) shows about connecting the user to Amazon EC2 and its components.

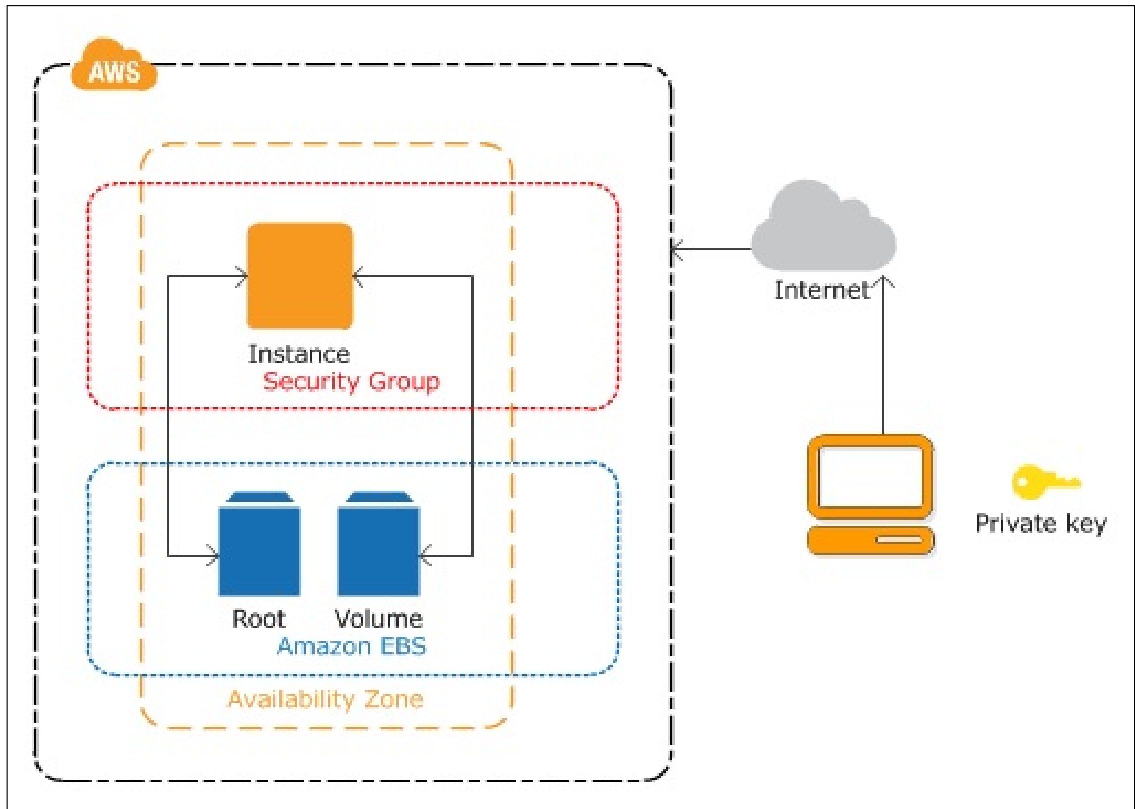


Figure 4.4: Amazon EC2 instance access

The more instructions for configuring the Amazon EC2 instances are available at:

`docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html`

In this thesis, there are three nodes used for the execution of CRanker. One is used as the Hadoop master node (Resource Manager) and the other two are used as slave nodes (DataNodes).

4.3.2 Setting Up the Software Infrastructure

Section 4.3.2 briefly discusses the process of setting up the CRanker and its required software components and installing Apache Hadoop in the fully distributed mode.

4.3.2.1 Installing CRanker and Its Components

The steps involved in installing and setting up of CRanker are discussed in detail in the user manual provided, along with the CRanker installation package [Xia, 2013].

4.3.2.2 MapReduce Next Generation - Cluster Setup

The details of installing and setting up the Hadoop cluster is contained in the following reference [Hadoop, 2015].

Now that the hardware and software setup is completed, the design and implementation is discussed in the section 4.4.

4.4 Proposed Framework Architecture, Design, and Implementation

Section 4.4 describes the proposed idea, execution, design, architecture and implementation.

4.4.1 Idea Execution and Design

The main goal of the thesis is to design and develop a program to make CRanker execution faster to reduce the processing time of PSM data sets. To achieve this objective Apache Hadoop MapReduce, a distributed data-processing model, and Hadoop Distributed File System (HDFS) have been used.

Many bioinformatics algorithms are parallelizable, but parallelism is not readily available in their original code. The user who needs to make the algorithm parallelizable may have to rewrite the complete code.

The other options which the user may consider:

1. Create or adjust existing software to distribute and oversee parallel jobs.

2. Change existing applications to make utilization of libraries that encourage distributed programming, for example, RPC and RMI.

When changing applications, the exertion is intermittent since new versions of the first sequential code may render the parallelized application obsolete. The measure of work included may prompt parallel versions that slack and need components of the most recent sequential tool version. Regularly, altered applications will exclude mechanisms to handle failures naturally.

It is always good to go with the first option as it is too expensive to modify the existing code. The source code may not be available, or the user may not have complete knowledge of the algorithm. This is good when it is relatively easy to parallelize applications that can run on multiple nodes, each of which executes the sequential application of a subset of the given input. However, this requires additional software or framework to manage job distribution and fault tolerance. The work advocates the use of the MapReduce framework which combines many features and lessons learned from the distributed computing.

To better depict the parallelization process, it is useful to consider a particular algorithm that is going through a process called CRanker. CRanker is used to identify the correct PSM based on the post database searching. Given a set of the input data file within PSM records, CRanker validates each PSM record individually. The usual approach to ex-

ecuting CRanker in a distributed environment is to divide the input PSM data set so that each separate instance of CRanker can process the divided input PSM data set.

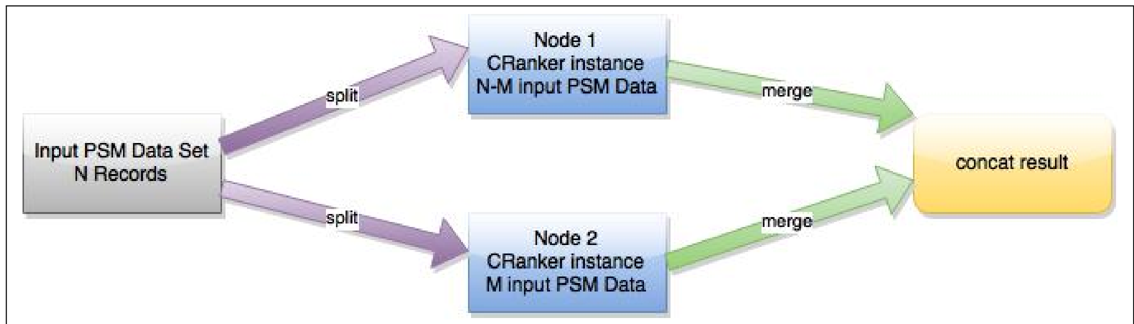


Figure 4.5: CRanker Input File Split

Dividing the input PSM data set can be easily implemented using a scripting language to identify the input PSM data sets, execute CRanker, and concatenate the result. For example, a bash script would simply use csplit, ssh/rsh, and cat commands. In any case, clients still face the accompanying difficulties:

- Discovering the perfect number of worker nodes to utilize (or that are accessible) and split appropriately.
- Load balancing and giving balanced partitions, subsequent to the time of CRanker execution is profoundly reliant on the size of a PSM data set.
- Recouping from the potential failures of a few working nodes with a specific end goal to abstain from acquiring just incomplete results.

To handle the above-specified issues more efficiently, the use of MapReduce is a good approach. MapReduce as a programming framework for distributed processing with adaptation to internal failure was proposed by Google, propelled by functional languages, to acknowledge parallel computing on countless resources. The software engineer

is obliged to execute a map and a reduce function. MapReduce framework handles automatically dividing the input PSM data set and distributing each chunk of PSM data set to worker nodes (Mappers) on multiple machines. The output of each mapper is grouped and sorted by an intermediate key, passing these values to working nodes (Reducers) on multiple resources. The execution of mappers and reducers are monitored as to re-execute them when failures are detected in any of those working nodes. These features greatly simplify deployment and management of jobs, as jobs are submitted and monitored from a centralized location.

The core part is to run the CRanker program in the MapReduce framework to assign each mapper the execution of the CRanker operation for the subset of PSM data sets thus eliminating the need of the reducer. However, the reducer may be required in the other bioinformatics applications to arrange the output in the required manner. In this work, Apache Hadoop, an open-source implementation of the MapReduce framework and HDFS, was used to parallelize the execution of CRanker. The parallelization approach consists of dividing the input PSM data set to store on HDFS and running multiple instances of an unmodified CRanker version on each divided PSM data subset. Hadoop offers streaming that allows easy execution of such third party applications. HDFS splits the input PSM data set based on the block size established by the Hadoop user (default 64 MB). To combine all the results that were generated by CRanker instances, the merge command of Hadoop Distributed File System was used.

4.4.2 Architecture

Section 4.4.2 describes the architectural approach that was followed in this thesis.

The distributed approach architecture was carefully designed by keeping the Apache

Hadoop architecture and Hadoop execution workflow in mind. Apache Hadoop architecture consists of Hadoop Distributed File System as the bottom layer that is the self-healing clustered storage system above HDFS. MapReduce is the distributed data processing framework that can process the data that is stored in the HDFS. By using this advantage the input PSM dataset can be stored in the HDFS and be dealt with by the MapReduce. The basic functionality in MapReduce version 2 is to divide JobTracker, Resource Management, and administration into separate daemons. The thought is to have a global ResourceManager (RM) and per-application ApplicationMaster (AM). An application is either a single job in the established sense of MapReduce jobs or a group of jobs.

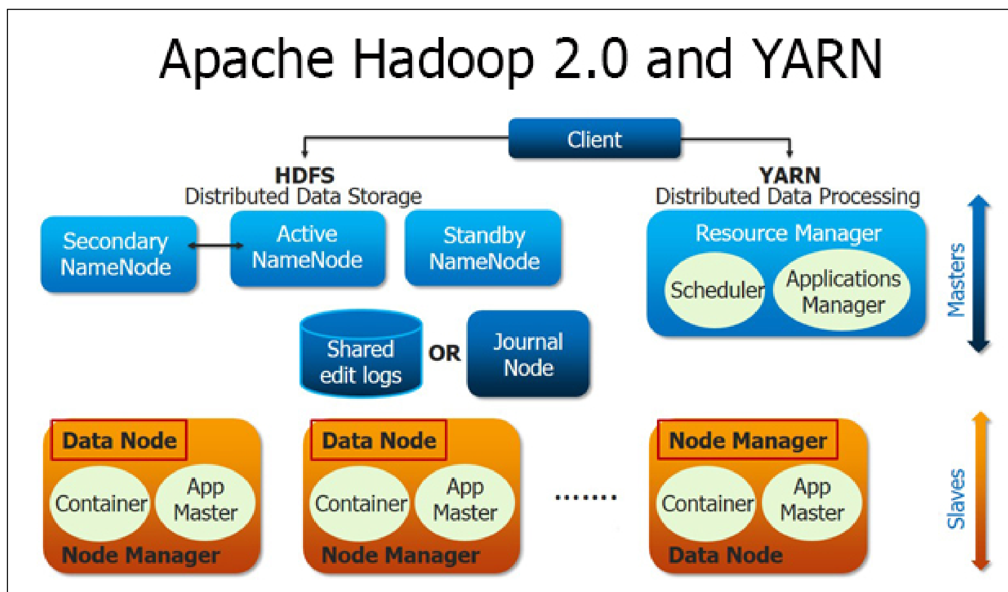


Figure 4.6: Hadoop Resource Management Architecture

The ResourceManager has two important components: Scheduler and ApplicationsManager. The Scheduler is in charge of assigning resources to the different running applications subject to recognizable imperatives of capacities, queues, and so forth. The ApplicationsManager handles tolerating job-entries, arranging the first resource for exe-

cuting the application particular to ApplicationMaster and gives the support of restarting the ApplicationMaster container on failure. The NodeManager is the per-machine configuration agent that handles containers, monitoring their asset utilization (CPU, memory, disk, network), and reporting the same to the ResourceManager/Scheduler. The per-node ApplicationMaster has the obligation of arranging proper asset containers from the Scheduler following their status and observing for advancement.

By carefully considering the Hadoop Resource Management Architecture, the proposed framework is designed, the architecture has taken the resource management leverage from Apache Hadoop and the input PSM data set is stored in the HDFS. The PSM subsets will automatically be divided and replicated in the HDFS and stored all over the nodes in the Apache Hadoop cluster.

As shown in the Figure 4.7 the input PSM data set is determined and store in the Hadoop Distributed File System. Input PSM data set split is based on HDFS block size and replication factor values that are set during the Apache Hadoop cluster installation. The designed architecture contains the JSON properties that is used to convert the data set values into objects and inject them into the CRanker instance while execution. The usage of JSON object files avoids the problem that was caused by PSM data set split.

