The Computer Journal Advance Access published November 4, 2015

© The British Computer Society 2015. All rights reserved.
For Permissions, please email: journals.permissions@oup.com
doi:10.1093/comjnl/bxv097

# *DiVA*: Using Application-Specific Policies to 'Dive' into Vector Approximations

KONSTANTINOS TSAKALOZOS[1], SPIROS EVANGELATOS[2], FOTIS PSALLIDAS[3], MARCOS R. VIEIRA[4], VASSILIS J. TSOTRAS[5] AND ALEX DELIS[2]*

[1] *Microsoft Corp., London EC1N 2ST, UK*
[2] *University of Athens, Athens 15784, Greece*
[3] *Columbia University, New York, NY 10027, USA*
[4] *Hitachi America Ltd, R&D, Santa Clara, CA 95050, USA*
[5] *University of California, Riverside, CA 92521, USA*
**Corresponding author: ad@di.uoa.gr*

**In high-dimensional data domains, the performance of conventional tree-based access structures is occasionally outperformed by simple sequential scans. To this end, the introduction of approximation-based methods helped speed-up queries by providing compact representations of stored data. Approximation methods exploit vector quantization to index data mainly presumed to follow a uniform distribution. In real-world environments however, we mostly encounter both skewed data and query distributions. To address this dual challenge, we propose *DiVA* that combines the selective use of an approximation approach with an indexing mechanism to organize data subspaces in a high fan-out hierarchical structure. Moreover, *DiVA* reorganizes its own elements after receiving application hints regarding data access patterns. These hints or policies trigger the restructuring and possible expansion of *DiVA* so as to offer finer indexing granularity and improved access times in subspaces emerging as 'hot-spots'. The novelty of our approach lies in the self-organizing nature of *DiVA* driven by application-provided policies; the latter effectively guide the refinement of *DiVA*'s elements as new data arrive, existing data are updated and the nature of query workloads continually changes. An extensive experimental evaluation using real data shows that *DiVA* reduces up-to 64% of the total number of I/Os if compared with state-of-art methods including the *VA*-file, *GC*-tree and *A*-tree.**

*Keywords: self-organizing indexing methods; accessing skewed multidimensional data and queries; vector approximation techniques*

*Received 27 April 2015; revised 1 September 2015*
Handling editor: Fionn Murtagh

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

## 1. INTRODUCTION

Numerous applications in the fields of earth and space sciences, data analysis, scientific computing, multimedia retrieval, bio-informatics, among others, operate on multidimensional data [1–7]. To efficiently evaluate similarity-based queries, data objects from such domains are typically mapped into *data vectors* in *n*-dimensional spaces. To speed-up query evaluation in high-dimensional and large-volume datasets, specialized index methods have been proposed [8–14]. These indexing schemes use a set of feature vectors, representing objects in the original domain and a distance function to perform similarity searches. The two most common similarity search operations in such domains are the Range (i.e. 'retrieve all objects within a distance from a query object') and the *k*-Nearest-Neighbor (*k-NN*) queries (i.e. 'retrieve the *k*-closest objects to query object').

In high-dimensional spaces, tree-based indexing methods employed to evaluate similarity queries are ineffective [8]. This is due to the fact that index performance deteriorates rapidly as the number of dimensions increases. This renders

simple linear scan an efficient approach [15, 16]. This 'curse of dimensionality' problem has led to the introduction of approximation-based access methods [9, 17–21] whose aim is to reduce search costs by performing *sequential scan* on compact and approximate representations of data. In this context, vector quantization techniques, e.g. *VA*-files [15], are effective as they employ scans on approximate quantized data to partially lift the curse of dimensionality. By and large, most of the so far proposed vector quantization techniques assume uniformly distributed data. Thus, to overcome this drawback pre-processing data methods have been proposed as a way to 'smooth' skewed data so that approximation-based methods can work more efficiently [22, 23]. Nevertheless, not only skewed data but also clustered query distributions are frequently encountered in real-world settings [24, 25].

To curtail I/O costs originating from fruitless sequential scans in large datasets, a number of techniques [26–28] have attempted to leverage on hierarchical-based data partitioning structures. For instance, the *A*-tree [26] avoids large areas of the search by introducing the concept of virtual bounding rectangles (VBRs); they are tightly packed quantized minimum-bounding rectangles (MBRs). The *IQ*-tree [27] employs MBR in a three-level tree-structure that points to compressed representations of data vectors. The space partitioning employed by the *GC*-tree [28] provides higher indexing detail in areas of dense data distribution. Overall, the combined use of hierarchical space partitioning and data quantization has shown promise where traditional tree-based and simple approximation approaches do not perform well. Nevertheless, these methods are limited by their predefined and fixed heuristics set at index construction time. Moreover, these heuristics do ignore application requirements and possibly changing access patterns.

To addresses the above limitations, we present *DiVA* that blends two key characteristics as it:

(i) combines the selective use of an approximation approach with an indexing mechanism to manage data subspaces in a high fan-out hierarchical structure and
(ii) works in conjunction with application-provided 'hints' (or policies) and exploits changing query workloads to reorganize itself.

*DiVA* employs an adaptation mechanism that helps either 'zoom' (or 'dive') into its structure and provide access to objects with progressively increasing granularity, or restructure the index in order to more rapidly materialize queries in certain areas. External to *DiVA* software components termed observers, realize application-specific policies that drive *DiVA*'s adaptation mechanism. We describe two such policies: the first seeks to reduce query turnaround time by decreasing the overall I/O overhead, while the second favors specific groups of users.

We present a thorough experimental evaluation of our proposed *DiVA* method using a range of real and synthetic datasets. Our results show that *DiVA* outperformed well-known competing methods in terms of I/Os. For clustered high-dimensional spaces, *DiVA* achieves a notable improvement, while for uniformly distributed spaces, it yields performance comparable with its best competitor. In summary, in this paper we make the following contributions:

(i) we propose the *DiVA* indexing method that decouples index expansion from the rest of its query evaluation operations. *DiVA* is driven by application-specific policies registered while the index is kept on-line. In this way, the index may rapidly adapt to changes in the data distribution, or address changes in querying patterns that may require fine-granularity access to different subspaces. High-level application-specific requirements are allowed to help the index tuning.
(ii) *DiVA* selectively adapts its indexing granularity in specific subspaces. This is achieved by its hierarchical and highly compact structure;
(iii) *DiVA* uses multiple segments of approximated data that are sequentially scanned during the query evaluation. This strategy allows efficient storage of an arbitrary large number of approximations in each node, thus achieving good performance in high-dimensional datasets;
(iv) a prototype-based evaluation shows that *DiVA* outperforms a number of previously proposed access methods for high-dimensional data.

The rest of the paper is organized as follows: in Section 2, we review the related work. In Section 3, we formalize the main features of *DiVA*; Sections 4 and 5, describe how applications may dictate policies to guide the expansion and restructuring of *DiVA*. Section 6 discusses our experimental evaluation and Section 7 offers our conclusions.

## 2. RELATED WORK

Many indexing schemes for multidimensional data have been proposed [29, 30] so that queries such as range and *k-NN* are handled efficiently. Tree-based indexing methods attempt to speed-up query evaluation by visiting only fraction of their search space. To maintain the property that data objects under the same branch 'lie' within the same bounding rectangle or sphere, such methods deploy rather expensive updates; the effect of a single such update can affect the entire structure through repetitive nodes splits all the way to the root. The *X*-tree [31] attempts to reduce such costs by minimizing the overlap of bounding rectangles through the creation of specialized nodes that extend the capacity of overload nodes. The *SS*-tree [32] and *SR*-tree [8], respectively, use spheres and the combination of rectangles and spheres as bounding shapes to represent nodes. There is a similar line of structures whose operation is based on the distance of index objects to selected reference points. The *M*-tree [33] is a height-balanced

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

partitioning approach that organizes objects based on their distance to reference objects associated to nodes. The *iDistance* [34] uses a $B^+$-tree to index the distances of objects to reference objects. The *Omni-Family* [35] extends the idea to other index structures such as the *R*-trees. To minimize overlapping among sibling nodes, some indexes employ partitioning methods that generate non-overlapping cells. In this respect, the *Grid-file* [36] organizes cells that adapt to changing data object distributions while the *K-D-B*-trees [37] address diverse data distributions through space partitioning. Lastly, the *hybrid*-tree combines the advantages of both space and data partitioning approaches [38].

All above approaches suffer from the 'dimensionality curse' [17, 39, 40]. Under fairly common conditions and as dimensionality increases, sequential scan eventually outperforms all space partitioning and clustering methods [15]. The *VA*-file was proposed as an alternative method for indexing datasets in *n*-dimensional space. The *VA*-file is an array of vector approximations with each approximation being a quantized, compact representation of an original data vector. The $VA^+$-file [22] uses Karhunen–Loeve Transformations (*KLT*) and a scheme of flexible bit allocation among dimensions to efficiently differentiate vectors through pre-processing of data. Data vectors are pre-processed so they become more uniformly distributed among the *n* dimensions. The higher the standard deviation of data over a dimension is, the more bits are allocated for the quantization of dimension at hand.

The *IQ*-tree [27] uses a directory of bounding rectangles to limit the number of approximations read. Prior to its creation, the *IQ*-tree examines the entire dataset to produce its optimized directory up-front. The *VQ*-index [41] divides space into Voronoi cells according to queries received in order to produce *approximate answers*. The data vectors of each cell are compressed using vector quantization and are placed in a single file; only pertinent such files are accessed during querying. The *GC*-tree [28] partitions index space so that cells containing clustered data are identified and further indexed in the lower tree levels. Here, the cubic cells produced are mapped to disk pages. Cells containing clusters are split into more fine grained cells while sparsely populated cells are packed together into virtual pages.

The *A*-tree [26] seeks to combine the advantages of both *VA*-file and *SR*-tree [8]. An *A*-tree is a hierarchy based on VBRs that are derived from MBRs with quantization. In general, an MBR is enclosed within a VBR so that no vectors are dropped during quantization. VBR stored in the inner-tree nodes are represented in relation to their ascendant MBR so as to maximize storage efficiency. Due to their smaller size, numerous VBR can be packed in a data block and thus, the fan-out of *A*-tree is increased; due to this property, the *A*-tree node layout resembles that of *VA*-file while the presence of bounded rectangles underlines influence from the *SR*-tree.

In specific application domains, the quality of results may be traded in favor of faster response times [41–44]. Hence, indexing methods that use locality sensitive hash (*LSH*) has been proposed. *LSH* groups data vectors into bins which in turn help materialize *k-NN* queries and it does produce approximate query results [45–47]. *LSH* functions produce similar hash-values for close-by data vectors and so help efficiently generate approximate solutions to similarity queries. Approximate results may also be provided through dimensionality reduction methods [48]. We consider dimensionality reduction methods largely orthogonal to the indexing approaches. As a matter of fact, dimensionality reduction can serve as a valuable pre-processing step for indexed data; nevertheless, it can by no means replace indexing. In similar spirit, in [49] bloom filters are used to trade accuracy for performance.

