DIVA: INDEXING HIGH-DIMENSIONAL DATA BY "DIVING" INTO VECTOR APPROXIMATIONS

Konstantinos Tsakalozos[1]*, Spiros Evangelatos*[2] *and Alex Delis*[3]

University of Athens, GR15748, Athens, Greece

E-mail: {*k.tsakalozos*[1]*, s.evangelatos*[2]*, ad*[3]*}@di.uoa.gr*

ABSTRACT

Contemporary multimedia, scientific and medical applications use indexing structures to access their high-dimensional data. Yet, in sufficiently high-dimensional spaces, conventional tree-based access methods are eventually outperformed by simple serial scans. Vector quantization has been effectively used to index data that are mostly distributed uniformly. However, in real-world applications, clustered data and skewed query distributions are the norm. In this paper, we propose *DiVA*, an approach that selectively adapts the quantization step to accommodate varying indexing needs. This adaptation mechanism triggers the restructuring and possible expansion of *DiVA* so as to provide finer indexing granularity and enhanced access performance in certain "hot" areas of the search space. User-supplied policies help both identify such "hot" areas and satisfy versatile application requirements. Experimentation with our detailed prototype shows that in a real-world data set, *DiVA* yields up-to 64% reduced I/O compared to competing methods such as the *VA*-file and the *A*-tree.

1. INTRODUCTION

A wide range of contemporary applications in the fields of scientific computing, multimedia retrieval, earth and space sciences, as well as bioinformatics operate on multi-dimensional data. In order to help speed-up the evaluation of queries in these high-volume/high-dimensional data sets, specialized indexes have been proposed [1, 2, 3]. Such indexing mechanisms are created using a set of feature vectors –collectively known as the "database"– and the adoption of a distance function. Two key operations for similarity searches in *n*-dimensional spaces are the range and the *k*-nearest-neighbor (*k-NN*) queries. In the former, all pertinent vectors within an area are retrieved while in the latter, the k-closest vectors to a query vector are retrieved.

In high-dimensional spaces, tree-based indexing methods are known to be occasionally outperformed even by simple serial scans [1, 2]. This "curse of dimensionality", has led to the introduction of approximation-based access methods that try to reduce search cost by *serially scanning* compact, approximate, representations of data. In this context, vector quantization has been established as an effective technique. *VA*-files have successfully exploited scans on approximate, quantized data to partially lift the dimensionality curse [4]. By and large, vector quantization approaches have assumed mostly uniformly-distributed data. Pre-processing of data has been proposed as a way to "smooth" skewed data so that approximation-based methods can work more efficiently [5]. However in real-world settings, not only skewed data but also clustered query distributions are frequently encountered [6].

One should keep in mind that the cost of serial scans, including scans performed on approximations, scales linearly with the volume of the indexed data. But hierarchical space partitioning methods can be used to allow better scaling against voluminous data sets. Due to this fact, a number of space partitioning methods that also employ approximations have been proposed. The *A*-tree [3] attempts to eliminate large areas of the search space by introducing the notion of virtual bounding rectangles (VBRs) which are tightly packed quantized minimum-bounding rectangles (MBRs). MBRs are also used in the three-level tree structure proposed in the *IQ*-tree [7]. In this structure the first level plays the role of a directory of MBRs pointing to the quantized pages of the second level. In similar spirit, the space partitioning employed by the *GC*-tree [8] attempts to provide higher indexing detail in areas of dense data distribution.

In all of the above indexing methods, hierarchical space partitioning is used in conjunction with data quantization. Yet, these methods are limited to predefined heuristics for constructing the index, ignoring any application specific needs. In this paper, we address this limitation. We propose *DiVA*, an indexing method whose operation allows for modular components to drive its expansion and structure. Using this approach, we implemented a policy that targets query turnaround time by reducing the overall I/O overhead. *DiVA* can facilitate access to clustered data and at the same time efficiently index specific areas that receive a high-traffic of fine-granularity queries. In summary, the contributions of *DiVA* are the following:

- *DiVA* decouples the index expansion from the struc-

This work has been partially supported by the *D4Science I & II* EU FP7 projects.

ture definition. The structure of the index is driven by application-specific policies that register with *DiVA* while the latter stays on-line. High-level application-specific requirements are allowed to help in tuning the method's performance with pertinent policies.

