Query Refinement in Similarity Retrieval Systems *

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

Michael Ortega
University of Illinois
miki@acm.org

Kriengkrai Porkaew
King Mongkut's University
porkaew@it.kmutt.ac.th

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 the 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 similarity-based retrieval, the answers returned by the system to a user query often do not satisfy the user's information 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. The most commonly used technique to overcome this problem is query refinement. In this technique, the user provides to the system some feedback on the "relevance" of the answers to the user's query. The system then analyzes the feedback, refines the query (i.e., modifies the weights, distance functions, target values etc.) evaluates it and returns the new results. In this paper, we provide an overview of the techniques used to construct the refined query based on the feedback from the user as well as the techniques to evaluate the refined query efficiently. We present experimental results demonstrating the effectiveness of the techniques discussed in the paper.

1 Introduction

Similarity-based retrieval is becoming common in many modern-day database applications. Unlike in a traditional relational database system (RDBMS) where a selection query consists of one or more precise selection conditions and the user expects to get back the exact set of objects that satisfy those conditions, in similarity 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, similarity queries arise naturally in a variety of today's applications.

Example 1 (Multimedia Databases): Consider a content-based image retrieval system [13, 24, 15]. Each image is represented using visual features like color, texture, layout and shape [14]. The similarity between any

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was partially supported by the NSF under grant numbers IIS 0083489, ITR IIS 0086124 and CAREER award IIS-9734300.

two images 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 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.

An important aspect of similarity queries is user subjectivity [11, 7]. 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 (i.e., images) 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 (DFs) over both the color and texture features to be Euclidean (all dimensions weighted equally), 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 to the user 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 (i.e., houses) 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 which is captured by the system using weights and DFs. [6, 3, 16, 5]. At the time of the query, the system *acquires* information from the user based on which it determines the weights and DFs that best capture the perception of this particular user. Based on these weights and DFs, it then retrieves the top answers by first executing a k nearest neighbor (k-NN) algorithm on each individual feature[1] and then "merging" them to get the overall answers [15, 12, 6, 7].

Due to the subjective nature of similarity queries, the answers returned by the system to a user query often do not satisfy the user's need right away [16, 3, 20, 9]. One reason is that the starting weights and DFs may not accurately capture the perception of the user. This happens when the user does not provide enough information to the system initially, either because it is too burdensome for her to do so or because she does not have a clear mental model of her exact information need at the beginning of the query. Another reason for unsatisfactory answers is that the starting examples/target values may not be the best ones to capture the information need (IN) of the user. The most commonly used technique to overcome this problem is query refinement. In this technique, the user provides to the system some feedback on the relevance of answers returned, i.e., she specifies which answers are relevant to her query and which are not [19, 20, 16, 3, 9]. [2] The system then analyzes the feedback,

[1] 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.

[2] An alternate way to provide feedback is to explicitly modify the weights and/or the DFs so as to better capture her perception of similarity [11, 7]. This requires the user to have more domain expertise compared to the example-based feedback; the techniques discussed in this paper apply to this type of feedback as well.

refines the query (i.e., modifies the weights, DFs, target values etc.), evaluates it and returns the new results. The user can continue refining the query over many iterations until she is fully satisfied with the results.

In this paper, we provide an overview of the techniques used to construct the refined query based on the feedback from the user. This includes computing the new query points/values for each feature, the new intra-feature DFs and the new inter-feature weights. We also discuss techniques to evaluate the refined query efficiently. We present experimental results demonstrating (1) the effectiveness of the query refinement techniques in terms of improvement in the the quality of answers returned by the system and (2) the efficiency of the techniques for evaluating refined queries.

2 Query Refinement Techniques

In a similarity retrieval system, the user poses a query Q by providing target values of one or more features and specifying the number k of matches desired. [3] The target values can be either explicitly specified by the user (as in Example 2) or extracted from the examples 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. Subsequent matches can be retrieved incrementally. If the user is not satisfied with the answers, she provides feedback to the system. Based on the feedback, the system modifies the query representation to better suit the user's information need. The 'refined' query is then evaluated and top results are returned with additional matches being retrievable incrementally. The feedback process can be continued for several iterations until the user is fully satisfied. In this section, we first discuss the representation of database objects (object model) and user queries (query model) and formally define the notion of similarity between the query and objects. We then present the techniques used to construct the refined query based on user feedback.

