Combining Fuzzy Information: an Overview*

Ronald Fagin
IBM Almaden Research Center
650 Harry Road
San Jose, California 95120-6099

email: fagin@almaden.ibm.com
http://www.almaden.ibm.com/cs/people/fagin/

Abstract

Assume that each object in a database has m grades, or scores, one for each of m attributes. For example, an object can have a color grade, that tells how red it is, and a shape grade, that tells how round it is. For each attribute, there is a sorted list, which lists each object and its grade under that attribute, sorted by grade (highest grade first). Each object is assigned an overall grade, that is obtained by combining the attribute grades using a fixed monotone *aggregation function*, or *combining rule*, such as min or average. In this overview, we discuss and compare algorithms for determining the top k objects, that is, k objects with the highest overall grades.

1 Introduction

It is becoming increasingly important for database systems to be able to access multimedia data, such as images and video, from a variety of data repositories. Such data is inherently fuzzy. For example, we do not say that a given image is simply either "red" or "not red". Instead, there is a degree of redness, which ranges between 0 (not at all red) and 1 (totally red). In response to a query asking for red objects, a multimedia system might typically assign such a redness score to each object. The result of the query is then a "graded" (or "fuzzy") set [Zad69]. A graded set is a set of pairs (x, g), where x is an object, and g (the grade) is a real number in the interval $[0, 1]$. Graded sets are usually presented in sorted order, sorted by grade.

What graded set should correspond to a compound query, that might ask, for example, for objects that are both red and round? Here we make use of an *aggregation function* (or *scoring function*), that combines the redness score and the roundness score to obtain an overall score. Among the most commonly used aggregation functions are the min (the standard aggregation function corresponding to a conjunction in fuzzy logic [Zad69]) and the average. There is a large literature on choices for the aggregation function (see Zimmermann's textbook [Zim96] and the discussion in [Fag99]).

Assume that there are m attributes, and that the aggregation function is the m-ary function t. If $x_1, \ldots, x_m$ (which we usually assume are each in the interval $[0, 1]$) are the grades of object R under each of the m attributes, then $t(x_1, \ldots, x_m)$ is the (overall) grade of object R. We shall often abuse notation and write $t(R)$ for the grade $t(x_1, \ldots, x_m)$ of R. We say that an aggregation function t is *monotone* if $t(x_1, \ldots, x_m) \le t(x'_1, \ldots, x'_m)$ whenever $x_i \le x'_i$ for every i. Certainly monotonicity is a reasonable

*Database Principles Column. Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada, email libkin@cs.toronto.edu.

property to demand of an aggregation function: if for every attribute, the grade of object R' is at least as high as that of object R, then we would expect the overall grade of R' to be at least as high as that of R. We shall restrict our attention to monotone aggregation functions. Our goal is to find the top k objects for some fixed choice of k, that is, the k objects R with the highest overall grades $t(R)$. In this paper, we shall discuss and compare algorithms for finding the top k objects.

In Section 2, we discuss the kind of database systems we are dealing with, which are middleware. In Section 3, we formally define our setup. In Section 4, we present three algorithms for obtaining the top k answers, namely the "naive algorithm", "Fagin's Algorithm", and the "threshold algorithm". In Section 5, we present an approximate version of the threshold algorithm, which is appropriate when we care only about the "approximately" top k answers. In Section 6, we consider situations where sorted access is impossible for certain of the sorted lists. In Section 7, we present algorithms that are appropriate when random accesses are expensive or impossible. In Section 8, we give a summary.

2 Middleware

Because of the many varieties of data that a multimedia database system must handle, such a system may often really be "middleware". That is, the system is "on top of" various subsystems, and integrates results from the subsystems. A single query to a middleware system can access data in a number of different subsystems. An example of a nontraditional subsystem that a middleware system might access is QBIC [NBE+93], which can search for images by various visual characteristics such as color and texture.

What can we assume about the interface between a middleware system and a subsystem such as QBIC? In response to a query, such as a query that asks for red objects, we can assume that the subsystem will output the graded set consisting of all objects, one by one, each along with its grade, in sorted order based on grade, until the middleware system tells the subsystem to stop. Then the middleware system could later tell the subsystem to resume outputting the graded set where it left off. Alternatively, the middleware system could ask the subsystem for, say, the top 10 objects in sorted order, each along with its grade, then request the next 10, etc. We refer to such types of access as "sorted access".

