RJ 10205 (95079) December 18, 2000
Computer Science

# Research Report

## OPTIMAL AGGREGATION ALGORITHMS FOR MIDDLEWARE

Ronald Fagin

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Amnon Lotem

University of Maryland-College Park
Department of Computer Science
College Park, Maryland 20742

Moni Naor

Dept. of Computer Science and Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, Israel

RJ 10205 (95079) December 18, 2000
Computer Science

# Optimal Aggregation Algorithms for Middleware

Ronald Fagin*
Amnon Lotem†
Moni Naor‡

**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). There is some monotone *aggregation function*, or *combining rule*, such as min or average, that combines the individual grades to obtain an overall grade.

To determine objects that have the best overall grades, the naive algorithm must access every object in the database, to find its grade under each attribute. Fagin has given an algorithm ("Fagin's Algorithm", or FA) that is much more efficient. For some distributions on grades, and for some monotone aggregation functions, FA is optimal in a high-probability sense.

We give an elegant and remarkably simple algorithm ("the threshold algorithm", or TA) that is optimal in a much stronger sense than FA. We show that TA is essentially optimal, not just for some monotone aggregation functions, but for all of them, and not just in a high-probability sense, but over every database. Unlike FA, which requires large buffers (whose size may grow unboundedly as the database size grows), TA requires only a small, constant-size buffer.

*IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120. Email: fagin@almaden.ibm.com

†University of Maryland-College Park, Department of Computer Science, College Park, Maryland 20742. Email: lotem@cs.umd.edu

‡Dept. of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Email: naor@wisdom.weizmann.ac.il. The work of this author was performed while a Visiting Scientist at the IBM Almaden Research Center.

# 1 Introduction

Early database systems were required to store only textual information. Thus, the data was quite homogeneous. Today, we wish for our database systems to be able to deal not only with traditional textual data, but also with a heterogeneous variety of multimedia data (such as images, video, and audio). Furthermore, the data that we wish to access and combine may reside in a variety of data repositories, and we may want our database system to serve as middleware that can access such data.

One fundamental difference between traditional textual data and multimedia data is that multimedia data may have attributes that are 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).

One approach [Fa99] to deal with such fuzzy data is to make use of an *aggregation function* $t$. If $x_1, \ldots, x_m$ (each in the interval $[0,1]$) are the grades of object $R$ under the $m$ attributes, then $t(x_1, \ldots, x_m)$ is the overall grade of object $R$. As we shall discuss, such aggregation functions are useful in other contexts as well. There is a large literature on choices for the aggregation function (see Zimmermann's textbook [Zi96] and the discussion in [Fa99]).

One popular choice for the aggregation function is min. In fact, under the standard rules of fuzzy logic [Za65], if object $R$ has grade $x_1$ under attribute $A_1$ and $x_2$ under attribute $A_2$, then the grade under the fuzzy conjunction $A_1 \wedge A_2$ is $\min(x_1, x_2)$. Another popular aggregation function is the average (or the sum, in contexts where we do not care if the resulting overall grade no longer lies in the interval $[0,1]$).

We say that an aggregation function $t$ is *monotone* if $t(x_1, \ldots, x_m) \leq t(x'_1, \ldots, x'_m)$ whenever $x_i \leq x'_i$ for every $i$. Certainly monotonicity is a reasonable 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 $R$.

The notion of a query is different in a multimedia database system than in a traditional database system. Given a query in a traditional database system (such as a relational database system), there is an unordered set of answers.[1] By contrast, in a multimedia database system, the answer to a query can be thought of as a sorted list, with the answers sorted by grade. As in [Fa99], 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.

**Other applications:** There are other applications besides multimedia databases where we make use of an aggregation function to combine grades, and where we want to find the top $k$ answers. One important example is information retrieval [Sa89], where the objects $R$ of interest are documents, the $m$ attributes are search terms $s_1, \ldots, s_m$, and the grade $x_i$ measures the relevance of document $R$ for search term $s_i$, for $1 \leq i \leq m$. It is common to take the aggregation

[1] Of course, in a relational database, the result to a query may be sorted in some way for convenience in presentation, such as sorting department members by salary, but logically speaking, the result is still simply a set, with a crisply-defined collection of members.

algorithm $\mathcal{A}$ on input $\mathcal{D}$, or in this paper, the middleware cost incurred by running algorithm $\mathcal{A}$ over database $\mathcal{D}$. We shall mention examples later where $cost(\mathcal{A}, \mathcal{D})$ has an interpretation other than being the amount of a resource consumed by running the algorithm $\mathcal{A}$ on input $\mathcal{D}$.

We say that an algorithm $\mathcal{B} \in \mathbf{A}$ 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 $cost(\mathcal{B}, \mathcal{D}) \leq c \cdot cost(\mathcal{A}, \mathcal{D}) + c'$ for every choice of $\mathcal{A}$ and $\mathcal{D}$. We refer to $c$ as the *optimality ratio.* It is similar to the competitive ratio in competitive analysis (we shall discuss competitive analysis shortly). We use the word "optimal" to reflect that fact that $\mathcal{B}$ is essentially the best algorithm in $\mathbf{A}$.

