In-Memory Distributed Indexing for Large-Scale Media Data Retrieval

Yinmiao Ma[1], Danlu Liu[1], Grant Scott[1], Jeffrey Uhlmann[1], Chi-Ren Shyu[1,2,3*]
Electrical Engineering and Computer Science Department[1], Informatics Institute[2]
University of Missouri, Columbia, USA
Bioinformatics and Biomedical Engineering, Asia University, Taichung, Taiwan[3]
shyuc@missouri.edu*

Abstract— **Data retrieval serves a critical role in the development of multimedia applications. However, due to the exponential growth of multimedia data, high-speed and efficient indexing is becoming more and more difficult than ever. In this paper, we propose a novel approach to speed up the retrieval process by adopting a distributed computing paradigm through the Apache Spark framework. Utilizing search trees in a Big Data ecosystem leads to fast and cost-effective media database retrievals by caching indexing structures into memory and aggregating ranked results with flexibilities for users to specify the importance of search cues. We conducted computational experiments on large-scaled vector files for remote sensing image database and synthesized pollen image database to demonstrate the effectiveness and scalability of our system with reasonably high accuracy.**

Keywords- In-Memory Computing, Big Data, Distributed Indexing, Image Database Retrieval

I. INTRODUCTION

In many multimedia applications, range search over multi-dimensional space plays a significant role to understand various visual cues from human perception. Early contributions that were adopted in multimedia search include the *k*-d tree [1], R-tree [2] and their variations. The paradigm of this approach is to create indexing for accelerated access to the content, thereby facilitating search for content via querying, such as content-based image retrieval (CBIR). For search problems that could be modeled in a metric space or using statistical properties, metric tree [3], M-tree [4], Entropy Balanced Statistical (EBS) *k*-d Tree [5], vantage-point tree [6], and multi-vantage-point tree [7] were applied to speed up multimedia search. Fleites *et al.* [8] show the advantage of high-level semantics to solve the I/O overhead problem. Mocofan *et al.* [9] utilized a supervised tree algorithm to speed up the searching and offers the possibility of image features selection.

In addition to tree-based search, methods based on Locality Sensitive Hashing (LSH) can be used for efficient searching when exact retrieval is not essential [10]. It hashes the items by utilizing many hash functions to make sure that items close to each other will be put together with high probabilities. Error Weighted Hashing (EWH) performs better than LSH for the Hamming space regarding nearest neighbor distances [11]. Local Hash-indexing tree (LHI-tree) [12] accelerates space partition localization by random access memory (RAM) and uses hard disks for the hash index. In [13] a CBIR technique that combines a local descriptor with an effective and fast object matching operation is proposed to improve both the search speed and the retrieval accuracy.

Constructing an indexing for large datasets has enormous requirements on computational resource and memory requirement; a single machine usually cannot provide enough support to process massive amounts of multimedia content. Mohamed proposed an MRO-MPI model [14] for multimedia indexing, which uses MapReduce to divide data into groups. A Message Passing Interface (MPI) is involved in transferring data among different processes. However, it does not have an integrated distributed indexing strategy within its retrieval framework; and therefore will experience limited scalability. Given the high complexities of graph indexing, in-memory indexing on a single-machine cannot satisfy the demands of massive graphs. In [15], the authors propose highly efficient distributed indexing system shows that distributed indexing reduces the time for updating the index, and as a result, the index is more up to date at any given time. In [16], a novel disk-based indexing strategy is proposed by distributing indexes across multiple machines and constructing indexes using a MapReduce based method. In the parallel top-*k* search algorithm, the approach is to prune the irrelevant vertices for a specified query in the top-*k* answers before search. It enumerates the best local answers and then returns the global top-*k* answers. This approach can improve efficiency on massive graphs. However, this work relies heavily on disk operation that is normally expensive in I/O.

The emergence of big data challenges traditional indexing methods. The high-speed requirements of search and index generation are critical considerations for processing large amounts of heterogeneous data. Several modified methods are applied in the latest Big Data research which exploit appropriate computing ecosystems. A novel hash based method named Sparse Hashing (SH) was proposed to search in high-dimensional data [17]. For tree-based methods, a new Hadoop-based distributed B+-tree (HB+-tree) was designed by Kaplanis in 2015 [18], which performs better than a centralized concurrent B+-tree index. Bitmap indexes use bitmaps and answer queries by performing bitwise logical operations. It is efficient on large-scale graphs [19] and still has potential to be improved by using more complex partition methods and bitmap compression. Solr [20] provides distributed, real-time index. Solr uses inverted indices, which are common to many information retrieval systems to achieve fast response time for search. It is reliable, scalable and fault tolerant and used in a variety of research fields [21].

The high-performance computing equipment of this research is supported by the National Science Foundation (CNS-1429294).