Unlike the above hash-based mechanisms [45–47, 49, 50], *DiVA* provides exact answers to both *range* and *k-NN* queries. In contrast to *VA*-file, *DiVA* offers multiple levels of progressive index granularity through its hierarchical organization. *DiVA* differs from most tree-based counterparts in that a node may store an arbitrary high number of approximations which are sequentially scanned. In [26–28], tree nodes are mapped into disk blocks. In *DiVA*, a node consists of files that simply grow allowing for almost unlimited fan-out, i.e. *DiVA* delegates the allocation of more spaces for a node to the file system.

It is worth pointing out that previous existing methods [18, 19, 26, 27, 41] heavily rely on static and/or predefined rules (e.g. node utilization) to initiate the creation of new nodes and/or reorganize their structures. In contrast, *DiVA* reorganizes its structure on-the-fly according to application-provided policies. As mentioned earlier, such policies are based on query workload characteristics, application-provided hints and features of the host computer system.

### 2.1. The *VA*-file and skewed data

*DiVA* treats the 'dimensionality curse' in the same way as the *VA*-file, through vector approximations. While skewed data and queries are efficiently handled by structural changes in the hierarchy of its nodes. Here we describe the salient features of the *VA*-file, and discuss how an unbalanced structure can address its limitations.

As dimensionality of the dataset increases a serial scan is eventually more efficient than space partitioning and clustering [15]. For each point (vector) in the *n*-dimensional space, the *VA*-file produces an approximation vector that is ultimately used to improve the performance of the sequential scan. To construct these compact vector approximations, the space is divided into $2^b$ cells, with *b* being the bit length of each approximation. Each *VA*-file vector belongs to a single cell and each vector within a cell is approximated by the *b* bit representation of the corresponding cell. When more than one vectors fall within the same cell, the same representation is repeated in the approximations array. In this manner, the same approximation may appear more than once in the VA-file.

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

The idea behind *VA*-file is to process queries in a 2-phase filtering process: in the first phase, a sequential scan of all vector approximations locates candidate cells. For each approximation, the lower and upper distance bounds are computed; in the second phase, cells that are not pruned during the first phase are further examined. The trade-off between cost and accuracy of the sequential scan in the 2-phase filtering is controlled by the parameter $b$. As $b$ increases the space cells become smaller and we expect them to include less approximations. Smaller cells call for longer sequential scans (first phase) but less vectors to be examined within during the second phase. Since vector approximations are placed sequentially in the same file, the *VA*-file achieves good performance when pages are placed sequentially on the disk. The reason is that this approach makes efficient use of cache and prefetch mechanisms available in the current systems.

In light of clustered data however, large numbers of vectors tend to fall within the same cell as they feature identical approximations. Here, approximations offer limited capacity in differentiating among the populous clustered data vectors that now have to be all retrieved. Further more, the *VA*-file offers the same index granularity ($b$ bits) across the entire dataset. Thus, there is no differentiation between 'hot' and 'cold' space areas.

To alleviate the above limitations, *DiVA* adopts a hierarchical structure where higher index granularity is offered through lower level nodes. Such granularity adjustments accommodate both skewed data as well as skewed queries. Quantization is employed to produce vector approximations, which are stored in nodes of a hierarchical structure. Extra nodes are used to index particular space areas and bare significant resemblance to *VA*-files. Each child node provides higher index detail over a portion of the area indexed by its parent. The end structure is an unbalanced tree (more levels are used to index 'hot' areas). Balancing such structure would needlessly increase index granularity across the entire search space. Moreover, 'hot' areas should be scanned first, to accommodate this the proposed structure intentionally restructures itself in an unbalanced form.

### 2.2. Earlier work on *DiVA*

A preliminary version of our work appeared in [51]. Compared with this, the current extended version entails the following:

(i) in-depth discussions on several key architectural aspects of *DiVA*: We present how the event-based mechanism enables the implementation of multiple application driven policies. Section 4.2 outlines the tuning options of such adaptation policies. We also present and evaluate the favoring groups of users (FGU) policy (Section 5.2) that allows for specific users groups to be favored over others.

(ii) detailed presentation of important core operations: Sections 3.2 and 3.3 outline the data layout and the way we manipulate bits in ways that help achieve the sought performance. The behavior of *DiVA*'s internal components is evaluated in Section 6.1.

(iii) in-depth discussions on the interface operations of the proposed index. In addition to the *k-NN* algorithm, we also present the implementation of the range search as well as the vector insertion algorithm along with pertinent complexity analyses.

(iv) improved and comprehensive experimental evaluation of our approach: Using prototypes we have developed, we carried out a wide range of diverse experiments that reveal both pros and cons of *DiVA* and other examined techniques. The latter include the competing *GC*-tree [28] access method as well as the *VA*-file, the *VA*+-file and the *A*-tree.

(v) comprehensive comparison with prior work in the area: This related work section outlines a number of prior research efforts and qualitatively compares how our proposed *DiVA* approach advances the work through its unique feature of 'diving' fast into query regions of interest as well as its *DiVA*'s hint-based application driven reorganization method.

## 3. THE *DIVA* INDEX

*DiVA* is a non-balanced hierarchical structure whose every node resembles to the *VA*-file. An non-balanced structure was chosen since in high-dimensional spaces, balanced structures generally result in large, ineffective bounding volumes. Approximation data are used to speed up the search within each node; every such node also contains data vectors. This approach aims to combine the good properties of *VA*-files in high-dimensional spaces with *DiVA*'s hierarchical structure that offers enhanced refinement indexing capabilities. Node creation and index maintenance are controlled by application-specific policies. Finally, *DiVA* by design carries out I/O operations using only forward file seeks so as to better exploit the underlying storage subsystem.

### 3.1. Structure and operation of *DiVA*

Non-uniform datasets accessed via skewed query access patters necessitate fine(r) index granularity in corresponding data subspaces. *DiVA* uses a hierarchical structure of nodes whose every successive level provides for greater indexing accuracy. A *DiVA* node is similar to the organization of the *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. This lossy approximation process, introduces a degree of uncertainty in

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

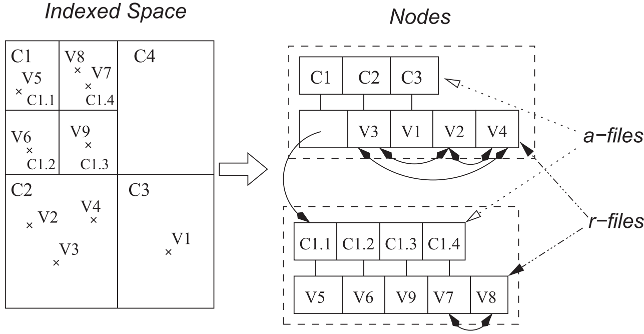


**FIGURE 1.** Sample of a *DiVA* structure in a 2D space.

the exact location of the original data vector. In essence, each approximation represents a rectangular cell in the indexed space. In Fig. 1, we present one such 2D space indexed by *DiVA*. Each vector *Vx* belongs to a cell according to the prefix of its coordinates. Thus, data vectors belonging to the same cell, have also the same approximation. On the right side of Fig. 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*, *V3* and *V4*, while the list of *C3* consists of a single vector. Cell *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 distinct 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*. For example, vectors *V3*, *V2* and *V4* are placed in a list for they belong to cell *C2*.

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 Fig. 1, cells *C1.x* are used to subdivide cell *C1*.

Each stored approximation has a corresponding record in the *r-file*. In our current implementation, a record is an array of dimension values. The size of the array is the same as the number of dimensions (one value per dimension).[1] By having the *n*th approximation correspond to the *n*th record in the *r-file*, we eliminate the need for pointers in the *a-file* which in turn allows for tighter packing of approximations. The absence of pointers from the *a-file* to corresponding records in the *r-file* may demand an occasional record relocation during insertion. During such a data vector insertion, any record already occupying the corresponding position of the new approximation has to be moved to the end of the *r-file*. Two pointers, stored in each record, are used to connect the vectors of the same cell in a circular doubly linked list. This doubly linked-list organization renders the relocation of records a constant cost operation.

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; we have implemented this batch insertion technique to speed-up our experimentation.

### 3.2. Approximations and data packing

*DiVA* enables us to provide higher indexing granularity for specific areas of interest. Varying levels of vector quantization allow us to adapt the indexing granularity. Each $n$-dimensional data vector $v$ is a sequence of coordinates $\{c_1, \ldots, c_{n-1}, c_n\}$ and in turn, for dimension $i$, each coordinate $c_i$ is represented by a sequence of bits $(b^i_l, b^i_{l-1}, \ldots, b^i_1)$ with $l$ being the bit length of the coordinate at hand. Starting from the root node, the most significant bits of each dimension are used to construct the approximations. Moving to lower levels in the index hierarchy, more bits are used for the quantization process. The number of bits used from each coordinate/dimension adjusts the index granularity.

The root node approximations follow the form:

$$\{(b^1_l, b^1_{l-1}, \ldots, b^1_{r_1}), (b^2_l, b^2_{l-1}, \ldots, b^2_{r_2}), \ldots,\\ (b^i_l, b^i_{l-1}, \ldots, b^i_{r_i}), \ldots, (b^n_l, b^n_{l-1}, \ldots, b^n_{r_n})\}$$

where $r_i$ is the number of bits used to build the approximation for the $c_i$ coordinate. Approximations of the above form have a bit length $m = \sum_{i=1}^{n}(l - r_i + 1)$ and partition the indexed space into $2^m$ cells. A child node would provide higher granularity through approximations that use more bits than its parent. That is done by choosing $s_i \leq r_i, \forall i \in [1, n]$. In producing the approximations at hand, the dimensions where $s_i = r_i$ are ignored as no extra bits are used. Yet, we must have at least one dimension contributing a minimum of at least one additional bit. Thus, we require that $\exists i \in [1, n] : s_i < r_i$. The approximation of a child node is of the following form:

$$\{(b^1_l, b^1_{l-1}, \ldots, b^1_{s_1}), (b^2_l, b^2_{l-1}, \ldots, b^2_{s_2}), \ldots,\\ (b^i_l, b^i_{l-1}, \ldots, b^i_{s_i}), \ldots, (b^n_l, b^n_{l-1}, \ldots, b^n_{s_n})\}$$

Each child node provides further indexing detail to a single cell of the parent node. Data vectors stored in the child node always 'fall' within the parent cell. Therefore, the most significant bits per dimension of the produced approximations are common for all approximations of the child node. These

[1] It is trivial to replace the above vector representation to any object as long as we have away to extract the dimension values.

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

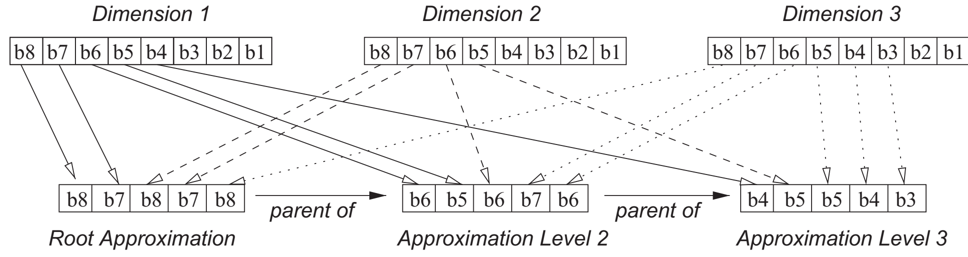


**FIGURE 2.** A 3D, 8 bit vector along with approximations for the root and two children nodes.

common bits are stored only once, in the *a-file* of each node, and are not repeated in every approximation. Consequently, the aforementioned child approximations that are ultimately stored are of the form:

$$\{(b^1_{r_1-1}, b^1_{r_1-2}, \ldots, b^1_{s_1}), (b^2_{r_2-1}, b^2_{r_2-2}, \ldots, b^2_{s_2}), \ldots, (b^i_{r_i-1}, b^i_{r_i-2}, \ldots, b^i_{s_i}), \ldots, (b^n_{r_n-1}, b^n_{r_n-2}, \ldots, b^n_{s_n})\}$$

The per dimension quantization steps may vary from node to node and are decided upon node creation. *DiVA* can produce suitable quantization steps by performing statistical analysis[2] on the data and queries within the area of the node that is being created.

