Evaluating Refined Queries in Top-*k* Retrieval Systems

Kaushik Chakrabarti, Michael Ortega-Binderberger, *Member*, *IEEE*, Sharad Mehrotra, *Member*, *IEEE*, and Kriengkrai Porkaew

Abstract—In many applications, users specify target values for certain attributes/features without requiring exact matches to these values in return. Instead, the result is typically a ranked list of "top k" objects that best match the specified feature values. User subjectivity is an important aspect of such queries, i.e., which objects are relevant to the user and which are not depends on the perception of the user. Due to the subjective nature of top-*k* queries, the answers returned by the system to an user query often do not satisfy the users need right away, either because the weights and the distance functions associated with the features do not accurately capture the users perception or because the specified target values do not fully capture her information need or both. In such cases, the user would like to refine the query and resubmit it in order to get back a better set of answers. While there has been a lot of research on query refinement models, there is no work that we are aware of on supporting refinement of top-*k* queries efficiently in a database system. Done naively, each "refined" query can be treated as a "starting" query and evaluated from scratch. This paper explores alternative approaches that significantly improve the cost of evaluating refined queries by exploiting the observation that the refined queries are not modified drastically from one iteration to another. Our experiments over a real-life multimedia data set show that the proposed techniques save more than 80 percent of the execution cost of refined queries over the naive approach and is more than an order of magnitude faster than a simple sequential scan.

Index Terms—Multidimensional indexing, *k*-nearest neighbor search, similarity queries, query refinement, relevance feedback.

1 Introduction

Top- k selection queries are becoming common in many modern-day database applications. Unlike in a traditional relational database system (RDBMS) where a selection query consists of a precise selection condition and the user expects to get back the exact set of objects that satisfy the condition, in top-*k* queries, the user specifies target values for certain attributes and does not expect exact matches to these values in return. Instead, the result to such queries is typically a *ranked list* of the "top k" objects that best match the given attribute values. As the following examples illustrate, top-*k* queries arise naturally in a variety of today's applications.

Example 1 (Multimedia Databases). Consider a content-based image retrieval system [15], [27], [17]. Each image is represented using features like color, texture, layout, and shape [16]. The similarity between any two objects is computed by first computing their similarities based on the individual features and then combining them to obtain the overall similarity. Typically, the user submits an example as the query object and requests for a few objects that are "most similar" to the submitted example (Query By Example (QBE)). The DBMS ranks the images according to how well they match the query image and returns the best few matches to the user in a ranked fashion, the most similar images first followed by the less similar ones.

Example 2 (E-Commerce). Consider a real-estate database that maintains information like the location of each house, the price, the number of bedrooms, etc. [5] (e.g., MSN HomeAdvisor). Suppose that a potential customer is interested in houses in the Irvine, CA, area, with four bedrooms and with a price tag of around $300,000. Again, the DBMS should rank the available houses according to how well they match the given user preference and return the top houses for the user to inspect. If no houses match the query specification exactly, the system might return houses in Santa Ana (a city near Irvine) or two bedroom houses in Irvine or more expensive houses as the top matches to the query.

An important aspect of top-*k* queries is user subjectivity [13], [7]. To return "good quality" answers, the system must understand the user's *perception* of similarity, i.e., the relative importance of the attributes/features[1] to the user. The system models user perception via the distance functions (e.g., Euclidean in the above example) and the weights associated with the features [6], [4], [18], [5]. At the time of the query, the system *acquires* information from the user based on which it determines the weights and distance functions that best capture the perception of this particular user and instantiates the model with these values. Note that this instantiation is done at query time since the user perception differs from user to user and from query to query. Once the model is instantiated, we retrieve, based on the model, the

- *K. Chakrabarti is with Microsoft Research, One Microsoft Way, Redmond, WA 98052. kaushik@microsoft.com.*
- *M. Ortega-Bindernerger is with the IBM Silicon Valley Lab, 555 Bailey Ave., San Jose, CA 95141. E-mail: miki@acm.org.*
- *S. Mehrotra is with the School of Information and Computer Science, University of California at Irvine, 444 Information and Computer Science Bldg., Irvine, CA 92697. E-mail:sharad@ics.uci.edu.*
- *K. Porkaew is with the School of Information Technology, King Mongkut's University of Technology Thonburi, 91 Suksawasd 48, Bangmod, Thungkru, BKK 10140. E-mail: porkaew@it.kmutt.ac.th.*

Manuscript received 27 Dec. 2000; revised 20 May 2002; accepted 30 Sept. 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 113372.

top answers by first executing a k nearest neighbor (k-NN) algorithm on each individual feature[2] and then *merging* them to get the overall answers [17], [14], [6], [7].

Due to the subjective nature of top-k queries, the answers returned by the system to a user query usually do not satisfy the user's need right away [18], [4], [23], [11]. This can happen due to several reasons: The starting examples may not be the best ones to capture the information need (IN) of the user or the starting weights may not accurately capture the users perception or both. In this case, the user would like to *refine* the query and resubmit it in order to get back a better set of answers. We refer to this process as *query refinement* and the new query is called the "refined" query. In a QBE environment (e.g., multimedia databases), the user typically refines the query by finding, among the answers returned to the "starting" query, one or more objects that are closest to what she wants and requesting for more objects like those [22], [23], [18], [4], [11]. Based on the user feedback, the system will compute the new query objects and the new weights and execute the refined query. Another way to refine the query is that the user explicitly modifies the perception model, i.e., she explicitly changes the weights of the features so as to better capture her perception of similarity [13], [7]. In either case, the user can continue refining the query over as many iterations she wants till she is satisfied with the results. Recent work shows that query refinement techniques significantly improve the quality of answers and answers improve with more iterations of feedback [23], [18], [11].

While there has been a lot of research on improving the effectiveness of query refinement as well as on evaluating top-k queries efficiently, there exists no work that we are aware of on *how to support refinement of top-k queries efficiently inside a DBMS*. We explore such approaches in this paper. A naive approach to supporting query refinement is to treat a refined query just like a starting query and execute it from scratch. We observe that the *refined queries are not modified drastically* from one iteration to another; hence, executing them from scratch every time is wasteful. Most of the execution cost of a refined query can be saved by appropriately *exploiting the information generated during the previous iterations of the query*. We show how to execute subsequent iterations of the query efficiently by utilizing the cached information. Note that, since the query changes, albeit slightly, from iteration to iteration (i.e., the query points and/or the weights/distance functions are modified), we, in general, cannot answer a refined query entirely from the cache, i.e., we still need to access some data from the disk. Our technique minimizes the amount of data that is accessed from a disk to answer a refined query. Furthermore, our technique does not need to examine the entire cached priority queue to answer a refined query, i.e., it explores only those items in the cache that are necessary to answer the query, thus saving unnecessary distance computations (CPU cost).[3] Our experiments on real-life multimedia data sets show that our techniques improve the execution cost of the refined queries by up to two orders of magnitude over the naive approach.

A secondary contribution of this paper is a technique to evaluate *multipoint queries* efficiently. In order to support refinement, we need to be able to handle multipoint queries as shown in the MARS and FALCON papers [18], [4], [28]. Such queries arise when the user submits multiple examples during feedback. We first formally define the multipoint query and then develop an efficient k-NN algorithm that computes the k nearest neighbors to such queries. Our experiments show that our algorithm is more efficient compared to the multiple expansion approach proposed in FALCON [28] and MARS [20].

The rest of the paper is organized as follows: In Section 2, we provide an overview on top-k selection queries and query refinement techniques. Section 3 presents the k-NN algorithm for multipoint queries. Section 4 describes the techniques to evaluate "refined" queries and is the main contribution of this paper. In Section 5, we present the performance results. Section 6 offers concluding remarks.

2 The Model

In a top-k retrieval system, the user poses a query Q by providing target values for each feature and specifying the number k of matches desired. The target values can be either explicitly specified by the user (as in Example 2) or are extracted from the example submitted by the user (as in Example 1). We refer to Q as the "starting" query. The starting query is then matched to the set of objects in the database and the top k matches are returned. If the user is not satisfied with the answers, she provides feedback to the system either by submitting interesting examples or via explicit weight modification. Based on the feedback, the system refines the query representation to better suit the user's information need. The "refined" query is then evaluated and the process continues for several iterations until the user is fully satisfied. When the user is satisfied with the answers returned, she can request for additional matches incrementally. The process of feedback and requesting additional matches can be interleaved arbitrarily. We now discuss how each object is represented in the database (the object model), how the user query is represented (the query model), how the retrieval takes place (the retrieval model), and how refinement takes place (the refinement model).

2.1 How are Objects Represented?

Objects are vectors in a multidimensional space. Let S be a d_S-dimensional space. We view an object O as a point in this multidimensional space, i.e., O is a d_S-dimensional vector. Many image retrieval systems represent image features in this way. How the objects O are obtained (i.e., the feature extraction) depends on the application (e.g., in Example 1, special image processing routines are used to extract the color and texture from the image; there, a 2D feature space S_C is associated with the color feature).

2.2 How are Queries Represented?

A query is represented as a set of objects from the multidimensional space S and a distance function $\mathcal{D}_Q$ that

2. In this paper, we assume that all the feature spaces are metric and an index (called the Feature-index or F-index) exists on each feature space. A F-index is either single dimensional (e.g., B-tree) or multidimensional (e.g., R-tree) depending on the feature space dimensionality.

3. Our techniques are independent of the way the user provides feedback to the system, i.e., it does not matter whether she uses the QBE (i.e., "give me more like this") interface or the explicit weight modification interface.

computes the distance between any object and the set of query objects. This distance function associates weights μ_Q with each dimension of the space S and a weight w_Q with each query point. The reason for including multiple points in the query is that during refinement, the user might submit multiple examples to the system as feedback (those that she considers relevant) leading to multiple points in the space (see Section 2.3). We refer to such queries as *multipoint queries*. The user may also specify the relative importance of the submitted examples, i.e., the importance of each example in capturing her information need (e.g., relevance levels in MARS [23], "goodness scores" in Mindreader [11]). To account for importance, we associate a weight with each point of the multipoint query. We now formally define a multipoint query.

