

STUDY OF SIMILARITY COEFFICIENTS USING MAPREDUCE
PROGRAMMING MODEL

A Paper
Submitted to the Graduate Faculty of the
North Dakota State University
of Agriculture and Applied Science

By

GhanaShyam Nath Nayakam

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

December 2012

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Study Of Similarity Coefficients Using MapReduce
Programming Model

By

GhanaShyam Nath Nayakam

The Supervisory Committee certifies that this *disquisition*
complies with North Dakota State University's regulations and
meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig

Chair

Dr. Kendall Nygard

Dr. Jun Kong

Dr. Mahmud Zakaria

Approved:

12/18/2012

Date

Dr. Brian Slator

Department Chair

ABSTRACT

MapReduce is a programming model for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

Similarity metric is the basic measurement used by a number of data mining algorithms. It is used to measure similarity between data objects. These objects may have one or more than one attributes related to them.

In this paper, for a given input data of users and their page entity pairs we calculate the similarity index between the users with respect to the page edits. We consider four different algorithms for the calculation of similarity coefficients. They are Jaccard, Cosine, Tanimoto and Dice's coefficient. We implement these algorithms using MapReduce Programming structure, and study their behavior with respect to different input sizes and cluster sizes.

ACKNOWLEDGEMENTS

I would take this opportunity to thank my advisor, Dr. Simone Ludwig, who has given me valuable support, encouragement and advice without which this work would not have been completed. I am thankful to the members of the committee, Dr. Kendall Nygard, Dr. Jun Kong and Dr. Mahmud Zakaria, for their support. I would also like to thank my parents for their valuable support and constant encouragement.

TABLE OF CONTENTS

ABSTRACT.....	iii
ACKNOWLEDGEMENTS.....	iv
LIST OF TABLES	vii
LIST OF FIGURES.....	viii
1. INTRODUCTION.....	1
2. RELATED WORK.....	5
3. APPROACH	10
3.1. MapReduce	10
3.2. Implementation.....	18
3.3. Algorithms	23
3.3.1. Jaccard Similarity Coefficient	23
3.3.1.1. Module One.....	23
3.3.1.2. Module Two	24
3.3.1.3. Module Three	24
3.3.2. Cosine Similarity Coefficient	25
3.3.2.1. Module Three	26
3.3.3. Dice’s Similarity Coefficient	26
3.3.3.1. Module Three	27
3.3.4. Tanimoto Similarity Coefficient	27

3.3.4.1. Module Three	28
4. EXPERIMENTS AND RESULTS	29
4.1. Experiment One.....	29
4.2. Experiment Two.....	30
4.3. Experiment Three	31
4.4. Experiment Four.....	33
4.5. Experiment Five	34
4.6. Experiment Six.....	35
4.7. Experiment Seven.....	36
4.8. Experiment Eight.....	37
4.9. Experiment Nine.....	38
5. CONCLUSION AND FUTURE WORK	41
REFERENCES.....	43

LIST OF TABLES

<u>Table</u>	<u>Page</u>
4.1. Average Time Taken vs. Input Sizes for Experiment One	30
4.2. Average Time Taken vs. Input Sizes for Experiment Two	31
4.3. Average Time Taken vs. Input Sizes for Experiment Three.....	32
4.4. Average Time Taken vs. Input Sizes for Experiment Four	33
4.5. Average Time Taken vs. Cluster Sizes for Experiment Five.....	34
4.6. Average Time Taken vs. Cluster Sizes for Experiment Six	36
4.7. Average Time Taken vs. Cluster Sizes for Experiment Seven	36
4.8. Average Time Taken vs. Cluster Sizes for Experiment Eight	38
4.9. Average Time Taken vs. Cluster Sizes for Experiment Nine	39

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
3.1. Basic MapReduce Representation	10
3.2. Execution Overview	13
3.3. Hadoop Daemons in a Cluster.....	21
4.1. Experiment One: Average Execution Time vs. Input Sizes for Cluster Size of Five.....	29
4.2. Experiment Two: Average Execution Time vs. Input sizes for Cluster Size of Ten.....	31
4.3. Experiment Three: Average Execution Time vs. Input Sizes for Cluster Size of Fifteen.....	32
4.4. Experiment Four: Average Execution Time vs. Input sizes for Cluster Size of Twenty.....	33
4.5. Experiment Five: Average Execution Time vs. Cluster Sizes for Input Size of 100 .	34
4.6. Experiment Six: Average Execution Time vs. Cluster Sizes for Input Size of 1,000	35
4.7. Experiment Seven: Average Execution Time vs. Cluster Sizes for Input Size of 10,000	37
4.8. Experiment Eight: Average Execution Time vs. Cluster Sizes for Input Size of 100,000	38
4.9. Experiment Nine: Average Execution Time vs. Cluster sizes for Input Size of 1,000,000	39

1. INTRODUCTION

Similarity metric is the basic measurement used by a number of data mining algorithms. It is used to measure similarity or dissimilarity between two data objects. These objects may have one or more than one attributes related to them. In other words similarity is a numerical measure of the degree to which the two objects are alike. This measure is usually non-negative and often between 0 and 1, where 0 means both the objects are mutually exclusive or they have no similarity, and 1 means both objects are completely similar.

There are different data types with a number of different attributes associated with them. It is important to use the appropriate similarity metric to measure the similarity between two objects. For example, Euclidean distance and Pearson's correlation coefficient are useful for measuring the similarity between dense data such as time series or two dimensional points. Jaccard index and Cosine index are used to measure the similarity between sparse data like documents, or binary data. In this study we use Jaccard index, Cosine Index, Tanimoto Coefficient and Dice's coefficient are used because of the nature of the input data.

When dealing with data objects that have binary attributes, it is more effective to calculate similarity using a Jaccard Coefficient. The Jaccard Coefficient measures the similarity between the sample sets, and is defined by the size of intersection between the two sets divided by the size of union of the two sets. In other words, Jaccard index J can be written as:

$$J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}.$$

The M_{11} represents the total number of attributes where both data objects have a 1. The M_{10} and M_{01} represent the total number of attributes where one data object has a 1 and the other has a 0. In other words M_{11} represents the data items present in both the sample sets. M_{10} represents the data items present in the first object but not in second object. M_{01} represents the vice versa, i.e., the data items present in the second object but not in the first object. A Jaccard index of 1 tells us that both data objects are completely similar, whereas a Jaccard index of 0 says they are completely different and mutually exclusive.

Cosine similarity is a measure of similarity between two vectors by measuring the cosine of the angle between them. This method is often used in text mining, it is also used to measure cohesion within clusters in the field of data mining. Cosine similarity uses the Vector form of the data objects to find the similarity. Similarity is then measured as the angle between them.

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}.$$

In the above equation, A represents the vector form of the first data object and B represents the vector form of the second data object. This method is useful when finding the similarity between two text documents whose attributes are word frequencies. If the two documents have a perfect similarity, the Cosine Coefficient will be 1 (or an angle 0 degree), and 0 otherwise (or an angle of 90 degree).

The Tanimoto coefficient can be called as an extended version of the Jaccard Coefficient and Cosine Similarity Coefficient. Like Cosine Similarity Coefficient, it

uses the vector form of the data objects to find their similarity index. The Tanimoto index can be represented as:

$$T(A, B) = \frac{A \cdot B}{\|A\|^2 + \|B\|^2 - A \cdot B}.$$

In the equation, A and B are data objects represented by their corresponding vectors. The similarity score is the dot product of A and B divided by the squared magnitudes of A and B minus the dot product.

In this paper, we perform a comparative study between different similarity coefficients, on authorship data from Wikipedia. Due to the large scale nature of the data and computationally intensive task at hand, we decided to counter this problem by using the MapReduce framework on a Hadoop Cluster.

MapReduce is a software framework introduced by Google to support distributed computing on large datasets on clusters of computers. This framework is inspired by the *Map* and *Reduce* functions commonly used in functional programming, although their purpose in the MapReduce framework is not same as their original forms.

Hadoop is a software framework that supports data-intensive distributed applications under a free license. It enables applications to work with thousands of nodes and petabytes of data. The Hadoop framework transparently provides applications both reliability and data motion. In addition, it provides a distributed file system (HDFS) that stores data on compute nodes, providing very high aggregate bandwidth across the cluster. Both MapReduce and Hadoop Distributed File System are designed so that node failures are automatically handled by the framework.

In this paper, for the input data of users and their page entity pairs we calculate the similarity index between the users with respect to the page edits. We consider four different algorithms for the calculation of similarity coefficients between the input dataset. They are Jaccard coefficient, Cosine coefficient, Tanimoto coefficient and Dice's coefficient. We study the behavior of these four algorithms on a range of cluster sizes for different input sizes.