Intuitively, instance optimality corresponds to optimality in every instance, as opposed to just the worst case or the average case. There are many algorithms that are optimal in a worst-case sense, but are not instance optimal. An example is binary search: in the worst case, binary search is guaranteed to require no more than $\log N$ probes, for $N$ data items. However, for each instance, a positive answer can be obtained in one probe, and a negative answer in two probes.

We consider a nondeterministic algorithm correct if on no branch does it make a mistake. We take the middleware cost of a nondeterministic algorithm to be the minimal cost over all branches where it halts with the top $k$ answers. We take the middleware cost of a probabilistic algorithm to be the expected cost (over all probabilistic choices by the algorithm). When we say that a deterministic algorithm $\mathcal{B}$ is instance optimal over $\mathbf{A}$ and $\mathbf{D}$, then we are really comparing $\mathcal{B}$ against the best nondeterministic algorithm, even if $\mathbf{A}$ contains only deterministic algorithms. This is because for each $\mathcal{D} \in \mathbf{D}$, there is always a deterministic algorithm that makes the same choices on $\mathcal{D}$ as the nondeterministic algorithm. We can view the cost of the best nondeterministic algorithm that produces the top $k$ answers over a given database as the cost of the shortest proof for that database that these are really the top $k$ answers. So instance optimality is quite strong: the cost of an instance optimal algorithm is essentially the cost of the shortest proof. Similarly, we can view $\mathbf{A}$ as if it contains also probabilistic algorithms that never make a mistake.

FA is optimal in a high-probability sense (actually, in a way that involves both high probabilities and worst cases; see [Fa99]), under certain assumptions. TA is optimal in a much stronger sense: it is instance optimal, for several natural choices of $\mathbf{A}$ and $\mathbf{D}$. In particular, instance optimality holds when $\mathbf{A}$ is taken to be the class of algorithms that would normally be implemented in practice (since the only algorithms that are excluded are those that make very lucky guesses), and when $\mathbf{D}$ is taken to be the class of all databases. Instance optimality of TA holds in this case for all monotone aggregation functions. By contrast, high-probability optimality of FA holds only under the assumption of "strictness" (we shall define strictness later; intuitively, it means that the aggregation function is representing some notion of conjunction).

The definition we have given for instance optimality is formally the same definition as is used in *competitive analysis* [ST85, BE98], except that in competitive analysis we do not assume that $\mathcal{B} \in \mathbf{A}$. In competitive analysis, typically (a) $\mathbf{A}$ is taken to be the class of offline algorithms that solve a particular problem, (b) $cost(\mathcal{A}, \mathcal{D})$ is taken to be taken to represent performance (where bigger numbers correspond to worse performance), and (c) $\mathcal{B}$ is a particular online

shape). In response to a query, such as *Color='red'*, the subsystem will output the graded set consisting of all objects, one by one, along with their grades under the query, in sorted order based on grade, until the middleware system tells the subsystem to halt. 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, along with their grades, then request the next 10, etc. This corresponds to what we have referred to as "sorted access".

There is another way that we might expect the middleware system to interact with the subsystem. The middleware system might ask the subsystem for the grade (with respect to a query) of any given object. This corresponds to what we have referred to as "random access". In fact, QBIC allows both sorted and random access.

There are some situations 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.

Our measure of cost corresponds intuitively to the cost incurred by the middleware system in processing information passed to it from a subsystem such as QBIC. Let $s$ be the number of sorted accesses. 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 $s = 120$. Let $r$ be the total number of grades of objects obtained from the database under random access. The *middleware cost* is taken to be $sc_S + rc_R$, for some positive constants $c_S$ and $c_R$. The fact that $c_S$ and $c_R$ may be different reflects the fact that the cost to a middleware system of a sorted access and of a random access may be different.

# 3 Fagin's Algorithm

In this section, we discuss FA (Fagin's Algorithm) [Fa99]. This algorithm is implemented in IBM's Garlic system [CHS+95]; see [WHTB99] for interesting details about the implementation and performance in practice. 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 to each of the lists $L_i$ to find the $i$th field $x_i$ of $R$.

3. Compute the grade[5] $t(R) = t(x_1, \ldots, x_m)$ 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 $\{(R, t(R)) \mid R \in Y\}$.

[5] We shall often abuse notation and write $t(R)$ for the grade $t(x_1, \ldots, x_m)$ of $R$.

**Proof:** Let $Y$ be as in Part 3 of TA. We need only show that every member of $Y$ has at least as high a grade as every object $z$ not in $Y$. By definition of $Y$, this is the case for each object $z$ that has been seen in running TA. So assume that $z$ was not seen. Assume that the fields of $z$ are $x_1, \ldots, x_m$. Therefore, $x_i \leq \underline{x}_i$, for every $i$. Hence, $t(z) = t(x_1, \ldots, x_m) \leq t(\underline{x}_1, \ldots, \underline{x}_m) = \tau$, where the inequality follows by monotonicity of $t$. But by definition of $Y$, for every $y$ in $Y$ we have $t(y) \geq \tau$. Therefore, for every $y$ in $Y$ we have $t(y) \geq \tau \geq t(z)$, as desired. ∎

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).