In Fig. 2, we present a 3D data vector along with approximations of three levels. For the first dimension, the root approximation uses the two most-significant bits, the first child approximation uses the third and fourth bits, while the third-level approximation uses only the fifth bit. As Fig. 2 depicts, we concatenate bits of all three dimensions to form the approximations of the three levels. The gradual increase of the used bits in child nodes results in progressively smaller, non-cubic, cells.

### 3.3. A rapid binary 'within range' operation

During a range query, the search algorithm has to be able to rapidly determine whether a given data vector falls within the query area. The implementation of this '*within range*' operation largely depends on the shape of the query area. In *DiVA*, we define the range query area as the area enclosed in an $n$-dimensional rectangle, rectilinear to the coordinate axes. A rectangle $P$ is defined by providing two of its opposite vertices, most commonly the vertices $p^{\min}$ and $p^{\max}$, which are the closest and farthest vertices to the origin, respectively. The test to determine if a vector $v$ is located inside a rectangle $P = (p^{\min}, p^{\max})$, as given in Equation (1), requires $2n$ comparisons:

$$inside(P, v) = \bigcap_{i=1}^{n} p_i^{\min} \leq v_i \leq p_i^{\max} \tag{1}$$

[2] Outlined in Algorithm 5.

*DiVA* has to determine if a vector falls within a query area using only a vector approximation. Each approximation corresponds to a cell, with a certain volume, in the indexed space. An approximation cell is identified by a quantized vector $va$ and has length $d_i$, equal to the quantization step, in each dimension. An approximation $va$ can be entirely contained within a rectangle $P$ or partially overlap with $P$ (Equations (2) and (3)). If an approximation is contained in the query area $P$, we can safely conclude that the original vector is within the same area.

$$contains(P, va) = \bigcap_{i=1}^{n} p_i^{\min} \leq va_i \leq p_i^{\max} - d_i \tag{2}$$

$$overlaps(P, va) = \bigcap_{i=1}^{n} p_i^{\min} - d_i \leq va_i \leq p_i^{\max} \tag{3}$$

The aforementioned checks (Equations (2) and (3)) requires first *unpacking* the approximated data to reconstruct the coordinates $va_i$ of the quantized vector. The reconstruction requires a series of bitwise masking, shifting and *XOR* operations that have a significant CPU overhead. To address this issue, we develop a more relaxed yet faster predicate which can quickly disqualify non-needed vector approximations. We call this predicate 'quick within-range'. The approximations that pass this test have to be unpacked and checked using Equations (2) and (3).

The *quick within-range* predicate is based on the principle that the binary representations of all numbers within a certain range, have a common prefix. Checking a binary number $c$ for a known prefix $\{p_n, p_{n-1} \cdots p_k\}$ requires one bitwise *AND* ($\otimes$) operation with a mask $m = \{1_n, \ldots, 1_k,\ 0 \cdots 0\}$ to isolate the most significant bits of $c$ and a comparison of the result to the value $p = \{p_n, p_{n-1} \cdots p_k,\ 0 \cdots 0\}$:

$$c \otimes m = p \tag{4}$$

If Equation (4) holds true, then Equation (5) is also true. This means that the binary operation of Equation (4) is equivalent to that of (5):

$$\sum_{i=k}^{n} p_i 2^i \leq c \leq \sum_{i=k}^{n} p_i 2^i + \sum_{i=0}^{k-1} 2^i \tag{5}$$

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

Any number within a range $[a,b]$ has the same prefix as the longest common binary prefix of $a$ and $b$. In other words, some of the most significant bits of any number in that range are known. By applying this property on the $n$ inequalities of Equation (3), we construct $n$ bitmasks $m_i$ that isolate the common prefixes from the boundaries of each interval. We combine these values in a vector $m = (m_1, \ldots, m_n)$ and by approximating vector $m$ we obtain a mask $m_a = approximate(m)$. When $m_a$ is applied on an unpacked vector approximation, it isolates the bits whose expected values are already known. These bits are expected to have the same values with $q_a = m_a \otimes approximate(p_x)$, where $p_x$ is one of the vertices of the query area. Finally, any packed approximation $v_a$ can be checked against Equation (6) using only a bitwise *AND* followed by an equality operation.

$$v_a \otimes m_a = q_a \tag{6}$$

If Equation (6) is not true for a given vector approximation, we can then rapidly conclude that the vector in question does not overlap with the query area; thus, it is excluded from further examination.

The effectiveness of the 'quick within-range' predicate depends on the number of common bits in the two data points defining the query range. In a query range with a mask of $f$ bits turned on and data distributed uniformly, we expect to find vector approximations that have to be unpacked with a probability of $1/2^f$.

## 4. APPLICATION-DRIVEN *DIVA* REORGANIZATION

Statistics along with heuristics are commonly used to pinpoint 'hot' areas in search spaces [26–28]. In practice however, index granularity and performance of the corresponding structure are predominantly affected by factors that become known at runtime. These factors include dynamic characteristics exposed by the application layer as well as features of the underlying hardware. Hence, conventional approaches that predominantly use data and query distributions to identify hot-spots are not always effective [26–28]. As application requirements are impossible to be known *a priori*, we introduce the notion of user-provided policies that help decide when and how to reorganize *DiVA*. These policies are realized within software components we call *observers*.

An observer monitors *DiVA*'s operation and intercepts events that occur during query evaluation. Using this information, an observer helps realize a policy to identify highly contented subspaces and guide index expansion and restructuring by providing 'hints'. Figure 3 shows the interaction among observer, index and application. Applications are expected to designate an observer (and thus a refinement policy) that suits their needs. We should indicate that *DiVA* can function under the combined regime of multiple policies through the simultaneous operation of multiple observers.

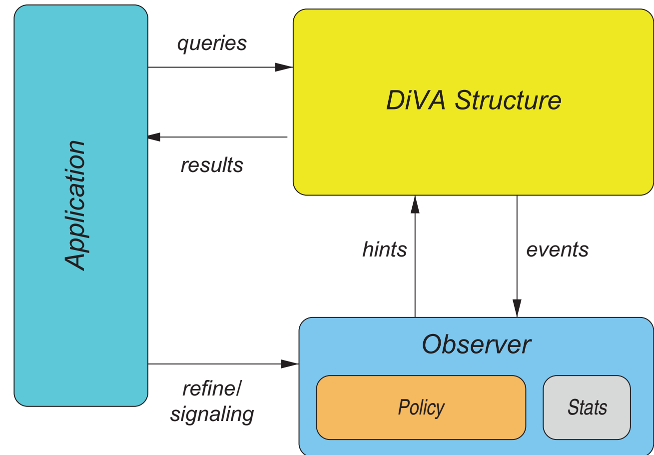


**FIGURE 3.** Interaction of *DiVA* elements with application layer.

Observers implement an interface through which *DiVA* can let them know about events occurring during query evaluation. Events that get reported include:

(a) start and end of query evaluation,
(b) time when a node is accessed,
(c) scan time of a record representing a cell,
(d) number of approximations and data vectors examined,
(e) number of candidates identified during an *a-file* scan and
(f) final set of results.

All above events are accompanied by a *session ID* so that statistics of different user queries can be readily identified. A description of all events is provided in Section 4.1 where we present *DiVA*'s range and *k-NN* search algorithms.

As index refinement is deemed costly, we allow the application to determine the most suitable period during which *DiVA* can carry out its reorganization. Once index refinement is triggered, the observer identifies existing hot-spots thus far. *DiVA* tunes its query performance so that traffic on such hot-spots can be addressed.

The cost of sending/receiving events between *DiVA* and an observer is the penalty incurred by a single *C++* function call. In general, we anticipate that an observer will be mostly a light-weight process as it intercepts and stores simple facts. Occasionally, when an application necessitates *DiVA*'s restructuring the respective observer may have to carry out significant processing to identify hot-spots. This lengthy operational costs affects the time required to complete the reorganization in question. As observers realize application-specific policies for which *DiVA* remains agnostic, accurate overhead estimations for the restructuring cannot be readily established. Anytime the application seeks a new performance optimization, a respective observer should be realized and register with *DiVA*; as soon as



Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

the service of an observer is no longer of value, the application may signal its termination (Fig. 3).

Below in Section 4.1, we describe the search *DiVA* algorithms along with the events occurring during query evaluation. Section 4.2 outlines two low-level refinement operations that heavily influence and assist toward an effective query evaluation process.

### 4.1. Algorithms

*DiVA* supports the full-range of lookup operations including exact, range and *k-NN* queries. We present here the algorithm to insert a data point as well as the range and *k-NN* search operations. The exact query algorithm is similar to the data insertion.

*Insertion:* The insertion of a data vector, as presented in Algorithm 1, descends recursively into the hierarchical structure until it finds the node where the vector must be stored. Initially, the algorithm computes the approximation of the new vector using the quantization step designated for the node at hand. Subsequently, the *a-file* is scanned for a match of the produced approximation. This may result in three possible outcomes: (a) the approximation exists and pertinent data vectors are stored in the current node, (b) the approximation exists but it corresponds to a child node with higher index granularity and (c) the approximation does not exist in the *a-file* of the node examined. In the first two cases, at least one data vector with the same approximation has already been indexed. When the approximation found corresponds to a list of records, the new data vector has to be appended in the *r-file* as a new record of the list at hand. This procedure, denoted as 'append' in line 7, ensures that the list can be traversed by moving only forward in the *r-file*. The latter results in reduced I/O seek overhead when reading the list. When the approximation exists but the corresponding record is pointing to a child, the insertion progresses recursively in the child node (line 9). When an approximation is not found in the current node, the data vector belongs to a cell which is currently empty. Thus, the new vector has to be added as a new entry in the node at hand. This requires appending the produced approximation to the *a-file* and placing the data vector to the corresponding position of the *r-file*.

When dealing with a large number of insertions, Algorithm 1 becomes inefficient as it has to traverse the hierarchical structure for each insertion. We address this issue by introducing a batch insertion procedure that takes as input a list of data vectors and a node to insert the vectors to. We produce approximations for all data vectors and place them in a hash-table. For each approximation of the *a-file*, we perform a look-up in the hash-table and a potential match is handled identically to the steps in lines 6–10 and 16–21 of Algorithm 1. The above approach greatly reduces cost as a large number of vectors can be inserted within a node in a single pass through its *a-file*.