The remainder of the paper is organized as follows. Section 2 describes the related work done in this area. In Section 3, the similarity coefficients are explained and their implementation using MapReduce are described. In Section 4, the results are shown in graphical format and the tables supporting the graphs are presented in the appendix. Section 5 concludes the paper and also talks about the future directions this research can take.

2. RELATED WORK

Since MapReduce was proposed by Google as a programming model for developing distributed data intensive applications in data centers, it has received much attention from various fields and industries. Many projects are exploring ways to support MapReduce on various types of distributed architecture and for a wider range of applications. Below are some instances which have used this programming model to support their studies.

In [1], the authors compare two types of similarity coefficients, the Jaccard's coefficient and the production volume based coefficient. Each is used to form a cellular manufacturing system whose performance was used as a measure of effectiveness based on the similarity coefficient. The sum of intercellular and intracellular material handling costs was used as a criterion for performance evaluation. A major step in the development of a cellular manufacturing system is identification of part families and formation of associated machine cells. Among all the approaches to machine cell formation, the similarity coefficient method is more effective because of its capability in forming machine cells in the presence of exceptional parts (parts being processed in more than one machine cell), and its flexibility in incorporating production volume, sequences of operations, and operational times into the machine cell formation process.

In [2], the authors compare two commonly used distance measures in vector models, namely, Euclidean distance and Cosine angle distance, for Nearest Neighbor queries in high dimensional data spaces. Using theoretical analysis and experimental results, the authors show that the retrieval results based on Euclidean distance are

similar to those based on the Cosine angle distance when the dimension is high. The authors apply the Cosine angle distance for content based image retrieval (CBIR). Retrieval results show that Cosine angle distance works no worse than Euclidean distance, which is a commonly used distance measure for CBIR. The authors compare these two distance measures experimentally using normalized datasets and clustered datasets.

In [3], the authors assess the variations caused by three commonly used similarity coefficients including Jaccard, Sorensen-Dice and simple matching in the clustering and ordination of seven Iranian native silkworm strains analyzed by amplified fragment length polymorphism markers. Comparisons among the similarity coefficients were made using the Spearman correlation analysis, dendrogram evaluation projection efficiency in a two-dimensional space, and groups formed by the Tocher optimization procedure. Their results demonstrated that for almost all methodologies, the Jaccard and Sorensen-Dice coefficients revealed extremely close results, because both of them exclude negative co-occurrences. The use of coefficients such as Jaccard and Sorensen-Dice that do not include negative co-occurrences was imperative for closely related organisms.

In [4], the authors give a comprehensive overview and discussion for similarity coefficients used in solving the cell formation problem. The authors explain the reason explicitly why the similarity coefficient method is more flexible than other cell formation methods. The authors also propose a taxonomy which is combined by two distinct dimensions. The first dimension is the general-purpose similarity coefficients

and the second is the problem-oriented similarity coefficients. The difference between the two dimensions is discussed through three similarity coefficients.

In [5], the author presents a review of the literature on similarity coefficients between machines in Cellular Manufacturing System, to propose a new similarity coefficient between machines incorporating all important properties of similarity. The author also proposes a machine cell heuristic approach to group machines into machine cells.

In [6], the authors introduce the Tanimoto based similarity measure for host-based intrusions using binary feature set for training and classification. The k-nearest neighbor classifier was used to classify a given process as either normal or attack. The experimentation was conducted on the DARPA-1998 database for intrusion detection and was compared with other existing techniques. It was also observed that these schemes using Tanimoto based similarity measure produced better results than other techniques.

In [7], the author describes an algorithm which identifies dense clusters where similarity within each cluster reflects the Tanimoto value used for the clustering, and where the cluster centroid will be at least similar, at the given Tanimoto value, to every other molecule within the cluster in a consistent and automated manner. The similarity term was used throughout the paper to reflect the overall similarity between two given molecules, as defined by Daylight's fingerprints and the Tanimoto similarity index.

In [8], for a given large collection of sparse vector data in a high dimensional space, the authors investigate the problem of finding all pairs of vectors whose

similarity score (as determined by a function such as cosine distance) is above a given threshold. The authors propose a simple algorithm based on novel indexing and optimization strategies that solve this problem without relying on approximation methods or extensive parameter tuning. The authors support this approach by showing how it efficiently handles a variety of datasets across a wide setting of similarity thresholds, with large speedups over previous state-of-the-art approaches.

The authors in [8], show that a parsimonious indexing approach combined with several other subtle yet simple optimizations yield dramatic performance improvements. They validate their algorithms on a dataset comprised of publicly available data from the DBLP server, and on two real-world web applications, generating recommendations for the Orkut social network, and computing pairs of similar queries among the 5 million most frequently issued Google queries represented as vectors generated from their search result snippets.

In [9], the authors study different similarity coefficients that are applied in the context of a program spectral approach to software fault localization (single programming mistakes). The coefficients studied are taken from the systems diagnosis / automated debugging tools Pinpoint (Jaccard), Tarantula, and Ample, and from the molecular biology domain (the Ochiai coefficient). The authors evaluate these coefficients on the Siemens Suite of benchmark faults, and assess their effectiveness in terms of the position of the actual fault in the probability ranking of fault candidates produced by the diagnosis technique.

In [10], the authors compare the genetic similarity coefficients (Jaccard, Dice and Simple Matchmaking) and different clustering method combinations for cultivated

olives. The authors use 12 samples of olives which were screened with RAPD-PCR (Randomly Amplified Polymorphic DNA-Polymerase Chain Reaction). The authors show that results using Jaccard and Dice similarity coefficients were high and significant because of the high correlation value they show.

3. APPROACH

In this chapter, we will see different stages in MapReduce programming model, how it works, how it overcomes other programming models out there and how we have implemented it in our study. We will also see the algorithms that have been used in this study for the calculation of Jaccard Index, Dice's coefficient, Cosine similarity coefficient and Tanimoto Coefficient.

3.1. MapReduce

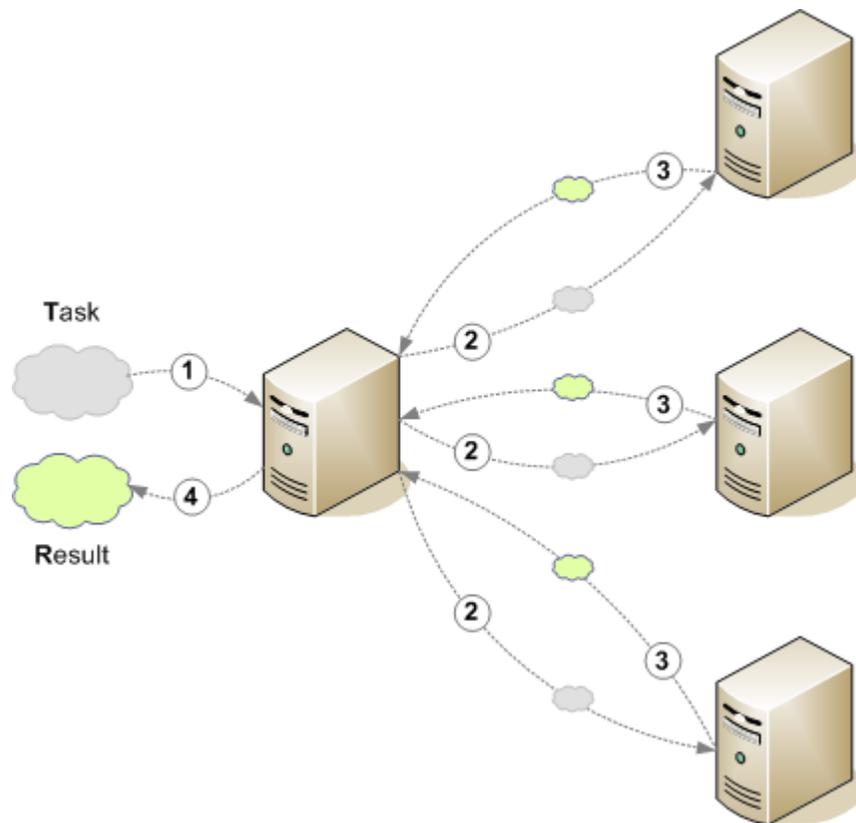


Figure 3.1. Basic MapReduce Representation [11]

MapReduce is a programming framework popularized by Google and used to simplify data processing across massive data sets. As people rapidly increase their online activity and digital footprint, organizations are finding it vital to quickly analyze the huge amounts of data their customers and audiences generate to better understand and serve them. MapReduce is the tool that is helping those organizations.

Most enterprises deal with multiple types of data (text, rich text, RDBMS, graph, etc.), and need to process all this data quickly and efficiently to derive meaningful insights that bring business value to the organization.

With MapReduce, computational processing can occur on data stored either in a file system or within a database. There are two fundamental pieces of a MapReduce process:

Map step: The master node takes the input, chops it up into smaller sub-problems, and distributes those to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes that smaller problem, and passes the answer back to its master node.