- It can selectively adapt its indexing granularity in specific sub-spaces. This is facilitated by its hierarchical and highly compact structure.
- *DiVA* uses multiple segments of approximated data which are ultimately scanned sequentially to yield data vectors relevant to the queries under evaluation.

Our experimentation with both synthetic and real data sets indicates that *DiVA* outperforms both the *VA*-file and the *A*-tree in terms of I/O load, achieving a notable improvement in clustered, high-dimensional spaces. In uniformly populated multi-dimensional spaces, *DiVA* matches the performance of the *VA*-file and consistently outperforms the *A*-tree in presence of highly-clustered data sets.

2. THE *DiVA* INDEX

DiVA is an unbalanced hierarchical structure whose every node resembles the *VA*-file. An unbalanced structure was chosen since in high-dimensional spaces, balanced structures generally result in large, ineffective bounding volumes. Approximated data are used to speed up the search within each node; every such node also contains data vectors. This approach aims to retain the good properties of *VA*-files in high dimensional spaces while *DiVA*'s hierarchical structure provides enhanced adaptation capabilities. Node creation and index structure may be controlled by pluggable application-specific policies. Finally, *DiVA* carries out I/O operations using only forward seeks so as to better exploit the underlying storage subsystem.

2.1. Structure and Operation of *DiVA*

DiVA uses a hierarchical structure of nodes whose each successive level provides greater indexing accuracy. Each node is similar to a *VA*-file and comprises of two files: *a)* a file with approximations termed *a-file* and *b)* a file of records termed *r-file* holding data vectors or pointers to other nodes.

Approximations stored in the *a-file* are produced through quantization of the corresponding data vectors. In essence, each approximation represents a rectangular cell in the indexed space. In Figure 1, we present one such 2-dimensional space indexed by *DiVA*. On the right side of Figure 1, we show the structure of *DiVA*. Cells *C1, C2, C3* have corresponding entries in the root node. Cell *C4* does not appear in the index since it does not contain any data vectors. For the approximations of *C2* and *C3*, there are two lists of data vectors. The list corresponding to *C2* contains data vectors *V2* and *V3*, while the list of *C3* consists of a single vector. Cell

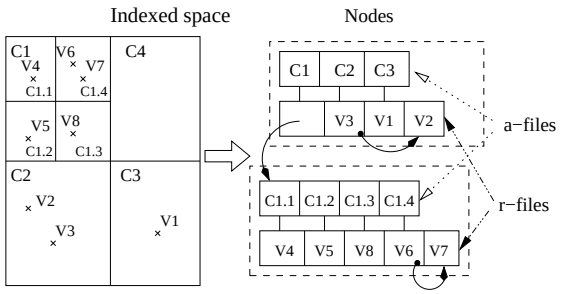


Fig. 1. Sample of a *DiVA* structure in a 2-D space.

C1 is further indexed by a second level node. The approximation of *C1* in the *r-file* of the root node points to a record which in turn points to a child node. The child node contains the approximations of cells *C1.1, C1.2, C1.3, C1.4* and the corresponding data vector lists in the child node's *r-file*.

Contrary to the *VA*-file, *DiVA* always stores each vector approximation only once, regardless of the number of data vectors in the approximation cell. In effect, vectors of the same cell are stored in a list of records formed inside the *r-file*.