Definition 1 (Multipoint Query). *A multipoint query $Q = \langle n_Q, \mathcal{P}_Q, \mathcal{W}_Q, \mathcal{D}_Q \rangle$ for the space S consists of the following information:*

- *The number n_Q of points in Q.*
- *A set of n_Q points $\mathcal{P}_Q = \{{P_Q}^{(1)}, \ldots, {P_Q}^{(n_Q)}\}$ in the d_S-dimensional feature space S.*
- *A set of n_Q weights $\mathcal{W}_Q = \{{w_Q}^{(1)}, \ldots, {w_Q}^{(n_Q)}\}$, the ith weight ${w_Q}^{(i)}$ being associated with the ith point ${P_Q}^{(i)}$ ($1 \geq {w_Q}^{(i)} \geq 0, \Sigma_{i=1}^{n_Q} {w_Q}^{(i)} = 1$).*
- *A distance function $\mathcal{D}_Q$ which, given a point O in the space S, returns the distance between the query and the point. To compute the overall distance, we use a point to point distance function D_Q which, given two points in S, returns the distance between them. We assume D_Q to be a weighted L_p metric, i.e., for a given value of p, the distance between two points T_1 and T_2 in S is given by:*[4]

$$D_Q(T_1, T_2) = [\Sigma_{j=1}^{d_S} {\mu_Q}^{(j)} (|T_1[j] - T_2[j]|)^p]^{1/p}, \quad (1)$$

where ${\mu_Q}^{(j)}$ denotes the weight associated with the ith dimension of S. ($1 \geq {\mu_Q}^{(j)} \geq 0, \Sigma_{j=1}^{d_S} {\mu_Q}^{(j)} = 1$). D_Q specifies which L_p metric to use (i.e., the value of p) and the values of the dimension weights. We use the point to point distance function D_Q to construct the aggregate distance function $\mathcal{D}_Q(Q, O)$ between the multiple query points ($\mathcal{P}_Q$) and the object O (in S) $\mathcal{D}_Q(Q, O)$ is the aggregate of the distances between O and the individual points ${P_Q}^{(i)} \in \mathcal{P}_Q$.

$$\mathcal{D}_Q(Q, O) = \sum_{i=1}^{n_Q} {w_Q}^{(i)} D_Q(\mathcal{P}_Q^{(i)}, O). \quad (2)$$

We use weighted sum as the aggregation function, but any other function can be used as long as it is weighted and monotonic [6].

The choice of $\mathcal{D}_Q$ (i.e., the choice of the L_p metric) and the intrafeature weights capture the user perception within the space as shown in Example 3. Section 2.3 discusses how to choose the weights/metric. Table 1 provides a summary about notation.

Example 3 (Dimension Weighting). In Example 1, let us consider the distance between $Q = (0.2, 0.4)$ and $B = (0.9, 0.3)$ with respect to the color feature, i.e., $\mathcal{D}_Q(Q, B)$. With L_2 and equal weights, $\mathcal{D}_Q(Q, B)$ is 0.50. If the user weighs the first dimension twice as much as the second, $\mathcal{D}_Q(Q, B)$ is

$$\sqrt{\frac{2 * (0.2 - 0.9)^2 + (0.4 - 0.3)^2}{3}} = 0.58,$$

i.e., Q and B are not a close match in the color space. If she weighes the second dimension twice as much as the first, $\mathcal{D}_Q(Q, B)$ is

$$\sqrt{\frac{(0.2 - 0.9)^2 + 2 * (0.4 - 0.3)^2}{3}} = 0.41,$$

i.e., Q and B are a better match in this case than before. The intrafeature weights thus capture the user's subjective perception of similarity of color and determine what is a close match in the color space and what is not.

Example 4 (Multipoint Query). For example, let the query Q consist of multiple points in the color space. Let $Q = \{2, \{(0.2, 0.4), (0.4, 0.1)\}, \{0.7, 0.3\}$, *Euclidean (equal weights)*, and $B = (0.9, 0.3)$, then,

$$\begin{aligned} \mathcal{D}_Q(Q, B) = 0.7 * \sqrt{\frac{(0.2 - 0.9)^2 + (0.4 - 0.3)^2}{2}} \\ + 0.3 * \sqrt{\frac{(0.4 - 0.9)^2 + (0.1 - 0.3)^2}{2}} = 0.46. \end{aligned} \quad (3)$$

Note that the multipoint query is a generalization of the single point query (the latter is a special case of the former with $n_Q = 1$).

2.3 How do We Learn the Weights?

We model the user's perception using multiple points and weights, but how do we obtain the right values of the weights/functions that best capture the user's perception so that we can accurately instantiate the model for the user? One solution is for the user to explicitly specify the weights, as proposed by Motro in the VAGUE system [13]. Otherwise, we start with an arbitrary set of weights (e.g., all weights equal) which are modified later based on user feedback. We then execute the starting query and return the best matches to the user. If the user is not satisfied with the answers, either the query points do not properly capture the user's information need or the weights do not capture the user's perception accurately or both. We fix the above two problems by modifying the query points and the query weights, respectively, based on the user feedback as described below (e.g., MARS [24], [23], [18], Mindreader [11]):[5]

4. Note that this assumption is general since most commonly used distance functions (e.g., Manhattan distance, Euclidean distance, Bounding Box distance) are special cases of the L_p metric. However, this excludes distance functions that involve "cross correlation" among dimensions. Handling cross-correlated functions has been addressed in [25] and can be incorporated with the techniques developed in this paper.

5. Besides these two steps, some researchers have considered cross-correlation among the dimensions in S [11]. Although we do not consider cross-correlation in this paper, our techniques are applicable even when it is considered.

TABLE 1
Summary of Symbols, Definitions, and Acronyms

Symbol	Definition	Symbol	Definition
S	Multidimensional space	d_S	Dimensionality of S
O	A database object/record	Q	User query (multipoint)
n_Q	Number of points in Q	$\mathcal{P}_Q$	Set of n_Q points in Q
$P_Q^{(i)}$	The ith point in $\mathcal{P}_Q$	$\mathcal{W}_Q$	Set of n_Q weights associated with $\mathcal{P}_Q$
$w_Q^{(i)}$	ith weight in$\mathcal{W}_Q$ (associated with $P_Q^{(i)}$)	$\mu_Q{}^{(j)}$	Weight (intra-feature) for jth dimension of S
$\mathcal{D}_Q$	Distance function for Q	$\mathcal{D}_Q(Q,O)$	Overall distance between Q and O
QPM	Query Point Movement	QEX	Query Expansion
FR	Full Reconstruction Approach	SR	Selective Reconstruction Approach
LBD	Lower Bounding Distance	BR	Bounding Rectangle

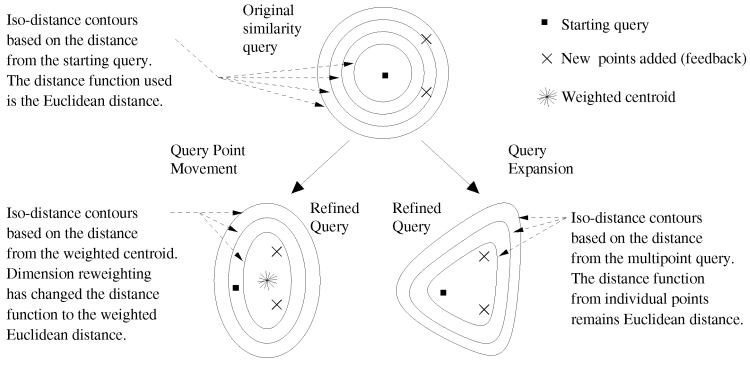

Fig. 1. Query refinement models.

1. **Query Modification** modifies the location of the query point(s) $\mathcal{P}_Q$ in the space S as well as their weights $\mathcal{W}_Q$. If the query interface is the explicit attribute value specification interface, the user can modify the attribute values explicitly. If the query interface is QBE, the user submits examples that better represent her information need and the system automatically derives from them the query points $\mathcal{P}_Q$ and their weights $\mathcal{W}_Q$. There are two models to do this:
   - *Query Point Movement (QPM)*. In this model, the relevant examples submitted by the user during feedback are represented by a single point: the weighted centroid (see Fig. 1). Effectively, the query point is moved toward the relevant objects. For example, if $A = (0.4, 0.5)$ and $B = (0.9, 0.3)$ are the two relevant examples submitted by the user and A is twice as relevant as B (derived from the relevance levels specified by the user [18]), the refined query for color is
   $$Q_{color} = \langle\{1, \left(\frac{2*0.4+0.9}{3}, \frac{2*0.5+0.3}{3}\right), 1, \mathcal{D}_Q\},$$
   where $\mathcal{D}_Q$ (i.e., the $\mu_{color}^{(j)}$s) is computed based on the reweighting criteria described below. This model was proposed in Mindreader [11] and MARS [22], [24], [23].
   - *Query Expansion (QEX)*. In this model, the submitted examples are represented by *multiple* points $\mathcal{P}_Q$ (called *representatives* [18]) giving rise to *multipoint* queries (see Fig. 1). The weight $w_Q{}^{(i)}$ of any representative $P_Q^{(i)}$ in the multipoint query is proportional to the total weight of all the objects $P_Q^{(i)}$ represents. For example, if $A = (0.4, 0.5)$ and $B = (0.9, 0.3)$ are the two relevant examples (see above) submitted by the user and A is twice as relevant as B, one possible choice of representatives are the points themselves, i.e.,
   $$Q_{color} = \langle 2, \{(0.4, 0.5), (0.9, 0.3)\}, \{0.67, 0.33\}, \mathcal{D}_Q\rangle.$$
   There are other possible choices of representatives (see [18] for representative selection techniques). This model was proposed in MARS [18], [19], [20].
2. **Reweighting** adjusts the dimension weights (i.e., the μ_Q^js in (1)) to better capture the user's perception within each feature and across features. If the refinement interface is the explicit weight specification interface, the user can modify the weights directly as in Motro's VAGUE system [13]. In a QBE environment (i.e., "more like this" interface), the system automatically derives the weights from the relevant examples submitted by the user [23], [11] (typically by estimating the importance of the dimension through the variance among values of relevant points in that dimension).