978-1-5386-2937-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ISM.2017.38

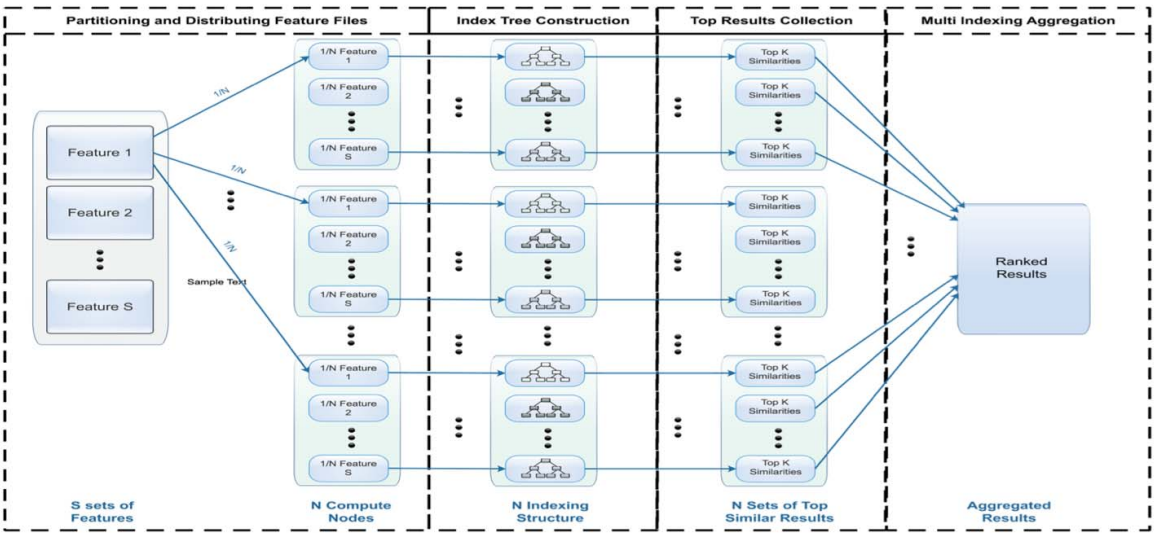

Figure 1 *The system architecture of the in-memory distributed indexing*

Much of the prior research has demonstrated that a key task in distributed indexing is reducing I/O time to speed up the retrieval in big data ecosystems. Apache Spark, an open-source in-memory computing framework, has the strengths to facilitate distributed indexing while minimizing I/O using the Hadoop Distributed File System (HDFS) [22]. Utilizing Spark's in-memory computing capability may further advance the search efficiency and accuracy from large-scaled multimedia data. Recently, researches have introduced advanced techniques for large-scale image retrieval on Spark platform. The utility library, IRlib [23], offers a uniform set of APIs for large-scale image retrieval on Spark platform and reduces search time by making use of GPU. Duan *et al* [24] proposed a CBIR system based on Apache Spark to reduce image retrieval costs with less read and write operations on HDFS and more tasks. In [25], a hashing function is utilized to put images with similar content into the same or adjacent buckets to accelerate the search times. Spark-Fat-Thin-Grid-Index (SFTGridIndex) [26] builds grid index for spatial polygons. In [27], Apache Spark is utilized for parallel search through tree-structured JSON data. A Shuffle-Efficient Similarity Search scheme based on LSH is proposed in [28] to improve the indexing and query performance over LSH. These new advancements bring to the multimedia community the evidences to design algorithms using a Big Data ecosystem.

In this work we propose an I/O efficient, high-scalable, distributed content-based retrieval framework built upon the Spark big data framework. We demonstrate that our approach is scalable and facilitates fast and accurate retrievals over comparative approaches, such as the typical divide and conquer brute-force approaches (over 30x speed up). Additionally, our proposed framework supports user tuning of feature space importance (e.g., colors versus textures) within a multi-index organization, as well as caching of search results. In this system, we have made the following contributions: First, it dramatically reduces the performance time of searches. Second, this system can be extended to different types of indexing structures. Third, for large-scale datasets, this system significantly improves the performance while maintaining a high coverage ratio. Finally, this system is adaptable, allowing users to set the weights on one or more features to customize their desired results.

II. SYSTEM ARCHITECTURE

The system architecture is depicted in Figure 1. There are four modules in our design of the in-memory distributed indexing system. The first module involves partitioning feature files into a certain number of chunks and then distributing them to compute nodes. The feature files contain many instances, each of which represents one image in our system. All feature data are collected in the Hadoop File System (HDFS), where intermediate and end results are also stored in the system. Each feature file for image data, which can vary in dimensionality, is split into multiple sets of features, or chunks, of approximately equal size. The number of chunks (feature sets) for one feature file is set equivalent to or a multiple of the number of compute nodes. In the second module, each compute node separately builds the local indexing structures from the partitioned chunks that it acquired from the previous module. All of the tree structures are stored on the HDFS as intermediate results and later cached back to the memory for subsequent queries. The third module provides the functionality to retrieve top-*k* similar results for each feature group. In the fourth module, multi-indexing aggregation enables us to merge the top *k* results together based on user's requirements to dynamically generate the weighted retrieval. These modules are discussed in detail in Section III.

III. METHODOLOGY

A. Module 1: Partitioning and Distributing Feature Files

The first module presents the procedures of reading feature files from the HDFS, partitioning each feature file into a certain

number of chunks, and distributing all the chunks to compute nodes on the Spark cluster. The pseudocode is shown in Algorithm 1. The feature files (D) archived from HDFS are partitioned into a number of chunks with approximately equal size. Each set of features represents a quantitative measure of visual cues, such as color, texture, shapes, etc. As the feature sets for image data may be different in dimensionality, the chunk size for various feature sets may also vary during the process of partitioning. On line 2, File_Partition(D_i, N_C) utilizes a Spark built-in method called *textFile*(). N_C in the pseudocode determines the number of chunks which should be equivalent to or multiples of the number of compute nodes. The intermediates are then stored into a new RDD object RDD_c. As N_C compute nodes are used, and there are N_f feature files applied in this system, each compute node contains N_f chunks. However, for a large-scale dataset, Spark prefers to get a single partition for a single block of HDFS. The default setting for HDFS block size is 128MB. The number of partitions, which can be set in *textFile*() method, should be equivalent or larger than the number of blocks.