Deleting a data vector has the same cost as the insertion. However, deleting data vectors cause the respective approximations

**Algorithm 1** Insertion

**Input:** $n$: Starting insertion node
$v$: Data vector to insert

1: va := approximate $v$ using quantization of $n$
2: **for** approx $\in$ {approximations in *a-file* of $n$} **do**
3:   **if** approx = va **then**
4:     *pos* := position of approx in *a-file* of $n$
5:     *rec* := record in *r-file* of $n$ at position *pos*
6:     **if** *rec* is a data vector list **then**
7:       append $v$ in list of *rec*
8:     **else if** *rec* points to child node with higher index granularity **then**
9:       **call** insert(*rec*.child, $v$)
10:     **end if**
11:     */*v is now inserted*/*
12:     **return**
13:   **end if**
14: **end for**
15: */*approximation va not found in node n. Adding it.*/*
16: append va in *a-file* of $n$
17: target := position of just appended va in *a-file* of $n$
18: **if** position target in *r-file* of $n$ is occupied **then**
19:   relocate record from position target to end of *r-file* of $n$
20: **end if**
21: store $v$ at position target in *r-file* of $n$

within the *a-files* to point to empty records within the *r-file*. Such stray pointers hamper the efficiency of the index. To recover from this inefficient state, we need a compaction phase that would eliminate the approximations pointing to empty space cells. The cost of this operation is significant and depends on the fragmentation ratio present in the *r-file*. Updating a single data vector calls for a deletion and a reinsertion of the data vector as new.

*Range search:* Algorithm 2 performs the *DiVA* recursive range search. There are three distinct phases when searching a node: first, the *a-file* is scanned, to locate candidate approximations (lines 3–9). Second, the records corresponding to the candidate approximations are examined (**while** block between lines 11–24). And third, children nodes are visited (**for loop** in line 26). This three-phase algorithm ensures that all *a-files* and *r-files* involved are accessed one at a time and in a record-number ascending order; this reduces the I/O time by avoiding costly random seeks.

During the *a-file* scan phase, all space cells, represented by each approximation, are checked so as to identify those that are within the range in question; this check exploits the 'quick within range' predicate of Section 3.3. Candidate cells may either partially overlap with the desired range or be entirely contained in the query area. This information is exploited as follows: data vectors belonging to a cell entirely contained in the query area are added to the results without any further check.

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

**Algorithm 2** RangeSearch

**Input:** $r$: Query range
$n$: Starting query node
`sessionID`: The application level ID associated with the submitted query
**Output:** $R$: Set of query results

1: $O \leftarrow \emptyset$; $C \leftarrow \emptyset$
2: ***sendEvent***(rangeStart, *sessionID*, $n$.id, $r$)
3: **for** $i \in [1,\ len(n.\text{approx})]$ **do**
4: $va \leftarrow n.\text{approx}[i]$
5: **if** $r$.overlaps($va$) **then**
6: $contained \leftarrow r.\text{contains}(va)$
7: $O \leftarrow O \cup \{(i, contained)\}$
8: **end if**
9: **end for**
10: ***sendEvent***(approxScan, *sessionID*, $n$.id, $O$, $len(n.\text{approx})$)
11: **while** $O \neq \emptyset$ **do**
12: $(i, contained) \leftarrow min(O)$
13: $O \leftarrow O - \{(i, contained)\}$
14: rec $\leftarrow n.\text{records}[i]$
15: ***sendEvent***(recordRead, *sessionID*, $n$.id, $i$, rec)
16: **if** (rec.isVlist() **and** *contained*) **or** (rec.isVlist() **and** $r$.contains(rec.vector)) **then**
17: $R \leftarrow R \cup \{\text{rec.vector}\}$
18: **else if** rec.isPointer() **then**
19: $C \leftarrow C \cup \{(\text{rec.child}, contained)\}$
20: **end if**
21: **if** rec.isVlist() **and** rec.hasNext() **then**
22: $O \leftarrow O \cup \{(\text{rec.next}, contained)\}$
23: **end if**
24: **end while**
25: ***sendEvent***(recordScan, *sessionID*, $n$.id, $C$)
26: **for all** $(child, contained) \in C$ **do**
27: **if** *contained* **then**
28: $R \leftarrow R \cup \text{allVectorsBeneath}(child)$
29: **else**
30: $R \leftarrow R \cup \text{rangeSearch}(r, child)$
31: **end if**
32: **end for**
33: ***sendEvent***(rangeStop, *sessionID*, $n$.id, $R$)
34: **return** $R$

Set $O$ in Algorithm 2 is populated with the record numbers of candidate records that need to be visited. Each record number is annotated with a flag indicating if the corresponding approximation is entirely contained within the query area. All candidate records are accessed in a record-number ascending order so that the *r-file* is scanned using only forward seeks. That is achieved with the help of function *min* which in every loop retrieves the record with lowest number from set $O$. If the record examined is part of a data vectors list with consecutive nodes, line 22 inserts in $O$ the record number of the next record in the list. This is done in order to ensure that records are visited in the correct order. Upon reading a record that carries a data vector, the vector is checked to determine if it is contained in the query area. This check is omitted if the approximation cell is marked to be entirely contained within the query area.

Records pointing to children nodes are kept in the $C$-set and are recursively visited during the last phase of the algorithm. In this phase, if a child node is known—with the help of the flag *contained*—to lead to data vectors residing in a cell entirely within the query area, the subtree under that child node is added to the results without performing any further checks. This is accomplished through the *allVectorsBeneath* call. If the approximation of the child node is partially within the query area, data vectors under this node are examined via a recursive call to the range search algorithm.

Observers are informed of the progress of a range search through the ***sendEvent*** calls in lines 2, 10, 15, 25 and 33; the intercepted events indicate the start and finish of each of the range search phases and also the accessing of a record in the *r-file*. The first three input parameters of each such call are:

(i) a string literal indicating the type of the message sent,
(ii) a session ID used to associate the query with an application level activity and
(iii) the ID of the *DiVA* node that the algorithm examines when the event is sent.

In addition to these three parameters, each ***sendEvent*** informs the observer in use of several search internal metrics that compute search efficiency statistics. More specifically, the event in

(i) line 2 provides the query,
(ii) line 10, dispatched after the first phase, includes the approximations selected and the total amount of approximations checked,
(iii) line 15 includes the record read and its position in the *r-file*,
(iv) line 25 indicates the end of the records scanning phase and also includes the child nodes that we have to visit, and finally,
(v) line 33 marks the end of the query evaluation and also includes a pointer to the results gathered.

*k-NN Query:* Algorithms 3 and 4 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 heap-based container $H$. Every visited node, is searched in two distinct phases presented in Algorithms 3 and 4, respectively. In Algorithm 3, 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 (*recno*) with the current node identifier (*node-id*), so as to differentiate them from record numbers referring to other nodes. Scanning the *a-file* is interrupted if the approximation of

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

**Algorithm 3** *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
`sessionID`: The application level ID associated with the submitted query
**Output:** $H$: Heap-like container of results

```
1:  sendEvent(knnStart, sessionID, n.id, q, k)
2:  for va ∈ {approximations in a-file of n} do
3:    if va.low(q) ≤ kthElement(H, k).up then
4:      recno ← position of va in a-file of n
5:      H.insert(recno, va.up(q), va.low(q), id of n)
6:      H ← trimDown(H, k)
7:      if va.low(q)= 0 then
8:        H ← k-NN DataScan(k, q, H, n, sessionID)
9:        qva ← approximate q using quantization of n
10:       sendEvent(knnDepth, sessionID, n.id, recno,
          va, H)
11:       if closestBorder(qva, q) > kthElement(H, k).up
          then
12:         /*All k-NN were inside qva cell.*/
13:         sendEvent(knnStopDepth, sessionID, n.id, H)
14:         return H
15:       end if
16:     end if
17:   end if
18: end for
19: H ← k-NN DataScan(k, q, H, n, sessionID)
20: sendEvent(knnStop, sessionID, n.id, H)
21: return H
```

**Algorithm 4** *k-NN* DataScan

**Input:** $k$: Number of nearest neighbors
$q$: Query vector
$H$: Heap-like container of intermediate results
$n$: Node whose records to scan
`sessionID`: The application level ID associated with the submitted query
**Output:** $H$: Heap-like container of results

```
1:  sendEvent(dataScanStart, sessionID, n.id)
2:  O ← entries of H with id equal to that of n
3:  while O ≠ ∅ do
4:    (recno, low, up) ← element in O with minimum recno
5:    remove from H entry identified by recno and id of n
6:    rec ← record in r-file of n at position recno
7:    sendEvent(recordRead, sessionID, n.id, rec)
8:    if rec is part of a data vector list then
9:      d ← dist(q, vector in rec)
10:     if d ≤ kthElement(H, k).up then
11:       H.insert(rec.vector, d)
12:     end if
13:     next ← position of the next record in the data list of rec
14:     if next > recno then
15:       H.insert(next, up, low, n.id)
16:     end if
17:   else if rec points to child node with higher index
      granularity then
18:     H ← k-NN Search(k, q, H, rec.child)
19:   end if
20:   H ← trimDown(H, k)
21:   O ← entries of H with id equal to that of n
22: end while
23: sendEvent(dataScanStop, sessionID, n.id, H)
24: return H
```

the cell where the query vector belongs is encountered (line 7 Algorithm 3). 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 Algorithm 4. After this interruption, 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 firstly, making sure that we have gathered at least $k$ results and secondly, by comparing the distance of the query vector to closest border of the enclosing cell against the upper distance of the thus far $k$th nearest neighbor to the query vector. In Algorithm 4, the corresponding records are retrieved from the *r-file*, starting from the ones with the greatest potential to be closer to the query vector $q$. If a pointer to a child node is found, a call to Algorithm 3 is issued, using the child node as the starting query node. When Algorithm 3 ends, $H$ contains the data vectors that form the results of the query.

$H$ holds two types of elements: (a) pairs of the type (*recno, node-id*), where *recno* is the identifier of a potentially matching record and (b) data vectors. For every element in $H$, regardless its type, we compute its upper and lower distance bound from the query vector $q$. For a data vector, the distance bounds coincide with the distance of the vector itself from $q$. In the case of a matching approximation, the span between lower and upper distance bound is the result of the uncertainty introduced by the lossy approximation process. Here the bounds computed are passed along with the corresponding record number to the container $H$ in line 5 of Algorithm 3.

The elements in $H$ are kept in ascending order based on their upper bound distance. 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, if $x_k$ is the $k$th element in $H$, all other items in the container $H$ must satisfy the following equation:

$$lowerBound(x) \le upperBound(x_k), \quad \forall x \in H \tag{7}$$

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

Additionally, any item encountered that satisfies Equation (7), is inserted in $H$ (line 6 Algorithm 3 and line 20 Algorithm 4). Note that, at any moment, more than $k$ elements may exist in $H$.

As *k-NN* 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 Equation (7) must be dropped from $H$. We refer to the process of dropping these elements as the *trim down* operation. Overall, Algorithms 3 and 4 seek to minimize the number of record reads by visiting as early as possible, the areas closer to the query vector $q$.

