DMTree: A Novel Indexing Method for Finding Similarities in Large Vector Sets

Phuc Do[1]
Faculty of Information Technology
University of Information Technology (UIT)
VNU-HCM
Ho Chi Minh City
Vietnam

Trung Phan Hong[2]
Faculty of Science and Information Technology, University of Information Technology (UIT), VNU-HCM, Faculty of Information Technology, Hoa Sen University
Ho Chi Minh City, Vietnam

Huong Duong To[3]
Faculty of Information Technology
Hoa Sen University
Ho Chi Minh City
Vietnam

***Abstract*—In a vector set, to find similarities we will compute distances from the querying vector to all other vectors. On a large vector set, computing too many distances as above takes a lot of time. So we need to find a way to compute less distance and the MTree structure is the technique we need. The MTree structure is a technique of indexing vector sets based on a defined distance. We can solve effectively the problems of finding similarities by using the MTree structure. However, the MTree structure is built on one computer so the indexing power is limited. Today, large vector sets, not fit in one computer, are more and more. The MTree structure failed to index these large vector sets. Therefore, in this work, we present a novel indexing method, extended from the MTree structure, that can index large vector sets. Besides, we also perform experiments to prove the performance of this novel method.**

Keywords—MTree; DMTree; spark; distributed k-NN query; distributed range query

I. Introduction

In the real world, graph applications appear everywhere and graphs get bigger and bigger. Graph with millions, billions of vertices are very popular. However, applying of mathematical calculations and machine learning algorithms on graphs is limited and very difficult. Therefore, a new and promising graph processing technique, which is widely interested in research circles, is the graph embedding technique [1]–[3]. The graph embedding technology is developed based on word2vec technology [4][5]. Word2vec is a technology of mapping words to vectors. This technology has helped solve a series of Natural Language Processing problems with much greater accuracy than before. In the graph embedding, each vertex of a graph is mapped to a vector with 64, 128, 256... dimensions, each dimension is a real number. In order to preserve as much information as possible in the original graph, the greater is the number of dimensions of the vector set. Today, large graphs are very popular, so large vector sets are also very popular.

On the other hand, to find similarities in vector sets, we must compute distances from the querying vector to all other vectors. In a large vector set, we cannot compute too many distances as above. Through the research process, we realize that the MTree structure is a technique of indexing vector sets based on a defined distance. Using the MTree structure, we can solve the problems of finding similarities effectively [6]. Since the MTree structure is only built on one computer, it can only index small vector sets, where all vectors can store into one computer. Today, large vector sets, where vectors are distributed in a computer cluster, are increasingly popular. The MTree structure fails to index these large vector sets. That is why in this work we build the distributed MTree structure (DMTree for short) by extending the MTree structure on Spark, a famous framework for distributed processing, for indexing large vector sets. Besides, we also perform experiments on both structures to prove that the performance of the DMTree structure is better than that of the MTree structure.

The main contributions of our paper are as the following:

- Proposing a method to build the DMTree structure to index large vector sets.
- Using the DMTree structure for finding similarities in large vector sets.
- Presenting experiments to prove that the DMTree structure is better than the MTree structure.

There are six sections in this paper, including: I. Introduction, II. Related works, III. Preliminaries, IV. Methodology, V. Experiments, and VI. Conclusion and future works.

II. Related Works

Inspired by [6], there are many works on implementing, developing the MTree structure and other indexing structures.

Author in [7] has proposed the MVPTree structure (the multi-vantage point tree structure) to partition vector sets. Experiments has performed to compare the MVPTree structure and the MTree structure.

Author in [8] has built a framework for finding similarities, where data is indexed locally by using the MTree structure. This work has used a super-peer architecture, where super-peers are responsible for query routing, for supporting scalability and efficiency of finding similarities.

Author in [9] has researched a tree structure for indexing and querying data based on a metric. This work has also performed experiments to compare its method with others, such as the MMTree structure and the SliMTree structure. This work has proposed a classification of indexing methods also.

And most recently, [10] has built the SuperMTree structure by extending the MTree structure for indexing vector sets. This work has proposed a generalized concept of metric spaces that is metric subset spaces. Various metric distance functions can be extended to metric subset distance functions.

Most of the previous works do not refer to indexing large vector sets, where vectors are distributed in a computer cluster, except [8]. However, [8] has built a new framework for finding similarities in a distributed manner. In this work, we will not create any framework, we only use Spark, which is the famous framework for distributed processing, to build the DMTree structure for indexing large vector sets effectively.