In CRanker execution, the first step is to read the PSM data set and identify the attribute names available in the starting row. The order of the attributes in data representation does not matter, but the names of the attributes must be correct. When the input PSM data set gets split into chunks, the first row (attributes of PSM data) in the data set is only available with one of the subsets and all other subsets will miss those attribute names. In this case, each CRanker instance (Mapper) that executes the missing attribute input chunk

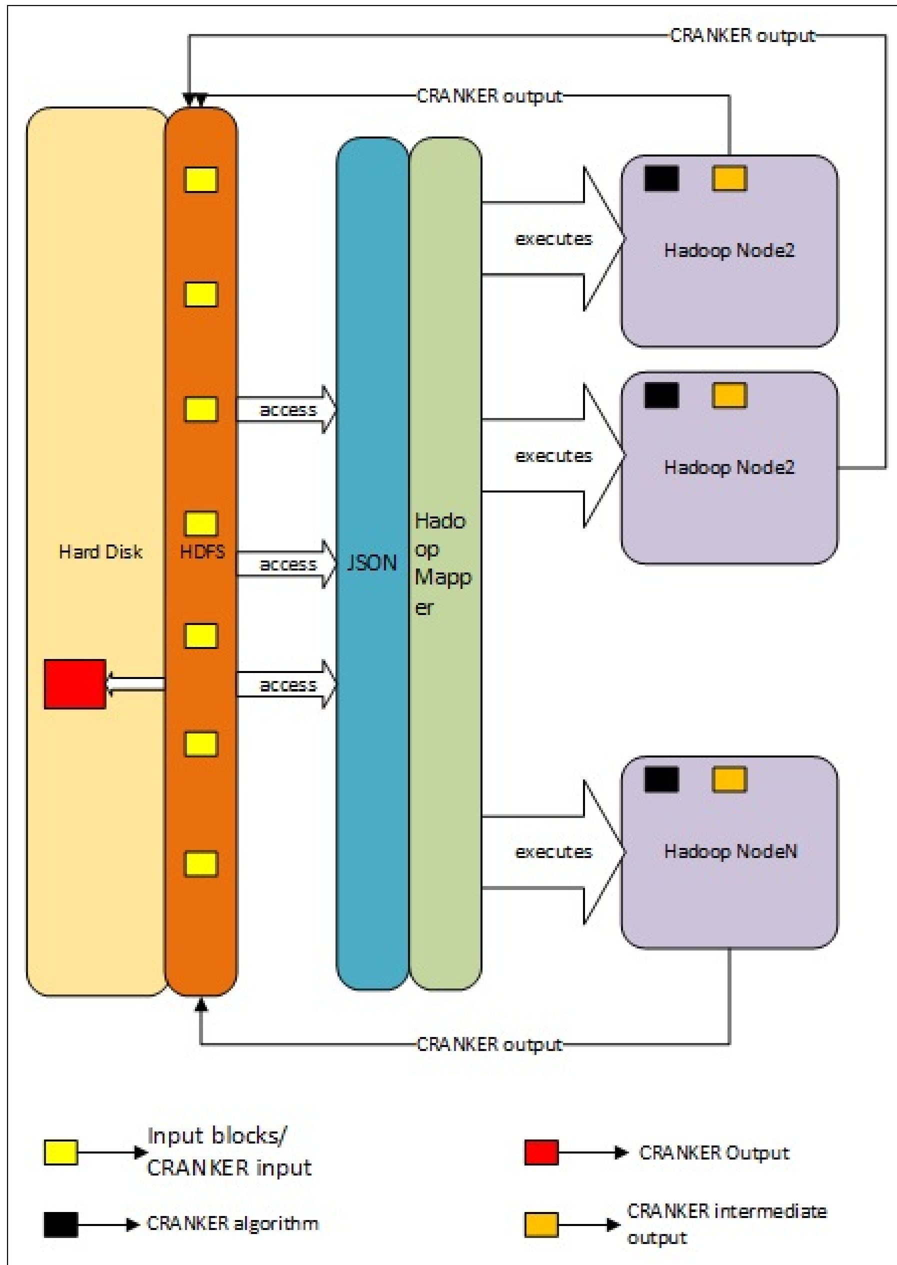


Figure 4.7: Proposed Framework Architecture

treats the data subset as an individual set. It then fails to identify the attributes that caused the failure of the MapReduce job (CRanker execution). This architecture is designed in such a way as to overcome this problem using JSON objects to transmit data consisting of attribute–s value pairs to all the mappers running on the nodes in the Apache Hadoop

cluster. The proposed application accepts the input PSM data set without PSM attributes and those attributes must passed in to the application as an user input parameters during the execution of CRanker distributed execution command. For instance refer the CRanker execution command on Apache Hadoop provided in appendix A.B.1.

4.4.3 Implementation

Section 4.4.3 describes the implementation of the proposed framework.

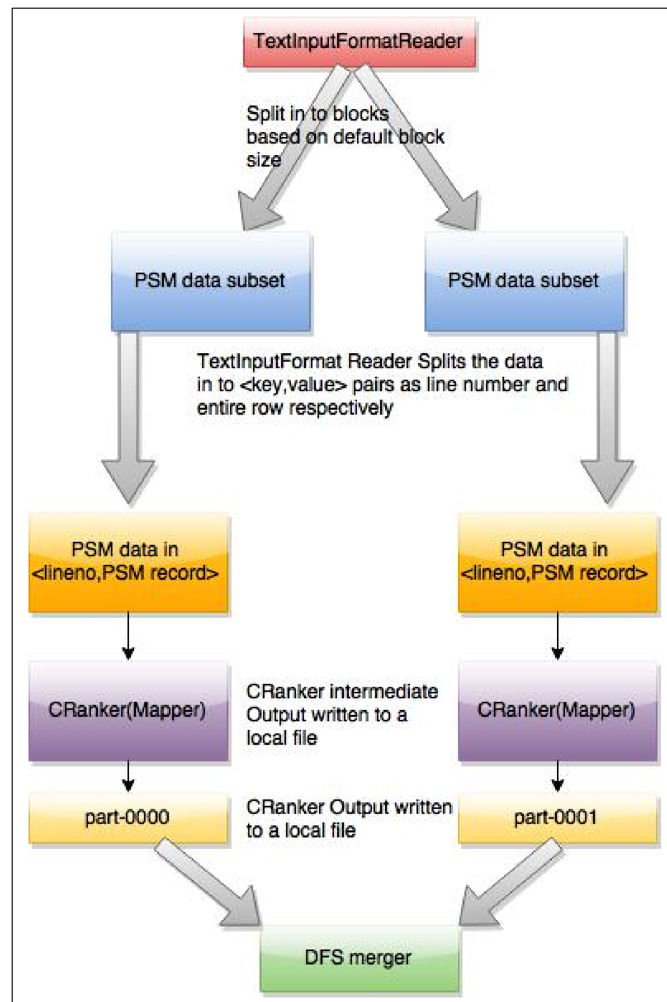


Figure 4.8: CRanker with MapReduce Dissected

The input PSM data file will split into blocks of defined size in hdfs-site.xml while storing it on the Hadoop Distributed File System. The framework will trigger the CRanker

execution (Mapper) where each data block is stored and process the data. The mapper will then try to process the data which is available locally. Each mapper labeled as a MapReduce job will contain all the CRanker execution tasks. The MapReduce job has the following steps, and all the steps run sequentially:

1. "Read task" reads the PSM data subset and writes the data to ".mat" file that is stored on the HDFS staging location at the local node.

2. "Solve task" reads the ".mat" file generated in the above step, calculates the score of each PSM record and stores the data in the temp file on HDFS staging at the same local node.

3. "Write task" reads the two temp files that are generated during steps one and two, then writes out the identified reliable PSMs into HDFS.

Step three is the final step to execute inside Mapper. If all of the above steps were successfully executed the job is considered as a pass otherwise, it fails. The job tracker will monitor the execution of jobs and the jobs with failed status will get triggered by the job tracker. HDFS replicates the data on multiple nodes. In case of node failures, the resource manager will update the list of all available nodes through heartbeat. Using the replication mechanism, the data in the failure node will be available on the other node and the mapper will trigger the job to execute that data. For instance, Hadoop 2 has a better way to handle failures in the NameNode. The user has a chance to run two multiple NameNodes alongside one another, so that in the case of a NameNodes failure, the cluster will quickly switch over to the other NameNode. The way it works is straightforward. Essentially, the DataNodes

will send messages to both NameNodes, which is called heartbeat, so that if one falls flat the other one will be prepared to work as part of the dynamic mode. What‘s more, for the client, it just contacts each NameNode designed until it finds the dynamic one. So in the event that it gets an answer saying to attempt somewhere else, or if the NameNode does not answer, it realizes that it needs to utilize an alternate NameNode.

Chapter 5

DEVELOPMENT AND EXECUTION

Chapter 5 will discuss the details about setting up the resources needed for development, procedures in development and execution.

The first step is to find the resources that can help in development and execution of the application. Chapter 5 mostly focuses on development and implementation by introducing the resources that were used during this process.

5.1 Setting up of Resources

Amazon Infrastructure as Service (IaaS) is used to setup the hardware resources. In this process, two types of Amazon instances are used.

Initially, an Amazon instance of Type1 is launched. Software components like Apache Hadoop and CRanker components are installed as specified in Chapter 4. After the software setup is completed, the Amazon Machine Image (AMI) of Type1 is created and is then used to launch the Type2 instances. AMI is used to launch as many cases as are required in the future, which reduces the software setup time. AMI provides the user

	Type1	Type2
CPU	Intel Xeon	Intel Xeon
Cores	1	2
Memory (GiB)	1	4
Cache(MB)	2	2
OS	Ubuntu 14.10	Ubuntu 14.10

Table 5.1: Hardware used for development and execution

with the flexibility to initiate the instance of a required hardware configuration. Developed applications can be encapsulated into Amazon instances and, since AMI are used as templates to deploy multiple copies when required, only the template AMIs need maintenance. When application updates are necessary, a new template AMI is created, and the created copies are distributed by the admin for the users. For this thesis, two test beds (clusters) were used. They are as follows:

1. **Test bed 1**: Resource Manager and two data nodes of type1 are used.

2. **Test bed 2**: Resource Manager of type1 and two data nodes of type2 are used.

5.2 Determining the Approach for CRanker Execution

Before describing the algorithm for the Distributed approach CRanker execution, it is good to know the steps involved. The CRanker first reads the PSM data set that is a text file and creates the ".mat" file then solves the .mat file by identifying the scores for each PSM record and creates scores ".mat" file. It then uses both ".mat" and scores ".mat" to complete the final step of writing the correct peptides information into the local file system. This execution sequence should be implemented in the Apache Hadoop using MapReduce and HDFS for processing and storage respectively.

Making an algorithm for a distributed approach does not change the execution behavior of CRanker. The CRanker will continue its normal execution and will identify the correct PSMs based on the input PSM data provided to it. A new algorithm is then proposed to validate the generated output. The proposed algorithm will compare the CRanker output generated by distributed environment with the output generated by actual CRanker execution. The file comparison algorithm details will be discussed in the coming sections.

5.2.1 Outline of the Algorithm

In the proposed distributed algorithm, the JSON properties file will be created. The file contains the staging directory location of CRanker, MCR root, and a cache location. It will also contain the name of the algorithm to execute and the commands involved in executing it. Along with those, it also contains input file and output file HDFS locations. The file will be the sole configuration file for the entire application and any changes to the file will impact the execution behavior of the framework. Once the configuration object is prepared based on the values in JSON properties that object will be loaded into memory. Then the file writer will open the files to write the staging values on to the local file system. An option is available to check whether the header values (PSM data attributes) are required at each stage. Completing this step will setup all the preliminary necessities to execute CRanker, and all these steps will be enclosed in the method. Once the setup is complete, the map implementation will write each record of the input split into intermediate staging files on the local file system. Each mapper will have its own such files, and the file split becomes the input for each CRanker instance that sits on the Apache Hadoop nodes. Finally, the mapper starts closing all the file writers that were open during the setup. The environment variables and the commands that are required to execute CRanker are readily available and are already prepared in the setup. Now the shell script commands of CRanker start execution as per the given order in the configuration file. The process will repeat at all the nodes in the Apache Hadoop cluster. The code for MapReduce algorithm can be found appendix A.B for more details.

5.2.2 Execution of the Algorithm

The approach is designed in such a way that HDFS will split the input PSM data set based on the specified block size. The PSM data subsets will then be distributed and stored across all the DataNodes in the Apache Hadoop cluster. The mappers will try to consume the data available at their local node and trigger the execution of a CRanker instance that is already installed on that node. The CRanker processes the input PSM dataset and identifies the correct Peptides.

When the PSM dataset is loaded into the HDFS it is divided into blocks based on the specified block. HDFS will replicate the blocks at different nodes in the Apache Hadoop cluster based on the replication factor that is defined by the Apache Hadoop administrator. Once the task of blocks and replication is complete, it is now MapReduce‘s job to read, process, and write the data in to HDFS again. The CRanker distributed execution input PSM data set is loaded into HDFS using dfs commands. The loaded input PSM dataset is split into blocks and stored in multiple nodes in the Apache Hadoop cluster.

The MapReduce job configuration is the main interface for a user to specify the MapReduce job for Apache Hadoop execution. The job contains certain parameters that can define the execution flow. Some of them may include Mappers, Reducers, InputFormatter, OutputFormatter, execution of Mappers and Reducers, and maximum attempts to execute Mappers and Reducers. Job represents the CRanker execution job that is created as part of the framework. In CRanker execution MapReduce job submission and execution steps are as follows:

1. Checking the input data and output data specifications of the job.

2. Computing the input PSM data set split values for the job.

3. Setting up the required information for the DistributedCache of the job.

4. Submitting the CRanker execution job to the resource manager and monitors the status of the execution job.

InputFormat describes the input data specification of the job and in this context the "FileInputReader" is the input reader for the Mapper. The input file that is the PSM data subset will split into logical InputSplit instances. These logical instances will then be assigned to the Mapper for execution. Mappers will write input splits to a staging area on the local file system. Once the InputSplit is copied to the local file system from HDFS, these staged input files will then be given to CRanker algorithm for execution. Typically, InputSplit presents a byte-oriented view of the input. Apache Hadoop comes with the configured single mandatory queue called default. All the MapReduce jobs are scheduled in the default queue for execution but before scheduling to the default the DistributedCache setup will be done. This facility is provided by the MapReduce framework to cache the files that are needed by applications. Now the MapReduce job that contains the execution of the CRanker algorithm will be submitted to the ResourceManager (RM). It is the RM‘s responsibility to track and monitor the job. The failed jobs try to get executed again based on the maximum attempts configured in the job. The submitted job contains the shell scripts to execute all the steps of the CRanker. The input PSM dataset for CRanker is the input split instance in this context, and CRanker reads the input split and the intermediate files during the process. They are then stored in the staging directory of the local file system. Once the steps in the CRanker execution complete the final output, it is moved into HDFS. Now,