Reduce step: The master node then takes the answers to all the sub-problems and combines them in a way to get the output - the answer to the problem it was originally trying to solve.

Logically, the Map and Reduce functions of MapReduce are both defined with respect to data structured in (key, value) pairs. Map takes one pair of data with a type in one data domain and returns a list of pairs in a different data domain.

$$\text{Map}(k1, v1) \rightarrow \text{list}(k2, v2)$$

The Map function is applied in parallel to every item in the input dataset. This produces a list of $(k2, v2)$ pairs for each call. Then, the MapReduce framework collects all pairs with the same key from all lists and groups them together, thus creating one group for each one of the different generated keys.

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

$$\text{Reduce}(k2, \text{list}(v2)) \rightarrow \text{list}(v3)$$

Each Reduce call produces either one value $v3$ or an empty return, though one call is allowed to return more than one value. The returns of all calls are collected as the desired return list.

MapReduce achieves reliability by parceling out a number of operations on the set of data to each node in the network. Each node is expected to report back periodically with completed work and status updates. If a node falls silent for longer than that interval, the master node records the node as dead and sends out the node's assigned work to other nodes. Individual operations use atomic operations for naming file outputs as a check to ensure that there are no parallel conflicting threads running. When files are renamed, it is possible to also copy them to another name in addition to the name of the task.

The reduce operations operate much the same way. Because of their inferior properties with regard to parallel operations, the master node attempts to schedule reduce operations on the same node, or in the same rack as the node holding the data being operated on. This property is desirable as it conserves bandwidth across the backbone network of the data center.

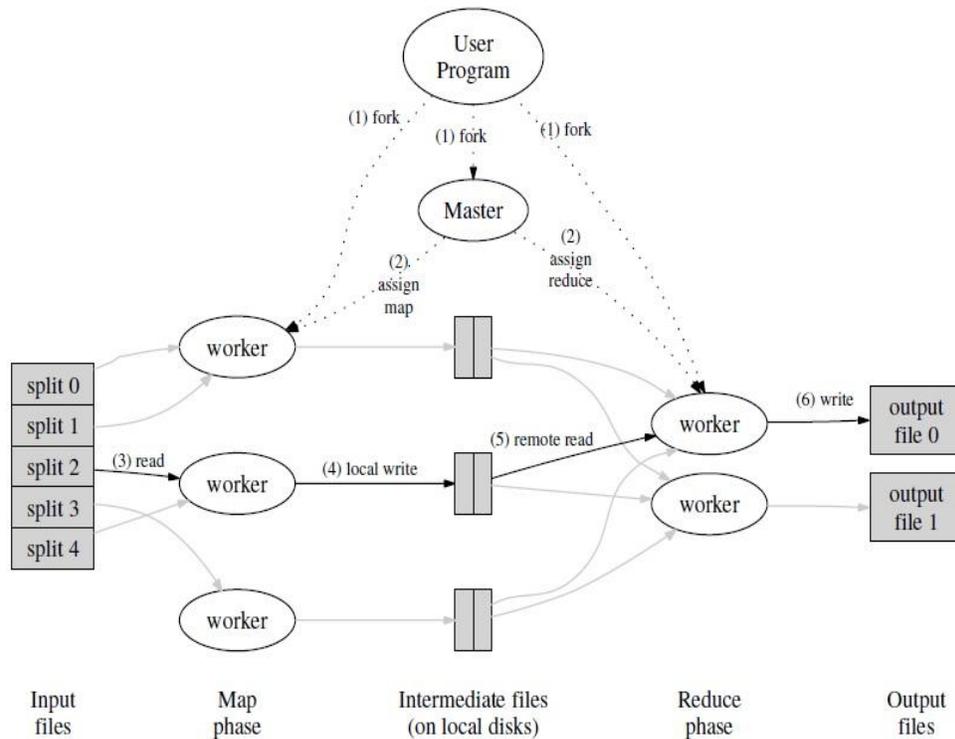


Figure 3.2. Execution Overview [12]

Figure 1 shows the overall flow of a Map Reduce operation in our implementation. When the user program calls the Map Reduce function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The Map Reduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special, the master. The rest are workers that are assigned work by the master. There are M Map tasks and R Reduce tasks to

assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the Map Reduce call in the user program returns back to the user code.

After successful completion, the output of the Map Reduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file, they often pass these files as input to another Map Reduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

The master keeps several data structures. For each map task and Reduce task, it stores the state and the identity of the worker machine. The master is the medium through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

Also, it is entirely possible that the Master node can fail too. In such case it is advised to make the master write periodic checkpoints of the master data structures. If the master task dies, a new copy can be started from the last checkpoint state. However, given that there is only a single Master, its failure is unlikely. Therefore our current implementation aborts the Map Reduce computation if the master fails. Clients can check for this condition and retry the Map Reduce operation if it is needed.

Let us look at a simple example. Assuming that the input file contains two columns that represent city and the corresponding temperature recorded in that city for every two days per week. In this example the input might look like below.

New York, 22
Washington, 32
New York, 18
Washington, 34
New York, 25

Washington, 28

Out of all the data collected, we want to find the maximum temperature for each city. Using MapReduce framework, let us break this down into two Map tasks, where each Map task works on a part of input data (split) and returns the city/temperature pair for every entry in its corresponding split. For example, the results produced from the map tasks for the data above would look like below.

Map 1:

(New York, 22)

(Washington, 32)

(New York, 18)

Map 2:

(Washington, 34)

(New York, 25)

(Washington, 28)

These intermediate pairs would then be fed into Reduce tasks, which combine the results from the Map tasks and output a single value for each city, producing the final result pair as below.

Reduce:

(New York, 25)

(Washington, 32)

These will be written into a file as output for the original problem.

The Map Reduce programming model has been successfully used at Google for many different purposes. The main advantage with this programming model is that a

large variety of problems are easily expressible as Map Reduce computations and implementation of Map Reduce can be scaled to large clusters comprising of thousands of machines.

3.2. Implementation

For this study we have used Hadoop [13]. Hadoop is Apache's free and open source implementation of MapReduce. Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner.

To configure the Hadoop cluster we have to configure the environment in which the Hadoop daemons execute as well as the configuration parameters for the Hadoop daemons. The Hadoop daemons are NameNode, SecondaryNameNode, DataNode, JobTracker and TaskTracker. The function of each Hadoop daemon is explained below.

NameNode serves as both directory namespace manager and *inode* [14] table for the Hadoop DFS. There is a single NameNode running in any DFS deployment. The NameNode controls two critical tables:

1. Filename \rightarrow blocksequence (namespace)
2. Block \rightarrow machinelist (inodes)

The first table is stored on disk and is very precious. The second table is rebuilt every time the NameNode comes up. NameNode refers to both this class as well as the NameNode server. The 'FSNamesystem' class actually performs most of the filesystem management. The majority of the NameNode class itself is concerned with

exposing the IPC interface to the outside world, plus some configuration management. NameNode implements the Client protocol interface, which allows clients to ask for DFS services. ClientProtocol is not designed for direct use by authors of DFS client code. NameNode also implements the Datanode protocol interface, used by DataNode programs that actually store DFS data blocks. These methods are invoked repeatedly and automatically by all the DataNodes in a DFS deployment. NameNode also implements the Namenode Protocol interface, used by secondary namenodes or rebalancing processes to get partial namenode's state.

The Secondary NameNode is a helper to the NameNode. The Secondary NameNode is responsible for supporting periodic checkpoints of the HDFS metadata. The current design allows only one Secondary NameNode per HDFS cluster. The Secondary NameNode is a daemon that periodically wakes up, triggers a periodic checkpoint and then goes back to sleep. The Secondary NameNode uses the Client Protocol to talk to the primary NameNode.

DataNode stores a set of blocks for a DFS deployment. A single deployment can have one or many DataNodes. Each DataNode communicates regularly with a single NameNode. It also communicates with client code and other DataNodes from time to time. DataNodes store a series of named blocks. The DataNode allows the client code to read these blocks, or to write new block data. The DataNode may also, in response to instructions from its NameNode, delete blocks or copy blocks to/from other DataNodes. The DataNode maintains just one critical table: block → stream of bytes. This info is stored on a local disk. The DataNode reports the table's contents to the NameNode upon startup and every so often afterwards. DataNodes spend their lives

in an endless loop of asking the NameNode for something to do. A NameNode cannot connect to a DataNode directly; a NameNode simply returns values from functions invoked by a DataNode. DataNodes maintain an open server socket so that client code or other DataNodes can read/write data. The host/port for this server is reported to the NameNode, which then sends that information to clients or other DataNodes that might be interested.