Similar to the events dispatched during the range search, the events of the *k-NN* algorithm also include a string literal, the application-provided session ID and the node ID we are examining. In particular:

(i) The events of lines 1 and 20 of Algorithm 3 mark the start and finish of the query evaluation, thus they include the query details and the results. As our search performs a depth first search in cells containing the query vector, we provide two additional events to inform observers of such activity.
(ii) Line 10 of Algorithm 3 sends an event indicating that a 'dive' is performed and we also mark the approximation (cell) that caused the search in depth. In case this activity yields all $k$-nearest neighbors, we inform a pertinent observer of the final results with the event in line 13.
(iii) In Algorithm 4, we have two events that mark the start and finish of the data scan (lines 1 and 23) and an event triggered each time we access a record (line 7).

The computational complexity of the *k-NN* search algorithm is affected by the overhead added from observers processing the events. However, we can estimate the complexity when no observers are present. Given that (a) the average number of approximations in a node is $n$, (b) the average number of candidate records to be checked is $m$, (c) the average number of branches to child nodes (fan-out) is $c$, (d) the probability of ceasing the sequential scan of a node due to a depth first search is $p$ and (e) the cost of using the heap $H$ is $\log(k)$ as we expect it to host $k$ elements, the computational complexity of the *k-NN* search is as follows: In each node, we visit $n(1-p)$ approximations and $m(1-p)$ candidate records. The cost of placing all approximations and records in the heap is $(n+m)\log(k)$; this is the cost of visiting a single node. To find the average number of nodes, we first compute the probability of a record pointing to a child node. This probability is $c/n$ as all approximations are $n$ and the number of pointers to child nodes is $c$. In each *DiVA* node, we examine $m(1-p)$ candidate records from which $(c/n)m(1-p)$ are, on average, pointers to child nodes. Therefore, the overall complexity is:

$$O\left(c\frac{m(1-p)}{n}(m+n)(1-p)\log(k)\right) \tag{8}$$

The above complexity analysis shows how we can improve the effectiveness of *DiVA*'s search capabilities. First, we should try to increase probability $P$ so as to perform depth first searches on hot areas. Second, keep $c$ low by adding additional nodes only for space areas we visit frequently.

### 4.2. Restructuring operations for *DiVA*

*DiVA* provides two low-level operations that help transform its structure and so assist in a more efficient processing of queries:

(1) *adding a new node*: new nodes enhance the indexing granularity of hot space areas. Space areas correspond to approximations stored in *DiVA*'s *a-files*. These approximations of hot areas must be specified in order to increase the index granularity of the respective hot area. Furthermore, the approximation must point to a record (inside the *r-file*) that is a list of data vectors. This list will be erased and all its data vectors will be placed in the new node. The record that was pointing to the list of data vectors will be updated to point to the newly created node. Building the new node requires setting the granularity of the index. This translates to the quantization step (bits per dimension) that will be used to build the new approximations. Both the approximation—corresponding to the hot-spot—and the quantization step of the new node have to be specified by the application-designated observer (discussed in the next section).
(2) *scan first hot-spot areas*: to do so, the approximations representing hot-spot cells must be moved toward the beginning of the *a-file*. The performance gain from this reordering stems from the fact that *DiVA* carries out a depth first search when it encounters the cell where the query vector belongs to (line 7 Algorithm 3). Placing hot cell approximations near the beginning of the *a-file* causes these hot-spots to be searched first, resulting in the quick location of the results or a rapid decrease of the search radius in *k-NN* queries. With this operation, we improve the probability $P$ in Equation (8). In Fig. 4, the length of the dashed-arrow corresponds to the number of approximations scanned. When hot areas are scanned earlier, *DiVA* may even skip searching the remainder of the *a-file* (line 11 Algorithm 3).

Both of the above operations result into structural changes. In the case of a new node addition, we need to compute the approximations of the data points in the new node. When optimizing for 'hot' areas, we need to update the data layout. The performance penalty for such operations may be significant and will affect not only the index but the operation of the system serving the end-user applications as well. The structure optimizations should be triggered when the application is idle or willing to pay the penalty incurred. Specifying the frequency of triggering such restructuring operations is out of the scope

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

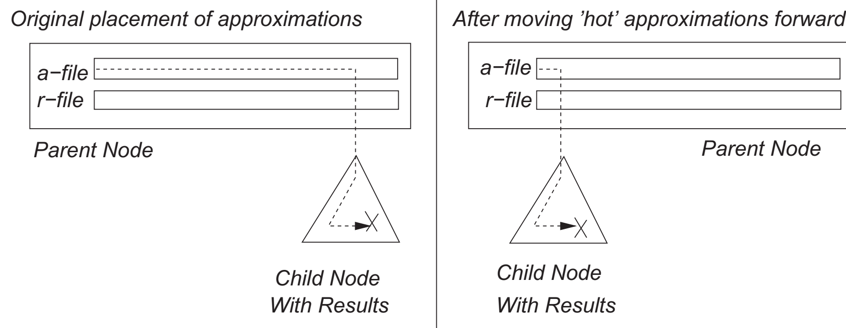


**FIGURE 4.** Moving the cell approximation of the query vector closer to the beginning of the *a-file* helps avoid long sequences of I/Os (as shown in the original placement).

of this manuscript, as this tightly relates to the needs of each specific application.

## 5. *DIVA* APPLICATION-PROVIDED REFINEMENT POLICIES

In this section, we present two application-provided *policies* that seek to optimize diverse aspects in *DiVA*'s operation: the first policy reduces I/O overheads and offers wall-time results that are competitive if compared with those obtained from conventional indexing approaches [15, 22, 26, 28]. The second policy favors queries of specific cohorts of users by appropriately exploiting respective *user-IDs*. Policies can be developed around the optimization of diverse aspects of the underlying storage management/medium.

### 5.1. Policy 1: minimize turnaround time

In its operation, minimize turnaround time (*MTT*) takes into account query workloads, data distribution, as well as data access and processing times; the latter are exclusively computer-system-specific. The *observer* that implements *MTT* produces the hot-spots to be further indexed as well as the index granularity each of the new nodes should display.

**Identifying hot-spots:** A hot-spot corresponds to a list of records whose further indexing may reap performance benefits. Frequently accessed lists are promising such candidates and can be selected based on the query workloads they serve. Data distribution and access-delays are used to rank lists according to their indexing prospect and in doing so, *MTT* evaluates a score for each such list of records; among those lists, the one giving the highest score is selected for expansion. This *Score* is projected as the difference between future (*Future*) and current (*Current*) processing costs and/or overheads incurred by operations taking place within the list of records in question:

$$Score = Current - Future \tag{9}$$

**TABLE 1.** Statistics maintained by the *MTT observer*.

| | |
|---|---|
| $R$ | Cost of accessing and processing a data record |
| $s$ | Cost of accessing and processing an approximation |
| $l$ | Length of records list |
| $h$ | Result hits in current records list |
| $q_s$ | Number of queries accessing current records list |
| $o$ | Average initialization cost of node access |

Table 1 shows the statistics collected during normal *DiVA* operation through the respective emitted events. These statistics are used in the evaluation of *Score* (9). The *observer* measures the average time $R$ required to access and process a single record. Similarly, we obtain the average time $s$ expended in dealing with a single approximation as well as the initialization cost $o$ required for opening files hosting *DiVA* nodes. $R$, $s$ and $o$ capture the I/O overhead and processing times required by underlying computing system *DiVA* operates on. Three more 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). In fact, we maintain very few statistics as we are only concerned about lists of records that are actually involved in serving query workload(s). Out of these lists, we do not consider at all those having a single data vector as any attempt for further indexing would incur an overhead higher than reading the vector itself. Thus, sparse data generate practically very few statistics whereas clustered data require statistics only for the cells enclosing a cluster. In what follows, we consider only a single records list and so the statistics correspond to the record list at hand.

Using the statistics of Table 1, the current list-scan cost for all queries $q_s$ is:

$$Current = q_s \times R \times l \tag{10}$$

The estimation of the total future cost is:

$$Future = q_s \times (o + Approx + ProjReads) \tag{11}$$



Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

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 *ProjReads* is the expected delay in reading the records of the new node.

The time required for scanning the approximations *Approx* is proportional to the number of new approximations as all of them are typically examined during an *a-file* scan. Under the assumption that in the new node each data vector will have a corresponding approximation, we can estimate the delays entailed in accessing and manipulating them (Equation (12)).

$$Approx = s \times l \tag{12}$$

*ProjReads* is an estimate of the per query delay 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 (13). 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 subspace 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. This is depicted in a 2D example in Fig. 5.

Considering the above, the per query expected time cost for reading records is:

$$ProjReads = R \times (h + m)/q_s \tag{13}$$

Providing an exact estimate of $m$ would require us to access all non-matching records within the subspace under examination. This is a costly operation as it would have to be repeated for every list of records whose score needs to be computed. To ameliorate this cost, our *MTT*-policy assumes that the data vectors located within the subspace we examine are uniformly distributed. Under this assumption, we are able to swiftly estimate $m$ for all candidate record lists. The uniform data assumption is lifted in a later stage where we have identified the list with the highest score and we need to determine the bits per dimension used in the creation of the vector approximations. However, for now and under the uniformity assumption, we estimate the density $D$ of data vectors within each cell as:

$$D = \frac{l}{2^{vaBits}} \tag{14}$$

where *vaBits* is the bit length of each approximation in the new node.

If $B$ is the number of cells that intersect with the surface of the $n$-cube enclosing the results, we expect on average only half the volume of these $B$ cells to reside within the $n$-cube. Consequently, $(B \times D)/2$ vectors residing in the aforementioned $B$ cells will not match the query. So, the following equation becomes:

$$ProjReads = R \times (h/q_s + B \times D/2) \tag{15}$$

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:

$$V = e^n = h/(q_s \times D) \Rightarrow e = \sqrt[n]{h/(q_s \times D)} \tag{16}$$

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

$$B = 2ne^{n-1} = 2n\left(\frac{h}{q_s \times D}\right)^{(n-1)/n} \tag{17}$$

With the help of Equations (10), (11), (12), (15), (17), the *Score* of Equation (9) is computed for each vector list; the one with the overall highest positive score is selected for further indexing; negative scores trigger no further action.

*Setting the quantization step:* As soon as the highest scoring record list is identified, *MTT* determines the per dimension quantization step. The number of bits to be used for the approximations of the new node is 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 [22]. More bits are assigned to dimensions over which data have greater variance so as to maximize the efficiency of the quantized approximations. Algorithm 5 outlines this heuristic. In each step, we assign one bit to the dimension with the highest deviation and before assigning the next bit, this deviation is halved. We derive $\sigma$ by scanning through all data vectors in the selected record list. At this stage, we are only interested in the cost of accessing all elements of one record list.

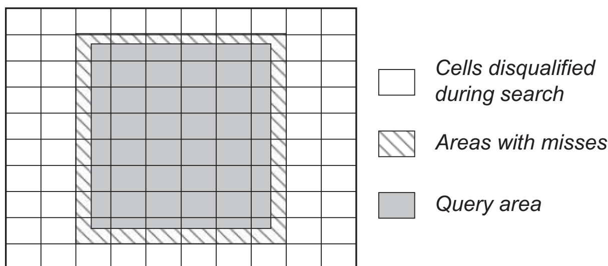


**FIGURE 5.** A 2D area inside which a new node introduces partitions. Results of a query are enclosed inside the inner dark square.