"OutputFormat" describes the output specification of the Mapreduce job. There is a check to whether the specified output directory already exists, and it will clean up all the jobs after execution. For example, it removes temporary output directories. The steps of execution specified in the framework occur at each of the Mappers, and those Mappers consume all the blocks of PSM datasets that are stored on the nodes of the Apache Hadoop cluster.The CRanker execution result is stored in HDFS in the form of blocks, and these blocks are distributed across all the nodes in the Apache Hadoop cluster. To combine all results, merge command of HDFS is used, and the merged file can be downloaded to the local file system. In this process, the reducer is not used as all the tasks of the CRanker are designed to be performed in Mapper. This entire approach reduces the total time taken to the complete the processing of large PSM data sets and improves the efficiency in processing. This entire process is backed by the Apache Hadoop; CRanker will benefit from the advantages of the Apache Hadoop

5.3 Determining the Joins for File Comparison

The output generated by the mappers will get merged into the distributed file system command and then the generated output will need to be compared to the result produced by the normal CRanker execution. The comparison can be done using normal java solution, but Apache Hadoop also provides an approach to accomplish this. The distributed solution for this is crucial to processing the large datasets, and this approach requires a join operation with MapReduce. An algorithm was designed to perform the comparison with joins. In this algorithm, the join is implemented on the reducer side as the map task will only pre-process the tuples of datasets to organize them into the key value pairs. The map function reads one tuple at a time from both of the datasets via an input stream from HDFS. The values

from the column specified by the user on which the join is being done are fetched as keys to the map function. The rest of the tuple is fetched as the value associated with that key. The code for the algorithm implementation can be referred in appendix B.A

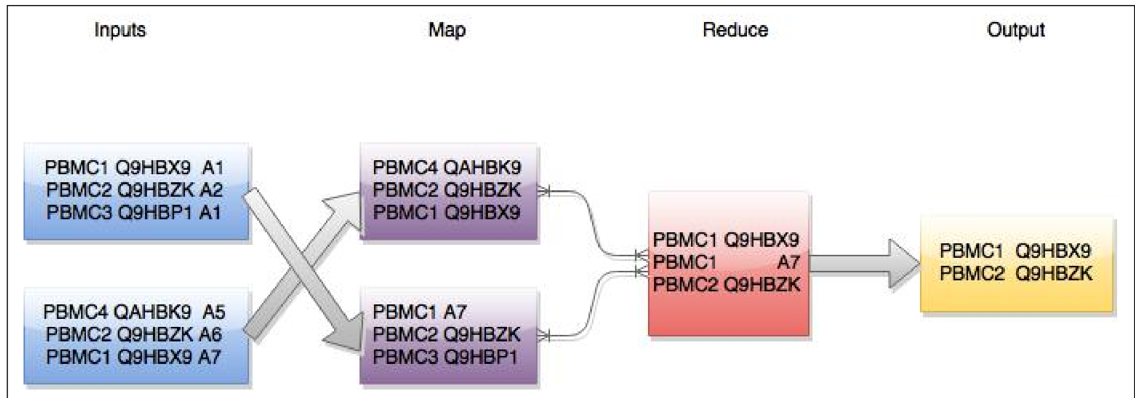


Figure 5.1: MapReduce Join

5.3.1 Outline of MapReduce Join Algorithm

Based on the idea of MapReduce joins, two different Mappers are created to read the two input identified PSM datasets. Each mapper will generate the key value pairs based on the user choice, i.e., the user will provide the column number for the key in both of the input files, and the rest will be chosen as values. The input file reader will be "TextInputFormat" for the two mappers and the output of them is the input of the reducer. However, the order in which values arrive at the reducer is unspecified. Sorting in Apache Hadoop is performed on a key-by-key basis, and all keys for a particular user are identical. For the reducer to join the two data sets together it must read all values in the memory, find the one containing the given user value, and then emit the remaining values along with it. The reducer will identify the source of the value based on the prefix file tag added. On the reducer, every key would have two values based on the given two file tags. (For simplicity let's assume only two values, in real time it can be more). Identify the records and from fileTag1, get

the PSM corresponding to the given input key and from fileTag2, get the corresponding values. After obtaining the values, increment the file counter and match counter. So finally the output Key values from the reducer would be as follows:

- Key: column chosen by user in PSM data set.
- Value: rest of the values with the key.

Along with the matched key value pairs, the algorithm will also calculate the total percentage matched based on the file counter and match counter in respect to the two input files.

Chapter 6

EVALUATIONS AND ANALYSIS

Chapter 6 covers the data collected during the CRanker normal execution and the CRanker distributed execution. It also includes file comparison results between them. The results are collected under three test cluster conditions. The hardware configurations in those clusters are based on Type1 and Type2 as specified in Chapter 5. The types of testbeds used for the evaluations in this thesis are specified in the Table 6.1

The test results are compared with normal CRanker execution against CRanker using Apache Hadoop single node setup, setup using "Cluster 1", and setup using "Cluster 2" clusters. The PBMC datasets used are: "pbmc_orbit_ mips.txt", "pbmc_ orbit_nomips.txt", "pbmc_ velos_ mips.txt", and "pbmc_ velos_ nomips.txt". Since Amazon EC2 has been used as a Infrastructure as a Service (IaaS), the burden caused by the virtualization should also be considered in line with the burden already being caused by Apache Hadoop during execution.

Initially, CRanker localhost execution results are compared against the CRanker on Apache Hadoop localhost installation and CRanker on Apache Hadoop Amazon EC2

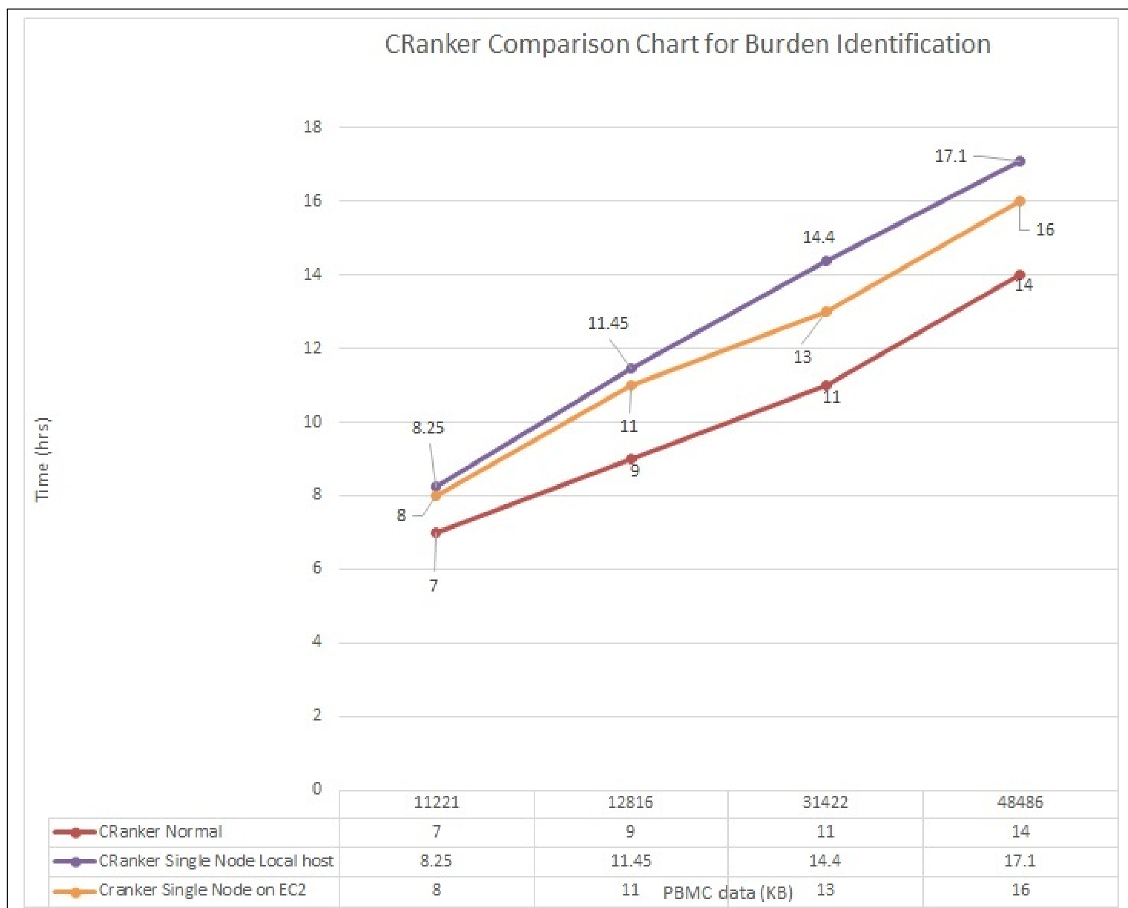
	DN(Type)	NN(Type)	RM(Type)	Total Nodes(s)
Cluster 1	2 (Type 1)	1 (Type 1)	1 (Type 1)	3
Cluster 2	2 (Type 2)	1 (Type 1)	1 (Type 1)	3

Table 6.1: Test Beds Used for CRanker Distributed Execution Using Amazon EC2

	localhost
CPU	Intel i5
Memory (GiB)	4
Cache (MB)	2
OS	Ubuntu 14.04

Table 6.2: Test Bed for Apache Hadoop CRanker Execution on Localhost

to check the burden caused by Apache Hadoop and virtualization. There is a very minimal effect on the burden caused by virtualization. However, running CRanker on Apache Hadoop using single node setup has a huge effect on the execution time as it is significantly increased. The reason for this is because Apache Hadoop and CRanker share the same resources on a single computer. The figured graph 6.1 shows the differences:



	11221	12816	31422	48486
CRanker Normal	7	9	11	14
CRanker Single Node Local host	8.25	11.45	14.4	17.1
Cranker Single Node on EC2	8	11	13	16

Figure 6.1: CRanker Execution Burden Comparison

The execution time for CRanker on a single node that is setup on the localhost has more issues. The problem here is that the resources are being shared by CRanker, Apache Hadoop, and other Operating system components. Next, the CRanker on a single node that is setup on Amazon EC2 has almost the same execution time. There is a slight advantage due to the non-GUI version of the Operating system dedicated to run only CRanker. Clearly, this indicates that the Apache Hadoop burden is worse on CRanker while executing. Based on these results, Apache Hadoop execution is used to process large datasets on huge clusters only after careful consideration.

6.1 Evaluating the Distributed Execution of CRanker Algorithm Using Apache Hadoop Approach

Section 6.1 describes memory utilization and the execution time of CRanker using various test setups.

6.1.1 Evaluating the Memory Considerations and its Analysis

Section 6.1.1 evaluates the memory consumption while executing the CRanker in different test environments. When the “pbmc_ velos_ nomips.txt”dataset is used as the input for CRanker on a single node (Type1) that is setup on EC2, the jobs failed multiple times. Apache Hadoop tried to develop a reason and found it is due to a lack of enough memory for the job execution. This instance triggered the need to evaluate the memory consumption of MapReduce jobs, i.e., the execution of CRanker on Apache Hadoop clusters. The CRanker memory consumption is assessed against the normal CRanker execution on a single node on the localhost. The results are displayed on the bar chart that is shown in the figured graph 6.2.

These results in the figured graph 6.2 portrayed that the CRanker execution on

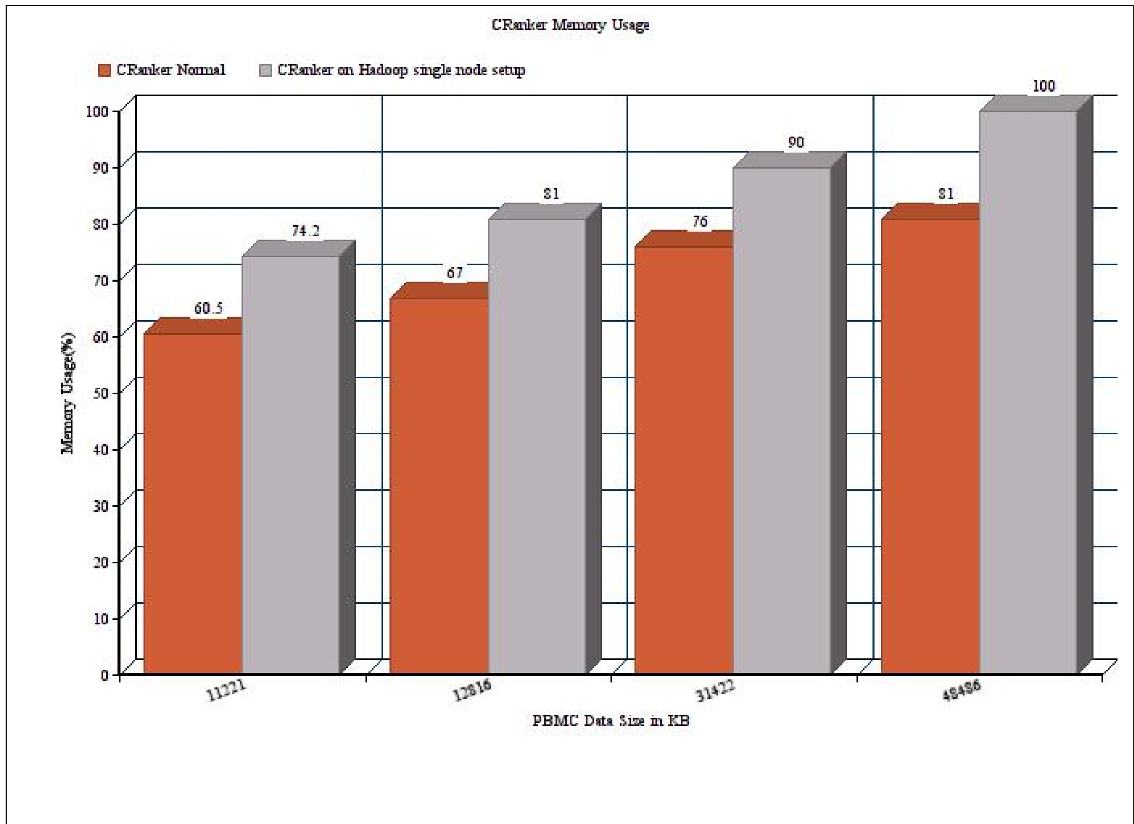


Figure 6.2: CRanker Normal Memory Usage Vs Apache Hadoop Jobs Memory Consumption on Localhost

Apache Hadoop consumed more memory than the CRanker normal execution. The MapReduce job is the first one to load into the memory and then the CRanker initiates, which further consumes memory. Then, initiation of the CRanker instance consumes memory even further due to its execution. All the MapReduce jobs will execute on a single machine, so these results provide us with the insight to distribute the MapReduce jobs and setup an Apache Hadoop cluster to process them.

