Paper Number: #174

Merging results from Multi-parametric Ranked Queries

Vagelis Hristidis
University of California, San Diego
vagelis@cs.ucsd.edu

Yannis Papakonstantinou
University of California, San Diego
yannis@cs.ucsd.edu

Abstract

Many Web sources rank objects according to multiple attributes. As the number of such sources increases, so does the number of meta-brokers, which provide integrated access to the sources. Meta-brokers answer to user queries by accessing the sources and merging the source responses. We study the fundamental problem of how a metabroker can deliver a prefix of the user query result (i.e., the top results of the user query) by accessing only a minimum prefix of the result of each source. The challenge is that the sources support different ranking functions from each other and from the ranking that the user query requests. We present an algorithm that inputs the query ranking function and the ranking functions of the sources and computes the minimum prefix required by each source in order to deliver a prefix of the query result. We consider ranking functions that are either linear functions of the object attributes or linear combinations of the attributes' logarithms or cosine functions. Our algorithm can be used in many applications, such as data streaming and multimedia, that need to efficiently merge ranked lists.

1 Introduction

An increasing number of Web applications allow queries that rank the source objects according to a function of their attributes [CG96, Fag96, GM97]. For example, consider the used car section of `www.personallogic.com`. The underlying database contains attributes such as the price, year, and mileage of the listed used cars. The user expresses his preference by picking weights for each car attribute from a menu of possible weights. Then the system orders the used cars according to a function of the user-provided weights and outputs the top results to the user. Other Web applications allow their users to rank their source objects according to a few "canned" ranking functions. For example, many Web-based brokers (e.g., Schwab, E-Trade) allow the user to declare his/her investment goals and then return to the user a list of mutual funds that is ranked in accordance to the goals. For example, if the user declares that he/she is an aggressive long-term investor the system ranks the mutual funds using a function that gives high weight to the growth rate, low weight to the volatility, and very low weight to the produced income. Note that the source objects may be relational tables, documents, images, or other types of data where ranked search makes sense.

As this type of Web sources increases, so does the number of *meta-brokers* [GM97, GCMP97] (also called meta-searchers). A meta-broker uses multiple underlying sources to answer to user queries by merging the results that these sources produce (see Figure 1). For example, a used car meta-broker allows the user to provide the weight he assigns to the price, year, and mileage of the car. Then the meta-broker contacts multiple used car sites, obtains car records and merges them into a single answer list.

An example of such a Web site is *mySimon.com*, which is a meta-searher for various products like books, computers, etc. It sends the user's query to multiple underlying sources (online retailers) and then merges the results ordered by an attribute of the products. Notice that our algorithm is more general in that it allows the objects to be ordered according to a function of their attributes. Consider for example two online bookstores that rank their resulting books according to their price and their delivery time, measured in number of days from today. The first bookstore assigns a weight of 0.8 to price and 0.2 to delivery time and the second 0.9 and 0.1 respectively. Now suppose a user request from the meta-broker to rank the books assigning weight 0.7 to price and 0.3 to delivery time. The naive algorithm would get all results from both sources, evaluate the user's preference function for each book and output the ranked results. On the other

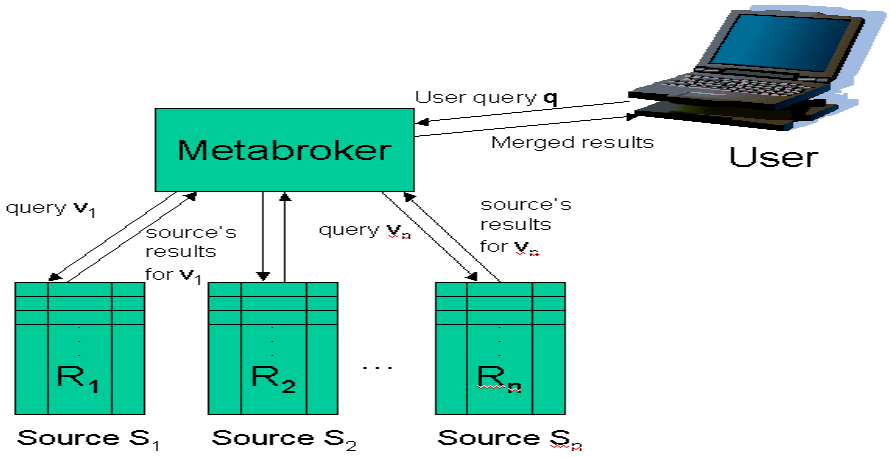


Figure 1: Meta-broker Architecture

hand our algorithm only retrieves a prefix of the results from each source and combine these books to output the top results at the meta-broker. It then keeps retrieving, merging and outputting results in a pipelined manner.

The key efficiency problem is how to output the top-K results of the user query, where K is a user-provided number, by accessing the minimum prefix of the result list that each source produces. This is a hard problem because the sources typically use different ranking functions from each other and from the function requested by the user. The meta-broker needs to efficiently access the sources and merge the results.

