M^+-tree: A New Dynamical Multidimensional Index for Metric Spaces

Xiangmin Zhou[1], Guoren Wang[1], Jeffrey Xu Yu[2], Ge Yu[1]

[1] Northeastern University, Shenyang, China
{wanggr, yuge}@mail.neu.edu.cn
[2] The Chinese University of Hong Kong, Hong Kong, China
yu@se.cuhk.edu.hk

Abstract

In this paper, we propose a new metric index, called M^+-tree, which is a tree dynamically organized for large datasets in metric spaces. The proposed M^+-tree takes full advantages of M-tree and MVP-tree, with a new concept called *key dimension*, which effectively reduces response time for similarity search. The main idea behind the key dimension is to make the fanout of tree larger by partitioning a subspace further into two subspaces, called twin-nodes. We can double the filtering effectiveness by utilizing the twin-nodes. In addition, for the purpose of ensuring high space utilization, we also conduct data reallocation between the twin nodes dynamically. Our experiment shows that higher filtering efficiency can be obtained by using the key dimensions for r-neighbor search and k-NN (k-nearest neighbor). We will report our experimental results in this paper.

Keywords: Multidimensional index, Metric space, Key dimension, Range search, k-NN search.

1 Introduction

Recently, an incommensurable amount of audiovisual information becomes available in digital libraries, digital archives, personal and professional databases, the World Wide Web, and broadcast data streams. Besides, the data bulk continues to grow rapidly. A wide range of applications including image processing, geography system, medical applications and biomedicine, etc., highly demand fast processing content-based similarity search in a very large databases.

In order to respond such requests, there exist a large number of multidimensional indexes. As referred in the works (N.Berkmann, H.-P. Krigel. R. Schneider, and B.Seeger.1990, N.Katayama and S.Satoh. 1997, D.A.White and R.Jain.1996, K.-I. Lin,H. V. Jagadish, and C. Faloutsos.1994), R-tree and its variants are widely used in geographical information systems. But they cannot be directly applicable to handle large datasets in metric spaces. Metric-based indices have been proposed for a generic metric space, including VP-tree (J. K. Uhlmann.1991), MVP-tree (T.Bozkaya, M.Ozsoyoglu. 1997), M-tree(P.Zezula, P.Ciaccia, and F.Rabitti.1996, P.Ciaccia, M.Patella, P.Zezula.1997) and MB+tree (M. Ishikawa, H. chen, K, Furuse, J.Xu Yu, N.Ohbo.2000). These index structures are different from the R^*-tree and its other variants. They do not deal with the relative positions in a vector space, but rather handle the distances between objects. VP-tree (J. K. Uhlmann.1991) is designed with a hierarchical index structure for similarity search. It partitions a data set according to distances the objects have with respect to a vantage point. The median value of such distances is used as a separator to partition objects into two balanced subsets. At the same time, the same procedure is applied recursively. VP-tree is the first one among the metric-based indices that utilizes the triangle inequality to filter and reduce the similarity search cost for multimedia information systems. However, due to small fanout, VP-tree is very high thus a search operation needs a large number of distance calculations, which is time-consuming.

MVP-tree extends the idea of VP-tree by using *multiple vantage points*, and exploits pre-computed distances to reduce the number of distance computations at query time. In comparison with VP-tree, the fanout of MVP-tree is increased, and the height is reduced. MVP-tree outperforms VP-tree, but, as the same as VP-tree, MVP-tree is built from top to bottom. The top-down index construction strategy implies that the index is static, and cannot be dynamically updated according to database changes. The cost of reconstructing the whole index frequently becomes unacceptable for the database that may possibly change frequently.

Unlike VP-tree and MVP-tree, M-tree is a paged and balanced metric tree that is built from bottom to top, with node promotion and split mechanisms. M-tree can handle reconstruct the tree dynamically with low costs. M-tree takes the complexity of distance computation into account, and is a very efficient index. But M-tree's subspaces overlap is considerable large, which affects its performance.

Practically, data of multimedia databases are often in metric spaces. But, almost all of spatial access methods (SAMs), e.g. R-tree and its variants, are not applicable to multimedia database. They are valid only when the following conditions are satisfied: (1) indexed objects are represented as feature values in a multidimensional vector space; (2) the similarity of two objects is measured by Euclidean distance. However, metric trees such as MVP-tree and M-tree circumvent these problems. Therefore, they have a higher practical value.

Because M-tree is one of the best among metric-based indices, this paper proposes a new metric-based index, called M^+-tree. It improves the performance of M-tree. First, like M-tree, M^+-tree is a dynamical paged and balance tree. It inherits M-tree's promotion mechanism, triangle inequality and the branch and bound technique. Second, M^+-tree fully utilizes the filtering twice idea used in MVP-tree. Third, M^+-tree adopts the similar ideas of key dimension and the key dimension shift used in TV-tree (K.-I. Lin,H. V. Jagadish, and C. Faloutsos. 1994) in a novel way, based on our

observations: a) dimension can be ordered by their significance in a metric-space, and b) the active dimensions can be shift for enhancing the efficiency. The key dimension and the shift of the *key dimension* reduce the distance computation significantly.