This implies that for every database, the sorted access cost for TA is at most that of FA. This does not imply that the middleware cost for TA is always at most that of FA, since TA may do more random accesses than FA. However, since the middleware cost of TA is at most the sorted access cost times a constant (independent of the database size), it does follow that the middleware cost of TA is at most a constant times that of FA. In fact, we shall show that TA is instance optimal, under natural assumptions.

We now consider the intuition behind TA. For simplicity, we discuss first the case where $k = 1$, that is, where the user is trying to determine the top answer. Assume that we are at a stage in the algorithm where we have not yet seen any object whose (overall) grade is at least as big as the threshold value $\tau$. The intuition is that at this point, we do not know the top answer, since the next object we see under sorted access could have overall grade $\tau$, and hence bigger than the grade of any object seen so far. Furthermore, once we do see an object whose grade is at least $\tau$, then it is safe to halt, as we see from the proof of Theorem 4.1. Thus, intuitively, the stopping rule of TA says: "Halt as soon as you know you have seen the top answer." Similarly, for general $k$, the stopping rule of TA says, intuitively, "Halt as soon as you know you have seen the top $k$ answers." So we could view TA as saying

**Do sorted access (and the corresponding random access) until you know you have seen the top $k$ answers.**

A little more generally, we can view TA as saying

**Gather what information you need to allow you to know the top $k$ answers, and then halt.**

These "programs" can be viewed as very high-level, "knowledge–based programs" [FHMV97]. In fact, TA can be viewed as being "designed" by thinking in terms of these knowledge-based programs. Later, we shall give another scenario (situations where random accesses are forbidden) where we make use of these same knowledge–based programs, but where the implementation is different. Interestingly, when we consider the scenario where random accesses are expensive relative to sorted accesses, but are not forbidden, we need an additional design principle beyond the knowledge-based programs to design an optimal algorithm.

The next simple theorem gives a useful property of TA, that further distinguishes TA from FA.

Define $\epsilon$ to be $t(0,\ldots,0)$. By monotonicity, $\epsilon$ is the minimal possible grade (we have $\epsilon = 0$ for reasonable choices of $t$, but we are not making this assumption). Let $\tau_{\mathcal{A}}$ be the threshold value when algorithm $\mathcal{A}$ halts. Thus, if $\underline{x}_i$ is the grade of the last object seen under sorted access to list $L_i$ for algorithm $\mathcal{A}$, for $i = 1,\ldots,m$, then $\tau_{\mathcal{A}} = t(\underline{x}_1,\ldots,\underline{x}_m)$. There are two cases, depending on whether $\tau_{\mathcal{A}} = \epsilon$ or $\tau_{\mathcal{A}} > \epsilon$.

*Case 1:* $\tau_{\mathcal{A}} = \epsilon$. TA needs to make at most $k-1$ more sorted accesses for each list $L_i$ before it halts. So TA has sorted access cost at most $k$ times that of algorithm $\mathcal{A}$.

*Case 2:* $\tau_{\mathcal{A}} > \epsilon$. If when algorithm $\mathcal{A}$ halts, it has seen at least $k$ objects whose grade is at least $\tau_{\mathcal{A}}$, then the sorted access cost of algorithm $\mathcal{A}$ over database $\mathcal{D}$ is at least equal to that of TA, so we are done. Therefore, assume that when algorithm $\mathcal{A}$ halts, it has seen less than $k$ objects whose grade is at least $\tau_{\mathcal{A}}$. If when algorithm $\mathcal{A}$ halts, it has seen at least $N/2$ objects, then we are done, since the sorted access cost of TA is at most a constant times $N$. Therefore, assume that when algorithm $\mathcal{A}$ halts, it has seen less than $N/2$ objects. Since we can also assume without loss of generality that $N/2 > k+1$ (because $k$ is a constant), it follows that we can assume that when algorithm $\mathcal{A}$ halts, there are at least $k+1$ objects that it has not seen. Let $S$ be the set of objects that algorithm $\mathcal{A}$ has seen, and $\overline{S}$ the set of objects that it has not seen. From what we just said, we can assume that $\overline{S}$ contains at least $k+1$ objects.

We now define two databases $\mathcal{D}'$ and $\mathcal{D}''$, each derived from $\mathcal{D}$, on one of which algorithm $\mathcal{A}$ gives the incorrect answer. For every $i$, the grade of every object in $S$ in the $i$th list is exactly the same in $\mathcal{D}'$ and $\mathcal{D}''$ as in $\mathcal{D}$. Hence, algorithm $\mathcal{A}$ performs exactly the same, and in particular gives the same answer, for databases $\mathcal{D}'$ and $\mathcal{D}''$ as for database $\mathcal{D}$.