Once the query points and the weights are determined (i.e., the construction of the refined query is complete), the refined query is evaluated and the answers are returned to the user. The above refinement step indeed improves the quality of the answers and the answers progressively

TABLE 2
Incremental Query Evaluation (The k-NN Algorithm)

```
variable queue : MinPriorityQueue;
GetNext(Q)
while not queue.IsEmpty() do
        top=queue.Pop();
        if top is an object
          return top;
        else if top is a leaf node
          for each object O in top
              queue.push(O, D_Q(Q, O));
        else /* top is an index node */
          for each child node N in top
              queue.push(N, MINDIST(Q, N));
        endif
enddo
```

improve with more refinement iterations as demonstrated in references [22], [24], [23], [11], [18], [19], [20]. Note that how the query points/weights are obtained (i.e., whether we use the VAGUE approach, the MARS approach, or the Mindreader approach) is inconsequential to the discussion in the rest of the paper. *This paper discusses how to evaluate the refined query after it has been constructed using one of the above models: Hence, the techniques presented here will work irrespective of how it was constructed.* For those discussions, we refer readers to papers on VAGUE, Mindreader, and MARS [13], [23], [11], [18].

2.4 Query Evaluation

We first describe how to evaluate a "starting" query. While sequential scan is an option, it is prohibitively expensive when the database is large (compare Fig. 11). The most commonly used approach is to maintain a multidimensional index Idx (called the F-index). The $GetNext(Q)$ operation is implemented using the k-nearest neighbor search (k-NN) algorithm executing on the underlying index Idx (proposed in [10], [21]).

The *k*-NN algorithm. The algorithm is shown in Table 2. It maintains a priority queue that contains index nodes as well as data objects prioritized based on their distance from the query Q, i.e., the smallest item (either node or object) always appears at the top of the queue (min-priority queue). Initially, the queue contains only the root node (before the first $GetNext(Q)$ is invoked). At each step, the algorithm pops the item from the top of the queue: If it is an object, it is returned to the caller; if it is a node, the algorithm computes the distance of each of its children from the query and pushes it into the queue. The distances are computed as follows: If the child is a data object, the distance is that between the query and the object point; if the child is a node, the distance is the minimum distance (referred to as MINDIST [10]) from the query point to the nearest (according to the distance function) boundary of the node.

2.5 Problem Statement

In this paper, we address the following problem: Given a database DB and a refined query Q, how to evaluate Q and return $Ans(Q)$ as *efficiently* as possible. As mentioned before, since Q is given to us (by the refinement model), the techniques proposed in the paper have no effect on the quality of the answers, i.e., on $Ans(Q)$; the only goal is efficiency. To achieve the goal, we need to address the following problems:

- **Multipoint queries and arbitrary distance functions**. The $GetNext(Q)$ operation is performed by executing the *k*-NN algorithm on the corresponding F-index Idx. Traditionally, the *k*-NN algorithm has been used for single point queries, i.e., Q is a single point in S and the Euclidean distance function, i.e., $\mathcal{D}_Q$ is Euclidean (no dimension weights) [21], [10]. In a query refinement environment, the above assumptions do not hold. We discuss how we implement $GetNext(Q)$ efficiently when 1) Q can be a multipoint query and 2) $\mathcal{D}_Q$ can be any L_p metric and can have arbitrary dimension weights.[6] We discuss this in Section 3.
- **Optimization of refined queries**. The multipoint query optimization is a necessary building block for the main contribution of this paper. Our main contribution is the development of techniques to execute refined queries efficiently. We focus our work on first achieving I/O optimality and then to further improve by achieving computational optimality. We show that, in general, it is not possible to achieve computational optimality under the design constraints (e.g., incremental processing) and propose a heuristic approach to get close to the optimality criteria. We present our techniques in Section 4.

3 k-Nearest Neighbor Algorithm for Multipoint Queries

The $GetNext(Q)$ function in Table 2 can handle only single point queries, i.e., $n_Q = 1$. One way to handle a multipoint query $Q = \langle n_Q, \mathcal{P}_Q, \mathcal{W}_Q, \mathcal{D}_Q \rangle$ is by incrementally determining the next nearest neighbor $NN_Q^{(i)}$ of each point $P_Q^{(i)} \in \mathcal{P}_Q$ by invoking $GetNext(P_Q^{(i)})$ (for $i = [1, n_Q]$) computing its overall distances $\mathcal{D}_Q(Q, NN_Q^{(i)})$ (using (2)) and storing them in a buffer until we are sure that we have found the next nearest neighbor to Q. This technique, referred to as the *Multiple Expansion Approach*, was proposed in MARS [20] and FALCON [28], which showed it to perform better than other approaches like the centroid-based expansion and single point expansion approaches [20].

In this paper, we propose an alternate approach to evaluating multipoint queries. We refer to it as the *multipoint approach*. In this approach, we modify the $GetNext(Q)$ function to be able to handle multipoint queries, i.e., it should be able to compute the distances of the nodes and

6. Note that, it is possible to avoid multipoint queries (i.e., support only single point queries) by always using QPM as the query modification technique. However, Porkaew et al. show that QEX-based techniques usually perform better than QPM-based ones in terms of retrieval effectiveness [19], [18]. Hence, supporting multipoint queries efficiently is important for effective and efficient query refinement. Also, since multipoint queries are a generalization of single point queries, supporting multipoint queries makes the techniques developed in this paper applicable irrespective of the query modification technique used.

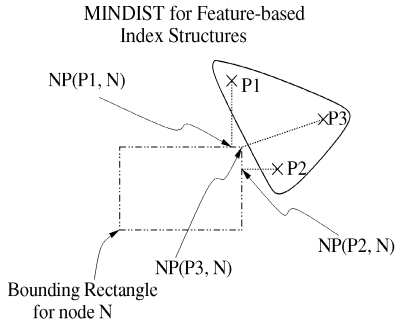

Fig. 2. MINDIST computation.

objects directly from the multipoint query Q and explore the priority queue based on those distances. The basic algorithm in Table 2 does not change, the only changes are the distance computations. As before, there are two types of distance computations: 1) distance of an object to the multipoint query, i.e., $\mathcal{D}_Q(Q,O)$, and 2) distance of a node to the multipoint query (MINDIST), i.e., $MINDIST(Q,N)$. The computation of 1 follows directly from (2). So, all we need to do is to define the distance $MINDIST(Q,N)$ between a multipoint query Q and a node N of the F-index. As in the single point case, the definition of MINDIST depends on the shape of the node boundary. We assume that the F-index Idx is a feature-based index (e.g., Hybrid tree, R-tree, X-tree, KDB-tree, hB-tree) because distance-based index structures (e.g., SS-tree, SR-tree, TV-tree, M-tree) cannot handle arbitrary dimension weights (see [3] for proof). In a feature-based index, the bounding region (BR) of each node N is always, either explicitly or implicitly, a d_S-dimensional rectangle in the feature space S [2].

Definition 2 (MINDIST for Multipoint Queries). *Given the bounding rectangle (BR) $R_N = \langle L, H\rangle$ of a node N, where $L = \langle l_1, l_2, \ldots, l_{d_S}\rangle$ and $H = \langle h_1, h_2, \ldots, h_{d_S}\rangle$ are the two endpoints of the major diagonal of R_N, $l_i \le h_i$ for $1 \le i \le d_S$. The nearest point $NP(P_Q^{(i)}, N)$ in R_N to each point $P_Q^{(i)}$ in the multipoint query $Q = \langle n_Q, \mathcal{P}_Q, \mathcal{W}_Q, \mathcal{D}_Q\rangle$ is defined as follows (explained in Fig. 2):*

$$NP(P_Q^{(i)}, N)[j] = \begin{cases} l_j & \text{if } P_Q^{(i)}[j] < l_j \\ h_j & \text{if } P_Q^{(i)}[j] > h_j \\ P_Q^{(i)}[j] & \text{otherwise,} \end{cases} \quad (4)$$

where $NP[j]$ denotes the position of NP along the jth dimension of the feature space S, $1 \le j \le d_S$. $MINDIST(M,N)$ is defined as:

$$MINDIST(Q,N) = \sum_{i=1}^{n_Q} w_Q^{(i)} \mathcal{D}_Q\Big(P_Q^{(i)}, NP\Big(P_Q^{(i)}, N\Big)\Big). \quad (5)$$

The $GetNext$ function can handle arbitrary distance functions $\mathcal{D}_Q$ (i.e., L_p metrics with arbitrary intrafeature weights). The above algorithm is correct (i.e., $GetNext(Q)$ returns the next nearest neighbor of Q) if $MINDIST(Q,N)$ always *lower bounds* $\mathcal{D}_Q(Q,T)$, where T is any point stored under N.

Lemma 1 (Correctness of GetNext algorithm). *$MINDIST(Q,N)$ lower bounds $\mathcal{D}_Q(Q,T)$.*

Proof. Let N be a node of the index structure, R be the corresponding bounding rectangle, and T be any object under N. Let us assume that $\mathcal{D}_Q$ is monotonic. We need to show that $MINDIST(Q,N) \le \mathcal{D}_Q(Q,T)$. Let $T = \langle t_1, t_2, \ldots, t_{d_S}\rangle$ be the d_S-dimensional vector and $R = \langle l_1, l_2, \ldots, l_{d_S}, h_1, h_2, \ldots, h_{d_S}\rangle$ be the d_S-dimensional bounding rectangle. Since T is under N, T must be spatially contained in R.

$$l_j \le t_j \le h_j, j = [1, d_S]. \quad (6)$$

Using the definition of $NP(P_Q^{(i)}, N)[j]$ and $\mathcal{D}_Q(P_Q^{(i)}, T)$, (6) implies

$$|P_Q^{(i)}[j] - NP\Big(P_Q^{(i)}, N\Big)[j]| \le |P_Q^{(i)}[j] - T[j]|, j = [1, d_S], \quad (7)$$

$$\Rightarrow \mathcal{D}_Q\Big(P_Q^{(i)}, NP\Big(P_Q^{(i)}, N\Big)\Big) \le \mathcal{D}_Q\Big(P_Q^{(i)}, T\Big) \quad (8)$$
$$since\ \mathcal{D}\ is\ a\ weighted\ L_p\ metric,$$

$$\Rightarrow \sum_{i=1}^{n} w_i \mathcal{D}_Q\Big(P_Q^{(i)}, NP\Big(P_Q^{(i)}, N\Big)\Big) \le \sum_{i=1}^{n} w_Q^{(i)} \mathcal{D}_Q\Big(P_Q^{(i)}, T\Big)$$
$$since\ w_i \ge 0, \quad (9)$$

$$\Rightarrow MINDIST(Q,N) \le D_Q(Q,T). \quad (10)$$

□

Our experiments show that the multipoint approach is significantly more efficient compared to the previously proposed multiple expansion approach (see Section 5).

