A Study of Indexing Strategies for Hybrid Data Spaces

Changqing Chen[1], Sakti Pramanik[1], Qiang Zhu[2], and Gang Qian[3]

[1] Department of Computer Science and Engineering
Michigan State University, East Lansing, MI 48824, USA
chencha3@cse.msu.edu, pramanik@cse.msu.edu
[2] Department of Computer and Information Science
The University of Michigan - Dearborn, Dearborn, MI 48128, USA
qzhu@umich.edu
[3] Department of Computer Science
University of Central Oklahoma, Edmond, OK 73034, USA
gqian@uco.edu

Abstract. Different indexing techniques have been proposed to index either the continuous data space (CDS) or the non-ordered discrete data space (NDDS). However, modern database applications sometimes require indexing the hybrid data space (HDS), which involves both continuous and non-ordered discrete subspaces. In this paper, the structure and heuristics of the ND-tree, which is a recently-proposed indexing technique for NDDSs, are first extended to the HDS. A novel power value adjustment strategy is then used to make the continuous and discrete dimensions comparable and controllable in the HDS. An estimation model is developed to predict the box query performance of the hybrid indexing. Our experimental results show that the original ND-tree's heuristics are effective in supporting efficient box queries in the hybrid data space, and could be further improved with our proposed strategies to address the unique characteristics of the HDS.

Keywords: Hybrid data space, Database, Access method, Multidimensional indexing, Box query.

1 Introduction

In many contemporary database applications, indexing of hybrid data which contains both continuous and discrete dimensions is required. For example, when indexing weather data of different locations, the daily temperature, precipitation, humidity should be treated as continuous information while other information such as the type of precipitation is typically regarded as discrete. Different indexing techniques have been proposed for either the CDS or the NDDS. Examples for CDSs indexing methods are the R-tree [6], R*-tree [1], K-D-B-tree [12] and LSDh-tree [7]. NDDSs indexing techniques include the ND-tree [9,10] and the NSP-tree [11]. Not surprisingly, all these indexing methods could not be applied to the HDS directly because they rely on domain-specific characteristics (e.g., the order of data in the CDS) of their own data spaces.

One way of applying the CDS/NDDS indexing techniques to the HDS is to transform data from one space to the other. For example, discretization methods [2,5,8] could be

J. Filipe and J. Cordeiro (Eds.): ICEIS 2009, LNBIP 24, pp. 149–159, 2009.

utilized to convert data from continuous space to discrete space. If we discretize the weather data mentioned before, daily temperature could be converted to three discrete values: cold, warm and hot. However, this approach clearly changes the semantics of the original data.

The C-ND tree [4] is recently proposed to create indexes for the hybrid data space, and is optimized to support range queries in the HDS. In this paper, we evaluate the effectiveness of the extended ND-tree structure and heuristics for box queries in the HDS. The ND-tree structure and building algorithms/heuristics are extended to handle both continuous and discrete information of the HDS. A novel strategy using power adjustment values to balance the preference for continuous and discrete dimensions is presented to handle the unique characteristics of the HDS. And an effective cost model to predict the performance of HDSs indexing is introduced. Our experimental results show that, the extended heuristics are effective in supporting box queries in the HDS, and the cost estimates from the presented performance model are quite accurate.

The rest of the paper is organized as follows. In Section 2 the ND-tree data structures and heuristics are extended to the HDS, and an approach of using different power (exponent) values to handle the unique characteristics of the HDS is presented. Section 3 reports our experimental results, which demonstrate that hybrid space indexing is quite promising in supporting efficient box queries in HDSs. Section 4 outlines a model to predict the performance of hybrid space indexing. Section 5 describes the conclusions and future work.

2 The Extended Hybrid Indexing

2.1 Hybrid Geometric Concepts and Normalization

To efficiently build indexes for the HDS, some geometric concepts need to be extended from the NDDS to the HDS. The hybrid geometric concepts used in this paper are the *hybrid (hyper-)rectangle* in the HDS, the *edge length* of a hybrid rectangle on a given dimension, the *area* of a hybrid rectangle, the *overlap* between two hybrid rectangles and the *hybrid minimum bounding rectangle (HMBR)* of a set of hybrid rectangles. Detailed definition of these concepts could be found in [4] and is omitted here due to page limit.