After setting up the cluster, the experiment was conducted again. However, this time the execution setup was with a Cluster 1. After the execution of the CRanker job on the Cluster 1, the results were compared with the normal CRanker execution. An inter-

esting result was recorded as Cluster memory consumption is higher compared to normal execution flow. The results of this experiment are shown in the figured graph 6.3.

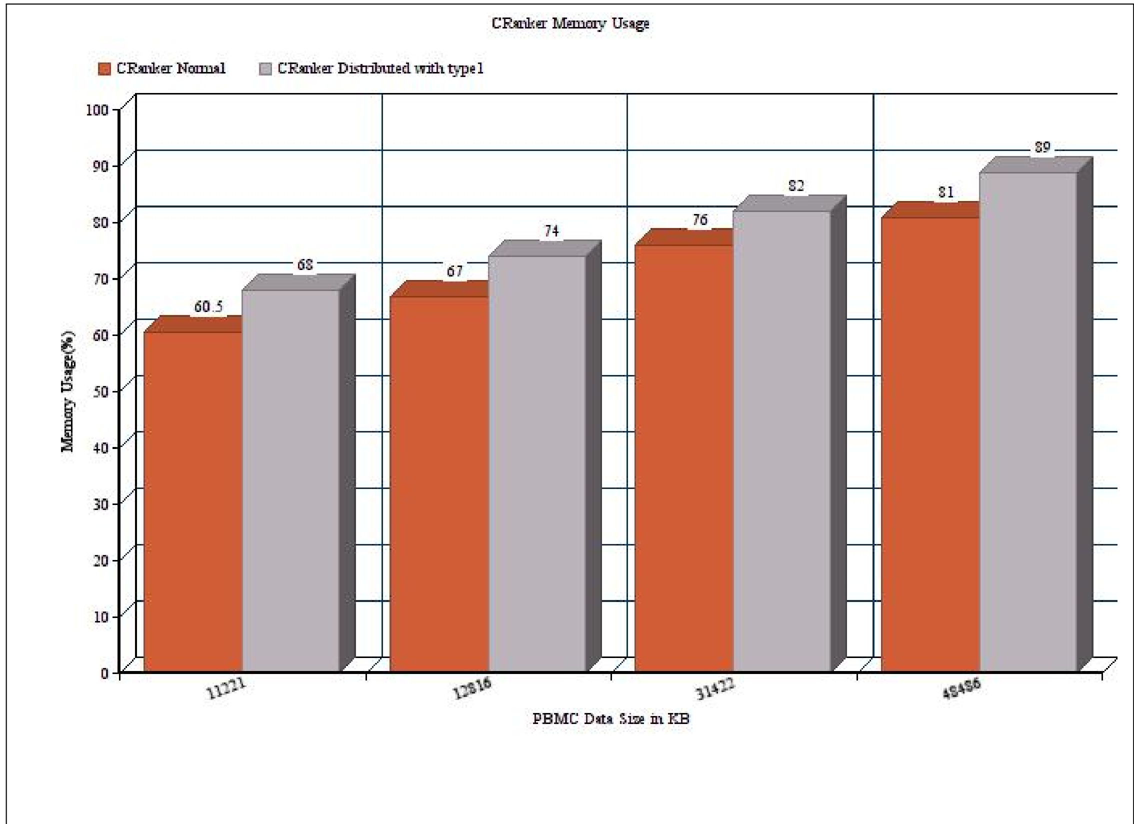


Figure 6.3: CRanker Normal Memory Usage Vs Apache Hadoop Jobs Memory Consumption on Cluster 1

Memory usage is higher in a Cluster 1 compared to the normal CRanker execution. When the Apache Hadoop resource manager starts dumping the jobs, these jobs may frequently fail due to a lack of memory. The Apache Hadoop speculative algorithm will keep trying to execute the failed jobs which keeps the memory occupied, and will surge the memory utilization on the data nodes. In general, this is due to a lesser number of nodes because if more nodes are available, the MapReduce jobs will get distributed across those available nodes. Obviously, this will reduce the memory usage of the data node. To overcome this problem, a better cluster was created (Cluster 2) with more powerful machines.

A more powerful machine was necessary because this research does not have the resources to add more nodes to the cluster. When the CRanker distributed execution is triggered on the Cluster 2 for all the input data sets, the results are recorded and displayed in a bar graph against the normal CRanker execution. The figured graph 6.4 shows the significant reduction in the memory usage:

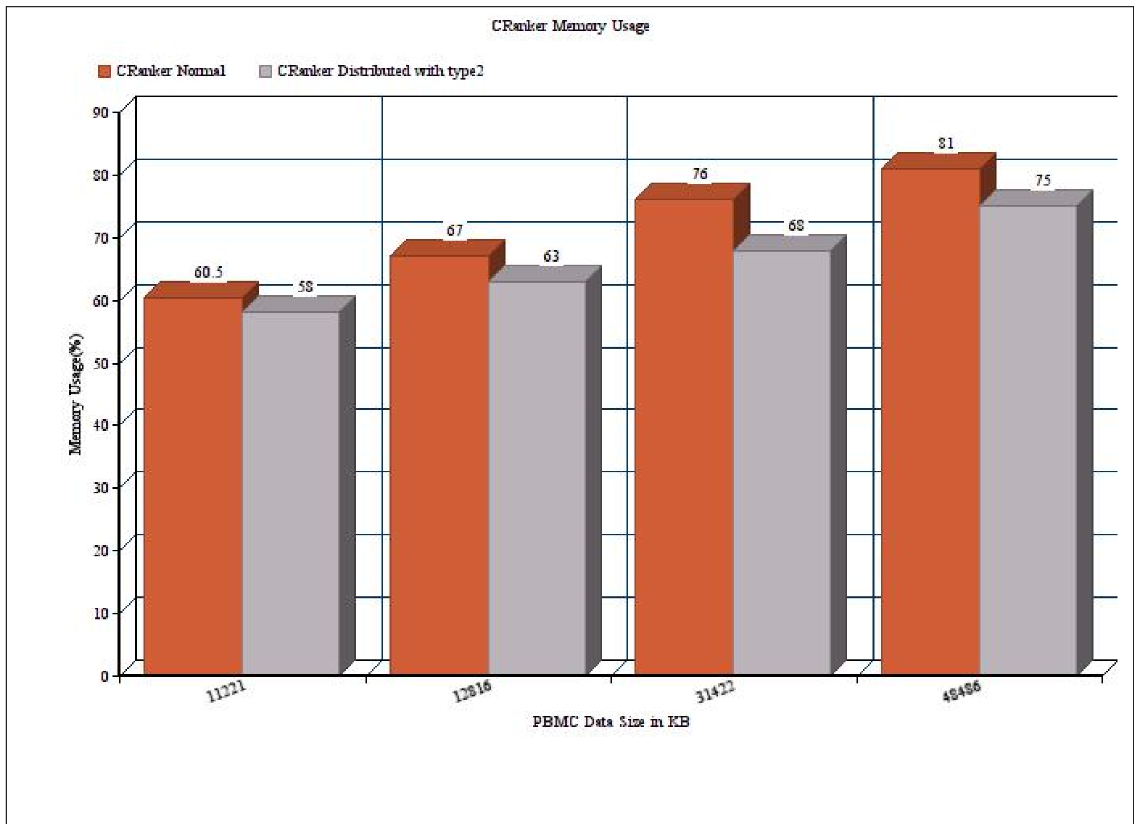


Figure 6.4: CRanker Normal Memory Usage vs Apache Hadoop Jobs Memory Consumption on Cluster 2

The reduction in memory consumption is due to usage of powerful data nodes in the Cluster 2. These data nodes are capable of handling multiple MapReduce jobs simultaneously which keeps the job execution successful. In most cases, there are no failures which keeps the Apache Hadoop resource manager free. This complete successful execution keeps the memory utilization free for most of the time.

To summarize all the memory consumption results of CRanker, a line graph is used. The line graph consists of the data that is collected from all the CRanker execution flows on the different testbeds. The figured chart 6.5 displays the significant improvement in the memory management of the CRanker application with the Apache Hadoop implementation.

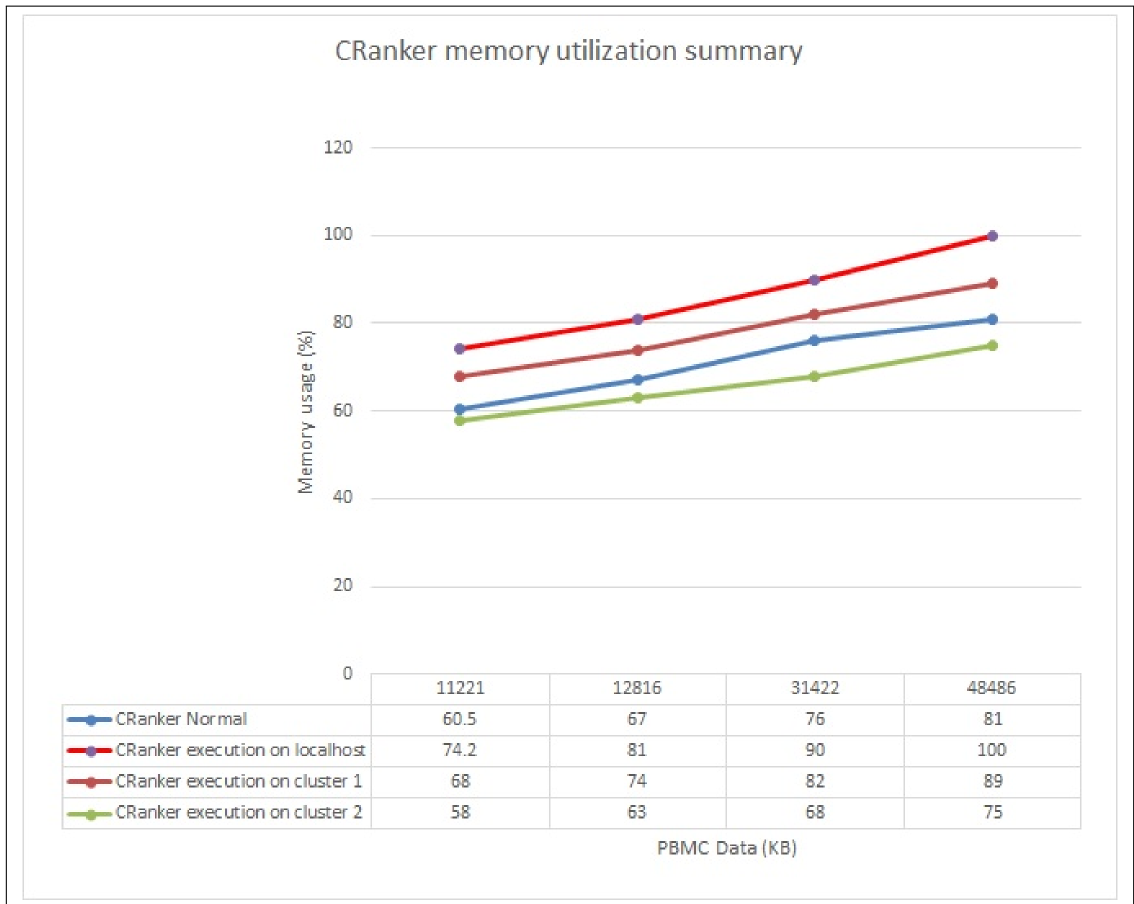


Figure 6.5: Summary of CRanker Memory Utilization

6.1.2 Evaluating the CRanker Execution Time

Section 6.1.2 discusses the CRanker execution time results and analyzes the reasons for the time taken to execute on different testbeds. The input datasets used for execution are "pbmc datasets", which is specified above. The first test is the comparison of CRanker normal flow execution time against the CRanker execution time on the Apache Hadoop

single node that is on the localhost. The results are bit disappointing because the CRanker execution using Apache Hadoop took longer to process the input datasets used. The time of execution is displayed in the figured graph 6.6.

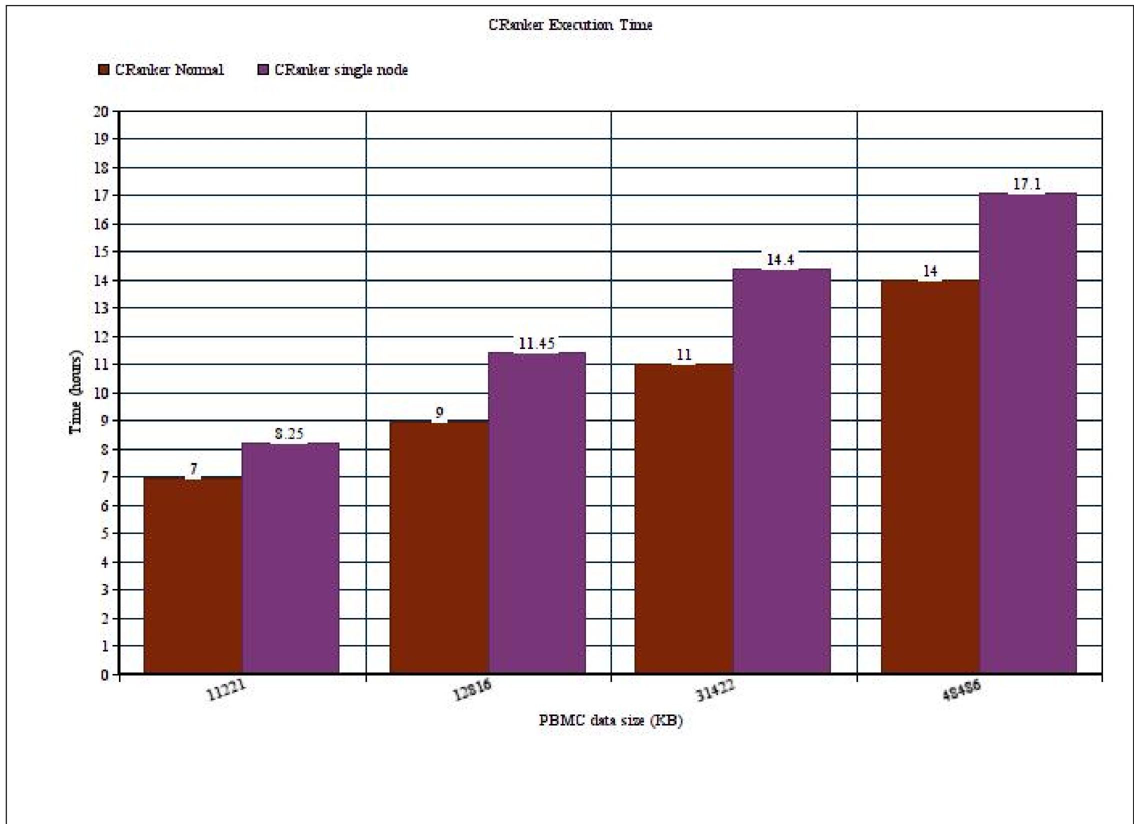


Figure 6.6: CRanker Execution Time:Normal Execution Vs Single Node Execution on Localhost

The longer time to execute CRanker on a single node at localhost is because of the memory consumption by both the Apache Hadoop and CRanker applications. The Apache Hadoop jobs may not have sufficient memory and CPU to be scheduled, and this may cause a delay in the execution. To evaluate the execution time of the CRanker on the cluster setup cluster 1 is used, and the results are evaluated against the normal execution flow of CRanker. The results are very interesting because for some data sets the execution

time is higher than the normal execution flow and for others the execution time is less than the normal execution. This is recorded and plotted as figured graph 6.7.