2.1 Representation of Objects

Let $\mathcal{F}$ be the set of features. For example, in Example 1, $\mathcal{F} = \{\rfloor\wr\updownarrow\wr\nabla, \sqcup\rceil\S\sqcup\sqcap\nabla\rceil\}$. Each object O in the database can be viewed as a collection $O = \{O_F | F \in \mathcal{F}\}$ of feature vectors where O_F denotes O's value (a vector) of feature 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 then view the object A mentioned in Section 1 as the point $(0.4, 0.5)$ in R_C and the point $(0.9, 0.9)$ in R_T.

2.2 Representation of Queries

Similar to the object, a query Q (either starting or refined query) 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 may consist of *multiple* points in R_F. The reason is that during refinement, the user might submit multiple examples to the system as feedback (those that she considers relevant) leading to multiple points per feature space (cf. Section 2.4). We refer to such queries as *multipoint queries*. The user may also specify the relative importance of the submitted examples, i.e., how important each example is in capturing her information need. To account for that, we associate a weight with each point of the multipoint query. We now formally define a multipoint query.

[3] We restrict our discussion to selection queries in this paper.

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 $\mathcal{P_F} = \{\mathcal{P_F}^{(\infty)}, ..., \mathcal{P_F}^{(\backslash_\mathcal{F})}\}$ in the d_F-dimensional feature space R_F
- a set of n_F weights $\mathcal{W_F} = \{\sqsupset_\mathcal{F}^{(\infty)}, ..., \sqsupset_\mathcal{F}^{(\backslash_\mathcal{F})}\}$, 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[4]

$$\mathcal{D_F}(\mathcal{S}, \mathcal{T}) = [\pm_{|=\infty}^{\lceil_\mathcal{F}} \mu_\mathcal{F}{}^{(|)}(|\mathcal{S}[|] - \mathcal{T}[|]|)^{\checkmark}]^{\infty/\checkmark} \quad (1)$$

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.

Example 3: In Example 1, let us consider the distance between Q and B with respect to the color feature, i.e., $\mathcal{D}_{\rceil\wr\updownarrow\wr\bigtriangledown}(\mathcal{Q_C}, \mathcal{B_C})$. With L_2 and equal weights, $\mathcal{D}_{\rceil\wr\updownarrow\wr\bigtriangledown}(\mathcal{Q_C}, \mathcal{B_C})$ is 0.50. If the user weighs the first dimension twice as much as the second, $\mathcal{D}_{\rceil\wr\updownarrow\wr\bigtriangledown}(\mathcal{Q_C}, \mathcal{B_C})$ 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, $\mathcal{D}_{\rceil\wr\updownarrow\wr\bigtriangledown}(\mathcal{Q_C}, \mathcal{B_C})$ 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 constitutes a close match in terms of color and what is not. How the weights are chosen for a user is discussed in Section 2.4.

2.3 Definition of Similarity between Query and Objects

We define similarity using distance functions, similarity being inversely proportional to distance. We start with the DFs for individual features followed by the overall multifeature DF. 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 O_F 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 [6].

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

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,

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

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

[4]Note that this assumption is general since most commonly used DF (e.g., Manhattan distance, Euclidean distance, Bounding Box distance) are special cases of the L_p metric. However, this excludes DFs that involve "cross correlation" among dimensions. Handling cross-correlated functions has been addressed in [22] and can be incorporated to the techniques developed in this paper.

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 the features are *relative* to each other. How the weights are obtained for a particular user is discussed in Section 2.4. 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.

2.4 Computing the weights/DFs based on feedback

As discussed above, we model the user perception using the intra-feature and inter-feature weights and the DFs. But how do we obtain the right values of the weights/DFs that best capture the user's perception? One solution is for the user to explicitly specify the weights. This technique was proposed by Motro and was used in the VAGUE system [11]. Otherwise, we start with an arbitrary set of weights (e.g., all weights equal). [5] 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 starting weights and DFs do not accurately capture the user's perception or the query points do not properly capture the user's information need or both. We now discuss techniques to modify, based on user feedback, the query points and the weights/DFs so as to represent the user's information need better.