This paper solves the minimum prefix problem for the case where the ranking functions are (i) linear combinations of the source attributes, or (ii) linear combinations of the logarithms of the source attributes, or (iii) cosine functions (in the spirit of cosine functions used in document retrieval).

2 Related Work

Recently the problem of rank merging has received a lot of attention. [GM97] defined the term "user query is manageable by source" to indicate that a ranked user query issued to the metabroker can be answered using only a prefix of the source data. [GM97] made the same architectural assumptions with our work (known source ranking functions, metabroker retrieves a prefix from each source) but did not focus on any specific class of ranking functions. Consequently, [GM97] did not provide a prefix computing algorithm. Our work focuses on a specific class of functions described in Section 3 and fully solves the problem for this class of functions.

Our work assumes that the source ranking functions are known. This is an assumption that does not generally apply today to Web sources and search engines. Such systems typically do not disclose their ranking functions. However, our work can be materialized on top of a class of works that propose ways to get information about the source ranking functions by asking some training queries as in [VGJL95] or by calibrating the document scores of the sources using statistics as in [CLC95]. We can employ similar training techniques to learn the source ranking functions if they are not given. Equally important for our work are recent initiatives, such as [GCMP97], that allow search engines to export their ranking functions.

Multimedia sources have also received significant attention in the context of ranked queries. [Fag96] and [CG96] propose rank merging solutions for such sources, where objects are ranked according to how well they match the query values. The solutions in [Fag96], [CG96] and [FLN01] are based on a richer architectural framework than the one we assume in our work. In particular, our algorithm does not require any random accesses to the sources, which is a very important property as explained in [FLN01]. Other random access assumptions have been made in [Fag96] and [CG96].

A considerable amount of work ([FLN01], [GBK01], [GBK00], [NR99]) is available for combining ranked lists of attributes in order to efficiently retrieve the top results according to an aggregate function of the

attributes. In these works a sorted list is used for each attribute in order to efficiently retrieve the top-K ranked results from a single source. Our work is more general because we access multiple sources that order their objects according to functions of their attributes, instead of ranking by one attribute each time.

Our solution is built on an appropriate adaptation of the "watermark" concept, which was introduced in [HKP01], where the goal was to create and use materialized views of a relation in order to answer ranked queries efficiently and in a pipelined manner. The key difference between [HKP01] and the current work is that [HKP01] did not consider merging data of multiple sources. Besides the different functionality, the current work also considers a wider class of functions than the purely linear ones considered by [HKP01].

3 Notation and Definitions

Consider a set of sources $S_1, \ldots, S_n$. Each source S_i exports a relation R_i. The exported relations have the same schema or at least some common attributes. A metabroker M over $S_1, \ldots, S_n$ exports the relation $R = R_1 \cup R_2 \cup \ldots \cup R_n$ if all relations have the same schema. If the sources do not have the same schema, R is defined as follows:

- The set of attributes $(A_1, \ldots, A_n)$ of R is the union of the sets of attributes of $R_1, \ldots, R_n$.
- If the relation $R_i(A_{i_1}, \ldots, A_{i_m})$ has a tuple $t \equiv [A_{i_1} : a_{i_1}, \ldots, A_{i_m} : a_{i_m}]$ then the relation R has a tuple $t' \equiv [A_1 : a_1, \ldots, A_k : a_k]$ where
  - $a_j = a_{i_l}$, if $A_j \equiv A_{i_l}$ and
  - $a_j = NULL$, if $A_j \notin \{A_{i_1}, \ldots, A_{i_m}\}$

Let us assume that R has k attributes $(A_1, \ldots, A_k)$. Let $[m_i, M_i]$ be the domain of attribute A_i, $1 \leq i \leq k, m_i, M_i \in \mathcal{R}^+$. The notation $A_i(t)$ refers to the value of attribute A_i in the tuple t.

The user issues queries that order the tuples of R according to a weighted preference function. We refer to such queries as *preference* queries. Every query q consists of a *preference function* $f(.)$ and a single relation R. The preference function $f(t)$, $\prod_{i=1}^{k}[m_i, M_i] \rightarrow \mathcal{R}^+$ defines a numeric *score* for each tuple $t \in R$. The output of the query is the *query result sequence* $R_q = [t_q^1, t_q^2, \ldots, t_q^n]$ of the tuples of R such that $f(t_q^1) \geq f(t_q^2) \geq \ldots \geq f(t_q^n)$. Note that we use the notation t_q^i to denote the tuple in the i-th position in the result sequence of q.

In this paper we focus on queries that use three kinds of preference functions:

- linear, $f_{\vec{v}}(t) = \sum_{j=1}^{k} v_j A_j(t)$
- logarithmic, $f_{\vec{v}}(t) = \sum_{j=1}^{k} v_j log(A_j(t))$ and
- cosine, $f_{\vec{v}}(t) = \frac{1}{|\vec{t}||\vec{v}|} \sum_{j=1}^{k} v_j A_j(t)$, where $|\vec{t}|$ denotes the Euclidean norm of vector $\vec{t}$.