One challenge in applying hybrid geometric concepts is how to make the measures for the discrete and continuous dimensions comparable. For example, how to compare the size of 2 for a discrete component set (i.e., 2 letters/elements) of an HMBR with the length of 500 for a continuous component interval in the same HMBR? To solve the problem, we adopt normalized measures for hybrid geometric concepts introduced in [4]. In the rest of this paper, we always use normalized hybrid geometric measures unless stated otherwise.

2.2 Extending the ND-Tree to the HDS

Each non-leaf node entry in the hybrid indexing tree keeps an HMBR of a child node and a pointer to that child node. Information for discrete dimensions in the HMBR

is stored using a bitmap representation and information for continuous dimensions is stored by recording the corresponding lower and upper bounds of each dimension. Each leaf node entry stores the discrete and continuous component of every dimension as well as a pointer pointing to the actual data associated with the key in the database.

Two critical tasks, namely choosing a leaf node to insert a new vector and overflow treatment are extended to the HDS, by using the corresponding geometric concepts defined in Section 2.1. When splitting an overflowing node, sorted entry lists are generated for a continuous dimension C by sorting all entries' lower bound values and then upper bound values on C. Detailed discussion of these two tasks is omitted in this paper and could be found in [9,10].

Given the extended insert operation in the HDS, the delete operation is implemented as follows. If no underflow occurs after an entry is removed from a leaf node, only the ancestor nodes' HMBRs are adjusted. In case of an underflow, the whole node is removed and all the remaining entries in that node are reinserted.

A query box in the HDS is a hybrid rectangle containing a query range/set on each dimension. For a continuous dimension, a query range is specified by its upper and lower bounds. For a discrete dimension, a query set is specified by a subset of letters/elements from its domain. A traditional depth-first search algorithm is implemented for the hybrid indexing tree and the details are omitted in this paper.

2.3 Enhanced Strategy for Prioritizing Discrete/Continuous Dimensions

As mentioned in Section 2.1, to make a fare comparison between discrete and continuous dimensions, we have employed normalized edge lengths when calculating geometric measures for an HDS. This strategy allows each dimension to make suitable contributions relative to its domain size in the HDS.

We also notice that the (non-ordered) discrete and continuous dimensions usually have different impacts on the performance of hybrid indexing due to their different domain properties. For example, a discrete dimension is more flexible when splitting its corresponding component set of an HMBR due to the non-ordering property, resulting in a higher chance to obtain a better (smaller overlap) partition of the relevant overflow node. Assume $S_1 = \{a, g, t, c\}$ is the component set of an HMBR on a discrete dimension, the letters in S_1 can be combined into two groups arbitrarily (subject to the minimum space utilization constraint of the node) to form a split of the set, e.g., $\{g\}/\{a, t, c\}$, $\{a, t\}/\{g, c\}$, etc. On the other hand, for the component set on a continuous dimension, say $S_2 = [0.2, 0.7]$, the way to distribute a value in S_2 totally depends on the splitting point. If the splitting point is 0.5 (i.e., having a split $[0.2, 0.5]/(0.5, 0.7]$), the values less than or equal to 0.5 have to be in the first group, and the others belong to the second group, because of the ordering property of a continuous dimension. The challenge is how to make use of this observation to balance the preference for discrete and continuous dimensions in the HDS.

One might suggest adopting different weights for the (normalized) edge lengths of an HMBR on the discrete and continuous dimensions, respectively. Unfortunately, this approach can not work. Two important measures used in the tree construction algorithms are the area of an HMBR (or overlap between HMBRs) and the span of an HMBR on a particular dimension (i.e., the edge length of the HMBR on the given dimension).

Suppose we have HMBRs (or overlaps) R_1 and R_2 in a two-dimensional HDS with the following relationship: $Area(R_1) = L_{11} \times L_{12} < Area(R_2) = L_{21} \times L_{22}$, where L_{ij} is the (normalized) edge length of R_i on the *j-th* dimension ($j = 1, 2$). Assume that the first dimension is discrete and the second one is continuous. If we assign a weight w_d to the discrete edge length and another weight w_c to the continuous edge length when calculating the area, we will still have $Area(R1) = (w_d \times L_{11}) \times (w_c \times L_{12}) < Area(R2) = (w_d \times L_{21}) \times (w_c \times L_{22})$ because the weight factors on both sides of the inequality cancel each other. The same observation can be obtained for spans.

