Efficient Evaluation of Relevance Feedback for Multidimensional All-Pairs Retrieval

Michael Ortega
Computer Science
University of Illinois at Urbana-Champaign
miki@acm.org

Kaushik Chakrabarti
Microsoft Research
One Microsoft Way
kaushik@microsoft.com

Sharad Mehrotra
Inf. and Computer Science
University of California, Irvine
sharad@ics.uci.edu

Abstract

New retrieval applications support flexible comparison for all-pairs best match operations based on a notion of similarity or distance. The distance between items is determined by some arbitrary distance function. Users that pose queries may change their definition of the distance metric as they progress. The distance metric change may be explicit or implicit in an application (e.g., using relevance feedback). Recomputing from scratch the results with the new distance metric is wasteful. In this paper, we present an efficient approach to recomputing the all-pairs best match (join) operation using the new distance metric by re-using the work already carried out for the old distance metric. Our approach reduces significantly the work required to compute the new result, as compared to a naive re-evaluation.

1 Introduction

Traditional search engines and Information Retrieval systems have neglected an important class of searches: best all-pairs matches. The traditional retrieval paradigm consists of a user providing a suitable expression of a desired target object and let the system find those items (documents [19], web pages, images [18], etc.) that best match what the user specified, and return them arranged in a ranked list. In essence, the system is looking for the "nearest neighbors" of the specified search item.

A different class of search arises when two datasets are compared to each other in an *all-pairs* best match manner. This kind of search arises in applications such as job searching: "find the best matching $\langle resume - job\ description \rangle$ pairs", or in a real estate application: "find the closest $\langle home - school \rangle$ pairs." This type of search augments the familiar *join* operation in relational databases with the notion of similarity, top-k and ranking so familiar in *Information Retrieval* [19]. As the following example illustrates, this class of search arises naturally in a variety of today's applications:

Example 1.1 (E-commerce) Consider a real-estate database that maintains information like the location of each house, the price, the number of bedrooms etc. [4] (e.g., MSN HomeAdvisor). It also contains information like locations of schools, hospitals, shopping malls and other amenities. Suppose that a potential customer is interested in houses that are close to schools and priced around $300,000 as expressed by the following SQL query:

select *similarity*, * **from** Houses H, Schools S **where**
price_similar(H.price, 300,000) **and**
close_to(H.location, S.location)
order by *similarity* **desc**

The query has two similarity predicates *price_similar*, and *close_to*. The first predicate is a selection predicate in which we want data items that are closest to a given target value ($300,000 dollars). The second predicate corresponds to a *spatial join* that retrieves the closest pairs in the database. The degree of match of both predicates is combined to obtain the overall similarity of a house to the user query. The retrieval system ranks the houses based on the overall similarity and returns the top houses for the user to inspect. ■

An important aspect of top-k queries is user subjectivity [18, 5]. Let us consider two houses in the database: a house A that is priced at $400,000 about 1 mile from a school and a house B that is priced at $325,000 about 10 miles from a school. Which one is a better match? It depends on the user. If she is very particular about the proximity to school and flexible about the price, A is a better match. Conversely, if she has a limited budget and does not mind the 10 miles distance, B is better. To return "good quality" answers, the system must understand the user's *perception* of similarity, i.e., the relative importance of the attributes/features[1] to the user. The system models user perception via the distance functions (e.g., Euclidean in example 1.1) and the weights associated with the features [6, 3, 15, 4]. At query time, the system *acquires* information from the user based on which it determines the weights and distance functions that best capture the perception of this particular user and instantiates the model with these values. This instantiation is done at query time since the users perception differs from user to user and query to query. Once the model is instantiated, we retrieve the top results for each predicate[2] and then *merge* them to get the overall answers [14, 13, 6, 5].

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

[2] In this paper, we assume that all the feature spaces are metric and an index (called the Feature-index or F-index) exists on each

SAC 2003, Melbourne, Florida, USA

Due to the subjective nature of top-k queries, the answers returned by the system to a user query usually do not satisfy the user's need right away [15, 3, 9]. This can happen due to several reasons: the starting examples may not be the best ones to capture the information need (IN) of the user or the starting weights may not accurately capture the users perception or both. In this case, the user would like to *refine* the query and resubmit it in order to get back a better set of answers. We refer to this process as *relevance feedback* and the new query is called the "refined" query. In a *query-by-example* (QBE) environment (e.g., multimedia databases), the user typically refines the query by finding, among the answers returned to the "starting" query, one or more results that best reflect what she wants, and requesting for more objects like those [15, 21, 9]. Based on the user feedback, the system computes new query parameters and executes the refined query. Another way to refine the query is for the user to explicitly modify the perception model, i.e., to explicitly change weights to better capture her perception of similarity [12, 5]. In either case, the user may continue refinement iterations until she is satisfied with the results. Recent work shows that query refinement techniques significantly improve the quality of answers over a few iterations [15, 9, 18, 21].

