An Approach to Integrating Query Refinement in SQL*

Michael Ortega-Binderberger[1], Kaushik Chakrabarti[1], and Sharad Mehrotra[2]

[1] University of Illinois, Urbana-Champaign, Urbana, IL, 61801, USA
miki@acm.org, kaushikc@uiuc.edu
[2] University of California, Irvine, Irvine, CA, 92697-3425, USA
sharad@ics.uci.edu

Abstract. With the emergence of applications that require content-based similarity retrieval, techniques to support such a retrieval paradigm over database systems have emerged as a critical area of research. 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. Query refinement is used to handle user subjectivity in similarity search systems. This paper explores how to enhance database systems with query refinement for content-based (similarity) searches in object-relational databases. Query refinement is achieved through relevance feedback where the user judges individual result tuples and the system adapts and restructures the query to better reflect the users information need. We present a query refinement framework and an array of strategies for refinement that address different aspects of the problem. Our experiments demonstrate the effectiveness of the query refinement techniques proposed in this paper.

1 Introduction

Similarity retrieval has been recently explored for multimedia retrieval, especially in the context of image databases. Multimedia objects are typically represented using a collection of features such as color, texture, shape, etc. that partially capture the content of the multimedia object. Since features imperfectly represent the content of the multimedia object, when searching for a multimedia object, it is infeasible to insist on retrieving only those objects that exactly match the query features. Instead, a similarity feature matching approach to retrieving multimedia objects makes much more sense. Accordingly, users expect to see the results of their query in ranked order, starting with the results most similar to the query first. Multimedia similarity retrieval has been studied extensively by several researchers and projects including QBIC [10], MARS [16], WebSeek[7], DELOS [13] and HERMES [3] among others. Another field where similarity retrieval has been explored extensively is in Text Information Retrieval [4].

* 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. Michael Ortega-Binderberger is supported in part by CONACYT award 89061.

The initial result ranking computed in response to a query may not adequately capture the users preconceived notion of similarity or *information need* [4], prompting the user to refine the query. The gap between the actual and desired rankings is due to the subjective perception of similarity by users on one hand, and the difficulty to express similarity queries that accurately capture the users information need on the other. This gap may arise due to many reasons, including the users unfamiliarity with the database, difficulty in assigning weights to different similarity predicates, differing priorities, etc. Therefore, in an effort to close this gap, users initiate an information discovery cycleby iteratively posing a query, examining the answers, and re-formulating the query to improve the quality (ranking) of the answers they receive. Research in multimedia [19,12,21,5] and text retrieval [18,4,6] has therefore also focused on refining a query with the help of the users feedback.

This paper advances the notion that similarity retrieval and query refinement are critical not just in the context of multimedia and/or text retrieval, but in general in the context of structured and semi-structured data. Some example applications where such a need arises are:

Example 1 (Job Openings). In a job marketplace application, a listing of job openings (description, salary offered, job location, etc.) and job applications (name, age, resume, home location, etc.) are matched to each other. Resume and job description are text descriptions, location is a two dimensional (latitude, longitude) position, and desired and offered salary are numeric. Applicants and job listings are *joined* with a (similarity) condition to obtain the best matches. Unstated preferences in the initial condition may produce an undesirable ranking. A user then points out to the system a few desirable and/or undesirable examples where job location and the applicants home are close (short commute times desired); the system then modifies the condition and produces a new ranking that emphasizes geographic proximity. ■

Example 2 (Multimedia E-catalog search). Consider an online garment retailer. The store has an online catalog with many attributes for each garment: the manufacturer, price, item description, and garment picture. We can define similarity functions for each of these attributes, e.g., we can define how similar two given prices are and can extract visual features from the garment image [10]. Through a *query by example* (QBE) user interface, users select or enter a desired price, description, and/or select pictures of attractive garments. Users then request the "most similar" garments in terms of those attributes. The initial result ranking may not be satisfactory. The user then expresses her preferences by marking the attributes appropriately, and re-submits the query. The system computes a new set of answers ranked closer to the users desire. ■

We propose to shift as much as possible the burden of refinement from the user to the database management system (DBMS). When users examine answers, they naturally know which ones fit the desired results, instead of initiating the mental process of refinement, users off-load the task to the DBMS by merely judging some of the answers. Judging takes the form of a user interaction with the system where *relevance judgments* about results – which results the user thinks are good examples, and which ones are bad examples – are *fed back* to the DBMS. The DBMS uses these *relevance feedback judgments* to refine the query

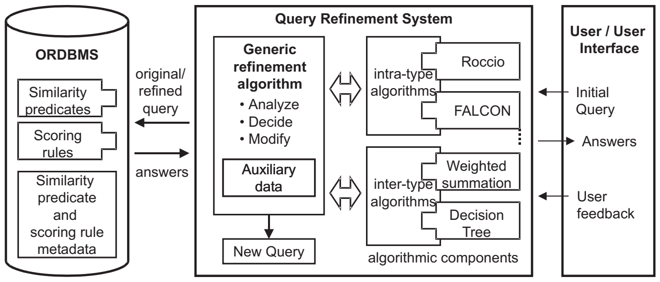


Fig. 1. Query Refinement System Architecture

and re-evaluates[1] it to produce a new ranked answer set which may be subject to more iterations of *query refinement*. The burden on the user is thus reduced to selecting a subset of results as good or bad and informing the DBMS. This interaction can be done directly through SQL, or through a context specific user interface reducing the users burden. This approach mirrors Information Retrieval and Cooperative Databases that involve the user in the query answering loop.

We integrate data-type specific refinement algorithms into a framework that combines multiple such algorithms and provides refinement across them. This two-level approach to query refinement is also shown in figure 1, as intra-, and inter- type refinement modules with specific algorithm plug-ins. Refinement algorithms for specific data-types have been explored in the past, such as Rocchios [18] algorithm for text vector models, MARS and Mindreader algorithms for image retrieval [19,17,12], and FALCON [21] for metric spaces, among others. Our framework uses these and other algorithms as building blocks of a complete system with query refinement support.