There is another way that we could expect the middleware system to interact with the subsystem. Specifically, the middleware system could ask the subsystem the grade under a given attribute (such as redness) of any given object. We refer to this as "random access".

Because of the limited modes of access to subsystems, issues of efficient query evaluation in a middleware system are very different from those in a traditional database system. In fact, it is not even clear what "efficient" means in a middleware system.

We now give the performance cost of an algorithm, as defined in [Fag99]. This measure of cost corresponds intuitively to the amount of information that an algorithm obtains from the database. The *sorted access cost* is the total number of objects obtained from the database under sorted access. For example, if there are only two lists, and some algorithm requests altogether the top 100 objects from the first list and the top 20 objects from the second list, then the sorted access cost for this algorithm is 120. Similarly, the *random access cost* is the total number of objects obtained from the database under random access. The *middleware cost* is taken to be $c_S \cdot s + c_R \cdot r$, where s is the sorted access cost, r is the random access cost, and where c_S and c_R are positive constants.[1] The middleware cost is not a measure of total system cost, since it ignores the costs inside of a "black box" like QBIC. There are situations (such as in the case of a query optimizer) where we want a more comprehensive cost measure. Finding such cost measures is an interesting issue, that requires further study.

[1] In [Fag99], the middleware cost is taken for convenience to be simply $s + r$, the sum of the sorted access cost and the random access cost. It is then noted that these two notions of middleware cost (namely, $s + r$ and $c_S \cdot s + c_R \cdot r$) are within constant multiples of each other.

As in [Fag99], we shall identify a query with a choice of the aggregation function t. The user is typically interested in finding the *top k answers*, where k is a given parameter (such as $k = 1$, $k = 10$, or $k = 100$). This means that we want to obtain k objects (which we may refer to as the "top k objects") with the highest grades on this query, along with their grades (ties are broken arbitrarily). For convenience, throughout this paper we will think of k as a constant value, and we will consider algorithms for obtaining the top k answers in databases that contain at least k objects.

3 The Model

We now describe the model more formally. We assume that each database consists of a finite set of *objects*. We shall typically take N to represent the number of objects. Associated with each object R are *m fields* $x_1, \ldots, x_m$, where $x_i \in [0, 1]$ for each i. We may refer to x_i as the ith field of R. The database can be thought of as consisting of a single relation, where one column corresponds to the object id, and the other columns correspond to m attributes of the object. Alternatively, the way we shall think of a database in this paper is as consisting of m sorted lists $L_1, \ldots, L_m$, each of length N (there is one entry in each list for each of the N objects). We may refer to L_i as *list i*. Each entry of L_i is of the form (R, x_i), where x_i is the ith field of R. Each list L_i is sorted in descending order by the x_i value. We take this simple view of a database, since this view is all that is relevant, as far as our algorithms are concerned. We are taking into account only access costs, and ignoring internal computation costs. Thus, in practice it might well be expensive to compute the field values, but we ignore this issue here, and take the field values as being given.

We consider two modes of access to data. The first mode of access is sorted (or sequential) access. Here the middleware system obtains the grade of an object in one of the sorted lists by proceeding through the list sequentially from the top. Thus, if object R has the ℓth highest grade in the ith list, then ℓ sorted accesses to the ith list are required to see this grade under sorted access. The second mode of access is random access. Here, the middleware system requests the grade of object R in the ith list, and obtains it in one random access. If there are s sorted accesses and r random accesses, then the *sorted access cost* is $c_S s \cdot s$, the *random access cost* is $c_R \cdot r$, and the *middleware cost* is $c_S \cdot s + c_R \cdot r$ (the sum of the sorted access cost and the random access cost), for some positive constants c_S and c_R.

4 Algorithms

Our focus in this paper is on algorithms for determining the top k answers. In this section, we present three algorithms.

