Clustering using Simhash and Locality Sensitive Hashing in Hadoop – HDFS : An Infrastructure Extension

Kala Karun.A and Chitharanjan. K
Sree Chithra Thirunal College of Engineering, Thiruvananthapuram, Kerala.
kalavipin@gmail.com, chithanjan_k@yahoo.com

Abstract— Apache’s Hadoop\textsuperscript{1}, an open source implementation of Google’s Map/Reduce framework, is a software framework for large scale data analysis and storage. For data storage, Hadoop uses Hadoop distributed File System (HDFS) which is an open source implementation of Google File System (GFS). Majority of Map/Reduce applications uses HDFS for storing data. By default, the block placement strategy doesn’t consider the data characteristics into account and places the data blocks randomly based on data node availability. Customized strategies in block placement can improve the performance of HDFS. Most of the applications using HDFS require streaming access to the files and if the related files are stored in the same set of data nodes, the query performance can be increased. This paper discusses a method for clustering documents to the same set of data nodes using the technique of Locality Sensitive Hashing. The method uses the compact bitwise representation of document vectors called fingerprints to increase the data processing speed and performance. A new placement policy is implemented without affecting the default fault tolerant properties of Hadoop and requires only minimal changes to the Hadoop framework. The proposed clustering method and the new placement policy will be beneficial to a large number of applications using Hadoop.

Index Terms— Fingerprint, Hadoop, HDFS, Locality Sensitive Hashing.

I. INTRODUCTION

Apache’s Hadoop is an open source implementation of Google’s Map/Reduce [1] framework. It enables distributed, data intensive and parallel applications by dividing a massive job into smaller subtasks called maps and huge datasets into smaller data partitions such that each task processes a different partition in parallel. Hadoop uses Hadoop distributed File System (HDFS) [2] which is an implementation of the Google File System (GFS) [3] for data storage. HDFS has a high degree of fault-tolerance and provides efficient access to application data and is suitable for applications having huge data sets. Similarity search is used in various areas of computing and in low dimensional metric space, the problem is well solved. But in high dimensional data spaces, the current solutions are not much satisfactory. Most of them provide very little improvement over the linear solution and many of the indexing solutions degrade to linear search for very large dimensions [4]. In big data environment, clustering methods which are very fast and scalable is needed. The running time of similarity search can be reduced using approximation as shown in [5, 6, 7, 8, 9, 10]. An approximate nearest neighbour search based on Locality Sensitive Hashing was

\textsuperscript{1} http://hadoop.apache.org

DOI: 01.IJCS.1.1.2
©Association of Computer Electronics and Electrical Engineers, 2014
introduced in [9, 10] for high dimensional data space with proven sub linear dependence on the dataset size. Hence the clustering will be comparatively faster. The idea behind Locality Sensitive Hashing is to hash the points using different hash functions in such way that for every hash function, the probability of collision will be higher for similar points than the dissimilar ones. The nearest neighbors can be found by hashing the query points and retrieving the values from the hash buckets corresponding to the hash value of the query point.

A dimensionality reduction technique called simhash [11] is used in the proposed method for the representation of documents. Simhash converts high dimensional vectors into small fingerprints. Fingerprints can be generated using different methods. The main techniques are using simhash and shingles [12]. The main advantage of using simhash over shingles is that simhash requires only a lesser number of bits to represent fingerprints as compared to shingles [13]. Simhash has two main properties: (i) The fingerprint of a document is the hash of its features and (ii) Similar documents will have similar fingerprints. It is practically shown that simhash is very useful in representing documents for the detection of near duplicates in multibillion page repository [14].

A. Default Data Placement in HDFS

HDFS uses a simple and efficient policy to allocate replicas for a block. If a process running on any of the HDFS cluster nodes opens a file for writing a block, then one replica of that block is allocated to the same machine on which the client is running. The second replica is allocated on a randomly chosen rack that is different from the rack on which the first replica was allocated. The third replica is allocated on a random machine on the same remote rack that was selected in the earlier step.