The main contribution of this paper is a proposal on how to support query refinement through user relevance feedback in an object-relational DBMS (ORDBMS). Specifically, our contributions are:

- a generic refinement framework, or *meta model*, which accepts specific algorithm “plug-ins”
- DBMS support interfaces and data structures to enable query refinement
- review several domain specific and domain independent query refinement techniques

We implemented our proposed framework as a wrapper on top of the Informix Object Relational DBMS. Figure 1 shows the system architecture. A user interface client connects to our wrapper, sends queries and feedback and gets answers incrementally in order of their relevance to the query. We implemented the similarity and refinement functionality described in this paper and also developed a sample application to validate our approach in addition to our experiments.

[1] Re-evaluation of a refined query may benefit from re-using the work done to answer the original (or previous) query, but is not strictly part of the query refinement process, therefore we do not discuss this topic in this paper. We assume a naive re-evaluation that treats the refined query as an original query.

The rest of this paper is developed as follows. Section 2 presents some background. Section 3 presents our query refinement framework, and section 4 presents some specific refinement algorithms. Section 5 presents some experiments and an example query refinement application. Section 6 presents related work. Section 7 offers some conclusions.

2 Preliminaries — Similarity Retrieval Model

Before we embark on an explanation of query refinement, we first discuss the similarity retrieval model on which we build our query refinement work. Let us consider an example query:

Example 3 (Similarity Query). The following query joins a table of houses with a table of schools. The house table has price and availability attributes, and both tables share a location attribute:

```
select $wsum(p_s, 0.3, l_s, 0.7)$ as $S, a, d$
   from Houses H, Schools S
   where H.available and similar_price(H.price, 100000, “30000”, 0.4, $p_s$)
                     and close_to(H.loc, S.loc, “1, 1” , 0.5, $l_s$)
   order by $S$ desc
```

This query finds homes that are close to schools, are priced around $100,000, and are available. The query has two similarity predicates: *similar_price*, and *close_to*. Both predicates return similarity scores (p_s, l_s) that are combined into a single score S for the whole tuple, here we use the *wsum scoring rule* to combine them. Note that a similarity query contains both precise predicates and similarity predicates. ■

In the context of a similarity query, the following concepts are important: *similarity score, similarity predicate, scoring rule, and ranked retrieval.* Ranked retrieval stems from the intuitive desire of users of similarity based searching systems to see the best answers first [15,4], therefore the answers are sorted, or *ranked* on their overall score S. We now cover the remaining concepts in detail.

A similarity based search in a database is an SQL query with both precise and *similarity predicates.* Precise predicates return only those data objects that exactly satisfy them, i.e., traditional Boolean predicates. Similarity predicates instead specify target values and do not produce exact matches, they return a similarity *score* S for objects that approximately match the predicate. To be compatible with SQL, similarity predicates return Boolean values if the similarity score is higher than a threshold. Data is represented in the object-relational model which supports user-defined types and functions. In this work we focus on select-project-join queries.

A similarity score provides a fine grained differentiation for a comparison between values or objects, i.e., how good or bad they match. This is in contrast to the prevalent precise comparisons that yield Boolean values, i.e., complete match or not. We define a similarity score as follows:

Definition 1 (Similarity score S). *S is a value in the range [0,1]. Higher values indicate higher similarity.* ■

Similarity predicates compare values and compute a similarity score S for the degree to which the values match. For example, a similarity predicate for 2 dimensional geographic locations may say that identical locations are 100% close ($S = 1$), 5 kilometers distance is 50% close ($S = 0.5$) and 10 kilometers and beyond is 0% close ($S = 0$), etc. We define similarity predicates as follows:

Definition 2 (Similarity predicate). *A similarity predicate is a function with four inputs: (1) a value to be compared, (2) a set of query values, (3) a parameter string to configure the similarity predicate, and (4) a similarity cutoff value α in the range [0,1] (also called* alpha cut *or threshold), and returns two values: (1) a Boolean value (true, false), and (2) a similarity score S (as a side effect). The function returns true if $S > \alpha$, else it returns false. The function has the form:*
$Similarity\ Predicate(input\ value, set\ of\ query\ values, parameters, \alpha, S) \rightarrow \{true, false\}$ ■

This function compares an input value to one or more values of the same or a compatible type. We support a set of values since in a Query-by-Example paradigm it often makes sense to compute the similarity of an object with respect to multiple examples rather than just one [17,21]. To offer flexibility to the user, similarity predicates may take additional parameters to configure the predicate and adapt it to a specific situation, for example parameters may include weights that indicate a preferred matching direction between geographic locations, or select between Manhattan and Euclidean distance models. We use a string to pass the parameters as it can easily capture a variable number of numeric and textual values. The similarity score S output parameter is set during the matching. Similarity predicates of this form were used in other similarity searching systems such as VAGUE [15] and others [1].

We further designate some similarity predicates as *joinable*:

Definition 3 (Joinable similarity predicate). *A similarity predicate is joinable iff it is not dependent on the set of query values remaining the same during a query execution, and can accept a query values set with exactly one value which may change from call to call.* ■

Joinable similarity predicates can be used for both join and selection predicates, while non-joinable predicates can only be used as selection predicates.

Our system stores the meta-data related to similarity predicates in the table *SIM_PREDICATES(predicate_name, applicable_data_type, is_joinable).*

In queries with multiple similarity predicates, a *scoring rule* combines the individual scores of predicate matches, weighted by their relative importance, into a single score [16,9]. We store the meta-data related to scoring rules in the table *SCORING_RULES(rule_name).*

Definition 4 (Scoring rule). *Given a list of scores $s_1, s_2, ..., s_n$, $s_i \in [0,1]$, and a weight for each score $w_1, w_2, ..., w_n$ with $w_i \in [0,1]$ and $\sum_i w_i = 1$, the* *scoring rule function returns an overall score in the range* $[0,1]$. *A scoring rule function has the form: scoring rule*$(s_1, w_1, s_2, w_2, ..., s_n, w_n) \rightarrow [0,1]$[2] ■

In addition to the scoring rule and similarity predicate meta-data, we maintain several tables that keep operational data for the similarity predicates and scoring functions. For each similarity predicate (SP) in the query, we store the user supplied parameters, cutoff, score variable, and the attribute(s) and query values involved. The schema for this table is *QUERY_SP(predicate name, parameters, α, input attribute, query attribute, list of query values, score variable).* For the scoring rule, we store the scoring rule (SR) used, and a list of output score variables and their corresponding weights. The schema for this table is *QUERY_SR(rule name, list of attribute scores, list of weights).*

3 Query Refinement System

In this section, we describe our query refinement system. We built our system as a wrapper on top of a commercial database and support refinement over SQL queries. We support a minimally modified SQL syntax to express refinement. Figure 1 shows our system architecture, it shows a generic refinement algorithm which accepts specific similarity predicates, scoring rules and refinement algorithms as plug-ins. We now discuss this generic refinement algorithm and in section 4 present some specific refinement algorithms we use.