While there has been a lot of research on improving the effectiveness of query refinement as well as on evaluating top-k queries efficiently, there exists no work that we are aware of on *how to efficiently support refinement of top-k queries inside a retrieval system.* In this paper, we explore one approach to solve the all-pairs (join) problem for multidimensional data (e.g., geographic locations, visual image features, document vectors). A naive approach to supporting query refinement is to treat a refined query just like a starting query and execute it from scratch. We observe that the *refined queries are not modified drastically* from one iteration to another; hence executing them from scratch every time is wasteful. Most of the execution cost of a refined query can be saved by appropriately *exploiting the information generated during the previous iterations of the query.* Specifically, we cache the priority queues generated by the retrieval algorithm during the execution of the query. We show how to execute subsequent iterations of a join query efficiently by utilizing the cached information. Note that since the query changes, albeit slightly, from iteration to iteration (i.e., the query points and/or the weights/distance functions are modified), we, in general, cannot answer a refined query entirely from the cache i.e., we still need to access some data from the disk. Our technique minimizes the amount of data that is accessed from disk to answer a refined query. Our techniques are independent of the way the user provides feedback to the system, i.e., it does not matter whether she uses a QBE (i.e., "give me more like this") interface or an explicit weight modification interface. Our experiments on real-life datasets show that our technique significantly improves the execution cost of refined queries over the naive approach.

The rest of this paper is developed as follows. Section 2 presents some background. Section 3 presents our algorithm for efficient evaluation of refined join queries. Section 4 presents experiments to validate our approach. Section 5 presents some related work, and section 6 offers some conclusions.

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.

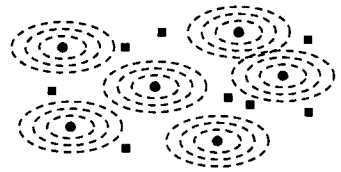

Figure 1: NN Selection and Join Distance Function

2 Background

Before we embark on an explanation of our algorithm, we first review the concepts and previous work that underlie our approach. We describe our interpretation of distance, followed by the incremental nearest neighbor on which our join algorithm builds. Finally, we discuss how the distance metric can change in response to a users feedback, and how this relates to the efficiency of re-executing a query with a modified distance metric. We base our work on a variant of the well known R-tree, the R*-tree [1], but any data structure that supports MINDIST (defined below) is also possible.

2.1 Query representation

We briefly review the interpretation of distance that we use in the latter parts of the paper. While the standard distance metric used to build and query R-trees is the Euclidean distance (also called an L_2 metric), any weighed L_p metric can be used. The distances for the query results depend on the shape of the distance metric. The shape of the distance function is based on any weighted L_p metric, each dimension i carries a weight w_i:

Definition 2.1 (DIST) The DIST distance function compares two points from different datasets with the same dimensionality. Let the dimensional weights be w_i, $1 \leq i \leq d$ where d is the dimensionality of the space, and all weights are on average 1 ($d = \sum_i(w_i)$).

$$DIST(P1, P2) = \sqrt[p]{\sum_{i=1}^{d} w_i |\ P1[i] - P2[i]\ |^p}$$ ■

In a multidimensional join, a pair with one point from each data source is compared using DIST. Figure 1 shows how DIST looks like in a 2 dimensional space for two datasets being joined together.

2.2 Nearest Neighbor selection

We first discuss the more familiar *nearest neighbor* selection since it is more familiar in the context of Information Retrieval and because it forms the foundation for the join algorithm.

