ICDE Submission Number 329

Evaluating Refined Queries in Top-k Retrieval Systems *

Kaushik Chakrabarti
University of Illinois
kaushikc@cs.uiuc.edu

Kriengkrai Porkaew
University of Illinois
kporkaew@cs.uiuc.edu

Michael Ortega
University of Illinois
miki@acm.org

Sharad Mehrotra
University of California
sharad@ics.uci.edu

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 user's need right away; either because the weights and the distance functions associated with the features do not accurately capture the user's 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 dataset show that the proposed techniques save more than 80% 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.

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 [6] (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 dollars. 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 2 bedroom houses in Irvine or more expensive houses as the top matches to the query.*

*This work was supported by NSF CAREER award IIS-9734300, and in part by the Army Research Laboratory under Cooperative Agreement No. DAAL01-96-2-0003.

An important aspect of top-k queries is user subjectivity [13, 8]. In Example 1, let us assume that there are just 2 features: color and texture. Let $Q = \langle Q_C, Q_T \rangle$ be the query image where $Q_C = (0.2, 0.4)$ and $Q_T = (0.4, 0.5)$ are the color and texture vectors extracted from Q (both feature spaces are 2-dimensional). Let us consider two objects in the database: A and B. Let $A_C = (0.4, 0.5)$ and $A_T = (0.9, 0.9)$ while $B_C = (0.9, 0.3)$ and $B_T = (0.3, 0.4)$. Assuming the distance functions over both the color and texture features to be Euclidean (all weights equal), the distances between Q and A with respect to color and texture are $Euclidean(Q_C, A_C) = 0.15$ and $Euclidean(Q_T, A_T) = 0.45$ respectively; those between Q and B are $Euclidean(Q_C, B_C) = 0.50$ and $Euclidean(Q_T, B_T) = 0.10$ respectively. Which is one is a better match to Q? It depends on the user: if similarity with respect to color is more important than that with respect to texture, A is a better match; otherwise B is better. The same issue arises in Example 2. Let us consider the query $Q = \langle Irvine, 4, 300000 \rangle$ in Example 2 (the first, second and third elements are the location, the number of bedrooms and the price in dollars respectively). Let us consider two objects in the database: $A = \langle Irvine, 3, 400000 \rangle$ and $B = \langle SantaAna, 4, 325000 \rangle$. Which one is a better match to Q? Again, it depends on the user. If she is very particular about the location and is flexible about the price, A is a better match. On the other hand, if she has a limited budget and needs 4 bedrooms, B is better. 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 [7, 3, 18, 6]. 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 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, 7, 8].

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, 3, 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, 3, 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, 8]. 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*. Specifically, we cache the priority queues generated by the k-NN algorithm (at individual feature level) and the merge algorithm (at the multifeature level) during the execution 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

[1]We use the terms attribute and feature interchangeably in this paper.

[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.

that is accessed from disk. More specifically, *our techniques are provably optimal in terms of the number of disk accesses*. Furthermore, our technique does not even need to see the entire priority queue to answer the query i.e. it explores only those items in the cache that are necessary to answer the refined query, thus saving unnecessary CPU cost. 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. In this paper, we restrict ourselves to techniques to evaluate single feature queries. The developed techniques can be easily extended to multifeature queries (see [4] for details). Our experiments on real-life multimedia datasets 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, 3, 28]. Such queries arise when the user submits multiple examples during feedback. Each point in the multipoint query can have a weight associated with it. The distance between a multipoint query Q and an object point P is defined as the aggregation of the distances of the points in Q from P. We first formally define the multipoint query and then develop an efficient k-NN algorithm that executes over the F-index to compute 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 section 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 is continued 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 (i.e. the object model), how the user query is represented (i.e. the query model), how the retrieval takes place (i.e. the retrieval model) and how refinement takes place (i.e. the refinement model).

2.1 How are objects represented?

Let $\mathcal{F}$ be the set of features. For example, in Example 1, $\mathcal{F} = \{color, texture\}$. Each object O in the database is represented as a collection $O = \{O_F | F \in \mathcal{F}\}$ of feature vectors where O_F denotes the feature vector for feature $F \in \mathcal{F}$. How the O_F's are obtained from O (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). We associate a multidimensional feature space R_F, of dimensionality d_F, with each feature $F \in \mathcal{F}$ and view O_F as a point in R_F. In Example 1, we associate a 2-d feature space R_C with color and a 2-d feature space R_T with texture. We can view object A as the point $(0.4, 0.5)$ in R_C and the point $(0.9, 0.9)$ in R_T.

2.2 How are queries represented?

Similar to the object, the query Q is also represented as a collection $Q = \{Q_F | F \in \mathcal{F}\}$ of single feature (SF) queries where Q_F denotes the SF query for feature F. The Q_F's are obtained from Q in the same way as O_F's are extracted from O. However, unlike O that is represented using a single point O_F in each feature space R_F, Q_F can

Symbol	Definition	Symbol	Definition
$\mathcal{F}$	Set of features	F	A feature in $\mathcal{F}$
R_F	Feature space associated with F	d_F	Dimensionality of R_F
O	A database object/record	O_F	Feature vector of O for feature F
Q	User query (multifeature, multipoint)	Q_F	Single-feature (multipoint) query for $F \in \mathcal{F}$
n_F	Number of points in Q_F	$\mathcal{P}_F$	Set of n_F points in Q_F
$P_F^{(i)}$	The ith point in $\mathcal{P}_F$	$\mathcal{W}_F$	Set of n_F weights associated with $\mathcal{P}_F$
$W_F^{(i)}$	ith weight in$\mathcal{W}_F$ (associated with $P_F^{(i)}$)	${\mu_F}^{(j)}$	Weight (intra-feature) for jth dimension of R_F
$\mathcal{D}_F$	Distance function for Q_F	ν_F	Weight (inter-feature) associated with F
QPM	Query Point Movement	$D(Q, O)$	Overall distance between Q and O
QEX	Query Expansion	$D_F(Q_F, O_F)$	Distance between Q and O in F
FR	Full Reconstruction Approach	SR	Selective Reconstruction Approach
LBD	Lower Bounding Distance		