The remainder of the paper is structured as follows. Section 2 gives the problem definition. Section 3 introduces M^+-tree, containing its data structures, key techniques and basic algorithms. Section 4 presents experimental results and performance evaluations. Section 5 concludes this paper.

2 Problem Definition

A metric space, M, is defined as, M = (O, d), where O is the domain of feature values and d is the distance function with the following properties.

1. $d(O_x, O_y) = d(O_y, O_x)$
2. $d(O_x, O_y) > 0$ ($O_x \neq O_y$), $d(O_x, O_x) = 0$
3. $d(O_x,O_y) \leqslant d(O_x, O_z) + d(O_z, O_y)$

where O_x, O_y and O_z are objects in O. The (dis)similarity between objects can be measured based on the given distance function d. R-neighbor search and k-nearest neighbor search are two basic types of similarity queries, defined as follows.

Definition 2.1 (r-neighbor search) Given a query object $q \in O$ and a non-negative query radius r, the r-neighbor search of q is to retrieve the objects o satisfying the condition: $o \in O$ and $d(q, o) \leqslant r$.

Definition 2.2(k-nearest neighbor search) Given a query object $q \in O$ and an integer $k \geqslant 1$, the k-NN query is to retrieve k objects with the shortest distance from q.

Indexing a metric space aims to provide an efficient support for retrieving objects similar to a reference (query) object (r-neighbor search or k-nearest neighbor search).

3 The M^+-tree

M^+-tree is a dynamical paged and balance tree. It combines binary MVP-tree and M-tree but improves the partition of binary MVP-tree and the node structure of M-tree. In binary MVP-tree, a data space is partitioned into four subspaces with two vantage points while in M^+-tree the partition is done through one vantage point and a key dimension. Because there is no distance computation for partitioning data space by key dimension, M^+-tree has fewer distance computations than MVP-tree. The main idea behind the key dimension is to make the fanout of tree larger by partitioning a subspace further into two subspaces, called twin-nodes. We can double the filtering effectiveness by utilizing the twin-nodes.

3.1 The Key Dimension

3.1.1 Method of Key Dimension Selection

The key dimension is a dimension that affects mostly distance computation. Generally speaking, different data distribution of dimensions has different effect on the distance computation. A key dimension can be used to minimize the overlap, and thus avoid much too unnecessary paths traversal.

In SS-tree and SR-tree, the most optimal partition method is to partition the data space along the axis that has maximal variance, which has been proved to be efficient for their index methods. It keeps the optimization of data space partition and reduces the number of paths traversed. So, in M^+-tree, the dimension having maximal variance is selected to serve as the key one.

3.1.2 The Validity of Key Dimension Filtration

It is a simple process to use the key Dimension to filter. However, some inactive sub-trees may not be filtered. But it always keeps all correct results. The correctness can be deduced from the following formula.

Let $O_i(d_1,d_2,\ldots d_n)$ and $O_j(D_1,D_2,\ldots D_n)$ be two data objects. The distance between the two objects is represented as follows:

$$D(O_i,O_j)=\sqrt{(d_1-D_1)^2+(d_2-D_2)^2+\cdots+(d_n-D_n)^2}$$

Let k be the key dimension number and search radius be r, then $|d_k-D_k| \leqslant D(O_i,O_j)$. If $D(O_i,O_j) \leqslant r$, then $|d_k-D_k| \leqslant r$. So the active data cannot be filtered.

3.2 Partition of Space

Space partition is one of the most important issues in the metric indexes. M-tree partitions object space according to their relative distances. It grows in a bottom-up fashion. By allocating a new node, the overflow of node is managed. At the same level of this node, the entries are partitioned between these two nodes. To reference the two nodes, two reference objects are promoted. In M-tree, partitioning by m-RAD-2 is best among all the partition methods.

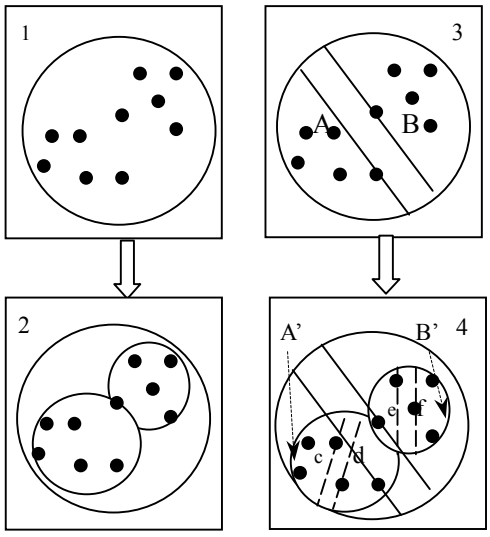

Figure 1: Partition in M-tree v.s. Partition in M^+-tree

Figure 1 (3) and (4) give the process of partitioning space in M^+-tree. A space corresponding to an entry in a tree node consists of two twin spaces, e.g., A and B. The nodes corresponding to twin spaces in M^+-tree are called twin nodes. These two sub-spaces are expressed through two boundary values of a key dimension, i.e. the maximal key dimension value of the left twin space and the

minimal key dimension value of the right twin space. Two boundary values are used to achieve higher filtering ability, because the bigger the gap between the maximal key dimension value of the left twin space and the minimal key dimension of the right twin space, the better the filtering ability.