There are several approaches to compute the nearest neighbors of a spatial object or point [17, 7]. Given the constraints of the application area we envision, the following properties are desirable in an algorithm:

- it works with R*-trees and other data partitioning indexing structures that support the MINDIST operator
- the output is sorted: it returns the nearest neighbors ordered by their distance to the query
- the algorithm is easily implemented in a pipeline fashion and has no a priori restriction on the largest distance or number of neighbors to be returned

- avoids the potential of query restarts inherent in the arbitrary selection of an epsilon for a range query

An algorithm that fulfills these requirements is an incremental nearest neighbor algorithm based on the work of Roussopoulos [17] and Hjaltson [7]. The algorithm uses a priority queue to keep the minimum distance (MINDIST) from the query Q for tree nodes, or the true distance for data points (DIST, defined above). MINDIST has the same interpretation as DIST, but is adjusted to return the minimum distance from a query Q to the minimum bounding rectangle (MBR) of a node. We next define MINDIST:

Definition 2.2 (MINDIST(Q, N)) Given the d dimensional bounding rectangle $R_N = \langle L, H \rangle$ of a node N, where $L = \langle l_1, l_2, ..., l_d \rangle$ and $H = \langle h_1, h_2, ..., h_d \rangle$ are the two endpoints of the major diagonal of R_N, $l_i \leq h_i$ for $1 \leq i \leq d$. The nearest point $NP(P^i, N)$ in R_N to each point P^i in the multi-point query Q is defined as follows:

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

where $NP[j]$ denotes the position of NP along the jth dimension. $MINDIST(Q, N)$ is defined as:

$$MINDIST(Q, N) = \sum_{i=1}^{n} DIST(P^i, NP(P^i, N))$$

The incremental NN algorithm can handle arbitrary distance functions $DIST$ (i.e., L_p metrics with arbitrary weights). The algorithm is correct if $MINDIST(Q, N)$ always lower bounds $DIST(Q, T)$ where T is any point stored under N (for a proof, see [3]). ■

2.3 Multidimensional Join

Among the many available algorithms, the incremental join algorithm by Hjaltson and Samet [8] stands out for satisfying several desirable properties:

- it is similar in spirit to the incremental Nearest Neighbor algorithm we use
- it works with R*-trees and other data partitioning indexing structures that support the MINDIST operator
- the output is sorted: it returns the closest pairs first, followed by more distant pairs
- the algorithm is easily implemented in a pipeline fashion and has no a priori restriction on the largest distance or number of pairs to be returned
- it is optimized for the case where the number of pairs shown to the user is small as compared to other algorithms that must first compute the full result, then sort it before being presented to the user

For these reasons, we used this algorithm as a basic building block to develop an algorithm for efficient re-evaluation of refined queries.

Hjaltson presented several variations of the join algorithm which differ in the policy for navigating the trees [8]. Some policies give preference to a depth-first approach, while a symmetric approach is more breath-first in nature. Regardless of the traversal policy, the algorithm uses a priority queue to process a pair of nodes or data items at a time. The priority queue returns the closest pair of ⟨*node*, *node*⟩, ⟨*node*, *point*⟩, ⟨*point*, *node*⟩, ⟨*point*, *point*⟩ seen so far. If the pair consists of data items only, it is returned with the corresponding distance. Else, the pair is explored (refined) into its components

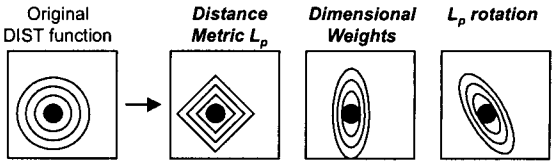

Figure 2: Feedback Parameters to Adjust

for which new distances are computed and are added to the priority queue. The algorithm also incorporates a pruning option to limit the maximum distance between pairs which in turn influences the size of the priority queue.

In addition to the DIST and MINDIST functions from definitions 2.1 and 2.2, the algorithm also uses an auxiliary function MINDIST_RECT to compare pairs of node MBRs. We define MINDIST_RECT next:

Definition 2.3 (MINDIST_RECT(N, N)) Given two d dimensional bounding rectangles $R1_N = \langle L1, H1 \rangle$ and $R2_N = \langle L2, H2 \rangle$ of the nodes N_1 and N_2, where $L = \langle l_1, l_2, ..., l_d \rangle$ and $H = \langle h_1, h_2, ..., h_d \rangle$ are the two endpoints of the major diagonal of R_N, $l_i \leq h_i$ for $1 \leq i \leq d$.