- **Intra-feature Refinement**: Within each feature, the system modifies the position of the query point(s) $\mathcal{P}_{\mathcal{F}}$ in each feature space $F \in \mathcal{F}$ as well as their weights $\mathcal{W}_{\mathcal{F}}$. The intra-feature weights (i.e. the μ_{Fj}'s in Equation 1) are also modified to better capture the user's perception within each feature. [6] If the query interface is the explicit attribute value specification interface (as in Example 2), the user can modify the attribute values explicitly (e.g., modify query from $\langle Irvine, 4, 300,000 \rangle$ to $\langle Irvine, 3, 350,000 \rangle$). The same is true for intra-feature weights. If the query interface is QBE, the user submits relevant examples based on which the system computes the new query points and weights for each feature space $F \in \mathcal{F}$. Two models have been proposed for the above computation:
  - *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 of the points corresponding to the submitted examples in F (see Figure 1). For example, if A and B are the two relevant examples submitted by the user (cf. Section 1) and A is indicated to be twice as relevant as B, the single query point for the color space is $(\frac{2*0.4+1*0.9}{3}, \frac{2*0.5+1*0.3}{3})$ and that for the texture space is $(\frac{2*0.9+1*0.3}{3}, \frac{2*0.9+1*0.4}{3})$.
    The QPM model computes the intra-feature weights for F (i.e. the μ_{Fj}'s in Equation 1) as follows. Let σ_{Fj} denote the standard deviation of the feature values of the relevant examples for feature F along the

[5] The initial weights can be chosen more intelligently. [1] proposes to collect feedback information provided by different users over time and use it to determine the best initial weights for the particular user. This technique reduces the number of feedback iterations, especially for already-seen queries.

[6] Besides these 2 steps, some researchers have considered cross-correlation among the dimensions in R_F [9].

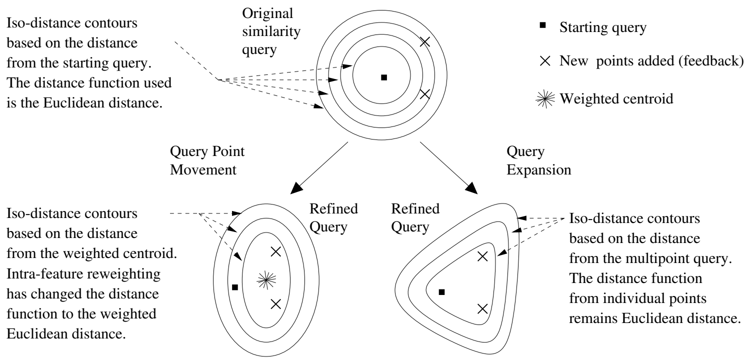

Figure 1: Query Refinement Models

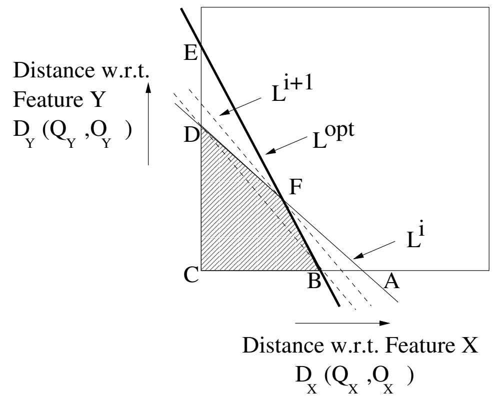

Figure 2: Inter-feature Reweighting

jth dimension. μ_{Fj} is assigned the value $\frac{1}{\sigma_{Fj}}$. The intuition is that among the objects considered relevant by the user, the higher the variance along a dimension, the lower the significance of that dimension.
This approach is used in MARS [19, 21, 20] and also in Mindreader [9]. For each feature space, it chooses a single query point and reweighs the dimensions so that the sum of its distances from the relevant points is minimized [9]. With each iteration of relevance feedback, the query point moves moves towards the relevant points and the distance function is adjusted to fit the distribution of the relevant points.

- *Query Expansion (QEX)*: In this model, the submitted examples are represented by *multiple* points $\mathcal{P}_{\mathcal{F}}$ (called *representatives* [16]) in each feature space $F \in \mathcal{F}$, giving rise to *multipoint* queries (see Figure 1). 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 indicated to be twice as relevant as B, one possible choice of representatives is the points themselves with weights 0.67 and 0.33 respectively for both color and texture spaces. If the user submits many examples, a better approach is to select a small number of good representatives to construct the multipoint query for each feature space. [16] proposes to use a clustering algorithm to cluster the relevant points in each feature space and use the cluster centroids as representatives for that feature. The weight of each representative is chosen to be proportional to the number of relevant objects in the corresponding cluster [16]. The QEX model does not modify the intra-feature weights (i.e. the μ_{Fj}'s in Equation 1) as it adjusts the intra-feature distance function by adding new query points (see Figure 1). This model was proposed in MARS [16, 17, 18].