In M^+-tree, partition is performed in two steps. First, the twin spaces are regarded together as a whole space and it is partitioned with the m-RAD-2 way, as in M-tree. As a result, two new sub-spaces are got, e.g., A' and B' in Figure 1(4). Second, the sub-space A' is further partitioned into two twin sub-spaces c and d according to the selected key dimension. The partitioning process of the sub-space B' is the same as that of the sub-space A'.

3.3 The Structure of M^+-tree Nodes

In this paper, M^+-tree shares the term used in M-tree. So what stored in the internal nodes is termed as routing objects. Therefore, there are two types of node objects, i.e. routing objects and leaf objects. The structure of leaf entries is denoted as the following form:

$L(O_j ,oid(O_j), d(O_j , P(O_j))$

Leaf entry in M^+-tree is quite similar to that of in M-tree. Here, O_j denotes the feature value of a DB object, $oid(O_j)$ an object identifier, and $d(O_j,P(O_j))$ the distance of O_j from its parent.

The structure of routing objects is denoted as the following form:

$R(O_r, r(O_r), d(O_r, P(O_r)), D_{NO,}$ leftTwinPtr($T_{lt}(O_r)$), $M_{lmax,}$ $M_{rmin,}$ rightTwinPtr($T_{rt}(O_r)$))

Where O_r is the feature value of the routing object, $r(O_r)$ the covering radius of O_r, $d(O_r,P(O_r)$ the distance of O_r from its parent, D_{NO} key dimension number, leftTwinPtr($T_{lt}(O_r)$) the pointer to the left twin sub-tree, rightTwinPtr($T_{rt}(O_r)$) the pointer to the right twin sub-tree, M_{lmax} the maximal value of key dimension in the left twin sub-tree and M_{rmin} is the minimal value of key dimension in the right twin sub-tree.

For each routing object, there are two twin pointers to the root of left twin sub-tree and of right twin sub-tree respectively. This data structure increases the fanout of the tree and lowers the height of the tree. Figure 2 indicates an M^+-tree structure.

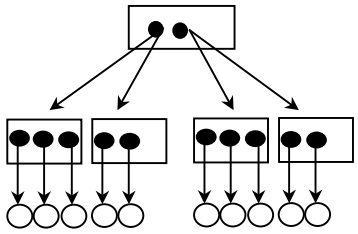

Figure 2:The M^+-tree structure

3.4 Query processing

3.4.1 Range Query

Given a query object q and radius r, the range query starts from the root node and recursively traverses all the paths in which the objects matching condition might exist. Range search algorithm of M^+-tree is described as follows:

```
Algorithm RS (N:node，Q:queryObj，r(Q):queryRad)
1.  begin
2.      Op= ParentNode(N);
3.      if N is Not Leaf
4.          ∀ Or in N, do:
5.          if |d(Op,Q)-d(Or,Op)|<=r(Q)+r(Or)
6.              Compute d(Or,Q);
7.              if d(Or,Q)<=r(Q)+r(Or)
8.                  if keydimVal(Q) -r(Q)≤M1max
9.                      RangeSearch(*leftTwinPtr(Tlt(Or)),
10.                                 Q, R(Q));
11.                 end if
12.                 if keydimVal(Q)+r(Q)≥M2min
13.                     RangeSearch(*rightTwinPtr(Trt(Or)),
14.                                 Q,R(Q));
15.                 end if
16.             end if
17.         end if
18.     else
19.         ∀ Oj in N do:
20.         if |d(Op,Q)-d(Or,Op)| ≤r(Q)
21.             Compute d(Oj,Q);
22.             if d(Oj,Q) ≤r(Q)
23.                 add oid(Oj) to the result;
24.             end if
25.         end if
26.     end if
27. end
```

Range search begins from root firstly. An entry of the node keeps the distance from its parent. Thus, the sub-trees not containing the query results can be filtered using triangular inequality. If the sub-tree is not filtered, the distance between the querying object and the routing object is calculated and further filtering can be done still based on triangular inequality, like in the M-tree. Then, filtering based on key dimension is performed on the remainder twin nodes. The process is done recursively until the leaf. In leaf, the results can be obtained by computation and comparison.

3.4.2 Nearest Neighbour Searching

Given a query object q and the number of objects searching k, k-NN search retrieves the k nearest neighbors of a query object q. Sharing the method proposed in M-tree, M^+-tree uses PR, a *priority queue* that contains pointers to active sub-trees, and NN, an array used to store the final search results.