### 5.2. Policy 2: FGU

With *FGU*, we show how the separation between the structure's reorganization policy from the rest of *DiVA*'s operations



Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

**Algorithm 5** Deciding the per dimension bits allocation

**Input:** *dims*: Number of dimensions,
$B$: Total number of bits,
$\sigma_i$: Standard deviation of data in dimension i
**Output:** $b_i$: No. of bits assigned to dimension $i$

1: **for** $i \in dims$ **do**
2: $b_i \leftarrow 0$
3: **end for**
4: **for** $k \in B$ **do**
5: $z \leftarrow \text{arg_max}(\sigma_i)$
6: $b_z \leftarrow b_z + 1; \sigma_z \leftarrow \sigma_z/2$
7: **end for**

can serve application-specific needs. Often applications need to adjust their performance based on criteria that cannot be predicted in the context of an indexing method. A typical such example is applications that differentiate the quality of service delivered based on the privileges specific classes of users have.

An *observer* implementing *FGU* reduces the I/O overhead encountered in queries of specific user groups by exploiting the 'scan first' low-level operation of *DiVA*. With the 'scan first', hot cells created by queries of those specific groups are moved toward the beginning of the search operations (Fig. 4). As *DiVA* carries out a depth first node traversal when it comes across the cell the query vector belongs to (Algorithm 3), it would be beneficial for that cell to be found in the early stages of scanning the *a-file* of the root. In effect, the root serves as a space partitioning directory through which queries are dispatched to proper subspaces. *FGU* rearranges the cells of the *DiVA*'s root through suitable 'scan first' operations.

To realize *FGU* operation, we need to store the access frequency ($f$) of each record (*recno*) of the root node that triggers a depth first traversal. The access frequency is stored in a per user-group (*groupID*) fashion. During normal operation, the *FGU-observer* maintains a list of key-value pairs of the following form:

$$\langle key,\ value\rangle = \langle (groupID,\ recno), f\rangle$$

The *FGU-observer* exploits the *session ID* of Algorithms 2 and 3 to differentiate *DiVA*'s record access statistics among separate user groups. This is achieved by using the *groupID* as *session ID*.

As soon as the application decides on refining *DiVA* (e.g. when load is low), all records of *DiVA*'s root node are ranked. The rank of a record $i$ is computed by the following summation:

$$\text{Rank}_i = \sum_{g \in groupID} (w_g * f_{\{g,i\}})$$

The weight $w_g$ assigned to the group $g$ is application specific and is used to favor important user-groups over others. Based on the above ranking, the *FGU*-based observer designates a favorable sequence of records in *DiVA*'s root node. This sequence is applied through a series of 'scan first' operations that the policy instructs *DiVA* to follow.

## 6. EXPERIMENTAL EVALUATION

We first examine *DiVA* in isolation so as to determine the impact of the encoding used in the approximations as well as *DiVA*'s effectiveness when a node is added in the structure. We then compare *DiVA* against other multidimensional indexing methods. In order to fairly compare such methods with *DiVA*, we deploy the *MTT*-observer that attempts to reduce the overall I/O volume fetched, as is the case with most indexing methods. We use both synthetic and real datasets while experimenting with our prototype; real datasets consist of feature vectors from an image database [52]. Using the real datasets, we show how the second *FGU*-observer enhances query performance of specific, user targeted, space areas.

When *k-NN* queries are involved, we evaluate *DiVA* against Sequential Scan, the *VA*-file [15], the *A*-tree [26], *GC*-tree [28] and an index we call *VAp*file. *VAp*file is our implementation of the most important features suggested by the $VA^+$file [22], namely the 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. With the exception of the *A*-tree, we have implemented in *C++* all indexing methods used in our experimentation. The *A*-tree is implemented in *C* and its source code has been made publicly available [53].

In all our evaluation scenarios, data points are vectors of 4-byte unsigned integers with an extent up to $2^{32}$. Unless otherwise stated, the *VA*-file and *VAp* file are set to create files that are 12.5% the size of the original data, a value commonly used in *VA*-files [15]. Four bits per dimension are also used for the root node of *DiVA*. The *A*-tree uses the default value of 8 KB per data page.

To eliminate caching effects and accurately measure the impact of I/O, we flush caches prior to posting each query. Cached data affect the adaptation policy of *DiVA* in the following way: in low-dimensional spaces (less than 32) unpacking approximations requires more CPU processing than data vectors manipulation. So if both data and approximations reside in memory, *DiVA* is more conservative in producing new nodes. On the other hand, having data residing on disk favors node creation. In this case, the I/O cost for accessing data vectors by far exceeds that of accessing approximations, therefore *DiVA* introduces more approximations in an attempt to trade CPU processing time for reduced I/O.

We contacted all experiments on a 3 *GHz Pentium 4* machine with 2 *Giga Bytes* of main memory running *Linux v.2.6.27*.

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

### 6.1. Evaluating *DiVA* in isolation

**'Quick Within Range' predicate:** As we discussed in Section 3.2, *DiVA* greatly reduces CPU overhead by quickly identifying that a vector approximation is not within a range through a few binary operations. In Fig. 6, we measured the impact of the 'quick within range' predicate on the CPU processing time. During this evaluation, we perform a range query in a 32D uniformly populated space and vary the amount of indexed vectors. As shown, without the 'quick within range' predicate the required CPU time for the evaluation of a range query is increased by a factor of 6.

**New node addition:** *DiVA* exploits the existence of hot areas. By selectively adding new nodes, it refines its operational granularity and 'dives' into hot regions in the search space. To show how *DiVA*'s structure is progressively refined with additional nodes as suggested by the *MTT*-observer, we force *DiVA* to create more nodes than it would normally do by restricting the quantization step. In particular, *DiVA* is requested not to exceed the limit of 32 bits for approximations stored in child nodes, while the root node approximations use 8 bits per dimension. Thus, in the used 32D space, the root node approximations are 32 bytes long. This space is populated by a single cluster; it is on this cluster that we place all queries. The clustered data follow a Gaussian distribution, so by changing its standard deviation we show how *DiVA* reacts to different levels of data density. Data with higher variance require more nodes to be effectively indexed.

Figure 7 depicts how *DiVA*'s behavior is affected by the addition of new nodes; the graphs show that both CPU processing time and I/O volume are dramatically reduced as more nodes are added. We attribute this to the enhanced indexing provided by the additional nodes that vastly lower the number of data vectors read and processed. For both CPU (Fig. 7a) and I/O (Fig. 7b), less than three nodes result into no performance gain, since after each node creation the entire cluster is still stored in a single record list of the leaf node. The number of nodes needed to reduce the I/O and CPU cost of the query is relative to the data density, as shown by the distributions with the three standard deviations (500, 1000 and 2000). After creating a certain number of nodes, no further performance gain is expected and thus, *DiVA* refrains from creating more nodes.

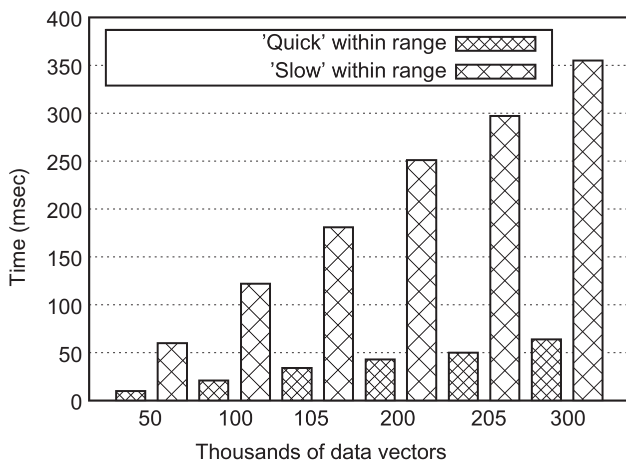


**FIGURE 6.** Performance gain provided by the 'quick within range' operation used in *DiVA*.

### 6.2. Synthetic dataset

Here, we create a synthetic dataset and measure the I/O performance of *DiVA* and other indexing methods while varying the following properties: (a) dimensionality, (b) volume of indexed data and c) percentage of clustered data vectors. We employ the *MTT* policy to drive *DiVA*'s expansion.

The space we use as the *base case* consists of 200 000 data vectors featuring 32D. 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 $\sigma$ 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 $k = 100$.

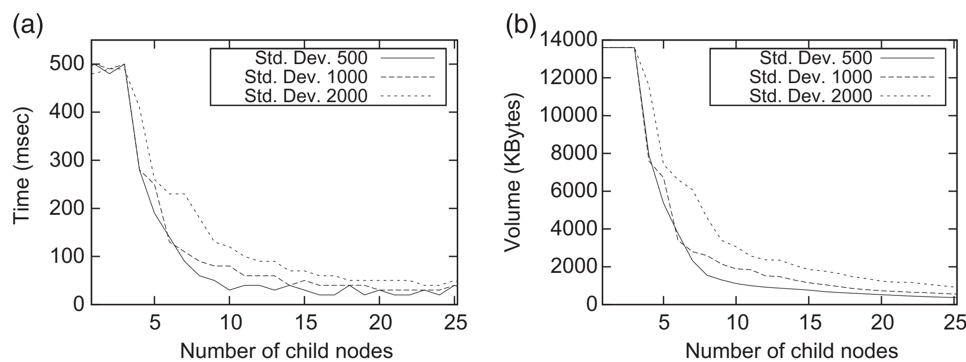


**FIGURE 7.** Performance for clustered data when increasing the number of nodes in *DiVA*. (**a**) Processing time and (**b**) I/O.



Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

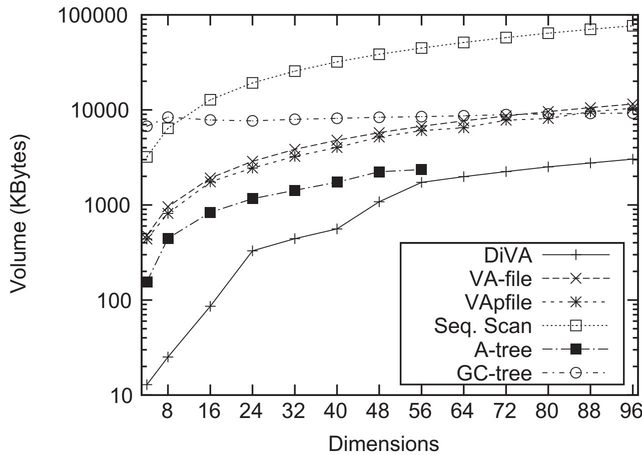


**FIGURE 8.** Dimensionality impact on indexing methods.

*Dimensionality:* We vary the data dimensionality from 4 to 96 and measure how *DiVA* fares in terms of I/O performance (Fig. 8). 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 Fig. 8. For *GC*-tree the trend is harder to identify. Query evaluation in the *GC*-tree includes a phase of scanning all data approximations that yields candidate data vectors. The approximations have a fixed size regardless the number of dimensions. In this experiment, this constant cost approximations scanning phase dominates the *GC*-tree I/O. *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 [53], we were able to evaluate its performance only up to 56D. As shown, *DiVA* exhibits a clear advantage in the entire range of dimensions tested.

*Volume of vectors:* More data vectors force *DiVA* to examine more approximations on one hand; on the other, tree-based indexes try to partition space so that less data are examined. Therefore, at least in low-dimensional spaces, they are less influenced by an increase in data volume. *DiVA* does combine features from both sequential search and tree-based access methods. In 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 9a shows the I/O load for each index in this evaluation scenario. Thanks 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 with the rest of the tested indexes.