The JobTracker is the service within Hadoop that farms out Map Reduce tasks to specific nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack. The Client applications submit jobs to the JobTracker. The JobTracker then talks to the NameNode to determine the location of the data. The Jobtracker locates TaskTracker nodes with available slots at or near the data. The Task Tracker nodes are monitored. If they do not submit their signals often enough, they are marked to have failed and the work is scheduled on a different TaskTracker. When the work is completed the Job tracker updates its status. JobTracker is the point of failure for most Hadoop Map Reduce service. If it goes down, all the running jobs are halted.

A TaskTracker is a node in the cluster that accepts tasks from a JobTracker. Every TaskTracker is configured with a set of slots, these indicate the number of tasks that it can accept. When the JobTracker tries to find somewhere to schedule a task within the MapReduce operations, it first looks for an empty slot on the same server that hosts the DataNode containing the data, and if not, it looks for an empty slot on a machine in the same rack.

The TaskTracker spawns a separate JVM process to do the actual work. This is to ensure that process failure does not take down the task tracker. The TaskTracker monitors these spawned processes, capturing the output and exit codes. When the process finishes, successfully or not, the tracker notifies the JobTracker. The TaskTrackers also send out status messages to the JobTracker, usually every few minutes, to reassure that it is still alive. These messages also inform the JobTracker of the number of available slots, so the JobTracker can stay up to date with where in the cluster work can be delegated. Figure 2 shows how all Hadoop daemons work in a cluster.

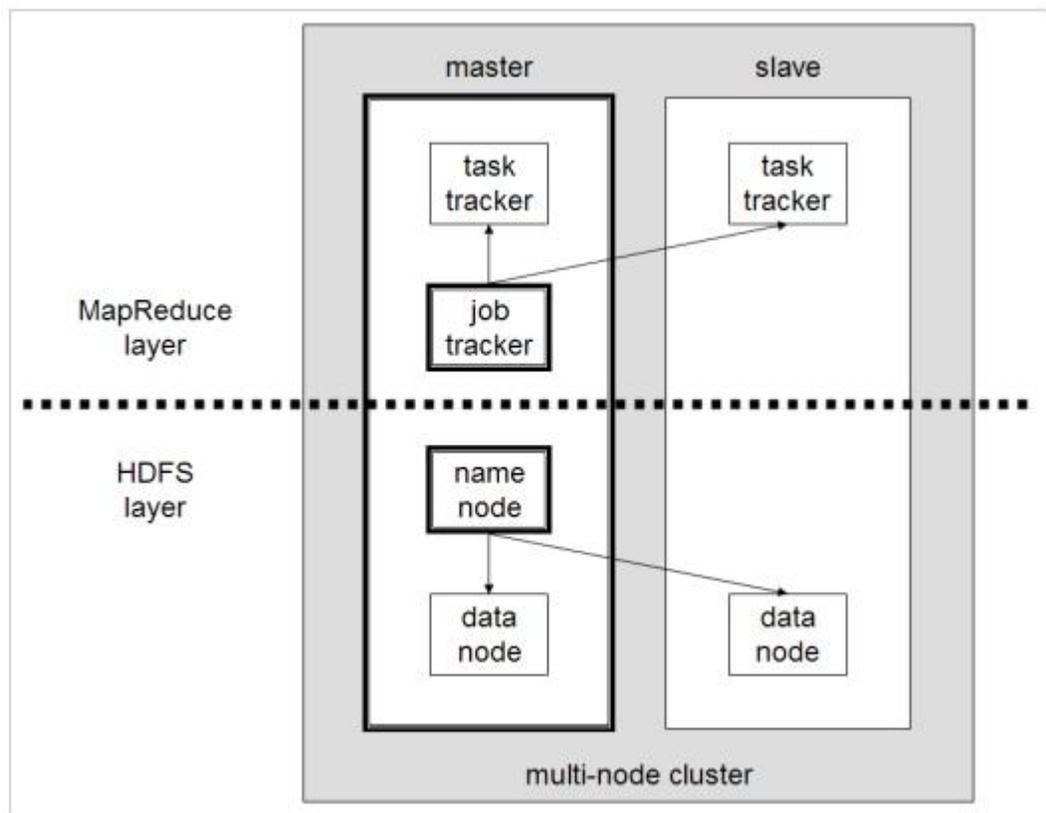


Figure 3.3. Hadoop Daemons in a Cluster [15]

A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master, 'JobTracker', and one slave, 'TaskTracker', per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

Applications specify the input output locations and supply map and reduce functions implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the 'Job Configuration'. The Hadoop 'Job Client' then submits the job (jar/executable, etc.) and configuration to the JobTracker which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job client. In this study, we have used the Hadoop framework in Java running on a Linux platform.

3.3. Algorithms

3.3.1. Jaccard Similarity Coefficient

To compute the Jaccard similarity coefficient for a pair of X elements, the calculation of two quantities is needed. The first is the number of Y elements that occur with both X elements, and the second is the number of Y elements that occur with one or both. Then, the first quantity is divided by the second quantity to calculate the coefficient. We break the above mentioned logic into three Map and Reduce modules where each quantity is calculated, and finally ending up calculating the similarity coefficient. These modules are reused for calculation of other similarity coefficients as well.

3.3.1.1. Module One

XY Identity Map

This Map takes in a key/value pair and returns them unchanged.

Input: $\underline{Text}, Text$

Output: $\underline{Text}, Text$

$\underline{X}, Y \rightarrow \underline{X}, Y$

Count Y Reduce

This Reduce method takes a key and the set of all corresponding values, and computes the cardinality of the set of values, C , and returns the input key with each input value and C . This quantity, C , is eventually used to compute the denominator of the Jaccard similarity coefficient.

Input: $\underline{Text}, \{Text\}$

Output: $\underline{Text}, CPair$

$\underline{X}, \{Y\} \rightarrow \underline{X}, (Y, C)$ where $C = \{\text{input values}\}$

3.3.1.2. **Module Two**

XY to YX Map

This map method takes the key/value pair and switches the String in the key with the String in the value. It then returns the modified key and value. Using the Y element as the key allows the combine stage to collect all of the X elements that correspond to each Y element. This enables the later collection of all pairs of X that occur with a single Y .

Input: $\underline{Text}, CPair$

Output: $\underline{Text}, CPair$

$\underline{X}, (Y, C) \rightarrow \underline{Y}, (X, C)$

Split Reduce

This reduce task takes in a key and set of values, splits up the set of values, and outputs an empty key with each of the smaller lists of values. This is done to load balance the collection of pairs in the next map method.

Input: $\underline{Text}, \{CPair\}$

Output: $\underline{Text}, \{CPair\}$

$\underline{Y}, \{(X, C)\} \rightarrow \underline{R}, \{(X, C)\}$ where R is an empty placeholder

3.3.1.3. **Module Three**

Pair Collect Map

This map takes in the empty Text key and the list of values. It then creates a series of X_i, X_j pairs with the first element in the value list and each of the following

elements. A list of size n will produce $n-1$ such pairs, one for each element in the list, except for the first.

Input: Text, {CPair}

Output: Pair, IntWritable

\underline{R} , $\{(X, C)\} \rightarrow (\underline{X_i}, \underline{X_j}), C_{ij}$ where $C_{ij} = C_i + C_j$

Normalization Reduce

This reduce method calculates the number of co-occurrences of X_i and X_j by counting the number of elements in the set of input values. It then divides the number of co-occurrences by the total number of Y elements that occur with one or both of X_i and X_j .

Input: Pair, {IntWritable}

Output: Pair, FloatWritable

$(X_i, X_j), \{C_{ij}\} \rightarrow (X_i, X_j), (|\{C_{ij}\}|/C_{ij} - |\{C_{ij}\}|)$

3.3.2. Cosine Similarity Coefficient

To compute the Cosine similarity coefficient for a pair of X vectors, calculation of two quantities is needed. The first is the dot product of the two vectors, and the second is the product of the magnitude of both vectors. Then, the first quantity is divided by the second quantity to calculate the cosine coefficient. We use the same three module approach from the previous algorithm. We reuse *Module 1* and *Module 2* from the Jaccard algorithm, because the calculation of Cosine Similarity Coefficient is done in *Module 3*, which is described below.

3.3.2.1. Module Three

Pair Collect Map

This map takes in the empty Text key and the list of values. It then creates a series of X_i, X_j pairs with the first element in the value list and each of the following elements. A list of size n will produce $n-1$ such pairs, one for each element in the list, except for the first.

Input: Text, {CPair}

Output: Pair, IntWritable

$R, \{(X, C)\} \rightarrow \langle X_i, X_j \rangle, C_{ij}$ where $C_{ij} = C_i * C_j$

Normalization Reduce

This reduce method calculates the dot product vectors X_i and X_j . It is then divided by the product of magnitude of both vectors.