We chose these functions because they are widely used in Web, streaming and multimedia applications that require ranking and they can be efficiently pipelined using the techniques we present in this paper. The vector $\vec{v} = (v_1, \ldots, v_k)$ is called the *preference vector* of the query (view) and each coordinate of the vector is called *attribute preference*. We use $f_v(.)$ to indicate that f_v is a preference function with preference vector $\vec{v}$. Without loss of generality, we assume that attribute preferences are normalized in $[0, 1]$ and that $\sum_{j=1}^{k} v_j = 1$. This assumption is not restrictive, since whatever the range of attribute preferences would be, they can always be normalized instantly by the system. Moreover, we choose to adopt such a normalization since we believe it is in agreement with the notion of preference. The total preference of a user is 1 and the preference on individual attributes is expressed as an additive term towards the total preference.

Each source S_i supports a set Q_i of preference queries over the exported relation R_i. We will focus on the case where each source exports a single ranking function. The metabroker supports preference queries over R that have the same kind of preference function with the underlying sources, which also have the same kind of preference functions with each other. We need efficient pipelined execution of the queries on R. In particular we want to be able to derive the top-m tuples $[t_q^1, \ldots, t_q^m]$ from relation R according to query q by reading a minimal subset of each relation R_i.

4 How to Pipeline a Ranked Query at the Metabroker

The *PipelinedMerge* algorithm presented next efficiently queries the underlying sources $S_1, \ldots, S_n$ and merges their results, when a query q is presented to the metabroker. It outputs the results in a pipelined manner, i.e. without retrieving the complete result sequences from the sources. We assume that each source S_i exports exactly one query v_i. The key to the algorithm is the computation of a prefix $R^1_{v_i}$ of the result sequence R_{v_i} of source S_i that is sufficient to assure that the first tuple of R_i according to f_q is in $R^1_{v_i}$. The *Watermark* vector definition presents the intuition behind the *PipelinedMerge* algorithm described next.

Definition 1 (First Watermark Vector) *Consider*

- *a set of sources $S_1, \ldots, S_n$ consisting of the functions $f_{v_1}, \ldots, f_{v_n}$ applied on relations $R_1, \ldots, R_n$ respectively,*
- *the query q consisting of the function f_q applied on the relation R and*
- *a tuple $t^{top} \in R$*

The first watermark vector $(T^{top}_{v_1,q}, \ldots, T^{top}_{v_n,q})$ has the following property:

$$\forall i \in 1, \ldots, n, \forall t \in R_i, f_{v_i}(t) < T^{top}_{v_i,q} \Rightarrow f_q(t) < f_q(t^{top}) \quad (1)$$

When we refer to a source S_i we simply use the term "watermark of S_i" instead of "the i-th coordinate of the first watermark vector". According to the definition, if a tuple t in the source S_i is below the watermark value $T^{top}_{v_i,q}$ (that is, $f_{v_i}(t) < T^{top}_{v_i,q}$) then t cannot be the top result t^1_q of the query, since at least t^{top} is higher in the query result (according to the property $f_q(t) < f_q(t^{top})$). This implies that $f_v(t^1_q) \geq T^{top}_{v,q}$. Hence, in order to find t^1_q one has to scan R_i from the start and retrieve the prefix $[t^1_{v_i}, t^2_{v_i}, \ldots, t^{w-1}_{v_i}, t^w_{v_i})$, where $t^w_{v_i}$ is the first tuple in R_i with $f_{v_i}(t^w_{v_i}) < T^{top}_{v_i,q}$, i.e., $t^{w-1}_{v_i}$ is the last tuple of R_i that is above the watermark.

We present the *PipelinedMerge* algorithm in two steps. The first step, which is presented in this section, merges the results from the sources in a pipelined manner, given an "oracle" that provides the first watermark vector. The algorithm applies to all types of functions for which such a watermark oracle is feasible as discussed in Section 5. Section 5 provides the computation of the watermark (*DetermineWatermark* function in Figure 2) for the class of functions described in Section 3.

The algorithm *PipelinedMerge* is described in Figure 2. It inputs:

- the preference function $f_{v_1}, \ldots, f_{v_n}$ for each source
- the result sequences $R_{v_1}, \ldots, R_{v_n}$ that the sources produce for these functions
- the user's preference function f_q and
- the number N of desired results

In each iteration, WINDOW stores all tuples t that are candidates to have the maximum $f_q(t)$ value among all tuples in R that have not yet been output. Notice that we choose as t^{top} the tuple with the maximum f_q value from the remaining tuples in WINDOW instead of the next unprocessed tuple in each result sequence R_{v_i} separately. This choice optimizes the algorithm as the windows of tuples selected from each source will be smaller, since the watermark values that depend on t^{top} will be greater. We repeat the steps of the PipelinedMerge algorithm until the user's number of requested results N are output. The theorem below explains why in each iteration we output the first s tuples from WINDOW and not just the first.