III. Preliminaries

A. Similarity Query Definitions

In vector sets, the common task is finding similarities. Specifically, we usually find k vectors closest to the querying vector; or find all vectors in the range of radius r, and the center is the querying vector. Those are the k-nearest neighbors query (k-NN query for short) and the range query [6][11]. The following are definitions of them.

1) Definition 1. K-nearest neighbors query: Given a vector set S, a distance function d, a querying vector $v \in S$, and an integer number $k \geq 1$. The k-nearest neighbors query $kNNQuery(v, k)$ selects k vectors in S that are closest to v.

2) Definition 2. Range query: Given a vector set S, a distance function d, a querying vector $v \in S$, and a radius r. The range query $rangeQuery(v, r)$ selects all vectors v_i in S such that $d(v_i, v) \leq r$.

B. The MTree Structure

The MTree structure is a technique of indexing vector sets based on a defined distance function [6][13]. In terms of internal structure, it is a balanced tree but not require periodic rebuilding.

A node of the MTree structure can contain at most C objects, C is called the capacity of nodes. The leaf nodes contain indexed objects. The rest nodes contain routing objects.

The format of a routing object is as follows:

$$[\, O_r, r_r, d(O_r, O_r^p), ptr(T_r) \,]$$

Where O_r represents the routing object; $r_r \geq 0$ is the covering radius of O_r; $d(O_r, O_r^p)$ is the distance between O_r and O_r^p which is the parent object of O_r; $ptr(T_r)$ is the reference of subtree T_r which is the covering tree of O_r.

The format of an indexed object O_i is as follows:

$$[\, O_i, d(O_i, O_i^p) \,]$$

Where O_i represents the indexed object, $d(O_i, O_i^p)$ is the distance between O_i and O_i^p which is the parent object of O_i.

Fig. 1 is an instance of the MTree structure with C = 3. This MTree structure includes:

- Node 1 is the root node which contains two routing object O_1 and O_5.
- Node 2 and node 3 are the internal nodes. The internal nodes contain routing objects. For example, node 2 contains two routing objects are O_1 and O_8. In node 2, consider the routing object O_8, $[\, O_8, 3.5, 1.3, ptr(T_r) \,]$; 3.5 is the covering radius of O_8; 1.3 is the distance between O_8 and O_1 (in node 1) which is the parent object of O_8.
- Node 4, 5, 6, 7, 8 are the leaf nodes. The leaf nodes contain indexed objects. For example, leaf node 4 contains three indexed objects are O_1, O_2 and O_4. In node 4, consider the indexed object O_2, $[\, O_2, 2.5]$; 2.5 is the distance between O_2 and O_1 (in node 2) which is the parent object of O_2.

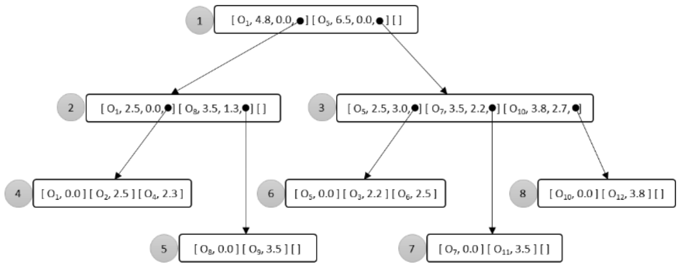
1
[O1, 4.8, 0.0, ●] [O5, 6.5, 0.0, ●] [ ]
2
[O1, 2.5, 0.0, ●] [O8, 3.5, 1.3, ●] [ ]
3
[O5, 2.5, 3.0, ●] [O7, 3.5, 2.2, ●] [O10, 3.8, 2.7, ●]
4
[O1, 0.0] [O2, 2.5] [O4, 2.3]
5
[O8, 0.0] [O9, 3.5] [ ]
6
[O5, 0.0] [O3, 2.2] [O6, 2.5]
7
[O7, 0.0] [O11, 3.5] [ ]
8
[O10, 0.0] [O12, 3.8] [ ]

Fig. 1. An Instance of the MTree Structure with C = 3.

Please refer [6][13] for more information about the MTree structure.

IV. Methodology

At first, we build the MTree structure running on one computer. After that, we extend the MTree structure to build the DMTree structure running on a Spark cluster consisting of multiple computers. We also perform many experiments on both structures to prove that the DMTree structure is better than the MTree structure. The following are details of our solution.