The users querying process starts with several tables, user-defined data-types, similarity predicates, and scoring rules, and proceeds as follows:

1. The user expresses her abstract *information need* as an SQL statement with a scoring rule and one or more similarity predicates over some tables. Figure 2 shows a query with two similarity predicates on table T.
2. The system executes the query and populates a temporary *answer* table with the results. Figure 2 shows this *answer* table.
3. The user incrementally browses the answers in rank order, i.e., the best results first. As the user sees the results, she can mark *entire tuples* or *individual attributes*[3] as good, bad, or neutral examples, this is recorded in a *feedback* table (figure 2). It is not necessary for the user to see all answers or to provide feedback for all answer tuples or attributes.
4. With the users feedback, the system generates a new query by modifying the scoring rule and similarity predicates and goes back to step 2, executes the new query and populates the *answer* table with the new results.
5. This process is repeated as desired.

To support query refinement the DBMS must maintain sufficient state information and provide supporting interfaces to query refinement functions, we now discuss these tables and functions.

[2] *In the implementation, we used arrays for the similarity scores and weights to accommodate the variable number of parameters to scoring rule functions.*

[3] When submitting feedback, our system offers two granularities: tuple and attribute (or column). In tuple level feedback the user selects entire tuples for feedback. In attribute level feedback, the user provides finer grained information, judging individual attributes.

T:

tid	a b c d

select $\theta(b_s, v_b, c_s, v_c)$ *as* S, a, b
from T
where $d > 0$ **and**
P(b, $\hat{b}$, w_b, b_α, b_s) **and**
Q(c, $\hat{c}$, w_c, c_α, c_s)
order by S **desc**

θ	scoring rule function
O, P, Q	similarity predicates
w_a, w_b, w_c	weights for O, P, Q
$\hat{a}, \hat{b}, \hat{c}$	query values for O, P, Q
$a_\alpha, b_\alpha, c_\alpha$	cutoff values for O, P, Q
a_s, b_s, c_s	output score for O, P, Q
v_a, v_b, v_c	scoring rule weights

Answer:

tid	S	a	b	c
1	s_1	a_1	b_1	c_1
2	s_2	a_2	b_2	c_2
3	s_3	a_3	b_3	c_3
4	s_4	a_4	b_4	c_4

Feedback:

tid	tuple	a	b
1	1	0	0
2	0	0	1
3	0	-1	1
4	0	0	-1

Scores:

tid	$O(a, a_1)$	$P(b, \hat{b})$	$Q(c, \hat{c})$
1	1.0	0.8	0.9
2	–	0.9	–
3	0.2	0.8	–
4	–	0.3	–

Fig. 2. Data Structure Example for Query Refinement – Single Table

Support Tables. To refine a query, all values of tuples where the user gave feedback judgments must be available, they are needed since refinement is guided by the values of relevant and non-relevant tuples. In some queries, maintaining this information may be complicated due to loss of source attributes in projections or joins where attributes may be eliminated. To enable query refinement, we must include sufficient information in the answer to recompute similarity scores. Therefore, we construct an *Answer* table as follows:

Algorithm 1 (Construction of temporary *Answer* table for a query) The *answer* table has the following attributes: (1) a tuple id (*tid*), (2) a similarity score S that is the result of evaluating the scoring rule, (3) all attributes requested in the select clause, and (4) a set H of hidden attributes. The set of hidden attributes is constructed as follows: for each similarity predicate (from the *SIM_PREDICATES* table) in the query, add to H all fully qualified[4] attribute(s) that appear and are not already in H or the attribute was requested in the query select clause. All attributes retain their original data-type. ■

Results for the hidden attributes are not returned to the calling user or application. We also construct a *Feedback* table for the query as follows:

Algorithm 2 (Construction of temporary *Feedback* table for a query) The attributes for the *Feedback* table are: (1) the tuple id (tid), (2) a *tuple* attribute for overall tuple relevance, and (3) all attributes in the select clause of the query. All attributes are of type integer (except the tid). ■

Example 4 (Answer and Feedback table construction). Figure 2 shows a table T, and a query that requests only the score S, and the attributes a and b. The query has similarity predicates P on b, and Q on c, so attributes b and c must be in H. But b is in the select clause, so only c is in H and becomes the only hidden attribute. The feedback table is constructed as described and contains the *tid*, *tuple*, a, and b

[4] Same attributes from different tables are listed individually.

R:

tid	a b c

S:

tid	b c d

select $\theta(a_s, v_a, b_s, v_b)$ *as* S, a, d
 from R, S
 where $d > 0$ **and**
  O(a, $\hat{a}$, w_a, a_α, a_s) **and**
  P(R.b, S.b, w_b, b_α, b_s)
 order by S **desc**

θ	scoring rule function
O, P, U	similarity predicates
w_a, w_b	weights for O, P
$\hat{a}, \hat{b}$	query values for O, P
a_α, b_α	cutoff values for O, P
a_s, b_s	output score for O, P
v_a, v_b	scoring rule weights

Answer:

tid	S	a	d	⟨b⟩
1	s_1	a_1	d_1	$R.b_1, S.b_1$
2	s_2	a_2	d_2	$R.b_2, S.b_2$
3	s_3	a_3	d_3	$R.b_3, S.b_3$
4	s_4	a_4	d_4	$R.b_4, S.b_4$

Feedback:

tid	tuple	a	d
1	1	0	0
2	-1	0	0
3	1	0	0
4	0	-1	0

Scores:

tid	$O(a, \hat{a})$	$U(d, d_1)$	$P(R.b, S.b)$
1	0.7	0.9	0.8
2	0.8	0.5	0.7
3	0.3	0.4	0.6
4	0.6	–	–