Algorithm 1 Partitioning the Feature Files

D: All feature files
N_f: The number of feature files
N_C: The number of compute nodes
RDD_C: Sets of partitioned chunks

1: for $i \leftarrow 1$ to N_f **do**
2: $RDD_C(i) \leftarrow$ File_Partition(D_i, N_C) for each feature file that $D_i \in D$
3: end for
4: return RDD_C

We defined a new data structure in Spark named *FeatureVector*, which is used to persist the feature vector information for each feature instance. Since the data type of the *textFile*() output is String, the *map*() transformation is necessary to store the feature vectors in RDD_C. In each feature file, the first column is the feature identifier (ID) for each instance and the remaining columns contain feature information. In the *FeatureVector* data structure, we define the *externKey* to store the ID information and *features* to persist the dimensions of data. After transforming the data into *FeatureVectors*, the new *FeatureVectors* replace the previous string representations and are stored into RDD_C.

B. Module 2: Build Indexing Structure

In this module, the indexing structures for the *FeatureVector* chunks need to be built in each compute node. The pseudocode is provided in Algorithm 2. The input is a *FeatureVector* RDD. On line 1, in order to build the indexing tree structure on each compute node, we perform the *mapPartition*() transformation to iteratively insert the value of each chunk of the input RDD into the index structure, which is initially *empty* for each executor. For this paper, we present the implementation using M-tree, which could be replaced by other tree-based or hash-based indexing structures, to accelerate the similarity matching within metric space [4]. Each worker node is concurrently building separate indexes for each feature set. From line 7 to 12 of Algorithm 2, the system constructs M-trees from RDD.

Algorithm 2 Build index structure

Require RDD_C, N_f, N_C, M_{tree}
RDD_C : Sets of partitioned chunks
N_f : The number of feature files
N_C : The number of compute nodes
M_{tree} : Sets of M-trees
RDD_{mtree} : Sets of Indexes
1: function Build_Index(($RDD_C(i), j, M_{tree}(i,j)$))
2: **for** each feature vector $fv \in RDD_C(i)$ in jth compute node **do**
3: Insert(fv, $M_{tree}(i,j)$)
4: **end for**
5: **return** $M_{tree}(i,j)$
6: end function
7: for $i \leftarrow 1$ to N_f **do**
8: **for** $j \leftarrow 1$ to N_C **do**
9: $M_{tree}(i,j) \leftarrow NIL$
10: $RDD_{mtree}(i,j) \leftarrow$ Build_Index($RDD_C(i), j, M_{tree}(i,j)$)
11: **end for**
12: end for
13: return RDD_{mtree}

After tree construction, the output, a collection of M-tree structures, is persisted into the memory of corresponding compute nodes, and will be used for future retrieval purposes. The number of output tree structures is equivalent to the number of input chunks. For example, there are N_f sets of features, and the total number of the M-tree structures distributed on N_C compute nodes is $N_f \times N_C$, which implies that each compute node contains N_f chunks. For the smallest datasets applied in our system, each chunk in any compute node represents for one feature. A new data structure, *MT_Node*, is defined to store each obtained M-tree structure, which can be utilized to retrieve the entire M-Tree structure from the root node to the leaf nodes.

All items inside one partition are sequentially inserted into the root node that it received. Insertions are local operations on each node, which are performed simultaneously in parallel across the cluster. Spark provides two methods to keep the data in memory: *cache*() and *persist*(). Considering that the machine may encounter power issues or any other problems, we could store these intermediate results on the HDFS using *saveAsObject*() operation. Users could reload the data structure to the memory without repartitioning the feature sets and rebuilding the same tree structures.

C. Module 3: Top Results Collection (Indexing Part)

The general purpose of this module is to find top similar cases based on the query instance. In our implementation, the query instances are acquired from all feature sets. Each feature set of the query instances is sent to the corresponding indexing structures persisted in the local memory. For example, in our experiments, the pollen image data includes three sets of features: color, shape and texture. N number of M-trees for each feature file are generated in the previous module with a total of 3N trees for all features. When the indexing starts, the color feature set for the query instance is sent to N color feature M-trees simultaneously. The *k*-NN_search algorithm helps to retrieve the *k* most similar images for the query cases among the M-trees [4].