4.1 The Naive Algorithm

There is an obvious naive algorithm for obtaining the top k answers. Under sorted access, it looks at every entry in each of the m sorted lists, computes (using t) the overall grade of every object, and returns the top k answers. The naive algorithm has linear middleware cost (linear in the database size), and thus is not efficient for a large database.

4.2 Fagin's Algorithm

The first algorithm for our problem to appear in the literature[2] is now often called "Fagin's Algorithm" [Fag99], which we shall modestly refer to as FA. Chaudhuri and Gravano [CG96] consider ways to

[2] This algorithm first appeared in the PODS '96 conference version of [Fag99].

simulate FA by using "filter conditions", which might say, for example, that the color score is at least 0.2. FA works as follows.

1. Do sorted access in parallel to each of the m sorted lists L_i. (By "in parallel", we mean that we access the top member of each of the lists under sorted access, then we access the second member of each of the lists, and so on.) Wait until there are at least k "matches", that is, wait until there is a set H of at least k objects such that each of these objects has been seen in each of the m lists.

2. For each object R that has been seen, do random access as needed to each of the lists L_i to find the ith field x_i of R.

3. Compute the grade $t(R) = t(x_1, \ldots, x_m)$ for each object R that has been seen. Let Y be a set containing the k objects that have been seen with the highest grades (ties are broken arbitrarily). The output is then the graded set $\{(R, t(R)) \mid R \in Y\}$.

FA is correct for monotone aggregation functions t (that is, the algorithm successfully finds the top k answers) [Fag99]. If there are N objects in the database, and if the orderings in the sorted lists are probabilistically independent, then the middleware cost of FA is $O(N^{(m-1)/m}k^{1/m})$, with arbitrarily high probability [Fag99]. We shall not discuss the probability model of [Fag99], including the notion of "independence", since it is off track, and is not needed for our subsequent discussions of other algorithms.

An aggregation function t is *strict* [Fag99] if $t(x_1, \ldots, x_m) = 1$ holds precisely when $x_i = 1$ for every i. Thus, an aggregation function is strict if it takes on the maximal value of 1 precisely when each argument takes on this maximal value. We would certainly expect an aggregation function representing the conjunction to be strict (see the discussion in [Fag99]). In fact, it is reasonable to think of strictness as being a key characterizing feature of the conjunction. It can be shown that the algorithm FA is optimal with high probability in the worst case if the aggregation function is strict, and if the orderings in the sorted lists are probabilistically independent [Fag99].

4.3 The Threshold Algorithm

Although FA is optimal in a certain sense, there are many situations where FA performs rather poorly. In [Fag99], it was noted that there are no guarantees when the aggregation function is not strict. As the most obvious example, assume that the aggregation function is constant; in this case, there is a trivial algorithm that gives us the top k answers (any k objects will do) with $O(1)$ middleware cost. By contrast, FA, whose access pattern is oblivious to the choice of aggregation function, has $\Omega(N^{(m-1)/m}k^{1/m})$ middleware cost. As a more interesting example, when the aggregation function is max (which is not strict), it is shown in [Fag99] that there is a simple algorithm that makes at most mk sorted accesses and no random accesses that finds the top k answers.

The problems with FA run deeper than simply the fact that it may perform badly when the aggregation function is not strict. Even when the aggregation function is strict, there may be some databases where FA is much too conservative. Several groups [NR99, GBK00, FLN01] independently found a new algorithm, which is called the "threshold algorithm" in [FLN01]. As we shall discuss, the threshold algorithm is essentially optimal (up to a constant factor) over every database.

Amnon Lotem first defined TA, and did extensive simulations comparing it to FA, as a project in a database course taught by Michael Franklin at the University of Maryland–College Park, in the Fall of 1997. A few years later, Michael Franklin brought the existence of this work to the attention of the author. This led to a collaboration that produced the paper [FLN01]. Nepal and Ramakrishna [NR99] were the first to publish an algorithm that is equivalent to TA. They essentially restricted attention to the situation where the aggregation function is the min (for more details on their restriction, see [FLN01]). Güntzer, Balke, and Kiessling [GBK00] also define an algorithm that is equivalent to TA. They call this algorithm "Quick-Combine (basic version)" to distinguish it from their algorithm of interest, which they call "Quick-Combine". The difference between these two algorithms is that Quick-Combine provides a heuristic rule that determines which sorted list L_i to do the next sorted access on. The intuitive idea is that they wish to speed up TA by taking advantage of skewed distributions of grades. They do extensive simulations to compare Quick-Combine against FA. Unlike TA, which as we shall discuss, is optimal in a strong sense ("instance optimal"), it turns out that Quick-Combine is not instance optimal.