Fig. 3. Data Structure Example for Query Refinement – Similarity Join

attributes. A sample Feedback table[5] is shown with both tuple and attribute level feedback. Figure 3 shows Answer and Feedback tables when a join involves a similarity predicate. When constructing the Answer table, we do not include attribute a in H since it is listed in the select clause. We include two copies of attribute b in the set H since it comes from two different tables. The figure lists them together. The Feedback table is constructed as above and populated with sample data. The headings a_1 and d_1 will be explained later. ■

We build an auxiliary *Scores* table that combines the user supplied feedback and the values from the answer table, and then populate the table:

Algorithm 3 (Construction of *Scores* table) The Scores table has the following attributes: (1) a tuple id (tid), (2) all attributes in the select clause, (3) all attributes in the hidden set H. If two attributes appear in a similarity join predicate, they are fused into a single attribute. All attributes except the tid are real valued in the range [0,1] or are undefined. The table is populated with the scores of values for which the user specifies feedback and whose attributes are involved in some similarity predicate in the query. For a pair of values such as in a join predicate, a single score results. Figure 4 shows the algorithm to populate the *Scores* table. ■

Example 5 (Scores table). Figs. 2 and 3 show corresponding *Scores* tables. ■

4 Query Refinement Strategies

Our query refinement system supports an interface for refinement algorithms paired to their corresponding similarity predicates, here we present several specific strategies for query refinement we have implemented. Query refinement is

[5] It is not necessary for the user to give feedback for all tuples in the answer set despite what is shown here.

```
——**for-all** tuples $t$ **in** *Feedback* table
— —**for-all** attributes $x$ **in** $t$
— —**if** $((t.x \neq 0) \vee (t.tuple \neq 0) \wedge \exists\ predicate\ Z\ on\ x$
— —// there is non-neutral feedback on attribute $x$
— —*Scores* table $[tid = t.tid, attribute = x] =$
— —$Z($*Answer* table $[tid = t.tid, attribute = x])$ —
—// recreate detailed scores
```

Fig. 4. *Scores* Table Computation

achieved by modifying queries in three ways: (1) changing the interpretation and weights of scoring rules that connect predicates together (inter-predicate refinement), (2) adding or removing predicates to the scoring rule (inter-predicate refinement), and (3) changing the interpretation of the user-defined domain-specific similarity predicate functions (intra-predicate refinement). Many different refinement implementations are possible, here we only present some of those that we implemented.

The scores of the several predicates in the query condition are combined via weights and a scoring rule function denoted by θ in figures 2 and 3. The techniques discussed here are possible implementations for the scoring rule refinement function discussed in section 3.

The goal of inter-predicate refinement via query re-weighting is to find the optimal relative weights among different predicates used in the scoring rule. Let $p_1, p_2, ..., p_n$ be similarity predicates, $s_1, s_2, ..., s_n$ their similarity scores, and $v_1^{opt}, v_2^{opt}, \ldots, v_n^{opt}$ the weights used in the scoring rule θ of a query, such that $\theta(s_1, v_1^{opt}, s_2, v_2^{opt}, ..., s_n, v_n^{opt})$ captures the user's notion of similarity of a tuple to the users query. Let $v_1^0, v_2^0, \ldots, v_n^0$ be the initial weights associated with the scoring rule (for simplicity start with equal weights for all predicates). Re-weighting modifies these weights based on the user's feedback to converge them to the optimal weights: $\lim_{k \longrightarrow \infty} v_i^k = v_i^{opt}$. For each predicate, the new weight is computed based on the similarity score of all attribute values for which relevance judgments exist. In section 3 we described how a *Scores* table is derived from the answer table and the feedback values. Using this information we develop two strategies:

- *Minimum Weight:* use the minimum relevant similarity score for the predicate as the new weight. When the new weight is high, all relevant values are scored high, and the predicate is a good predictor of the users need. A low weight indicates the low predictive importance of the predicate. Following figure 2, the new weight for $P(b)$ is: $v_b = \min_{i \in relevant(b)}(P(b_i)) = \min(0.8, 0.9, 0.8) = 0.8$, similarly, $v_c = 0.9$. Non-relevant judgments are ignored.
- *Average Weight:* use the average of relevant minus non-relevant scores as the new weight. Here, the score of all relevant and non-relevant values is considered, therefore this strategy is more sensitive to the distribution of scores among relevant and non-relevant values. Generically, for attribute x and predicate X, we use $v_x = \max(0, \frac{\sum_{x_i \in relevant(x)} X(x_i) - \sum_{x_j \in non\text{-}relevant(x)} X(x_j))}{|relevant(x)|\ +\ |non\text{-}relevant(x)|}$.

We use *max* to ensure that $v_x \geq 0$. Following figure 2, the new weight for $P(b)$ is: $v_b = \frac{\sum_{b_i \in relevant(b)} P(b_i) - \sum_{b_j \in non\text{-}relevant(b)} P(b_j)}{|relevant(b)| + |non\text{-}relevant(b)|} = \frac{0.8+0.9+0.8-0.3}{3+1} = 0.55$, similarly, $v_c = 0.9$.

In both strategies, if there are no relevance judgments for any objects involving a predicate, then the original weight is preserved as the new weight. These strategies also apply to predicates used as a join condition.

Once the new weights are computed, they are normalized, and their values updated in the QUERY_SR table.

The inter-predicate selection policy adds predicates to or removes them from the query and scoring rule. Users may submit coarse initial queries with only a few predicates, therefore supporting the addition of predicates to a query is important. In figure 2, attribute a was ranked relevant for tuple 1 (since the tuple is relevant overall), and non-relevant for tuple 3. This suggests there might be merit in adding a predicate on attribute a to the query and adding it to the scoring rule. We now discuss if and how this can be achieved.

Intuitively, we must be conservative when adding a new predicate to the query as it may significantly increase the cost of computing the answers, and may not reflect the users intentions. For these reasons, when adding a predicate, it must be (1) added with a small initial weight in the scoring rule to keep the addition from significantly disrupting the ranking of tuples, and (2) have a very low cutoff since before the addition the effective predicate on the attribute was *true* thus returning all values and therefore equivalent to a cutoff of 0.