Input: Pair, {IntWritable}

Output: Pair, FloatWritable

$(X_i, X_j), \{C_{ij}\} \rightarrow (X_i, X_j), (\{C_{ij}\}/C_{ij})$

3.3.3. Dice's Similarity Coefficient

To compute the Dice's similarity coefficient for a pair of X elements, calculation of two quantities is needed. The first is the number of Y elements that occur with both X elements, and the second is the sum of magnitude of vector representation of the sets. To calculate the Dice's coefficients we divide twice the first quantity by the second. We reuse *Module 1* and *Module 2* because the calculation of Dice's coefficients is done in *Module 3* as described below.

3.3.3.1. Module Three

Pair Collect Map

This map takes in the empty Text key and the list of values. It then creates a series of X_i, X_j pairs with the first element in the value list and each of the following elements. A list of size n will produce $n-1$ such pairs, one for each element in the list, except for the first.

Input: Text, {CPair}

Output: Pair, IntWritable

R, {(X, C)} \rightarrow (X_i, X_j), C_{ij} where $C_{ij} = C_i + C_j$

Normalization Reduce

This reduce method calculates the number of co-occurrences of X_i and X_j by counting the number of elements in the set of input values. It then divides the number of co-occurrences by the average of number of Y elements that occur with X_i and X_j .

Input: Pair, {IntWritable}

Output: Pair, FloatWritable

(X_i, X_j), {C_{ij}} \rightarrow (X_i, X_j), $(2*|\{C_{ij}\}|/C_{ij})$

3.3.4. Tanimoto Similarity Coefficient

To compute the Tanimoto similarity coefficient for a pair of X elements, the calculation of two quantities is needed. The first is the number of Y elements that occur with both X elements, and the second is the sum of magnitude of vector representation of the sets. To calculate the Dice's coefficients we divide twice the first quantity by the second. We reuse *Module 1* and *Module 2* because the calculation of Dice's coefficients is done in *Module 3* as described below.

3.3.4.1. Module Three

Pair Collect Map

This map takes in the empty Text key and the list of values. It then creates a series of X_i, X_j pairs with the first element in the value list and each of the following elements. A list of size n will produce $n-1$ such pairs, one for each element in the list, except for the first.

Input: Text, {CPair}

Output: Pair, IntWritable

R, {(X, C)} → (X_i, X_j), C_{ij} where C_{ij} = C_i + C_j

Normalization Reduce

This reduce method calculates the number of co-occurrences of X_i and X_j by counting the number of elements in the set of input values. It then divides the number of co-occurrences by the number of Y elements that occur once within X_i and X_j .

Input: Pair, {IntWritable}

Output: Pair, FloatWritable

(X_i, X_j), {C_{ij}} → (X_i, X_j), (({C_{ij}}/(({|C_i|}+{|C_j|}-{|C_{ij}|})))

4. EXPERIMENTS AND RESULTS

The different parameters for experiments are number of nodes in the cluster (cluster size), the number of lines in the input files (input size), and the time taken for each algorithm (time taken in ms) to calculate the various similarity coefficients. In the following experiments, the time taken by each algorithm for different input sizes and cluster sizes is calculated by the average time taken by the algorithm after repeating it twenty five times.

4.1. Experiment One

In this experiment, the average time taken for each algorithm was calculated keeping the cluster size = 5 nodes, and the input size is varied from 100 to 1,000,000 multiplied by 10. Figure 4.1 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

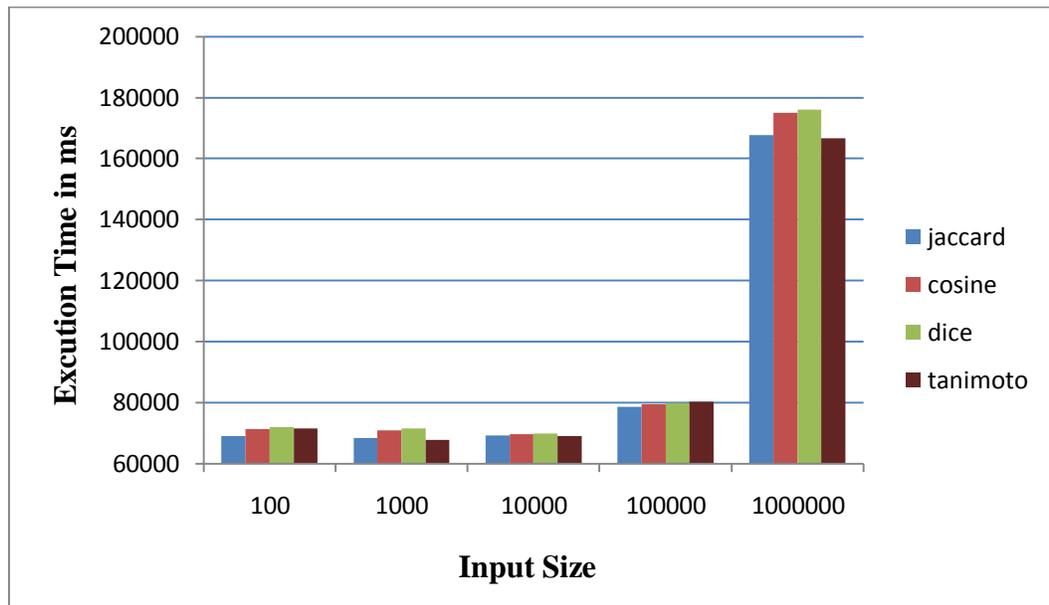


Figure 4.1. Experiment One: Average Execution Time vs. Input Sizes for Cluster Size of Five

Table 4.1. Average Time Taken vs. Input Sizes for Experiment One

Input Size	100	1000	10000	100000	1000000
Jaccard's	69013.3	68387.4	69101.4	78520.8	167752.2
Cosine	71360.8	70844.1	69506.7	79469.2	175055.6
Dice's	71924.0	71418.2	69888.7	79868.2	175990.8
Tanimoto	71440.4	67755.2	69078.1	80354.2	166740.1

It can be seen from Figure 4.1, for input sizes 100 to 10,000 the average execution time is 70,000ms. For, input size of 100,000 the average execution time increases to 79,000ms, and for input size of 1,000,000 the average execution time increases exponentially to 170,000ms.

4.2. Experiment Two

In this experiment, the average time taken for each algorithm was calculated keeping the cluster size = 10 nodes, and the input size is varied from 100 to 1,000,000 multiplied by 10. Figure 4.2 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

It can be seen from Figure 4.2, for input sizes 100 to 1,000 the average execution time is 70,000ms. For input size of 10,000, the average execution time is 75,000ms. For input size of 100,000, the average execution time increases to 80,000ms, and for input size of 1,000,000 the average execution time increases exponentially to 165,000ms.

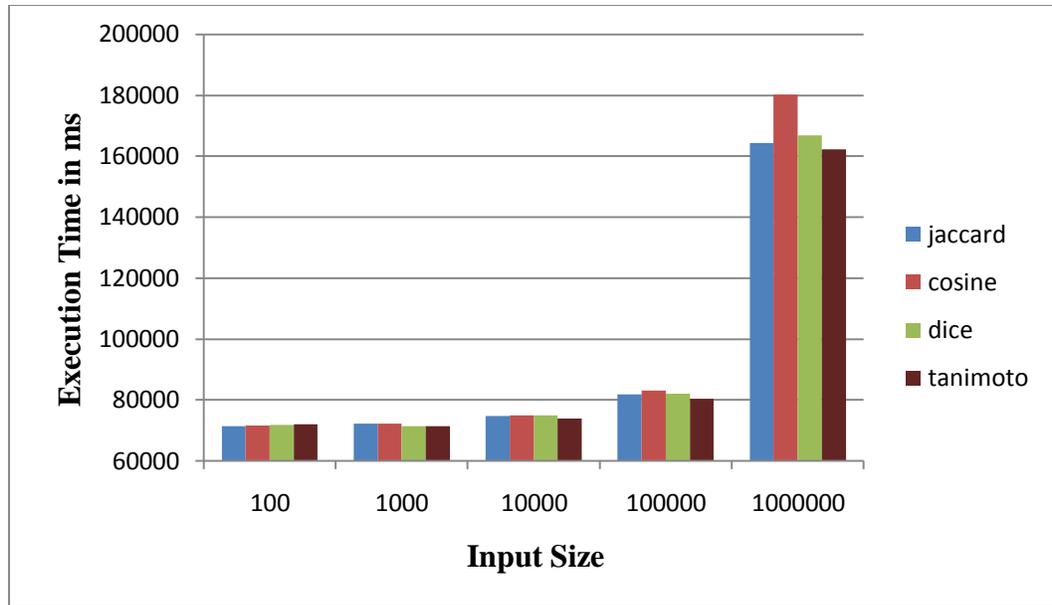


Figure 4.2. Experiment Two: Average Execution Time vs. Input sizes for Cluster Size of Ten