We now present the threshold algorithm (TA).

1. Do sorted access in parallel to each of the m sorted lists L_i. As an object R is seen under sorted access in some list, do random access to the other lists to find the grade x_i of object R in every list L_i. Then compute the grade $t(R) = t(x_1, \ldots, x_m)$ of object R. If this grade is one of the k highest we have seen, then remember object R and its grade $t(R)$ (ties are broken arbitrarily, so that only k objects and their grades need to be remembered at any time).

2. For each list L_i, let $\underline{x}_i$ be the grade of the last object seen under sorted access. Define the *threshold value* τ to be $t(\underline{x}_1, \ldots, \underline{x}_m)$. As soon as at least k objects have been seen whose grade is at least equal to τ, then halt.

3. Let Y be a set containing the k objects that have been seen with the highest grades. The output is then the graded set $\{(R, t(R)) \mid R \in Y\}$.

TA is correct for each monotone aggregation function t [FLN01]. We now show that the stopping rule for TA always occurs at least as early as the stopping rule for FA (that is, with no more sorted accesses than FA). In FA, if R is an object that has appeared under sorted access in every list, then by monotonicity, the grade of R is at least equal to the threshold value. Therefore, when there are at least k objects, each of which has appeared under sorted access in every list (the stopping rule for FA), there are at least k objects whose grade is at least equal to the threshold value (the stopping rule for TA).

We note that Natsev et al. [NCS+01] observe that the scenario we have been discussing can be thought of as taking joins over sorted lists where the join is over a unique record ID present in all the sorted lists. They generalize by considering arbitrary joins.

Instance Optimality of TA: It is shown in [FLN01] that TA is optimal in a strong sense, which we now define. Let $\mathbf{A}$ be a class of algorithms, let $\mathbf{D}$ be a class of databases, and let $cost(\mathcal{A}, \mathcal{D})$ be the middleware cost incurred by running algorithm $\mathcal{A}$ over database $\mathcal{D}$. An algorithm $\mathcal{B}$ is *instance optimal over* $\mathbf{A}$ *and* $\mathbf{D}$ if $\mathcal{B} \in \mathbf{A}$ and if for every $\mathcal{A} \in \mathbf{A}$ and every $\mathcal{D} \in \mathbf{D}$ we have

$$cost(\mathcal{B}, \mathcal{D}) = O(cost(\mathcal{A}, \mathcal{D})). \tag{1}$$

Equation (1) means that there are constants c and c' such that

$$cost(\mathcal{B}, \mathcal{D}) \leq c \cdot cost(\mathcal{A}, \mathcal{D}) + c' \tag{2}$$

for every choice of $\mathcal{A} \in \mathbf{A}$ and $\mathcal{D} \in \mathbf{D}$. We refer to c as the *optimality ratio*. Intuitively, instance optimality corresponds to optimality in every instance, as opposed to just the worst case or the average case.

FA is optimal in a high-probability worst-case sense under certain assumptions. TA is optimal in a much stronger sense, and without any underlying probabilistic model or probabilistic assumptions: it is instance optimal, for several natural choices of $\mathbf{A}$ and $\mathbf{D}$ [FLN01] (unless otherwise stated, throughout this overview paper, whenever we speak of instance optimality, we shall take $\mathbf{D}$ to be the class of all databases). In particular, an important case where TA is instance optimal occurs when $\mathbf{A}$ is the class of algorithms that would normally be implemented in practice (since the only algorithms that are excluded are those that make *wild guesses*, which are random accesses on an object not previously encountered by sorted access). Instance optimality of TA holds in this case for all monotone aggregation functions. By contrast, as we discussed, high-probability worst-case optimality of FA holds only under the assumption of strictness of the aggregation function.