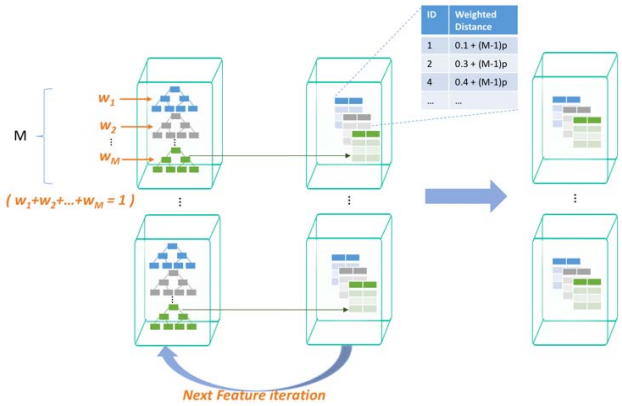

Figure 2. *The process of top similarities indexing.*

Algorithm 3 presents the pseudocode for the *k*-NN_search implementation for similarity queries. On line 1, *flatMapToPair*() transformation in Spark is adopted to implement Knn_Search() in our system to rank the most similar cases in each M-tree. Buffering size *k*, which is the retrieval size for top similarities performed in each indexing structure, is set by the users before the queries are processed. The returned top *k* similar cases collected from each indexing structure are stored as <*key, value*> pairs into a new RDD object, RDD_{knn}. For example, in our implementation, the identifier of each instance in the pollen image dataset is treated as *key* and the measured distance serves as *value* in each pair. On line 1, the query instance q is passed into Knn_Search() function. Lines 1 through 7 are applied to calculate the top *k* results for each M-tree. The buffering size *k*, which is defined as a global parameter by users before the searches, is also passed into the Knn_Search() method. In order to reduce the Spark shuffle operations among compute nodes and to make the parallel computation more efficient, a penalty value p is initially added to each measured distance value universally on line 3 of Algorithm 3. This step will reduce the creation of extra RDDs as intermediate results due the nature of the reducer that favors the operation of key match, instead of mismatch which will require shuffling operations. In the later steps of the process to aggregate ranking, retrieval results appear in the top *k* results from a pair of indexes will receive a reward to remove the initially assigned penalties and results that appear in only one index will be kept and passed to the next pair of indexes for ranking aggregation. This is opposite to the traditional approach to assign penalty to those with key mismatches from retrieval results between a pair of indexes. Those mismatches will be set aside and require shuffling operation in Spark. In our system, penalty p is set approximately 10 times of the maximal distance from the top *k* results and then multiplied by $(N_f - 1)$, where N_f equals to the number of feature files. At last, all results are gathered and returned to the final defined RDD object. On line 10, w_i is set as weights to emphasize the contribution of each feature index. This weight setting provides the flexibilities for the users to select which visual cues are relevant for their research. The transformed pairs are merged together in each compute node. Figure 2 demonstrates the process of finding top similar cases with a buffering size based on the query instance. In Figure 2, there are *M* total feature indexes persisted into the memory on each worker node. Users can set different weights for their desired features $\{w_1, w_2, \ldots, w_M\}$. The system searches the top *k* results for the query object inside each index, where the same feature is searched simultaneously across the nodes. Each distance measurement adds a penalty to the result for that database object. When the search for one feature is finished, the system will repeat the process for the remaining features. Finally, all top results collected from all indexes are merged into a final RDD.

Algorithm 3 Query search

RDD_{mtree} : Sets of Indexes
$RDD_{mtree}(i,j)$: The index for ith feature on jth compute node
N_f : The number of feature files
N_C : The number of chunks
k : The number of top results
w : Weight to emphasis contribution of index
p : The penalty value
Knn_Res: A list of top *k* nearest neighbors of q in $RDD_{mtree}(i,j)$
Knn_ID: A list of identifiers in *Knn_Res*
Knn_Dist: A list of distance value in *Knn_Res*
RDD_{knn} : Sets of top *k* nearest neighbors of q
1: function Knn_Search($q, RDD_{mtree}(i,j), k, N_f, p$)
2: **for** $m \leftarrow 1$ to k **do**
3: *Knn_Dist(m)* ← *Knn_Dist(m)* + $(N_f - 1) \cdot p$
4: **end for**
5: *Knn_Res* ← {*Knn_ID, Knn_Dist*}
6: **return** *Knn_Res*
7: end function
8: for $i \leftarrow 1$ to N_f **do**
9: **for** $j \leftarrow 1$ to N_C **do**
10 $RDD_{knn}(i,j) \leftarrow$ $w_i \cdot$Knn_Search($q, RDD_{mtree}(i,j), k, N_f, p$)
11: **end for**
12: end for
13: return RDD_{knn}

D. Module 4: Multi Indexing Aggregation

In this module, the Multi Indexing Aggregation is applied to combine the top similar cases acquired in module 3 to obtain the overall ranked results. The maximum distance value in the candidate list is utilized as the penalty to handle retrieval results that are missing in the top ranked results from some indexes as reported in our previous work for multi-index CBR search [29]. In this distributed approach, to reduce the number of actions in Spark, the penalty value is also used to mark the missing data. Figure 3 presents the procedure of removing the penalty from the distance value for instances sharing the same identifier. For example, there are three tables with each representing one feature set: {*a,b,c,e*}, {*c,a,d,f*}, {*b,e,a,d*}. In the previous module, a penalty $(N_f - 1) \times p$ was added to distance value in each instance, where N_f is the number of total feature files. Then aggregate the first two datasets and the result was merged with the third dataset. Finally, from the result list, if the instance occurs

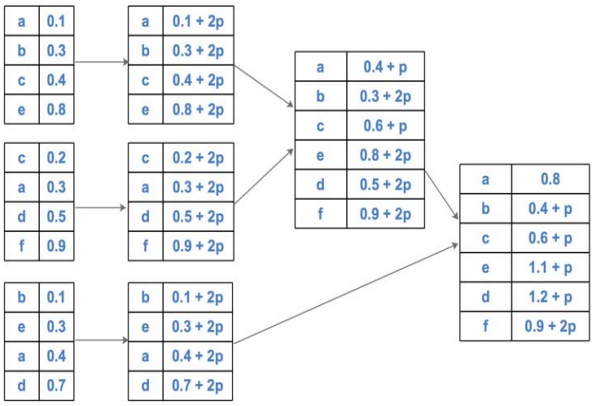

Figure 3. *The chart shows the process to remove the penalty from the distance value.*

in all three tables (features), for example *a*, then this instance would be recorded without penalty through rewarding process. If the instance only appears once, for example *f*, then it would be recorded with two times the penalty *p*.

Before the aggregation starts, the weight values can be set dynamically by users for different types of features. For example, if one user cares more about the color, the weight of the color feature could be set to a larger value than the other features. The total sum of the weights is 1. In this approach, equal weights are set as the default. The pseudocode of the Multi Indexing Aggregation is given in Algorithm 4. On line 1, all feature sets are combined together and the returned results are stored into a new RDD object RDD_{res} preserves both the content and the number of chunks in the old RDDs. On line 1, the *reduceByKey*() action in Spark is used to implement the Result_Reduce() function to merge all the <*key, value*> pairs where the reduction on line 7 only works for the cases that share the same key. This rewarding process removes penalty p from the distance in each pair of indexes.