A single record in the *r-file* may be either *a)* a pointer to a child node or *b)* part of a list of data vectors. All entries in a records list contain data vectors from the same space cell. The same applies to all data vectors encountered by following a pointer to a child node. Lower level nodes are used to further divide a cell into multiple cells with higher granularity. For instance in Figure 1, cells *C1.x* are used to subdivide cell *C1*.

Each stored approximation has a corresponding record in the *r-file*. The structure of *DiVA* allows us to store an arbitrarily high number of approximations per node. Yet, to guarantee the uniqueness of approximations, during insertion, the entire *a-file* has to be scanned. This insertion cost can be lowered by performing batch insertions of data vectors, an effective technique that we implemented and allowed us to speed-up our experiments significantly.

2.2. Algorithms

DiVA supports a full range of lookup operations including exact, range and *k-NN* queries. Here for brevity, we only present the *k-NN* search algorithms.

k-NN Query: Alg. 1 and 2 collaboratively perform the *DiVA k-NN* search starting from the root and then progressing through the nodes recursively. The results of the search, together with potential matching record numbers, are kept in the special container H. Every visited node, is searched in two distinct phases presented in Alg. 1 and Alg. 2. In Alg. 1, the *a-file* of the node is scanned sequentially to locate potential matching records. The recursive nature of the algorithm necessitates the tagging of the record numbers selected with the current node identifier, so as to differentiate them from record numbers referring to other nodes. Scanning the *a-file*

is interrupted if the approximation of the cell where the query vector belongs is encountered (line 6 Alg. 1). This cell is likely to contain the nearest neighbors of the query q, thus we choose to temporarily interrupt the approximations scanning and proceed with examining the relevant data vectors by issuing a call to Alg. 2. After this pause, we cancel the scanning of the rest of the approximations in the *a-file* if we are certain that there are no data vectors closer than the ones gathered so far. We perform this check by first making sure that we have gathered at least k results and comparing *a)* the distance of the query vector to closest border of the enclosing cell against *b)* the upper distance of the thus far k-th nearest neighbor to the query vector. During the second phase (Alg. 2), the corresponding records are retrieved from the *r-file*, starting from the ones with the most potential to be closer to the query vector q. If a pointer to a child node is found, a call to Alg. 1 is issued, using the child node as the starting query node anew. When the Alg. 1 ends, H contains the data vectors that form the results of the query.

H holds two types of elements: *a)* pairs of the form (*recno, node-id*), where *recno* is the identifier of a potentially matching record and *b*) data vectors. For all elements, an upper and a lower distance bound from the query vector q are maintained. For each matching approximation found in the *a-file*, the upper and lower distance bounds from the query vector q are computed and passed along with the corresponding record number to the container H (line 4 of Alg. 1). For a data vector, the upper and lower distance bounds coincide with the distance of the vector itself from q.

The elements in H are kept in ascending order based on their upper distance. The upper distance bound of the k-th element is maintained and cached as a number of important conditions depend on this value. Any element whose lower distance bound is less than or equal to the upper distance of the k-th element can be, or lead to, data vectors among the k nearest neighbors. Thus, at any moment, all items in the container H must satisfy Expr. 1:

$$lower\_bound(x) \leq upper\_bound(x_k), \forall x \in H \qquad (1)$$

Additionally, any item encountered that satisfies Expr. 1, is inserted in H (lines 5 Alg. 1 & line 18 Alg. 2). Note that, at any moment, more than k elements may exist in H.

As the algorithm progresses, element insertion and removal may cause the k-th element in H to change, leading to the gradual decrease of the upper distance bound of the k-th element. Consequently, a number of elements, that no longer satisfy Expr. 1 must be dropped from H. We refer to the process of dropping these elements as the *trim down* operation. Overall, Alg. 1 and Alg. 2 aim to minimize the number of record reads by visiting as early as possible, the areas closer to the query vector q.