Another advantage of TA over FA is that TA requires only bounded buffers, whose size is independent of the size of the database. By contrast, FA requires buffers that grow arbitrarily large as the database grows, since FA must remember every object it has seen in sorted order in every list, in order to check for matching objects in the various lists.

5 Turning TA into an Approximation Algorithm, and Allowing Early Stopping

TA can easily be modified to be an *approximation algorithm*. It can then be used in situations where we care only about the *approximately* top k answers. Thus, let $\theta > 1$ be given. Define a *θ-approximation to the top k answers (for t over database $\mathcal{D}$)* to be a collection of k objects (and their grades) such that for each y among these k objects and each z not among these k objects, $\theta t(y) \geq t(z)$. To find a θ-approximation to the top k answers, modify the stopping rule of TA in Step 1 to say "As soon as at least k objects have been seen whose grade is at least equal to τ/θ, then halt." This approximation algorithm is called TA_θ.

If $\theta > 1$ and the aggregation function t is monotone, then TA_θ correctly finds a θ-approximation to the top k answers for t [FLN01]. It is also shown in [FLN01] that when we restrict attention to algorithms that do not make wild guesses, TA_θ is instance optimal.

It is straightforward to modify TA_θ into an interactive process where at all times the system can show the user the current top k list along with a guarantee about the degree of approximation to the correct answer. At any time, the user can decide, based on this guarantee, whether he would like to stop the process. Thus, let β be the grade of the kth (bottom) object in the current top k list, let τ be the current threshold value, and let $\theta = \tau/\beta$. If the algorithm is stopped early, we have $\theta > 1$. It is easy to see that the current top k list is then a θ-approximation to the top k answers. Thus, the user can be shown the current top k list and the number θ, with a guarantee that he is being shown a θ-approximation to the top k answers.

6 Restricting Sorted Access

Bruno, Gravano, and Marian [BGM02] discuss a scenario where it is not possible to access certain of the lists under sorted access. They give a nice example where the user wants to get information about restaurants. The user has an aggregation function that gives a score to each restaurant based on how good it is, how inexpensive it is, and how close it is. In this example, the Zagat-Review web site gives ratings of restaurants, the NYT-Review web site gives prices, and the MapQuest web site gives distances. Only the Zagat-Review web site can be accessed under sorted access (with the best restaurants at the top of the list).

Let Z be the set of indices i of those lists L_i that can be accessed under sorted access. We assume that Z is nonempty, that is, that at least one of the lists can be accessed under sorted access. We take m' to be the cardinality $|Z|$ of Z (and as before, take m to be the total number of sorted lists). As in [FLN01], define TA_Z to be the following natural modification of TA, that deals with the restriction on sorted access. In the case where $|Z| = 1$, algorithm TA_Z is essentially the same as the algorithm TA-Adapt in [BGM02].

1. Do sorted access in parallel to each of the m' sorted lists L_i with $i \in Z$. As an object R is seen under sorted access in some list, do random access as needed to the other lists to find the grade x_i of object R in every list L_i. Then compute the grade $t(R) = t(x_1, \ldots, x_m)$ of object R. If this grade is one of the k highest we have seen, then remember object R and its grade $t(R)$ (ties are broken arbitrarily, so that only k objects and their grades need to be remembered at any time).

2. For each list L_i with $i \in Z$, let $\underline{x}_i$ be the grade of the last object seen under sorted access. For each list L_i with $i \notin Z$, let $\underline{x}_i = 1$. Define the *threshold value* τ to be $t(\underline{x}_1, \ldots, \underline{x}_m)$. As soon as at least k objects have been seen whose grade is at least equal to τ, then halt.[3]

3. Let Y be a set containing the k objects that have been seen with the highest grades. The output is then the graded set $\{(R, t(R)) \mid R \in Y\}$.

TA_Z is correct, and is instance optimal when we restrict attention to algorithms that do not make wild guesses [FLN01].