B. Scope

The default data placement policy in Hadoop places data blocks in the data nodes without considering the data characteristics. Blocks will be placed randomly considering mainly the data node availability. Each block will be considered independent and can’t control the placement locations. Customised strategies can improve the performance of the placement process to a great extent. Placing a related set of data items to the same set of nodes will be highly advantageous for a number of applications like plagiarism detection, spam detection, web crawling, near duplicate detection in large text corpora, answering the query to find related documents, analysis and recommendations of large scale collections of scientific publications etc. Using the clustering feature, map only applications can be created which further enhance the performance of Map/Reduce applications. Also, if the related files are placed in the same set of nodes, applications can be created with less number of mappers using the CombineInputFileFormat and thus the resource consumption and hence the energy consumption can be reduced. The objective of this paper is to efficiently represent documents using simhash, cluster large collections of documents using locality sensitive hashing, implement a new placement policy in HDFS which will allocate data nodes for data placement based on the file name format and to evaluate the performance of the proposed method with the already existing default Hadoop.

II. RELATED WORK

The performance of join operation on large collections of input can be improved by co - partitioning of the input files. If the similar items are clustered, the performance in Hadoop can be improved by using map-only join algorithms by reducing the overhead of remote I/O. Most of the past work provides no methods for clustering in Hadoop. Some of the techniques suggested [15, 16] provides some degree of co-location but require immense changes in the default framework and the physical data layout.