To overcome the above problem, we adopt another approach by assigning different power (exponent) values p_d and p_c to the discrete and continuous edge lengths, respectively, when calculating area values. With a normalization, we can assume $p_c = (1 - p_d)$. For R_1 and R_2 in the above example, if$L_{11} = 0.1, L_{12} = 0.3, L_{21} = 0.2, L_{22} = 0.2, p_c = 0.1, p_d = 0.9$, we have $Area(R_1) = L_{11}^{p_d} \times L_{12}^{p_c} = 0.1^{0.1} \times 0.3^{0.9} \approx 0.27 > Area(R_2) = L_{21}^{p_d} \times L_{22}^{p_c} = 0.2^{0.1} \times 0.2^{0.9} \approx 0.20$, while the original area values have the relationship $Area(R_1) = 0.1 \times 0.3 = 0.3 < Area(R_2) = 0.2 \times 0.2 = 0.4$. Hence, we can change the area comparison result by using different power adjustment values. Since the edge length is normalized to be between 0 and 1, the larger the power value is, the smaller the adjusted length would be (unless the edge length is 1 or 0). For the heuristics involving areas comparison during tree construction, we always prefer a smaller area. Therefore, if we increase power p_d (i.e., reduce p_c) for discrete dimensions, we make the discrete edge lengths contribute less to the area calculation while making the continuous edge lengths contribute more to the area calculation. In this sense, we make the discrete dimensions more preferred. The way to make the continuous dimensions more preferred is similar.

We can also assign power adjustment values q_d and $q_c = (1 - q_d)$ to the discrete and continuous edge lengths, respectively, when calculating the span value for every dimension. However, during tree construction time a dimension with a larger span is more preferred. Therefore, if we want to make a discrete dimension more preferable, we need to decrease power value q_d for that dimension, which is different from the situation of calculating areas values. The discussion for the span value on continuous dimensions is similar. If we want to make discrete dimensions consistently more preferred during the tree construction, we need to increase p_d and, in the meantime, decrease q_d.

To reduce the number of parameters for the algorithm, we simply let $q_d = (1 - p_d)$. Hence, we only need to set a value for one parameter p_d , other parameters (i.e., $p_c,\ q_d,\ q_c$) are generated according to p_d. The experimental results in Section 3 show that this power adjustment strategy further improves the performance of the hybrid indexing.

3 Experimental Results

3.1 Experimental Setup

Data sets used for our experiments were randomly generated which consist both continuous and discrete dimensions. For a discrete dimension if the alphabet size is A, a discrete value was created by generating a random integer between 0 and $A - 1$. For a

continuous dimension if the range is A, the possible values are decimal numbers ranging between 0 and A. For a test query, a box size X is used to define the volume of its query box. Given a query box with box size X, each discrete dimension has X letters and each continuous dimension has length X. The query performance is measured by the number of I/Os (i.e., the number of index tree nodes accessed, assuming each node occupies one disk block) and is computed by averaging the I/Os over 200 queries.

In the following subsections we compare performances of the hybrid indexing tree, the ND-tree, the R*-tree and the 10% liner scan[3]. For the same reason discussed in [4], we keep both continuous and discrete data in the leaf nodes of the ND-tree and the R* tree. Various parameters such as database sizes and alphabet sizes are considered in our experiments. A symbol δ is used to represent the additional dimensions utilized by the hybrid indexing approach. For example, given a HDS with i continuous dimensions and j discrete dimensions, by indexing the whole HDS we have $\delta(\delta = j)$ extra dimensions to use when compared to the R*-tree approach, and $\delta(\delta = i)$ extra dimensions to use when compared to the ND-tree approach.

In our experiments we create the R*-tree for a 4-dimensional continuous subspace, which is a typical dimension number for the R*-tree to avoid the dimensionality curse problem. For the discrete subspace we use 8 dimensions because an effective ND-tree could not be built if the number of dimensions is too low (there are too many duplicate vectors in the subspace).

From the experiment results we see that the hybrid indexing outperforms the other three approaches. In some of the cases, the performance gain is quite significant.