Table 1: Summary of symbols, definitions and acronyms

consist of *multiple* points in R_F. The reason is that during refinement, the user might submit multiple examples to the system as feedback (i.e. those that she considers relevant) leading to multiple points per feature space (cf. 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. how important is each example in capturing her information need (e.g., relevance levels in MARS [23], "goodness scores" in Mindreader [11]). To account for that, 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 $Q_F = \langle n_F, \mathcal{P}_F, \mathcal{W}_F, \mathcal{D}_F \rangle$ for feature F consists of the following information:

- the number n_F of points in Q_F
- a set of n_F points $\mathcal{P}_F = \{{P_F}^{(1)}, ..., {P_F}^{(n_F)}\}$ in the d_F-dimensional feature space R_F
- a set of n_F weights $\mathcal{W}_F = \{{w_F}^{(1)}, ..., {w_F}^{(n_F)}\}$, the ith weight ${w_F}^{(i)}$ being associated with the ith point ${P_F}^{(i)}$ $(1 \geq {w_F}^{(i)} \geq 0, \Sigma_{i=1}^{n_F} {w_F}^{(i)} = 1)$.
- a distance function $\mathcal{D}_F$ which, given two points in R_F, returns the distance between them. We assume $\mathcal{D}_F$ to be a weighted L_p metric i.e. for a given value of p, the distance between two points S and T in R_F is given by[3]

$$\mathcal{D}_F(S, T) = [\Sigma_{j=1}^{d_F} {\mu_F}^{(j)} (|S[j] - T[j]|)^p]^{1/p} \tag{1}$$

  where ${\mu_F}^{(j)}$ denotes the (*intra-feature*) weight associated with the jth dimension of R_F. $(1 \geq {\mu_F}^{(j)} \geq 0, \Sigma_{j=1}^{d_F} {\mu_F}^{(j)} = 1)$. D_F specifies which L_p metric to use (i.e. the value of p) and the values of the intra-feature weights. ■

The choice of $\mathcal{D}_F$ i.e. the choice of the L_p metric and the intra-feature weights captures the user perception within the feature F as shown in Example 3. How the weights/metric are chosen is discussed in Section 2.3.

Example 3 *In Example 1, let us consider the distance between Q and B with respect to the color feature i.e. $\mathcal{D}_{color}(Q_C, B_C)$. With L_2 and equal weights, $\mathcal{D}_{color}(Q_C, B_C)$ is 0.50. If the user weighs the first dimension twice as much as the second, $\mathcal{D}_{color}(Q_C, B_C)$ 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, $\mathcal{D}_{color}(Q_C, B_C)$ 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 intra-feature 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.*

[3]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 to the techniques developed in this paper.

The distance $D_F(Q_F, O_F)$ between the multipoint query Q_F and an object point O_F (in R_F) with respect to feature F is defined as the aggregate of the distances between P and the individual points ${P_F}^{(i)} \in \mathcal{P}_F$ in Q_F. We use weighted sum as the aggregation function but any other function can be used as long as it is weighted and monotonic [7].

$$D_F(Q_F, O_F) = \sum_{i=1}^{n_F} {w_F}^{(i)} D_F({Q_F}^{(i)}, O_F) \tag{2}$$

For example, let the query Q consist of multiple points in the color space. Let $Q_C = \{2, \{(0.2, 0.4), (0.4, 0.1)\}, \{0.7, 0.3\},$ Euclidean (equal weights) $\}$, then,

$$D_{color}(Q_C, B_C) = 0.7 * \sqrt{\frac{(0.2-0.9)^2 + (0.4-0.3)^2}{2}} + 0.3 * \sqrt{\frac{(0.4-0.9)^2 + (0.1-0.3)^2}{2}} = 0.46. \tag{3}$$

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

Once we have determined the individual feature-wise distances $D_F(Q_F, O_F), F \in \mathcal{F}$, we compute the overall (i.e. multifeature) distance $D(Q, O)$ between the query Q and object O by aggregating the feature-wise distances:

$$D(Q, O) = \sum_{F \in \mathcal{F}} \nu_F D_F(Q_F, O_F) \tag{4}$$

where ν_F denotes the (inter-feature) weight associated with feature $F \in \mathcal{F}$ ($\nu_F \geq 0, \Sigma_F \nu_F = 1$). While the intra-feature weights model user perception *within* each feature, the inter-feature weights model user perception *between* features i.e. how important are the features *relative* to one another (to the user). How the weights are obtained for a particular user is discussed in Section 2.3. Note that the perception model (i.e. the weights/distance functions) is represented inside the query since the user perception is specific to a query i.e. it differs from one query to another. The system should return the following set $Ans(Q)$ of objects as the answers to query Q: $Ans(Q) = \{O | \forall O' \in (DB - Ans(Q)), D(Q, O) \leq D(Q, O')$.