Table 4.2. Average Time Taken vs. Input Sizes for Experiment Two

Input Size	100	1000	10000	100000	1000000
Jaccard's	71359.2	72279.8	74717.1	81847.9	164337.2
Cosine	71522.4	72213.9	74874.8	83046.5	180222.5
Dice's	71829.1	71389.5	75046.4	82058.0	166926.6
Tanimoto	71997.0	71414.0	73820.6	80386.8	162182.1

4.3. Experiment Three

In this experiment, the average time taken for each algorithm was calculated keeping the cluster size = 15 nodes, and the input size is varied from 100 to 1,000,000 multiplied by 10. Figure 4.3 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

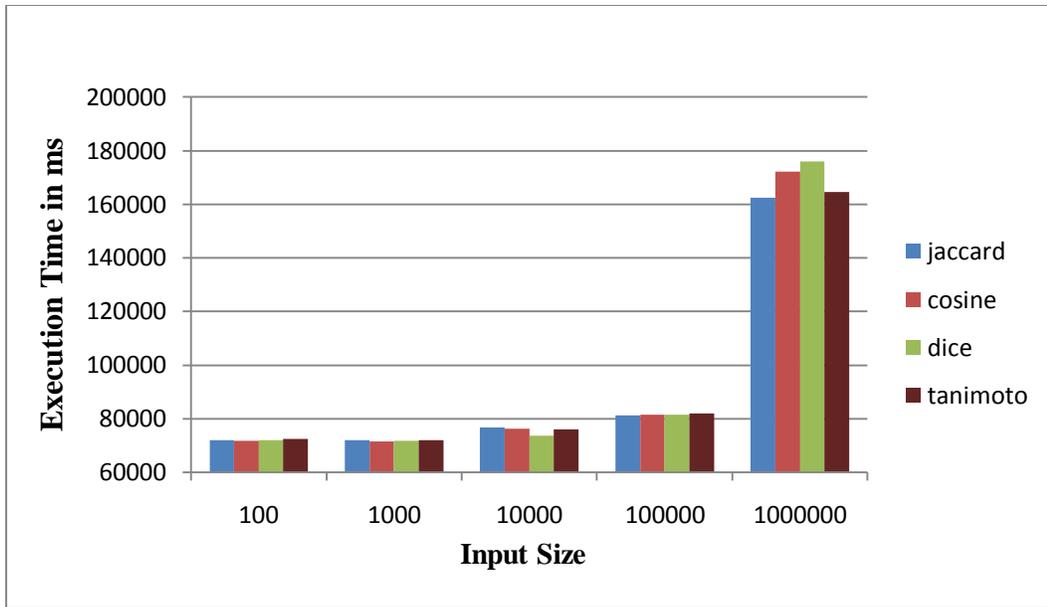


Figure 4.3. Experiment Three: Average Execution Time vs. Input Sizes for Cluster Size of Fifteen

Table 4.3. Average Time Taken vs. Input Sizes for Experiment Three

Input Size	100	1000	10000	100000	1000000
Jaccard's	72046.2	71914.3	76746.6	81196	162370.2
Cosine	71740.0	71443.4	76372.3	81534.6	172311.2
Dice's	72120.2	71862.8	73715.0	81382.8	175908.1
Tanimoto	72418.6	72048.2	75940.0	81880.8	164638.3

It can be seen from Figure 4.3, for input sizes 100 to 1,000 the average execution time is 70,000ms. For input size of 10,000, the average execution time is 75,000ms. For input size of 100,000, the average execution time increases to 80,000ms, and for input size of 1,000,000 the average execution time increases exponentially to 165,000ms.

4.4. Experiment Four

In this experiment, the average time taken for each algorithm was calculated keeping the cluster size = 20 nodes, and the input size is varied from 100 to 1,000,000 multiplied by 10. Figure 4.4 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

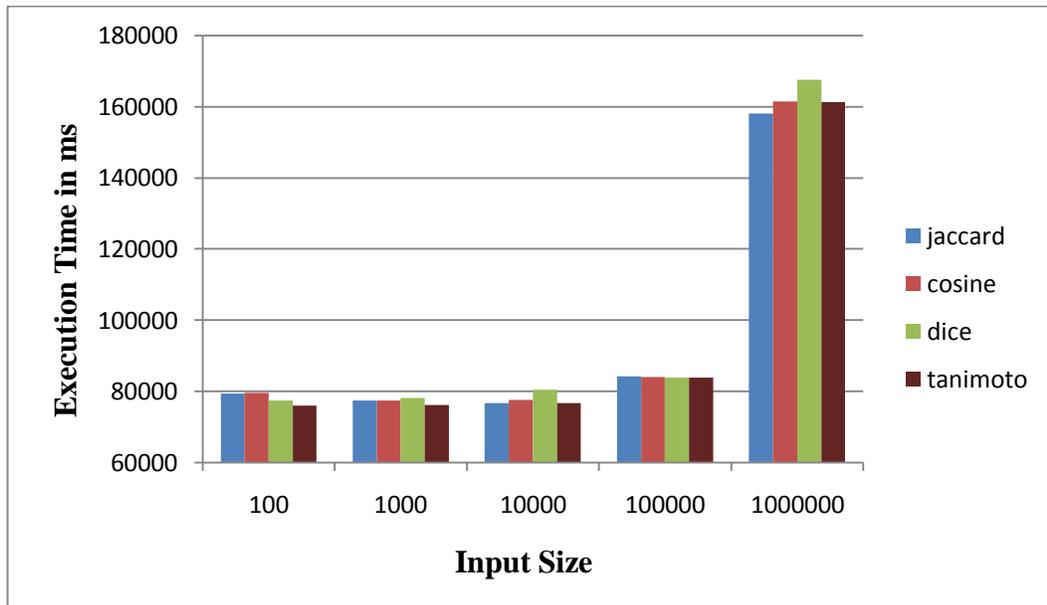


Figure 4.4. Experiment Four: Average Execution Time vs. Input sizes for Cluster Size of Twenty

Table 4.4. Average Time Taken vs. Input Sizes for Experiment Four

Input Size	100	1000	10000	100000	1000000
Jaccard's	79437.1	77400.4	76725.6	84304.5	158107.8
Cosine	79670.1	77398.5	77672.0	84007.8	161519.0
Dice's	77432.8	78112.9	80502.9	83808.5	167610.8
Tanimoto	75955.8	76156.8	76762.9	83831.8	161196.9

It can be seen from Figure 4.4, for input size 100 the average execution time is 80,000ms. For input size of 1,000 and 10,000, the average execution time is 78,000ms. For input size of 100,000, the average execution time increases to

84,000ms, and for input size of 1,000,000 the average execution time increases exponentially to 160,000ms.

4.5. Experiment Five

In this experiment, the average time taken for each algorithm was calculated keeping the input size = 100 lines, and varying the cluster size from 5 nodes to 20 nodes by increments of 5. Figure 4.5 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

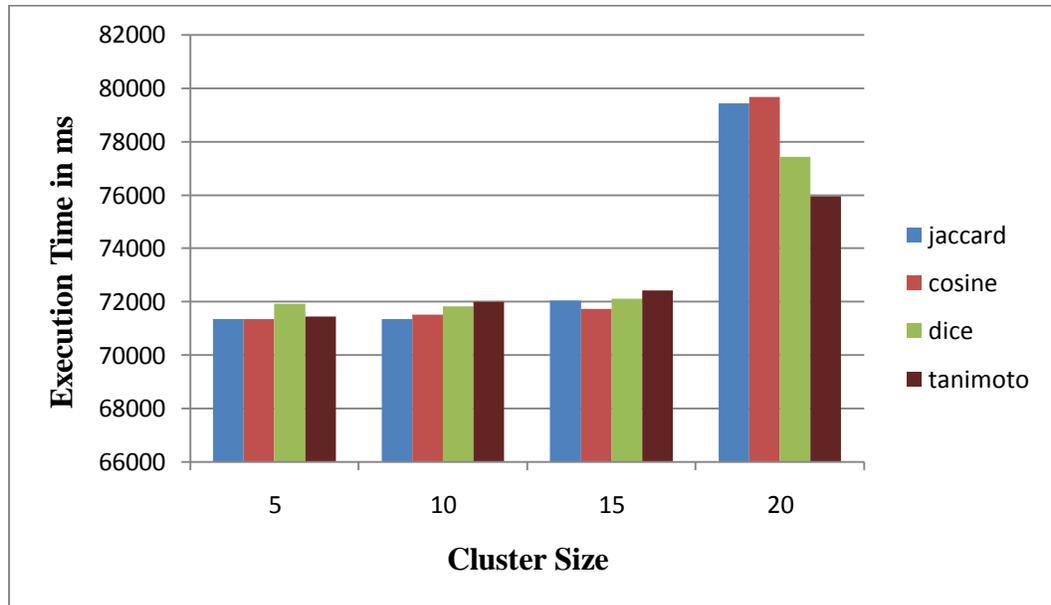


Figure 4.5. Experiment Five: Average Execution Time vs. Cluster Sizes for Input Size of 100

Table 4.5. Average Time Taken vs. Cluster Sizes for Experiment Five

Cluster Size	5	10	15	20
Jaccard's	71342.0	71359.0	72046.2	79437.1
Cosine	71360.8	71522.4	71740.0	79670.1
Dice's	71924.0	71829.1	72120.2	77432.8
Tanimoto	71440.4	71997.0	72418.6	75955.8

It can be seen from Figure 4.5, for input size of 100 the average execution time is 72,000ms for cluster size of 5, 10 and 15. The average execution time increases to 78,000ms when the cluster size is 20.