To add a predicate on an attribute to the query, a suitable predicate must be found that applies to the given data-type, fits good the given feedback, and has sufficient support. Our algorithm for predicate addition is as follows: Based on the SIM_PREDICATES table, a list of similarity predicates that apply to the data-type of attribute a is constructed, this list is denoted by $applies(a)$. The *Answer* table is explored to find the highest ranked tuple that contains positive feedback on attribute a and take a's value. In figure 2, this value is a_1. This value (a_1) becomes the plausible query point for the new predicate. Next, we iterate through all predicates in the list $applies(a)$, and for each value of attribute a in the *Answer* table that has non-neutral feedback, we evaluate its score using a_1 as the query value, and set the score in the *Scores* table. The weights for the predicate are the default weights, and the cutoff value is 0. Say predicate $O(a, a_1, ...) \in applies(a)$ is under test. Under the *Scores* table in figure 2, $O(a_1) = 1.0$ and $O(a_3) = 0.2$ where a_1 is relevant and a_3 is not relevant. A predicate has a good fit if the average of the relevant scores is higher than the average of the non-relevant scores. In this case, this is true. In addition to this good fit, to justify the added cost of a predicate, we require that there be sufficient support. Sufficient support is present if there is significant difference between the average of relevant and non-relevant scores; this difference should be at least as large as the sum of one standard deviation among relevant scores and one standard deviation among non-relevant scores. If there are not enough scores to meaningfully compute such standard deviation, we empirically choose a default value of one standard deviation of 0.2. In this example thus, the default standard deviations for relevant and non-relevant scores add up to $0.2+0.2 = 0.4$ and since $average(relevant) - average(non\text{-}relevant) = 1.0 - 0.2 = 0.8 > 0.4$ then we decide that predicate $O(a)$ is a good fit and has sufficient support (i.e., separation) and is therefore a candidate to being added to the query. We choose $O(a, a_1, ...)$ over other predicates in $applies(a)$ if the separation among the averages is the largest among all predicates in $applies(a)$.

The new predicate $O(a, a_1, ...)$ is added to the query tree and to the scoring rule with a weight equal to one half of its fair share, i.e., $1/(2 \times \mid$ *predicates in scoring rule* $\mid)$. We do this to compensate for the fact that we are introducing a predicate not initially specified and do not want to greatly disrupt the ranking. If there were four predicates before adding O, then O is the fifth predicate and its fair share would be 0.2, we set a weight of $0.2/2 = 0.1$, and re-normalize all the weights in the scoring rule so that they add up to 1 as described above, and update the QUERY_SR table appropriately.

Predicate Deletion. A predicate is eliminated from the query and scoring rule function if its weight in the scoring rule falls below a threshold during re-weighting since its contribution becomes negligible, the remaining weights are re-normalized to add up to 1. For example, if we use *average weight* re-weighting in figure 3, then the new weight for attribute a is: $\max(0, \frac{(0.7+0.3)-(0.8+0.6)}{2+2}) = \max(0, -0.1) = 0$. Therefore, predicate $O(a, \hat{a})$ is removed.

In intra-predicate refinement, the similarity predicates are modified in several ways. Predicates, in addition to a single query point (value), may use a number of points as a query region, possibly with its own private, attribute-specific scoring rule function [17]. The refinement task is to use the provided feedback to update the query points, parameters, and cutoff values in the QUERY_SP table, to better reflect the users intentions.

Intra-predicate refinement is by its very nature *domain dependent*, hence each user-defined type specifies its own *scoring* and *refinement* functions to operate on objects of that type. We present some strategies in the context of a 2D geographic *location* data-type (cf. *close-to* in example 3). In the context of figure 2, let attribute b be of type *location*, and P be the *close_to* function, then we invoke the corresponding refinement function $close\_to\_refine(\ \{b_1, b_2, b_3, b_4\}, \{1, 1, 1, -1\})$, with the list of b values for which the user gave feedback and the corresponding feedback. The following approaches are used for this type:[6]

- **Query Weight Re-balancing** adjusts the weights for individual dimensions to better fit the users feedback. The new weight for each dimension of the query vector is proportional to the importance of the dimension, i.e., low variance among relevant values indicates the dimension is important [12,19]. If $b_1.x$, $b_2.x$, and $b_3.x$ are similar (i.e., small variance), and $b_1.y$, $b_2.y$, and $b_3.y$ are different (i.e., larger variance), we infer that the x dimension captures the users intention better, and set $w_x = \frac{1}{std\ dev(b_x)}$ and $w_y = \frac{1}{std\ dev(b_y)}$, and then re-normalize the weights such that $w_x + w_y = 1$.

[6] Note that this discussion uses distance functions rather than similarity functions since it is more natural to visualize the distance between points than their similarity; distance can easily be converted to a similarity value.

- **Cutoff Value Determination**, since this setting does not affect the result ranking, we leave this at 0 for our experiments, but one useful strategy is to set it to the lowest relevant score.
- **Query Point Selection** computes a new point or set of points as query values that is/are closer to relevant and away from non-relevant objects. Notice that query point selection relies on the query values remaining stable during an iteration and is therefore suited only for predicates that are not involved in a join, or predicates that are not joinable (cf. definition 3). Two ways to compute new query points are:
  - *Query Point Movement* uses a single value or point per predicate as the query, and we use Rocchios method for vector space models [18,19] for refinement. The query value $\hat{b}$ migrates to $\hat{b}$' by the formula (Rocchio) $\hat{b}' = \alpha \times \hat{b} + \beta \times \frac{\sum(b_1,b_2,b_3)}{3} - \gamma \times \frac{\sum(b_4)}{1}$, $\alpha+\beta+\gamma = 1$ where α, β, and γ are constants that regulate the speed at which the query point moves towards relevant values and away from non-relevant values.
  - *Query Expansion* uses multiple values per attribute as a *multi-point query* [17,21], multiple points are combined with a scoring rule λ. Good representative points are constructed by clustering the relevant points and choosing the cluster centroids as the new set of query points, this can increase or decrease the number of points over the previous iteration. Any clustering method may be used such as the *k-means* algorithm, as well as the technique presented in FALCON [21] which we also implemented in our system.

This set of intra-predicate refinement techniques provides for rich flexibility in adapting predicates to the users intentions. These techniques may be used individually, or combined to effect single-attribute predicate changes. Specific datatypes may have specially designed similarity predicates and refinement functions that achieve refinement in a different way, but as long as they follow the established interface, they can be seamlessly incorporated into our system.