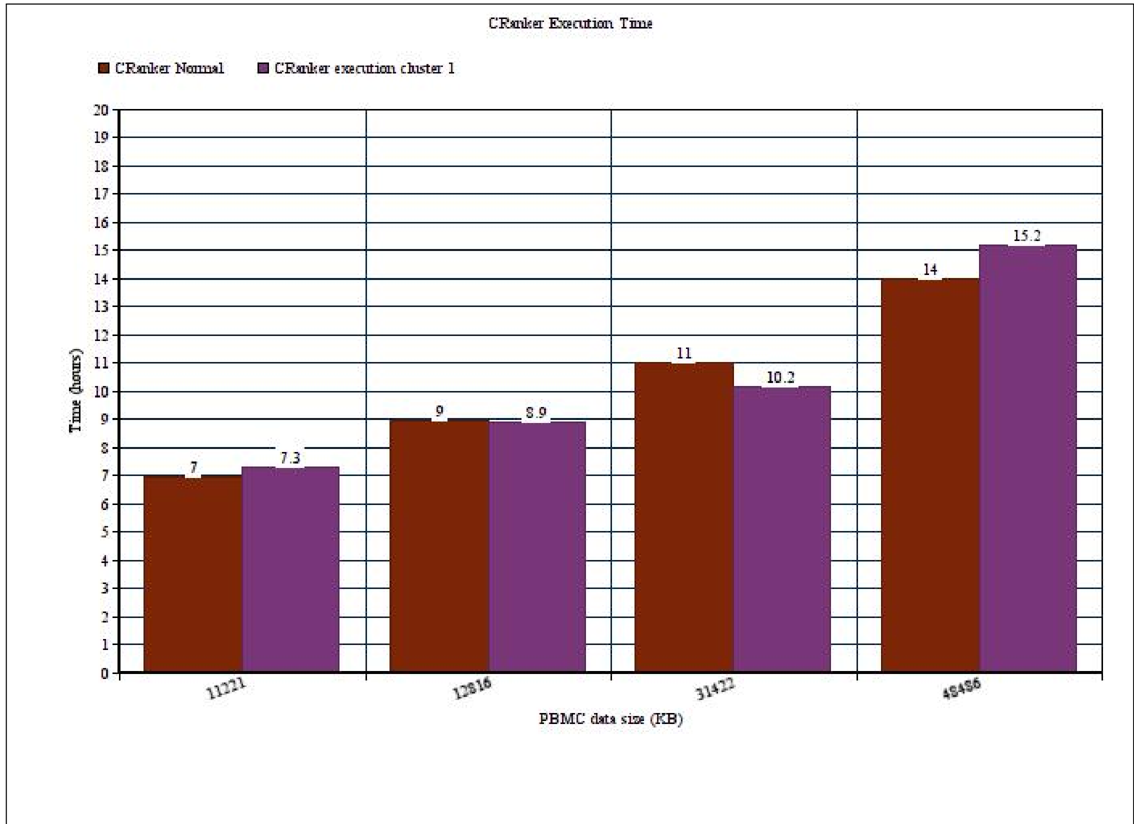


Figure 6.7: CRanker Execution Time:Normal Execution Vs Cluster 1

The values that are recorded in the graph 6.7 are the mean values based on four executions in the table 6.3 on cluster 1. During each execution the values differ and are not consistent. The inconsistency is mainly due to the amount of MapReduce job failures. The more the jobs fail due to a lack of memory, the more times the Apache Hadoop resource manager tries to execute the failed jobs, and this delays the CRanker execution time even more. The below table focuses on the CRanker execution times that were recorded during the test.

The execution results may not be convincing because at each instance, the total time

PBMC data (KB)	Attempt 1 (hrs)	Attempt 2 (hrs)	Attepmt 3 (hrs)	Attempt 4 (hrs)
11221	6.5	7.3	8.2	7.4
12816	9.9	6.8	8.9	10
31422	10.2	12	9.8	8.7
48486	15.2	17.1	14.8	13.6

Table 6.3: CRanker Execution Times on Cluster 1

taken to complete the MapReduce jobs are different. Moreover, this behavior completely depends on many factors like distribution cache, MapReduce job failures, and total time taken to execute. There are certain scenarios that the CRanker execution completely fails due to the lack of memory available on cluster 1 nodes. The scalable options are likely to increase the number of nodes in the cluster or increase the computational power of data nodes. Cluster 2 is then used to avoid the job failures and improve the execution condition. On cluster 2, when the CRanker distributed algorithm is started, the execution performance is much better than cluster 1 and the results are very promising. The recorded results are recorded in the figured graph 6.8 against the normal CRanker execution.

There is an increase in the performance of execution when the CRanker distributed execution algorithm is tested on "Cluster 2", mainly because of the "Type 2" data nodes. The MapReduce job distribution is even and well managed here due to the abundant availability of resources that reduce the MapReduce job failures. The overall results are satisfactory when the CRanker distributed execution algorithm is tested on the proper cluster. The results show that the proposed algorithm has greatly reduced the CRanker execution time. The above results clearly show that having more data nodes in the cluster can significantly increase the performance of CRanker and reduce the execution time. The summary of execution time by various CRanker approaches are depicted the figured graph 6.9:

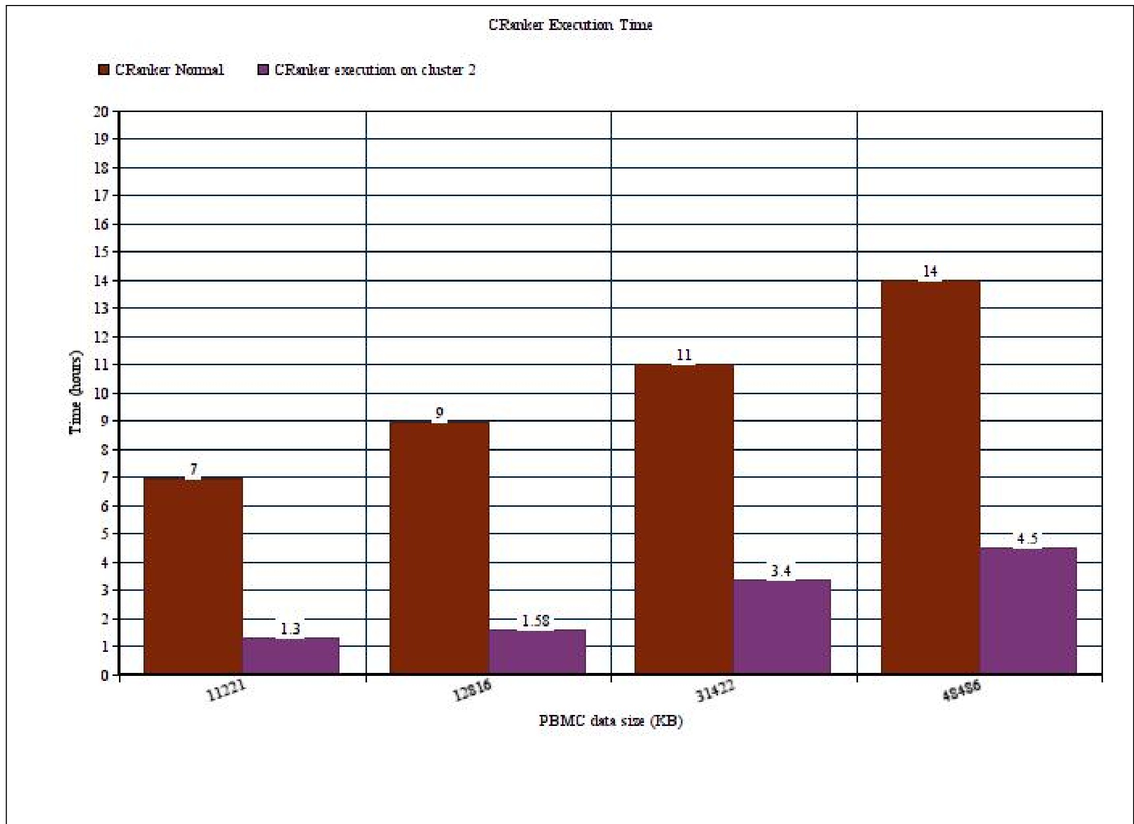


Figure 6.8: CRanker Execution Time:Normal Execution Vs Cluster 2

The main part of the evaluation and analysis is now concluded. The pending section describes the algorithm designed to compare the two CRanker output files.

6.2 Evaluation and Analysis of File Comparison Algorithm

Section 6.2 summarizes the comparison of output results generated by CRanker normal execution against the CRanker distributed execution, a step that could be the verification of the CRanker distributed execution algorithm. The graph below summarizes the percentage of the data matched with the distributed approach against the normal approach. It also depicts the matched percentage data.

Table 6.4 data is depicted in the figure graph **??**

The percentage calculated is based on the total number of matches divided by the

Figure 6.9: CRanker Execution Summary

file count value incremented with respect to the each row matched. It is represented by following a mathematical formula.

$$\frac{match \times 100}{filecount} \tag{6.1}$$

The evaluations and analysis of algorithms of the CRanker distributed execution and CRanker output comparison is hereby concluded.

pbmc input data set	Matched (%)
pbmc_ orbit_ mips.txt (%)	98
pbmc_ orbit_ nomips.txt (%)	96
pbmc_ velos_ mips.txt (%)	97
pbmc_ velos_ mips.txt (%)	96

Table 6.4: Matched Percentage of the CRanker Output

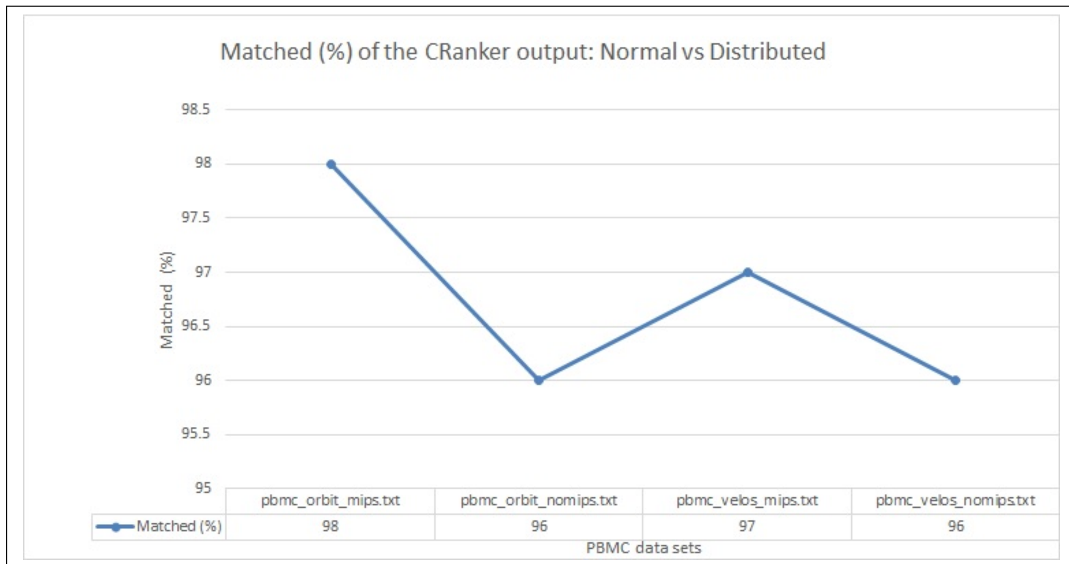


Figure 6.10: Matched Percentage of CRanker Output

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis, the main focus is to improve the performance and execution of CRanker by developing and implementing a CRanker distributed approach algorithm. Along with this, the CRanker output comparison algorithm is also developed to compare the output generated by different CRanker instances. The crux of the methodology is the joining of MapReduce to deal with the parallel execution of applications with the embodiment of programming and data in Amazon EC2 associated by networks.

The effects of the algorithms in CRanker distributed execution and CRanker output comparison results are also summarized. CRanker distributed execution utilizing Apache Hadoop displayed better performance over the most recent version of CRanker. CRanker distributed execution utilizing Apache Hadoop has advantages of being less difficult to improve, easier administration, and better maintainability because it effectively updates to new forms of CRanker. This framework can easily accommodate applications like CRanker with minimal changes. CRanker distributed execution utilizing Apache Hadoop indicated performance gains with expansions in the number of accessible processors. Recognizable distinction in execution time was observed when utilizing all resources. All of the programming segments, except Amazon EC2, utilized as a part of this work are open-source and accessible from the respectable project sites.