HadoopDB [15] stores data in the local RDBMS’s using ACID conforming DBMS engines which will affect the dynamic scheduling and fault tolerance of Hadoop. HadoopDB changes the interface to SQL and hence the programming model is not as simple as Map/Reduce. Also intensive changes are required to glue together Hadoop and Hive frameworks in HadoopDB. Hadoop++ [16] is Hadoop with Trojan Index and Trojan Join. In terms of query processing Hadoop++ matches and improves the query run times of HadoopDB. A non intrusive approach is used in Hadoop++ without changing the underlying Hadoop framework. It changes the internal layout of a split, which is a horizontal partition of the data and writes appropriate user defined functions. The basic idea behind Hadoop Aggressive Indexing Library (HAIL) [17] is to create indexes on attributes of interest at load time with minimal changes to Hadoop. HAIL maintains the block replicas in different sort orders and different clustered indexes. So the chance of finding an appropriate index increases and improves the runtime performance. Indexes are created while uploading data to HDFS and hence
additional data read is not required. HAIL is not suited for processing of full text files since some sort of schema is required to create indexes. Elastic Replica Management System (ERMS) [18] introduces an active/standby storage model which takes advantage of a high performance complex event processing engine to distinguish the real time data types and brings an elastic replication policy for the different types of data. Based on the data access patterns, data in Hadoop can be classified into different types. Hot data – data having a large number of concurrent access and high intensity of access, cold data- unpopular and rarely accessed data, normal data – rest of the data other than hot and cold. ERMS introduces active/standby storage model which classifies the storage nodes into active nodes and stand by nodes. In Trojan HDFS [19], a new data layout called Trojan Layout is used which internally organizes data blocks into different attribute groups according to the workload to improve the access performance. But it has not adapted to changes in workload and also index creation is expensive. Query performance improves only for the indexed attributes and can’t improve I/O performance. Cheetah [20] is a scalable and flexible custom data warehouse built on top of Hadoop. It combines the benefits of both data warehouse and Map/Reduce technologies. It provides a simple query language which is easily understood by people having little SQL knowledge. It utilizes Hadoop’s optimization techniques for data compression, access methods, materialized views and provides high performance by processing 1 GB of raw data per second. Since data is usually stored in column format, it has the overhead of query reconstruction and cannot guarantee that all fields of the needed record are present in the same Data Node. RCFFile [21] is a data placement structure introduced for Map/Reduce based data warehouses like HDFS. It combines the advantages of both row and column stores. Similarity with row stores is that data in a row will be present in the same node and hence the tuple reconstruction cost is less. As a column store it can perform data compression and can avoid unnecessary column read operations. Since the columns are highly interleaved in a single block, efficient I/O elimination will become difficult. Distributed Adaptive Replication (DARE) [22] is an adaptive data replication mechanism that helps in achieving a high degree of data locality. Using probabilistic sampling and competitive aging the problem of determining the replication factor of a file and the placement locations is solved. Data with correlated access are distributed to various nodes as new replicas are created and old ones expire which also enhance data locality. No extra network overhead is incurred since it is making use of existing remote data retrieval. The algorithm creates replicas of popular files and at the same time minimizes the number of replicas of unpopular files. Clydesdale [23] is a research prototype built on top of Hadoop for structured data processing. It improves performance without making changes to the underlying platform. It inherits the scalability, fault tolerance and elasticity of Map/Reduce. For storing data, Clydesdale uses HDFS. Column oriented layout is used to store data and tries to co-locate the columns of a given row to the specified node. Map tasks are carefully designed so that the data structures used for query processing can be shared by multiple threads and allows multiple tasks to execute successively on any node. As of now, no SQL parser is present and the SQL queries are submitted as Map/Reduce programs in Java. Updates to dimension tables are not permitted and scheduling of Clydesdale workloads along with Map/Reduce loads pose network load management problems. Also, parallelism may be limited for small data sets. In Co-Hadoop [24], application programs give hints regarding the locations where the files are to be stored. Based on the hints, the files will be stored together. Detailed information is not provided on how to determine the hints for data placement. It also requires detailed knowledge of input data. The client programs need to explicitly specify to HDFS which files should be placed together in the data nodes through hints. Intense modification to the existing Hadoop framework is needed to incorporate the changes. It is more flexible compared to Hadoop++ and HadoopDB but slow due to high network utilization. In the proposed method, there is no need for the client to provide the hint explicitly. The concepts of Locality Sensitive Hashing and Fingerprints are effectively included to improve the performance. Also, the pluggable placement policy and interface [29] of Hadoop will be used which will implement the new placement policy without affecting the default Hadoop framework. Clustering is the classification of objects (data, articles, documents etc.) into different groups in such a way that the objects in the same group will have some same characteristic. Single Pass Clustering [25] processes the documents one by one by comparing each document to the existing clusters. A fixed similarity threshold value is used and if the similarity between the document and any cluster is above the similarity threshold, the document will be added to the most similar cluster, otherwise a new cluster will be formed. In K-Nearest Neighbor Clustering [26], for every document, its similarity to every other document is computed and selects the top k documents. The new document is assigned to the cluster having a majority of the top k documents. In Suffix Tree Clustering [27], a tree of phrase suffixes shared among multiple documents are created. The documents sharing a suffix are treated as the base cluster and are merged together if they have a document overlap greater than 50%. DC- Tree Clustering [28] is based on B+ trees and is based on the vector space
representation of documents. Each node in the tree denotes a cluster represented by the combined feature vector of its individual documents. It has two disadvantages. First one is that once a document is assigned to a cluster, it can’t be reassigned later to a new cluster. Second is cluster overlapping is not permitted. The main problem with hierarchical and partitional clustering is that they are computationally intensive. They require more memory and reduce the clustering rate of the system. The growths of the internet allow massive dissemination of online data from Google, Yahoo etc. Conventional clustering methods are inadequate in these cases due to inaccuracies and huge delays. This is because these methods may need multiple iterations and whenever an object arrives, it may need to be compared with all of the existing clusters to find an appropriate one. Since the number of clusters is very large, this incurs much delay. The above mentioned clustering techniques can’t always satisfy the speed and scalability requirements of the big data environment. So some alternate methods are needed which will improve the clustering of massive online data.

Currently most of the big enterprises are moving towards using Hadoop –HDFS for the storage and analysis of online or streaming data. Co-location of data of online applications using HDFS can improve the performance and can reduce the access latency. It can be achieved through incremental clustering. In incremental clustering algorithms, the data objects are considered one at a time in the arriving order and are assigned to the respective clusters.

Instead of using the above mentioned clustering approaches, this paper proposes a method for clustering related documents incrementally in Hadoop – HDFS using Locality Sensitive Hashing (LSH). The main idea behind LSH is that a set of different hash functions is used for hashing data points to hash values or signatures and there is a high probability of collision for the data points which are similar i.e. similar data objects will be having the similar signatures or hash values. Similar data items will be hashed to the same buckets. No changes are required in Hadoop for implementing LSH based clustering since the clustering is controlled by the client side and only the data placement targets are determined in HDFS using a new placement policy. The new placement policy is also non invasive since the placement policies are made pluggable in the later versions of Hadoop [29].