4 Evaluation of Refined Queries

A naive way to evaluate a single feature refined query is to treat it just like a starting query and execute it from scratch as discussed in Section 2.4. This approach is wasteful as we can save most of the execution cost of the refined query, both in terms of disk accesses (I/O cost) and distance calculations (CPU cost), by exploiting information generated during the previous iterations of the query. In this section, we discuss how to optimize the $GetNext(Q)$ function. In the naive approach, the same nodes of Idx may be accessed from scratch by the k-NN algorithm repeatedly iteration after iteration. In other words, a node of Idx is being accessed from disk multiple times during the execution of a query (over several iterations) causing unnecessary disk I/O. For example, let us consider the query shown in Fig. 3. Region $R1$ represents the iso-distance range corresponding to the distance of the kth NN of the starting query—it is the region in the feature space already explored to return the top k matches to the starting query, i.e., all nodes overlapping with $R1$ were accessed and all objects (k in number) in that region were returned. $R2$ represents the iso-distance range corresponding to the distance of the kth NN of the refined query—it is the region that needs to be explored to answer the refined

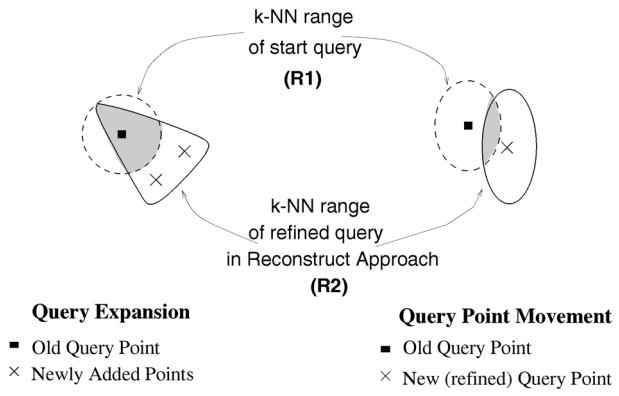

Fig. 3. I/O Cost of naive and reconstruction approaches.

query, i.e., all nodes overlapping with $R2$ need to be explored and all objects (k in number) in that region need to be returned. If no buffering is used, to evaluate the refined query, the naive approach would access *all* the nodes overlapping with $R2$ from the disk, thus accessing those nodes that overlap with the shaded region from the disk twice. If traditional LRU buffering is used, some of the nodes overlapping with the shaded region may still be in the database buffer and would not require disk accesses during the evaluation of the refined query. Our experiments show that, for reasonable buffer sizes, most of the nodes in the shaded region get ejected from the buffer before they are accessed by the refined query. This is due to the use-recency-based page replacement policy used in database buffers. The result is that the naive buffering approach still needs to perform large numbers of repeated disk accesses to evaluate refined queries (see Fig. 8). Concurrent query sessions by different users and user think time between iterations would further reduce buffer hit ratio, causing the naive approach to perform even more poorly.

Our goal in this paper is to develop a more I/O-efficient technique to evaluate refined queries. If a node is accessed from disk once during an iteration of a query, we should keep it in main memory (by caching) so that it is not accessed from disk again in any subsequent iteration of that query. In Fig. 3, to evaluate the refined query, an I/O optimal algorithm would access from disk only those nodes that overlap with region $R2$, but do not overlap with region $R1$ and access the remaining nodes (i.e., those that overlap with the shaded region) from memory (i.e., from the cache). We formally state I/O optimality as follows:

Definition 3 (I/O Optimality). *Let N be a node of the F-index Idx. An algorithm executing the refined query Q^{new} is I/O optimal if it makes a disk access for N* only if *1) there exists at least one desired answer O such that $\mathcal{D}_Q(Q^{new}, O) \geq MINDIST(Q^{new}, N)$ and 2) N is not accessed during any previous iteration (say Q^{old}).*

Condition 1 is necessary to guarantee no false dismissals, while Condition 2 guarantees that a node is accessed from the disk at most once throughout the execution of the query across all iterations of refinement. The naive approach is not I/O optimal as it does not satisfy condition 2.

We achieve I/O optimality by caching on a per-query basis instead of using a common LRU buffer for all queries. To ensure condition 2, we need to "cache" the contents of each node accessed by the query in any iteration and retain them in memory till the query ends (i.e., till the last iteration) at which time the cache can be freed and the memory can be returned to the system. Since the priority queue generated during the execution of the starting query contains the information about the nodes accessed, we can achieve the above goal by caching the priority queue. We assume that, for each item (node or object), the priority queue stores the feature values of the item (i.e., the bounding rectangle if the item is a node, the feature vector if it is an object) in addition to its id and its distance from the query. This is necessary since, in some approaches, we need to recompute the distances of these items based on the refined queries. We also assume that the entire priority queue fits in memory as is commonly assumed in other works on nearest-neighbor retrieval [12], [26].[7] In the following sections, we describe how we can use the priority queue generated during the starting query (that contains items ordered based on distance from the starting query) to efficiently obtain the top k matches based on their distances from the refined query.

4.1 Full Reconstruction (FR)

We first describe a simple approach called the full reconstruction approach. In this approach, to evaluate the refined query, we "reconstruct" a new priority queue $queue_{new}$ from the cached priority queue $queue_{old}$ by popping each item from $queue_{old}$, recomputing its distance from the refined query Q^{new}, and then pushing it into $queue_{new}$. We discard the old queue when all items have been transferred. We refer to this phase as the *transfer phase*. Subsequently, the multipoint k-NN algorithm proposed in Section 3 is invoked with Q^{new} as the query on $queue_{new}$. We refer to this phase as the *explore phase*. The queue is handed from iteration to iteration through the reconstruction process, i.e., the $queue_{new}$ of the previous iteration becomes the $queue_{old}$ of the current iteration and is used to construct the $queue_{new}$ of the current iteration. Thus, if a node is accessed once, it remains in the priority queue for the rest of the query and, in any subsequent iteration, is accessed directly from the queue (which is assumed to be entirely in memory) instead of reloading from the disk. The entire sequence of iterations is managed using two queues and swapping their roles (old and new) from iteration to iteration. It is easy to see that this approach is I/O optimal.

Now that we have achieved I/O optimality, let us consider the CPU cost of the approach. The CPU cost is proportional to the number of distance computations performed. The algorithm performs distance computations during the transfer phase (one computation each time an item is transferred from $queue_{old}$ to $queue_{new}$) and also during

7. This assumption is reasonable when the number k of top matches requested is relatively small compared to the size of the database which is usually the case [1]. For example, in our experiments, when $k = 100$, the size of the priority queue varies between 512KB and 640KB (the size of the F-index is about 11MB). The techniques proposed in the paper would work even if the priority queue does not fit in memory and would still perform better than the naive approach; however, they may not be I/O optimal.

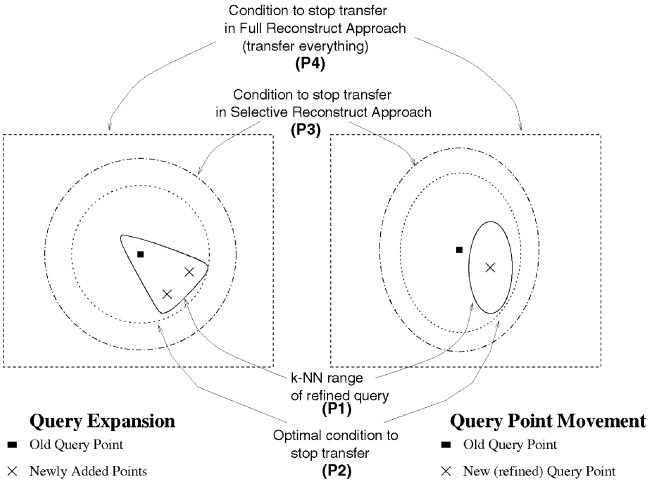

Fig. 4. CPU cost of full and selective reconstruction approaches.

the explore phase (one computation for each child in the node being explored). Since the technique is I/O optimal, there is no node being explored unnecessarily, i.e., we cannot save any distance computations in the explore phase (already optimal). However, we can reduce the CPU cost of the transfer phase if we avoid transferring each and every item from $queue_{old}$ to $queue_{new}$ without sacrificing correctness. In other words, we should transfer only those items that are *necessary* to transfer to avoid false dismissals.

4.2 Selective Reconstruction (SR)

Our Selective Reconstruction approach transfers incrementally only as many items as are needed to produce the *next* result. As with I/O optimality in FR, our goal here is to ensure that any item comparisons and transfers are truly necessary to produce a result. We first describe how to achieve this within a single feedback iteration and describe the problems found, then extend it to function properly for multiple iterations and discuss the limits of this technique.

4.2.1 Single Iteration Selective Reconstruction

We modify the FR approach to reduce the CPU cost while retaining the I/O optimality. We only transfer those items from $queue_{old}$ to $queue_{new}$ that are necessary to ensure correctness. We achieve this by imposing a *stopping condition* on the transfer as illustrated in Fig. 4. Suppose that $GetNext$ is returning the kth NN of Q^{new}. $P1$ is the iso-distance range corresponding to the distance of the kth NN of Q^{new}, i.e., any object outside this region is at least as far as the kth NN. $P2$ is the smallest iso-distance range from Q^{old} totally containing $P1$. Therefore, to return the kth NN of Q^{new}, we can stop transferring when we reach the range $P2$ in $queue_{old}$ without sacrificing correctness since, at this point, any object left unexplored in $queue_{old}$ is at least as far from Q^{new} as the kth NN of Q^{new}. This represents the *optimal* stopping condition because we cannot stop before this without sacrificing correctness. In other words, if we stop before this, there will be at least one false dismissal. The FR approach has no stopping condition at all as is graphically shown in Fig. 4 by region $P4$, i.e., the entire feature space.

While in the FR approach, the entire transfer phase is followed by the entire explore phase, here the two phases are interspersed. The algorithm pseudocode is shown in Table 3. During the transfer phase, we transfer items from $queue_{old}$ to $queue_{new}$ until we are sure that no item exists in $queue_{old}$ that is closer to Q^{new} than the top item of $queue_{new}$ (i.e., the next item to be explored—top_{new}). In other words, the lower bounding distance $LBD(queue_{old}, Q^{new})$ of any item in $queue_{old}$ is greater than or equal to the distance between top_{new} and Q^{new} ($top_{new}.distance = \mathcal{D}_Q(Q^{new}, top^{new})$), this is the stopping condition in line 5. If we are sure, we go into the explore phase, i.e., we explore top_{new}. If top_{new} is an object, it is *guaranteed* to be the next best match to Q^{new} and is returned to the caller. Otherwise, it is a node; in that case, we access it from disk, compute the distance of each child from Q^{new}, and push it back into $queue_{new}$; then, we return to the transfer phase. It is easy to see that the selective reconstruction technique is I/O optimal.