Algorithm 1 *k-NN*_Search

Input: k: Number of nearest neighbors
q: Query vector
H: Heap-like container of intermediate results
n: Node whose approximations to scan
Output: H: Heap-like container of results

```
1:  for va ∈ { approximations in a-file of n } do
2:    if va.low(q) ≤ kth_element(H, k).up then
3:      recno := position of va in a-file of n
4:      H.insert(recno, va.up(q), va.low(q), id of n)
5:      H := trim_down(H, k)
6:      if va.low(q)= 0 then
7:        H := k-NN_ Data_scan(k, q, H, n)
8:        qva := approximate q using quantization of n
9:        if closest_border(qva, q) > kth_element(H, k).up then
10:         /*All k-NN were inside qva cell.*/
11:         return H
12:       end if
13:     end if
14:   end if
15: end for
16: H := k-NN_ Data_scan(k, q, H, n)
17: return H
```

3. ADAPTATION MECHANISM

Statistics in conjunction with numerous heuristics are commonly used to pinpoint "hot" areas [3, 7, 8]. But indexing granularity also largely depends on the needs of the application and can be affected by the characteristics of the underlying hardware. This suggests that index granularity and structure should not be determined only by examining data or query distributions. We allow the application to designate a suitable index expansion policy as well as to choose the appropriate timing for triggering the index adaption. Policies essentially designate the *DiVA* structure changes that should occur to improve performance. In case of a new node addition the policy should provide: a) Suggestions regarding which space cells should be further indexed ("hot" space areas). b) The per-dimension quantization step to be used in a newly created node. *DiVA* places all vectors that fall within the same cell in the same record list since they all produce the same approximation. Therefore, the data vectors that will be indexed with finer granularity are all part of a single list of records.

In what follows we present a policy that tries to minimize the wall-time incurred in the evaluation of queries and results in drastically reduced overall I/O overhead. This allows us to compare the performance of *DiVA* to other indexing approaches in terms of I/O.

Identifying "hot" spots: This policy computes a score for each list of records that may be indexed in finer granularity. The record-list with the highest score is selected for expansion. The score function evaluated for all candidate record

Algorithm 2 *k-NN_* Data_scan

Input: k: Number of nearest neighbors
q: Query vector
H: Heap-like container of intermediate results
n: Node whose records to scan
Output: H: Heap-like container of results

1: $O :=$ entries of H with id equal to that of n
2: **while** $O \neq \emptyset$ **do**
3: $(recno, low, up) :=$ element in O with minimum recno
4: remove from H entry identified by recno and id of n
5: $rec :=$ record in *r-file* of n at position recno
6: **if** rec is part of a data vector list **then**
7: $d :=$ dist(q, vector in rec)
8: **if** $d \leq$ kth_element(H, k).up **then**
9: H.insert(rec.vector, d)
10: **end if**
11: $next :=$ position of the next record in the data list of rec
12: **if** $next > recno$ **then**
13: H.insert($next$, up, low, n.id)
14: **end if**
15: **else if** rec points to child node with higher index granularity **then**
16: $H :=$ *k-NN*_Search($k, q, H, rec.child$)
17: **end if**
18: $H :=$ trim_down(H, k)
19: $O :=$ entries of H with id equal to that of n
20: **end while**
21: **return** H

lists is:

$$Score = Current - Future \tag{2}$$

The *Score* is estimated through a projection of the future cost (*Future*) and the current processing costs (*Current*) entailed in the operations performed within the record list examined.

While operational, the *DiVA* measures the average time R required to access and process a single record. Similarly, the average time s expended in dealing with a single approximation is also obtained. These two statistics capture the I/O and processing time required by the current hardware and software environment. The initialization cost o needed by the computer system for opening files hosting *DiVA* nodes is also gathered. Three extra statistics, l, h and q_s, are maintained on a per-record-list basis. A list containing l records is scanned during the evaluation of q_s queries. The total number of records of the list at hand that were part of the result in those q_s queries is denoted as h (hits).