Theorem 1 *Consider*

- *the tuples $[t^1_v, \ldots, t^{w-1}_v]$ ranked according to f_v that are above the watermark of t^1_v,*
- *$[t^1_q, \ldots, t^{w-1}_q]$ which is the ranked order, according to q of $[t^1_v, \ldots, t^{w-1}_v]$*
- *s which is the index of t^1_v in $[t^1_q, \ldots, t^{w-1}_q]$, i.e., $t^1_v \equiv t^s_q$*

Then $t^1_q, \ldots, t^s_q$ are the tuples with the highest rank in the answer of q.

```
Algorithm PipelinedMerge(R_{v_1},...,R_{v_n}, f_q, f_{v_1},...,f_{v_n}, N){
for i = 1 to n do {
  Retrieve first tuple t_{v_i}^{top} from R_{v_i} and compute f_q(t_{v_i}^{top})
  }
Let t^{top} be the tuple t_{v_i}^{top} that has the maximum f_q(t_{v_i}^{top})
while (less than N tuples in the output) {
  Let (T_{v_1,q}^{top},...,T_{v_n,q}^{top}) = DetermineWatermarkVector(t^{top}, v_1,...,v_n, q)
  for i = 1 to n do {
    Scan R_{v_i} and determine the first tuple t_w with f_{v_i}(t_w) < T_{v_i,q}^{top}
    Add all non-processed tuples up to t_w to temporary relation WINDOW
  }
  Sort WINDOW by f_q and let s be the index of t^{top} in WINDOW
  Output and delete the first s tuples from WINDOW
  if WINDOW is empty then
    Add to WINDOW the first unprocessed tuple from each source
    Sort WINDOW by f_q
  Let t^{top} be the first tuple in WINDOW.
  }

}
```

Figure 2: Algorithm to output the first N tuples according to q

Proof: Clearly $f_q(t_q^1) \geq \ldots \geq f_q(t_q^{w-1})$. Moreover due to the watermark property (Equation 1) $\forall t, f_v(t) < T_{v,q}^1 \Rightarrow f_q(t) \leq f_q(t_q^s)$. The theorem follows, since $f_q(t_q^s) \leq f_q(t_q^{s-1}) \leq \ldots \leq f_q(t_q^1)$. ∎

Theorem 2 (Correctness) *Algorithm PipelinedMerge outputs the correct ranked results.*

Proof: First we prove that the top tuple according to f_q of a single source S_i is contained in the window of tuples from R_i ending at the last tuple that has a smaller or equal f_{v_i} value to the watermark value $T_{v_i,q}^{top}$. From the definition of the watermark vector we see that all tuples t below the watermark value have $f_q(t) < f_q(t^{top})$. That is, none of these tuples could be the top one according to f_q. Hence the top tuple according to f_q is in the retrieved window.

So in each iteration of the *PipelinedMerge* algorithm (each iteration of the while loop) the top tuple according to q from each source is contained in the tuples that are added to WINDOW. Hence the top tuple according to f_q over the union of all relations will be in WINDOW and will be output. ∎

Theorem 3 *There is no other pipelined algorithm that retrieves the same tuples from relations $R_{v_1},\ldots,R_{v_n}$ with PipelinedMerge algorithm in each iteration and outputs more or the same results in every iteration (and there is at least one iteration that it outputs more).*

Proof: Assume that there was such an algorithm A'. Assume that at some iteration A' outputs one more tuple t' than *PipelinedMerge* algorithm and exactly the same number of tuples in the previous iterations. Given that *PipelinedMerge* algorithm outputs all tuples t for which $f_q(t) \geq f_q(t^{top})$, it follows that $f_q(t') < f_q(t^{top})$. Also notice that t^{top} is the same tuple in both algorithms since the same number of tuples have been output in all previous iterations. But nothing can guarantee that there is no tuple t'' in a source that has not been retrieved yet and has $f_q(t'') > f_q(t')$ (recall that since we pick the tuple with the biggest f_q value as t_{top}, the Watermarks provide the tightest bounds for all f_i's). So Algorithm A' is not correct. ∎

Example Let us present an example of the algorithm's operation. Assume q is a query with $\vec{q} = (0.1, 0.6, 0.3)$ and there are two sources S_1 and S_2 that produce the result sequences R_{v_1} and R_{v_2} respectively. Their preference vectors are $\vec{v_1} = (0.2, 0.4, 0.4)$ and $\vec{v_2} = (0, 0.5, 0.5)$ respectively. The query and the sources

tuple	A1	A2	A3	$f_{v_1}(t)$	$f_q(t)$
t_2	10	17	20	16.8	17.2
t_1	20	20	11	16.4	17.3
t_6	15	10	5	9	9
t_7	12	5	5	6.4	5.7

(a) R_{v_1}

tuple	A1	A2	A3	$f_{v_2}(t)$	$f_q(t)$
t_3	17	18	12	15	16.1
t_4	5	10	12	11	10.1
t_5	15	10	8	9	9.9

(b) R_{v_2}

Figure 3: Sources S_1, S_2 and scores of each tuple based on f_{v_1}, f_{v_2} and f_q

have linear preference functions. Let $m_1 = m_2 = m_3 = 5$ and $M_1 = M_2 = M_3 = 20$. The sequences R_{v_1} and R_{v_2} are shown in Figure 3.