In the k-NN search algorithm of M^+-tree, the key dimension filtration is added and the priority queues operation is improved. In the PR, let the node be N, if the twin node of N is active, the flag of N is set to *TRUE*. Or this flag is set to *FALSE*. When the twin nodes are all active, only one PR access is needed to do. In this way, many PR accesses are saved. As a result, the cost of query is lowered. The k-NN search algorithm is described as follows.

```
Algorithm k_NN_Search (T: root, Q: queryObj,
                       k: integer)
1.  begin
2.      PR=[T,_];
3.      for i=1 to k do:
4.          NN[i]=[_,∞];
5.      end for
6.      while PR≠Φ  do:
7.          Next_Node=ChooseNode(PR);
8.          k-NN_NodeSearch(Next_Node,Q,k);
9.          if the flag of Next_Node is TRUE
10.             Next_Node =TwinNode(Next_Node );
11.             k-NN_NodeSearch(Next_Node,Q,k);
12.         end if
13.     end while
14. end
```

First, the root is kept into the priority queue PR. The maximal distance is kept in the array NN. Then priority node is chosen from PR and node search are performed. If the flag of this node is *TRUE*, the same search process is needed to do for its twin.

K-NN_NodeSearch function is an important part in the search process. It can be described as follows:

```
Algorithm k-NN_NodeSearch(N:node,Q:queryObj,
                          k:integer)
1.  begin
2.      Op= ParentNode(N);
3.      if N isn't leaf
4.          ∀ Or in N, do:
5.          if |d(Op,Q)-d(Or,Op)|≤dk+r(Or)
6.              Compute d(Or,Q);
7.              if dmin(T(Or))≤dk
8.                  FilterByKeyDim();
9.                  SetFlag( TRUE/FALSE );
10.                 PushPR( Node );
11.             end if
12.             if dmax(T(Or))<dk
13.                 dk=NN_Update([_,dmax(T(Or))]);
14.                 RemoveFromPR ( dmin(T(Or))>dk );
15.             end if
16.         end if
17.     else
18.         ∀ Oj in N, do:
19.         if |d(Op,Q)-D(Oj,Op)|≤dk
20.             Compute d(Oj,Q);
21.             if d(Oj,Q)≤dk
22.                 dk=NN_Update([oid(Oj),d(Oj,Q)]);
23.                 RemoveFromPR(dmin(T(Or))>dk);
24.             end if
25.         end if
26.     end if
27. end
```

Most search operations are implemented within the k-NN_NodeSearch function. First, in an internal node, active sub-nodes are determined. If the twin sub-nodes are all active, the flag of left twin sub-node is set to *TRUE* and left twin sub-node is inserted into the PR. If only one sub-tree is active, the flag of this sub-node is set to *FALSE* and the sub-node is inserted into the PR. Second, if the minimal distance between the covering tree of current node and the query object Q is less than d_k =NN[k-1], the array NN will be updated with an ordered insertion. Third, for the leaf, if the distance between a leaf entry and query object is less than d_k, NN is updated. Repeat the steps above until PR is null. Finally, when the minimal distance between Q and the priority sub-node in PR is greater than d_k, all pointers in PR will be removed and the array NN is returned.

3.5 Construction of M$^+$-tree

A new object is inserted into M$^+$-tree in the following way. First, from the root, the appropriate node is found. If the twin nodes are all full, a split is needed. The split is performed in two steps. First is the splitting based on the distances among objects, and the other is the splitting based on the key dimension. If the node is full but its twin node is not full, then the entries of the twin nodes are reallocated. The process of reallocation is the same as that of the splitting with the key dimension. If the node is not full, the object is inserted directly.

Now, the tree construction and node split algorithms are given in the following. In the insert algorithm, entry(O_n) is an entry to be inserted and N is a node into which the entry is inserted.

```
Algorithm Insert(N：node，entry(On)：M+-tree_entry)
1.  begin
2.      if N is Not Leaf
3.          Node=ChooseSubtree(N);
4.          Insert(Node, entry(On) );
5.      else
6.          if N is Not Full
7.              StoreEntry(N, entry(On) );
8.          else
9.              if TwinNode(N) is Not Full
10.                 Reallocate(N, TwinNode(N),entry(On));
11.             else
12.                 Split(nEntry, entry(On) );
13.             end if
14.         end if
15.     end if
16. end
```

M$^+$-tree performs node split in a bottom-up fashion. It shares the promotion and partition mechanism with M-tree. At the same time, M$^+$-tree performs node split by two steps: splitting with m-RAD-2 like in M-tree and splitting with key dimension. The split algorithm of M$^+$-tree and its input parameters are described as follows.

nEntry: the entry associated to the node splitting

M$^+$-tree entry: object to be inserted;

```
Algorithm Split (N：nEntry；E：M+-tree_entry)
1.  begin
2.      Nt=entries(*N->leftTwinNode)∪
3.      entries(*N->rightTwinNode)∪{E};
4.      if N is not root
5.          let Op let the parent of N, stored in Np node;
6.      end if
7.      Allocates a new node N'
8.      promote(N,Op1,Op2);
9.      partition(N,Op1,Op2,N1,N2);
10.     keydimSplit(N, N1);
11.     keydimSplit(N',N2);
12.     if N is the current root
13.         Allocate a new root node: Np;
```

```
14.          Store entry(O_p1) and entry(O_p2) in N_p;
15.      else
16.          Replace entry(O_p) with entry(O_p1);
17.          if node N_p is full
18.              if the twin node of N_p is full
19.                  Split(NP,entry(O_p2));//NP: entry to N_p
20.              else
21.                  Allocate the entries of parent twins
22.              else
23.                  store entry(O_p2) in N_p;
24.              end if;
25.          end if
26.      end if
27.  end
```