TABLE 3
The Single Iteration GetNext Algorithm
for Refined Queries Using the SR Approach

```
variable Q^old: Query // (of previous iteration)
variable queue_old: MinPriorityQueue // (of previous iteration)
variable queue_new: MinPriorityQueue // (of current iteration)
GetNext(Q^new)

1  while (not queue_new.IsEmpty()) do
       /* TRANSFER PHASE (Lines 2-10)*/
2      while (not queue_old.IsEmpty()) do
3          top_old=queue_old.Top();
4          top_new=queue_new.Top();
           /* STOPPING CONDITION (Line 5)*/
5          if (LBD(queue_old, Q^new) ≥ top_new.distance) break
6          else queue_old.Pop();
7              Recompute top_old.distance based on Q^new
8              Push top_old into queue_new
9          endif
10     enddo
       /* EXPLORE PHASE (Lines 11-21) */
11     queue_new.Pop();
12     if top_new is an object
13       return top_new;
14     else if top_new is a leaf node
15       for each object in top_new
16         queue_new.push(object, D_Q(Q^new, object));
17     else /* top_new is an index node */
18       for each child of top_new
19         queue_new.push(child, MINDIST(Q^new, child));
20     endif
21 enddo
```

Let us now explain the notion of LBD in the stopping condition (in line 5). $LBD(queue_{old}, Q^{new})$ denotes the the smallest distance any item in $queue_{old}$ can have with respect to the new query Q^{new}, i.e.,

$$LBD(queue_{old}, Q^{new}) \leq \mathcal{D}_Q^{new}(Q^{new}, O)$$

for all $O \in queue_{old}$. To complete the algorithm, we need to specify how to compute $LBD\ (queue_{old}, Q^{new})$. We need to compute it without incurring *any* extra cost, i.e., in a single constant time operation. If we can obtain the optimal $LBD(queue_{old}, Q^{new})$ (i.e., there exists an object $O \in queue_{old}$, where $LBD(queue_{old}, Q^{new}) = \mathcal{D}_Q^{new}(Q^{new}, O)$), we can achieve the optimal stopping condition (shown by $P2$ in Fig. 4) and, hence, the optimal CPU cost.

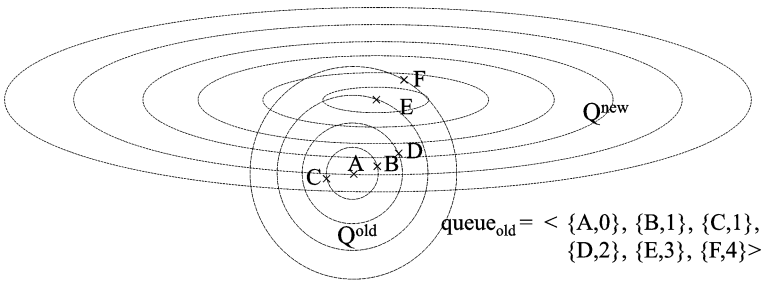

Fig. 5. LBD example.

Example 5 (SR Algorithm). Fig. 5 shows old and new queries (Q^{old} and Q^{new}) with iso-distance curves at one unit intervals. For example, in the figure, the outermost iso-distance curve of Q^{new} is the optimal stopping condition for any points within the first iso-distance curve of Q^{old} (i.e., points A, B, C). Following Fig. 5, the SR algorithm examines $queue_{old}$ in distance order. It first compares the top item from $queue_{new}$ which at first is empty (and, thus, 0) to the LBD of point A and finds that the condition is not met, A is then transferred to $queue_{new}$. This continues until E is retrieved from $queue_{old}$. For E, the optimal stopping condition should be true as there are no items in $queue_{old}$ that can possibly be closer to Q_{new}. Thus, E would be returned by GetNext.

Lemma 2 (CPU Optimality). *The number of distance computations (i.e., the stopping condition) is optimal if $LBD(queue_{old}, Q^{new})$ is optimal.*

Proof. Let I_{new} be an item in $queue_{new}$. Let I_{old} be the item at the top of $queue_{old}$. Assuming $LBD(queue_{old}, Q^{new})$ is tight, we need to show that, if I_{old} is transferred to $queue_{new}$, there exists an item I'_{old} in $queue_{old}$ such that $\mathcal{D}_Q^{old}(I_{old}, Q^{old}) \leq \mathcal{D}_Q^{old}(I'_{old}, Q^{old})$ and

$$\mathcal{D}_Q^{new}(I'_{old}, Q^{new}) \leq \mathcal{D}_Q^{new}(I_{new}, Q^{new}).$$

Let us assume I_{old} is transferred, i.e, the stopping condition is not satisfied ($LBD(queue_{old}, Q^{new}) < top_{new}.distance$). Since LBD is tight, there exists an unexplored item I'_{old} in $queue_{old}$ such that $\mathcal{D}_Q^{new}(I'_{old}, Q^{new}) \leq top_{new}.distance$. Since

$$\begin{aligned} top_{new}.distance &\leq \mathcal{D}_Q^{new}(I_{new}, Q^{new}), \mathcal{D}_Q^{new}(I'_{old}, Q^{new}) \\ &\leq \mathcal{D}_Q^{new}(I_{new}, Q^{new}). \end{aligned}$$

Since I_{old} is at the top of $queue_{old}$ and I'_{old} is yet unexplored, $\mathcal{D}_Q^{old}(I_{old}, Q^{old}) \leq \mathcal{D}_Q^{old}(I'_{old}, Q^{old})$. □

We believe that it is not possible to compute the optimal LBD without exploring $queue_{old}$ (which would defeat the purpose of the reconstruction approach). Instead, we derive a conservative estimate of LBD. For correctness, the estimated LBD must be an LBD, i.e., it must always underestimate the optimal LBD. The closer the estimate to the optimal LBD, the fewer the number of transfers, and the lower the CPU cost. In the derivation, we exploit the fact that $top_{old}.distance$ lower bounds the distance of any item in $queue_{old}$ from the *old* query Q^{old}. For simplicity of notation, we define the distance $D(Q^{old}, Q^{new})$ between two multipoint queries as follows: Let $Q^{old} = \langle n_Q^{old}, \mathcal{P}_Q^{old}, \mathcal{W}_Q^{old}, \mathcal{D}_Q^{old} \rangle$ be the old multipoint query where $\mathcal{P}_Q^{old} = \{P_Q^{old(1)}, \cdots, P_Q^{old(n_Q^{old})}\}$ and $\mathcal{W}_Q^{old} = \{w_Q^{old(1)}, \cdots, w_Q^{old(n_Q^{old})}\}$. Let

$$Q^{new} = \langle n_Q^{new}, \mathcal{P}_Q^{new}, \mathcal{W}_Q^{new}, \mathcal{D}_Q^{new} \rangle$$

be the new multipoint query where

$$\mathcal{P}_Q^{new} = \{P_Q^{new(1)}, \cdots, P_Q^{new(n_Q^{new})}\}$$

and

$$\mathcal{W}_Q^{new} = \{w_Q^{new(1)}, \cdots, w_Q^{new(n_Q^{new})}\}.$$

The distance $D(Q^{old}, Q^{new})$ is defined as the weighted all-pairs distance between the constituent points:

$$D(Q^{old}, Q^{new}) = \sum_{i=1}^{n_Q^{old}} \left(w_Q^{old(i)} \sum_{j=1}^{n_Q^{new}} w_Q^{new(j)} \mathcal{D}_Q^{new}\left(P_Q^{old(i)}, P_Q^{new(j)}\right) \right). \tag{11}$$

Also, let $\mu_Q^{old(k)}, k = [1, d_S]$ and $\mu_Q^{new(k)}, k = [1, d_S]$ denote the old and new intrafeature weights, respectively:

$$\mathcal{D}_Q^{old}(T_1, T_2) = \left[\Sigma_{j=1}^{d_S} \mu_Q^{old(j)} (|T_1[j] - T_2[j]|)^p\right]^{1/p}, \tag{12}$$

$$\mathcal{D}_Q^{new}(T_1, T_2) = \left[\Sigma_{j=1}^{d_S} \mu_Q^{new(j)} (|T_1[j] - T_2[j]|)^p\right]^{1/p}. \tag{13}$$

The following lemma defines the stopping condition (SC) in line 5.

Lemma 3 (Stopping Condition).

$$\left(\frac{top_{old}.distance}{K} - D(Q^{new}, Q^{old})\right)$$

lower bounds the distance of any unexplored object in $queue_{old}$ from Q^{new}, where

$$K = max_{k=1}^{d_S} \left(\frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}\right).$$

Proof. Let I be any item in $queue_{old}$. We need to show

$$\mathcal{D}_Q^{old}(I, Q^{old}) \geq top_{old}.distance \Rightarrow \mathcal{D}_Q^{new}(Q^{new}, I) \geq \frac{top_{old}.distance}{max_{k=1}^{m} \frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}} - \mathcal{D}_Q^{new}(Q^{new}, Q^{old}).$$

Let us assume

$$\mathcal{D}_Q^{old}(Q^{old}, I) \geq top_{old}.distance. \quad (14)$$

Let $P_Q^{old(i)} \in P_Q^{old}$ and $P_Q^{new(j)} \in P_Q^{new}$. By triangle inequality,

$$\mathcal{D}_Q^{new}(P_Q^{new(j)}, I) \geq \mathcal{D}_Q^{new}(P_Q^{old(i)}, I) - \mathcal{D}_Q^{new}(P_Q^{new(j)}, P_Q^{old(i)}). \quad (15)$$

From (12) (assuming $\mathcal{D}$ is monotonic),

$$\mathcal{D}_Q^{new}\left(P_Q^{old(i)}, I\right) \geq \frac{\mathcal{D}_Q^{old}\left(P_Q^{old(i)}, I\right)}{max_{k=1}^{d_S} \frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}}. \quad (16)$$