First we retrieve the first tuple from each sequence and find the one that has the maximum f_q value. It is $f_q(t_2) > f_q(t_3)$, so $t^{top} \equiv t_2$. We calculate the watermark vector $(T^{top}_{v_1,q} = T^{top}_{v_2,q}) = (14.26, 15.33)$. The calculation of the watermark is described in Section 5. Hence t_2 and t_1 are added to WINDOW because $f_{v_1}(t_1) > T^{top}_{v_1,q}$. We sort the WINDOW by f_q and output t_1 and t_2, which have bigger or equal f_q values than t^{top}. Next we add two fresh tuples t_6 and t_3, because WINDOW is empty, and t_3 becomes t^{top}. We calculate the watermark vector $(T^{top}_{v_1,q} = T^{top}_{v_2,q}) = (13.1, 13.5)$. So no tuples are added to WINDOW and t_3 is output. Now $t^{top} \equiv t_6$. The algorithm continues and outputs t_4, t_5, t_6 and t_7.

5 Watermark Calculation

In this section we present algorithms for the calculation of the watermark vector for the three types of preference functions described in Section 3. Each coordinate of the watermark vector is calculated independently from the others. Thus we describe the *DetermineWatermark* algorithm, which is the main component of the *DetermineWatermarkVector* algorithm used in *PipelinedMerge* algorithm. Notice that we assume that all sources rank their objects using the same kind of preference functions as the query, but with different preference vectors.

The calculation of the watermark vector is feasible when the preference functions of the query and the sources are monotone for each attribute value. That is, for all $f \in \{f_q, f_{v_1}, \ldots, f_{v_n}\}$, $f(A_1(t_1), \ldots, A_k(t_1)) \leq f(A_1(t_2), \ldots, A_k(t_2))$ when $A_i(t_1) \leq A_i(t_2)$ for $i \in 1, \ldots, k$.

The inputs of the *DetermineWatermark* algorithm are the user function f_q, a source's function f_v and a tuple t^{top}. Consider that the highest possible $f_v(t)$ is achieved for an imaginary tuple t'. Thus we will determine the maximimum $f_v(t')$ value while satisfying the following equation and thus the watermark.

$$f_q(t') < f_q(t^{top}) \tag{2}$$

5.1 Watermark computation for Linear functions

We will now use Equation 1 to determine the watermark value $T^{top}_{v,q}$ in the case of linear functions f_q and f_v. We assume that view R_v is ordered by decreasing values of the score of f_v. Thus we will determine the tuple t' that maximizes $f_v(t')$ while satisfying $f_q(t') < f_q(t^{top})$. Since we know the values of t^{top}, $\vec{q} \equiv (q_1, \ldots, q_k)$ and $\vec{v} \equiv (v_1, \ldots, v_k)$, we need to come up with bounds for the values of $t \equiv (A_1(t), \ldots A_k(t))$ using the known parameters to maximize $f_v(t')$ while satisfying the inequality of Equation 1 for all $t \in R$. We will subsequently use these bounds to derive the watermark. Let us express $f_q(t) = \sum_{i=1}^{k} q_i A_i(t)$ as a function of $f_v(t) = \sum_{i=1}^{k} v_i A_i(t)$. Thus,

$$f_q(t) = \sum_{i=1}^{k} q_i A_i(t) = f_v(t) + \sum_{i=1}^{k} (q_i - v_i) A_i(t) \tag{3}$$

By substituting Equation 3 into Equation 1 we get

$$\forall t \in R, f_v(t) \leq T^{top}_{v,q} \Rightarrow f_v(t) + \sum_{i=1}^{k} (q_i - v_i) A_i(t) \leq f_q(t^{top}) \tag{4}$$

$$A_i(t') = \begin{cases} min(\frac{f_v(t') - \sum_{j<>i}^{k} v_j m_j}{v_i}, M_i) & q_i > v_i <> 0 \\ M_i & q_i > v_i = 0 \\ 0 & q_i = v_i \\ max(\frac{f_v(t') - \sum_{j<>i}^{k} v_j M_j}{v_i}, m_i) & q_i < v_i \end{cases} \quad (8)$$

Figure 4: Bounds for A_i

Consider that the highest possible $f_v(t)$ is achieved for t'. It is:

$$f_v(t') + \sum_{i=1}^{k} (q_i - v_i) A_i(t') \leq f_q(t^{top}) \quad (5)$$

We will treat Equation 5 as equality; since the left side of Equation 5 is linear on $f_v(t')$, the corresponding inequality is trivially satisfied. Since out objective is to determine the maximimum $f_v(t')$ value that satisfies Equation 5, which is linear in $f_v(t')$, we will determine bounds for each attribute $A_i(t')$ in a way that the left part of Equation 5 is maximized. We determine the bounds for each attribute $A_i(t')$, by the following case analysis. Recall also that each attribute A_i has domain $[m_i, M_i]$.