5 Experiments

Similarity retrieval and query refinement represent a significant departure from the existing access paradigm (based on precise SQL semantics) supported by current databases. Our purpose in this section is to show some experimental results, and illustrate that the proposed access mechanism can benefit realistic database application domains.

We built our prototype system as a wrapper on top of the Informix Universal Server ORDBMS. We place our system in-between the user and the DBMS as shown in figure 1. We handle user queries, analyze and rework them, and forward relevant queries and data to the DBMS for query execution. The client application navigates through the results, starting with the best ranked ones, and submits feedback for some or all tuples seen so far. The client can stop at any time, refine and re-execute the updated query.

We study our systems various components, and present a realistic sample e-commerce catalog search application to validate the approach.

5.1 Evaluation Methodology

To evaluate the quality of retrieval we establish a baseline ground truth set of relevant tuples. This ground truth can be defined objectively, or subjectively by a human users perception. When defined subjectively, the ground truth captures the users information need and links the human perception into the query answering loop. In our experiments we establish a ground truth and then measure the retrieval quality with precision and recall [4]. *Precision* is the ratio of the number of relevant tuples retrieved to the total number of retrieved tuples: $precision = \frac{|relevant \bigcap retrieved|}{|retrieved|}$, while *recall* is the ratio of the number of relevant tuples retrieved to the total number of relevant tuples: $recall = \frac{|relevant \bigcap retrieved|}{|relevant|}$. We compute precision and recall after each tuple is returned by our system in rank order.

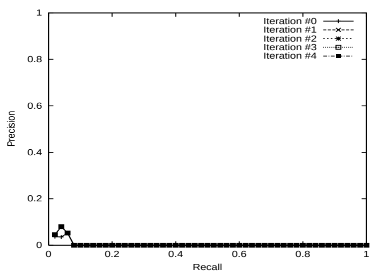
The effectiveness of refinement is tied to the convergence of the results to the users information need. Careful studies of convergence have been done in the past in restricted domains, e.g., Rocchio [18] for text retrieval, MARS [19,17], Mindreader [12], and FeedbackBypass [5] for image retrieval, and FALCON [21] for metric functions. We believe (and our experiments will show) that these convergence experiments carry over to the more general SQL context in which we are exploring refinement.

5.2 Experimental Results

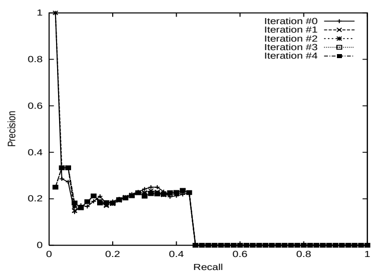
For these experiments, we used two datasets. One is the fixed source air pollution dataset from the AIRS[7] system of the EPA and contains 51,801 tuples with geographic location and emissions of 7 pollutants (carbon monoxide, nitrogen oxide, particulate less than 2.5 and 10 micrometers, sulfur dioxide, ammonia, and volatile organic compounds). The second dataset is the US census data with geographic location at the granularity of one zip code, population, average and median household income, and contains 29470 tuples. We used several similarity predicates and refinement algorithms.

For the first experiment we implemented a similarity predicate and refinement algorithm based on FALCON [21] for geographic locations, and a query point movement and dimension re-weighting for the pollution vector of the EPA dataset. We started with a conceptual query looking for a specific pollution profile in the state of Florida. We executed the desired query and noted the first 50 tuple as the ground truth. Next, we formulated this query in 5 different ways, similar to what a user would do, retrieved only the top 100 tuples, and submitted tuple level feedback for those retrieved tuples that are also in the ground truth. Here we performed 5 iterations of refinement. Figure 5a) shows the results of using only the location based predicate, without allowing predicate addition. Similarly figure 5b) shows the result of using only the pollution profile, without allowing predicate addition. Note that in these queries feedback was of little use in spite of several feedback iterations. In figure 5c) we instead used both

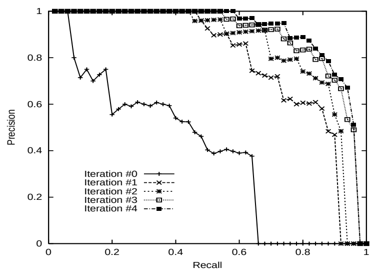
[7] http://www.epa.gov/airs



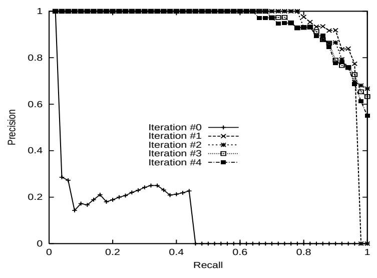
a) Location alone



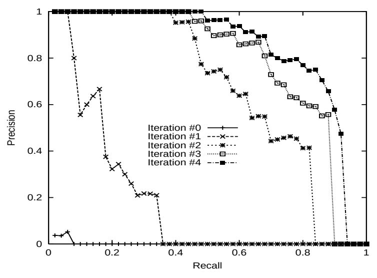
b) Pollution alone



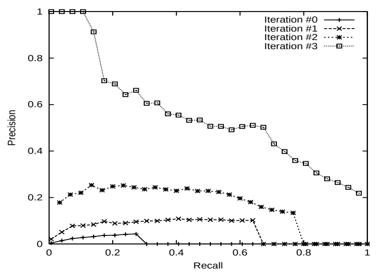
c) Location and pollution



d) Pollution, add location pred.



e) Location, add pollution pred.



f) Similarity join query

Fig. 5. Precision–recall graphs for several iterations, and refinement strategies

predicates but with the default weights and parameters, notice how the query slowly improves. Figure 5d) starts the query with the pollution profile only, but allowing predicate addition. The predicate on location is added after the first iteration resulting in much better results. Similarly figure 5e) starts only with the location predicate. The initial query execution yields very low results, but the pollution predicate is added after the initial query resulting in a marked improvement. In the next iteration, the scoring rule better adapts to the intended query which results in another high jump in retrieval quality. It is sufficient to provide feedback on only a few tuples, e.g., in this query, only 3 tuples were submitted for feedback after the initial query, and 14 after the first iteration. The number of tuples with feedback was similarly low (5%-20%) in the other queries.