A. Building a DMTree Object

At first, we create the MTree class to represent the MTree structure based on [6][13]. The MTree class has the following important methods:

- function insertObject(n: Node, v: Vector): locates the most suitable leaf node in the subtree of node n to store a new vector v. It is possible to trigger splitting the leaf node if the leaf node is full. This method is used to build an MTree object from a vector set.
- function kNNQuery(v: Vector, k: Integer): Set[Vector]: executes a k-NN query and return k vectors that are closest to v.
- function rangeQuery(v: Vector, r: Double): Set[Vector]: executes a range query and returns all vectors such that the distances from v to them less than or equal to r.

Please refer [6][13] for more details of these methods.

Next, we build the DMTree structure by defining the DMTree class based on the MTree class. A DMTree object includes a set of MTree objects built from a vector set. The vector set is distributed in partitions of an RDD (Resilient Distributed Dataset), which is a fundamental data structure of Apache Spark and is a fault-tolerant collection of elements that can be operated on in parallel [14][15]. Properties and methods of the DMTree class are shown in TABLE I. and Table II.

The process of building a DMTree object from a vector set is shown in Fig. 2. First, a vector set is loaded from distributed files into an RDD[Vector] object in a Spark cluster [14]. Second, a DMTree object is created. Then, using the Map transformation, which transforms an RDD[X] object to another RDD[Y] object in parallel (suppose that X and Y are data types), maps each partition of the RDD[Vector] object to an MTree object inside the DMTree object. The number of partitions of an RDD[Vector] object can be configured, so is the number of MTree objects of a DMTree object. The process of building a DMTree object from a vector set is described in Algorithm 1.

TABLE I. PROPERTIES OF THE DMTREE CLASS

Properties	**Data Types**	**Descriptions**
C	Integer	The capacity of DMTree objects. In essence, this property is the capacity of nodes of MTree objects inside a DMTree object.
mtrees	RDD[MTree]	Containing an MTree object set inside a DMTree object.

TABLE II. METHODS OF THE DMTREE CLASS

Methods	**Descriptions**
function build(path: String, C: Integer): DMTree	
	Building a DMTree object with capacity C from distributed files that contain a vector set.
function store(path: String, dmtree: DMTree)	
	Storing a DMTree object into distributed files.
function rebuild(path: String): DMTree	
	Rebuilding a DMTree object from distributed files that contain the DMTree object.
function kNNQuery(v: Vector, k: Integer): Set[Vector]	
	Executing a distributed k-NN query.
function rangeQuery(v: Vector, r: Double): Set[Vector]	
	Executing a distributed range query.

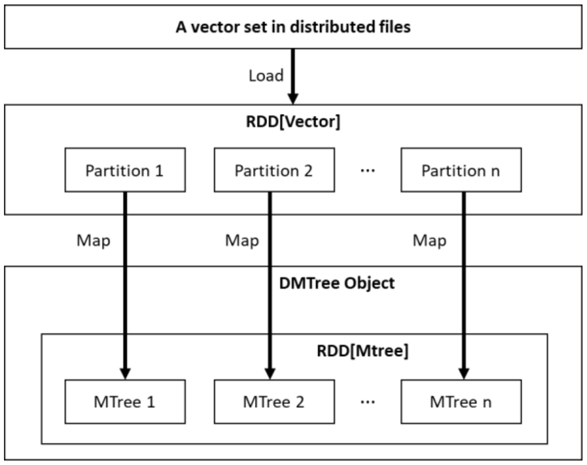

Fig. 2. The Process of Building a DMTree Object from a Vector Set.

Algorithm 1. Building a DMTree Object.

- Function: building a DMTree object from a vector set in distributed files.

- Input: 1) path: the path of the distributed files that contain the vector set. 2) C: the capacity of the DMTree object.

- Output: the DMTree object.

```
1.  function build(path: String, C: Integer): DMTree{
2.      load the vector set from the distributed files into an
    RDD[Vector] object.
3.      create a new empty DMTree object with capacity C.
4.      map each partion of the RDD[Vector] object to an MTree
    object in the RDD[MTree] object inside the DMTree object.
5.      return the DMTree object.
6.  }
```

B. Storing a DMTree Object into Distributed Files

Because creating DMTree objects is quite time consuming, we will store DMTree objects for reuse later. A DMTree object can be very large, exceeding the capacity of a file in a local file system; so it must be stored in distributed files. Storing a DMTree object into distributed files is described in Algorithm 2.

Algorithm 2. Storing a DMTree Object.

- Function: storing a DMTree object into distributed files.