- **Inter-feature Refinement**: The system modifies the inter-feature weights (the ν_F's in Equation 4) in order to better capture the user's perception across features. If the query interface allows explicit weight specification, the user can modify the weights directly as in Motro's VAGUE system [11]. In a QBE environment, the system automatically derives the inter-feature weights from the relevant examples submitted by the user [17]. A simple approach is to choose ν_F to be proportional to the number of relevant examples in the top t matches with respect to feature F (t is selected by the system) [21]. More the number of top matches of a feature marked relevant by the user, more important is the feature to this user.
A more advanced approach to determine inter-feature weights was proposed in [17]. For simplicity, let us assume that the query Q consists of two features: X and Y. We assume that there exists weights ν_X^{opt} and ν_Y^{opt} such that $\nu_X^{opt} D_X(Q_X, O_X) + \nu_Y^{opt} D_Y(Q_Y, O_Y)$ accurately captures the user's perception of distance between the query Q and an object O (see Equation 4). The user's perception can be visualized as a line L^{opt} defined by the following equation in the distance space defined over features X and Y (illustrated in Figure

2, L^{opt} is the bold line BE):

$$\nu_X^{opt} D_X(Q_X, O_X) + \nu_Y^{opt} D_Y(Q_Y, O_Y) = \delta \tag{5}$$

δ is a threshold such that any object with distance below δ is deemed relevant by the user. The slope of L^{opt} represents the relative importance of the features X and Y.
Let ν_X^i and ν_Y^i denote the weights used by the system for the ith iteration of feedback. The corresponding line L^i is defined by (Line AD in Figure 2):

$$\nu_X^i D_X(Q_X, O_X) + \nu_Y^i D_Y(Q_Y, O_Y) = \delta^i \tag{6}$$

Consider the line BD in Figure 2 joining the intercepts made by lines L^{opt} and L^i with the X and Y axis respectively. It can be shown that the slope of line BD (i.e., $-\frac{CD}{BC}$) lies in between the slope of L^i and L^{opt}. If the weights ν_X^i and ν_Y^i are modified such that slope of the line L^{i+1} corresponding to the $(i+1)$th iteration is the same as that of line BD, the weights will converge to the optimal weights ν_X^{opt} and ν_Y^{opt}. This, however, requires the system to know the slope $-\frac{CD}{BC}$. Since the system does not know L^{opt}, it attempts to estimate the lengths of BC and CD based on the feedback provided by the user. Note that the system retrieves objects below L^i during the ith iteration (i.e., those in triangle ACD). Only those items below L^{opt} (i.e., those in triangle BCE) are considered relevant by the user. So, only those objects in the region BCDF (the shaded region) will be marked relevant by the user. Based on this observation, [17] proposes different strategies to estimate the slope $-\frac{CD}{BC}$. One strategy is to estimate BC by $\Delta X = max_{O \in \mathcal{S}} D_X(Q_X, O_X)$ and CD by $\Delta Y = max_{O \in \mathcal{S}} D_Y(Q_Y, O_Y)$ where $\mathcal{S}$ denotes the set of relevant examples submitted by the user. The slope of line BD is therefore $-\frac{\Delta Y}{\Delta X}$. The weights ν_X^{i+1} and ν_Y^{i+1} are chosen to be

$$\nu_X^{i+1} = \frac{\Delta Y}{\Delta X + \Delta Y} \tag{7}$$

$$\nu_Y^{i+1} = \frac{\Delta X}{\Delta X + \Delta Y} \tag{8}$$

Details of other strategies of selecting intra-feature weight can be found in [17].

3 Evaluation of Refined Queries

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. In this section, we discuss techniques to evaluate refined queries efficiently. A naive way to evaluate a refined query is to treat it just like a starting query and execute it from scratch. We observe that 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 start with a discussion on techniques to evaluate starting queries and then optimize them for refined queries.