- $(q_i - v_i) > 0$ and $v_i <> 0$: In this case we have that

$$A_i(t') = \frac{f_v(t') - \sum_{j<>i}^{k} v_j A_j(t')}{v_i} \leq \frac{f_v(t') - \sum_{j<>i}^{k} v_j m_j}{v_i} \quad (6)$$

  We set $U_i = \frac{f_v(t') - \sum_{j<>i}^{k} v_j m_j}{v_i}$. Since $A_i(t') \leq M_i$, we have that $A_i(t') = min(U_i, M_i)$.

- $(q_i - v_i) > 0$ and $v_i = 0$: then $A_i(t') = M_i$

- $(q_i - v_i) = 0$: we can ignore this term

- $(q_i - v_i) < 0$ and $v_i <> 0$: In this case we have that:

$$A_i(t') = \frac{f_v(t') - \sum_{j<>i}^{k} v_i A_j(t')}{v_i} \geq \frac{f_v(t') - \sum_{j<>i}^{k} v_j M_j}{v_i} \quad (7)$$

  We set $L_i = \frac{f_v(t') - \sum_{j<>i}^{k} v_j M_j}{v_i}$. Since $A_i(t') \geq m_i$, we have that $A_i(t') = max(L_i, m_i)$.

Figure 4 summarizes the results of our analysis for each attribute value $A_i(t')$. Notice that we use the notation $A_i(t')$ to denote the bound for the value of attribute A_i. Also notice that when $(q_i - v_i) > 0$ we determine an upper bound for the value of $A_i(t')$ whereas when $(q_i - v_i) < 0$ we determine a lower bound. The main difficulty in solving Equation 5 directly, lies on the existence of min and max terms, with two operands each, in the expressions derived for the attribute bounds (Figure 4). [HKP01] presents an efficient methodology for solving this kind of equations. We include this methodology in the Appendix for completeness.

5.2 Watermark computation for Logarithmic functions

We follow a procedure similar to the one for linear functions. First we express $f_q(t) = \sum_{i=1}^{k} q_i log(A_i(t))$ as a function of $f_v(t) = \sum_{i=1}^{k} v_i log(A_i(t))$. Thus,

$$f_q(t) = \sum_{i=1}^{k} q_i log(A_i(t)) = f_v(t) + \sum_{i=1}^{k} (q_i - v_i) log(A_i(t)) \quad (9)$$

$$log(A_i(t')) = \begin{cases} min(\frac{f_v(t') - \sum_{j<>i}^{k} v_j log(m_j)}{v_i}, log(M_i)) & q_i > v_i <> 0 \\ log(M_i) & q_i > v_i = 0 \\ 0 & q_i = v_i \\ max(\frac{f_v(t') - \sum_{j<>i}^{k} v_j log(M_j)}{v_i}, log(m_i)) & q_i < v_i \end{cases} \quad (12)$$

Figure 5: Bounds for $log(A_i)$

By substituting Equation 9 into Equation 1 we get

$$\forall t \in R, f_v(t) \leq T_{v,q}^{top} \Rightarrow f_v(t) + \sum_{i=1}^{k} (q_i - v_i) log(A_i(t)) \leq f_q(t^{top}) \quad (10)$$

Let t' be the tuple that maximizes $f_v(.)$ while satisfying Equation 1.

$$f_v(t') + \sum_{i=1}^{k} (q_i - v_i) log(A_i(t')) \leq f_q(t^{top}) \quad (11)$$

Now the key idea is to upper bound the $(q_i - v_i)log(A_i(t'))$ terms. We perform a worst-case analysis by producing the $A_i(t')$ values that maximize the left hand side, i.e., turn Equation 11 into an equality. The resulting bounds are shown in Figure 5. The algorithm continues as in the linear functions' case.
Notice that we assume that m_i and M_i are not less than 1, which is very reasonable since applications that use logarithms typically have integer attribute values. Consider for example an attribute being the number of occurences of a keyword in a document, in which case the logarithm would prevent someone from writing a word a huge number of times on a Web page in order to receive high ranking. The watermark is then computed by plugging the bound values to Equation 11. Notice that the equation will contain min and max functions that contain the unknown $f_v(t')$. An algorithm that solves this equality in an efficient way is shown in the Appendix .