- Input: 1) path: the path of the distributed files. 2) dmtree: the DMTree object will be stored in the distributed files.

- Output: none.

```
1.  function store(path: String, dmtree: DMTree){
2.      store metadata (the capacity, number of MTree objects…) of
    the DMTree object.
3.      map each MTree object in the RDD[MTree] object inside the
    DMTree object to a distributed file.
4.  }
```

C. Rebuilding a DMTree object from Distributed Files

When reusing a DMTree object previously stored, it can be rebuilt from distributed files. Since a DMTree object includes an MTree object set, each executor should load at least one MTree object for effective processing. Therefore, the number of MTree objects inside a DMTree object should be a multiple of the number of executors. The process of rebuilding a DMTree object from distributed files in a Spark cluster is described in Algorithm 3.

Algorithm 3. Rebuilding a DMTree Object.

- Function: rebuilding a DMTree object in a Spark cluster from distributed files that contain the DMTree object.

- Input: 1) path: the path of the distributed files that contain the DMTree object.

- Output: the DMTree object.

```
1.  function rebuild(path: String): DMTree{
2.      load metadata (the capacity, number of MTree objects…) of
    the DMTree object from the distributed file containing metadata.
3.      create a new empty DMTree object with the capacity loaded.
4.      load MTree objects from the distributed files into the
    RDD[MTree] object inside the DMTree object.
5.      return the DMTree object.
6.  }
```

D. Executing a Distributed k-NN Query

A distributed k-NN query is a k-NN query running on a Spark cluster based on a DMTree object. The process of executing a distributed k-NN query is shown in Fig. 3.

First, the driver program in the master node invokes the kNNQuery(v, k) method on the DMTree object. This method uses the Map transformation to map each MTree object in the RDD[MTree] object inside the DMTree object to a Set[Vector] object, which is the result of invoking the kNNQuery(v, k) method on the MTree object.

Next, the driver program collects all Set[Vector] objects from the worker nodes to the master node and unions them to the final Set[Vector] object by using a Reduce action. On Spark, the Reduce action is an action that collects data of an RDD object from the worker nodes into the master node and performs a function (such as sum, min, max, union…) on the collected data set.

The final Set[Vector] object is sorted by distances in ascending order. The final result is the top k vectors are extracted from the final Set[Vector] object.

The process of executing a distributed k-NN query is described in Algorithm 4.

Algorithm 4. Executing a distributed k-NN query.

- Function: executing a distributed k-NN query on a DMTree object.

- Input: 1) v: the querying vector. 2) k: the number of nearest neighbors.

- Output: at most k vectors that are closest to v.

```
1.  function kNNQuery(v: Vector, k: Integer): Set[Vector]{
2.      map each MTree object in the RDD[MTree] object inside
    the DMTree object to a Set[Vector] object, which is the result of
    invoking the kNNQuery(v, r) method on the MTree object.
3.      collect all Set[Vector] objects from the worker nodes.
4.      union all Set[Vector] objects to the final Set[Vector] object.
5.      sort the final Set[Vector] object by distances in ascending
    order.
6.      return the top k vectors in the final Set[Vector].
7.  }
```

The distributed k-NN query problem is solved by Algorithm 4 successfully, but there is still a drawback that needs further improvement. That is, each MTree object returns at most k vectors to the master node, n MTree objects will return at most $(n \times k)$ vectors to the master node. Because the master node only extracts the top k vectors, there are a lot of vectors received by the master node but not used. This increases the data traffic transferred from the worker nodes to the master node, decreasing the performance of the algorithm. However, overcoming this weakness is not easy. We need to research further in the future.

E. Executing a Distributed Range Query

A distributed range query is a range query running on a Spark cluster based on a DMTree object. The process of executing a distributed range query is shown in Fig. 4.

First, the driver program in the master node invokes the rangeQuery(v, r) method on the DMTree object. This method uses the Map transformation to map each MTree object in the RDD[MTree] object inside the DMTree object to a Set[Vector] object, which is the result of invoking the rangeQuery(v, r) method on the MTree object.

Next, the driver program collects all Set[Vector] objects from the worker nodes to the master node and unions them to the final Set[Vector] object by using a Reduce action.

The final Set[Vector] object is sorted by distances in ascending order for ease of observation. The final result is the final Set[Vector] object.

The process of executing a distributed range query is described in Algorithm 5.

Algorithm 5. Executing a distributed range query.

- Function: executing a distributed range query on a DMTree object.

- Input: 1) v: the querying vector. 2) r: the radius.