For applications with a dependency structure that fits the MapReduce paradigm, the CRanker distributed execution case study suggests that few if any, performance gains would result from using a different approach that requires reprogramming. Conversely, a MapReduce implementation such as Hadoop provides significant advantages such as management of failures, data, and jobs. It also provides advantages to CRanker concerns such as resource sharing, concurrency, scalability, and fault tolerance.

The conclusion also reinforces similar claims of proponents of the MapReduce approach and demonstrates them in the context of bioinformatics applications. Utilizing Amazon EC2 with software and data required for execution of both the application and MapReduce extraordinarily encourages the disseminated organization of successive codes. The middleware utilized for creation, cloning and administration of Amazon Machine Images (AMI) can be presented to clients for easy maintenance.

7.2 Future Work

There is much scope for extended research in this area. The developed framework can be tested with larger data sets of more than 1 GB in size on a cluster which has a large amount of computing nodes to test the scalability. This can be tested with other Bioinformatics algorithms which has the same execution behavior like CRanker. There is a need to create a virtual cloud across different locations and set up the framework to execute the distributed application instead of using Amazon EC2 cluster. Setting up the virtual private cloud is necessary for scenarios like this to reduce costs and secure the classified biological data. Using Apache Spark [Spark, 2014] instead of Apache MapReduce will compute the datasets 100 times faster, and is currently generating more research interest.

It can also be implemented using GPU computing [Owens et al., 2008], [NVidia, 2009] which is currently inaccessible for this particular thesis.

REFERENCES

Altschul, S. F., T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman (1997). Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research 25*(17), 3389–3402.

Amazon, E. (2010). Amazon elastic compute cloud (amazon ec2). *Amazon Elastic Compute Cloud (Amazon EC2)*.

Bondyopadhyay, P. K. (1998). Moore's law governs the silicon revolution. *Proceedings of the IEEE 86*(1), 78–81.

Claasen, T. A. (1999). High speed: not the only way to exploit the intrinsic computational power of silicon. In *Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International*, pp. 22–25. IEEE.

Dowd, K. (1993). *High performance computing*. O'Reilly.

Fields.scripps.edu (2015). Proteomic mass spectrometry, yates lab scripps research institute.

Ghemawat, S., H. Gobioff, and S.-T. Leung (2003). The google file system. In *ACM SIGOPS operating systems review*, Volume 37, pp. 29–43. ACM.

Hadoop, A. (2011). Hadoop.

Hadoop, A. (2015). Hadoop cluster setup.

Jian, L., X. Niu, Z. Xia, P. Samir, C. Sumanasekera, Z. Mu, J. L. Jennings, K. L. Hoek, T. Allos, L. M. Howard, et al. (2013). A novel algorithm for validating peptide identification from a shotgun proteomics search engine. *Journal of proteome research 12*(3), 1108–1119.

Käll, L., J. D. Canterbury, J. Weston, W. S. Noble, and M. J. MacCoss (2007). Semi-supervised learning for peptide identification from shotgun proteomics datasets. *Nature methods 4*(11), 923–925.

Keller, A., A. I. Nesvizhskii, E. Kolker, and R. Aebersold (2002). Empirical statistical model to estimate the accuracy of peptide identifications made by ms/ms and database search. *Analytical chemistry 74*(20), 5383–5392.

Labrinidis, A. and H. Jagadish (2012). Challenges and opportunities with big data. *Proceedings of the VLDB Endowment 5*(12), 2032–2033.

Langmead, B., K. D. Hansen, J. T. Leek, et al. (2010). Cloud-scale rna-sequencing differential expression analysis with myrna. *Genome Biol 11*(8), R83.

Lehninger, A. L., D. L. Nelson, and M. M. Cox (2005). *Lehninger principles of biochemistry*. W.H. Freeman.

Liang, X., Z. Xia, X. Niu, and A. Link (2014). A weighted classification model for peptide identification. In *Computational Advances in Bio and Medical Sciences (ICCABS), 2014 IEEE 4th International Conference on*, pp. 1–2. IEEE.

Ma, K., O. Vitek, and A. I. Nesvizhskii (2012). A statistical model-building perspective to identification of ms/ms spectra with peptideprophet. *BMC bioinformatics 13*(Suppl 16), S1.

Manyika, J., M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers (2011). Big data: The next frontier for innovation, competition, and productivity.

Matrixscience. Sequence database searching.

Matrixscience.com (2015). Introduction to mascot server | protein identification software for mass spec data.

NVidia, F. (2009). Nvidia's next generation cuda compute architecture. *NVidia, Santa Clara, Calif, USA*.

Owens, J. D., M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips (2008). Gpu computing. *Proceedings of the IEEE 96*(5), 879–899.

Pappin, D. J., P. Hojrup, and A. J. Bleasby (1993). Rapid identification of proteins by peptide-mass fingerprinting. *Current biology 3*(6), 327–332.

Pearson, W. R. and D. J. Lipman (1988). Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences 85*(8), 2444–2448.

Schmeisser, M., B. C. Heisen, M. Luettich, B. Busche, F. Hauer, T. Koske, K.-H. Knauber, and H. Stark (2009). Parallel, distributed and gpu computing technologies in single-particle electron microscopy. *Acta Crystallographica Section D: Biological Crystallography 65*(7), 659–671.

Seidler, J., N. Zinn, M. E. Boehm, and W. D. Lehmann (2010). De novo sequencing of peptides by ms/ms. *Proteomics 10*(4), 634–649.

Spark, A. (2014). Apache spark–lightning-fast cluster computing.

Thegpm.org (2015). X! tandem.

Wang, J., D. Crawl, I. Altintas, K. Tzoumas, and V. Markl (2013). Comparison of distributed data-parallelization patterns for big data analysis: A bioinformatics case study. In *Proceedings of the Fourth International Workshop on Data Intensive Computing in the Clouds (DataCloud)*.

Weinberg, D. H., T. Beers, M. Blanton, D. Eisenstein, H. Ford, J. Ge, B. Gillespie, J. Gunn, M. Klaene, G. Knapp, et al. (2007). Sdss-iii: Massive spectroscopic surveys of the distant universe, the milky way galaxy, and extra-solar planetary systems. In *Bulletin of the American Astronomical Society*, Volume 39, pp. 963.

White, T. (2012). *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".

Xia, Z. (2013). Setting up of cranker.

Zaharia, M., A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica (2008). Improving mapreduce performance in heterogeneous environments. In *OSDI*, Volume 8, pp. 7.

Zealandpharma.com (2015). Zealand pharma - what are peptides.

Zhu, Y. and H. Jiang (2006). Ceft: A cost-effective, fault-tolerant parallel virtual file system. *Journal of Parallel and Distributed Computing 66*(2), 291–306.

Appendices

Appendix A

MAPREDUCE CODE FOR CRANKER DISTRIBUTED EXECUTION

This section shows the MapReduce code for CRanker Distributed Execution

A.A The driver for MapReduce CRanker distributed execution

```
1  package com.wku.mrexecutor.driver;
2
3  import java.io.BufferedReader;
4  import java.io.File;
5  import java.io.FileReader;
6  import java.io.IOException;
7
8  import org.apache.hadoop.conf.Configuration;
9  import org.apache.hadoop.fs.FileSystem;
10 import org.apache.hadoop.fs.LocatedFileStatus;
11 import org.apache.hadoop.fs.Path;
12 import org.apache.hadoop.fs.RemoteIterator;
13 import org.apache.hadoop.io.IntWritable;
14 import org.apache.hadoop.io.Text;
15 import org.apache.hadoop.mapreduce.Job;
16 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
17 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
18 import org.apache.hadoop.util.GenericOptionsParser;
19 import org.codehaus.jettison.json.JSONArray;
20 import org.codehaus.jettison.json.JSONException;
21 import org.codehaus.jettison.json.JSONObject;
22
23 import com.wku.mrexecutor.mapper.ExecutorMapper;
24
25 /**
26  *
27  * Main driver to execute certain algorithms using MapReduce framework.
28  *
29  */
30 public class Driver {
31
32   public static void main(String[] args) throws IOException,
33       ClassNotFoundException, InterruptedException, JSONException {
34
35     Configuration conf = new Configuration();
36
37     //Parse and set Hadoop related properties (set with -D) that are
       passed as arguments
38     String[] otherArgs = new GenericOptionsParser(conf, args)
39         .getRemainingArgs();
40
41     if (otherArgs.length < 2) {
42       System.err
```

```
43          .println("Usage: mrexecutor <algorithm> <properties_json_path>
    ");
44        System.exit(2);
45      }
46
47      // Read JSON configuration file
48      System.out.println("Reading properties json file...");
49      FileReader confFileReader = new FileReader(new File(otherArgs[1]));
50      BufferedReader br = new BufferedReader(confFileReader);
51
52      String confStr = "";
53      String line = null;
54
55      while ((line = br.readLine()) != null) {
56        confStr += line;
57      }
58
59      confFileReader.close();
60      br.close();
61
62      System.out.println("Properties json file read successfully...");
63
64      //Parse JSON String into JSON Object and get required key-values.
65      JSONObject jobConf = new JSONObject(confStr);
66      JSONArray algorithms = jobConf.getJSONArray("algorithms");
67      JSONObject currentAlgoJSON = null;
68
69      for (int i = 0; i < algorithms.length(); i++) {
70        //Get properties JSON object for algorithm to be executed
71        if (algorithms.getJSONObject(i).getString("name")
72            .equalsIgnoreCase(otherArgs[0])) {
73          currentAlgoJSON = algorithms.getJSONObject(i);
74        }
75      }
76
77      //If properties JSON Object is null, it's not been in set in
    properties file, abort.
78      if (currentAlgoJSON == null) {
79        System.out.println("FATAL: Configuration for algorithm '"
80            + otherArgs[0]
81            + "', could not be found in configuration file, '"
82            + otherArgs[1] + "'. Aborting.");
83        System.exit(1);
84      }
85
86      // Set algorithm specific values from config JSON
87      conf.set("OUT_DIR", currentAlgoJSON.getString("hdfs_out_dir"));
88      conf.set("ALGO_BIN_HOME", currentAlgoJSON.getString("binary_dir"));
89      conf.setBoolean("ADD_DATA_HEADER", currentAlgoJSON.getBoolean("
    add_data_header"));
90      conf.set("DATA_HEADER", currentAlgoJSON.getString("data_header"));
91
92      //Get the list of commands that are to be executed as pert of this
    algorithm
```

```java
93      JSONArray executables = currentAlgoJSON.getJSONArray("executables");
94      String[] cmd = new String[executables.length()];
95      for (int i = 0; i < executables.length(); i++) {
96        cmd[i] = executables.getJSONObject(i).getString("command");
97      }
98      conf.setStrings("COMMANDS", cmd);
99
100     // Set generic values from config JSON
101     conf.set("STAGE_DIR", jobConf.getString("stage_dir"));
102     conf.set("MCR_ROOT", jobConf.getString("mcr_root"));
103     conf.set("MCR_CACHE_ROOT", jobConf.getString("mcr_cache_root"));
104
105     //Set Job properties
106     Job job = Job.getInstance(conf);
107     job.setJobName(otherArgs[0] + "-MR-Executor");
108     job.setJarByClass(Driver.class);
109     job.setMapperClass(ExecutorMapper.class);
110     job.setNumReduceTasks(0);
111     job.setMapOutputKeyClass(Text.class);
112     job.setMapOutputValueClass(IntWritable.class);
113
114     //Set input file path. This path should be HDFS one.
115     //Mappers will write input splits from this input file to a staging
        area on local file system.
116     //These staged input files will then be given to algorithm,
        executables.
117     FileInputFormat.addInputPath(job,
118         new Path(currentAlgoJSON.getString("hdfs_in_dir")));
119
120     //Algorithm executables will produce output files in staging area
        which will be copied to HDFS
121     //directory represented by below path.
122     FileOutputFormat.setOutputPath(job,
123         new Path(currentAlgoJSON.getString("hdfs_out_dir")));
124
125     System.out.println("Submitting job on the cluster...");
126     int success = job.waitForCompletion(true) ? 0 : 1;
127
128     if(success==0) {
129       System.out.println("Job completed successfully...");
130     } else {
131       System.out.println("Job failed. Aborting.");
132       System.exit(1);
133     }
134
135     //Get handler to HDFS output directory
136     FileSystem fs = FileSystem.newInstance(conf);
137     RemoteIterator<LocatedFileStatus> i = fs.listFiles(new Path(
138         currentAlgoJSON.getString("hdfs_out_dir")), false);
139
140     System.out.println("Cleaning up part files from hdfs output
        directory...");
141     while (i.hasNext()) {
142       LocatedFileStatus f = i.next();
```

```
143        //Delete all the 'part' files generated by Mappers.
144        //These part files are empty and do not contain output.
145        //Actual output is contained by txt files written by algorithm
     executables
146        if (f.isFile() && f.getPath().getName().startsWith("part-")) {
147          fs.delete(f.getPath(), true);
148        }
149      }
150      fs.close();
151
152      System.out.println("Done. Exiting.");
153      System.exit(success);
154    }
155  }
```