Utilizing the operations provided by Spark to process the aggregated results, the *filter*() action is performed to remove irrelevant results which appear in fewer number of top *k* results for all feature sets ; *sort*() action is utilized to sort the combined results and *take*(*Nsize*) action returns the top *Nsize* results.

Algorithm 4 Multi Indexing Aggregation process

RDD_{knn} : Sets of top *k* nearest neighbors of *q*
$RDD_{knn}(i)$: The *i*th set of top *k* nearest neighbors
KID: A list of identifiers in $RDD_{knn}(i)$
KDist: A list of distance value in $RDD_{knn}(i)$
N_f : The number of feature files
N_C : The number of chunks
p : The penalty
RDD_{res} : A set of top similarities after aggregation
1: function Result_Reduce($RDD_{knn}(i), RDD_{knn}(i+1), N_f, p$)
2: ID ← $RDD_{knn}(i).KID \cup RDD_{knn}(i+1).KID$
3: $RDD_{temp} \leftarrow \{\}$
4: **for** $id \in ID$ **do**
5: **if** $\exists\, t_j \in RDD_{knn}(i), t_k \in RDD_{knn}(i+1)$ **and** $id == t_j.KID == t_k.KID$ **then**
6: $temp.KID \leftarrow id$
7: $temp.KDist \leftarrow t_j.KDist + t_k.KDist - N_f \cdot p$
8: **else if** $\exists\, t_j \in RDD_{knn}(i)$ **and** $id == t_j.KID$ **then**
9: $temp \leftarrow t_j$
10: **else**
11: $temp \leftarrow t_k$
12: **end if**
13: RDD_{temp}.ADD($temp$)
14: **end for**
15: **return** RDD_{temp}
16: end function
17: $Temp \leftarrow RDD_{knn}(1)$
18: for $i \leftarrow 2$ to $N_f \cdot N_C$ **do**
19 $Temp \leftarrow$ Result_Reduce ($Temp, RDD_{mtree}(i)$)
20: **end for**
21: end for
22: $RDD_{res} \leftarrow Temp$
23: return RDD_{res}

IV EXPERIMENTS AND RESULTS

A series of experiments were designed to evaluate the performance of the proposed system. Experimental results are reported in terms of time consumption, scalability performance and coverage ratio which is defined as:

$$coverage\ ratio = \frac{|Top_{MIA} \cap Top_{BF}|}{|Top_{BF}|} \times 100\%,$$

where Top_{MIA} represents the set of top similar objects acquired by utilizing our system. Top_{BF} refers to the set of most similarities obtained by brute force method without utilizing any indexes or aggregation on the Spark platform as brute force method. An ideal result for a distributed system is to cover all top *k* results from the brute force search.

Two datasets are used in the following experiments: first is a set of 397 pollen images, then a set of 1.4M temporal satellite image chips. Each instance per line in this file represents one pollen image, and every image is characterized by three sets of features: color with 77 dimensions, shape with 67 and texture with 141. The dataset is equally divided into three, six or nine sub files and each sub file is stored into a comma-separated values (CSV) file. Each instance located in the same line in each separated feature file would share the same identifier number. Adopting the approach developed by Cao et al. [30], up to 10 million (10K, 100K, 1M, and 10M) simulated cases were generated based on the 397 instances by adding or removing artificial states. As Spark needs to read data from HDFS to create RDDs, all the original and generated datasets are loaded onto HDFS in the beginning. Meanwhile, HDFS also helps to store the intermediates and final results. The other dataset is the remote sensing image dataset. The total number of remote sensing instances used in this experiment is 1,442,987 with seven sets of features (each with multiple dimensions), namely temporal change (17), entropy (33), skewness (33), maximal length (33), maximal length angle (33), minimal length (33), and minimal length angle (33). The features represent temporal image pairs of

visual change from a large-scale change detection system. Each output is saved into one CSV file, which contains a certain number of ranked results for the given query object. The length of ranked results is defined by users before indexing. Additionally, all the input files need to be stored onto HDFS in the beginning of searches. Intermediate results are saved onto HDFS.

All the experiments were done on two high performance computing clusters ranging from eight to 25 computer nodes with Intel(R) Xeon(R) CPU E5-2680 v4 2.40GHz (28 cores) with 256GB RAM and X5675 CPU 3.07GH (28 cores) with 96 GB RAM. All nodes on both servers share the same physical environment settings.

A. *Runtime for Different Dimensions with Increasing Number of Cases*

Figure 4 shows the average running time excluding the first query, which requires indexing tree loading, for different size of sets of features, with an increasing number of simulated cases for searching for top similarities in 1,000 query instances, by applying 10 compute nodes. Pollen image datasets are divided into three, six and nine sets of features. Retrieval size in each M-tree for *k*-NN Search execution is set as 200 for the case of 10,000, and 2,000 for others. The numbers of instances in this experiment are: 10K, 100K, 1M and 10M.