III. PROPOSED METHOD

A. Document Representation

The proposed method uses a vector space representation of the documents. The documents are preprocessed (stop words removal, stemming etc.) and represented as vectors of term frequencies or tfidf [30]. Considering News Groups, Web Crawlers etc., there will be millions of documents and the dimensionality will be very large. Hence, processing of millions of high dimensional vectors requires considerable amount of time. To make the process faster, a compact bit string representation of each document vector called signature or fingerprint is used. Signatures are compact bit string representation of the vectors. The main advantage of signatures is that they are many orders of magnitude smaller than the original feature vectors and similar documents have similar fingerprints which make the process of measuring the similarity very fast since only bitwise operations are needed.

If an ‘f’ bit fingerprint or signature is needed, then an ‘f’ dimensional vector ‘v’ is used which is initialized to 0. Each dimension of the document vector is hashed to an ‘f’ bit hash value, Murmur1 is used here choice since it is very fast. If the i\text{th} bit of this hash value is 1, then the i\text{th} component of vector ‘v’ is incremented by the weight of the dimension and if the i\text{th} bit is 0, the i\text{th} component of ‘v’ is decremented by the weight of the dimension. After considering all features of each document vector, vector ‘v’ with ‘f’ components which is either positive or negative is obtained. The final ‘f’ bit signature is obtained from the vector ‘v’ depending on the sign of the components of ‘v’.

B. Nearest Neighbors using Locality Sensitive Hashing

In most of the conventional clustering approaches each document vector needs to be compared with each of the existing cluster to find the most similar cluster. In large data environments, thousands of clusters will be present and this makes the process highly time consuming. The number of candidate clusters that needs to be compared can be considerably reduced by considering only the nearest neighbors. Locality sensitive hashing can be used to find the nearest neighbors since it has sub-linear dependence on the dataset size and hence very fast. So only the nearest neighbors will be checked to find the appropriate one rather than comparing the entire thousands of clusters.

1 https://sites.google.com/site/murmurhash/
In the proposed method, ‘l’ hash tables corresponding to ‘l’ locality sensitive hash functions $g_i(x)$, $g_2(x)$...,$g_l(x)$ is used. Each $g_i(x)$ is a ‘k’ bit Locality Sensitive Hash Function, $g_i(x) = [h_1(x), h_2(x),…, h_k(x)]$ where each $h_i(x)$ returns the $i^{th}$ bit of fingerprint ‘x’. Each $g_i(x)$ creates the hash key by concatenating the bits returned by the randomly selected hash functions $h_1, h_2,..., h_k$ [31]. Each cluster centroid fingerprint ‘x’ will be hashed to the ‘l’ hash tables by applying the hash functions $g_1(x), g_2(x)...g_l(x)$. The hash values will be cluster information. The algorithm for hashing the fingerprints to the hash tables is shown in Fig. 1. Here, documents are represented using document fingerprints and the clusters using the fingerprints of their centroids. The cluster centroid is initially represented as a vector created by averaging the feature vectors of the member documents which will then be converted to fingerprint using the same procedure for individual documents and will be hashed to the hash tables.

**Algorithm - Hashing Fingerprints to Hash tables**

**Input:** A set of fingerprints $X = \{x_1,...,x_n\}$, l (number of hash tables), 

**Output:** Hash tables $T_i$, i = 1, ..., l 

For each i = 1,..., l 

Create hash table $T_i$ by generating a random k bit hash function $g_i(x; x \in X) = (h_1(x),h_2(x),...,h_l(x))$, where each $h_i(x)$ is a random hash function returning the $i^{th}$ bit of fingerprint x. 