3.1 Evaluation of Starting Queries

One option to evaluate a starting query is to use a sequential scan. This technique is prohibitively expensive when the database is large. A commonly used approach is to maintain an index Idx_F (called the F-index) for each feature. For example, in Example 1, we can 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. Subsequently, a query can be evaluated using the algorithm (called the merge algorithm) shown in Table 1 [14, 15, 6]. To compute the next best match to query Q, the merge algorithm incrementally retrieves the next best match O_F to

```
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 1: Multifeature query evaluation (The merge algorithm).

```
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 2: Single feature query evaluation (The kNN algorithm).

Q with respect to each feature $F \in \mathcal{F}$ (by calling $SingleFeatureGetNext(Q_F)$ shown in Table 2), accesses the full object O from the database, computes its overall similarity $D(Q, O)$ to Q and inserts it into a sorted list $temp$ until it is safe to return the first object $first$ in $temp$. 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 [6, 12] for proof).

The $SingleFeatureGetNext(Q_F)$ operation is implemented using the k-nearest neighbor search (k-NN) algorithm executing on the underlying index Idx_F (see Table 2). 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. Initially, the queue contains only the root node. 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 [8]) 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.

3.2 Optimization for Refined Queries

In this section, we present a simple technique to optimize the $SingleFeatureGetNext(Q_F)$ algorithm. The proposed technique can be extended for optimizing the merge algorithm as described in [4].

In the naive approach where a refined query is executed like a starting query, the k-NN algorithm may access the same nodes of Idx_F from disk multiple times during the execution of a query (during different iterations) causing unnecessary disk I/O. This is true even when a buffer is used to keep the most recently used disk pages in memory. We present a simple algorithm (called *full reconstruction (FR)*) that guarantees that a node of the F-index is accessed from the disk at most once throughout the execution of the query across all iterations of refinement. We achieve this by caching 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

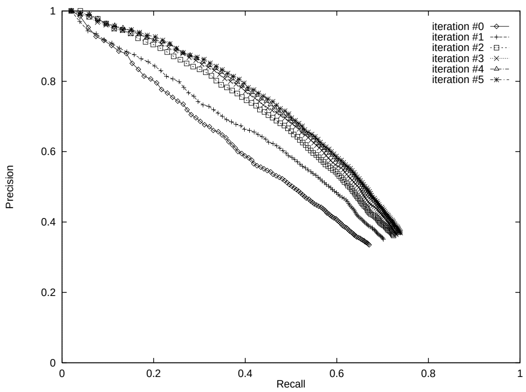

Figure 3: Precision Recall Graph for Query Expansion

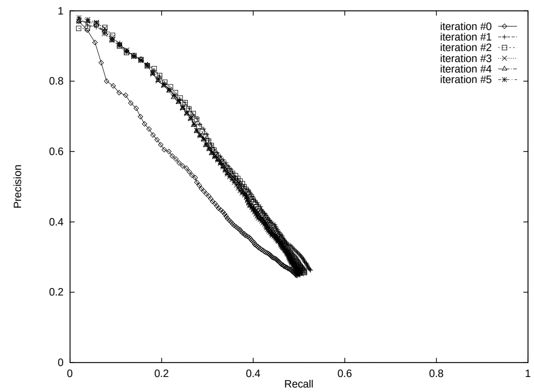

Figure 4: Precision Recall Graph for Query Point Movement

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. We also assume that the entire priority queue fits in memory as is commonly assumed in other works on nearest neighbor retrieval [10, 23].

The FR algorithm works as follows. To evaluate the refined query, it 'reconstructs' 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 k-NN algorithm (discussed in Table 2) is invoked with Q_F^{new} as the query on $queue_{new}$ to retrieve the new answers. 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 their roles (old and new) get swapped from iteration to iteration. Further optimizations of the above algorithm (viz., Selective Reconstruction) can be found in [4].

4 Experimental Evaluation

In this section, we present experimental results demonstrating the effectiveness of the query refinement techniques discussed in Section 2.4, i.e., how significantly do they improve the quality of answers returned by the system. Due to space limitations, we restrict the discussion to intra-feature refinement, experiments on the effectiveness of inter-feature techniques can be found in [17, 21]. [7] We conducted our experiments on the **COLHIST** dataset. It comprises of 4x4 color histograms extracted from 70,000 color images obtained from the Corel Database (available online at `http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.ht` [2, 16]. 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.