Using the above statistics, the current list-scan cost for all queries q_s is:

$$Current = q_s R l \tag{3}$$

The estimation of the total future cost is more complex:

$$Future = q_s(o + Approx + Proj\_Reads) \tag{4}$$

where o indicates the average delays of initializing internal structures for accessing a node, *Approx* is the expected time expended for scanning through the approximations that will be created and *Proj_Reads* is the expected delays in reading the records of the new node.

The time required for scanning the approximations *Approx* is proportional to the number of new approximations because during an *a-file* scan all the approximations are typically examined. Under the assumption that in the new node each data vector will have a corresponding approximation, this number can be estimated. Eq. 5 shows the expected delays entailed in accessing and manipulating the approximations.

$$Approx = s \times l \tag{5}$$

Proj_Reads is an estimate of the per query delay cost entailed in accessing the data vectors of the new node. The expected reads consist of: *a)* the data vectors that will be part of the results (as estimated from previous query evaluations, h) and *b)* some additional vectors that will be read but not match the query (misses), denoted as m in Equation 6. The non-matching vectors (m) are estimated as follows: the hits of each query are assumed to be inside an n-dimensional cube. Any time a new node appears, it partitions the sub-space containing this n-cube in finer grained cells. The extra m elements read but dropped from the results are expected to be in cells that intersect with the surface of the n-cube. Considering the above, the per query expected time cost for reading records is:

$$Proj\_Reads = R(h + m)/q_s \tag{6}$$

At this point and under the uniform distribution assumption, we can estimate the density D of data vectors within each cell as: $D = l/2^{va\_bits}$ where *va_bits* is the bit length of each approximation in the new node.

Let B be the number of cells that intersect with the surface of the n-cube enclosing the results. On average, we expect only half the volume of these B cells to reside within the n-cube. Consequently $\frac{B \times D}{2}$ vectors residing in the aforementioned B cells will not match the query. So, Eq. 6 becomes:

$$Proj\_Reads = R(h/q_s + B \times D/2) \tag{7}$$

The edge e of the n-cube, can be expressed in terms of cells if we consider the n-cube's volume V to be proportional to the number of hits per query:

$$V = e^n = h/(q_s \times D) \Rightarrow e = \sqrt[n]{h/(q_s \times D)} \tag{8}$$

Using e, we are able to estimate B, since B is essentially equal to the surface of the n-cube:

$$B = 2ne^{n-1} = 2n(\frac{h}{q_s \times D})^{\frac{n-1}{n}} \tag{9}$$

With the help of Eq. 3, 4, 5, 7, 9 the score function in Eq. 2 is computed for each vector list, the one with the overall highest score is selected for further indexing.

Setting the quantization step: As soon as the highest scoring record list is identified the policy determines the per-dimension quantization step. The number of bits to be used for the approximations of the new node are dynamically set according to the standard deviation and dimensionality of the data being indexed. More bits are used for high dimensional spaces with vectors displaying low deviation. The bit allocation among dimensions follows a heuristic proposed in quantization theory that is also used in [5]. More bits are assigned to dimensions over which data have greater variance so as to maximize the efficiency of the quantized approximations.

4. EXPERIMENTAL EVALUATION

In our evaluation we measure I/O overhead as this is the metric used to measure the effectiveness of most multidimensional indexing methods. When *k-NN* queries are involved, we evaluate *DiVA* against Sequential Scan, the *VA*-file, the *A*-tree and an index we call *VAp*file. *VAp*file is our implementation of the most important features suggested by the *VA*+file, namely the Karhunen Loeve Transformations (KLT) and a dynamic bits-per-dimension allocation scheme. Both these features enhance the effectiveness of the *VA*-file when correlated data vectors are present in the data distribution. However, applying the transformation in the entire data space results in approximate query results.