4 Performance Evaluation

Some experimental results of r-N search and k-NN search are given in this section. There are two data sets: a uniform data set and a real data set. The performance of the multidimensional index structures, M-tree and M^+-tree, is evaluated to compare their advantages and disadvantage.

The performance comparison and analysis of M-tree and M^+-tree include the following four aspects: (1) the size of dataset; (2) the dimension of data; (3) uniform data; and (4) real data.

The data sets used in these four tests are shown in Table 1, and the data features are shown in Table 2.

Table 1: Datasets used in the experiments

Test content	Dimension	Dataset size	Description
(1)	10	10,000~ 80,000	Uniform distributed
(2)	5~40	50,000	Uniform distributed
(3)	10	50,000	Uniform distributed
(4)	12	20,000	Real data

Table 2: Features of Datasets used in the experiments

Date set type	Property	Source	Generation method
Uniform data	Uniform	Synthetic	A time function randomly
Real data	Uneven	Real images	MPEG-7 tools

A 10 dimensional uniform data set is used for the performance comparison of k-NN search and range search. The uniform data set is synthetic that consists of the points distributed uniformly in the range [0,1] on each dimension. It is generated randomly through a time function. To evaluate the effect of dimensionality and data set size, the dimension of data space and the size of a data set are varied in 5-40 and 10,000-80,000, respectively.

For real data set, data is 12 dimensional points. They are generated using the MPEG-7 feature extraction tools. It consists of the real color layout feature vectors of images. Through standardizing the vectors, the data is transformed to Euclidean space and each dimension of the feature ranges between [0,1].

The testing environment is a Pentinum III 933MHz PC with 128MB memory and 30G hard disk. The data are stored in an object database system Fish (Yu G, Kaneko K, Bai G, Makinouchi A. 1996).

4.1 Data Set Size vs. Performance

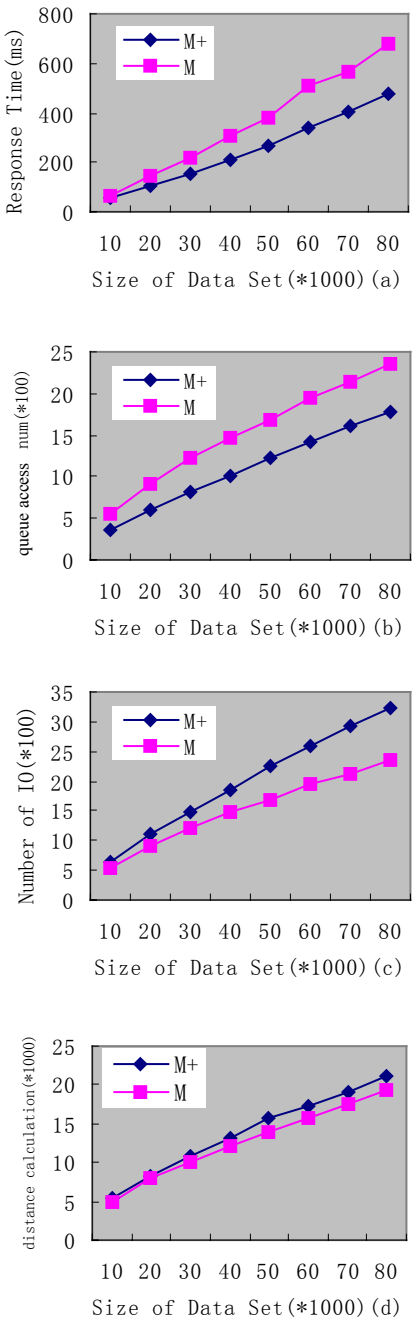

Figure 3: Comparison of k-NN searches by dataset size varying

We investigated how the performance of M-tree and M^+-tree varies with the data sets size. The performance metrics include *total response time*, *distance calculation, IO operations, and PR operations*. The data sets size is varied from 10,000 to 80,000. The number of retrieved object is fixed to 10. Figure 3 shows the performance comparison of M-tree and M^+-tree for k-NN search.

From the figure, we can see that M+-tree has more IO operations and distance calculations than M-tree. This is mainly because the k-NN search algorithm adopts the heuristic criterion and sets the search radius from maximum, while the key dimension has a poor filtering

ability when the search radius is longer. At the same time, query radius converges very slowly because the uniform data is sparse. However, the heuristic criterion is implemented with operations to the priority queue, in which many ordering operations are needed, this is much time-consuming, so the number of queue access is an important factor affecting the performance of the index. Due to the introduction of the twin nodes in M$^+$-tree, the number of queue access is reduced greatly, which can be seen from Figure 3(b). This saves much more query time and the response time is reduced remarkably. With the increasing of data sets size, M$^+$-tree has more advantage over M-tree on search performance. The performance advantage of M$^+$-tree will be more noticeable for real data.