The nearest distance nd between the rectangles $R1$ and $R2$ is defined as follows:

$$nd[j] = \begin{cases} R1.L[j] - R2.H[j] & \text{if } R1.L[j] > R2.H[j] \\ R2.L[j] - R1.H[j] & \text{if } R1.H[j] < R2.L[j] \\ 0 & otherwise \end{cases}$$

where $nd[j]$ denotes the nearest distance between $R1$ and $R2$ along the jth dimension.

$MINDIST\_RECT(N1, N2)$ if defined as:

$$MINDIST\_RECT(N1, N2) = \sqrt[p]{\sum_{i=1}^{n} w_i \times |nd[i]|^p}$$

The join algorithm can handle arbitrary L_p distance metrics with arbitrary weights for each dimension. The algorithm is correct if $MINDIST(N1, N2)$ always lower bounds $DIST(T1, T2)$ where $T1$ and $T2$ are any points stored under $N1$ and $N2$ respectively. ■

2.4 Relevance feedback

We turn our attention to how the distance functions can change in response to user feedback. Any change is reflected in the DIST, MINDIST, and MINDIST_RECT functions, specifically, in the functions' weights.

We present several specific strategies for modifying the distance function used in a query. Although many different relevance feedback algorithms are possible, we present several techniques that we have implemented. Figure 2 shows an overview of refinement techniques that are described next.

- **Distance Metric Selection** determines which of a set of distance metrics is most consistent with the supplied feedback. For example, any L_p metric can be used, not just Euclidean distance. This corresponds to changing the parameter p in the distance functions. We compute the L_1 and L_2 based distances between all values marked as relevant by the user and average them. The metric that gives the least average distance is the new distance metric.[3]

[3]Although this algorithm is $O(n^2)$, in practice there are very few

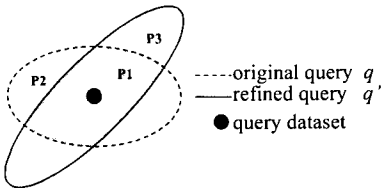

Figure 3: I/O overlap of original and new query

- **Query Weight Re-balancing** adjusts the weights for individual dimensions to better fit the feedback. The new weight for each dimension is computed as the inverse of the variance for that dimension among relevant values. These weights are proportional to the importance of the dimension, i.e., low variance among relevant values indicates the dimension is important [9, 18]. Mindreader [9] extends reweighting (stretching) with a rotatated L_p metric.

The query processing optimization for feedback we develop in this paper is independent of the feedback techniques used, thus one can easily develop more powerful feedback strategies and seamlessly incorporate them with our work.

3 Efficient evaluation of refined queries

A naive way to evaluate a refined query is to treat it just like a starting query and execute it from scratch. This approach is wasteful as we can save most of the execution cost of the refined query, both in terms of disk accesses (I/O cost) and distance calculations (CPU cost), by exploiting information generated during the previous iterations of the query.

We now discuss how to optimize join queries after feedback has been submitted and processed according to section 2.4. Due to the high cost of executing joins, it is inefficient to naively re-execute them from scratch after an iteration of relevance feedback. A naive re-execution will cause substantial unnecessary disk accesses, even when a traditional LRU database buffer cache is used due to the poor locality of the distance join algorithm.

As described in section 2.3, we use a distance based join algorithm with two index trees and a priority queue Q to maintain enough state to compute the next result pair. In addition to this priority queue, we cache the pairs already returned to the user in an unsorted list for later processing (Q_{old}).

For joins, the distance function can change in its eccentricity and orientation in space. Figure 3 shows iso-distance contours of the same distance for an original and refined distance function.[4] All data point and node pairs in the region of the original query have been explored and are included in Q or Q_{old}. All possible point or node pairs in the shaded region can be re-utilized for computing the result of the refined query. In the figure, point $P2$ was returned for this query point in the previous iteration and is also included in the new iteration. $P3$ however is in a region that has not been explored in connection with the present query point, and so may incur additional disk accesses.

Algorithm 3.1 improves the efficiency for subsequent query iterations by caching the priority queue and results from earlier iterations.[5] After a user is done viewing the results of a query iteration, submits feedback and the system computes a new distance function, we initialize a new iteration by constructing a new priority queue Q'. We construct the new priority queue Q' by recomputing the distance for each pair from the original priority queue Q and the data already returned to the user Q_{old}. This ensures that we have updated the algorithms state for the new distance function and preserve the correctness property so we can continue exploring at will. Note that if the list Q_{old} becomes too large to remain in memory, it can be sequentially written to disk and later sequentially read to be included in Q'. After all items are transferred, Q_{old}, and Q are discarded and Q' becomes Q ($Q \leftarrow Q'$). This process repeats for subsequent iterations.