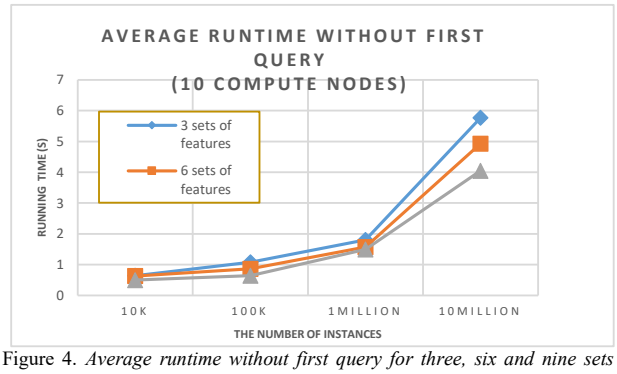

Figure 4. *Average runtime without first query for three, six and nine sets of features, with an increasing number of simulated cases.*

For example, considering the cases for three sets of features, comparing the results of 10K with the results of 10M, even though the storage size of latter one is 1,000 times as large as the size of previous one, the average time without first query only increases approximately 9 times, much less than 1,000. The reason for separating the average running time that includes first query and without first query is that during the first query, all indexing structures are constructed and persisted into the memory across all compute nodes. Thus, the time of constructing indexing structures is included in the average running time if the first query is included.

In Figure 4, it can be observed that as the number of simulated cases increase approximately linearly, all the curves in the chart increase smoothly. Comparing the results of three sets of features with nine sets of features in Figure 4, the difference of average runtime is small for small datasets, such as 10K instances: the average runtime difference is less than 0.15 seconds. However, as the dataset becomes larger – for example, when the size of instances is 10M – the average runtime difference between the two cases increases to approximately 1.8 seconds. Therefore, increasing the number of feature files will reduce the performing time, especially for large datasets.

B. *Comparison of Time Spent with Brute Force Method*

The second experiment compares the performing time between the proposed approach and a brute force method. In this experiment, we used 9 sets of features to repeat the process discussed in Experiment 1 using 26 compute nodes on the Lewis Research Server. For the brute force method, we split the features files into equal number of chunks and distributed these chunks without applying any indexing structures or Multi-Indexing methods. The numbers of cases in this experiment are: 10K, 100K, 1M and 10M.

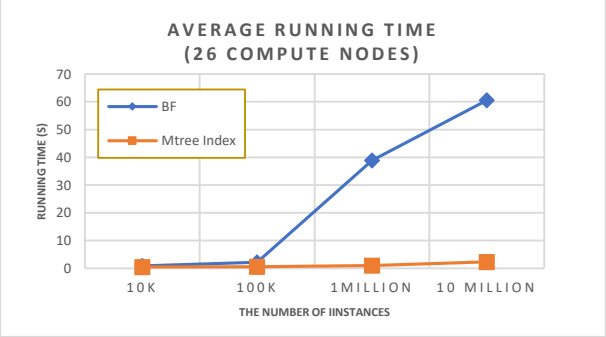

Figure 5. *Comparison of average runtime (without first query) between the proposed approach and brute force method by testing nine sets of features on 26 compute nodes.*

Figure 5 illustrates the results of average running time without first query for our proposed approach and brute force method. Referring to the brute force data compared to this proposed approach in Figure 5, it can be observed that as the number of simulated cases increase, the running time of both methods increases. However, the speed of the brute force method increased much faster than the speed of the indexing structure. For small datasets, such as the 10K case, the time difference between brute force and M-tree structure methods is less than 0.5 seconds. As the dataset becomes larger, the time difference grows as well. For example, when the number of instances is 10M, the runtime of the brute force method is about 26 times greater than that of the M-tree structure. In Figure 5, the time difference between brute force and M-tree methods turns to be quite large from 100K to 10M instances. In the figure, the reason for the line of BF increasing slowly between 10K and 100K is that communication time among multiple machines will play an important role in the total running time.

C. *Scalability Comparison*

A third experiment was designed to observe the running time of searches when the number of compute nodes varies. The number of nodes was varied from 15 to 25, with 100M instances divided into nine sets of features. Figure 6 shows the total running time for the large number of instances. From the chart in Figure 6, we observed that increasing the number of compute nodes (CN) decreases the total running time. For example, when running on 15 compute notes, the average running time without first query is approximate 25 seconds. However, when the number of compute nodes is 25, the running time for the same dataset decreases to 21

seconds, which takes 84% of the running time of the 15-nodes case. The average running time including first query is approximate 26 seconds for 15 compute nodes, but it only takes about 22 seconds when the number of compute nodes is 25, which only takes 85% of the running time of the former. Figure 6 presents the results our compute node scalability tests of runtime. It can be observed that, as expected, for the large-scale dataset, the higher number of compute nodes utilized in our system, the less time the query will take.

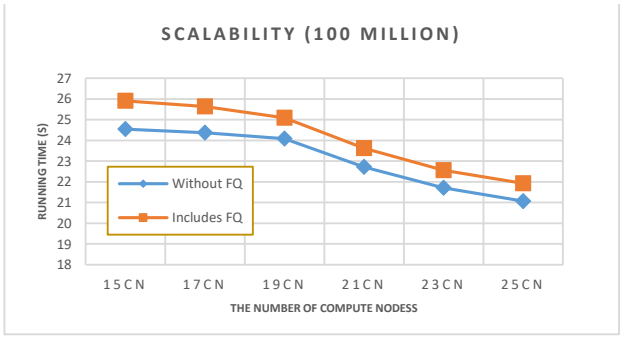

Figure 6. *Comparison of total running time with increasing numbers of compute nodes (CN) from 15 to 25, utilizing 100 million instances with nine sets of features.*

D. Coverage Ratio

The coverage ratio of the searched similarities is evaluated in this experiment by using the remote sensing datasets. Seven sets of features from the remote sensing dataset were applied in this experiment. A brute force method was used to search the similar cases in the remote sensing dataset and then to sort the returned cases in an increasing order of measured distance to the query. Eight compute nodes were used in this experiment.