7 Restricting Random Access

There are some scenarios where the middleware system is not allowed random access to some subsystem. An example might occur when the middleware system is a text retrieval system, and the subsystems are search engines. Thus, there does not seem to be a way to ask a major search engine on the web for its internal score on some document of our choice under a query. To deal with such scenarios, two groups [FLN01, GBK01] have devised modifications of TA that do no random access. The algorithm of [FLN01], which they call NRA ("no random accesses") is instance optimal when we restrict attention to algorithms that do not make random accesses. We shall give the algorithm NRA shortly.

There are other scenarios where random access is not impossible, but simply expensive. An example arises when the costs correspond to disk access (sequential versus random). Also, Wimmers et al. [WHRB99] discuss a number of systems issues that can cause random access to be expensive. Then we would like the optimality ratio to be independent of the ratio c_R/c_S (recall that c_R is the cost of a single random access, and c_S is the cost of a single sorted access). That is, if we allow c_R and c_S to vary, instead of treating them as constants, we would still like the optimality ratio to be bounded. Shortly, we shall give an algorithm (which depends on c_R/c_S) that is a combination of TA and NRA. This algorithm is called CA ("combined algorithm") in [FLN01]. Under natural restrictions on the aggregation function and the class of databases, CA is instance optimal, with optimality ratio independent of c_R/c_S.

In focusing on scenarios where random accesses are expensive or impossible, the criteria are changed in [FLN01] for the desired output. The criterion used so far has been that the output be the "top k answers", which consists of the top k objects, along with their (overall) grades. In the scenarios we are now considering, a weaker requirement is specified in [FLN01], namely, that the output consist of the top k objects, without their grades. The reason is that, since random accesses are expensive or impossible, there are instances where it is much cheaper to find the top k objects without their grades. This is because, as is shown by example in [FLN01], we can sometimes obtain enough partial information about grades to know that an object is in the top k objects without knowing its exact grade. The new requirement, as stated, says only that the output must consist of the top k objects, with no information being given about the sorted order (sorted by grade). If we wish to know the sorted order, this can easily be determined by finding the top object, the top 2 objects, etc.

Before we can present the algorithms NRA and CA, we need to define some notions corresponding to lower and upper bounds on the overall grade an object can attain, based on the information available at the time.

Given an object R and subset $S(R) = \{i_1, i_2, \ldots, i_\ell\} \subseteq \{1, \ldots, m\}$ of known fields of R, with values $x_{i_1}, x_{i_2}, \ldots, x_{i_\ell}$ for these known fields, define $W_S(R)$ to be the minimum (or *worst*) value the aggregation function t can attain for object R. When t is monotone, this minimum value is obtained by substituting

[3] Even though there are at least k objects, it is possible [FLN01] that after seeing the grade of every object in every list, and thus having done sorted access to every object in every list L_i with $i \in Z$, there are not at least k objects with a grade that is at least equal to the final threshold value τ. In this situation, TA_Z halts after it has seen the grade of every object in every list. This situation cannot happen with TA.

for each missing field $i \in \{1, \ldots, m\} \backslash S$ the value 0, and applying t to the result. For example, if $S = \{1, \ldots, \ell\}$, then $W_S(R) = t(x_1, x_2, \ldots, x_\ell, 0, \ldots, 0)$.

What about upper bounds? The best value an object can attain depends on other information we have. We will use only the *bottom values* in each field, defined as in TA: $\underline{x}_i$ is the last (smallest) value obtained via sorted access in list L_i. Given an object R and subset $S(R) = \{i_1, i_2, \ldots, i_\ell\} \subseteq \{1, \ldots, m\}$ of known fields of R, with values $x_{i_1}, x_{i_2}, \ldots, x_{i_\ell}$ for these known fields, define $B_S(R)$ to be the maximum (or *best*) value the aggregation function t can attain for object R. When t is monotone, this maximum value is obtained by substituting for each missing field $i \in \{1, \ldots, m\} \backslash S$ the value $\underline{x}_i$, and applying t to the result. For example, if $S = \{1, \ldots, \ell\}$, then $B_S(R) = t(x_1, x_2, \ldots, x_\ell, \underline{x}_{\ell+1}, \ldots, \underline{x}_m)$.

We now give the algorithm NRA, which makes no random accesses.