4 Experiments

Similarity retrieval and relevance feedback represent a significant departure from the existing precise matching semantics supported by current databases. Our purpose in this section is not to show the merits of relevance feedback, but of the efficiency of our algorithms to re-evaluate refined queries. The merits of query refinement have been discussed extensively elsewhere, e.g., [16, 14, 18, 9, 21].

Experimental setup. We implemented and tested algorithm 3.1. All experiments were conducted on a Sun Ultra Enterprise 450 with 3GB of physical memory and several GB of secondary storage, running Solaris 2.7.

In general, the priority queues for the NN and join algorithms may grow too large to remain in main memory. Many works on nearest neighbor retrieval assume that the entire priority queue fits in memory [20].[6] The size needed depends on the dimensionality of the data, and on the number of entries which itself depends on the degree of overlap between index nodes. Several approaches exist to store a priority queue on disk, e.g., the slotted approach presented in [8]. In our join experiments, even when fetching the top 10,000 pairs out of a possible 1.5 billion, the maximum size of the priority queue was about 1.6MB, well within limits of main memory. Therefore, for our experiments, we isolated our algorithm from the issue of an external priority queue implementation and assume the priority queue is small enough to fit in memory.

For this experiment, we chose a two dimensional join based on geographic data to simulate the geographic distance join of example 1.1. We used the following two datasets: (1) the fixed source air pollution dataset from the AIRS[7] system of the EPA which contains 51,801 tuples with geographic location and emissions of 7 pollutants , (2) the US census data which contains 29470 tuples that include the geographic location and some demographic data at the granularity of one zip code. We constructed an R*-tree for each of these datasets with a 2KB page size, for a maximum fanout of 85. The pollution dataset index had 923 nodes and the census index had 548 nodes.

We focused only on the geographic location attribute and constructed queries with different eccentricities and orientations. Then, we used these queries as starting and goal queries and let the query refinement generate the intermediate queries. We tested starting to goal query difference in eccentricity of

(<20) feedback values simplifying this problem.

[4]This figure shows distance functions overlapped with a single data point from the query dataset, these functions are conceptually overlapped with all data points in the query dataset and compared to the other dataset to find the nearest pairs of points, as depicted in figure 1.

[5]We implemented the pruning option from [8] (cf. sec. 2.3), we do shown it here for clarity reasons.

[6]This assumption is reasonable when the number k of top matches requested is relatively small compared to the size of the database which is usually the case [2].

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

Algorithm 3.1 (Incremental Join Algorithm)

```
type { (node | point) :A, (node | point) :B} : pair
variable Q: MinPriorityQueue(distance, pair);
variable Q_old: Queue(distance, pair);

function NewIteration()
  /* DIST, MINDIST, and MINDIST_RECT were */
  /* modified by the feedback process of section 2.4 */
  variable Q': MinPriorityQueue(distance, pair);
  while not Q_old.empty() do /* process earlier returns */
    top = Q_old.pop()
    /* top.A and top.B must be points */
    Q'.insert(DIST(top.A, top.B), ⟨top.A, top.B⟩)
  enddo
  while not Q.empty() do
    top = Q.pop()
    if top.A and top.B are points
      Q'.insert(DIST(top.A, top.B), ⟨top.A, top.B⟩)
    else if top.A and top.B are nodes
      Q'.insert(MINDIST_RECT(top.A, top.B),
                                  ⟨top.A, top.B⟩)
    else if top.A is a node and top.B is a point
      Q'.insert(MINDIST(top.B, top.A), ⟨top.A, top.B⟩)
    else if top.A is a point and top.B is a node
      Q'.insert(MINDIST(top.A, top.B), ⟨top.A, top.B⟩)
    endif
  enddo
  Q = Q' /* the re-processed queue becomes Q */
end function

function GetNextPair()
while not Q.IsEmpty() do
  top=Q.pop();
  if top.A and top.B are objects /*i.e., points*/
    Q_old.append(top); // keep it in history
    return top;
  else if top.A and top.B are nodes
    for each child o1 in top.A
     for each child o2 in top.B
       if o1 and o2 are nodes
         Q.insert(MINDIST_RECT(o1, o2), ⟨o1, o2⟩);
       if o1 is a node and o2 is a point
         Q.insert(MINDIST(o2, o1), ⟨o1, o2⟩);
       if o1 is a point and o2 is a node
         Q.insert(MINDIST(o1, o2), ⟨o1, o2⟩);
       if o1 is a node and o2 is a point
         Q.insert(DIST(o1, o2), ⟨o1, o2⟩);
  else if top.A is a node and top.B is a point
    for each child o1 in top.A
      if o1 is a node
        Q.insert(MINDIST(top.B, o1), ⟨o1, top.B⟩);
      if o1 is a point
        Q.insert(DIST(o1, top.B), ⟨o1, top.B⟩);
  else if top.A is a point and top.B is a node
    /* mirror of above with top.A and top.B reversed*/
  else if top.A is a point and top.B is a point
    Q.insert(DIST(top.A, top.B), ⟨top.A, top.B⟩);
  end if
enddo
```