Example 4 *In Example 1, let us consider the distance between Q and the objects A and B when the two features are weighted the same i.e. $\nu_C = \nu_T = 0.5$ (assuming Euclidean distance with equal weights for both the features). $D(Q, A) = \frac{Euclidean(Q_C, A_C) + Euclidean(Q_T, A_T)}{2} = 0.30$ and $D(Q, B) = \frac{Euclidean(Q_C, B_C) + Euclidean(Q_T, B_T)}{2} = 0.30$ i.e. A and B are equally good matches. If color is weighed twice as much as texture, $D(Q, A) = \frac{2*0.15+0.45}{3} = 0.25$ and $D(Q, B) = \frac{2*0.50+0.10}{3} = 0.37$ i.e. A is a better match. If texture is weighed twice as much as color, $D(Q, A) = 0.35$ and $D(Q, B) = 0.23$ i.e. B is a better match.*

2.3 How do we learn the weights?

As discussed above, we model the user perception using the intra-feature and inter-feature weights and the distance functions. 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. This technique was proposed by Motro and was used in the VAGUE system [13]. Otherwise, to start, we use an arbitrary set of weights (e.g., all weights equal) which is later modified 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 (e.g., MARS [24, 23, 18], Mindreader [11]), a detailed discussion can be found in Appendix A. For our experiments, we use query point movement (QPM) (which moves the query location to the weighted centroid of the submitted examples) and the query expansion (QEX) (which adds new points to the query) to construct the refined query (cf. Appendix A).

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

```
variable temp : List(sorted);
GetNext(Q)
while (TRUE) do
        for each F ∈ 𝓕
            O_F = SingleFeatureGetNext(Q_F);
            if O ∉ temp
               Access full object to obtain O_F, ∀F ∈ 𝓕
               Insert O, D(Q, O) into temp;
            endif
        for each F ∈ 𝓕 topdist_F=Topdist(Q_F);
        threshold = Σ_{F∈𝓕}(ν_F * topdist_F);
        first = the first object in temp;
        if first.dist ≤ threshold
           Remove first from temp;
           return first;
        endif
enddo
```

Table 2: Multifeature query evaluation.

```
variable queue : MinPriorityQueue;
SingleFeatureGetNext(Q_F)
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_F in top
                queue.push(O_F, D_F(Q_F, O_F));
        else /* top is an index node */
           for each child node N in top
                queue.push(N, MINDIST(Q_F, N));
        endif
enddo
```

Table 3: Single feature query evaluation (The kNN algorithm).

the quality of the answers and the answers progressively 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 interested readers to papers on VAGUE, Mindreader and MARS [13, 23, 11, 18].

2.4 Query Evaluation

We first describe how a 'starting' query is evaluated. One option is to use sequential scan. This technique is prohibitively expensive when the database is large (see Figure 8). The most commonly used approach is to maintain an index Idx_F (called the F-index) for each feature e.g., in Example 1, we would maintain a 2-d index on color and a 2-d index on texture. The dimensionality of Idx_F for feature F is d_F i.e. same as the dimensionality of the feature space. The query processing algorithm is shown in Table 2 [16, 17, 7]. The algorithm relies on a *stream* associated with each feature $F \in \mathcal{F}$ that given a SF query Q_F, incrementally returns, on demand, the next best match O_F to Q_F (i.e. the stream supports the $SingleFeatureGetNext(Q_F)$ operation). The $SingleFeatureGetNext(Q_F)$ operation is implemented using the k-nearest neighbor search (k-NN) algorithm executing on the underlying index Idx_F (proposed in [10, 21]).

The overall (multifeature) algorithm: Assuming that each single feature stream support the SingleFeatureGetNext operation, the multifeature query is executed by incrementally retrieving an object O_F from each stream $F \in \mathcal{F}$, accessing the full object O from the database, computing its overall similarity $D(Q, O)$ and inserting it into a sorted list $temp$ until it is safe to return the first object $first$ in $temp$ (see Table 2). It is safe to return $first$ when we are sure that there exists no unexplored object O' with $D(Q, O') \leq D(Q, first)$ i.e. $D(Q, first) \leq threshold$ (see [7, 14] for proof). Note that the entire process is incremental so that we can support the "give-me-more" facility efficiently i.e. the user can initially ask for a few answers and request for more if she wants. This algorithm, known as A_0, was proposed by Fagin [7, 8]. More efficient merging algorithms (e.g., avoiding the "full object" accesses) have been proposed by Ortega et. al. [16, 17] and, more recently, by Nepal et. al. [14]. We present our techniques in the context of Fagin's algorithm for simplicity of presentation. The techniques developed can be applied to any other merging algorithm. Our refinement techniques can also be applied to the work by Chaudhuri and Gravano that proposes to evaluate top-k queries by translating them into range queries [5, 6].

The k-NN (singlefeature) algorithm: The algorithm is shown in Table 3. It maintains a priority queue that contains index nodes as well as data objects prioritized based on their distance from the SF query Q_F 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 $SingleFeatureGetNext(Q_F)$ 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 point 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. $Topdist(Q_F)$ returns the distance of the top item of the queue from Q_F.

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. To achieve the goal, we need to address the following problems:

- **Multipoint queries and arbitrary distance functions**: The $SingleFeatureGetNext(Q_F)$ operation is performed by executing the k-NN algorithm on the corresponding F-index Idx_F. Traditionally, the k-NN algorithm has been used for (1) single point queries i.e. Q_F is a single point in R_F and Euclidean distance function i.e. D_F is Euclidean (no intra-feature weights) [21, 10]. In a query refinement environment, the above assumptions do not hold. We discuss how we implement $SingleFeatureGetNext(Q_F)$ efficiently when (1) Q_F can a multipoint query and (2) D_F can be any L_p metric and can have arbitrary intra-feature weights.[4]
- **Optimization of refined queries**: We develop techniques to execute refined queries efficiently. We propose them first in the context of single feature queries and then extend them for multifeature queries.