1. Do sorted access in parallel to each of the m sorted lists L_i. At each depth d (when d objects have been accessed under sorted access in each list):
   - Maintain the bottom values $\underline{x}_1^{(d)}, \underline{x}_2^{(d)}, \ldots, \underline{x}_m^{(d)}$ encountered in the lists.
   - For every object R with discovered fields $S = S^{(d)}(R) \subseteq \{1, \ldots, m\}$, compute the values $W^{(d)}(R) = W_S(R)$ and $B^{(d)}(R) = B_S(R)$. (For objects R that have not been seen, these values are virtually computed as $W^{(d)}(R) = t(0, \ldots, 0)$, and $B^{(d)}(R) = t(\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_m)$, which is the threshold value.)
   - Let $T_k^{(d)}$, the current top k list, contain the k objects with the largest $W^{(d)}$ values seen so far (and their grades); if two objects have the same $W^{(d)}$ value, then ties are broken using the $B^{(d)}$ values, such that the object with the highest $B^{(d)}$ value wins (and arbitrarily among objects that tie for the highest $B^{(d)}$ value). Let $M_k^{(d)}$ be the kth largest $W^{(d)}$ value in $T_k^{(d)}$.
2. Call an object R *viable* if $B^{(d)}(R) > M_k^{(d)}$. Halt when (a) at least k distinct objects have been seen (so that in particular $T_k^{(d)}$ contains k objects) and (b) there are no viable objects left outside $T_k^{(d)}$, that is, when $B^{(d)}(R) \leq M_k^{(d)}$ for all $R \notin T_k^{(d)}$. Return the objects in $T_k^{(d)}$.

NRA is correct for each monotone aggregation function t [FLN01]. Further, as we noted, it is instance optimal when we restrict attention to algorithms that do not make random accesses.

As we noted earlier, Güntzer, Balke, and Kiessling [GBK01] also give an algorithm for the situation where random accesses are impossible. As with TA, they define a basic algorithm, called "Stream-Combine (basic version)" and a modified algorithm ("Stream-Combine") that incorporates a heuristic rule that tells which sorted list L_i to do a sorted access on next. Neither version of Stream-Combine is instance optimal. The reason that the basic version of Stream-Combine is not instance optimal is that it considers only upper bounds on overall grades of objects, unlike NRA, which considers both upper and lower bounds. They require that the top k objects be given with their grades (whereas as we discussed, in [FLN01] it is not required that the grades be given in the case where random accesses are impossible). The algorithms of [GBK01] cannot say that an object is in the top k unless that object has been seen in every sorted list. Note that there are monotone aggregation functions (such as max, or more interestingly, median) where it is possible to determine the overall grade of an object without knowing its grade in each sorted list.

We now give the algorithm CA, which is appropriate when random accesses are expensive relative to sorted accesses.

1. Do sorted access in parallel to each of the m sorted lists L_i. At each depth d (when d objects have been accessed under sorted access in each list):
   - Maintain the bottom values $\underline{x}_1^{(d)}, \underline{x}_2^{(d)}, \ldots, \underline{x}_m^{(d)}$ encountered in the lists.

- For every object R with discovered fields $S = S^{(d)}(R) \subseteq \{1, \ldots, m\}$, compute the values $W^{(d)}(R) = W_S(R)$ and $B^{(d)}(R) = B_S(R)$. (For objects R that have not been seen, these values are virtually computed as $W^{(d)}(R) = t(0, \ldots, 0)$, and $B^{(d)}(R) = t(\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_m)$, which is the threshold value.)
- Let $T_k^{(d)}$, the current top k list, contain the k objects with the largest $W^{(d)}$ values seen so far (and their grades); if two objects have the same $W^{(d)}$ value, then ties are broken using the $B^{(d)}$ values, such that the object with the highest $B^{(d)}$ value wins (and arbitrarily among objects that tie for the highest $B^{(d)}$ value). Let $M_k^{(d)}$ be the kth largest $W^{(d)}$ value in $T_k^{(d)}$.

2. Call an object R *viable* if $B^{(d)}(R) > M_k^{(d)}$. Every $h = \lfloor c_R / c_S \rfloor$ steps (that is, every time the depth of sorted access increases by h), do the following: pick the viable object that has been seen for which *not* all fields are known and whose $B^{(d)}$ value is as big as possible (ties are broken arbitrarily). Perform random accesses for all of its (at most $m-1$) missing fields. If there is no such object, then do not do a random access on this step.