Let $R$ be some fixed member of $\overline{S}$. In database $\mathcal{D}'$, let the grade of $R$ in the $i$th list be $\underline{x}_i$, and for every $y$ in $\overline{S}$ except $R$, let the grade of $y$ in the $i$th list be 0, for $i = 1,\ldots,m$. Therefore, the grade of $R$ is $t(\underline{x}_1,\ldots,\underline{x}_m) = \tau_{\mathcal{A}}$, and the grade of every $y$ in $\overline{S}$ except $R$ is $t(0,\ldots,0) = \epsilon < \tau_{\mathcal{A}}$. Since there are at most $k$ objects in database $\mathcal{D}'$ with grade at least $\tau_{\mathcal{A}}$, algorithm $\mathcal{A}$ must output $R$ as one of the top $k$ objects.

In database $\mathcal{D}''$, let the grade of $R$ in the $i$th list be 0, and for every $y$ in $\overline{S}$ except $R$, let the grade of $y$ in the $i$th list be $\underline{x}_i$, for $i = 1,\ldots,m$. Therefore, the grade of $R$ is $\epsilon$ in $\mathcal{D}''$, and the grade of every $y$ in $\overline{S}$ except $R$ is $\tau_{\mathcal{A}}$. Since there are at least $k$ objects in database $\overline{S}$ with grade $\tau_{\mathcal{A}}$, and since $R$ has grade $\epsilon < \tau_{\mathcal{A}}$, we know that $R$ is not one of the top $k$ objects in database $\mathcal{D}''$. Since we showed that algorithm $\mathcal{A}$ must output $R$ as one of the top $k$ objects, it follows that algorithm $\mathcal{A}$ is incorrect. ∎

The next example shows that wild guesses can help.

**Example 4.4:** Assume that there are $2n+1$ objects, which we will call simply $1, 2, \ldots, 2n+1$, and there are two lists $L_1$ and $L_2$. Assume that in list $L_1$, the grades of the first $n+1$ objects $1, 2, \ldots, n+1$ are all 1, and the grade of the remaining $n$ objects $n+2, n+3, \ldots, 2n+1$ are all 0. Assume that in list $L_2$, the grade of the first $n$ objects $1, 2, \ldots, n$ are all 0, and the grade of the remaining $n+1$ objects $n+1, n+2, \ldots, 2n+1$ are all 1. Assume that the aggregation function is min, and that we are interested in finding the top answer (i.e., $k = 1$). It is clear that the top answer is object $n+1$ with (overall) grade 1 (every object except object $n+1$ has grade 0).

An algorithm that makes a wild guess and asks for the grade of object $n+1$ in both lists would determine the correct answer and be able to halt safely after two random accesses and no

list $L_i$, let $\underline{x}_i$ be the grade of the last object seen under sorted access by algorithm $\mathcal{A}$. That is, $\underline{x}_i$ is the $s$th highest grade in list $L_i$. Let $\underline{\underline{x}}_i$ be the $(s+1)$th highest grade in list $L_i$. For each $i$, let $B_i$ be the top $s+1$ objects in list $L_i$ (since there are no ties by the uniqueness property, this set is well-defined). Let $B = \cup_i B_i$.

Define $\tau'$ to be $t(\underline{\underline{x}}_1, \ldots, \underline{\underline{x}}_m)$. If TA halts after at most $s+1$ sorted accesses to each list (so that the threshold value is at least $\tau'$), then we are done, since TA makes at most one more sorted access to each list than $\mathcal{A}$ does. So assume that this is not the case; therefore, there are less than $k$ objects in $B$ whose grade is at least $\tau'$. As before, let $S$ be the set of objects seen by algorithm $\mathcal{A}$. Let $S_1$ consist of those members of $S$ that are seen under sorted access by $\mathcal{A}$, and let $S_2 = S \backslash S_1$. Thus, $S_2$ consists of those objects seen only under random access by $\mathcal{A}$ (these correspond to the wild guesses). For each $i$ and each object $y$ of $S_2$, the grade of $y$ in list $L_i$ is at most $\underline{\underline{x}}_i$. Hence, by monotonicity, the grade of $y$ is at most $\tau'$. That is, there are no objects in $S_2$ whose grade is greater than $\tau'$. Furthermore, since $S_1 \subseteq B$ and since there are less than $k$ objects in $B$ whose grade is at least $\tau'$, it follows that there are less than $k$ objects in $S_1$ whose grade is at least $\tau'$. It follows that there are less than $k$ objects in $S$ whose grade is greater than $\tau'$.

As in the proof of Theorem 4.3, we now define two databases $\mathcal{D}'$ and $\mathcal{D}''$, each derived from $\mathcal{D}$, on one of which algorithm $\mathcal{A}$ gives the incorrect answer. Just as before, the grade of every object in $S$ (the objects that algorithm $\mathcal{A}$ has seen) in every list $L_i$ is exactly the same in $\mathcal{D}'$ and $\mathcal{D}''$ as in $\mathcal{D}$. Hence, algorithm $\mathcal{A}$ performs exactly the same, and in particular gives the same answer for databases $\mathcal{D}'$ and $\mathcal{D}''$ as for database $\mathcal{D}$.