- Output: a set of vectors such that the distances from v to them less than or equal to r.

```
1.  function rangeQuery(v: Vector, r: Double): Set[Vector]{
2.      map each MTree object in the RDD[MTree] object inside the
    DMTree object to a Set[Vector] object, which is the result of invoking
    the rangeQuery(v, r) method on the MTree object.
3.      collect all Set[Vector] objects from the worker nodes.
4.      union all Set[Vector] objects to the final Set[Vector] object.
5.      sort the final Set[Vector] object by distances in ascending order.
6.      return the final Set[Vector].
7.  }
```

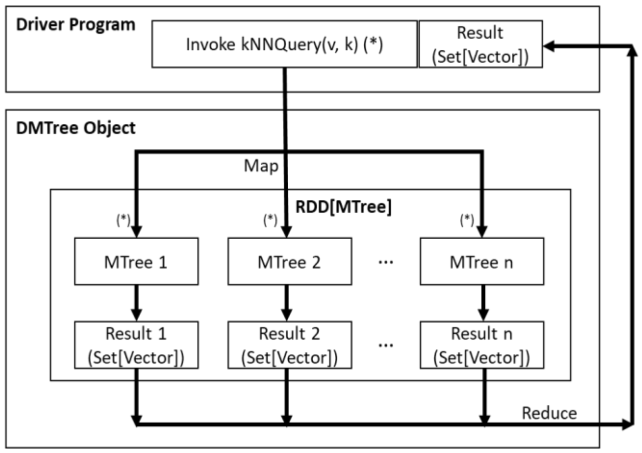

Fig. 3. The Process of Executing a Distributed k-NN Query.

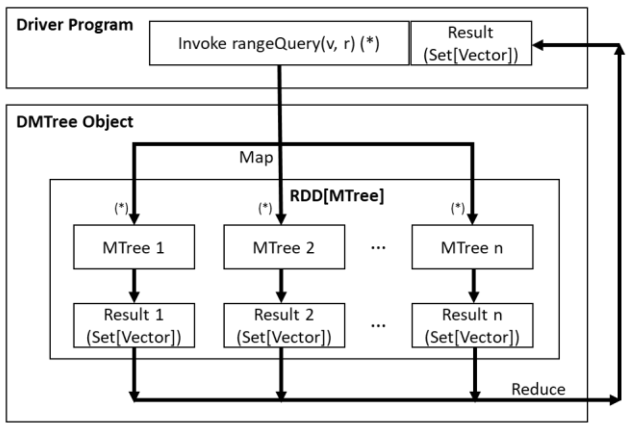

Fig. 4. The Process of Executing a Distributed Range Query.

V. Experiments

In this section, we show the experimental results of MTree and DMTree objects. We use Yago Knowledge Base [16], downloaded from the Max Planck Institute for Informatics website [12], to make data for experiments. At first, we build knowledge graphs from downloaded triples, then create 64-dimensional vector sets from the knowledge graphs by using the graph embedding technique. The following are details of the experiments.

A. Experiments on MTree Objects

In order to perform experiments on the MTree objects, we use one computer with configuration as the following:

- Processor: Intel(R) Core™ i5-6500 CPU @ 3.20GHz 3.20GHz
- RAM: 16.0 GB

We perform experiments on 64-dimensional vector sets. We create MTree objects with C = 1,000 from these vector sets. Table III shows experimental results (in seconds) of 4 functions:

- Building and Storing MTrees: includes building MTree objects in computer memory from vector sets stored in local files and storing the MTree objects into local files for reusing later.
- Rebuilding MTrees: rebuilds MTree objects from local files.
- Executing k-NN Queries: executes k-NN queries on MTree objects.
- Executing Range Queries: executes range queries on MTree objects.

We only conduct experiments of up to 2,000,000 vectors because if we experiment on larger vector sets that will exceed the capabilities of our computer.

Fig. 5 shows the chart that compares the time (in seconds) of building + storing and rebuilding MTree objects. This chart demonstrates that building and storing MTree objects to local files is slower than rebuilding MTree objects from local files into memory of the computer. Specifically, building and storing the MTree object from a vector set of 2 million vectors takes 128,10 seconds, on the contrary, rebuilding it takes 76,86 seconds.

Fig. 6 shows the chart that compares the time to execute k-NN and range queries on the MTree objects. This chart demonstrates that executing queries is quite fast and k-NN queries are always slower than range queries. Specifically, executing a k-NN query on the MTree object of 2 million vectors takes 7.82 seconds, while executing a range query on the same MTree object takes 5.65 seconds only.