3.2 Performance Gain with Increasing Database Sizes

In this group of tests the performance of hybrid indexing is compared with that of the ND-tree, the R*-tree and the 10% linear scan for various database sizes (i.e., the number of vectors indexed). The number of additional dimensions δ is set to 2. That is, we use the hybrid indexing approach to index the 4 continuous dimensions used by the R*-tree plus 2 additional discrete dimensions, and compare the query I/O with that of the R*-tree. Similarly, we compare the performance of indexing 8 discrete dimensions and 2 continuous dimensions against the ND-tree approach which indexes only the 8 discrete dimensions. The query I/O of hybrid indexing is also compared with that of the 10% linear scan approach, which utilizes all the dimensions as the hybrid indexing does. The alphabet size for each of the discrete dimensions is set to 10. Figure 1 shows that the hybrid indexing approach reduces box query I/Os and the performance gain generally increases with growing database sizes.

3.3 Performance for Various Additional Dimensions

In the following experiments, we varies the number of additional dimensions (i.e., the δ value) used by the hybrid indexing. The alphabet size and database size in these experiments are set to 10 and 10 million respectively. Our experimental results are reported in Figure 2. Again from these results we see that the hybrid indexing outperforms all the other approaches. In some cases (e.g., compared with the R*-tree) the performance improvement is significant.

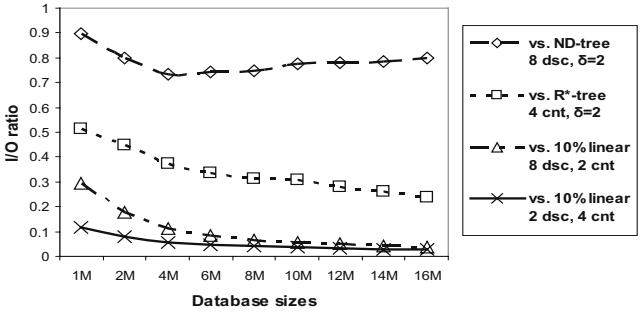

Fig. 1. Effect of various database sizes

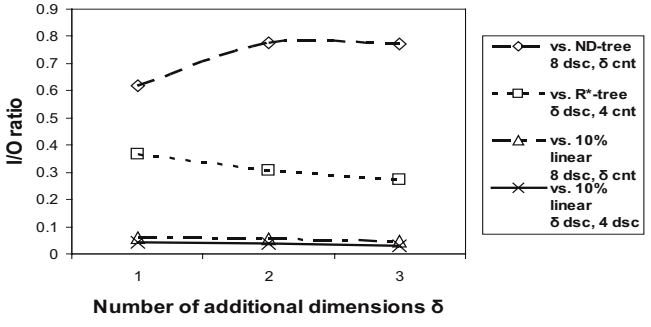

Fig. 2. Effect of additional dimensions

3.4 Performance for Different Alphabet Sizes

As we see from Figure 3, the hybrid indexing is much more efficient when compared to the R*-tree and the 10% linear scan. It also does better than the ND-tree approach. With increasing alphabet sizes the ND-tree performance gets closer to the hybrid indexing. However, in real world applications most NDDS domains are small. For example, genome data has a domain size of 4 (i.e., $\{a,\ g,\ t,\ c\}$). The database size used for this group of tests is 10 million.

3.5 Performance for Different Query Box Sizes

All of the above experiments show the performance comparisons for query box size 2. This group of tests evaluates the effect of different query box sizes. Query results for box size $1 \sim 3$ are reported here because as box sizes become larger, the 10% linear scan approach is more preferable. Not surprisingly, all indexing trees will eventually lose to linear scan when the query selectivity is high. The results in Figure 4 show that indexing the hybrid data space increases box query performance for all the box sizes given. Database size 10 million and alphabet size 10 are used for this group of tests.