Let $u_1, \ldots, u_r$ be the distinct members of $\overline{S}$. As noted, we can assume that the size $r$ of $\overline{S}$ is at least $k+1$. By the uniqueness property, we know that for every $i$, we have $\underline{x}_i > \underline{\underline{x}}_i$. In database $\mathcal{D}'$, define the grade (denoted $z_{ij}$) of $u_j$ in list $L_i$, for each $i, j$, in such a way that $\underline{x}_i > z_{i1} > z_{i2} > \cdots > z_{ir} > \underline{\underline{x}}_i$. By strict monotonicity, we know that each $u_j$ has grade greater than $\tau'$. Further, $u_1$ has strictly the highest grade of all objects in $\overline{S}$. Since also there are less than $k$ objects in $S$ whose grade is greater than $\tau'$, it follows that algorithm $\mathcal{A}$ must output $u_1$ as one of the top $k$ objects.

Let database $\mathcal{D}''$ be just like database $\mathcal{D}'$, except that the grades of $u_1$ and $u_r$ are reversed in each list. In particular, by strict monotonicity we know that $u_1$ has strictly the lowest grade of members of $\overline{S}$. Since $\overline{S}$ contains at least $k+1$ members, it follows that $u_1$ is not one of the top $k$ objects in database $\mathcal{D}''$. Since we showed that algorithm $\mathcal{A}$ must output $u_1$ as one of the top $k$ objects, it follows that algorithm $\mathcal{A}$ is incorrect. ∎

# 5 Minimizing Random Access

Thus far in this paper, we have not been especially concerned about the number of random accesses. In our algorithms we have discussed so far (namely, FA and TA), for every sorted access, up to $m-1$ random accesses take place. Recall that if $s$ is the number of sorted accesses, and $r$ is the number of random accesses, then the middleware cost is $sc_S + rc_R$, for some positive constants $c_S$ and $c_R$. Our notion of optimality ignores constant factors like $m$ and $c_R$ (they are simply multiplicative factors in the optimality ratio). Hence, there has been no motivation so far to concern ourself with the number of random accesses.

For example, if $S = \{1, \ldots, \ell\}$, then $B_S(R) = t(x_1, x_2, \ldots, x_\ell, \underline{x}_{\ell+1}, \ldots, \underline{x}_m)$. The following property is immediate from the definition:

**Proposition 5.2:** *If $S$ is the set of known fields of object $R$, then $t(R) \leq B_S(R)$.*

In other words, $B(R)$ represents an upper bound on the value $t(R)$ (or the *best* value $t(R)$ can be), given the information we have so far. Is it the best upper bound? If the lists may contain equal values (which in general we assume they can), then given the information we have it is possible that $t(R) = B_S(R)$. If the uniqueness property holds (equalities are not allowed in a list), then for continuous functions it is the case that $B(R)$ is the best upper bound on the value $t$ can have on $R$. In general, as an algorithm progresses and we learn more fields of an object $R$ and the bottom values $\underline{x}_i$ decrease, $B(R)$ can only decrease (or remain the same).

An important special case is an object $R$ that has not been encountered at all. In this case $B(R) = t(\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_m)$. Note that this is the same as the threshold value in TA.

## 5.1 No Random Access Algorithm - NRA

We now present an algorithm that does not use random accesses at all.

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 following:
   - The bottom values encountered in each list: $\underline{x}_1^{(d)}, \underline{x}_2^{(d)}, \ldots, \underline{x}_m^{(d)}$.
   - For every object $R$ with for which the discovered fields are $S = S^{(d)}(R) \subseteq \{1, \ldots, m\}$ the values $W^{(d)}(R) = W_S(R)$ and $B^{(d)}(R) = B_S(R)$.
   - 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 if there is a tie for the highest $B^{(d)}$ value). Denote this top $k$ list by $T_k^{(d)}$. Let $M_k^{(d)}$ be the $k$th largest $W^{(d)}$ value in $T_k^{(d)}$.
2. Call an object $R$ *viable* if $B^{(d)}(R) > M_k^{(d)}$. Halt when 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 $T_k^{(d)}$ with their grades.

We now show that NRA is correct for each monotone aggregation function $t$.

**Theorem 5.3:** *If the aggregation function $t$ is monotone, then NRA correctly finds the top $k$ answers.*

**Proof:** Assume that NRA halts after $d$ sorted accesses to each list, and that $T_k^{(d)} = \{R_1, R_2, \ldots, R_k\}$. Thus, the objects output by NRA are $R_1, R_2, \ldots, R_k$. Let $R$ be an object not among $R_1, R_2, \ldots, R_k$. We must show that $t(R) \leq t(R_i)$ for each $i$.

*Subcase 2:* $M_k^{(d-1)} \neq W^{(d-1)}(R_i)$, and so $M_k^{(d-1)} < W^{(d-1)}(R_i)$. From the inequalities in (3), we see that $M_k^{(d-1)} < t(R_i)$. So by (2), we have $t(R) < t(R_i)$, as desired.