3 k-nearest neighbor algorithm for multipoint queries

The $SingleFeatureGetNext(Q_F)$ function in Table 3 can handle only single point queries i.e. Q_F is a single point in R_F. One way to handle a multipoint query $Q_F = \langle n_F, \mathcal{P}_F, \mathcal{W}_F, \mathcal{D}_F \rangle$ is by incrementally determining the next nearest neighbor $NN_F^{(i)}$ of each point $P_F^{(i)} \in \mathcal{P}_F$ by invoking $SingleFeatureGetNext(P_F^{(i)})$ (for $i = [1, n_F]$) computing its overall distances $D_F(Q_F, NN_F^{(i)})$ (using Equation 2) and storing them in a buffer until we are sure that we have found the next nearest neighbor to Q_F. 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 $SingleFeatureGetNext(Q_F)$ function to be able to handle multipoint queries i.e. it should be able to compute the distances of the nodes and objects directly from the multipoint query Q_F and explore the priority queue based on those distances. The basic algorithm in Table 3 does not change, the only changes are the distance computations. As before, there are 2 types of distance computations: (1) distance of an object to the multipoint query i.e. $D_F(Q_F, O_F)$ (2) distance of a node to the multipoint query (MINDIST) i.e. $MINDIST(Q_F, N)$. The computation of (1) follows directly from Equation 2. So all we need to do is to define the distance $MINDIST(Q_F, N)$ between a multipoint query Q_F 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_F 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 intra-feature weights (see [4] for proof). In a feature-based index, the bounding region (BR) of each node N is always, either explicitly or implicitly, a d_F-dimensional rectangle in the feature space R_F [2].

Definition 2 (MINDIST for Multipoint Queries) Given the bounding rectangle $R_N = \langle L, H \rangle$ of a node N, where $L = \langle l_1, l_2, ..., l_{d_F} \rangle$ and $H = \langle h_1, h_2, ..., h_{d_F} \rangle$ are the two endpoints of the major diagonal of R_N,

[4]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.

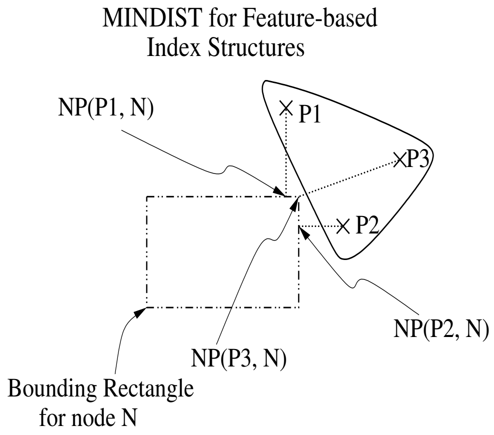

Figure 1: MINDIST Computation.

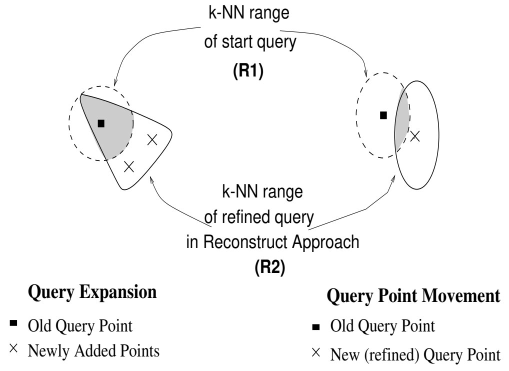

Figure 2: I/O Cost of Naive and Reconstruction Approaches.

$l_i \leq h_i$ for $1 \leq i \leq d_F$. The nearest point $NP(P_F^{(i)}, N)$ in R_N to each point $P_F^{(i)}$ in the multipoint query $Q_F = \langle n, \mathcal{P}_F, \mathcal{W}_F, \mathcal{D}_F \rangle$ is defined as follows (explained in Figure 1).

$$NP(P_F^{(i)}, N)[j] = \begin{cases} l_j & \text{if } P_F^{(i)}[j] < l_j \\ h_j & \text{if } P_F^{(i)}[j] > h_j \\ P_F^{(i)}[j] & \text{otherwise} \end{cases} \tag{5}$$

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

$$MINDIST(Q_F, N) = \sum_{i=1}^{n_F} w_F^{(i)} \mathcal{D}_F(P_F^{(i)}, NP(P_F^{(i)}, N)) \tag{6}$$

■

The $SingleFeatureGetNext$ function can handle arbitrary distance functions D_F (i.e. L_p metrics with arbitrary intra-feature weights). The above algorithm is correct (i.e. $SingleFeatureGetNext(Q_F)$ returns the next nearest neighbor of Q_F) if $MINDIST(Q_F, N)$ always *lower bounds* $D_F(Q_F, T)$ where T is any point stored under N. The correctness of the SingleFeatureGetNext is shown in [4].

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

4 Evaluation of Refined Queries

In this section, we present techniques to evaluate queries involving a single feature. We have extended the above techniques for evaluation of multifeature queries. We omit that discussion in this paper due to space limitations but can be found in the full version of the paper [4]. The rest of this section is devoted to evaluation of single feature refined queries.

A naive way of 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 $SingleFeatureGetNext(Q_F)$ function. In the naive approach, the same nodes of Idx_F may be accessed from scratch by the k-NN algorithm repeatedly iteration after iteration. In other words, a node of Idx_F 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 Figure 2. Region $R1$ represents the iso-distance range corresponding to the distance of the kth NN of the starting query. In other words, 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. In other words, it is the region that needs to be explored to answer the refined 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. To evaluate the refined query, the naive approach[5] 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. An I/O-efficient algorithm would not do that i.e. if a node is accessed from disk once during an iteration of a query, it would be kept in main memory (by caching) and hence would not be accessed from disk again in any subsequent iteration of that query. In Figure 2, 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_F. An algorithm executing the refined query Q_F^{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 $D_F(Q_F^{new}, O) \geq MINDIST(Q_F^{new}, N)$ and (2) N is not accessed during any previous iteration (say Q_F^{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).