For each hash table $T_i$, i = 1,..., l 

For each fingerprint $x_j$, j = 1,..., n 

Store fingerprint $x_j$ on bucket $g_i(x_j)$ of hash table $T_i$

**Algorithm - Approximate Nearest Neighbor Query**

**Input:** A query fingerprint ‘q’, ‘K’ (number of nearest neighbors), Access to hash tables $T_i$, i = 1,..., l generated by the previous algorithm 

**Output:** K (or less) approximate nearest neighbors $S = \emptyset$ 

For each i = 1,..., l 

$S = S \cup \{\text{fingerprints found in } g_i(q) \text{ bucket of table } T_i\}$

Return the K nearest neighbors of ‘q’ found in set $S$

When a query arises to find the ‘k’ nearest clusters of ‘x’, the more similar ‘k’ entries in the ‘l’ hash tables corresponding to the hash value of $g_1(x), g_2(x)...g_l(x)$ will be returned. The algorithm for finding the approximate nearest neighbors of a query fingerprint q is given in Fig.2. The client application program will be controlling the entire clustering process. When a document needs to be clustered, it will be converted to its fingerprint and hashed to get the hash value ‘v’. The ‘l’ hash tables will be searched with this hash value to get the ‘k’ nearest neighbors. Two different situations arise here: ‘k’ candidates will be returned - Then the similarity of the returned candidates will be checked with the document fingerprint and the cluster_id of the most similar cluster will be returned. If no existing cluster is having similarity greater than or equal to the fixed similarity threshold, a new cluster will be created with a unique cluster-id and the cluster fingerprint will be calculated and cluster_id will be returned. The ‘l’ hash tables and the cluster information maintained on the client side will be updated. No entry corresponding to the hash value – In this case, a new cluster will be created with a unique cluster-id and the cluster fingerprint will be calculated and the cluster_id will be returned. The ‘l’ hash tables and the cluster information maintained on the client side will be updated.

**C. Proposed Method for Data Placement**

The default data placement in Hadoop is already explained in section I. In the proposed method, placement of related data to the same set of nodes is controlled by the client side application program which requires only minimal changes to the Hadoop framework and will not affect the default fault tolerance and scalability properties of Hadoop. When a client wants to write a file, a write request will be given to the name node of HDFS. The file name format in the write request guides the block placement process. If the file name satisfies a particular format, the new placement policy will be used otherwise it will stick on to the default policy. Here, cluster_id computed by the client side aids the block placement. For every file, the cluster_id is appended to the file name and it will be provided by the client in the write request to the name node of
HDFS. A new placement policy is implemented which will place the blocks based on the given cluster_id. The files with same cluster_id will be replicated to the same set of data nodes.

```java
1: class BlockPlacementPolicyWithClusterId extends BlockPlacementPolicyDefault
2:   clusterIdToDatanodesMap = new HASHMAP() // To store the cluster-id to data nodes mapping
3:   function INITIALIZE() \rightarrow Used to setup a BlockPlacementPolicy object.
4:   function getClusterId(String srcPath) \rightarrow srcPath is assumed to be of the format "path@clusterId" and this will return the cluster_id from the file name.
5:   function CHOOSETARGETS(String srcPath, int numOfReplicas, DatanodeDescriptor writer, List<DatanodeDescriptor> chosenNodes, long blocksize) /* choose numOfReplicas nodes for writer to replicate a block with size blocksize if not, return as many as possible. */

Parameters:
- srcPath- a string representation of the file for which chooseTarget is invoked, numOfReplicas - number of replicas wanted. Writer - the writer's machine, null if not in the cluster. blocksize - size of the data to be written.

Returns:
- Array of DatanodeDescriptor instances chosen as targets and sorted as a pipeline.

6:   String clusterId = getClusterId(srcPath)
7:   if (clusterId == null) then
8:     datanodes= CHOOSETARGETSBYDEFAULTPOLICY() \rightarrow here, no co-location is needed, so select the target datanodes using the default policy.
9:   else DatanodeDescriptor[] datanodes = clusterIdToDatanodesMap.get(clusterId) \rightarrow get the list of data nodes from the map corresponding to the cluster_id
10:  if (datanodes == null) then \rightarrow No entry in the map corresponding to the cluster_id
11:  datanodes= CHOOSETARGETSBYDEFAULTPOLICY() \rightarrow select the target datanodes using the default policy
12:  clusterIdToDatanodesMap.put(clusterId, datanodes) \rightarrow adding the data nodes selected using default policy to the map for further use.
13:  for each d in datanodes
14:    if (!ISGOODTARGET(d)) then \rightarrow d is not a good target data node
15:      datanodes.remove(d) \rightarrow \textcolor{red}{d}= CHOOSETARGETSBYDEFAULTPOLICY()
16:  datanodes.put(d)
17:  end for
18: return datanodes \rightarrow Use the returned set of data nodes for block placement
```