The effectiveness of multimedia retrieval depends on the subjective judgement of the user. In order to make our measurements of retrieval effectiveness independent of the user subjectivity, we use system generated ground

[7] We have omitted the experiments showing the speedup achieved for refined queries by the evaluation techniques discussed in Section 3.2, they can be found in [4].

truth for our measurements. We choose a point Q_C randomly from the dataset as the query point and construct a graded set of its top 50 neighbors (based on L_1 distance)[8] We mark the top 10 answers as highly relevant, the next 15 as very relevant and the last 25 answers as relevant. The above set of answers constitutes the ground truth or the *relevant set* $Rel(Q_C)$ for Q_C. 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 simulate user feedback by marking those items in $Ret(Q_C)$ that are also in the top 10 items of $Rel(Q_C)$ as highly relevant, those that are also in the next 15 items of $Rel(Q_C)$ as very relevant and those that are also in the last 25 as relevant. Based on the simulated feedback, we construct the refined query using both intra-feature refinement techniques discussed in 2.4 (QPM and QEX). The query is then evaluated and the top 100 answers are retrieved which forms the new retrieved set. We repeat the above process for 5 feedback iterations (the starting query is counted as iteration 0).

To measure the effectiveness of the two approaches, we compute *precision* and *recall* based on the relevant and retrieved sets:

$$precision = \frac{|relevant \cap retrieved|}{|retrieved|} \tag{9}$$

$$recall = \frac{|relevant \cap retrieved|}{|relevant|} \tag{10}$$

Note that for any query, the relevant set is the same for both approaches but the retrieved sets differ. To construct the precision-recall graph for a query, we need precision values for various values of recall. We achieve this by retrieving one point at a time and calculating the precision and recall for $|retrieved| = 1$, $|retrieved| = 2$, ..., $|retrieved| = 100$. Figures 3 and 4 show the precision-recall graphs for query expansion and query point movement respectively. The measurements are averaged over 100 queries. Figures 3 and 4 show that for both approaches, the retrieval performance improves from one iteration to the next. In general, QEX performs better than QPM as QEX can better adjust the distance function to what the user has in her mind. QPM modifies only the intra-feature weights (μ_F's) but not the function $\mathcal{D}_\mathcal{F}$ itself. QEX, on the other hand, changes the distance function, albeit implicitly, by adding and removing points (as shown by the shape of iso-distance contours in Figure 1) and hence has better adjustment capabilities. Furthermore, QEX can capture local clusters, if present, in the ground truth while QPM, having the single point representation, ignores the cluster information.

5 Conclusion

Similarity queries are becoming common in many modern-day database applications. 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. The most common technique to overcome this problem is query refinement. In this technique, the user provides to the system some feedback on the relevance of the answers returned. The system then analyzes the feedback, refines the query (i.e., modifies the weights, distance functions and query values that capture the user's information need), evaluates it and returns the new results. In this paper, we discuss techniques to effectively refine the query based on user feedback. We also describe algorithms to evaluate refined queries efficiently. Our experiments show that the proposed techniques significantly improve the quality of the answers returned by the system.

[8] We use the L_1 metric (i.e., Manhattan distance) as the distance function $\mathcal{D}_\mathcal{F}$ for the color feature since it corresponds to the histogram intersection similarity measure, the most commonly used similarity measure for color histograms [14, 15].

References

[1] I. Bartolini, P. Ciaccia, and F. Waas. Feedbackbypass: A new approach to interactive similarity query processing. *Proceedings of VLDB Conference*, 2001.

[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, and S. Mehrotra. Evaluating refined queries in top-k retrieval systems. *Technical Report, MARS-TR-00-05*, 2000.

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

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

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

[8] G. R. Hjaltason and H. Samet. Ranking in spatial databases. *Proceedings of SSD*, 1995.

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

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

[11] 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.

[12] S. Nepal and M. V. Ramakrishna. Query processing issues in image databases. *Proc. of ICDE*, 1999.

[13] 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.

[14] M. Ortega, Y. Rui, K. Chakrabarti, S. Mehrotra, and T. Huang. Supporting similarity queries in mars. *Proc. of ACM Multimedia 1997*, 1997.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] T. Seidl and H. Kriegel. Optimal multistep k-nearest neighbor search. *Proc. of ACM SIGMOD*, 1998.

[24] D. White and R. Jain. Algorithms and strategies for similarity retrieval. 1996.