*Clustered vectors: VA*-file is known to perform well when there are no clustered data [15]. *DiVA* on the other hand, employs its hierarchical structure to efficiently index both uniform and clustered data. Our approach is able to 'dive in' to areas of interest and effectively respond to queries targeting cells with increased index granularity. Here, we vary the percentage of the clustered data while keeping the total number of indexed vectors fixed to 200 000.

Figure 9b 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. By having *VA*-file like nodes, *DiVA* performs similarly to the *VA*-file in areas with uniformly distributed data where no further indexing is desired. Under uniformly distributed data, the *A*-tree performs better than the *VA*-file but worse than *DiVA*. The *A*-tree produces 2.18 times more I/O than *DiVA* when only 15% of data vectors are clustered. As more data are moved to the clusters, *DiVA* extends its performance lead. In a setting of 90% clustered vectors, the *A*-tree reads 3.3 times more bytes than *DiVA*.

### 6.3. Image feature vectors

The real dataset used during our evaluation consists of 200 000 feature vectors extracted from images using methods similar to [54]. The dimensionality of this dataset 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 four 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 Fig. 10a, there is still room for improvement using *DiVA*. The *VA*-file is set to use 4 bits per dimensions and thus produce approximations 12.5% the size of the data vectors (that use 32 bit per dimensions). A *VA*-file of around 10% the size of the dataset is shown to be most efficient. For *DiVA*, we assign only 2 bits per dimension for the approximations of the root node and then we allow the *MTT* policy to further refine the index operation by reserving more bits for each new node. *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 fetched records that contain feature vectors (Fig. 10b). 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 41% for *DiVA*.

### 6.4. Favoring specific users

The real dataset is a sparsely populated space and the distance among data vectors increases even further as dimensionality increases. This is in favor of the *VA*-file and the *GC*-tree. In the

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

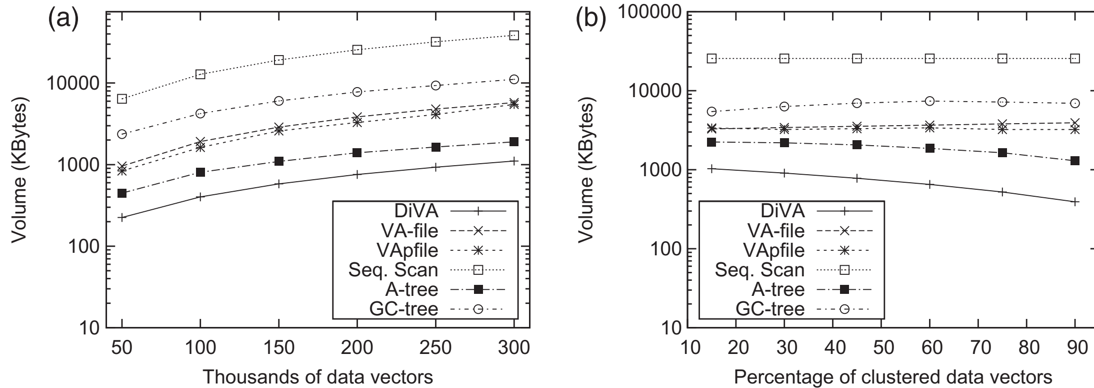


**FIGURE 9.** Performance of indexing methods for synthetic datasets. (**a**) Processing time and (**b**) I/O.

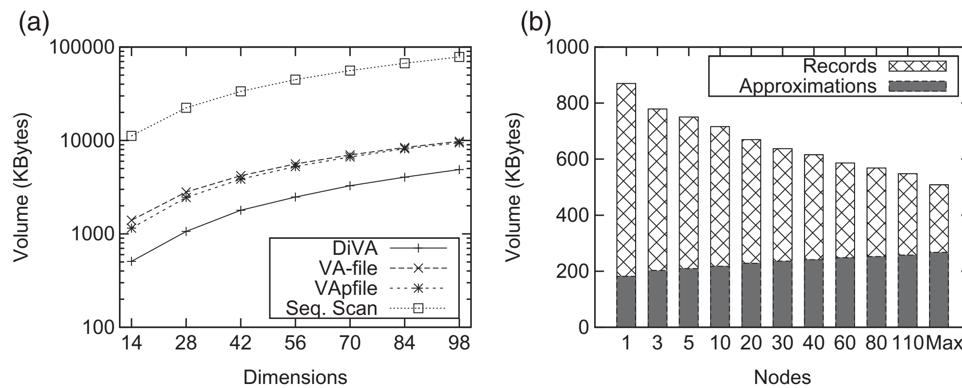


**FIGURE 10.** Range query evaluation of *DiVA* using real dataset. (**a**) I/O *vs.* dimensions and (**b**) I/O *vs.* dataset size.

*VA*-file, each data vector features a unique approximation while the *GC*-tree uses approximations of fixed size regardless the space dimension. In sparse spaces such as the one we have for the real dataset, the bounding area specified by *GC*-tree's LPC approximations are enough to effectively locate a data vector. Therefore, the I/O of the *GC*-tree remains unaffected by space dimensionality as it is dominated by the scan of the fixed-sized LPC approximations.

Using the *FGU* observer, we are able to further improve the performance of *DiVA* when accessing specific hot areas. Unlike the *VA*-file and the *GC*-tree, *DiVA* may skip scanning approximations in case the requested nearest-neighbors have already been encountered (Algorithm 3). For this to happen, all vectors returned by a user query have to be contained in a single cell. This can be easily attained either by using larger cells—especially in the root node—or by limiting the number of nearest neighbors a user can retrieve form a query. Here, we elect to present the outcome of our evaluation when using *k-NN* queries with $k = 1$ (i.e. more focused). This type of queries are most effectively handled by the *GC*-tree especially as dimensionality increases and space becomes sparsely populated. We also keep the root node set in a way favorable to the *VA*-file (i.e. size of approximations: 4 bits per dimension).

For this experiment, we assume that all user groups are of equal importance thus all groups are assigned a weight ($w_g$) set to 1. Furthermore, user groups are interested in separate space areas selected randomly at the beginning of our evaluation. We use the *FGU*-observer to have *DiVA* monitor the query areas of apparent interest to user(s) and then we restructure *DiVA* so as to optimize its performance for these specific regions. Figure 11a shows the I/O performance of the *A*-tree, the *VAp*file, the *VA*-file, the *GC*-tree and *DiVA* as the number of dimensions increases. *DiVA* configuration presented in Fig. 11b display different levels of refinement. '*DiVA* Orig'. marks the performance we get when no refinement 'scan first' operations are applied at all, while '*DiVA* 100%' designates the performance rates we obtain after refining the index using the *FGU* policy for all query regions. The '*DiVA* Orig'. configuration is able to outperform the *VA*-file, since it has the advantage of scanning only a portion of the approximations. The performance of '*DiVA* Orig'. is matched by the *GC*-tree for high dimensions in the range we examine (over 85 dimensions).

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

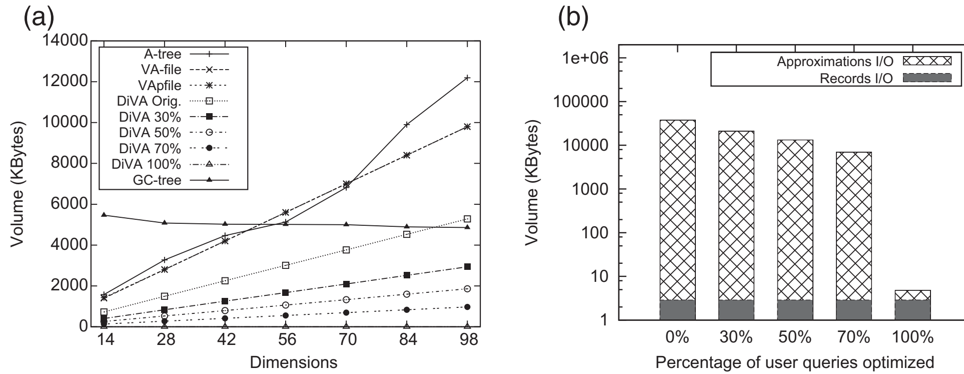


**FIGURE 11.** High-level refinement policy. Favoring specific user groups. (**a**) I/O *vs.* % of refined user queries and (**b**) approximation and record I/O.

In the '*DiVA* 100%' configuration, a query evaluation involves reading only the first few approximations of the root node's *a-file*. One of these approximations belongs to the vector we are querying for. Figure 11b shows the I/O overhead involved in reading approximations and records under different refinement levels. Here, we show the collective overhead involved in 10 different random *k-NN* queries on a 70D real feature vector dataset. As the refinement level increases, the overhead involved in accessing approximations decreases rapidly while the record reads stay the same. This happens as we gradually place hot approximations toward the beginning of the root's *a-file* while having almost on average one approximation per record.

## 7. CONCLUSIONS AND FUTURE WORK

In this paper, we propose *DiVA* that not only performs efficient range and *k-NN* queries in high-dimensional clustered data spaces but can also effectively deal with workloads of fine-granularity queries that present spatial locality. Although *DiVA* maintains a hierarchical structure, its node organization heavily borrows from *VA*-file. Every *DiVA* node uses approximated data to access data vectors within an *n*-dimensional subspace, while the hierarchical structure provides increasing levels of index detail. In this manner, *DiVA* offers fast navigation ('diving') to query regions of interest and the structure of its nodes helps efficiently handle high-dimensional vectors.

External-to-index software components, termed observers, enforce application-specific policies when it comes to guiding the expansion and restructure of *DiVA*. Such application-originated policies that help drive the reorganization of *DiVA* may take into account not only data and query distributions but also characteristics of the underlying computing system. We discuss two such policies that we have incorporated into our detailed prototype. Experimentation with both real and synthetic datasets shows that *DiVA* produces significant improvements compared with competing methods such as the *VA*-file, *VA*+-file, the *GC*-tree and the *A*-tree.

Our future work plans include: (a) the support for asynchronous messaging between *DiVA* and the observer(s), (b) the compaction of the records' lists stored in the *r-file* so as to reduce the disk seek operations and (c) the use of advanced statistical models in determining the optimal index granularity of hot space areas.

## ACKNOWLEDGEMENTS

We wish to thank the reviewers of this article for their many insightful and constructive comments.

## FUNDING

This work has been partially supported by the *D4Science II* and *iMarine* EU FP7 projects.

## REFERENCES

[1] Szalay, A.S., Gray, J., Thakar, A., Kunszt, P.Z., Malik, T., Raddick, J., Stoughton, C. and van den Berg, J. (2002) The SDSS SkyServer—Public Access to the Sloan Digital Sky Server Data. *Proc. ACM SIGMOD*, Madison, WI, June.

[2] Batko, M., Falchi, F., Lucchese, C., Novak, D., Perego, R., Rabitti, F., Sedmidubsky, J. and Zezula, P. (2010) Building a web-scale image similarity search system. *Multimedia Tools Appl.*, **47**, 599–629.

[3] Schaefer, G. (2010) A next generation browsing environment for large image repositories. *Multimedia Tools Appl.*, **47**, 105–120.