From (15) and (16),

$$\mathcal{D}_Q^{new}\left(P_Q^{new(j)}, I\right) \geq \frac{\mathcal{D}_Q^{old}\left(P_Q^{old(i)}, I\right)}{K} - \mathcal{D}_Q^{new}\left(P_Q^{new(j)}, P_Q^{old(i)}\right), \quad (17)$$

where

$$K = max_{k=1}^{m} \frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}.$$

Multiplying both sides by $w_Q^{old(i)}$ (since $w_Q^{old(i)} \geq 0$) and summing over $i = 1$ to n_{old},

$$\sum_{i=1}^{n_{old}}\left(w_Q^{old(i)}\mathcal{D}_Q^{new}\left(P_Q^{new(j)}, I\right)\right) \geq \frac{1}{K}\sum_{i=1}^{n_{old}}\left(w_Q^{old(i)}\mathcal{D}_Q^{old}\left(P_Q^{old(i)}, I\right)\right) - \sum_{i=1}^{n_{old}}\left(w_Q^{old(i)}\mathcal{D}_Q^{new}\left(P_Q^{new(j)}, P_Q^{old(i)}\right)\right), \quad (18)$$

$$\Rightarrow \mathcal{D}_Q^{new}\left(P_Q^{new(j)}, I\right) \geq \frac{\mathcal{D}_Q^{old}(Q^{old}, I)}{K} - \mathcal{D}_Q^{new}\left(P_Q^{new(j)}, Q^{old}\right). \quad (19)$$

Multiplying both sides by $w_Q^{new(j)}$ (since $w_Q^{new(j)} \geq 0$) and summing over $j = 1$ to n_{new},

$$\sum_{j=1}^{n_{new}}\left(w_Q^{new(j)}\mathcal{D}_Q^{new}\left(P_Q^{new(j)}, I\right)\right) \geq \frac{1}{K}\sum_{j=1}^{n_{new}}\left(w_Q^{new(j)}\mathcal{D}_Q^{old}(Q^{old}, I)\right) - \sum_{j=1}^{n_{new}}\left(w_Q^{new(j)}\mathcal{D}_Q^{new}\left(P_Q^{new(j)}, Q_Q^{old}\right)\right), \quad (20)$$

$$\Rightarrow \mathcal{D}_Q^{new}(Q^{new}, I) \geq \frac{\mathcal{D}_Q^{old}(Q^{old}, I)}{K} - \mathcal{D}_Q^{new}(Q^{new}, Q^{old}). \quad (21)$$

Equations (14) and (21) imply

$$\mathcal{D}_Q^{new}(Q^{new}, I) \geq \frac{top_{old}.distance}{max_{k=1}^{m} \frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}} - \mathcal{D}_Q^{new}(Q^{new}, Q^{old}). \quad (22)$$

□

The $LBD(queue_{old}, Q^{new})$ in the stopping condition in line 5 of the SR algorithm (see Table 3) can now be replaced by the above estimate. Thus, the stopping condition is:

$$\left(\frac{top_{old}.distance}{K} - D(Q^{new}, Q^{old})\right) \geq top_{new}.distance. \quad (23)$$

Note that the above SC is general. It is valid irrespective of the query modification technique used, i.e., it is valid for both QPM and QEX models. It is also valid irrespective of whether intrafeature reweighting is used or not. If intrafeature reweighting is not used (i.e., all weights are considered equal), K turns out to be 1, which means there is no effect of intrafeature reweighting at all. Also, note that both K and $D(Q^{new}, Q^{old})$ are computed just once during the execution of the refined query. The computation of K is proportional only to the number of dimensions d_S in the space. The computation of LBD involves just two arithmetic operations, a division followed by a subtraction. Since we use an estimate of LBD and not the exact LBD, in general, SC is not optimal. This implies that, as shown in Fig. 4, the stopping condition is range $P3$ and not the optimal range $P2$. We perform experiments to compare the above stopping condition with the optimal one in terms of CPU cost (see Section 5).

Example 6. Consider again Fig. 5. The first item from $queue_{old}$ is A. Its distance to Q^{new} is 5 and Q^{new} has a high weight (0.66) in the vertical dimension and a small weight (0.33) in the horizontal dimension. So,

$$K = \max\left\{\frac{0.5}{0.33}, \frac{0.5}{0.66}\right\} = \frac{0.5}{0.33} = 1.5,$$

and notice that $D(Q^{new}, Q^{old}) = 5$. So, the LBD estimate for A is $\frac{0}{1.5} - 5 = -5$. Likewise, for point E, the LBD is $\frac{3}{1.5} - 5 = 2 - 5 = -3$. Here, we see the lower bounding nature of our approximation. If our LBD estimate were optimal, the LBD for point E should be 0 and E could be returned.

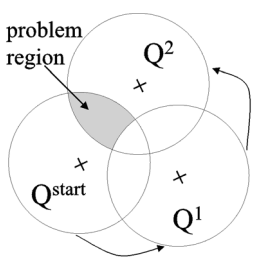

Fig. 6. Problem region for single iteration SR approach.

TABLE 4
The Multiiteration GetNext Algorithm
for Refined Queries Using the SR Approach

```
variable history: ⟨Q^(i), queue_i, K_I, D(Q^(i), Q^new)⟩: // query and queue history
variable queue_new: MinPriorityQueue // (of current iteration)
GetNext(Q^new)

1  while (not queue_new.IsEmpty()) do
       /* TRANSFER PHASE (Lines 2-10)*/
2      while (∃i | ¬history.queue_i.IsEmpty()) do
3          i=i | min{LBD(history.queue_i, Q^new)};
4          top_i=history.queue_i.Top();
5          top_new=queue_new.Top();
           /* STOPPING CONDITION (Line 6)*/
6          if (LBD(history.queue_i, Q^new) ≥ top_new.distance) break
7          else history.queue_i.Pop();
8              Recompute top_i.distance based on Q^new
9              Push top_i into queue_new
10         endif
11     enddo
       /* EXPLORE PHASE (Lines 11-21) */
12     queue_new.Pop();
13     if top_new is an object
14       return top_new;
15     else if top_new is a leaf node
16       for each object in top_new
17           queue_new.push(object, D_Q(Q^new, object));
18     else /* top_new is an index node */
19       for each child of top_new
20           queue_new.push(child, MINDIST(Q^new, child));
21     endif
22 enddo
```

4.2.2 Multiple Iteration Selective Reconstruction

The above discussion of the SR approach considered how to evaluate the first iteration of the query refinement given the priority queue $queue_{start}$ generated during the execution of the start query (iteration 0). The discussion omitted what to do when the iteration finishes and a new feedback iteration starts. If we discard the old queue, create a new, empty queue, and follow the same algorithm, we will sacrifice the correctness of the algorithm. Consider Fig. 6 which shows the starting query Q^{start}, and two feedback iterations Q^1 and Q^2 using the simple QPM model and no reweighting. If we follow the SR algorithm for Q^2, we will miss answers in the *problem region*. These items were present in $queue_{start}$, but not transferred to $queue_1$ (some may be included since our LBD is an approximation); therefore, they are irretrievably lost for Q^2.

To properly handle multiple feedback iterations using the SR algorithm, we must accommodate for unprocessed items in earlier iterations. Copying the remaining items to the new queue would be equivalent to the FR approach. Instead, our solution is to maintain a history of the queries and queues from the start. This approach stores the same number of items as the FR approach (no item is ever discarded), and has the benefits of the SR approach in exchange for some administrative complexity. Table 4 shows the algorithm that accounts for multiple iterations. We maintain a history list of tuples $\langle Q^{(i)}, queue_i, K_i, D(Q^{(i)}, Q^{new})\rangle$ that keep each query, queue, and additional parameters for each iteration. When a new iteration starts, we update all tuples in the history with the corresponding new values for K_i and $D(Q^{(i)}, Q^{new})$. The algorithm then extends the single iteration SR by selecting in each iteration the minimum LBD among all queries in the history to maintain the LBD correctness. The CPU complexity of the algorithm is only increased by selecting the minimum LBD among all previous queues; as discussed above, this means two arithmetic operations per iteration in addition to the single iteration SR algorithm. The storage requirements for each iteration consist of the query Q_i which is dependent only on the number of points and the dimensionality (but typically only a few hundred bytes in size), and the values K_i and $D(Q^{(i)}, Q^{new})$ which are just two constants. The total number of items in all queues is at most that of the FR approach (only necessary nodes are expanded, and no items are dropped).

4.2.3 Cost-Benefit Based Use of SR

As later iteration queries drift farther apart from the starting query, the storage and computation costs increase as more items and auxiliary information accumulate. We turn our attention to the problem of deciding whether a new query iteration should be evaluated by continuing the use of SR, or naively (followed by iterations using SR). We distinguish three distinct considerations involved:

- **I/O cost.** From an I/O perspective, the FR and SR techniques are I/O optimal; therefore, they always reduce the number of I/Os over a naive execution regardless of the convergence properties of the feedback. When queries converge over iterations, as is generally the case [28], [20], [11], the I/O count per iteration asymptotically approaches 0 (see experiments).
- **CPU cost.** With more items from early iterations lingering in the queues, the number of computations increases. An absolute upper bound on the number of iterations can be obtained by observing that each new SR iteration necessarily requires us to revise the K_i and $D(Q^{(i)}, Q^{new})$ values for each iteration.[8] The computation of these values depends on the dimensionality of the space and the number of points in the queries. For simplicity, we estimate the cost to be the same as one distance computation. By maintaining simple statistics, we can estimate the number of distance computations used in a naive reevaluation to be the number of items for which a computation was made in the starting query Q^0. This is the number of items in $history.queue_0$ plus the k items returned to the user plus any intermediate items that were discarded $m_0^{discarded}$. A definite CPU based upper bound is then when:

$$k + m_0^{discarded} + \mid history.queue_0 \mid < i,$$

that is, there are more iterations than elements in the starting query.

This bound is generally much too high. A new iteration starts by exploring the current and earlier queues and transfers items until the stopping condition permits the return of a result. In practice,

8. This must be done either explicitly as in our description or implicitly if we avoid caching these values in the history.

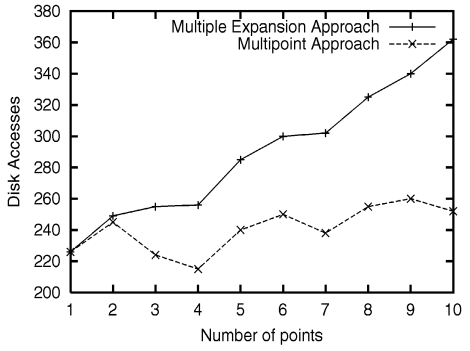

Fig. 7. I/O cost of multipoint approach and multiple expansion approach.