In the second experiment, we use a join query over the EPA and census datasets. Notice that we cannot use the location similarity predicate from the first experiment since the FALCON [21] based similarity predicate is not joinable. This is because it relies on a set of "good points" which must remain the same over the execution of a query iteration. If we change the set of good points to a single point from the joining table in each call, then this measure degenerates to simple Euclidean distance and the refinement algorithm does not work. Figure 5f) shows the results of a query where the census and EPA datasets are joined by location, and we're interested in a pollution level of 500 tons per year of particles 10 micrometers or smaller in areas with average household income of around $50,000. We constructed the ground truth with a query that expressed this desire and then started from default parameters and query values.

5.3 Sample E-Commerce Application

To illustrate a realistic application, we built a sample E-commerce application on top of our query refinement system. We felt that the flexibility and power offered by similarity retrieval and refinement is naturally more useful in tasks involving exploratory data analysis in which users are not initially sure of their exact information need. One such application domain is searching product catalogs over the Internet. We develop a sample catalog search application in the context of garment searching. We explore how the granularity (i.e., tuple vs. column level feedback) and the amount of feedback affect the quality of results. We emphasize that searching e-tailer catalogs represents only a single point in a wide array of application domains which can benefit from similarity retrieval and refinement. Other natural application domains include bioinformatics and geographic information systems.

E-Commerce Prototype Implementation. To build our sample garment catalog search application, we collected from various apparel web sites[8] 1747 garment item descriptions of a variety of garment types including pants, shirts, jackets, and dresses. For each item, we collected its manufacturer, type (e.g., shirt, jacket, etc.), a short and long description, price, gender (male, female, unisex), colors and sizes available, and a picture of the garment itself. We also built a Java based front end user interface for the garment search application that lets users browse, construct queries and refine answers.

Each data-type has its own similarity function implemented as a user-defined function (UDF). The similarity for textual data is implemented by a text vector model [4] and is used for the manufacturer, type, and short and long description attributes. The similarity function for the price attribute is: $sim_{price}(p_1, p_2) = 1 - \frac{|p_1 - p_2|}{6 \times \sigma(p)}$.[9] The Image is represented through multiple attributes: the image is stored as a blob itself, a small thumbnail image is also stored as a blob, and two image derived features that are used for content-based image retrieval [16,19].

[8] Sources: JCrew, Eddie Bauer, Landsend, Polo, Altrec, Bluefly, and REI.

[9] This assumes that prices are distributed as a Gaussian sequence, and suitably normalized.

The features used are for color the color histogram [16] feature with a histogram intersection similarity function, and for texture the coocurrence matrix texture feature with an Euclidean distance based similarity function [16]. The remaining attributes of the garment data set were not used for search. The similarity values of the data-type specific similarity functions are combined with a weighted linear combination (weighted summation) scoring rule function.

To complement the similarity and scoring functions for retrieval, we implemented refinement algorithms for each similarity predicate, and across predicates. We used Rocchios text vector model relevance feedback algorithm [18] for the textual data. For the price, color histogram and coocurrence texture attributes we use re-weighting, and query point movement and expansion as described in section 4 [19,17].

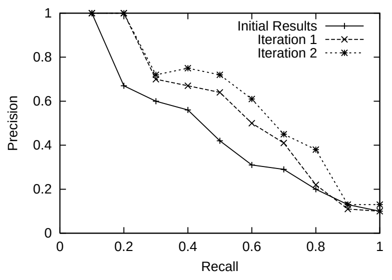
Example Query. We explore our system through the following conceptual query: "men's red jacket at around $150.00". We browsed the entire collection and constructed a set of relevant results (ground truth) for the query, we found 10 items out of 1747 to be relevant and included them in the ground truth. Several ways exist to express this query, we try the following:

1. Free text search of *type*, *short* and *long description* for "men's red jacket at around $150.00".
2. Free text search of *type* with "red jacket at around $150.00" and *gender* as male.
3. Free text search of *type* with "red jacket", *gender* as male, and *price* around $150.00.
4. Free text search of *type* with "red jacket", *gender* as male, *price* around $150.00, and pick a picture of a red jacket which will include the image features for search.

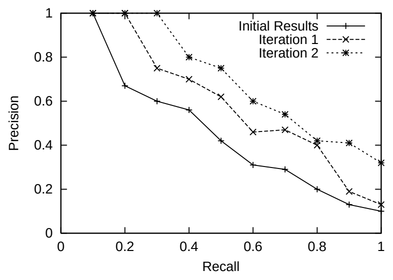
We ran all four queries with feedback and evaluate the effectiveness of refinement comparing tuple versus column level feedback using precision and recall.

Effect of Feedback Granularity on Refinement Quality. Our framework provides for two granularities of feedback: tuple and column level. In tuple level feedback the user judges the tuple as a whole, while in column level she judges individual attributes of a tuple. Column level feedback presents a higher burden on the user, but can result in better refinement quality. This is shown in figures 6a) which shows tuple level feedback and 6b) which shows column level feedback. For these figures we used 4 different queries with their ground truth, gave feedback on exactly 2 tuples and averaged the results. For tuple level feedback, 2 entire tuples were selected, for column level feedback, we chose only the relevant attributes within the tuples and judged those. As expected, column level feedback produces better results.

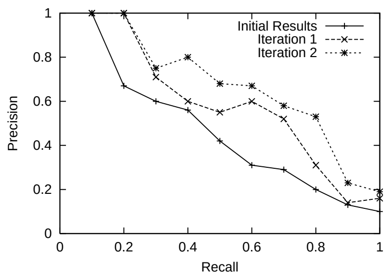
Effect of Amount of Feedback Judgments on Refinement Quality. While in general the refinement system adapts better to the users information need when it receives more relevance judgments, in practice this is burdensome to users. The amount of feedback may vary from user to user; here we explore how the amount of feedback affects the retrieval quality. Empirically, even a few feedback judgments can improve query results substantially. We explore the same 4 queries from above and give tuple level feedback for 2, 4 and 8 tuples. The results are shown in figures 6a,c,d) respectively. More feedback improves the results, but with diminishing returns, in fact column level feedback for 2 tuples



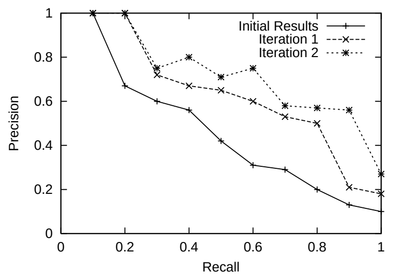
a) Tuple Feedback (2 tuple feedback)



b) Column Feedback (2 tuple feedback)



c) Tuple Feedback (4 tuple feedback)



d) Tuple Feedback (8 tuple feedback)

Fig. 6. Precision–recall graphs averaged for 5 queries for different amount and granularity of feedback

is competitive with tuple level feedback for 8 tuples. Based on this, we conclude that users need not expend a huge effort in feedback to improve their answers.