4.2 Dimensionality of Data vs. Performance

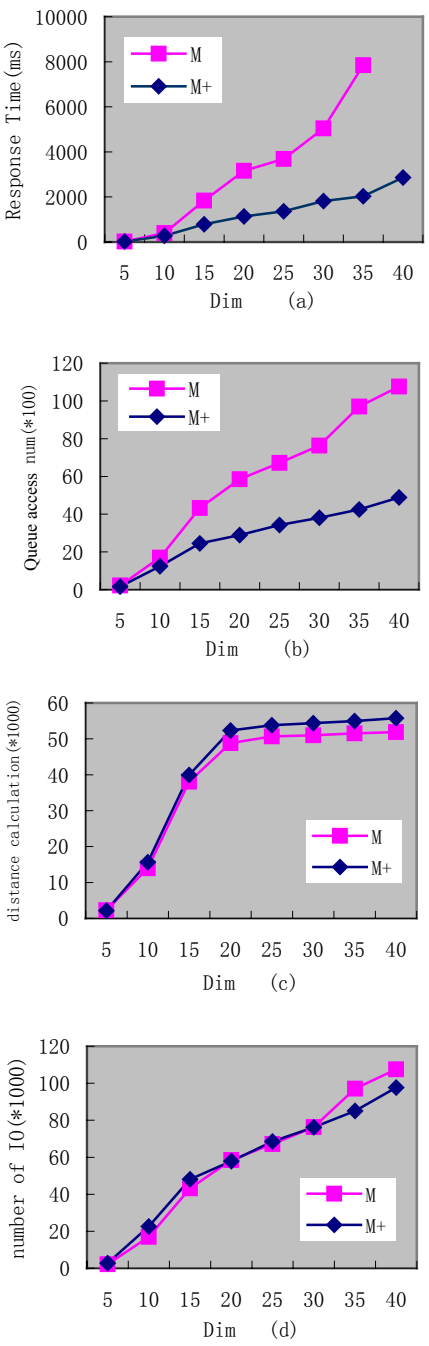

Figure 4: comparison of k-NN searches by dimension varying

To compare the different characteristics of M$^+$-tree and M-tree, we measured the performance of them with varying the dimensionality. In this experiment, Data set is uniform and the size is fixed to 50,000. Data space dimensionality varies from 5 to 40.

From Figure 4, we can see that the number of distance calculation and IO access is very close for the two indexes with the dimension varying. Moreover, when the dimension increases to 25, they both have to traverse nearly the whole tree because the subspaces are all overlapped. The search performance is mainly determined by queue access. While the queue access number of M$^+$-tree is well under that of M-tree when the dimension becomes larger, M$^+$-tree has a much better performance than M-tree.

4.3 Performance on Uniform Data

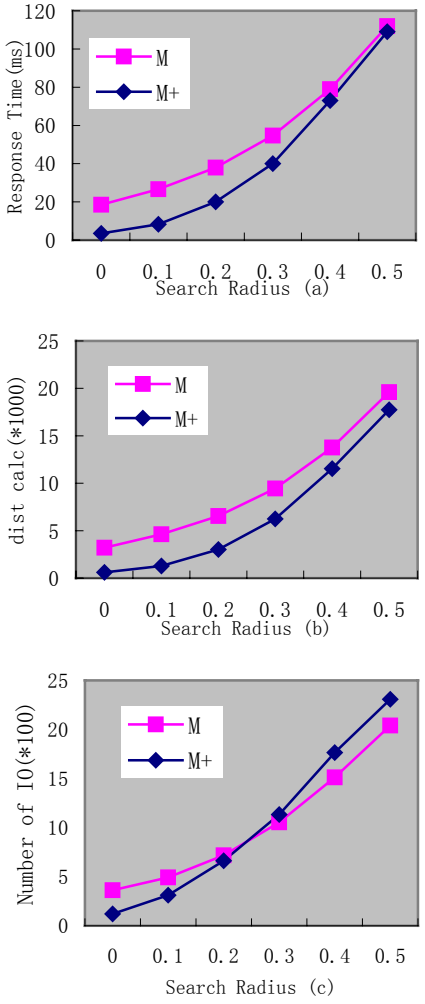

Figure 5: Comparison of r-N search for Uniform Data

In this section, we compare the range search and k-NN search of M$^+$-tree with M-tree' when the data is uniform. We compare their performance for range search in terms of response time, the number of distance calculation and the number of IO. Figure 5 gives the experimental result.

The performance of M$^+$-tree are superiority over the M-tree' in the three factor when the search radius is less, and the less search radius, the better search performance. When the search radius closes to zero, about half data can be filtered in the twin nodes. M$^+$-tree is faster twice than

M-tree because of the higher filtering ability. As the search radius increases, the filtering ability of M^+-tree becomes weak and the two indexes' performance contrast is unapparent gradually.