A.B The MapReduce code for CRanker distributed execution

```java
package com.wku.mrexecutor.mapper;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileFilter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

/**
 * This mapper does nothing but writes input splits to a staging area
 * on local file system and then execute binaries of algorithms which
   use
 * these staged files and produce output.
 *
 * These output files are then copied back to HDFS by this mapper in its
    cleanup step.
 *
 *  setup() phase initializes required properties and objects.
 *  map() phase writes records from input split to a staging area.
 *  cleanup() step triggers required algorithms and copies back their
   output to HDFS.
 *
 */
public class ExecutorMapper extends Mapper<Object, Text, Text,
   IntWritable> {

  //Logs will be accessible on this app's Application master's Web UI
   and Job History server.
  private final static Log logger = LogFactory.getLog(ExecutorMapper.
   class);

  //Points to directory on local file system where intermediate files
   are staged.
  private String stagingBaseDirname = "";
  //Points to directory on local file system where intermediate input
   files to algorithm are staged.
  private String stagingInDirname = "";
  //Points to directory on local file system where intermediate ouput
   files from an algorithm are staged.
  private String stagingOutDirname = "";
```

```
44  //Points to file in input staging dir in which input split records are
      written.
45  private String stagingInputFile = "";
46
47  private String hdfsOutDir = "";
48
49  private FileWriter stagedInputFileWriter = null;
50  //This mapper's task id.
51  private String myTaskId = "";
52  //This mapper's attempt id.
53  private String myAttemptId = "";
54
55  private String dataHeader = null;
56  private String algoBinHome = null;
57  private String mcrRoot = null;
58
59  /**
60   * Prepares this mapper instance for execution.
61   *  1. Reads configuration values.
62   *  2. Opens file writer to intermediate staging input file on local
     file system.
63   *  3. Writes header to this intermediate staging input file if
     required.
64   */
65  @Override
66  protected void setup(Mapper<Object, Text, Text, IntWritable>.Context
     context)
67      throws IOException, InterruptedException {
68    super.setup(context);
69    //Get task and attempt ids.
70    myTaskId = context.getTaskAttemptID().getTaskID().toString();
71    myAttemptId = context.getTaskAttemptID().toString();
72
73    logger.info("My task id = " + myTaskId + ", my attempt id = " +
     myAttemptId);
74
75    logger.info("Reading properties from configuration object");
76    //Read properties from job configuration. These are set in Driver.
77    hdfsOutDir = context.getConfiguration().get("OUT_DIR");
78    logger.debug("hdfsOutDir=" + hdfsOutDir);
79    stagingBaseDirname = context.getConfiguration().get("STAGE_DIR") + "
     /"
80        + myTaskId + "/" + myAttemptId;
81    logger.debug("stagingBaseDirname=" + stagingBaseDirname);
82    algoBinHome = context.getConfiguration().get("ALGO_BIN_HOME");
83    logger.debug("algoBinHome=" + algoBinHome);
84    mcrRoot = context.getConfiguration().get("MCR_ROOT");
85    logger.debug("mcrRoot=" + mcrRoot);
86    dataHeader = context.getConfiguration().get("DATA_HEADER");
87    logger.debug("dataHeader=" + dataHeader);
88
89    stagingInDirname = stagingBaseDirname + "/in/";
90    stagingOutDirname = stagingBaseDirname + "/out/";
91
```

```
92      //Create staging directories.
93      logger.info("Creating input staging directory: " + new File(
        stagingInDirname).mkdirs());
94      logger.info("Creating output staging directory: " + new File(
        stagingOutDirname).mkdirs());
95
96      stagingInputFile = stagingInDirname + "/" + myAttemptId + ".txt";
97      logger.debug("stagingInDirname=" + stagingInDirname);
98
99      File sf = new File(stagingInputFile);
100     sf.createNewFile();
101     stagedInputFileWriter = new FileWriter(sf);
102     logger.info("Opened staging input file writer");
103
104     //Write header to this mapper's staged input file if required.
105     if(context.getConfiguration().getBoolean("ADD_DATA_HEADER", false))
        {
106       logger.info("Header written to staging input file");
107       stagedInputFileWriter.write(dataHeader + "\n");
108     }
109   }
110
111   /**
112    * Write each record from input split to intermediate staging file on
        local file system
113    * Each mappper will have it's own such file. Also, if algorithm
        expects header in input file, then each of this file should have
        header too.
114    * This header can be set in properties JSON
115    */
116   public void map(Object key, Text value, Context context)
117       throws IOException, InterruptedException {
118     stagedInputFileWriter.write(value.toString() + "\n");
119   }
120
121   /**
122    * Releases resources.
123    * Then triggers required algorithm with intermediate staged input
        file.
124    * And output from this algorithm is written back to HDFS.
125    *
126    */
127   @Override
128   protected void cleanup(
129       Mapper<Object, Text, Text, IntWritable>.Context context)
130       throws IOException, InterruptedException {
131     super.cleanup(context);
132     logger.debug("Closing staged input file writer");
133     stagedInputFileWriter.flush();
134     stagedInputFileWriter.close();
135
136     //Set environment variables for algorithm's shell scripts
137     String[] env = new String[1];
```

```
138     env[0] = "MCR_CACHE_ROOT=" + context.getConfiguration().get("
      MCR_CACHE_ROOT", "/tmp");
139
140     logger.debug("MCR_CACHE_ROOT = " + env[0]);
141
142     //Get the list of commands that are required to trigger algorithm.
143     String[] commands = context.getConfiguration().getStrings("COMMANDS"
      );
144
145     //Maintain a map of input and temporary file paths passed to
      algorithm scripts.
146     //This allows us to pass same paths to multiple, different scripts
      within same algorithm.
147     Map<String, String> argFileMap = new HashMap<String, String>();
148     argFileMap.put("%INPUT_FILE%", stagingInputFile);
149
150     int tmp_counter = 0;
151     for (String cmd : commands) {
152       logger.debug("Found command string = " + cmd);
153       //Replace standard variables from shell script arguments.
154       cmd = algoBinHome + "/" + cmd.replaceAll("%MCR_ROOT%", mcrRoot);
155       cmd = cmd.replaceAll("%INPUT_FILE%", stagingInputFile);
156
157       String[] tokens = cmd.split(" ");
158       for (String tok : tokens) {
159         //Check if a temporary .mat file path is to be passed to current
       command.
160         if(tok.startsWith("%TMP_MAT_FILE_")) {
161           if(argFileMap.get(tok) == null) {
162             argFileMap.put(tok, stagingOutDirname + "/" + myAttemptId +
      "_" + tmp_counter + ".mat");
163             tmp_counter++;
164           }
165           cmd = cmd.replaceAll(tok, argFileMap.get(tok));
166         }
167       }
168       //Trigger the execution of one script from this algorithm.
169       logger.debug("Executable command string = '" + cmd + "'");
170       executeSh(cmd, env);
171     }
172
173     logger.info("Copying output files to HDFS");
174     //Copy output .txt files generated by algorithm scripts to HDFS
      output directory.
175     FileSystem fs = FileSystem.newInstance(context.getConfiguration());
176     File outFiles = new File(stagingOutDirname);
177     File[] txtFiles = outFiles.listFiles(new FileFilter() {
178       public boolean accept(File pathname) {
179         return (pathname.isFile() && pathname.toString().endsWith(
180             ".txt"));
181       }
182     });
183     //Copy each .txt file from staged output directory on local file
      system to HDFS
```

```
184       for (File txtFile : txtFiles) {
185         Path srcPath = new Path(txtFile.toString());
186         Path destPath = new Path(hdfsOutDir);
187         logger.debug("Copying output file " + txtFile.toString() + " to "
      + hdfsOutDir + " on HDFS");
188         fs.copyFromLocalFile(srcPath, destPath);
189       }
190       logger.debug("Closing FS handler");
191       fs.close();
192     }
193 
194     /**
195      * Private utility method to trigger execution of script and read its
       output and error streams.
196      *
197      * @param command Shell command to be executed
198      * @param env Environment variables to be set for this shell execution
199      * @throws IOException
200      * @throws InterruptedException
201      */
202     private void executeSh(String command, String[] env) throws
      IOException,
203         InterruptedException {
204 
205       logger.debug("Starting execution of command '" + command + "'");
206       Process p = Runtime.getRuntime().exec(command, env);
207 
208       logger.debug("Reading output stream");
209       InputStream is = p.getInputStream();
210       InputStreamReader isr = new InputStreamReader(is);
211       BufferedReader br = new BufferedReader(isr);
212       String in = "";
213       do {
214         logger.debug(in);
215         in = br.readLine();
216       } while (in != null);
217 
218       logger.debug("Reading error stream");
219       InputStream es = p.getErrorStream();
220       InputStreamReader esr = new InputStreamReader(es);
221       BufferedReader ebr = new BufferedReader(esr);
222       String ein = "";
223       do {
224         logger.debug(ein);
225         ein = ebr.readLine();
226       } while (ein != null);
227 
228       logger.info("Process exited with status = " + p.waitFor());
229     }
230   }
```

A.B.1 CRanker Execution Command

To Trigger the CRanker execution on the Apache Hadoop Cluster the following command is issued on the Apache Hadoop master node terminal

```
hadoop/bin/yarn jar mr-executor/target/mrexecutor-0.2-SNAPSHOT.jar com.
    wku.mrexecutor.driver.Driver cranker mr-executor/properties.json
'spectrum peptide protein ions xcorr deltacn sprank hit_mass'
```

Appendix B
MAPREDUCE CODE FOR FILE COMPARISON
B.A MapReduce Code for File Comparison Using Joins

```
package com.wku;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class CSVCompare {

  public static void main(String[] args) {

    if (args.length != 5) {
      System.err
          .println("Incorrect arguments. Expected arguments: <xls file name 1> <xls file name 2> <column file1> <column file2> <output path>");
      return;
    }
    int col1 = 0;
    int col2 = 0;
    try {
      col1 = Integer.parseInt(args[2]);
      col2 = Integer.parseInt(args[3]);
    } catch (Exception e) {
      col1 = 0;
      col2 = 0;
    }
    col1--;
    col2--;
    if (col1 < 0 || col2 < 0) {
      System.err
          .println("Incorrect arguments. Expected arguments: <xls file name 1> <xls file name 2> <column file1> <column file2> <output path>");
      System.err
```

```
41          .println("<column file1> and <column file2> must be an integer
    and greater than 0");
42        return;
43      }
44
45      Configuration conf = new Configuration();
46      conf.setInt("com.wku.file1col", col1);
47      conf.setInt("com.wku.file2col", col2);
48
49      Job job;
50      try {
51        job = new Job(conf, "xlscompare");
52      } catch (IOException e) {
53        e.printStackTrace();
54        return;
55      }
56
57      job.setJarByClass(CSVCompare.class);
58      job.setOutputKeyClass(Text.class);
59      job.setOutputValueClass(Text.class);
60
61      job.setReducerClass(Reduce.class);
62
63      job.setInputFormatClass(TextInputFormat.class);
64      job.setOutputFormatClass(TextOutputFormat.class);
65
66      MultipleInputs.addInputPath(job, new Path(args[0]),
67          TextInputFormat.class, File1Mapper.class);
68      MultipleInputs.addInputPath(job, new Path(args[1]),
69          TextInputFormat.class, File2Mapper.class);
70      FileOutputFormat.setOutputPath(job, new Path(args[4]));
71
72      try {
73        job.waitForCompletion(true);
74      } catch (ClassNotFoundException | IOException | InterruptedException
    e) {
75        e.printStackTrace();
76        return;
77      }
78    }
79
80    public static class File1Mapper extends
81        Mapper<LongWritable, Text, Text, Text> {
82      private final static String fileTag = "F1~";
83
84      public void map(LongWritable key, Text value, Context context)
85          throws IOException, InterruptedException {
86        int colNum = context.getConfiguration().getInt(
87            "com.wku.file1col", 0);
88        String[] cols = value.toString().split(
89            "[,;](?=([^\"]*\"[^\"]*\")*[^\"]*$)");
90        if (colNum < cols.length) {
91          String strkey = cols[colNum];
92          // Remove quotes from string
```

```
93          if (strkey.charAt(0) == '"'
94              && strkey.charAt(strkey.length() - 1) == '"') {
95            strkey = strkey.substring(1, strkey.length() - 1);
96          }
97          context.write(new Text(strkey),
98              new Text(fileTag + value.toString()));
99        }
100     }
101   }
102
103   public static class File2Mapper extends
104       Mapper<LongWritable, Text, Text, Text> {
105     private final static String fileTag = "F2~";
106
107     public void map(LongWritable key, Text value, Context context)
108         throws IOException, InterruptedException {
109       int colNum = context.getConfiguration().getInt(
110           "com.wku.file2col", 0);
111       String[] cols = value.toString().split(
112           "[,;](?=([^\"]*\"[^\"]*\")*[^\"]*$)");
113       if (colNum < cols.length) {
114         String strkey = cols[colNum];
115         // Remove quotes from string
116         if (strkey.charAt(0) == '"'
117             && strkey.charAt(strkey.length() - 1) == '"') {
118           strkey = strkey.substring(1, strkey.length() - 1);
119         }
120         context.write(new Text(strkey),
121             new Text(fileTag + value.toString()));
122       }
123     }
124   }
125
126   public static class Reduce extends Reducer<Text, Text, Text, Text> {
127
128     private int File1Count = 0;
129     private int File2Count = 0;
130     private int Match = 0;
131
132     private static final String File1Tag = "F1";
133     private static final String File2Tag = "F2";
134
135     public void reduce(Text key, Iterable<Text> values, Context context)
136         throws IOException, InterruptedException {
137       // Update file counters and check if line matches
138       int sum = 0;
139       String line = null;
140       for (Text val : values) {
141         String tags[] = val.toString().split("~");
142         if (tags[0].equals(File1Tag)) {
143           line = tags[1];
144           File1Count++;
145         }
146         if (tags[0].equals(File2Tag))
```

```
147             File2Count++;
148           sum++;
149         }
150         if (sum == 2) {
151           // Write whole line
152           context.write(null, new Text(line));
153           // Update match count
154           Match++;
155         }
156       }
157 
158       @Override
159       protected void cleanup(Context context) throws IOException,
160           InterruptedException {
161         // Write count report
162         if (File1Count == File2Count) {
163           // Same number of rows in both files
164           context.write(new Text("Match percent: "), new Text(
165               Double.toString((Match * 100 / (double) File1Count))
166                   + " (" + Match + " out of " + File1Count + ")"));
167         } else {
168           // Different number of rows
169           context.write(new Text("File 1 match percent: "), new Text(
170               Double.toString((Match * 100 / (double) File1Count))
171                   + " (" + Match + " out of " + File1Count + ")"));
172           context.write(new Text("File 2 match percent: "), new Text(
173               Double.toString((Match * 100 / (double) File2Count))
174                   + " (" + Match + " out of " + File2Count + ")"));
175         }
176       }
177     }
178 }
```

Appendix C
APACHE HADOOP CONFIGURATION FILES

Chapter C displays the configuration files used in the Apache Hadoop cluster.