4.6. Experiment Six

In this experiment, the average time taken for each algorithm was calculated keeping the input size = 1,000 lines, and varying the cluster size from 5 nodes to 20 nodes by increments of 5. Figure 4.6 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

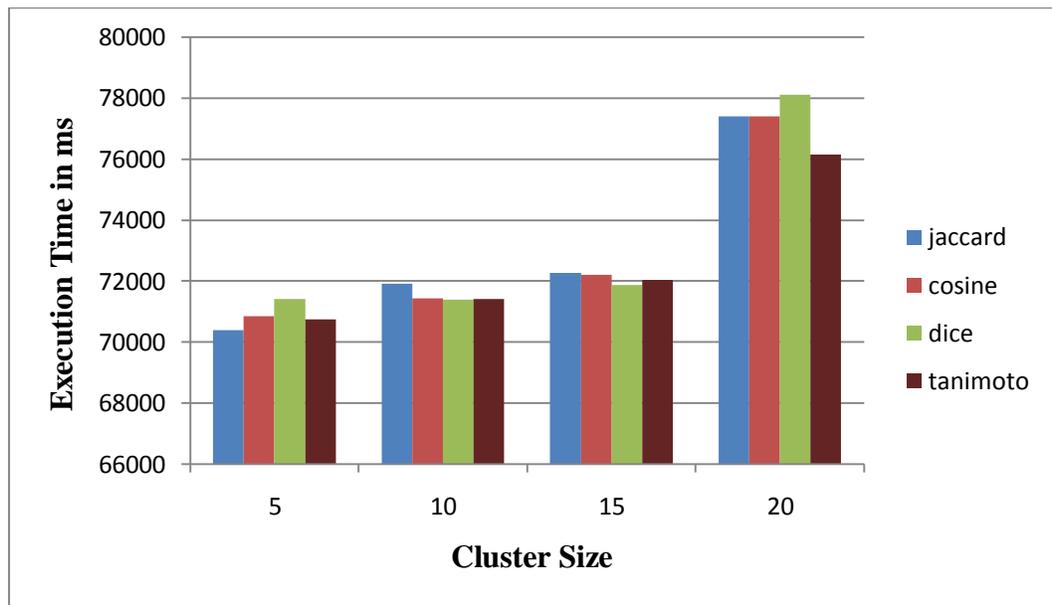


Figure 4.6. Experiment Six: Average Execution Time vs. Cluster Sizes for Input Size of 1,000

It can be seen from Figure 4.6, for input size of 1,000 the average execution time is 71,000ms for cluster size of 5 and 10. For cluster size 15, the average execution time is 72,000ms. The average execution time increases to 77,000ms when the cluster size is 20.

Table 4.6. Average Time Taken vs. Cluster Sizes for Experiment Six

Cluster Size	5	10	15	20
Jaccard's	70387.4	71914.3	72279.8	77400.4
Cosine	70844.1	71443.4	72213.9	77398.5
Dice's	71418.2	71389.5	71862.8	78112.9
Tanimoto	70755.2	71414.0	72048.2	76156.8

4.7. Experiment Seven

In this experiment, the average time taken for each algorithm was calculated keeping the input size = 10,000 lines, and varying the cluster size from 5 nodes to 20 nodes by increments of 5. Figure 4.7 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

It can be seen from Figure 4.7, for input size of 10,000 the average execution time is 68,500ms for cluster size of 5. For cluster size 10, the average execution time is 74,000ms. For cluster size 15, the average execution time is 76,000ms. The average execution time increases to 77,000ms when the cluster size is 20.

Table 4.7. Average Time Taken vs. Cluster Sizes for Experiment Seven

Cluster Size	5	10	15	20
Jaccard	69101.4	74717.1	76746.6	76725.6
Cosine	69506.7	74874.8	76372.3	77672.0
Dice	69888.7	75046.4	76715.0	78502.9
Tanimoto	69078.1	73820.6	75940.0	76762.9

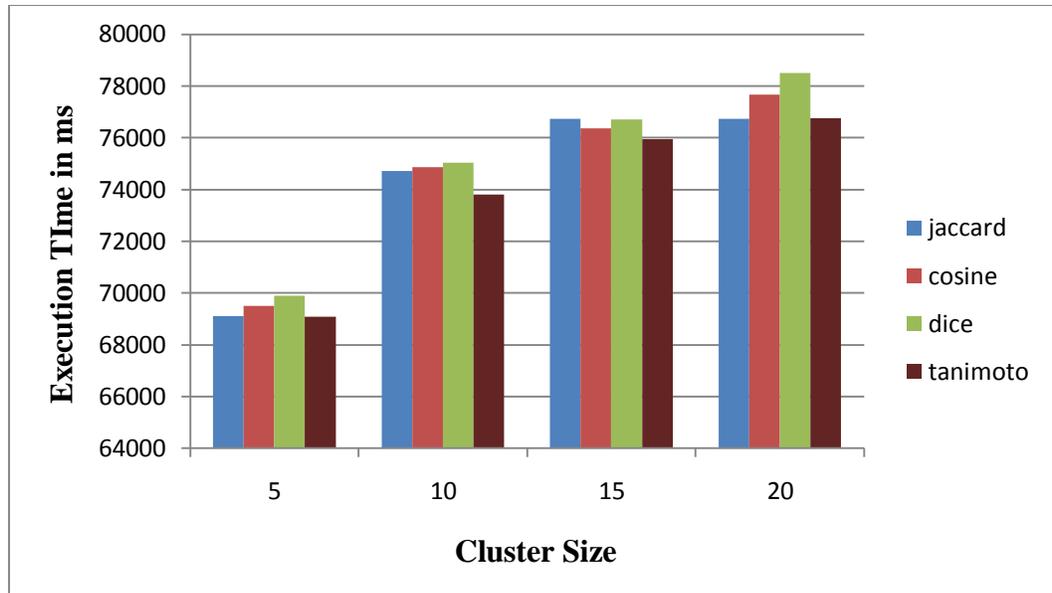


Figure 4.7. Experiment Seven: Average Execution Time vs. Cluster Sizes for Input Size of 10,000

4.8. Experiment Eight

In this experiment, the average time taken for each algorithm was calculated keeping the input size = 100,000 lines, and varying the cluster size from 5 nodes to 20 nodes by increments of 5. Figure 4.8 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

It can be seen from Figure 4.8, for input size of 100,000 the average execution time is 79,000ms for cluster size of 5. For cluster size 10, the average execution time is 82,000ms. For cluster size 15, the average execution time is 81,000ms. The average execution time increases to 84,000ms when the cluster size is 20.

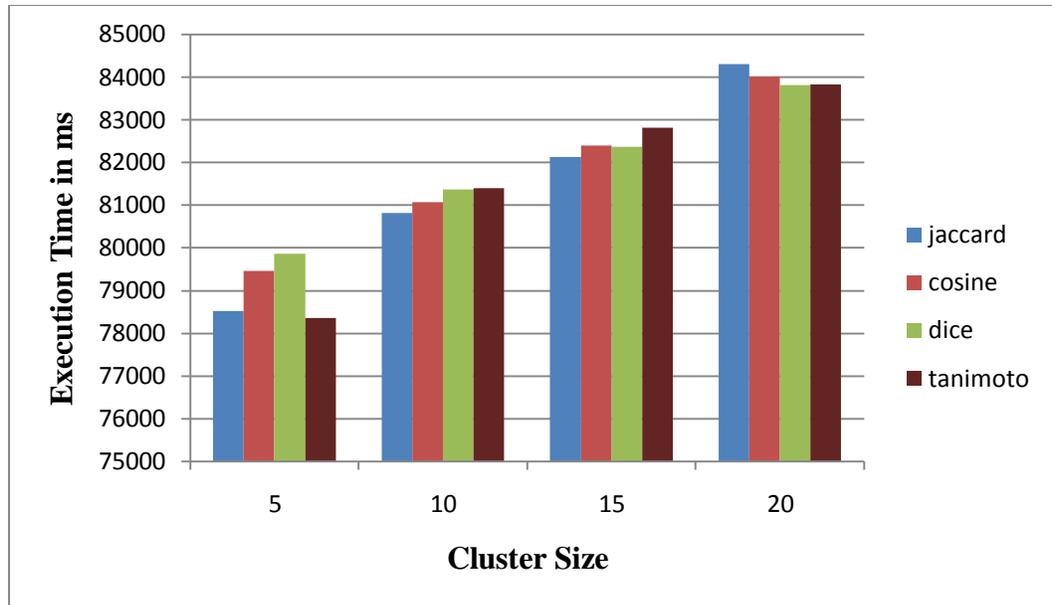


Figure 4.8. Experiment Eight: Average Execution Time vs. Cluster Sizes for Input Size of 100,000