M^+-tree needs less distance calculation, and the distance calculation number is closer with the increasing of search radius. At the same time, because the filtering ability of key dimension reduced with the increasing of search radius, M^+-tree's IO increases more quickly.

Because of the data's multidimensionality, distance calculation needs more time. At the same time, the distance calculation contrasts more bitterly. Therefore M^+-tree improves the query performance despite that it needs more IO access sometimes.

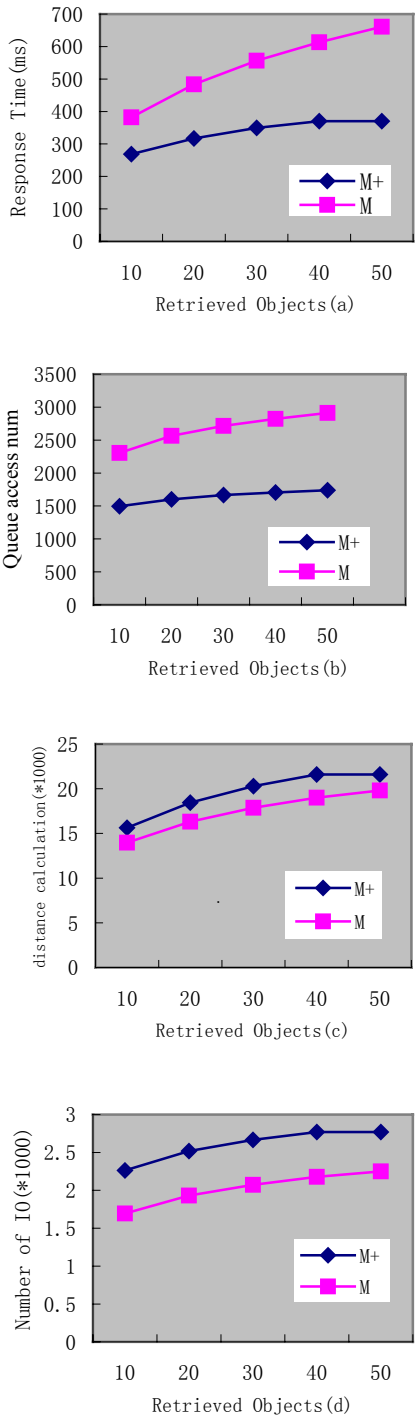

Figure 6: Comparison of k-NN search for Uniform data

Figure 6 shows the comparison result of k-NN search for uniformly distributed data. It compares the two indexes from four performance aspects: (1) response time, (2) the number of priority queue access, (3) distance calculation, (4) Number of IO. M^+-tree needs more distance calculations and IO access due to the uniformity of sparse data. But because of the using of twin nodes, M^+-tree needs less priority queue access operation, as a result, M^+-tree responds more quickly than M-tree. In brief, M^+-tree outperforms M-tree on query performance because of the introduction of key dimension and twin node.

4.4 Performance on Real Data

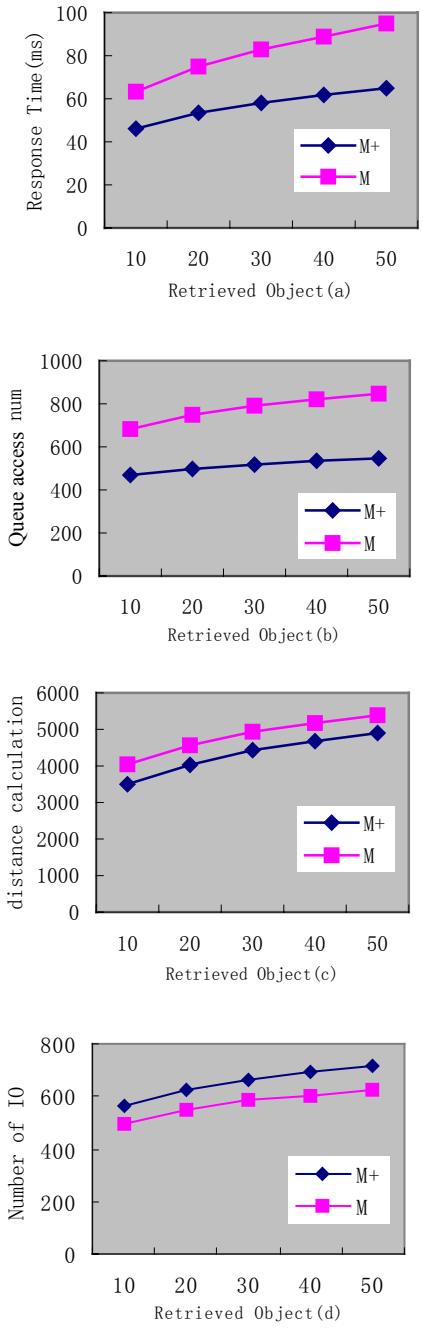

Figure 7: Comparison of k-NN search for real data

In our experiment, we also test the performance of k-NN search and range search using a set of real data. As Figure

7 and Figure 8 show, because the real data is distributed densely, and clusters more easily, the superiority of M^+-tree is more obvious. In k-NN search, as regards response time, priority queue access number and distance calculations, M^+-tree superiors to the M-tree. Only M^+-tree has more IO operations slightly. As a result, because of fewer queue access operations and distance calculation, the query using M^+-tree is more rapidly.