The separate feature spaces of the remote sensing data set represent complex visual characteristics, necessitating significant buffer sizes during distributed retrieval. On the Spark Server, the user can set the buffering size from 30,000 to 90,000. For this data set, this buffer size covers only 2-6% of the data.

The buffering size indicates the querying size performed in each indexing structure. If the querying size is small, the final aggregated results may have a significantly large size. The number of top similar results, k, means the number of results obtained from the final aggregated outcomes. A smaller top size k in the aggregated results leads to an extremely high coverage ratio. For example, when k is 200 the lowest coverage ratio is obtained with the buffer size of 30,000. However, the ratio is still 91.5%. In contrast, the ratio of the top 2,000 with a buffering size of 30,000 is 55.7%. Increasing the buffering size can improve the coverage ratio for various query sizes.

For example, for k = 1,000, when the buffering size is 30,000, the ratio is 68.2%; when the buffering size is 60,000, the coverage ratio is increased to 99.6%. From the above discussion, it can be expected that if the top size k is quite large, increasing the buffering size would dramatically improve the coverage ratio. Figure 7 shows the analysis of coverage ratio on increasing the buffering size in each index structure. When the buffering size is increased to 50,000 in each M-tree, the top 2,000 ranked results will reach more than 90% coverage ratio. In Figure 7, the horizontal value shows the buffering size for the system and each line is the desired value of k. The vertical shows the coverage ratio value. Figure 8 presents a 3-D perspective of the coverage ratio and shows that for large k numbers (more top-ranked results to show to the users), increasing buffering size will increase the coverage ratio more than it works on smaller k numbers. For the fixed buffering size, smaller number of top ranked results will produce higher coverage ratio.

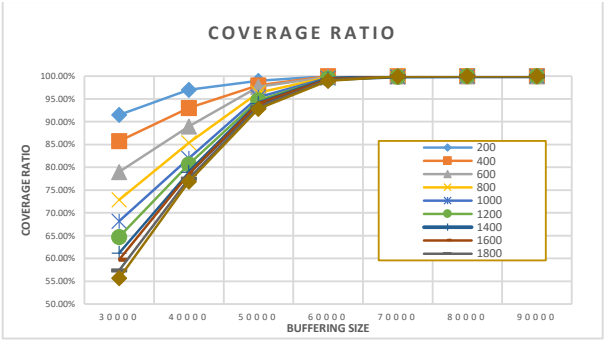

Figure 7. *Comparison of coverage ratio of search for different sizes of top k results, utilizing remote sensing datasets on 8 compute nodes.*

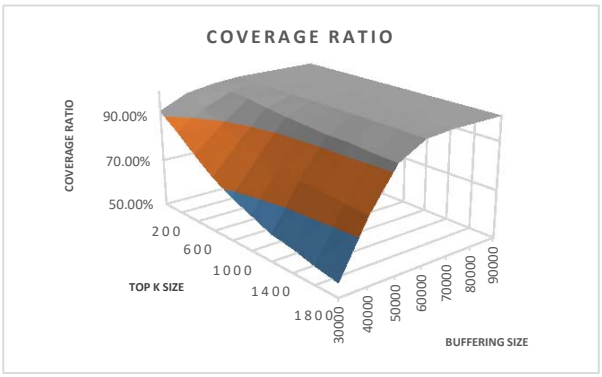

Figure 8. *3-D visualization for coverage ratio of the search.*

V. CONCLUSION

The purpose of developing this system is to utilize Spark to construct a distributed indexing structure for identifying search results with greatest similarity to the query object in multimedia data. This structure provides superior performance in retrieval time compared with both traditional search methods on a single machine and distributed *brute force* searches.

The indexing system reported in this paper demonstrates the following improvements: first, instead of performing on a single machine, our system utilizes the in-memory computing capability of the Spark environment. It can persist the indexing structures into memory across certain numbers of compute nodes, which speeds up the subsequent query access for large scale datasets. Additionally, storing the final and intermediate results onto a Hadoop file distributed system (HDFS) efficiently improves the search performance as well. Second, in order to efficiently aggregate results from multiple indexes, the system applies a distributed version of Multi Indexing Aggregation approach, which decreases the number of shuffling operations between compute nodes. The experimental results offer suggestions for

the multimedia practitioners to evaluate computing resources, desirable number of top-ranked results to display to the users, and search time on various scales of data sets. Moreover, this distributed indexing framework can provide users the capability to dynamically set weights of visual cues which are indexed separately in the distributed environment. In experiments, we have demonstrated that in order to receive results with higher coverage ratio, top *k* should be set smaller. If the top *k* is quite large, indexing buffer size needs to be increased to improve coverage ratio.

In our future work, we will utilize this system to various application domains, where the weighted-ranked results for searching similarities for large-scale datasets are required. Furthermore, additional indexing structures, both tree-based and hash-based, will be further studied.

REFERENCES

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM,* vol. 18, no. 9, pp. 509-517, 1975.

[2] M. Hadjieleftheriou, Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras, "R-Trees – A Dynamic Index Structure for Spatial Searching," in *Encyclopedia of GIS*, S. Shekhar and H. Xiong, Eds. Boston, MA: Springer US, 2008, pp. 993-1002.

[3] J. K. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," *Information processing letters,* vol. 40, no. 4, pp. 175-179, 1991.

[4] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces," in *Proceedings of the 23rd International Conference on Very Large Data Bases,* 1997.

[5] G. Scott and C. R. Shyu, "Knowledge-Driven Multidimensional Indexing Structure for Biomedical Media Database Retrieval," *IEEE Transactions on Information Technology in Biomedicine,* vol. 11, no. 3, pp. 320-331, 2007.