6 Related Work

While there is much research on improving the effectiveness of queries in specific applications, we are not aware of work that addresses generalized query refinement in general purpose SQL databases.

Traditionally, similarity retrieval and relevance feedback have been studied for textual data in the IR [4] community and have recently been generalized to other domains. IR models have been generalized to multimedia documents, e.g., image retrieval [19,16,10] uses image features to capture aspects of the image content, and provide searching. Techniques to incorporate similarity retrieval in databases have also been considered for text and multimedia [11,1], but generally do not consider query refinement.

Query refinement through relevance feedback has been studied extensively in the IR literature [4,18] and has recently been explored in multimedia domains, e.g., for image retrieval by Mindreader [12], MARS [19,17], and Photobook [14], among others. FALCON [21] uses a multi-point example approach similar to MARS [17] and generalizes it to any suitably defined metric distance function, regardless of the native space. FALCON assumes that all examples are drawn from the same domain and does not handle multiple independent attributes. Under our framework, we consider FALCON as a non-joinable algorithm since it is dependent on the good set of points remaining constant during a query iteration. FALCON however fits perfectly in our framework as an intra-predicate algorithm for selection queries as we discussed in section 4 and showed in our experiments. Given the applicability of refinement to textual and multimedia data, our paper takes the next step, extending refinement to general SQL queries.

Cooperative databases seek to assist the user to find the desired information through interaction. VAGUE [15], in addition to adding distance functions to relational queries, helps the user pose imprecise queries. The Eureka [20] browser is a user interface that lets users interactively manipulate the conditions of a query with instantly update the results. The goal of the CoBase [8] cooperative database system is to avoid empty answers to queries through condition relaxation. Agrawal [2] proposes a framework where users submit preferences (feedback) and explains how they may be composed into compound preferences.

7 Conclusions

In this paper, we concentrated on developing a data access model based on similarity retrieval and query refinement. We presented a framework to integrate similarity functions and query refinement in SQL databases, and presented several specific refinement strategies along with experiments to validate our approach.

References

1. S. Adali, P. Bonatti, M. L. Sapino, and V. S. Subrahmanian. A multi-similarity algebra. In *Proc. ACM SIGMOD98*, pages 402–413, 1998.
2. Rakesh Agrawal and Edward L. Wimmers. A framework for expressing and combining preferences. In *ACM SIGMOD*, 2000.
3. G. Amato, F. Rabitti, and P.Savino. Multimedia document search on the web. In *7th Int. World Wide Web Conference (WWW7).*
4. Ricardo Baeza-Yates and Ribeiro-Neto. *Modern Information Retrieval.* ACM Press Series/Addison Wesley, New York, May 1999.
5. Ilaria Bartolini, Paolo Ciaccia, and Florian Waas. Feedback bypass: A new approach to interactive similarity query processing. In *27th Very Large Databases (VLDB)*, Rome, Italy, September 2001.
6. J. P. Callan, W. B. Croft, and S. M. Harding. The inquery retrieval system. In *In Proceedings of the Third International Conference on Database and Expert Systems Applications*, Valencia, Spain, 1992.
7. Shih-Fu Chang and john R. Smith. Finding images/video in large archives. *D-Lib Magazine*, 1997.
8. Wesley W. Chu et al. CoBase: A Scalable and Extensible Cooperative Information System. *Journal of Intelligent Information Systems*, 6, 1996.
9. Ronald Fagin and Edward L. Wimmers. Incorporating user preferences in multimedia queries. In *Proc of Int. Conf. on Database Theory*, 1997.

10. M. Flickner, Harpreet Sawhney, Wayne Niblack, and Jonathan Ashley. Query by Image and Video Content: The QBIC System. *IEEE Computer*, 28(9):23–32, September 1995.
11. Norbert Fuhr. Logical and conceptual models for the integration of information retrieval and database systems. 1996.
12. Yoshiharu Ishikawa, Ravishankar Subramanya, and Christos Faloutsos. Mindreader: Querying databases through multiple examples. In *Int'l Conf. on Very Large Data Bases*, 1998.
13. Carlo Meghini. *Fourth DELOS Workshop – Image Indexing and Retrieval*. ERCIM Report, San Miniato, Pisa, Italy, August 1997.
14. T. P. Minka and R. W. Picard. Interactive learning using a "society of models". Technical Report 349, MIT Media Lab, 1996.
15. Amihai Motro. VAGUE: A user interface to relational databases that permits vague queries. *ACM TOIS*, 6(3):187–214, July 1988.
16. Michael Ortega, Yong Rui, Kaushik Chakrabarti, Kriengkrai Porkaew, Sharad Mehrotra, , and Thomas S. Huang. Supporting ranked boolean similarity queries in mars. *IEEE Trans. on Data Engineering*, 10(6), December 1998.
17. Kriengkrai Porkaew, Sharad Mehrotra, Michael Ortega, and Kaushik Chakrabarti. Similarity search using multiple examples in mars. In *Proc. Visual'99*, June 1999.
18. J.J. Rocchio. Relevance feedback in information retrieval. In Gerard Salton, editor, *The SMART Retrieval System*, pages 313–323. Prentice–Hall, Englewood NJ, 1971.
19. Yong Rui, Thomas S. Huang, Michael Ortega, and Sharad Mehrotra. Relevance feedback: A power tool for interactive content-based image retrieval. *IEEE CSVT*, September 1998.
20. John C. Shafer and Rakesh Agrawal. Continuous querying in database-centric web applications. In *WWW9 conference*, Amsterdan, Netherlands, May 2000.
21. L. Wu, C. Faloutsos, K. Sycara, and T. Payne. FALCON: Feedback adaptive loop for content-based retrieval. *Proceedings of VLDB Conference*, 2000.