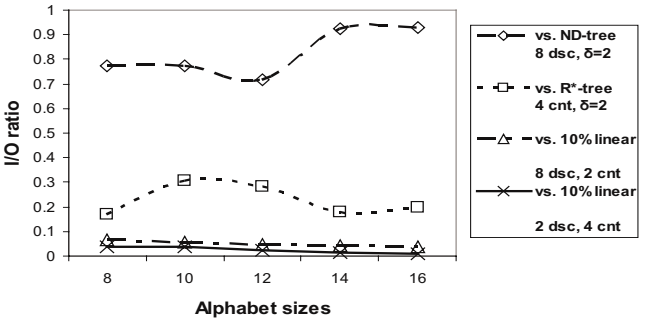

Fig. 3. Effect of different alphabet sizes

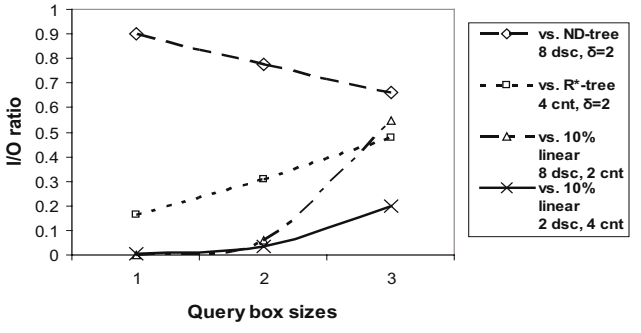

Fig. 4. Effect of different query box sizes

3.6 Effect of Enhanced Strategy with Power Value Adjustment

To examine the effectiveness of using exponent (power) values to adjust the edge lengths of an HMBR, as discussed in Section 2.3, we conducted relevant experiments. The query I/Os of using this enhanced strategy are shown in Figure 5. The x-axis indicates different power values (p_d) used for discrete dimensions. To eliminate the possible effect that different number of discrete and continuous dimensions might have on the enhanced strategy, we use an HDS with 4 discrete and 4 continuous dimensions. The alphabet size is set to 10 and number of vectors indexed is 10 million. The results in Figure 5 show that the new enhanced strategy could further improve the performance of the hybrid indexing using extended ND-tree heuristics. The exponent value of 0.5 corresponds to the situation of not applying any power value adjustment because both discrete and continuous dimensions have the same exponent value (0.5).

4 Performance Estimation Model

To predict the performance behavior of the hybrid indexing tree, we have developed a performance estimation model. Our model is divided into two parts: the first part for estimating the key parameters of the tree (e.g., the characteristics of HMBRs of tree

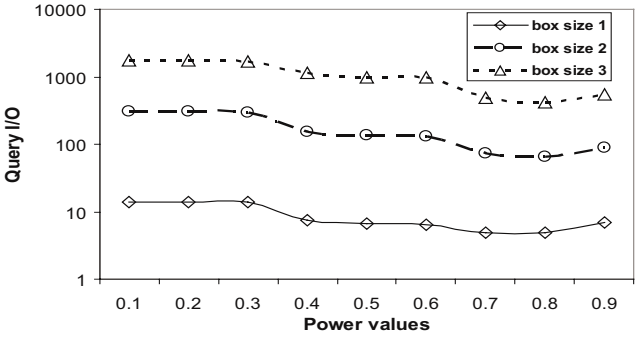

Fig. 5. Performance for enhanced strategy with power value adjustment

nodes) for a given HDS; and the second part for estimating the number of I/Os for arbitrary query box sizes based on the key parameters. If the tree is given and we want to predict the performance behavior of the tree, only the second part is needed. The two parts of our model are sketched as follows.

Part I: Estimating key parameters of a tree

In this part of the performance model, we are given an HDS and the number of vectors (to be indexed) in the HDS. Hence we have the following input parameters: the system disk block size, the number of continuous and discrete dimensions, the domain/alphabet size of the discrete domains, and the number of vectors to be indexed. We assume that the components of the vectors are uniformly distributed in their respective dimensions.

Using the above information, we can first determine the maximum number of entries M_n in a non-leaf node, and the maximum number of entries M_l in a leaf node. From the heuristics extended to the HDS, we notice that the numbers of splits different nodes (at the same level of the tree) went through usually either equal to each other or differ by 1. The tree growing process involves two time windows: during any time of the tree growth, when a node at certain level splits, all the other nodes at the same level split around the same time. We call this time period *the splitting (time) window*. After a node is split, the new nodes start to accumulate incoming data for some time. We call this period *the accumulating (time) window*. Since a new node created from a split is about half-full and takes quite some time before it becomes full again, the accumulating window is typically much larger than the splitting window. Thus we focus on capturing the performance behavior for the accumulating window in our performance model.

At the beginning of an accumulating window the node space utilization is about 50% because each overflow node has split into two half-full nodes. When the accumulating window ends, the node space utilization is close to 100%. On average the utilization would be 75%. However, we notice that, in real world situations, although most splits occur in the splitting window, there are some splits happening during the accumulating window. As a result, the number of nodes during the accumulating window is slightly more than what is estimated under the assumption that all splits happen in the splitting window. Hence the actual average node space utilization (around 70% from our experiments) is also lower than expected. Therefore, our estimates for the average numbers

of entries in a non-leaf node and a leaf node during the accumulating window are: E_n = 0.7 * M_n and E_l = 0.7 * M_l, respectively.