[6] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, 1993, vol. 93, no. 194, pp. 311-21.

[7] T. Bozkaya and M. Ozsoyoglu, "Indexing large metric spaces for similarity search queries," *ACM Transactions on Database Systems (TODS),* vol. 24, no. 3, pp. 361-404, 1999.

[8] F. C. Fleites, S. C. Chen, and K. Chatterjee, "AH+-Tree: An Efficient Multimedia Indexing Structure for Similarity Queries," in *2011 IEEE International Symposium on Multimedia*, 2011, pp. 69-76.

[9] M. Mocofan, I. Ermalai, M. Bucos, M. Onita, and B. Dragulescu, "Supervised tree content based search algorithm for multimedia image databases," in *2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2011, pp. 469-472.

[10] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM,* vol. 51, no. 1, pp. 117-122, 2008.

[11] M. M. Esmaeili, R. K. Ward, and M. Fatourechi, "A Fast Approximate Nearest Neighbor Search Algorithm in the Hamming Space," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 34, no. 12, pp. 2481-2488, 2012.

[12] L. Havasi, D. Varga, and T. Szirányi, "LHI-tree: An efficient disk-based image search application," in *2014 International Workshop on Computational Intelligence for Multimedia Understanding (IWCIM)*, 2014, pp. 1-5.

[13] M. Cedillo-Hernandez, F. J. Garcia-Ugalde, A. Cedillo-Hernandez, M. Nakano-Miyatake, and H. Perez-Meana, "Mexican archaeological image retrieval based on object matching and a local descriptor," in *2015 International Conference on Computer Communication and Informatics (ICCCI)*, 2015, pp. 1-6.

[14] H. Mohamed, St, #233, and p. Marchand-Maillet, "MRO-MPI: MapReduce overlapping using MPI and an optimized data exchange policy," *Parallel Comput.,* vol. 39, no. 12, pp. 851-866, 2013.

[15] R. Ji, L.-Y. Duan, J. Chen, L. Xie, H. Yao, and W. Gao, "Learning to distribute vocabulary indexing for scalable visual search," *IEEE Transactions on Multimedia,* vol. 15, no. 1, pp. 153-166, 2013.

[16] M. Zhong and M. Liu, "A distributed index for efficient parallel top-k keyword search on massive graphs," in *Proceedings of the twelfth international workshop on Web information and data management*, 2012, pp. 27-32: ACM.

[17] X. Zhu, Z. Huang, H. Cheng, J. Cui, and H. T. Shen, "Sparse hashing for fast multimedia search," *ACM Transactions on Information Systems (TOIS),* vol. 31, no. 2, p. 9, 2013.

[18] A. Kaplanis, M. Kendea, S. Sioutas, C. Makris, and G. Tzimas, "HB+tree: use hadoop and HBase even your data isn't that big," in *the Proceedings of the 30th Annual ACM Symposium on Applied Computing*, Salamanca, Spain, 2015.

[19] Z. Chen *et al.*, "A survey of bitmap index compression algorithms for Big Data," *Tsinghua Science and Technology,* vol. 20, no. 1, pp. 100-115, 2015.

[20] D. Yadav, S. Sanchez-Cuadrado, J. Morato, and J. B. L. Morillo, "An approach for spatial search using SOLR," in *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, 2013, pp. 202-208.

[21] E. Malherbe, M. Diaby, M. Cataldi, E. Viennet, and M. A. Aufaure, "Field selection for job categorization and recommendation to social network users," in *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*, 2014, pp. 588-595.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1-10.

[23] H. Wang, B. Xiao, L. Wang, and J. Wu, "Accelerating Large-scale Image Retrieval on Heterogeneous Architectures with Spark," in *the Proceedings of the 23rd ACM international conference on Multimedia,* Brisbane, Australia, 2015.

[24] Y. Duan, L. Jiang, X. Lin, and X. Chen, "An Improved Content Based Image Retrieval System On Apache Spark," in *2016 International Conference on Mechatronics, Control and Automation Engineering (MCAE 2016)* 2016, pp. 107-110.

[25] P. Mack and D. B. Megherbi, "A content-based image retrieval technique with tolerance via multi-page differentiate hashing and binary-tree searching multi-object buckets," in *2016 IEEE International Conference on Computational Intelligence and Virtual Environments for Measurement Systems and Applications (CIVEMSA)*, 2016, pp. 1-6.

[26] F. Wang, X. Wang, W. Cui, X. Xiao, Y. Zhou, and J. Li, "Distributed retrieval for massive remote sensing image metadata on spark," in *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 2016, pp. 5909-5912.

[27] L. Li, D. Taniar, and M. Indrawan-Santiago, "Parallel Search Processing of Tree-Structured Data in a Big Data Environment," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, 2017, pp. 379-386.

[28] D. Li, W. Zhang, S. Shen, and Y. Zhang, "SES-LSH: Shuffle-Efficient Locality Sensitive Hashing for Distributed Similarity Search," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 822-827.

[29] M. N. Klaric, G. J. Scott, and C. R. Shyu, "Multi-Index Multi-Object Content-Based Retrieval," *IEEE Transactions on Geoscience and Remote Sensing,* vol. 50, no. 10, pp. 4036-4049, 2012.

[30] H. Cao, Y. Li, C. M. Allen, M. A. Phinney, and C. R. Shyu, "Visual Reasoning Indexing and Retrieval Using In-memory Computing," in *2016 IEEE Second International Conference on Multimedia Big Data (BigMM)*, 2016, pp. 17-24.