In order to achieve I/O efficiency, 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). This will ensure condition (2). 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].[6] In the following subsections, 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 Approach

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 the distance of the item from the refined query Q_F^{new} using the feature values stored, and then pushing it into $queue_{new}$. When all the items have been transferred, the old queue is discarded. We refer to this phase as the *transfer phase*. Subsequently, the multipoint k-NN algorithm proposed in Section 3 is invoked with Q_F^{new} as the query on $queue_{new}$. We refer to this phase as the *explore phase*. The queue gets passed 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

[5] Note that some of the recently accessed nodes (those that overlap with the shaded region) may still be in the database buffer and would not require disk accesses. However, our experiments show that, for reasonable buffer sizes, the naive approach still performs a large number of repeated disk accesses (cf. Figure 5). User think time between iterations also reduces buffer hit ratio.

[6] 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 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.

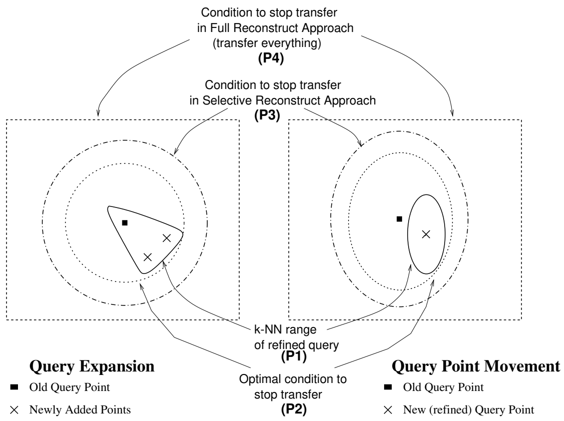

Figure 3: CPU Cost of Full and Selective Reconstruction Approaches

during the transfer phase (one computation each time an item is transferred from $queue_{old}$ to $queue_{new}$) and also during 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 cut down the CPU cost of the transfer phase if we can 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. We can achieve this by imposing a *stopping condition* on the transfer as illustrated in Figure 3. Again, suppose that $SingleFeatureGetNext$ is returning the kth NN of Q_F^{new}. $P1$ is the iso-distance range corresponding to the distance of the kth NN of Q_F^{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_F^{old} totally containing $P1$. So, to return the kth NN of Q_F^{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_F^{new} as the kth NN of Q_F^{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 Figure 3 by region $P4$ i.e. the entire feature space. We next show how to incorporate the above optimization into the FR approach and refer to it as the selective reconstruction approach.

4.2 Selective Reconstruction (SR)

We modify the FR approach to reduce the CPU cost while retaining the I/O optimality. In this approach, we only transfer those items from $queue_{old}$ to $queue_{new}$ that are necessary to ensure correctness. While in the FR approach, the entire transfer phase is followed by the entire explore phase, here the two phases are interspersed. The algorithm is shown in pseudo-code in Table 4. During the transfer phase, we keep transferring items from $queue_{old}$ into $queue_{new}$ until we are sure that there exists no unexplored item in $queue_{old}$ that is closer to Q_F^{new} than the item top_{new} at the top of $queue_{new}$ (i.e. the next item to be explored) from Q_F^{new} i.e. the lower bounding distance $LBD(queue_{old}, Q_F^{new})$ of any unexplored object in $queue_{old}$ from Q_F^{new} is greater than or equal to the distance $top_{new}.distance = D_F(Q_F^{new}, top^{new})$ of top_{new} from Q_F^{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_F^{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_F^{new} and push it back into $queue_{new}$; then we go back to the transfer phase. The I/O optimality of the selective reconstruction technique is shown in [4].

Let us now explain the notion of LBD in the stopping condition (in Line 5). $LBD(queue_{old}, Q_F^{new})$ denotes the lower bounding distance (LBD) of any unexplored object in $queue_{old}$ from Q_F^{new} i.e. $LBD(queue_{old}, Q_F^{new}) \leq D_F^{new}(O, Q_F^{new})$ for all $O \in queue_{old}$. To complete the algorithm, we need to specify how to compute LBD $(queue_{old}, Q_F^{new})$. We need to compute it without incurring *any* extra cost i.e. in a single constant time operation. If we can obtain the optimal $LBD(queue_{old}, Q_F^{new})$ (i.e. there exists an object $O_F \in queue_{old}$ where $LBD(queue_{old}, Q_F^{new}) = D_F^{new}(Q_F^{new}, O_F)$), we can achieve the optimal stopping condition (shown by $P2$ in Figure 3) and hence the optimal CPU cost. The conditions for optimality of the number of distance computations

```
SingleFeatureGetNext(Q_F^{new})
Query Q_F^{old} (of previous iteration), MinPriorityQueue queue_{old} (of previous
iteration), MinPriorityQueue queue_{new} (of current iteration)

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_F^{new}) ≥ top_{new}.distance) break
6            else queue_{old}.Pop();
7                 Recompute top_{old}.distance based on Q_F^{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_F(Q_F^{new}, object));
17      else /* top_{new} is an index node */
18        for each child of top_{new}
19             queue_{new}.push(child, MINDIST(Q_F^{new}, child));
20      endif
21 enddo
```

Table 4: The SingleFeatureGetNext algorithm for refined queries using the selective reconstruction approach.

(i.e. the stopping condition) can be found in [4].