The height h of the tree to index V number of vectors is estimated as: $h = \lceil log(\lceil V/E_l \rceil)/\lfloor logE_n \rfloor \rceil + 1$. The estimated number of nodes at the leaf level is $n_0 = \lceil V/E_l \rceil$. The estimated number of nodes at level i is: $n_i = \lceil n_{i-1}/E_n \rceil$ $(1 \leq i \leq h)$. The number of rounds of splits that every node at level i have gone through can be estimated as: $w_i = \lfloor log_2(n_i) \rfloor$. The number of nodes at level i which have gone through one more round of split is estimated as: $v_i = (n_i - 2^{w_i}) \times 2$.

After we know the height of a tree, the number of nodes at each level and the number of splits each node has gone through, we can estimate the parameters (e.g., edge length) for the HMBRs of nodes at each level. Details of the lengthy derivation are omitted here due to the space limitation.

Part II: Estimating query performance based on key parameters of a tree

The second part of our model estimates the number of I/Os needed for a hybrid indexing tree (defined by the key parameters discussed in *Part I*) given arbitrary query box sizes. The estimation has three steps:

(ES-1) Estimating the overlapping probability for one discrete/continuous dimension

For a discrete dimension d, assume that the domain size of the dimension is D_d, the set size of a node $N's$ HMBR on dimension d is L_d, and the set size of a query box on this dimension is T_d. Clearly, $L_d \leq D_d$ and $T_d \leq D_d$. The probability of the query box overlapping with $N's$ HMBR on dimension d is:

$$1 - C_{D_d - L_d}^{T_d} / C_{D_d}^{T_d}$$

For a continuous dimension c, without loss of generality, assume that the domain range/interval is $[0, C_c]$, and the lower and upper bounds of a node $N's$ HMBR on dimension c are L_c and U_c, respectively. Further suppose the edge length of a query box on this dimension is T_c. We have $0 \leq L_c, U_c \leq C_c$ and $T_c \leq C_c$. The probability for the query box to overlap with $N's$ HMBR on dimension c is:

$$(b - a)/(C_c - T_c)$$

where $a = max\{L_c - T_c,\ 0\}$ and $b = min\{U_c,\ C_c - T_c\}$. This probability calculation is based on the lower bound value p of the query box on dimension c. Clearly, p is within range $[0,\ C_c - T_c]$. If the query box has an overlap with interval $[L_c, U_c]$ on dimension c, p must be within $[L_c - T_c,\ U_c]$. a and b are used to handle boundary conditions when $(L_c - T_c) < 0$ and $(U_c + T_c) > C_c$.

(ES-2) Estimating the overlapping probability for one tree node

The probability of a tree node N overlapping with an arbitrary query box Q, is the product of the overlapping probabilities of N and Q on all dimensions, which are calculated by *ES-1*.

(ES-3)Estimating the I/O number for the tree

The number of I/Os for the tree to process a box query is estimated as the summation of the overlapping probabilities between the query box and every tree node, which could be calculated by *ES-2*.

We conducted experiments to verify the above performance model. Two sets of typical experimental results are shown in Figures 6 and 7. Each observed performance data

was measured using the average number of I/Os for 200 random queries. The HDS used in the experiments has 4 continuous dimensions and 4 discrete dimensions with an alphabet size of 10.

Figure 6 shows the comparison between our estimated and observed I/O numbers for queries with box sizes $1 \sim 3$. The number of indexed vectors ranges from 1 million to 10 million. Since the trees are given, the experimental results actually demonstrate the accuracy of the second part of our performance model. The experimental results show an average relative error of only 2.45% in such a case.