Table 4.8. Average Time Taken vs. Cluster Sizes for Experiment Eight

Cluster Size	5	10	15	20
Jaccard	78520.8	80820.6	82134.6	84304.5
Cosine	79469.2	81078.4	82404.8	84007.8
Dice	79868.2	81372.5	82363.9	83808.5
Tanimoto	78354.2	81400.7	82820.7	84831.8

4.9. Experiment Nine

In this experiment, the average time taken for each algorithm was calculated keeping the input size = 1,000,000 lines, and varying the cluster size from 5 nodes to 20 nodes by increments of 5. Figure 4.9 shows the graph of time taken by each algorithm plotting the time (in ms) on the y-axis and input sizes on the x-axis.

It can be seen from Figure 4.9, for input size of 1,000,000 the average execution time is 171,000ms for cluster size of 5. For cluster size 10, the average execution time

is 169,000ms. For cluster size 15, the average execution time is 165,000ms. The average execution time decreases to 161,000ms when the cluster size is 20.

It can be seen from Figure 4.9, for input size of 1,000,000 the average execution time is 171,000ms for cluster size of 5. For cluster size 10, the average execution time is 169,000ms. For cluster size 15, the average execution time is 165,000ms. The average execution time decreases to 161,000ms when the cluster size is 20.

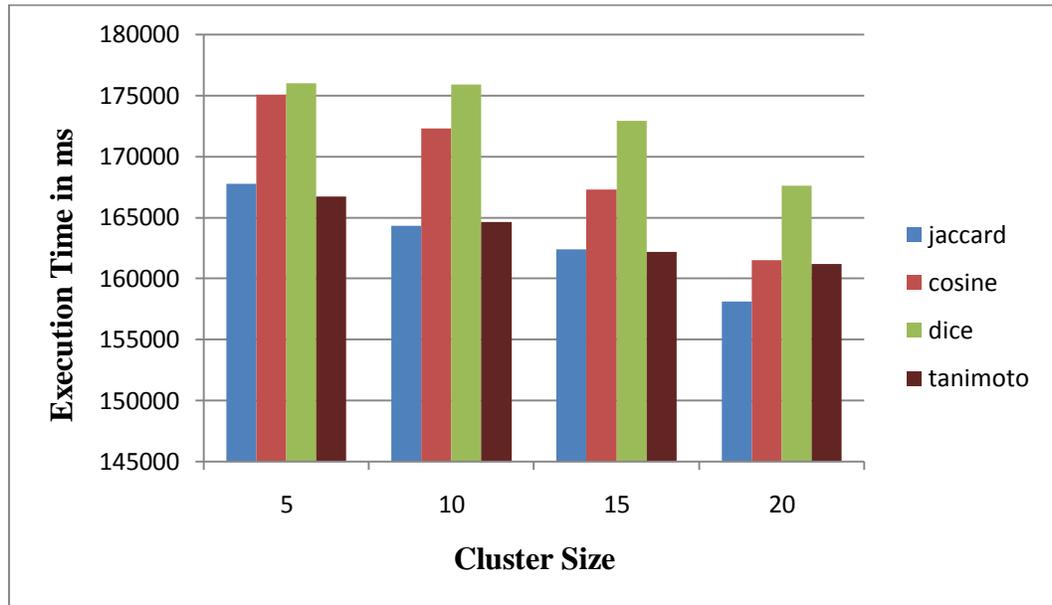


Figure 4.9. Experiment Nine: Average Execution Time vs. Cluster sizes for Input Size of 1,000,000

Table 4.9. Average Time Taken vs. Cluster Sizes for Experiment Nine

Cluster Size	5	10	15	20
Jaccard	167752.0	164337.0	162370.0	158108.0
Cosine	175055.6	172311.2	167311.2	161519.0
Dice	175991.0	175908.0	172908.0	167611.0
Tanimoto	166740.1	164638.4	162182.0	161196.9

From the experiments it can be observed that, as the cluster size increases the time taken to execute the algorithms decreases. It is a noteworthy observation that for smaller inputs (input size = 100, 1,000, 10,000), the execution time of the algorithms increases as the cluster size increases. This happens because the block size is too big for smaller inputs. In order to create enough splits of input, we define *split size*. The split size is calculated based upon the cluster size to be used. For larger inputs (input size = 1,000,000) the execution time decreases as the cluster size increases.

5. CONCLUSION AND FUTURE WORK

MapReduce has emerged as a popular way to harness the power of large clusters of computers. MapReduce allows programmers to think in a data-centric fashion, they focus on applying transformations to sets of data records, and allow the details of distributed execution, network communication and fault tolerance to be handled by the MapReduce framework.

The MapReduce programming model has been successfully used for many different purposes at many large-scale corporations for the following reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. Third, the implementation of MapReduce scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources, and therefore, is suitable for use on many of the large computational problems.

Overall, the experiments revealed that as the cluster size increases the execution time of the algorithms decreases, with the exception of input sizes 100, 1,000 and 10,000. The reason for this is that the block size is too large for smaller input files. For input sizes of 1,000,000 and above, the execution time decreases with increasing cluster sizes as expected resulting in a speedup. We have seen that the MapReduce programming model is beneficial for larger inputs. For smaller inputs, MapReduce may not be the ideal solution. But, when it comes to working on real time data in

large corporations, where the input size is in the order of terabytes, MapReduce can be the most efficient way to mine that data.

This work in this paper represents only a small step towards using the MapReduce programming technique in the analysis of large social network analysis. There are a number of extensions one can make to this particular work to make it more useful for the real world. One of those can be filtering out users and pages by certain characteristics. Another potential direction would be to filter the data into time slices to track trends over time. Furthermore, this paper does not currently weight a co-occurrence that appears many times over a co-occurrence that appears only once. All of these additions would be useful expansions of our current work.

REFERENCES

- [1] Seifoddini, Hamid. Djassemi, Manocher. Merits of the Production Volume Based Similarity Coefficient in Machine Cell Formation, (2007).
- [2] Qian, Gang. Sural, Shamik. Gu, Yuelong. Pramanik, Sakti. Similarity between Euclidean and cosine angle distance for nearest neighbor queries, (2003).
- [3] Bombyx, Mori Seyed. Dalirsefat, Benjamin. Meyer, Andréia Da Silva, Mirhoseini, Seyed Ziyaeddin. Comparison of similarity coefficients used for cluster analysis with amplified fragment length polymorphism markers in the silkworm, (2008).
- [4] Yin, Yong. Yasuda, Kazuhiko. Similarity coefficient methods applied to the cell formation problem: A taxonomy and review, (2002).
- [5] Garbiel, Ibrahim. Parsaei, Hamid. Leep, Herman. Machine Cell Formation Based on a New Similarity Coefficient, (2003).
- [6] Sharma, Alok. Lal, Sunil Pranit. Tanimoto Based Similarity Measure for Intrusion Detection System, (2011).
- [7] Butina, Darko. Unsupervised Data Base Clustering Based on Daylight's Fingerprint and Tanimoto Similarity: A Fast and Automated Way To Cluster Small and Large Data Sets, (1998).
- [8] Roberto J. Bayardo, Yiming Ma, Ramakrishnan Srikanth, Scaling Up All Pairs Similarity Search, (2009).
- [9] Abreu, Rui. Zoetewij, Peter. VanGemund, Arjan. An Evaluation of Similarity Coefficients for Software Fault Localization, (2010).

- [10] Sesli, Mehmet. Yegenoglu, Emine Dilsat. Compare various combinations of similarity coefficients and clustering methods for *Olea europaea sativa*, (2005).
- [11] http://www.cbsolution.net/ontarget/mapreduce_vs_data_warehouse, retrieved on July 10th, (2012).
- [12] Dean, Jeffrey. Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters, (2002).
- [13] <http://hadoop.apache.org>, retrieved on Oct 1st, (2012).
- [14] Noll, Michael. Running Hadoop On Ubuntu Linux (Multi-Node Cluster), (2007).
- [15] <http://www.linfo.org/inode.html>, "Inode Definition". The Linux Information Project. September 15, 2006. Retrieved on January 12th, 2012.