We believe that it is not possible to compute the optimal LBD without exploring the *old* queue (which would defeat the whole purpose of the reconstruction approach). So 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_F^{old}. For simplicity of notation, we define the distance $D_F(Q_F^{old}, Q_F^{new})$ between two multipoint queries as follows. Let $Q_F^{old} = \langle n_F^{old}, \mathcal{P}_F^{old}, \mathcal{W}_F^{old}, \mathcal{D}_F^{old} \rangle$ be the old multipoint query where $\mathcal{P}_F^{old} = \{P_F^{old(1)}, \cdots, P_F^{old(n_F^{old})}\}$ and $\mathcal{W}_F^{old} = \{w_F^{old(1)}, \cdots, w_F^{old(n_F^{old})}\}$. Let $Q_F^{new} = \langle n_F^{new}, \mathcal{P}_F^{new}, \mathcal{W}_F^{new}, \mathcal{D}_F^{new} \rangle$ be the new multipoint query where $\mathcal{P}_F^{new} = \{P_F^{new(1)}, \cdots, P_F^{new(n_F^{new})}\}$ and $\mathcal{W}_F^{new} = \{w_F^{new(1)}, \cdots, w_F^{new(n_F^{new})}\}$. The distance $D_F(Q_F^{old}, Q_F^{new})$ is defined as the weighted all-pairs distance between the constituent points:

$$\mathcal{D}_F(Q_F^{old}, Q_F^{new}) = \sum_{i=1}^{n_F^{old}} \left(w_F^{old(i)} \sum_{j=1}^{n_F^{new}} w_F^{new(j)} \mathcal{D}_F(P_F^{old(i)}, P_F^{new(j)}) \right) \tag{7}$$

Also, let $\mu_F^{old(k)}, k = [1, d_F]$ and $\mu_F^{new}{}_{(k)}, k = [1, d_F]$ denote the old and new intra-feature weights respectively:

$$\mathcal{D}_F^{old}(S,T) = [\Sigma_{j=1}^{d_F} \mu_F^{old(j)} (|S[j] - T[j]|)^p]^{1/p} \tag{8}$$

$$\mathcal{D}_F^{new}(S,T) = [\Sigma_{j=1}^{d_F} \mu_F^{new(j)} (|S[j] - T[j]|)^p]^{1/p} \tag{9}$$

The stopping condition is correct as $(\frac{top_{old}.distance}{K_F} - \mathcal{D}_F(Q_F^{new}, Q_F^{old}))$ lower bounds the distance of any unexplored object in $queue_{old}$ from Q_F^{new} where $K_F = max_{k=1}^{d_F}\left(\frac{\mu_F^{old(k)}}{\mu_F^{new(k)}}\right)$ (proof in [4]).

The $LBD(queue_{old}, Q_F^{new})$ in the stopping condition (SC) in Line 5 of the SR algorithm (cf. Table 4) can now be replaced by the above estimate. Thus, the stopping condition is:

$$(\frac{top_{old}.distance}{K_F} - \mathcal{D}_F^{new}(Q_F^{new}, Q_F^{old})) \geq top_{new}.distance \tag{10}$$

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 (cf. Appendix A). It is also valid irrespective of whether intra-feature reweighting is used or not. If intra-feature reweighting is not used (i.e. all weights are considered equal), K_F turns out to be 1 which means there is no effect of intra-feature reweighting at all. Also note that both K_F and $\mathcal{D}_F(Q_F^{new}, Q_F^{old}))$ are computed just once during the execution of the refined query: 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 Figure 3, 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 (cf. Section 5).

The above discussion of the SR approach has considered how to evaluate the second iteration of the query refinement given the priority queue $queue_{start}$ generated during the execution of the start query. One question remains – how should the future iterations be implemented. The most straightforward approach is to continue future iterations by using the same two priority queues: the queue $queue_{start}$ produced during the execution of the starting query Q_F^{start} and the queue $queue_{curr}$ for the current iteration. During the current iteration, we evaluate the refined query by invoking the SR algorithm (Table 4) with Q_{curr} (new query), Q_{start} (old query), $queue_{curr}$ (new queue) and $queue_{start}$ (old queue) as parameters. When the iteration ends, the remaining items in $queue_{curr}$ are 'pushed back' into $queue_{start}$ to empty up $queue_{curr}$ for use in the next iteration. When the items are transferred from $queue_{start}$ to $queue_{curr}$, their distances from Q_{start} are backed up in the item itself so that we do not have to recompute it while pushing back into $queue_{start}$. The 'push back' still needs to incur the cost of insertions into the priority queue but the hope is that the performance gain obtained by saving distance computations would outweigh the cost of 'push back'. There are several alternate strategies that can be used to significantly cut down and/or eliminate the cost of 'push back' but are more complex. The basic idea is to use more queues, possibly one queue per iteration, so that we do not need to push the items back into the starting queue at the end of the iteration but keep them in the current queue. These strategies are not included here due to space limitations but can be found in the full version of the paper [4]. However, even with this simple 'push back' approach, SR significantly outperforms the FR approach (see Section 5).

Finally, as mentioned before, the above approaches (both FR an SR) make sense only when there is significant overlap between the queries across the refinement iterations (as shown in Figure 2) which is usually the case. If the refined query happens to move far away from the query in the previous iteration, it might be better to follow the naive approach i.e. discard the cached priority queue and execute the query from scratch. Based on the overlap/distance between the 'before' and 'after' queries, we can decide whether to reconstruct or execute from scratch. The details can be found in [4].

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, (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 and (3) evaluate the above techniques to answering refined queries over multiple features. We conducted our experiments on real-life multimedia datasets. 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.

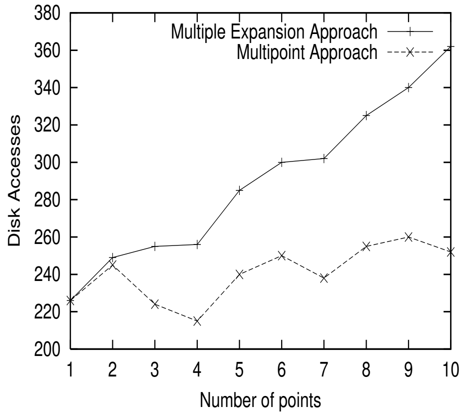

Figure 4: I/O cost of multipoint approach and multiple expansion approach.