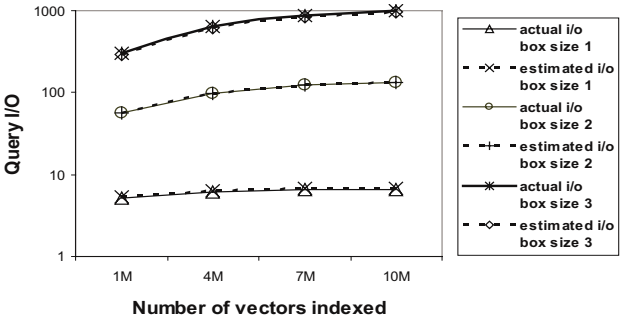

Fig. 6. Verification of performance model for given trees and given HDSs

Figure 7 shows the comparison between our observed I/O numbers (from queries on actual trees) and estimated I/O numbers (for the given HDS without building any tree). In this case, the tree parameters for the given HDS also need to be estimated using the first part of our model. From the figure, we can see that our performance model still give quite good estimates although the accuracy degrades a little bit due to the fact that more parameters need to be estimated. The average relative error is 5.76%.

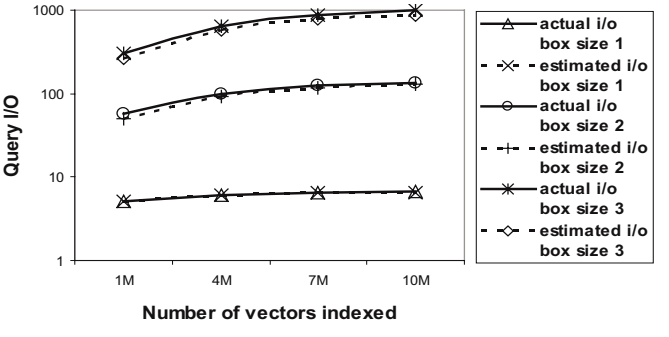

Fig. 7. Verification of performance model for given HDSs

5 Conclusions

In this paper, the original ND-tree structure and its building heuristics are extended to the HDS. A power value adjustment strategy is employed to make the measures on

continuous and discrete dimensions comparable and controllable. A theoretical model is also developed to predict the performance of the hybrid indexing in HDSs.

Our experimental results demonstrate that the extended ND-tree's heuristics are still effective in supporting box queries in the HDS. Using these heuristics to index the HDS is more efficient than the traditional linear scan, the method to index the continuous subspace of the underlying HDS using the R*-tree and the method to index the discrete subspace using the ND-tree. The reason is that during the query time the hybrid indexing approach could prune nodes based on information from additional dimensions which the R*-tree and ND-tree do not have.

Our future work includes developing more effective heuristics for the HDS indexing.

Acknowledgements. This research was supported by the US National Science Foundation (under grants # IIS-0414576 and # IIS-0414594), Michigan State University and the University of Michigan.

References

1. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of ACM SIGMOD, pp. 322–331 (1990)
2. Catlett, J.: On changing continuous attributes into ordered discrete attributes. In: Proceedings of the European Working Session on Machine Learning, pp. 164–178 (1991)
3. Chakrabarti, K., Mehrotra, S.: The hybrid tree: an index structure for high dimensional feature spaces. In: Proceedings of the 15th International Conference on Data Engineering, pp. 440–447 (1999)
4. Chen, C., Pramanik, S., Watve, A., Zhu, Q., Qiang, G.: The C-ND Tree: A Multidimensional Index for Hybrid Continuous and Non-ordered Discrete Data Spaces. In: Proceedings of the 12th International Conference on Extending Database Technology (2009)
5. Freitas, A.A.: A survey of evolutionary algorithms for data mining and knowledge discovery. In: Advances in Evolutionary Computing: Theory and Applications, pp. 819–845 (2003)
6. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of ACM SIGMOD, pp. 47–57 (1984)
7. Henrich, A.: The LSDh-tree: an access structure for feature vectors. In: Proceedings of the 14th International Conference on Data Engineering, pp. 362–369 (1998)
8. Macskassy, S.A., Hirsh, H., Banerjee, A., Dayanik, A.A.: Converting numerical classification into text classification. Artificial Intelligence 143(1), 51–77 (2003)
9. Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: The ND-tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. In: Proceedings of the 29th International Conference on VLDB, pp. 620–631 (2003)
10. Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: Dynamic indexing for multidimensional non-ordered discrete data spaces using a data-partitioning approach. Proceedings of ACM Transactions on Database Systems 31(2), 439–484 (2006)
11. Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces. ACM Trans. on Information Syst. 23(1), 79–110 (2006)
12. Robinson, J.T.: The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In: Proceedings of ACM SIGMOD, pp. 10–18 (1981)