B. Experiments on DMTree Objects

In order to perform experiments on DMTree objects, we built a Spark cluster includes 16 computers. Where one computer is both a master node and a worker node (For simplicity, we refer this computer as the master node), and fifteen computers work as worker node only (Similarly, we refer these computers as the worker nodes). The configuration of the master node as the following:

- Processor: Intel(R) Core™ i5-6500 CPU @ 3.20GHz 3.20GHz
- RAM: 16.0 GB

And the configuration of the worker nodes as the following:

- Processor: Intel(R) Core™ i5-6500 CPU @ 3.20GHz 3.20GHz
- RAM: 8.0 GB

TABLE III. Experimental Results (in Seconds) on MTree Objects

Vector Sets	Building and Storing MTrees	Rebuilding MTrees	Executing k-NN Queries	Executing Range Queries
0.1 m	56.68	34.01	0.26	0.27
0.2 m	58.09	34.85	0.53	0.48
0.3 m	60.52	36.31	0.99	0.76
0.4 m	65.13	40.68	1.34	1.02
0.5 m	69.98	44.39	1.71	1.34
0.6 m	75.09	45.05	1.88	1.63
0.7 m	77.78	46.67	2.36	1.87
0.8 m	81.06	48.64	2.50	2.16
0.9 m	86.91	52.14	2.85	2.46
1.0 m	91.84	55.10	3.19	2.81
1.1 m	97.35	58.41	3.84	3.00
1.2 m	102.28	61.37	3.86	3.29
1.3 m	104.05	63.83	4.57	3.63
1.4 m	110.41	66.24	4.62	3.90
1.5 m	112.59	67.56	5.61	4.19
1.6 m	114.30	68.58	5.57	4.62
1.7 m	116.53	69.92	5.76	4.83
1.8 m	118.34	71.00	6.22	5.23
1.9 m	123.27	73.96	6.62	5.54
2.0 m	128.10	76.86	7.82	5.65

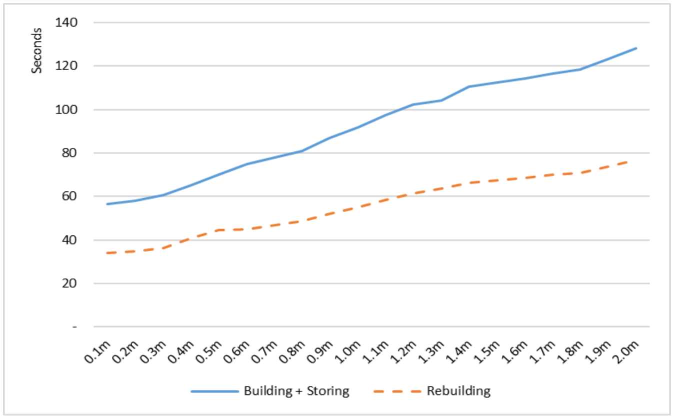

Fig. 5. The Comparison of the Time (in Seconds) of Building + Storing and Rebuilding MTree Objects.

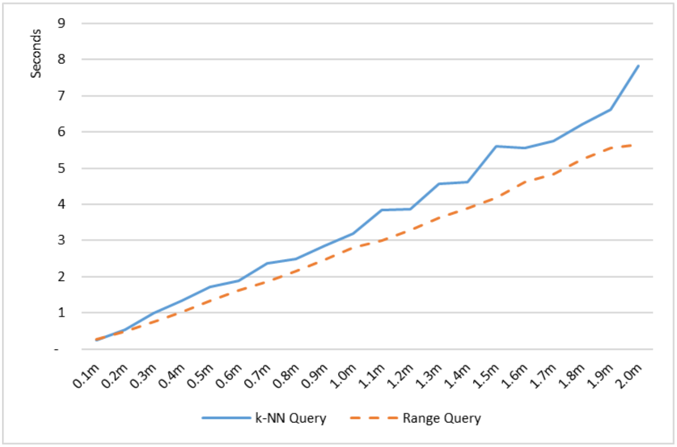

Fig. 6. The Comparison of the Time (in Seconds) of Executing k-NN and Range Queries on the MTree Objects.

Fig. 7 shows the architecture of our Spark cluster. Where:

- Master Node: is a computer running main programs, sending code to the worker nodes to execute in parallel, and collecting the results.
- Worker Node: is a computer participating in processing requests of the master node.
- Cluster Manager: is a component allocating resources across applications.

The software installed on our Spark cluster as showed in Table IV.