up to 2 orders of magnitude, which represents a substantial change in the distance function, far more than can be expected under normal query circumstances. Even so, our algorithm performed quite well.

For the same sets of starting and target queries, we ran tests to obtain the top 1, 10, 100, 1000, and 10,000 pairs[8] that match the query and performed feedback on them.[9] For each of these, we measured the number of disk I/Os performed, the cpu response time (ignoring queue overhead), and the wall clock response time (subtracting queue overhead).

Results. Figure 4 shows the number of node accesses performed (assuming no buffering) by the naive approach (re-executing the query from scratch every time) and our reconstruction approach. After the initial query, subsequent iterations perform minimal node accesses for the reconstruction approach while the naive approach roughly performs the same work at each iteration. We also compare to the nested loop join algorithm. Since we assume the priority queue for our algorithm has unlimited buffer space in memory, to be fair to nested loop join, we assume unlimited memory and fit both complete datasets in memory (thus only 1 access per page). We included the disk accesses needed by the nested loop join algorithm and divided this value by 10 to reflect the relative advantage of sequential reads over random reads [3]. Thus, the roughly 150 accesses already take into account the advantages gained from sequential access. While slightly lower than the naive re-execution, nested loop suffers from a very high cpu overhead and easily loses out to the naive algorithm when overall time is considered (2-3 orders of magnitude). Figure 5 shows the CPU time needed by the algorithm excluding the priority queue overhead. The time required for the nested loop join computations (ignoring the sort time since we ignored the priority queue overhead above) is roughly 21 minutes for the approximately 1.5 billion possible pairs. The cpu time includes the cost of reconstructing the priority queue for each new iteration. Figure 6 shows the total wall clock time (excluding priority queue overhead) for the queries. Nested loop join (ignoring sort time) takes roughly 22 minutes to complete this query even when both datasets fit entirely in memory. This is due to the very high number of distance computations required. The total response time using the reconstruction approach is typically 3-4 times faster than using the naive approach. Overall, the reconstruction technique significantly outperforms the naive approach or the nested loop join technique.

[8] We ran these tests independently, i.e., information in the priority queue was not shared among them.

[9] For the experiment that requests only the top pair we did not use feedback, since there is not enough information to meaningfully compute a new distance function. Instead we used the per-iteration distance functions of the top 100 query.

5 Related Work

Traditionally, similarity retrieval and relevance feedback have been studied for textual data in the IR community and have recently been generalized to other domains. IR has developed numerous models for the interpretation of similarity [19], e.g., Boolean, vector, probabilistic, etc. models. IR models have been generalized to multimedia documents, e.g., image retrieval [18] uses image features to capture aspects of the image content, and adapts IR techniques to work on them.

Techniques to incorporate similarity retrieval in databases have also been considered both for text and multimedia [12]. There are several algorithms to support top-k similarity queries, such as Fagins [6] algorithm. There are a plethora of spatial join algorithms that perform better than the reliable nested loop join, [11] presents a good overview of the algorithms available to solve this problem. Koudas [11] gives an algorithm based on space filling curves that does not need pre-built indices. Query refinement through relevance feed-