4.1. Synthetic Data Set

We created a synthetic data set and measured the I/O performance while varying the following properties: *a)* dimensionality, *b)* volume of indexed data and *c)* percentage of clustered data vectors. The space we use as the *base case* consists of $200,000$ data vectors featuring 32 dimensions. Out of the $200,000$ vectors $50,000$ are uniformly distributed and the rest are grouped into 30 clusters. Members of each cluster follow the Gaussian distribution with $\sigma = 10^6$. With this σ value we seek to produce clusters that will be indexed using a gradually increasing quantization step. One tenth of the clusters is considered "hot" and is targeted by queries. The query load consists of *k-NN* queries with k=100.

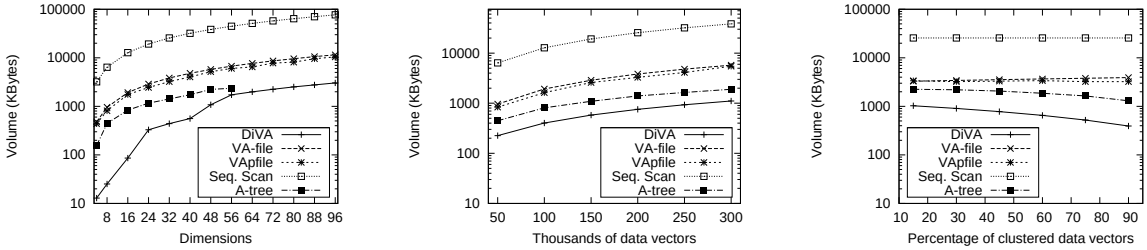
Dimensionality: we vary dimensionality from 4 to 96 and measure how *DiVA* copes with the curse of dimensionality in terms of I/O performance (Figure 2(a)). Increasing dimensionality results in larger data vectors, thus the total size of the indexed data increases. This trend is common for all indexing methods of Figure 2(a). *DiVA* performs grouping of multiple vectors under a single approximation, thus it manages to surpass the *VA*-file performance. With respect to the *A*-tree and while experimenting with its publicly available implementation [9], we were able to evaluate its performance only up to 56 dimensions. As shown, *DiVA* exhibits a clear advantage in the entire range of dimensions tested.

Volume of vectors: the volume of data affects indexes based on sequential search. *DiVA* does combine features from both sequential search and tree-based access methods. During this evaluation, we increase the amount of indexed data in such a way that the proportion between clustered and uniformly distributed vectors stays the same. We also keep the number of clusters in space fixed. Figure 2(b) shows the I/O load for each index in this evaluation scenario. Thank to their hierarchical structure, both *DiVA* and the *A*-tree are able to skip examining large volumes of data. Hence, they are less affected by increase in data volumes, compared to the *VA*-file based indexes.

Clustered vectors: *VA*-file is known to perform well when there are no clustered data [4]. *DiVA* on the other hand, employs its hierarchical structure to efficiently index both uniform and clustered data. Here, we vary the percentage of the clustered data while keeping the total number of indexed vectors fixed to $200,000$. Figure 2(c) presents the I/O load performance of all indexes. As data clustering increases, the *VA*-file is unable to exploit its approximations and thus, more data vectors have to be examined. The KLT applied by the *VAp*file provides an advantage over the *VA*-file as the cluster size increases. Under uniformly distributed data, the *A*-tree performs better than the *VA*-file but worse than *DiVA*. As more data are moved to the clusters, *DiVA* extends its performance lead.

4.2. Image Feature Vectors

The real data set used during evaluation consists of $200,000$ feature vectors extracted from images using methods similar to [10]. The dimensionality of this data set can be adjusted by altering the number of extracted features. The vector distribution produced clearly favors the *VA*-file. For each data vector a unique approximation is created using the 4 most significant bits from each dimension. In other words, the majority of space cells produced by the *VA*-file contain a single data vector. Thus, it makes little difference if KLT is used to treat data correlation by the *VAp*file. Yet, as shown in Figure 3, there is still room for improvement using *DiVA*. We assign only 2 bits per dimension for the approximations of the root node and then let the adaptation policy refine the indexing granularity. *DiVA* outperforms the *VA*-file based indexes by as much as 64% for a query load consisting of *range* queries. This is because with each new node *DiVA* increases the number of approximations read while at the same time reduces the number of read records that contain feature vectors (Fig. 4). Due to the fact that approximations are shorter than feature vectors and are hierarchically structured, the overhead sustained by introducing more approximations adds only 17% to the overall I/O overhead. At the same time, the fewer record reads reduce the overall I/O by 88%.