*Case 2:* Algorithm $\mathcal{A}$ does not output $R$ as one of the top $k$ objects. We construct a database $\mathcal{D}''$ where $\mathcal{A}$ errs as follows. Database $\mathcal{D}''$ is identical to $\mathcal{D}$ up to depth $d-1$. At level $d$ it gives each missing field $i \in \{1,\ldots,m\} \backslash S^{(d-1)}(R)$ of $R$ the value $\underline{x}_i^{(d-1)}$. For all remaining missing fields, including missing fields of $R_1,\ldots,R_k$, assign the value 0. Now $t(R) = B^{(d-1)}(R) > M_k^{(d-1)}$, whereas (a) for at least one $R_i$ (namely, that $R_i$ where $W^{(d)}(R_i) = M_k^{(d)}$) we have $t(R_i) = M_k^{(d-1)}$, and (b) for each object $R'$ not among $R_1, R_2, \ldots, R_k$ or $R$ we have that $t(R') \leq M_k^{(d-1)}$. Hence, algorithm $\mathcal{A}$ is in error in not outputting $R$ as one of the top $k$ objects. ∎

Note that the issue of "wild guesses" is not relevant here, since no algorithm that makes no random access can get any information about an object except via sorted access.

**Implementation of NRA:** Unfortunately, executing NRA may require a lot of bookkeeping at each step, since when NRA does sorted access at depth $t$ (for $1 \leq t \leq d$), the value of $B^{(t)}(R)$ must be updated for every object $R$ seen so far. This may be up to $dm$ updates for each depth $t$, which yields a total of $\Omega(d^2)$ updates by depth $d$. Furthermore, unlike TA, it no longer suffices to have bounded buffers. However, for a specific function like min it is possible that by using appropriate data structures the computation can be greatly simplified. This is an issue for further investigation.

## 5.2 Taking into Account the Random Access Cost

We now present the combined algorithm CA that does use random accesses, but takes their cost (relative to sorted access) into account. As before, let $c_S$ be the cost of a sorted access and $c_R$ be the cost of a random access. The middleware cost of an algorithm that makes $s$ sorted accesses and $r$ random ones is $sc_S + rc_R$. We know that TA is instance optimal; however, the optimality ratio is a function of the relative cost of a random access to a sorted access, that is $c_R/c_S$. Our goal in this section is to find an algorithm that is instance optimal and where the optimality ratio is independent of $c_R/c_S$. One can view CA as a merge between TA and NRA. Let $h = \lfloor c_R/c_S \rfloor$. We assume in this section that $c_R \geq c_S$, so that $h \geq 1$. The idea of CA is to run NRA, but every $h$ steps to run a random access phase and update the information (the upper and lower bounds $B$ and $W$) accordingly.

The algorithm CA is as follows.

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 following:

   - The bottom values encountered in each list: $\underline{x}_1^{(d)}, \underline{x}_2^{(d)}, \ldots, \underline{x}_m^{(d)}$.
   - For every object $R$ with for which the discovered fields are $S = S^{(d)}(R) \subseteq \{1,\ldots,m\}$ the values $W^{(d)}(R) = W_S(R)$ and $B^{(d)}(R) = B_S(R)$.
   - 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 if there is a tie for

that not only does CA fail to fulfill this hope, but so does every algorithm! In other words, neither of these scenarios is enough to guarantee the existence of an algorithm that is instance optimal, with optimality ratio independent of $c_R/c_S$.

However, we shall show that by slightly strengthening the assumption on $t$ in the second scenario, CA becomes instance optimal, with optimality ratio independent of $c_R/c_S$. Let us say that the aggregation function $t$ is *strictly monotone in each argument* if whenever one argument is strictly increased and the remaining arguments are held fixed, then the value of the aggregation function is strictly increased. That is, $t$ is strictly monotone in each argument if $x_i < x_i'$ implies that

$$t(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m) < t(x_1, x_2, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_m).$$

The average (or sum) is strictly monotone in each argument, whereas min is not.

We shall show (Section 5.4) that in the second scenario above, if we replace "The aggregation function $t$ is strictly monotone" by "The aggregation function $t$ is strictly monotone in each argument", then CA is instance optimal, with optimality ratio independent of $c_R/c_S$. We shall also show that the same result holds if instead, we simply take $t$ to be min, even though min is not strictly monotone in each argument.

## 5.4 Positive Results about CA

We now show that in the second scenario above, if we replace "The aggregation function $t$ is strictly monotone" by "The aggregation function $t$ is strictly monotone in each argument", then CA is instance optimal, with optimality ratio independent of $c_R/c_S$.

**Theorem 5.6:** *Assume that the aggregation function $t$ is strictly monotone in each argument. Let* **D** *be the class of all databases with the uniqueness property. Let* **A** *be the class of all algorithms that correctly find the top $k$ answers for $t$ for every database in* **D**. *Then CA is instance optimal over* **A** *and* **D**, *with optimality ratio independent of $c_R/c_S$.*

**Proof:** Assume $\mathcal{D} \in \mathbf{D}$. Assume that when CA runs on $\mathcal{D}$, it halts after doing sorted access to depth $d$. Thus, CA makes $md$ sorted accesses and $r$ random accesses, where $r \leq md/h$. Note that in CA the two components ($mdc_S$ and $rc_R$) of the cost $mdc_S + rc_R$ are roughly equal, and their sum is at most $2mdc_S$. Assume $\mathcal{A} \in \mathbf{A}$, and that $\mathcal{A}$ makes $d'$ sorted accesses and $r'$ random accesses. The cost that $\mathcal{A}$ incurs is therefore $d'c_S + r'c_R$.