5.3 Watermark computation for Cosine functions

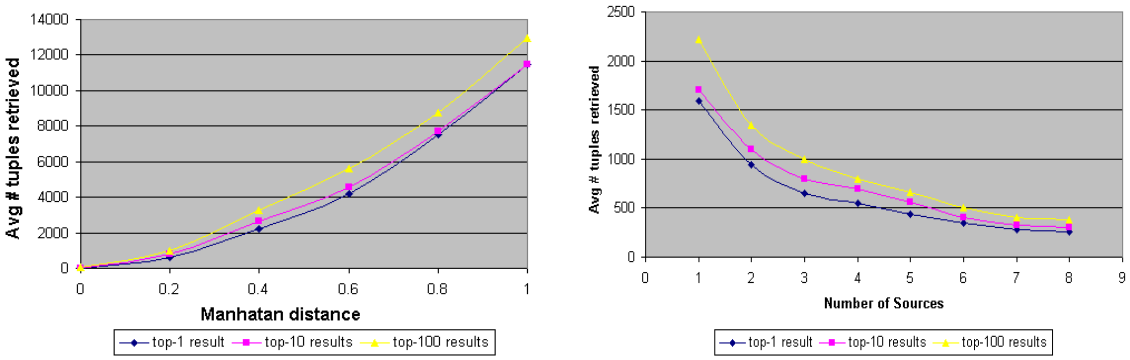
The calculation of the watermark for a cosine function differs from the linear and logarithmic cases because we can not have a separate term in the preference function for each attribute. The reason for this is the $|\vec{t}|$ term in $f(t) = \frac{1}{|\vec{t}||\vec{v}|} \sum_{j=1}^{k} v_j A_j(t)$. This means that we need to calculate an upper bound for $f_q(t) - f_v(t)$ as a whole. Hence Equation 1 can be written as:

$$f_v(t') + (f_q(t') - f_v(t')) \leq f_q(t^{top}) \quad (13)$$

where t' is the tuple that maximizes $f_v(t')$ while satisfying Equation 13 . Now we need to find an upper bound for $f_q(t') - f_v(t')$ to plug it in Equation 13.

$$f_q(t') - f_v(t') = \frac{\vec{q} \cdot \vec{t'}}{|\vec{q}||\vec{t'}|} - \frac{\vec{v} \cdot \vec{t'}}{|\vec{v}||\vec{t'}|} = \frac{\vec{t'}}{|\vec{t'}|}(\frac{\vec{q}}{|\vec{q}|} - \frac{\vec{v}}{|\vec{v}|}) \quad (14)$$

Since $(\frac{\vec{q}}{|\vec{q}|} - \frac{\vec{v}}{|\vec{v}|})$ is fixed for every pair of functions and $\frac{\vec{t}}{|\vec{t}|}$ has size 1, the above expression is maximized when $\vec{t}$ is parallel to $(\frac{\vec{q}}{|\vec{q}|} - \frac{\vec{v}}{|\vec{v}|})$ and its maximum value is $|\frac{\vec{q}}{|\vec{q}|} - \frac{\vec{v}}{|\vec{v}|}|$. This means that the watermark value for a tuple t^{top} is $f_v(t') = f_q(t^{top}) - |\frac{\vec{q}}{|\vec{q}|} - \frac{\vec{v}}{|\vec{v}|}|$.



(a) Varying Manhattan distance

(b) Varying number of sources

Figure 6: Experimental Results

6 Experimental Results

The performance of $PipelinedMerge$ depends on the number of sources and on the similarity of the function f_q, which is used by the user query, to the functions $f_{v_1}, \ldots, f_{v_n}$ supported by the sources. We performed two sets of experiments. First we measured the average prefix of the source relations as a function of the Manhattan distance MD between the preference vectors of the user query and the source queries ($MD = \sum_{j=1}^{k} |q_j - v_{i_j}|$). For simplicity in the report, we assumed the distance of the query from each source is the same. Then we measured again this prefix as a function of the number of sources.

The experiments use a synthetic dataset that represents house information. Each house has four attributes, price, bedrooms, bathrooms, square feet with corresponding cardinalities 1000000, 10, 8 and 3500. The attributes take random values within their domain. Every source's relation has 50,000 tuples. We use a discretization of 0.05 for the domain from which we draw source and query preference vectors (0 through 1, in increments of 0.05). Since we work with four attributes the maximum Manhattan distance of the query preference vector from each source preference vector is 2. Linear functions were used for all experiments.

Prefix size for varying distance between metabroker and sources queries. Our first experiment assesses the average prefix size that we need to retrieve from each of the underlying sources in order to present to the user of the metabroker the top 1, 10 and 100 results, as a function of the Manhattan distance between the metabroker query and the source queries. Notice that the Manhattan distance between any two source queries can be at most two times the distance between the metabroker and the source queries. We use four sources for this experiments and the resulting graph is shown in Figure 6 (a). We see a slightly superlinear performance deteriotation due to the fact that the preference scores have a concentration towards the "average" score.

Prefix size for varying number of sources. The second experiment measures the average prefix size that we need to retrieve from each of the underlying sources in order to present to the user of the metabroker the top 1, 10 and 100 results, as a function of the number of sources that are used. The Manhattan distance between the metabroker query and the source queries is fixed to 0.2. The result is shown in Figure 6 (b).

7 Conclusions

The increasing number of data sources and search engines that rank their objects according to a function of their attributes creates the need to find efficient ways of combining the results from multiple such sources on a meta-broker. We have presented an algorithm that performs the merging of ranked results from multiple sources in a pipelined manner, in the case where the preference functions used by the sources and the user are (i) linear, (ii) cosine, or (iii) linear combinations of the logarithms of the attributes. The algorithm can

be extended to other classes of functions as well, assuming corresponding watermark determination routines are built for those functions.

References

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. *Proceedings of ACM SIGMOD*, 1996.

[CLC95] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. *Proceedings of the* 18^{th} *Annual SIGIR Conference*, 1995.