Figure 3. Class BlockPlacementPolicyWithClusterId

The Java class “BlockPlacementPolicyWithClusterId” used for implementing the new placement policy is shown in Fig. 3. This class is responsible for choosing the desired number of targets for placing block replicas in such a way that all the files in the same cluster are placed in the same set of data nodes. The replica placement strategy is adjusted to: If the writer is on a data node, the 1st replica is placed on the local node (or local node-group), otherwise a random data node. The 2nd replica is placed on a data node that is on a different rack with 1st replica node. The 3rd replica is placed on a data node which is on a different node-group but the same rack as the second replica node.

The latest versions of Hadoop (since version Hadoop -0.21.0) have a pluggable interface for implementing block placement policies in HDFS. An API is also provided that permits a module external to HDFS to specify and control how data blocks should be placed. This will ease the implementation of the proposed method. The clients can use their own block placement policy by setting the configuration property “dfs.block.replicator.classname” in the configuration file hdfs-site.xml. This doesn’t require re-compiling Hadoop. The cluster_id calculated by the client side application will be attached to the file name using “@”. So the source file path will be of the form “sourcepath@cluster_id”. When the placement policy is called, a hash dictionary, clusterIdToDatanodesMap, is created to hold mappings from a cluster-id to the list of data nodes storing files having that cluster_id (line 2). When an HDFS client writes a block, the first task is to request the name node to choose target data nodes for the block and its replicas. The name node then retrieves the cluster_id from the source path (line 6). If co-location is not needed, the client provides path without cluster_id and name node retrieves a null value for the cluster_id. If the cluster_id is null, name node will find the target data nodes for block placement using the default placement policy. After getting the cluster_id, the data nodes corresponding to the cluster_id will be accessed from the hash map. If the cluster_id is not null, but no entry is present in the hash table corresponding to the cluster_id, then the target
data nodes are determined using the default placement policy and a new entry will be created in the hash map for the cluster_id and the selected list of data nodes will be added to it for further placement purposes. If cluster_id is not null and a list of data nodes is returned, then each node is checked for suitability and if any node is found to be not a good target, then it will be replaced by one selected by default placement policy. Finally the list of target data nodes will be returned to the client.

IV. PRACTICAL APPLICATIONS OF CLUSTREERING IN HDFS

The concept of clustering can be deployed in a variety of applications like plagiarism Detection, Spam Detection, web crawling, near duplicate detection, answering the query to find related documents etc. Here we consider how the clustering in HDFS can be exploited to improve the query to find the k nearest neighbors of the given input document. Identifying the similar documents benefits different applications like online news browsing using keyword search, large scale scientific publication analysis etc.

A. Map/Reduce Program to find K Nearest Neighbor

For a collection of documents that fit into main memory, the solution is straightforward. With the advancements in the fields of storage, multimedia and networking technologies, more and more massive data sets are emerging every day and so some efficient method is needed to solve the problem. The size of the data set indexed by Bing search engine of Microsoft is over 300 TB [32]. It is very clear that a single processor is insufficient because of the limitations on the processing speed and memory capacity. So map/reduce based systems are best suited in this scenario. The performance of the Map/Reduce program to find 'k' nearest documents to the given document is considered to evaluate the performance of the proposed method. The document directory, the query file name and the similarity threshold will be given as the input. Map will read the files from HDFS, compares each one with the query input file and finds the similarity. If the similarity is greater than the similarity threshold, it will output <filename, similarity> as the <key, value> pair and will be given to the reducer. In the reducer, the top k similar values will be selected and written to the output file.

Execution in Hadoop without the new placement policy: If the files are not clustered in HDFS, the default placement policy will be used by the name node of HDFS and the files will be placed without considering the data characteristics and they will be distributed randomly in the cluster and the files will be scattered around in the large cluster. So, the map phase will be executed in a very large number of nodes where the files are present and the reducer gets output from a large number of data nodes. So network traffic delays and communication overhead will be more in this case and hence the query latency. Since a large number of data nodes are involved, the shuffling phase also will take more time. The resource utilization will be increased which in turn results in more energy consumption.