Suppose that algorithm $\mathcal{A}$ announces that objects $R_1', R_2', \ldots, R_k'$ are the top $k$. First, we claim that each $R_i'$ appears in the top $d' + r' + 1$ objects of at least one list $L_j$. Suppose not. Then there is an object $R_i'$ output by $\mathcal{A}$ such that in each list there is a vacancy above $R_i'$ that has not been accessed either by sorted or random access. There is a database $\mathcal{D}'$ identical to $\mathcal{D}$ in all locations accessed by $\mathcal{A}$ but with an object $R' \notin \{R_1', R_2', \ldots, R_k'\}$ whose values reside in these vacancies. From the uniqueness property, for each field the value for $R'$ is strictly larger than that for $R_i'$, and from strict monotonicity of $t$ we have $t(R') > t(R_i')$, making $R'$ a mandatory member of the output. (Note: we used only strict monotonicity of $t$ rather than the stronger property of being strictly monotone in each variable.) This is a contradiction. Hence, each $R_i'$ appears in the top $d' + r' + 1$ objects of at least one list $L_j$.

$$\leq \quad (4m + k + 1)(d'c_S + r'c_R)$$

Since $d'c_S + r'c_R$ is the middleware cost of $\mathcal{A}$, we get that the middleware cost of CA is within a multiple of $4m + k + 1$ of that of $\mathcal{A}$.

So we need only show that we may assume $r' \geq 1$. Assume not. Then $\mathcal{A}$ makes no random accesses. Now by Theorem 5.4, NRA is instance optimal compared with algorithms that make no random access, and of course the optimality ratio is independent of $c_R/c_S$. Further, the cost of CA is at most twice that of NRA. So CA is instance optimal compared with algorithms that make no random access, such as $\mathcal{A}$, with optimality ratio independent of $c_R/c_S$. ∎

The next theorem says that for the function min (which is not strictly monotone in each argument), algorithm CA is still instance optimal.

**Theorem 5.7:** *Let* **D** *be the class of all databases with the uniqueness property. Let* **A** *be the class of all algorithms that correctly find the top $k$ answers when the aggregation function is* min *for every database in* **D**. *Then CA is instance optimal over* **A** *and* **D**, *with optimality ratio independent of $c_R/c_S$.*

**Proof:** (Sketch) The proof is similar to the proof of Theorem 5.6, where the key point is that for the function min at every step $d$ of CA there can be at most $m$ different $R$'s with the same $B^{(d)}(R)$ value, since $B^{(d)}(R)$ equals one of the fields of $R$ and the uniqueness property assures that there are at most $m$ different fields in *all* lists with the same value (this replaces the use of strict monotonicity in each argument). Therefore at step $d' + r' + 1 + h|C|$ there are at most $m$ objects with $B$ value that equals $S_k$, and there are no objects outside of $\{R'_1, R'_2, \ldots, R'_k\}$ whose $B$ value exceeds $S_k$. Since the $B$ value of each member of $\{R'_1, R'_2, \ldots R'_k\}$ is at least $S_k$, it follows that after $mh$ more steps all of $\{R'_1, R'_2, \ldots, R'_k\}$ will be randomly accessed, so there will be no viable objects left and CA will halt. The rest of the analysis is similar to the proof of Theorem 5.6. ∎

## 5.5 Negative Results about CA

In this section, we show that even under the scenarios of Theorems 4.3 and 4.5, there is no algorithm that is instance optimal, with optimality ratio independent of $c_R/c_S$. We also show that if we were to modify algorithm CA so that it were to become an intermittent algorithm that executes $h = \lfloor c_R/c_S \rfloor$ steps of NRA and then one step of TA, then the resulting algorithm would perform much worse than CA.

We begin by showing that the conditions of Theorem 4.3 are not sufficient to guarantee the existence of an instance optimal algorithm with optimality ratio independent of $c_R/c_S$, even when the aggregation function is min, and when $k = 1$ (so that we are interested only in the top answer).

**Theorem 5.8:** *Let* **D** *be the class of all databases. Let* **A** *be the class of all algorithms that correctly find the top answer for* min *for every database and that do not make wild guesses. There*

**Theorem 5.9:** *Let the aggregation function $t$ be given by $t(x_1, x_2, x_3) = \min(x_1 + x_2, x_3)$. Let* **D** *be the class of all databases that satisfy the uniqueness property. Let* **A** *be the class of all algorithms that correctly find the top answer for $t$ for every database in* **D**. *There is no deterministic algorithm (or even probabilistic algorithm that never makes a mistake) that is instance optimal over* **A** *and* **D**, *where the optimality ratio is independent of $c_R/c_S$.*