(a) Dimensionality impact on *DiVA*.

(b) *DiVA* under increasing index size.

(c) *DiVA* when clustered data increase.

Fig. 2. Performance evaluation of *DiVA*, *A*-tree, *VAp*file and *VA*-file using synthetic data sets.

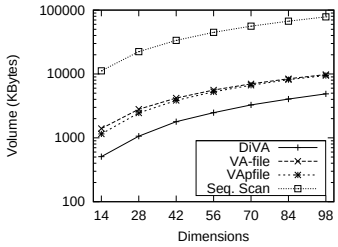


Fig. 3. I/O performance of *VA*-file and *DiVA* on real data set.

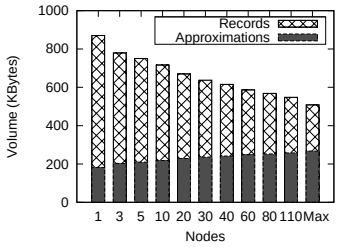


Fig. 4. *DiVA*'s I/O reduced as more nodes added.

5. CONCLUSIONS AND FUTURE WORK

DiVA offers fast navigation to query regions of interest and the structure of its nodes efficiently handles high-dimensional vectors. Experimentation with both real and synthetic data sets shows that *DiVA* produces significant improvements compared to competing methods. Our future work plans include: *a)* the use of advanced statistical models in determining the optimal granularity for node expansion and finally, *b)* the deployment of *DiVA* in parallel and distributed systems.

6. REFERENCES

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," in *Proc. of the 1990 ACM SIGMOD*, Atlantic City, NJ, May 1990.

[2] N. Katayama and S. Satoh, "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," in *Proc. of ACM SIGMOD*, Tucson, AZ, May 1997, ACM Press.

[3] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima, "The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation," in *Proc. of 26th VLDB Conf.*, Cairo, Egypt, Sept. 2000, pp. 516–526.

[4] R. Weber, H.-J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," in *Proc. of 24th VLDB Conf.*, San Francisco, CA, 1998.

[5] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi, "Vector Approximation-based Indexing for Non-uniform High Dimensional Data Sets," in *Proc. of the 9th CIKM Conf.*, McLean, VA, 2000, ACM.

[6] T. Barclay, D. Slutz, and J. Gray, "TerraServer: A Spatial Data Warehouse," in *Proc. of ACM SIGMOD Conf.*, Dallas, TX, 2000, pp. 307–318.

[7] S. Berchtold, C. Bohm, H.V. Jagadish, H-P. Kriegel, and J. Sander, "Independent Quantization: an Index Compression Technique for High-dimensional Data Spaces," in *Proc. of the 16th IEEE ICDE*, 2000.

[8] Guang-Ho Cha and Chin-Wan Chung, "The gc-tree: a high-dimensional index structure for similarity search in image databases," *IEEE Transactions on Multimedia*, vol. 4, no. 2, pp. 235–247, 2002.

[9] Y. Sakurai, "The A-tree: source code release," http://www.kecl.ntt.co.jp/csl/sirg/ people/yasushi /index.html, NTT Communication Science Laboratories, Seika, Soraku, Kyoto, Japan.

[10] Khanh Vu, Kien A. Hua, and Wallapak Tavanapong, "Image retrieval based on regions of interest," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 4, pp. 1045–1049, 2003.