Similar to the experiments on MTree objects, we perform experiments on 64-dimensional vector sets also. We create DMTree objects with C = 1,000 from these vector sets. Table V shows experimental results (in seconds) of four functions:

- Building and Storing DMTrees: includes building DMTree objects in the Spark cluster from vector sets stored in distributed files and storing the DMTree objects into distributed files for reusing later.
- Rebuilding DMTrees: rebuilds DMTree objects from distributed files in the Spark cluster.
- Executing Distributed k-NN Queries: executes k-NN queries on DMTree objects.
- Executing Distributed Range Queries: executes range queries on DMTree objects.

Fig. 8 is the chart that compares the time (in seconds) of building + storing and rebuilding DMTree objects. This chart demonstrates that building and storing DMTree objects is quite slow. However, loading DMTree objects from HDFS into the Spark cluster is much faster. Specifically, creating and storing the DMTree object from a vector set of 6 million vectors takes 98.06 seconds, while rebuilding this DMTree object takes 25.88 seconds only.

Fig. 9 is a chart comparing the time to execute distributed k-NN queries and distributed range queries on DMTree objects. This chart demonstrates that the execution of queries is quite fast, and distributed k-NN queries are always slower than distributed range queries. Specifically, executing a distributed k-NN query on the DMTree object of 6 million vectors takes 6.48 seconds, while executing a distributed range query on this DMTree object only takes 5.19 seconds.

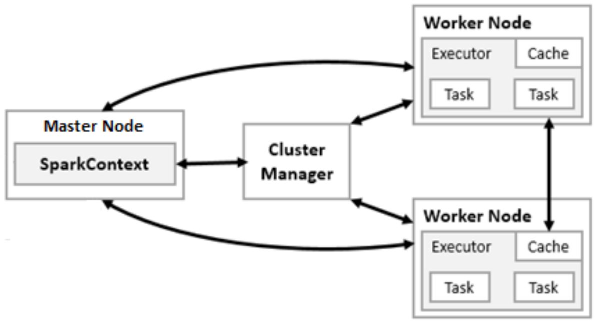

Fig. 7. The Architecture of our Spark Cluster.

TABLE IV. SOFTWARE IS INSTALLED ON OUR SPARK CLUSTER

Software	Version
Operating System	Ubuntu 18.04
Java	OpenJDK version 1.8.0_222
Scala	Version 2.11.12
Apache Hadoop	Apache Hadoop 2.8.5
Apache Spark	Apache Spark 2.4.4

TABLE V. EXPERIMENTAL RESULTS (IN SECONDS) ON DMTREE OBJECTS

Vector Sets	Building and Storing DMTrees	Rebuilding DMTrees	Executing Distributed k-NN Queries	Executing Distributed Range Queries
1 m	18.38	2.66	2.11	1.27
2 m	21.92	5.59	2.84	1.65
3 m	28.25	9.09	3.72	2.74
4 m	41.21	12.00	4.77	3.22
5 m	69.95	18.25	5.12	3.85
6 m	98.06	25.88	6.48	5.19

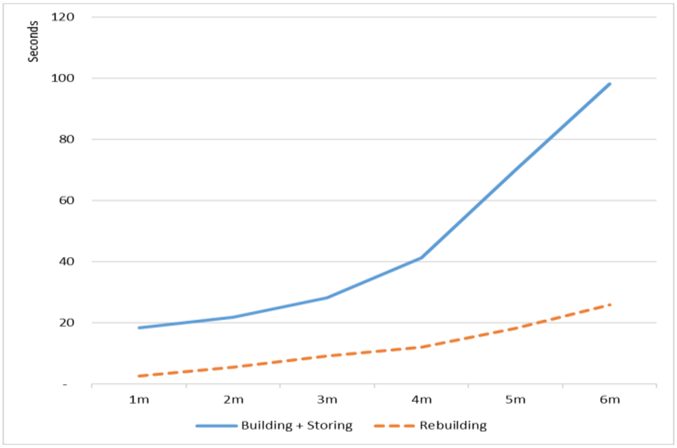

Fig. 8. The Time Comparison of Creating + Storing and Loading DMTree Objects.