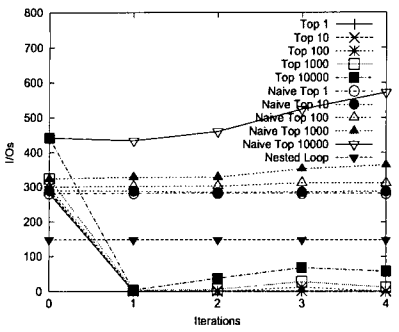

Figure 4: I/O cost of naive, reconstruction, and nested loop approaches

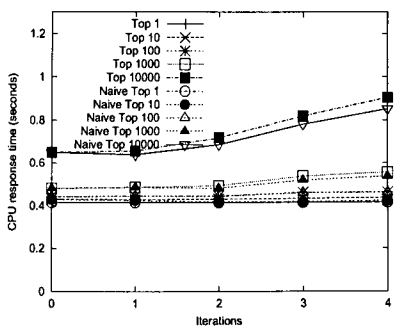

Figure 5: CPU time of naive and reconstruction approaches

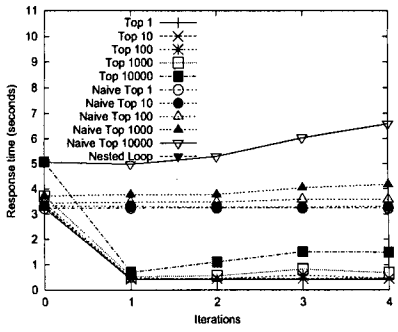

Figure 6: Total response time of naive and reconstruction approaches

back has been studied extensively in the IR literature [19, 16] and in multimedia domains, e.g., for image retrieval by Mindreader [9], FALCON [21], and MARS [18] among others. Any of these relevance feedback approaches can be used in conjunction with out approach. [10] considers a succession of *manually* modified *precise* queries as a *browsing session* and optimizes the computation of results by computing the differences between old cached query results and the new query.

6 Conclusions

Our goal is to enhance retrieval systems with similarity search and user guided query refinement through feedback. In this paper we concentrated on the all-pairs problem for multidimensional data and how to execute refined searches efficiently. Our experiments show that significant gains are possible with smart application oriented caching of the results of earlier iterations.

7 Acknowledgments

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

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B.Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc ACM SIGMOD*, 1990.

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

[3] Kaushik Chakrabarti, Michael Ortega, Kriengkrai Porkaew, and Sharad Mehrotra. Evaluating refined queries in top-k retrieval systems. *IEEE Transactions on Knowledge and Data Engieneerig*, (to appear), 2003.

[4] Surajit Chaudhari and Luis Gravano. Evaluating top-k selection queries. *Proc. of VLDB, Edinburgh, Scotland*, 1999.

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

[6] Ronald Fagin. Combining fuzzy information from multiple systems. *15th ACM PODS*, 1996.

[7] Gisly R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *Advances in Spatial Databases - 4th Symposium, SSD'95, M. J. Egenhofer and J. R. Herring, Eds., Lecture Notes in Computer Science 951, Springer-Verlag, Berlin*, pages 83–95, 1995.

[8] Gisli R. Hjaltson and Hanan Samet. Incremental distance join algorithms for spatial databases. In *Proc. ACM SIGMOD*, pages 237–248, 1998.

[9] Yoshiharu Ishikawa, Ravishankar Subramanya, and Christos Faloutsos. Mindreader: Querying databases through multiple examples. In *Int'l Conf. on Very Large Data Bases*, 1998.

[10] Martin L. Kersten and M.F.N de Boer. Query optimization strategies for browsing sessions. In *IEEE 10th Int. Conf. on Data Engineering (ICDE)*, pages 478–487, February 1994.

[11] Nick Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *IEEE International Conference on Data Engineering ICDE, pp. 466-475.*, 1998.

[12] Amihai Motro. VAGUE: A user interface to relational databases that permits vague queries. *ACM TOIS*, 6(3):187–214, July 1988.

[13] 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.

[14] Michael Ortega-Binderberger, Kaushik Chakrabarti, and Sharad Mehrotra. An Approach to Integrating Query Refinement in SQL. In *Proc. EDBT*, March 2002.

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

[16] J.J. Rocchio. Relevance feedback in information retrieval. In Gerard Salton, editor, *The SMART Retrieval System*, pages 313–323. Prentice–Hall, Englewood NJ, 1971.

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

[18] 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.

[19] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw Hill Computer Science Series, 1983.

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

[21] L. Wu, C. Faloutsos, K. Sycara, and T. Payne. FALCON: Feedback adaptive loop for content-based retrieval. *Proceedings of VLDB Conference*, 2000.