Execution in Hadoop with new placement policy: Here, if the files are clustered, the name node will use the new placement policy for data placement and the related set of files will be replicated to the same set of data nodes based on the cluster_id. So the majority of map tasks producing output will be running on a small set of data nodes and reducer will get the output from this lesser number of data nodes. So compared to plain Hadoop (without new placement policy), the network traffic delays are less and the output will be obtained fast from the reducer. Here, the resource utilization and hence the energy consumption will be less. If the files are clustered, applications can be created which will process the input files with lesser number of mappers (using “CombineFileInputFormat”) or having mappers only without reducers (map only jobs) which will further enhance the performance to a great extend.

V. EXPERIMENTAL TESTING AND ANALYSIS

A. Experiment Environment

We evaluated the performance of the k nearest neighbor Map/Reduce program in a private cluster with one name node and eighteen data nodes, all commodity computers. The nodes have Intel x86 64-bit chip architecture, 2 CPUs, 2.5 GHz per core, 4 GB RAM and Standard 1 Gb Ethernet NIC. The Java version is Java 1.7 and the operating system is CentOS. The Hadoop version is Hadoop-0.21.0.

B. Dataset Description

For demonstration purposes, Cluto datasets\(^2\) are used which are in sparse matrix format.

\(^2\)http://glaros.dtc.umn.edu/gkhome/fetch/sw/cluto/datasets.tar.gz
C. Performance and Analysis

Data loading time and query latency are the two metrics considered in evaluating the performance. The query latency directly shows the system performance.

Data Loading Time – The time required to pre-process and load the data are compared for both Plain Hadoop and Hadoop with new policy. Both Plain Hadoop and Hadoop with the new policy load the dataset using Map/Reduce program. The datasets considered are tr11, tr12, tr23, sports, and wap. The data loading time comparison is shown in Fig. 4. Even though the data loading time is slightly more for the Hadoop with new policy, it is less significant while considering the performance gain. The loading time is slightly more because of the increased pre-processing required for the data placement in new Hadoop.

Query latency - The average query latency is compared for both Plain Hadoop and Hadoop with new policy for different cluto datasets for the k nearest neighbor query with different values for k and the similarity threshold. The comparison results are shown in Fig. 5 and Fig. 6.

D. Advantages of the proposed method

i. Minimize network communication overhead: The average number of data nodes that are involved in query processing will be significantly reduced since the related set of data items is present in the same set of data nodes. Communication overhead includes the entire communication involved in executing a query which is a big concern in large distributed environments. HDFS, tries to place all replicas of a data item in a single rack to minimize inter-rack data transfers [33]. By default HDFS doesn’t consider multiple data items during placement. Through clustering, multiple data items can be placed in the same rack to enhance the network performance.

ii. Lesser Resource Consumption and energy consumption: If the related files of a dataset are placed together, applications can reduce the number of mappers involved in the processing using CombineFileInputFormat and hence can reduce the number of nodes involved in the Map/Reduce computations.

iii. Increase the Map/Reduce Performance: The co-location of data reduces the access latency and improves the performance.

iv. Faster clustering through Locality sensitive Hashing and efficient and compact representation of documents through fingerprints.

One concern about the clustering is whether it affects the load balancing in HDFS as it may lead to load unbalancing among the data partitions. Comparing lesser resource consumption and query performance, this is not a big concern.

![Figure 4. Data loading time](image-url)

Query 1: k=7, threshold=0.5
Query 2: k=8, threshold=0.9
VI. CONCLUSIONS

A new approach to cluster documents in HDFS is proposed which will cluster similar documents in the same set of data nodes with minimal changes to the existing framework. For faster clustering operations bit wise representation of the feature vectors called fingerprints or signatures are used. To reduce the number of cluster centroid comparisons, only the nearest neighbors are considered using the technique of Locality Sensitive Hashing which is proved to have a sub linear dependence on the dataset size. A new placement policy is implemented in HDFS which will allocate data nodes for block placement considering the file name format in the write request given by the HDFS client. If the file name is of the form filename ‘@’ cluster_id, data nodes will be allotted based on the cluster_id else, the default placement strategy will be used. The method has proven to improve the performance of Map/Reduce programs. Here, only text documents are considered for clustering and the future plan is to extend it for all kinds of data objects.

REFERENCES