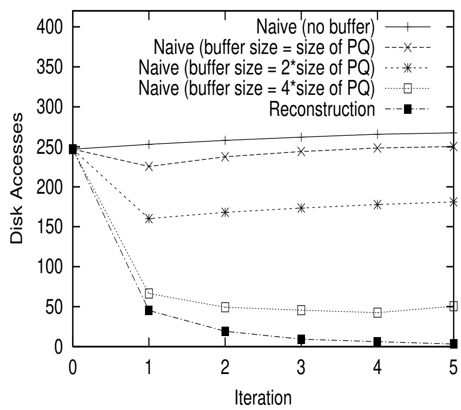

Figure 5: I/O cost of naive and reconstruction approaches for single feature refined queries.

- **Speedup obtained for refined queries over a single feature**: 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.
- **Speedup obtained for refined queries over multiple features**: The FR and SR approaches are significantly more efficient than the naive approach for evaluating refined queries over multiple features, the SR approach being the most efficient technique. The sequential scan is much slower than 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. We only present the experimental results for single feature queries in this paper. The results for multifeature queries can be found in the full paper [4]. 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** dataset comprised of 4x4 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) (cf. Section 3) and (2) It scales well to high dimensionalities and large-sized databases [2]. We choose a point Q_C randomly from the dataset and construct a graded set of its top 50 neighbors (based on L_1 distance)[7] 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 intra-feature weights were calculated using the techniques described in [18]. In all the experiments, we perform 5 feedback iterations in addition to the starting query (counted as iteration 0). All the measurements are averaged over 100 queries. In our experiments, we fix the hybrid tree page size to 4KB (resulting in a tree with 2766 nodes).

5.2 Experimental 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]. Figure 4 compares the cost of the two approaches in terms of the

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

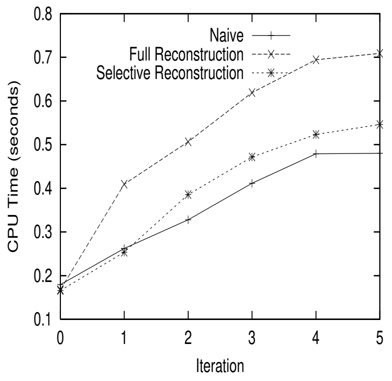

Figure 6: CPU time of full and selective reconstruction approaches for single feature refined queries.

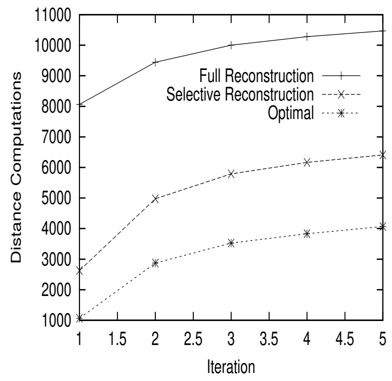

Figure 7: Distance computations saved by stopping condition in SR approach.

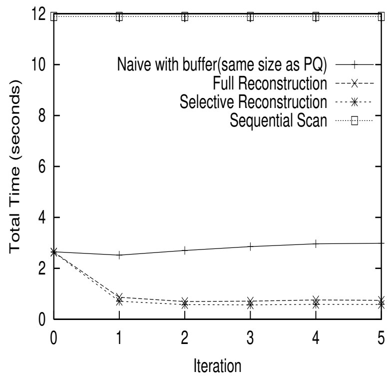

Figure 8: Response time of naive, full and selective reconstruction approaches for single feature refined queries.

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.

We performed several experiments to evaluate the reconstruction approach for processing refined queries for both QEX and QPM models. We first present the results for the QEX model. Figure 5 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 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 2766 nodes) respectively. Note that the above 3 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. 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 4 times more). In other words, given the same amounts of memory, caching the priority queue provides much better savings in the I/O cost than caching disk pages. Figure 6 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. Figure 7 compares the number of distance computations performed by the FR approach, the SR approach with the proposed stopping condition and the SR approach with the optimal stopping condition (range P2 in Figure 3). The proposed stopping condition saves more than 50 % of the distance computations performed by FR while optimal stopping condition would have saved about 66 % of the distance computations. This shows that the proposed stopping condition is quite close to the optimal one. Figure 8 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. We obtained similar results for the QPM model; we are not presenting them here but can be found in the full version of the paper [4].

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 to evaluate 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.

References

[1] M. Carey and D. Kossmann. On saying ”enough already” in sql. *Proc. of SIGMOD*, 1997.

[2] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. *Proceedings of the IEEE International Conference on Data Engineering*, March 1999.

[3] K. Chakrabarti, K. Porkaew, and S. Mehrotra. Efficient query refinement in multimedia databases. *Proceedings of International Conference in Data Engineering(ICDE)*, 2000.

[4] K. Chakrabarti, K. Porkaew, M. Ortega-Binderberger, and S. Mehrotra. Efficient query refinement for top-k selection queries. *Technical Report, MARS-TR-00-05*, 2000.

[5] S. Chaudhari and L. Gravano. Optimizing queries over multimedia repositories. *Proc. of SIGMOD*, 1996.

[6] S. Chaudhari and L. Gravano. Evaluating top-k selection queries. *Proceedings of VLDB Conference*, 1999.

[7] R. Fagin. Combining fuzzy information from multiple systems. *Proc. of the 15th ACM Symp. on PODS*, 1996.

[8] R. Fagin. Fuzzy queries in multimedia database systems. *Proceedings of PODS*, 1998.

[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. *Proceedings of SSD*, 1995.

[11] Y. Ishikawa, R. Subramanya, and C. Faloutsos. Mindreader: Querying databases through multiple examples. *Proc. of VLDB*, 1998.