[Fag96] R. Fagin. Combining fuzzy information from multiple systems. *ACM Symposium on Principles of Database Systems*, 1996.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. *Proceedings of ACM PODS, Santa Barbara CA*, May 2001.

[GBK00] U. Guntzer, W. Balke, and W. Kiessling. Optimizing multi-feature queries in image databases . *Proceedings of Very Large Databases (VLDB)*, 2000.

[GBK01] U. Guntzer, W. Balke, and W. Kiessling. Towards Efficient Multi-Feature Queries in Heterogeneous Environments . *Proceedings of IEEE Intenational Conference of Information Technology (ITCC)*, 2001.

[GCMP97] L. Gravano, C. K. Chang, H. G. Molina, and A. Paepcke. STARTS:Stanford proposal for Internet meta-searcing. *Proceedings of ACM SIGMOD*, May 1997.

[GM97] L. Gravano and H. G. Molina. Merging Ranks from Heterogeneous Internet Sources. *Proceedings of VLDB, Athens Greece*, 1997.

[HKP01] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER:A System for the Efficient Execution of Multi-parametric Ranked Queries. *Proceedings of ACM SIGMOD, Santa Barbara CA*, May 2001.

[NR99] S. Nepal and M. Ramakrishna. Query processing issues in image (multimedia) databases . *Proceedings of International Conference on Data Engineering*, 1999.

[VGJL95] E. M. Voorhees, N. K. Gupta, and B. Johnson-Laird. The collection fusion problem. *Proceedings of* 3^{rd} *Text Retrieval Conference (TREC-3)*, 1995.

8 Appendix: Solving systems of inequalities with min, max terms (from [HKP01])

The main difficulty in solving Equation 5 directly, lies on the existence of *min* and *max* terms, with two operands each, in the expressions derived for the attribute bounds (Figure 4). Each *min* (equivalently *max*) term however, is linear on $f_v(t')$ thus it is easy to determine for which range of $f_v(t')$ values, each operand of *min* (equivalently *max*) applies, by determining the $f_v(t')$ value that makes both operands equal. Assume the expression for attribute bound $A_i(t')$ contains a *min* or a *max* term. Let e_i be the value for $f_v(t')$ that makes both operands of *min* or *max* equal. As $f_v(t')$ varies, we now know exactly which operand in each *min* or *max* term we should use to determine a bound on the attribute value. Since both U_i and L_i terms are linear on $f_v(t')$, we observe weather $f_v(t')$ lies on the left or right or e_i. There are at most k attribute bound expressions and thus $1 \leq i \leq k$. Possible values of $f_v(t')$ range between $\sum_{i=1}^{k} v_i m_i$ and $\sum_{i=1}^{k} v_i M_i$. If we order the e_i's, we essentially derive a partitioning of the range of possible values of $f_v(t')$ in $k+1$ intervals, $I_i, 1 \leq i \leq k+1$. For each value of $f_v(t')$ in these intervals the expressions used to compute each attribute bound are fixed and do not involve *min* or *max*.

We construct a table E having $k+1$ columns, denoting the value intervals for $f_v(t')$ and k rows, denoting the expressions for each attribute bound. For each entry $E(i,j), 1 \leq i \leq k, 1 \leq j \leq k+1$ in this table we record the exact expression that we will use to determine the bound for attribute A_i. If an attribute bound expression is not a function of $f_v(t')$ we can just record the value in the suitable entry as a constant. Once the table is populated, for each value of $f_v(t')$ we know the attribute bound formulas that comprise the left hand side of Equation 5. Thus we have $k+1$ possible expressions for the left side of Equation 5. Each expression, $E_j, 1 \leq j \leq k+1$ is produced by:

$$E_j = f_v(t') + \sum_{i=1}^{k} (q_i - v_i) E(i,j) \tag{15}$$

Theorem 4 *Setting $E_j = f_q(t_v^1), 1 \leq j \leq k+1$ and solving for $f_v(t')$ determines the watermark value.*

Proof: For each j two possibilities exist: (a) the $f_v(t')$ value computed does not fall in the j-th interval. In this case, the expression for E_j cannot yield $f_q(t_v^1)$ since E_j produces an upper bound for $f_q(t)$ by construction, (b) $f_v(t')$ falls in the j-th range. Since $E_j = f_q(t_v^1)$ is a linear function and has a unique solution in range j, $f_v(t')$ is the watermark $T_{v,q}^1$. Note that the range of possible values for $f_v(t')$ is the same with the range of possible values for E_j, thus j will always be identified.

Algorithm *DetermineWatermark* is shown in figure 7. The algorithm assumes that table E has been computed in a preprocessing step. The algorithm uses $O(k^2)$ space and determines the watermark solving k equations in the worst case.

```
Algorithm DetermineWatermark(tuple t_v^1) {
    for j from k+1 downto 1 {
      Solve E_j = f_q(t_v^1) and determine watermark
      if watermark ∈ I_j return watermark
}
}
```

Figure 7: Algorithm *DetermineWatermark*