C.A Apache Hadoop Master Node configuration Files

Section **??** displays the configurations needed to setup the Apache Hadoop Master node in a cluster

C.A.1 MapReduce Configuration

MapReduce configuration parameters are stored in mapred-site.xml file. The configurations made in this file will override the defaults of MapReduce parameters

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>

  <!--property>
  <name>mapred.job.tracker</name>
  <value>hadoopmaster:54311</value>

  -->

  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
    <description>The runtime framework for executing MapReduce jobs.
      Can be one of local, classic or yarn.
    </description>
  </property>

  <property>
    <name>mapreduce.jobtracker.address</name>
    <value>local</value>
```

```
38      <description>The host and port that the MapReduce job tracker runs
39        at. If "local", then jobs are run in-process as a single map
40        and reduce task.
41      </description>
42    </property>
43
44    <property>
45      <name>mapred.task.timeout</name>
46      <value>18000000</value>
47    </property>
48
49
50  </configuration>
```

C.A.2 HDFS Configuration

HDFS configuration parameters are stored in hdfs-site.xml. The configurations made in this file will override the default parameters of HDFS. In this file user can define the replication factor, block size, DataNode, and NameNode locations.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
  <configuration>

    <property>
      <name>dfs.datanode.data.dir</name>
      <value>file:///home/ubuntu/hadoop_dfs/data/datanode</value>
      <description>DataNode directory</description>
    </property>

    <property>
      <name>dfs.namenode.name.dir</name>
      <value>file:///home/ubuntu/hadoop_dfs/data/namenode</value>
      <description>NameNode directory for namespace and transaction logs
     storage.</description>
    </property>



    <property>
      <name>dfs.replication</name>
      <value>2</value>
    </property>

    <property>
      <name>dfs.permissions</name>
      <value>false</value>
    </property>


    <property>
```

```
48        <name>dfs.blocksize</name>
49        <value>512k</value>
50        <description>
51          The default block size for new files, in bytes.
52          You can use the following suffix (case insensitive):
53          k(kilo), m(mega), g(giga), t(tera), p(peta), e(exa) to specify
      the size (such
54          as 128k, 512m, 1g, etc.),
55          Or provide complete size in bytes (such as 134217728 for 128 MB)
      .
56        </description>
57      </property>
58
59
60      <property>
61        <name>dfs.namenode.fs-limits.min-block-size</name>
62        <value>32768</value>
63        <description>Minimum block size in bytes, enforced by the Namenode
       at create
64          time. This prevents the accidental creation of files with tiny
      block
65          sizes (and thus many blocks), which can degrade
66          performance.
67        </description>
68      </property>
69
70      <!--
71  <property>
72  <name>dfs.namenode.fs-limits.min-block-size</name>
73  <value>100</value>
74  <description>minimum block size of the data</description>
75  </property>
76
77  -->
78
79      <property>
80        <name>dfs.datanode.use.datanode.hostname</name>
81        <value>false</value>
82      </property>
83      <property>
84        <name>dfs.namenode.datanode.registration.ip-hostname-check</name>
85        <value>false</value>
86      </property>
87
88
89
90      <!--
91  <property>
92   <name>dfs.namenode.http-address</name>
93   <value>ec2-52-10-149-153.us-west-2.compute.amazonaws.com:50070</value>
94   <description>Your NameNode hostname for http access.</description>
95  </property>
96
97  <property>
```

```
98  <name>dfs.namenode.secondary.http-address</name>
99  <value>ec2-52-10-199-242.us-west-2.compute.amazonaws.com:50090</value>
100 <description>Your Secondary NameNode hostname for http access.</
    description>
101 </property>
102 -->
103
104     <property>
105       <name>dfs.namenode.rpc-address</name>
106       <value>hadoopmaster:9000</value>
107       <description>
108         RPC address that handles all clients requests. In the case of HA
    /Federation where multiple namenodes exist,
109         the name service id is added to the name e.g. dfs.namenode.rpc-
    address.ns1
110         dfs.namenode.rpc-address.EXAMPLENAMESERVICE
111         The value of this property will take the form of nn-host1:rpc-
    port.
112       </description>
113     </property>
114
115   </configuration>
```

C.A.3 Core Site Configuration

NameNode is identified based on the configuration settings in the core-site.xml. All the master and slave node should point their NameNodes to the single URI

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://hadoopmaster:9000</value>
    <description>Namenode URI</description>
  </property>
</configuration>
```

C.A.4 Apache Hadoop Yarn Configuration

Yarn configuration parameters are stored in yarn-site.xml file. The values that configured in this file will override the default values of yarn. This configurations in this file decides the ResourceManager and NodeManager function

```
<?xml version="1.0"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->
<configuration>

  <!-- Site specific YARN configuration properties -->


  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.resourcemanager.resource-tracker.address</name>
    <value>hadoopmaster:8025</value>
  </property>
  <property>
    <name>yarn.resourcemanager.scheduler.address</name>
    <value>hadoopmaster:8030</value>
  </property>
  <property>
    <name>yarn.resourcemanager.address</name>
    <value>hadoopmaster:8050</value>
  </property>

  <property>
    <name>yarn.nodemanager.pmem-check-enabled</name>
    <value>false</value>
  </property>

  <property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
```

```
49   </property>
50
51
52   <property>
53     <description>The hostname of the RM.</description>
54     <name>yarn.resourcemanager.hostname</name>
55     <value>hadoopmaster</value>
56   </property>
57
58   <property>
59     <description>Whether physical memory limits will be enforced for
60       containers.
61     </description>
62     <name>yarn.nodemanager.pmem-check-enabled</name>
63     <value>false</value>
64   </property>
65
66   <property>
67     <description>Whether virtual memory limits will be enforced for
68       containers.
69     </description>
70     <name>yarn.nodemanager.vmem-check-enabled</name>
71     <value>false</value>
72   </property>
73
74   <property>
75     <description>Whether to enable log aggregation. Log aggregation
    collects
76       each container's logs and moves these logs onto a file-system, for
     e.g.
77       HDFS, after the application completes. Users can configure the
78       "yarn.nodemanager.remote-app-log-dir" and
79       "yarn.nodemanager.remote-app-log-dir-suffix" properties to
    determine
80       where these logs are moved to. Users can access the logs via the
81       Application Timeline Server.
82     </description>
83     <name>yarn.log-aggregation-enable</name>
84     <value>true</value>
85   </property>
86
87 </configuration>
```

C.B Apache Hadoop Slave Nodes configuration Files

Section C.B displays the configuration properties required to setup Apache Hadoop slave nodes, in a cluster all the slaves have the same configuration properties. Ideally all the slave nodes in a cluster refer towards the ResourceManager and NameNode(s).

C.B.1 MapReduce Configuration

mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>

  <!--property>
  <name>mapred.job.tracker</name>
  <value>hadoopmaster:54311</value>

  -->

  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
    <description>The runtime framework for executing MapReduce jobs.
      Can be one of local, classic or yarn.
    </description>
  </property>

  <property>
    <name>mapreduce.jobtracker.address</name>
    <value>local</value>
    <description>The host and port that the MapReduce job tracker runs
      at. If "local", then jobs are run in-process as a single map
      and reduce task.
    </description>
  </property>

  <property>
    <name>mapred.task.timeout</name>
```

```
46       <value>18000000</value>
47    </property>
48
49
50 </configuration>
```

C.B.2 HDFS Configuration

hdfs-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
  <configuration>

    <property>
      <name>dfs.datanode.data.dir</name>
      <value>file:///home/ubuntu/hadoop_dfs/data/datanode</value>
      <description>DataNode directory</description>
    </property>

    <property>
      <name>dfs.namenode.name.dir</name>
      <value>file:///home/ubuntu/hadoop_dfs/data/namenode</value>
      <description>NameNode directory for namespace and transaction logs
     storage.</description>
    </property>



    <property>
      <name>dfs.replication</name>
      <value>2</value>
    </property>

    <property>
      <name>dfs.permissions</name>
      <value>false</value>
    </property>


    <property>
      <name>dfs.blocksize</name>
```

```
49        <value>512k</value>
50        <description>
51          The default block size for new files, in bytes.
52          You can use the following suffix (case insensitive):
53          k(kilo), m(mega), g(giga), t(tera), p(peta), e(exa) to specify
      the size (such
54          as 128k, 512m, 1g, etc.),
55          Or provide complete size in bytes (such as 134217728 for 128 MB)
      .
56        </description>
57      </property>
58
59
60      <property>
61        <name>dfs.namenode.fs-limits.min-block-size</name>
62        <value>32768</value>
63        <description>Minimum block size in bytes, enforced by the Namenode
       at create
64          time. This prevents the accidental creation of files with tiny
      block
65          sizes (and thus many blocks), which can degrade
66          performance.
67        </description>
68      </property>
69
70      <!--
71  <property>
72  <name>dfs.namenode.fs-limits.min-block-size</name>
73  <value>100</value>
74  <description>minimum block size of the data</description>
75  </property>
76
77  -->
78
79      <property>
80        <name>dfs.datanode.use.datanode.hostname</name>
81        <value>false</value>
82      </property>
83      <property>
84        <name>dfs.namenode.datanode.registration.ip-hostname-check</name>
85        <value>false</value>
86      </property>
87
88
89
90      <!--
91  <property>
92   <name>dfs.namenode.http-address</name>
93   <value>ec2-52-10-149-153.us-west-2.compute.amazonaws.com:50070</value>
94   <description>Your NameNode hostname for http access.</description>
95  </property>
96
97  <property>
98   <name>dfs.namenode.secondary.http-address</name>
```

```
99   <value>ec2-52-10-199-242.us-west-2.compute.amazonaws.com:50090</value>
100  <description>Your Secondary NameNode hostname for http access.</
     description>
101 </property>
102 -->
103
104      <property>
105         <name>dfs.namenode.rpc-address</name>
106         <value>hadoopmaster:9000</value>
107         <description>
108           RPC address that handles all clients requests. In the case of HA
     /Federation where multiple namenodes exist,
109           the name service id is added to the name e.g. dfs.namenode.rpc-
     address.ns1
110           dfs.namenode.rpc-address.EXAMPLENAMESERVICE
111           The value of this property will take the form of nn-host1:rpc-
     port.
112         </description>
113      </property>
114
115   </configuration>
```

C.B.3 Core Site Configuration

core-site.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
3 <!--
4   Licensed under the Apache License, Version 2.0 (the "License");
5   you may not use this file except in compliance with the License.
6   You may obtain a copy of the License at
7
8     http://www.apache.org/licenses/LICENSE-2.0
9
10  Unless required by applicable law or agreed to in writing, software
11  distributed under the License is distributed on an "AS IS" BASIS,
12  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied.
13  See the License for the specific language governing permissions and
14  limitations under the License. See accompanying LICENSE file.
15 -->
16
17 <!-- Put site-specific property overrides in this file. -->
18
19 <configuration>
20   <property>
21     <name>fs.defaultFS</name>
22     <value>hdfs://hadoopmaster:9000</value>
23     <description>Namenode URI</description>
24   </property>
25 </configuration>
```

C.B.4 Apache Hadoop Yarn Configuration

yarn-site.xml

```
<?xml version="1.0"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->
<configuration>

  <!-- Site specific YARN configuration properties -->


  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.resourcemanager.resource-tracker.address</name>
    <value>hadoopmaster:8025</value>
  </property>
  <property>
    <name>yarn.resourcemanager.scheduler.address</name>
    <value>hadoopmaster:8030</value>
  </property>
  <property>
    <name>yarn.resourcemanager.address</name>
    <value>hadoopmaster:8050</value>
  </property>

  <property>
    <name>yarn.nodemanager.pmem-check-enabled</name>
    <value>false</value>
  </property>

  <property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
  </property>
```

```
50
51
52   <property>
53     <description>The hostname of the RM.</description>
54     <name>yarn.resourcemanager.hostname</name>
55     <value>hadoopmaster</value>
56   </property>
57
58   <property>
59     <description>Whether physical memory limits will be enforced for
60       containers.
61     </description>
62     <name>yarn.nodemanager.pmem-check-enabled</name>
63     <value>false</value>
64   </property>
65
66   <property>
67     <description>Whether virtual memory limits will be enforced for
68       containers.
69     </description>
70     <name>yarn.nodemanager.vmem-check-enabled</name>
71     <value>false</value>
72   </property>
73
74   <property>
75     <description>Whether to enable log aggregation. Log aggregation
    collects
76       each container's logs and moves these logs onto a file-system, for
     e.g.
77       HDFS, after the application completes. Users can configure the
78       "yarn.nodemanager.remote-app-log-dir" and
79       "yarn.nodemanager.remote-app-log-dir-suffix" properties to
    determine
80       where these logs are moved to. Users can access the logs via the
81       Application Timeline Server.
82     </description>
83     <name>yarn.log-aggregation-enable</name>
84     <value>true</value>
85   </property>
86
87 </configuration>
```

LIST OF ABBREVIATIONS

AWS	Amazon Web Services
CC	Cluster Controller
CGI	Common Gateway Interface
CPU	Central Processing Unit
CRUD	Create, Read, Update, Repeat
DDP	Distributed Data-Parallelization
DES	Data Encryption Standard
DHCP	Dynamic Host Control Protocol
DOM	Document Object Model
DN	Data Node
EBS	Elastic Block Storage Controller
EC2	Elastic Compute Cloud
GB	Gigabyte
Gbps	Gigabits per second
GFS	Google File System
GiB	Gigabyte
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HDD	Hard Disk Drive

HDFS	Hadoop Distributed File System
HPCC	High-Performance Computing Center
HTML	Hypertext Markup Language
I/O	Input/Output
ILP	Instruction Level Parallelism
JSON	JavaScript Object Notation
LAN	Local Area Network
LTS	Long Term Support
MB/s	Megabytes per second
Mbps	Megabits per second
MCR	Matlab Compiler Runtime
MySQL	My Structured Qurey Language
NN	Name Node
OS	Operating System
PBMC	Human Peripheral Blood Mononuclear Cells
PEP	Posterior Error Probabilities
PSM	Paralogous Sequence Mismatches
RAID	Redundant Array of Independent Disks
RM	Resource Manager
RMI	Remote Method Invocation
RPC	Remote Procedural call
SVM	Support Vector Machine

US	United States
VM	Virtual Machine
YARN	Yet Another Resource Negotiator