[12] F. Korn, N. Sidiropoulos, and C. Faloutsos. Fast nearest neighbor search in medical image databases. *Proc. of VLDB*, 1996.

[13] A. Motro. Vague: A user interface to relational databases that permits vague queries. *ACM Transactions on Office Information Systems, Vol.6, No. 3*, July 1988.

[14] S. Nepal and M. V. Ramakrishna. Query processing issues in image databases. *Proc. of 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. In *Proc. of SPIE Conference on Storage and Retrieval for Image and Video Databases*, February 1993.

[16] M. Ortega, Y. Rui, K. Chakrabarti, S. Mehrotra, and T. Huang. Supporting similarity queries in mars. *Proc. of 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 Transactions on Knowledge and Data Engineering (TKDE)*, November 1998.

[18] K. Porkaew, K. Chakrabarti, and S. Mehrotra. Query refinement for content-based multimedia retrieval in MARS. *Proceedings of ACM Multimedia Conference*, 1999.

[19] K. Porkaew, S. Mehrotra, and M. Ortega. Query reformulation for content based multimedia retrieval in MARS. *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, 1999.

[20] K. Porkaew, S. Mehrotra, M. Ortega, and K. Chakrabarti. Similarity search using multiple examples in MARS. *Proceedings of International Conference on Visual Information Systems*, 1999.

[21] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *Proceedings of SIGMOD*, 1995.

[22] Y. Rui, T. Huang, and S. Mehrotra. Content-based image retrieval with relevance feedback in mars. *Proc. of IEEE Int. Conf. on Image Processing*, 1997.

[23] Y. Rui, T. Huang, and S. Mehrotra. Relevance feedback techniques in interactive content-based image retrieval. *Proc. of 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 Tran on Circuits and Systems for Video Technology*, September, 1998.

[25] T. Seidl and H. Kriegel. Efficient user-adaptable similarity search in large multimedia databases. *Proc. of VLDB*, 1997.

[26] T. Seidl and H. Kriegel. Optimal multistep k-nearest neighbor search. *Proc. of 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. *Proceedings of VLDB Conference*, 2000.

The material in this appendix can be read at the discretion of the reviewer and is included as background.

A Refining the query based on feedback

Based on the user feedback, the query is refined using the following two steps so as to better capture the user's information need:[8]

- **Query Modification** modifies the location of the query point(s) $\mathcal{P}_F$ in each feature space $F \in \mathcal{F}$ as well as their weights $\mathcal{W}_F$. If the query interface is the explicit attribute value specification interface (e.g., Example 2), the user can modify the attribute values explicitly (e.g., modify query from $\langle Irvine, 4, 300,000\rangle$ to $\langle Irvine, 3, 350,000\rangle$). 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 $\mathcal{P}_F$ and their weights $\mathcal{W}_F$ in each feature space $F \in \mathcal{F}$. 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 in each feature space F: the weighted centroid (see Figure 9). Effectively, for each F, the query point is moved towards the relevant objects. For example, if A and B are the two relevant examples submitted by the user (cf. Section 1) and A is twice as relevant as B (derived from the relevance levels specified by the user [18]), the SF refined queries for color and texture are $Q_C = \langle\{1, (\frac{2*0.4+0.9}{3}, \frac{2*0.5+0.3}{3}), 1, D_C\}$ and $Q_T = \langle\{1, (\frac{2*0.9+0.3}{3}, \frac{2*0.9+0.4}{3}), 1, D_T\}$ where D_C and D_T (i.e. the $\mu_C^{(j)}$'s and $\mu_T^{(j)}$'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 $\mathcal{P}_F$ (called *representatives* [18]) in each feature space $F \in \mathcal{F}$ giving rise to *multipoint* queries (see Figure 9). The weight ${w_F}^{(i)}$ of any representative $P_F^{(i)}$ in the multipoint query is proportional to the total weight of all the objects $P_F^{(i)}$ represents. For example, if A and B are the two relevant examples submitted by the user and A is twice as relevant as B, one possible choice of representatives are the points themselves i.e. $Q_C = \langle 2, \{(0.4, 0.5), (0.9, 0.3)\}, \{0.67, 0.33\}, D_C\rangle$ and $Q_C = \langle 2, \{(0.9, 0.9), (0.3, 0.4)\}, \{0.67, 0.33\}, D_C\rangle$. There are other possible choices of representatives (see [18] for the representative selection technique). This model was proposed in MARS [18, 19, 20].
- **Reweighting** adjusts the inter-feature weights (the ν_F's in Equation 4) and the intra-feature weights (i.e. the μ_{F_j}'s in Equation 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].

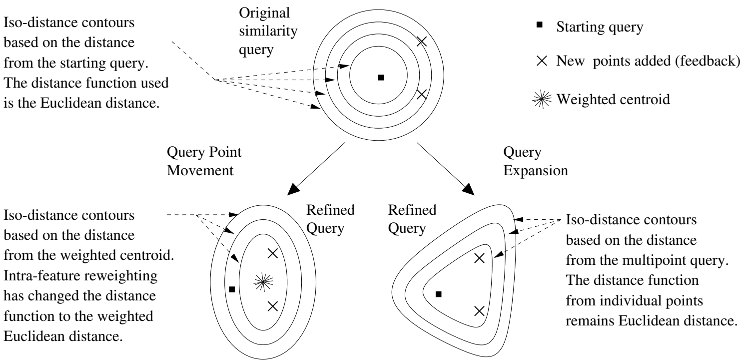

Figure 9: Query Refinement Models

[8]Besides these 2 steps, some researchers have considered cross-correlation among the dimensions in R_F [11]. Although we do not consider cross-correlation in this paper, our techniques are applicable even when it is considered.