[4] Akgül, C., Rubin, D., Napel, S., Beaulieu, C., Greenspan, H. and Acar, B. (2011) Content-based image retrieval in radiology: current status and future directions. *J. Digit. Imaging*, **24**, 208–222.

[5] Valle, E., Cord, M. and Philipp-Foliguet, S. (2008) High-Dimensional Descriptor Indexing for Large Multimedia Databases. *Proc. ACM Conf. Information and Knowledge Management (CIKM)*, Napa Valley, CA, pp. 739–748.

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

[6] Huang, Z., Shen, H., Liu, J. and Zhou, X. (2011) Effective Data Co-Reduction for Multimedia Similarity Search. *Proc. ACM SIGMOD Int. Conf. Management of Data*, Athens, Greece, June, pp. 1021–1032.

[7] Zhou, X., Zhou, X., Chen, L. and Bouguettaya, A. (2012) Efficient subsequence matching over large video databases. *VLDB J.*, **21**, 489–508.

[8] Katayama, N. and Satoh, S. (1997) The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. *Proc. ACM SIGMOD*, Tucson, AZ, USA, May. ACM Press.

[9] Sakurai, Y., Yoshikawa, M., Uemura, S. and Kojima, H. (2002) Spatial indexing of high-dimensional data based on relative approximation. *VLDB J.*, **11**, 93–108.

[10] Tao, Y., Yi, K., Sheng, C. and Kalnis, P. (2010) Efficient and accurate nearest neighbor and closest pair search in high dimensional space. *ACM Trans. Database Syst.*, **35**, 20:1–20:46.

[11] Yao, B., Li, F. and Kumar, P. (2010) K Nearest Neighbor Queries and KNN-Joins in Large Relational Databases (Almost) for Free. *Proc. IEEE Int. Conf. Data Engineering (ICDE)*, Long Beach, CA, March, pp. 4–15.

[12] Malik, R., Kim, S., Jin, X., Ramachandran, C., Han, J., Gupta, I. and Nahrstedt, K. (2009) MLR-Index: An Index Structure for Fast and Scalable Similarity Search in High Dimensions. *Proc. Int. Conf. Scientific and Statistical Database Management (SSDBM)*, New Orleans, LA, June, pp. 167–184.

[13] Zhang, Z., Ooi, B.C., Parthasarathy, S. and Tung, A.K.H. (2009) Similarity search on Bregman divergence: towards non-metric indexing. *Proc. VLDB Endow.*, **2**, 13–24.

[14] Zhang, D., Agrawal, D., Chen, G. and Tung, A. (2011) HashFile: An Efficient Index Structure for Multimedia Data. *Proc. of IEEE Int. Conf., on Data Engineering (ICDE)*, Hannover, Germany, April, pp. 1103–1114.

[15] Weber, R., Schek, H.-J. and Blott, S. (1998) A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *Proc. Int. Conf. Very Large Data Bases (VLDB)*, New York, NY, August, pp. 194–205.

[16] Beyer, K.S., Goldstein, J., Ramakrishnan, R. and Shaft, U. (1999) When Is 'Nearest Neighbor' Meaningful? *Proc. Int. Conf. Database Theory (ICDT)*, Jerusalem, Israel, pp. 217–235. Springer.

[17] Indyk, P. and Motwani, R. (1998) Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. *13th Annual ACM Symp. Theory of Computing (STOC)*, Dallas, TX, May, pp. 604–613.

[18] Ferhatosmanoglu, H., Tuncel, E., Agrawal, D., El Abbadi, A. (2001) Approximate Nearest Neighbor Searching in Multimedia Databases. *Proc. IEEE Int. Conf. Data Engineering (ICDE)*, Heidelberg, Germany, April, pp. 503–511.

[19] Shen, H.T., Zhou, X. and Zhou, A. (2007) An adaptive and dynamic dimensionality reduction method for high-dimensional indexing. *VLDB J.*, **16**, 219–234.

[20] Bernecker, T., Emrich, T., Graf, F., Kriegel, H.-P., Renz, M., Schubert, E. and Zimek, A. (2010) Subspace Similarity Search: Efficient k-NN Queries in Arbitrary Subspaces. *Proc. Int. Conf. Scientific and Statistical Database Management (SSDBM)*, Heidelberg, Germany, June, pp. 555–564.

[21] Kim, Y., Chung, C.-W., Lee, S.-K. and Kim, D.-H. (2011) Distance approximation techniques to reduce the dimensionality for multimedia databases. *Knowl. Inf. Syst.*, **28**, 227–248.

[22] Ferhatosmanoglu, H., Tuncel, E., Agrawal, D. and Abbadi, A.E. (2000) Vector Approximation-based Indexing for Non-uniform High Dimensional Data Sets. *Proc. 9th CIKM Conf.*, McLean, VA, November. ACM.

[23] Ferhatosmanoglu, H., Tuncel, E., Agrawal, D. and El Abbadi, A. (2006) High dimensional nearest neighbor searching. *Inf. Syst.*, **31**, 512–540.

[24] Günnemann, S., Kremer, H., Lenhard, D. and Seidl, T. (2011) Subspace Clustering for Indexing High Dimensional Data: A Main Memory Index Based on Local Reductions and Individual Multi-Representations. *Int. Conf. Extending Database Technology (EDBT)*, Uppsala, Sweden, March, pp. 237–248.

[25] Barclay, T., Slutz, D. and Gray, J. (2000) TerraServer: A Spatial Data Warehouse. *Proc. of ACM SIGMOD Int. Conf., on Management of Data*, Dallas, TX, May, pp. 307–318.

[26] Sakurai, Y., Yoshikawa, M., Uemura, S. and Kojima, H. (2000) The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. *Proc. of 26th Int. Conf., on Very Large Databases (VLDB)*, Cairo, Egypt, September, pp. 516–526.

[27] Berchtold, S., Bohm, C., Jagadish, H., Kriegel, H.-P. and Sander, J. (2000) Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces. *Proc. 16th IEEE ICDE*, San Diego, CA, February.

[28] Cha, G.-H. and Chung, C.-W. (2002) The GC-tree: a high-dimensional index structure for similarity search in image databases. *IEEE Trans. Multimedia*, **4**, 235–247.

[29] Gaede, V. and Günther, O. (1998) Multidimensional access methods. *ACM Comput. Surv.*, **30**, 170–231.

[30] Samet, H. (2005) *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[31] Berchtold, S., Keim, D. and Kriegel, H.-P. (1996) The X-tree: An Index Structure for High-Dimensional Data. *Porc. of 22nd Int., Conf., on Very Large Databases (VLDB)*, Mumbai, India, September, pp. 28–39.

[32] White, D. and Jain, R. (1996) Similarity Indexing with the SS-tree. *Proc. IEEE ICDE Int. Conf.*, New Orleans, LA, February. IEEE Comp. Society.

[33] Ciaccia, P., Patella, M. and Zezula, P. (1997) M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *Proc. of 23rd Int., Conf., on Very Large Databases (VLDB)*, Athens, Greece, August, pp. 426–435.

[34] Jagadish, H.V., Ooi, B.C., Tan, K.-L., Yu, C. and Zhang, R. (2005) *iDistance*: an adaptive $B^+$-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, **30**, 364– 397.

[35] Traina Jr, C., Filho, R.F.S., Traina, A.J.M., Vieira, M.R. and Faloutsos, C. (2007) The Omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. *VLDB J.*, **16**, 483–505.

[36] Freeston, M. (1987) The BANG File: A New Kind of Grid File. *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May.

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015

[37] Robinson, J. (1981) The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. *Proc. SIGMOD*, Ann Arbor, MI, April, pp. 10–18.

[38] Chakrabarti, K. and Mehrotra, S. (1999) The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces. *Proc. of ACM SIGMOD Int. Conf., on Management of Data*, Sydney, Australia, March.

[39] Bellman, R. (1957) *Dynamic Programming*. Princeton University Press, Princeton, NJ.

[40] Blott, S. and Weber, R. (2008) What's wrong with high-dimensional similarity search? *Proc. VLDB Endow.*, **1**, 3.

[41] Tuncel, E., Ferhatosmanoglu, H. and Rose, K. (2002) VQ-index: An Index Structure for Similarity Searching in Multimedia Databases. *Proc. 10th ACM Int. Conf. Multimedia (MM)*, Juan-les-Pins, France, pp. 543–552.

[42] Weber, R. and Böhm, K. (2000) Trading Quality for Time with Nearest Neighbor Search. *Proc. of 7th Int. Conf., on Extending Database Technology (EDBT)*, Konstanz, Germany, March, pp. 21–35.

[43] Chen, L., Chang, E., Garcia-Molina, H. and Wiederhold, G. (2002) Clustering for approximate similarity search in high-dimensional spaces. *IEEE Trans. Knowl. Data Eng.*, **14**, 792–808.

[44] Houle, M. and Sakuma, J. (2005) Fast Approximate Similarity Search in Extremely High-Dimensional Data Sets. *Proc. of 21st IEEE Int. Conf., on Data Engineering (ICDE)*, Tokyo, Japan, April, pp. 619–630.

[45] Gionis, A., Indyk, P. and Motwani, R. (1999) Similarity Search in High Dimensions via Hashing. *Proc. of 25th Int. Conf., on Very Large Databases (VLDB)*, Edinburgh, Scotland, September, pp. 518–529.

[46] Bawa, M., Condie, T. and Ganesan, P. (2005) LSH Forest: Self-Tuning Indexes for Similarity Search. *Proc. 14th Int. Conf. World Wide Web (WWW)*, Chiba, Japan, April, pp. 651–660.

[47] Lv, Q., Josephson, W., Wang, Z., Charikar, M. and Li, K. (2007) Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. *Proc. 33rd VLDB Conf.*, Vienna, Austria, September.

[48] Keogh, E.J., Chakrabarti, K., Mehrotra, S. and Pazzani, M.J. (2001) Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *Proc. of ACM SIGMOD Int. Conf., on Data Management*, Santa Barbara, CA, USA, May, pp. 151–162.

[49] Athanassoulis, M. and Ailamaki, A. (2014) BF-tree: approximate tree index. *VLDB J.*, **7**, 1881–1892.

[50] Sun, Y., Wang, W., Qin, J., Zhang, Y. and Lin, X. (2014) SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *VLDB J.*, **8**, 1–12.

[51] Tsakalozos, K., Evangelatos, S. and Delis, A. (2011) DiVA: Indexing High-Dimensional Data by 'Diving' into Vector Approximations. *Proc. IEEE Int. Conf. Multimedia and Expo 2011*, Barcelona, Spain, July.

[52] Wang, J., Li, J. and Wiederhold, G. (2001) SIMPLIcity: semantics-sensitive integrated matching for picture libraries. *IEEE Trans. Pattern Anal. Mach. Intell.*, **23**, 947–963.

[53] Sakurai, Y. (2011) *The A-Tree: Source Code Release*. NTT Communication Science Laboratories, Seika, Soraku, Kyoto, Japan.

[54] Vu, K., Hua, K.A. and Tavanapong, W. (2003) Image retrieval based on regions of interest. *IEEE Trans. Knowl. Data Eng.*, **15**, 1045–1049.

Downloaded from http://comjnl.oxfordjournals.org/ at Columbia University Libraries on December 2, 2015