**Proof:** Let $\mathcal{D}$ be the database where the top $d$ values in $L_1$ and $L_2$ are of the form $i/(2d)$ for $1 \leq i \leq d$, and the object with value $i/(2d)$ in list $L_1$ is the one with the value $(d-i+1)/(2d)$ in list $L_2$. Hence the $x_1 + x_2$ values of these $d$ objects is $1/2$. The values in $L_3$ are of the form $i/N$, for $1 \leq i \leq N$, and only one of the top $d$ objects in $L_1$ and $L_2$ is matched with a value at least $1/2$; all the others are matched with smaller values. Therefore the question is which one it is, since this would be the unique object whose $t(x_1, x_2, x_3) = \min(x_1 + x_2, x_3)$ value is $1/2$. Furthermore, simply based on the top $d$ objects in $L_1$ and $L_2$, it is clear that the top answer for $t$ is at most $1/2$. There is an algorithm (that does not make wild guesses) that accesses the top $d$ objects in lists $L_1$ and $L_2$, and then makes a single random access to $L_3$ to the object with grade in $L_3$ at least $1/2$. Its cost is at most $2dc_S + c_R$.

Consider the following distribution on databases: each member is as above and the object from $L_1$ and $L_2$ that is matched in $L_3$ with value at least $1/2$ is chosen at random among the top $d$ in $L_1$ and $L_2$. It is easy to see that the expected cost under this distribution of an arbitrary algorithm $\mathcal{A} \in \mathbf{A}$ is at least $(d/2)(c_S + c_R)$, whereas the best algorithm on each such database has cost at most $dc_S + c_R$. So the optimality ratio of $\mathcal{A}$ is at least

$$\frac{(d/2)(c_S + c_R)}{2dc_S + c_R}. \tag{6}$$

As before, by varying $d$, $c_R$, and $c_S$, we can make (6) arbitrarily large.

In the case of probabilistic algorithms that never makes a mistake, we conclude as in the conclusion of the proof of Theorem 5.8. ∎

**CA versus the intermittent algorithm:** We close by considering the choice we made in CAof doing random access to find the fields of the viable object $R$ whose $B^{(d)}$ value is the maximum. We compare its performance with what we could call the "intermittent algorithm", which is composed by running NRA with delayed TA every $h$ steps. That is, the intermittent algorithm does random accesses in the same time order as TA does, but simply delays them, so that it does random accesses every $h = \lfloor c_R/c_S \rfloor$ steps. We show a database where the intermittent algorithm does much worse than CA. Consider the aggregation function $t$ where $t(x_1, x_2, x_3) = x_1 + x_2 + x_3$. Let $h$ be large. Let $\mathcal{D}$ be a database where the top $h-2$ locations in $L_1$ and $L_2$ have values of the form $1/2 + i/(8h)$, for $1 \leq i \leq h-2$, and where none are matched with each other. Location $h-1$ in the two lists belong to same object $R$, with value $1/2$ in both of them. Location $h$ in the two lists both have the value $1/8$. In $L_3$ the top $h^2 - 1$ locations have values of the form $1/2 + i/(8h^2)$, for $1 \leq i \leq h^2 - 1$, and in location $h^2$, object $R$ has field value $1/2$. Note that the maximum $t$ value (which occurs for the object $R$) is $1\frac{1}{2}$ and that all objects that appear in one of the top $h-2$ locations in lists $L_1$ and $L_2$ have $t$ values that are at most $1\frac{3}{8}$. At step $h$ in CA we have that $B^{(h)}(R) \geq 1\frac{1}{2}$, whereas for all other objects their $B^{(h)}$ value is at most $1\frac{3}{8}$. Therefore on this database, CA performs $h$ sorted accesses in parallel and a single random access on $R$ and then halts. The intermittent algorithm, on the

[NR99] S. Nepal and M. V. Ramakrishna, Query Processing Issues in Image (Multimedia) Databases, *Proc. 15th International Conference on Data Engineering (ICDE)* (Mar. 1999), pp. 22–29.

[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, *SPIE Conference on Storage and Retrieval for Image and Video Databases* (1993), volume 1908, pp. 173–187. QBIC Web server is `http://wwwqbic.almaden.ibm.com/`.

[Sa89] G. Salton, *Automatic Text Processing, the Transformation, Analysis and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA (1989). .

[ST85] D. Sleator and R. E. Tarjan, Amortized Efficiency of List Update and Paging Rules, *Comm. ACM* **28** (1985), pp. 202–208.

[WHTB99] E. L. Wimmers, L. M. Haas, M. Tork Roth, and C. Braendli, Using Fagin's Algorithm for Merging Ranked Results in Multimedia Middleware. *Fourth IFCIS International Conference on Cooperative Information Systems* (Sept. 1999), IEEE Computer Society Press, pp. 267–278.

[Ya77] A. C-C. Yao, Probabilistic Computations: Towards a Unified Measure of Complexity, *Proc. 17th IEEE Symp. on Foundations of Computer Science* (1977), pp. 222–227.

[Za65] L. A. Zadeh, Fuzzy Sets, *Information and Control* **8** (1965), pp. 338–353.

[Zi96] H.-J. Zimmermann, *Fuzzy Set Theory*, Third Edition, Kluwer Academic Publishers, Boston (1996).