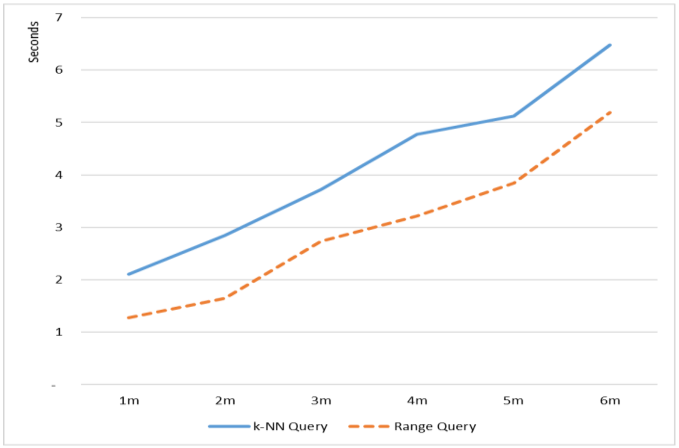

Fig. 9. The Time Comparison of Executing Distributed Queries on DMTree Objects.

It is also important to note that if we use MTree objects on one machine, we can only index vector sets of up to 2 million vectors; but when using DMTree objects on the Spark cluster, we can index much larger vector sets. This proves the limitations of the MTree structure and outstanding ability of the DMTree structure.

VI. Conclusion and Future Works

The MTree structure is a technique of indexing vector sets. We can solve effectively the problems of finding similarities (for example k-NN and range queries) in vector sets by using the MTree structure. However, for large vector sets, vectors are distributed across multiple computers, the MTree structure fails to index them. In order to overcome this drawback of the MTree structure, we extended the MTree structure to build the DMTree structure on the Spark cluster. We also perform experiments on both the structures to prove that the performance of the DMTree structure is better than that of the MTree structure.

Through the research process, we draw some advantages and disadvantages of the DMTree structure as follows.

Advantages:

- Take time to create the DMTree structure once, but it can be reused many times.
- Allow adding or removing entries to/from the DMTree structure without having to rebuild it.
- Help executing the k-NN query and the range query in large vector sets effectively.

Disadvantages:

- Implementing the DMTree structure is quite complicated.
- The MTree structure contains data in nodes that make the size of the MTree structure quite large, resulting in a large size of the DMTree structure.
- Finding similarities in large vector sets is still a bit slow due to the large cost of communication between the master node and the worker nodes.

In the near future, we will continue our research to reduce the DMTree structure and decrease communication costs between the master node and the worker nodes for improving the performance of finding similarities in large vector sets.

Acknowledgment

This research is funded by Vietnam National University Ho Chi Minh City (VNU-HCMC) under the grant number DS2020-26-01.

References

[1] P. Goyal and E. Ferrara, “Graph embedding techniques, applications, and performance: A survey,” Knowledge-Based Syst., 2018.

[2] H. Cai, V. W. Zheng, and K. C. C. Chang, “A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications,” IEEE Trans. Knowl. Data Eng., 2018.

[3] A. Grover and J. Leskovec, “Node2vec: Scalable feature learning for networks,” Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min., vol. 13-17-Augu, pp. 855–864, 2016.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector set,” ArXiv, pp. 1–12, 2013.

[5] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” 31st Int. Conf. Mach. Learn. ICML 2014, vol. 4, pp. 2931–2939, 2014.

[6] P. Ciaccia, M. Patella, and P. Zezula, “MTree: An efficient access method for similarity search in metric spaces,” in Proceedings of the 23rd International Conference on Very Large Databases, VLDB 1997, 1997.

[7] T. Bozkaya and M. Ozsoyoglu, “Indexing Large Metric Spaces for Similarity Search Queries,” ACM Trans. Database Syst., 1999.

[8] A. Vlachou, C. Doulkeridis, and Y. Kotidis, “Peer-to-peer similarity search based on MTree indexing,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2010.

[9] Z. Kouahla, “Exploring intersection trees for indexing metric spaces,” in CEUR Workshop Proceedings, 2011.

[10] J. P. Bachmann, “The SuperMTree: Indexing metric spaces with sized objects,” ArXiv, pp. 1–14, 2019.

[11] P. Zezula, G. Amato, V. Dohnal, and M. Batko, “Similarity Search: The Metric Space Approach,” Springer. 2006.

[12] “YAGO Homepage.” [Online]. Available: https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago. [Accessed: 21-Jan-2019].

[13] “MTree Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/MTree. [Accessed: 15-Mar-2019].

[14] P. Zecevic and M. Bonaci, Spark in Action. Manning Publications Co, 2017.

[15] R. Kienzler, Mastering Apache Spark 2.x : Scalable analytics faster than ever, vol. 22, no. S1. 2017.

[16] “YAGO (database) Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/YAGO_(database). [Accessed: 21-Jan-2019].