3. Halt when (a) at least k distinct objects have been seen (so that in particular $T_k^{(d)}$ contains k objects) and (b) there are no viable objects left outside $T_k^{(d)}$, that is, when $B^{(d)}(R) \leq M_k^{(d)}$ for all $R \notin T_k^{(d)}$. Return the objects in $T_k^{(d)}$.

CA is correct for each monotone aggregation function t [FLN01]. Further, as we noted, under natural restrictions on the aggregation function and the class of databases, CA is instance optimal, and has optimality ratio independent of c_R/c_S [FLN01].

8 Summary

This paper gives an overview of some recent algorithms on aggregating information from various sources, in order to obtain the overall top k objects. The first such algorithm was "Fagin's Algorithm", which is optimal in a high-probability worst-case sense under certain assumptions. This was supplanted by the threshold algorithm, which is optimal in a much stronger sense, and without any probabilistic assumptions: it is instance optimal, as long as we restrict attention to algorithms that do not make wild guesses (other scenarios are also presented in [FLN01] where the threshold algorithm is instance optimal). We discuss a version of the threshold algorithm that is appropriate when we care only about the "approximately" top k answers. We discuss algorithms that are appropriate when sorted access is impossible for certain of the sorted lists, and algorithms that are appropriate when random accesses are expensive or impossible. Although we have not considered the issue here, upper and lower bounds are derived in [FLN01] for the optimality ratio in various circumstances (recall that the optimality ratio is the constant c in Equation (2)). As an example, it is shown that the optimality ratio of TA when we restrict attention to algorithms that do not make wild guesses is $m + m(m-1)c_R/c_S$, where m is the number of sorted lists, and it is shown that no deterministic algorithm has a lower optimality ratio.

Acknowledgments

The author is grateful to Leonid Libkin for encouraging him to write this paper, and to Phokion Kolaitis and Leonid Libkin for helpful comments that improved readability.

References

[BGM02] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of the 18th International Conference on Data Engineering.* IEEE Computer Society, 2002.

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proc. ACM SIGMOD Conference*, pages 91–102, 1996.

[Fag99] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. System Sci.*, 58:83–99, 1999.

[FLN01] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. ACM Symp. on Principles of Database Systems*, pages 102–113, 2001. Expanded version appears on `http://www.almaden.ibm.com/cs/people/fagin/`.

[GBK00] U. Güntzer, W-T. Balke, and W. Kiessling. Optimizing multi-feature queries in image databases. In *Proc. 26th Very Large Databases (VLDB) Conference*, pages 419–428, 2000.

[GBK01] U. Güntzer, W-T. Balke, and W. Kiessling. Towards efficient multi-feature queries in heterogeneous environments. In *Proc. of the IEEE International Conference on Information Technology: Coding and Computing (ITCC 2001)*, Las Vegas, USA, April 2001.

[NBE+93] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, and P. Yanker. The QBIC project: Querying images by content using color, texture and shape. In *SPIE Conference on Storage and Retrieval for Image and Video Databases*, volume 1908, pages 173–187, 1993. QBIC Web server is `http://wwwqbic.almaden.ibm.com/`.

[NCS+01] A. Natsev, Y-C. Chang, J. R. Smith, C-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *Proc. 27th Very Large Databases (VLDB) Conference*, pages 281–290, Rome, Italy, 2001.

[NR99] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proc. 15th International Conference on Data Engineering (ICDE)*, pages 22–29, March 1999.

[WHRB99] E. L. Wimmers, L. M. Haas, M. Tork Roth, and C. Braendli. Using Fagin's algorithm for merging ranked results in multimedia middleware. In *Fourth IFCIS International Conference on Cooperative Information Systems*, pages 267–278. IEEE Computer Society Press, September 1999.

[Zad69] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1969.

[Zim96] H. J. Zimmermann. *Fuzzy Set Theory.* Kluwer Academic Publishers, Boston, 3rd edition, 1996.