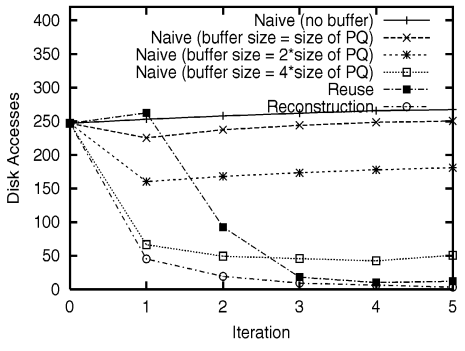

Fig. 8. I/O cost of naive and reconstruction approaches for EX queries.

this generally implies several hundred distance computations that must be done after the new iteration started. SR adds computation overhead proportional to the number of iterations over FR. A better estimate can thus be obtained by deducting those initial computations from the number of iterations. Let $t^{(i)}$ be the number of items transferred during iteration i to the current queue from all earlier queues before the first result is returned to the user. Then, a better estimate is:

$$k + m_0^{discarded} + \mid history.queue_0 \mid + t^{(i-1)} < i.$$

From our experiments, SR still pays off even after 50 iterations.

- **Memory cost.** To reduce the memory used, we must eliminate items from the queues. We follow a straightforward approach which drops all the queues whenever the DBMS faces a memory shortage and do a naive reexecution of the query, thus trading I/O cost for a reduced memory footprint. A general algorithm to consolidate individual items under their parent node is too expensive since it must examine large portions of the queues.

5 Experimental Evaluation

We conducted an extensive empirical study to 1) evaluate the multipoint approach to answering multipoint *k*-NN queries and compare it to the multiple expansion approach, and 2) evaluate the proposed techniques, namely, full reconstruction (FR) and selective reconstruction (SR), to answering refined queries over a single feature and compare them to the naive approach. We conducted our experiments on real-life multimedia data sets. The major findings of our study are:

- **Efficiency of multipoint approach.** The *k*-NN search based on our multipoint approach is more efficient than the multiple expansion approach. The cost of the latter increases linearly with the number of points in the multipoint query while that of the former is independent of the number of query points.
- **Speedup obtained for refined queries.** The FR and SR approaches speed up the execution of refined queries by almost two orders of magnitude over the naive approach. As expected, the SR approach is the most efficient among all approaches. The sequential scan is significantly slower than any of the index based approaches.

Thus, our experimental results validate the thesis of this paper that the proposed approaches to evaluating refined queries offer significant speedups over the naive approach. All experiments reported in this section were conducted on a Sun Ultra Enterprise 450 with 1GB of physical memory and several GB of secondary storage, running Solaris 2.7.

5.1 Experimental Methodology

We conducted our experiments for single feature queries on the **COLHIST** data set comprised of 4×4 color histograms extracted from 70,000 color images obtained from the Corel Database (obtained from http://corel.digitalriver.com) [2], [18]. We use the Hybrid tree as the F-index for the 16-dimensional color histograms [2]. We chose the hybrid tree since 1) it is a feature-based index structure (necessary to support arbitrary distance functions) (see Section 3) and 2) it scales well to high dimensionalities and large-sized databases [2]. We choose a point Q_C randomly from the data set and construct a graded set of its top 50 neighbors (based on L_1 distance),[9] i.e., the top 10 answers have the highest grades, the next 10 have slightly lower grades, etc. We refer to this set the relevant set $Rel(Q_C)$ of Q_C [18]. We construct the starting query by slightly disturbing Q_C (i.e., by choosing a point close to Q_C) and request for the top 100 answers. We refer to the set of answers returned as the retrieved set $Ret(Q_C)$. We obtain the refined query Q_C^{new} by taking the graded intersection of $Ret(Q_C)$ and $Rel(Q_C)$, i.e., if an object O in $Ret(Q_C)$ is present in $Rel(Q_C)$, it is added to the multipoint query Q_C^{new} with its grade in $Rel(Q_C)$. The goal here is to get $Ret(Q_C)$ as close as possible to $Rel(Q_C)$ over a small number of refinement iterations. The intrafeature weights were calculated using the techniques described in [18]. In all the experiments, we perform five feedback iterations in addition to the starting query (counted as iteration 0). All the measurements averaged more than 100 queries. In our experiments, we fix the hybrid tree page size to 4KB (resulting in a tree with 2,766 nodes).

9. We use L_1 metric (i.e., Manhattan distance) as the distance function $\mathcal{D}_Q$ for the color feature since it corresponds to the histogram intersection similarity measure, the most commonly used similarity measure for color histograms [16], [17].

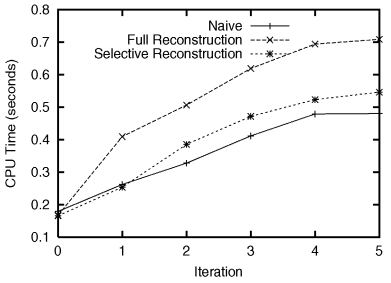

Fig. 9. CPU time of full and selective reconstruction approaches for QEX.

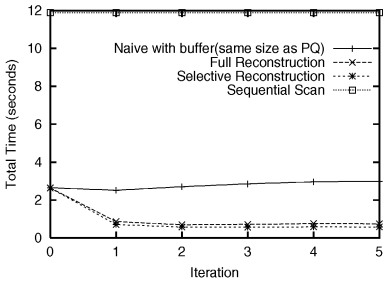

Fig. 11. Response time of naive, full, and selective reconstruction approaches for QEX.

5.2 Multipoint Query Results

We compare the multipoint approach to evaluating single feature k-NN queries to the multiple expansion approach proposed in FALCON [28] and MARS [20]. Fig. 7 compares the cost of the two approaches in terms of the number of disk accesses required to return the top 100 answers (no buffering used). The I/O cost of the multipoint approach is almost independent of the number of points in the multipoint query while that of the multiple expansion approach increases linearly with the number of query points. The reason is that, since the multiple expansion approach explores the neighborhood of each query point individually, it needs to see more and more unnecessary neighbors as the number of query points increases.

5.3 Query Expansion Results

We first present the results for the QEX model. Fig. 8 compares the I/O cost of the naive and reconstruction approaches for the QEX model. In this experiment, the size of cached priority queue varies between 512KB and 640KB (across the iterations). We first run the naive approach with no buffer. We also run the naive approach with an LRU buffer for three different buffer sizes: 540KB, 1080KB, and 2160KB, i.e., they would hold 135, 270, and 540 of the most recently accessed nodes (4KB each) of the F-index (which has 2,766 nodes), respectively. Note that the above three buffers use approximately the same amount of memory as the reconstruction approach (to keep the priority queue cached in memory), twice as much and four times as much, respectively. We also implemented a type of session aware cache we call the *Reuse* approach, which is a buffer organized as the priority queue itself. When a new iteration starts, we push back all the returned items into the queue without any modification, this has the effect of caching all the information seen so far. Note that, since at each iteration we start with all items seen in the previous iteration in the queue, this algorithm is not I/O optimal as it may explore nodes again with the new distance function. This buffer is then traversed based on a LBD, as in the SR approach, we used the optimal LBD for this experiment by exploring the entire queue to reflect the best possible use of this buffering approach. We implemented this session aware buffering approach to show that, even with intelligent buffering, our approach is superior.

The reconstruction approach (with no additional buffer besides the priority queue cache) significantly outperforms the naive approach, even when the latter uses much larger buffer sizes (up to four times more). While the reconstruction approach is I/O optimal (i.e., accesses a node from the disk at most once during the execution of a query), the naive buffer approach, given the same amount of memory, needs to access the same nodes multiple times from disk across the refinement iterations of the query. In more realistic environments where multiple query sessions belonging to different users run concurrently and users have "think time" between iterations of feedback during a session, we expect the buffer hit rate to drop even further, causing the naive approach to perform even more repeated disk accesses and making our approach of per-query caching even more attractive [8]. The reuse approach performs better than any of the LRU-based approaches after the second iteration (remember that the reuse approach is not IO optimal), but worse than our reconstruction approach. Even after five iterations, the reuse approach performs roughly three times as many I/Os than our reconstruction approach. Fig. 9 compares the CPU cost of the FR and SR approaches for the QEX model. The SR approach significantly outperforms the FR approach in terms of CPU time. Fig. 10 compares the number of distance computations performed by the FR approach, the SR

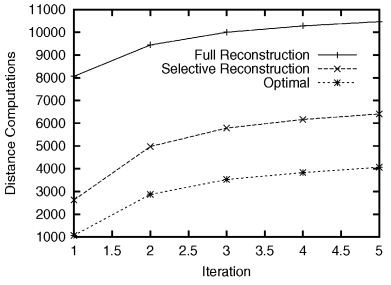

Fig. 10. Distance computations saved by stopping condition in SR approach.

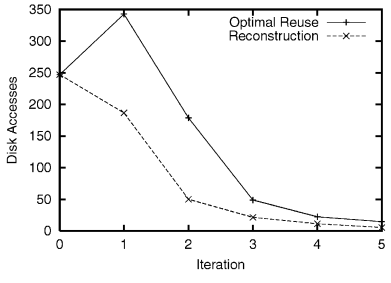

Fig. 12. I/O cost of reuse and reconstruction approaches for QPM.

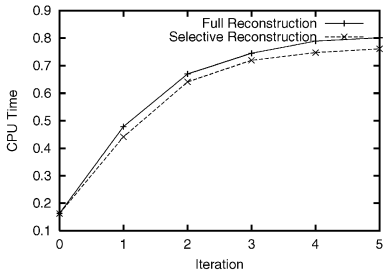

Fig. 13. CPU time for full and selective reconstruction approaches for QPM.

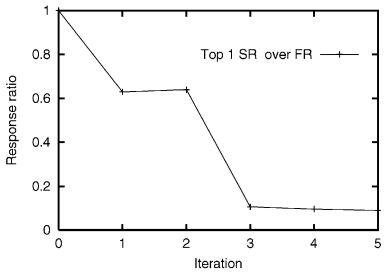

Fig. 15. Response ratio SR/FR for the top 1 item in a KNN query.

approach with the proposed stopping condition, and the SR approach with the optimal stopping condition (range P2 in Fig. 4). The proposed stopping condition saves more than 50 percent of the distance computations performed by FR, while the optimal stopping condition would have saved about 66 percent of the distance computations. This shows that the proposed stopping condition is quite close to the optimal one. Fig. 11 compares the average response time (sum of I/O wait time and CPU time) of a query for the naive (with buffer), FR, SR, and sequential scan approaches (assuming that the wait time of a random disk access is 10ms and that of a sequential disk access is 1ms [9]). The SR technique outperforms all the other techniques; it outperforms the naive approach by almost an order of magnitude over the naive approach and the sequential scan approach by almost two orders of magnitude.