In range search, the number of IO is very close in the two index structures. But M^+-tree needs fewer distance calculations, which saves much more time. Therefore, despite the performance curves of IO access have a cross, the response time is shorter in M^+-tree than in M-tree.

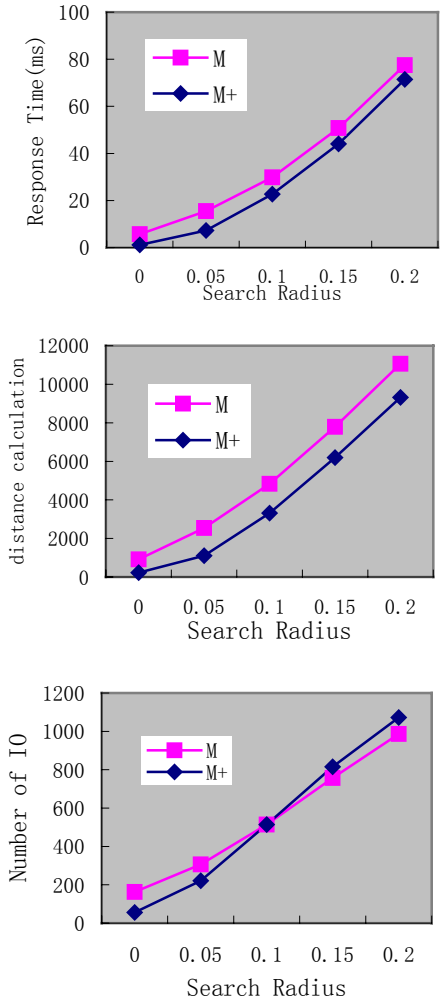

Figure 8:Comparison of r-NN search for real data

5 Conclusion

In this paper, we proposed M^+-tree that is a dynamically updateable page-based metric index. This approach has the following features.

(1) Using a key dimension to split a subspace into two twin subspaces, which are not overlapped.

(2) In k-NN search, when the twin nodes need to enter a queue, this operation is performed only once, by using a flag in each node to mark whether its twin node is contained in the queue.

(3) In range search, the key is used to perform effective filter while it is not needed to compute distance between objects.

(4) The M^+-tree adopts reallocation process between twin nodes to save storage space, and prevents the underflow of nodes.

Experimental results show that the M^+-tree has a higher query performance compared with M-tree.

6 Acknowledgement

This research was supported by the National Natural Science Foundation of China (No.60173051 and No.60273079), the Foundation for University Key Teacher and the Teaching and Research Award Program for Outstanding Young Teachers in High Education Institution of the Ministry of Education, China.

7 References

N.Berkmann, H.-P. Krigel. R. Schneider, and B.Seeger. (1990) "The R*-tree: an efficient and robust access method for points and rectangles." ACM SIGMOD, pp.322-331, Atlantic City, NJ.

N.Katayama and S.Satoh. (1997) The SR-tree: an index structure for high-dimensional nearest neighbor queries. Proc. ACM SIGMOD Intl. Conf. On Management of Data, pp369-380,.

D.A.White and R.Jain.(1996) "Similarity Indexing with the SS-tree," Proc.of the 12th Int.conf. on Data Engineering, New Orleans, USA, pp.516-523.

K.-I. Lin,H. V. Jagadish, and C. Faloutsos. (1994) "The TV-tree: An Index Structure for High-Dimensional Data," VLDB Journal, Vol. 3, No. 4, pp.517-542.

J. K. Uhlmann.(1991) " Satisfying General Proximity/ Similarity Queries with Metric Trees", Information Processing Letters, vol 40,pages 175-179.

P.Zezula, P.Ciaccia, and F.Rabitti.(1996) "M-tree: A dynamic index for similarity queries in multimedia databases". TR 7, HERMES ESPRIT LTR Project.

T.Bozkaya,M. Ozsoyoglu.(1997) "Distance-based indexing for high-dimensional metric spaces." Proc. the ACM SIGMOD International Conference on Management of Data,Tucson, Arizona, page 357-368.

P.Ciaccia, M.Patella, P.Zezula.(1997) "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces" Proc the 23rd VLDB Conference Athens, Greece.

M. Ishikawa, H. chen, K, Furuse, J.Xu Yu, N.Ohbo(2000) "MB+tree: a Dynamically Updatable Metric Index for Similarity Search" WAIM, PP356-3

Yu G, Kaneko K, Bai G, Makinouchi A. (1996) Transaction management for a distributed object storage system WAKSHI – design, implementation and performance. Proc. Of the 12th Int. Conf. On Data Engineering, USA.