5.4 Query Point Movement Results

Fig. 12 compares the I/O cost of the our reconstruction approach and the same reuse buffering approach with optimal LBD used in Section 5.3 (no additional buffer in either case) for the QPM model. Again, the I/O optimal reconstruction approach is far better compared to the reuse approach. Fig. 13 compares the SR and FR approaches with respect to the CPU cost. Unlike in the QEX approach where the SR approach is significantly better than the FR approach, in QPM the former is only marginally better. The reason is that, unlike in the QEX approach where the savings in distance computations far outweighs the cost of "push back," the savings only marginally outweighs the extra cost in the case of QPM. This is evident in Fig. 14 which shows that the the number of distance computations performed by the SR approach is much closer to that performed by the FR approach as compared to the QEX model. Even the optimal stopping condition would not save as many distance computations as the QEX model. This is due to dimension reweighting that causes the stopping condition to become more conservative (by introducing the factor K in Lemma 3) resulting in less savings in distance computations. However, with a more efficient "push back" strategy, we expect the SR approach to be significantly better than the FR approach in this case as well.

SR improves response time over FR for the top few matches. In an interactive user environment, it is important to consider not only the time required to compute the top k answers, but how soon they can be produced. Fig. 15 compares FR and SR for the QPM model in terms of the relative response time for the first answer out of top 100 queries. In FR, while the queue is transferred, no answers are produced. By contrast, SR will produce the first answer as soon as it is possible, therefore producing quicker the answers users see first. The figure plots the fraction of time SR needs to produce the top answer relative to FR which is always 100 percent.

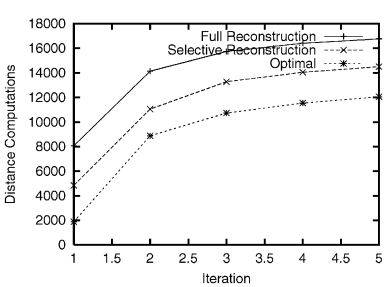

Fig. 14. Distance computations saved by stopping condition for QPM.

6 Conclusion

Top-k selection queries are becoming common in many modern-day database applications like multimedia retrieval and e-commerce applications. In such queries, the user specifies target values for certain attributes and expects the "top k" objects that best match the specified values. Due to the user subjectivity involved in such queries, the answers returned by the system often do not satisfy the user's need right away. In such cases, the system allows the user to refine the query and get back a better set of answers. Despite much research on query refinement models, there is no work that we are aware of on supporting refinement of top-k queries efficiently in a database system. Done naively, each "refined" query is treated as a "starting" query and evaluated from scratch. We propose several techniques for evaluating refined queries *efficiently*. Our techniques save most of the execution cost of the refined queries by appropriately exploiting the information generated during the previous iterations of the query. Our experiments show that the proposed techniques provide significant improvement over the naive approach in terms of the execution cost of refined queries.

Acknowledgments

This work was supported in part by US National Science Foundation grants CCR 0220069, IIS 0083489, IIS 0086124, and CAREER award IIS-9734300, and under Army Research Laboratory Cooperative Agreement DAAL01-96-2-0003.

REFERENCES

[1] M. Carey and D. Kossmann, "On Saying "Enough Already" in Sql," *Proc. SIGMOD,* 1997.

[2] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces," *Proc. IEEE Int'l Conf. Data Eng.,* Mar. 1999.

[3] K. Chakrabarti, K. Porkaew, and S. Mehrotra, "Efficient Query Refinement for Top-k Selection Queries," Technical Report MARS-TR-00-05, 2000.

[4] K. Chakrabarti, K. Porkaew, and S. Mehrotra, "Efficient Query Refinement in Multimedia Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* 2000.

[5] S. Chaudhari and L. Gravano, "Evaluating Top-k Selection Queries," *Proc. Very Large Data Bases Conf.,* 1999.

[6] R. Fagin, "Combining Fuzzy Information from Multiple Systems," *Proc. 15th ACM Symp. Principles of Database Systems (PODS),* 1996.

[7] R. Fagin, "Fuzzy Queries in Multimedia Database Systems," *Proc. Symp. Principles of Database Systems (PODS),* 1998.

[8] J. Gray and A. Reuter, *Transactions Processing: Concepts and Techniques.* San Mateo, Calif.: Morgan Kaufmann, 1993.

[9] J. Gray and P. Shenoy, "Rules of Thumb in Data Engineering," http://www.research.microsoft.com/gray, 1999.

[10] G.R. Hjaltason and H. Samet, "Ranking in Spatial Databases," *Proc. Int'l Conf. Stochastic Structural Dynamics (SSD),* 1995.

[11] Y. Ishikawa, R. Subramanya, and C. Faloutsos, "Mindreader: Querying Databases through Multiple Examples," *Proc. Very Large Data Bases Conf.,* 1998.

[12] F. Korn, N. Sidiropoulos, and C. Faloutsos, "Fast Nearest Neighbor Search in Medical Image Databases," *Proc. Very Large Data Bases Conf.,* 1996.

[13] A. Motro, "Vague: A User Interface to Relational Databases that Permits Vague Queries," *ACM Trans. Office Information Systems,* vol. 6, no. 3, July 1988.

[14] S. Nepal and M.V. Ramakrishna, "Query Processing Issues in Image Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* 1999.

[15] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, and P. Yanker, "The QBIC Project: Querying Images by Content Using Color, Texture and Shape," *Proc. SPIE Conf. Storage and Retrieval for Image and Video Databases,* Feb. 1993.

[16] M. Ortega, Y. Rui, K. Chakrabarti, S. Mehrotra, and T. Huang, "Supporting Similarity Queries in Mars," *Proc. ACM Multimedia 1997,* 1997.

[17] M. Ortega-Binderberger, Y. Rui, K. Chakrabarti, S. Mehrotra, and T. Huang, "Supporting Ranked Boolean Similarity Queries in Mars," *IEEE Trans. Knowledge and Data Eng.,* vol. 10, no. 6, Nov./Dec. 1998.

[18] K. Porkaew, K. Chakrabarti, and S. Mehrotra, "Query Refinement for Content-Based Multimedia Retrieval in MARS," *Proc. ACM Multimedia Conf.,* 1999.

[19] K. Porkaew, S. Mehrotra, and M. Ortega, "Query Reformulation for Content Based Multimedia Retrieval in MARS," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems,* 1999.

[20] K. Porkaew, S. Mehrotra, M. Ortega, and K. Chakrabarti, "Similarity Search Using Multiple Examples in MARS," *Proc. Int'l Conf. Visual Information Systems,* 1999.

[21] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," *Proc. SIGMOD,* 1995.

[22] Y. Rui, T. Huang, and S. Mehrotra, "Content-Based Image Retrieval with Relevance Feedback in Mars," *Proc. IEEE Int'l Conf. Image Processing,* 1997.

[23] Y. Rui, T. Huang, and S. Mehrotra, "Relevance Feedback Techniques in Interactive Content-Based Image Retrieval," *Proc. IS & T and SPIE Storage and Retrieval of Image and Video Databases,* 1998.

[24] Y. Rui, T. Huang, M. Ortega, and S. Mehrotra, "Relevance Feedback: A Power Tool in Interactive Content-Based Image Retrieval," *IEEE Trans. Circuits and Systems for Video Technology,* Sept. 1998.

[25] T. Seidl and H. Kriegel, "Efficient User-Adaptable Similarity Search in Large Multimedia Databases," *Proc. Very Large Data Bases Conf.,* 1997.

[26] T. Seidl and H. Kriegel, "Optimal Multistep k-Nearest Neighbor Search," *Proc. ACM SIGMOD,* 1998.

[27] D. White and R. Jain, *Algorithms and Strategies for Similarity Retrieval,* 1996.

[28] L. Wu, C. Faloutsos, K. Sycara, and T. Payne, "Falcon: Feedback Adaptive Loop for Content-Based Retrieval," *Proc. Very Large Data Bases Conf.,* 2000.

Kaushik Chakrabarti received the MS (1999) and PhD (2001) degrees from the University of Illinois at Urbana Champaign. He is a researcher in the Data Management, Exploration and Mining Group at Microsoft Research. His research interests include multimedia databases, information retrieval, decision support systems, database mining and exploration, and database systems for Internet applications like E-commerce and XML. He has published more than 30 technical papers in the above areas. His papers have received best paper awards at ACM SIGMOD (2001) and VLDB (2000) conferences. He is a member of the Phi Kappa Phi Honor Society (for having a perfect GPA in graduate school), ACM, ACM SIGMOD, and ACM SIGKDD.

Michael Ortega-Binderberger received the BEng degree in computer engineering from the Instituto Tecnologico Autonomo de Mexico (ITAM), Mexico, in 1994, and the MS and PhD degrees in computer science from the University of Illinois at Urbana Champaign in 1999 and 2002, respectively. Previously, he was a visiting researcher at the University of California, Irvine. Currently, he works at the IBM Silicon Valley Lab. His research interests include multimedia databases, information retrieval, decision support systems, data mining, highly available and reliable computing, and information security. He is a member of the IEEE and the ACM.

Sharad Mehrotra received the PhD degree in computer science from the University of Texas at Austin in 1993. He is currently an associate professor in the Information and Computer Science Department at the University of California, Irvine. He has previously been on the faculty of the Computer Science Department at the University of Illinois at Urbana-Champaign (1994-1998) and worked as a scientist at the Matsushita Information Technology Laboratory (1993-1994). He specializes in the areas of database management and distributed systems and has authored more than 85 research publications in top conferences and journals in various disciplines within these fields. He is the recipient of the US National Science Foundation Career Award, Bill Gear Outstanding junior faculty research award at the University of Illinois at Urbana-Champaign in 1997, and the prestigious ACM SIGMOD best paper award in 2001 for a paper entitled "Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases." He is a member of the ACM and the IEEE.

Kriengkrai Porkaew received the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign, in 1996 and 2000, respectively. He has been working as a lecturer in the School of Information Technology at King Mongkut's University of Technology Thonburi, Thailand, since 2000. His research interests are in the areas of database management, geographical information systems, and multimedia analysis and retrieval.