Optimal Aggregation Algorithms for Middleware

[Extended Abstract] *

Ronald Fagin
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
fagin@almaden.ibm.com

Amnon Lotem
University of Maryland–College Park
Dept. of Computer Science
College Park, Maryland 20742
lotem@cs.umd.edu

Moni Naor†
Weizmann Institute of Science
Dept. of Computer Science and Applied Mathematics
Rehovot 76100, Israel
naor@wisdom.weizmann.ac.il

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 analyze 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.

We distinguish two types of access: sorted access (where the middleware system obtains the grade of an object in some sorted list by proceeding through the list sequentially from the top), and random access (where the middleware system requests the grade of object in a list, and obtains it in one step). We consider the scenarios where random access is either impossible, or expensive relative to sorted access, and provide algorithms that are essentially optimal for these cases as well.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*multimedia systems, query processing*; E.5 [**Files**]: Sorting/searching; F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms

Algorithms, Performance, Theory

Keywords

middleware, instance optimality, competitive analysis

1. INTRODUCTION

Early database systems were required to store only small character strings, such as the entries in a tuple in a traditional relational database. Thus, the data was quite homogeneous. Today, we wish for our database systems to be able to deal not only with character strings (both small and large), 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 small character strings 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 [4] 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 [15] and the discussion in [4]).

[*] A full version of this paper is available on `http://www.almaden.ibm.com/cs/people/fagin/`

[†] The work of this author was performed while a Visiting Scientist at the IBM Almaden Research Center.

PODS '01 Santa Barbara, California USA

One popular choice for the aggregation function is min. In fact, under the standard rules of fuzzy logic [14], 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 [4], 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 [11], 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 function t to be the sum. That is, the total relevance score of document R when the query consists of the search terms $s_1, \ldots, s_m$ is taken to be $t(x_1, \ldots, x_m) = x_1 + \cdots + x_m$.

Another application arises in a paper by Aksoy and Franklin [1] on scheduling large-scale on-demand data broadcast. In this case each object is a page, and there are two fields. The first field represents the amount of time waited by the earliest user requesting a page, and the second field represents the number of users requesting a page. They make use of the product function t with $t(x_1, x_2) = x_1 x_2$, and they wish to broadcast next the page with the top score.

The model: 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 is thought of 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 completely ignoring computational issues. For example, 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 *middleware cost* is taken to be $sc_S + rc_R$, for some positive constants c_S and c_R.

Algorithms: There is an obvious naive algorithm for obtaining the top k answers. 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.

Fagin [4] introduced an algorithm ("Fagin's Algorithm", or FA), which often does much better than the naive algorithm. In the case where the orderings in the sorted lists are probabilistically independent, FA finds the top k answers, over a database with N objects, with middleware cost $O(N^{(m-1)/m} k^{1/m})$, with arbitrarily high probability.[2] Fagin also proved that under this independence assumption, along with an assumption on the aggregation function, every correct algorithm must, with high probability, incur a similar middleware cost.

We shall present the "threshold algorithm", or TA. This algorithm has been defined and studied by (at least) three groups, including Nepal and Ramakrishna [9] (who were the first to publish), Güntzer, Balke, and Kiessling [5], and ourselves.[3] For more information and comparison, see Section 6 on related work.

We shall show that TA is optimal in a much stronger sense than FA. We now define this notion of optimality, which we consider to be interesting in its own right.

Instance optimality: Let $\mathbf{A}$ be a class of algorithms, and let $\mathbf{D}$ be a class of legal inputs to the algorithms. We assume that we are considering a particular nonnegative cost measure $cost(\mathcal{A}, \mathcal{D})$ of running algorithm $\mathcal{A}$ over input $\mathcal{D}$. This cost could be the running time of 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 *B2stan*A: B2 yAo herd

[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.

we have

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

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. For convenience, in our proofs we shall always assume that $\mathbf{A}$ contains only deterministic algorithms, since the results carry over automatically to nondeterministic algorithms and to 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 [4]), 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* [2, 12], 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 a number that represents performance (where bigger numbers correspond to worse performance), and (c) $\mathcal{B}$ is a particular online algorithm. In this case, the online algorithm $\mathcal{B}$ is said to be *competitive*. The intuition is that a competitive online algorithm may perform poorly in some instances, but only on instances where every offline algorithm would also perform poorly.

Another example where the framework of instance optimality appears, but again without the assumption that $\mathcal{B} \in \mathbf{A}$, is in the context of *approximation algorithms* [7]. In this case, (a) $\mathbf{A}$ is taken to contain algorithms that solve a particular problem exactly (in cases of interest, these algorithms are not polynomial-time algorithms), (b) $cost(\mathcal{A}, \mathcal{D})$ is taken to be the resulting answer when algorithm $\mathcal{A}$ is applied to input $\mathcal{D}$, and (c) $\mathcal{B}$ is a particular polynomial-time algorithm.

Restricting random access: As we shall discuss in Section 2, there are some systems where random access is impossible. To deal with such situations, we show in Section 5.1 how to modify TA to obtain an algorithm NRA ("no random accesses") that does no random accesses. We prove that NRA is instance optimal over all algorithms that do not make random accesses and over all databases.

What about situations where random access is not forbidden, but simply expensive? Wimmers et al. [13] discuss a number of systems issues that can cause random access to be expensive. Although TA is instance optimal, the optimality ratio depends on the ratio c_R/c_S of the cost of a single random access to the cost of a single sorted access. We define another algorithm that is a combination of TA and NRA, and call it CA ("combined algorithm"). The definition of the algorithm depends on c_R/c_S. The motivation is to obtain an algorithm that is not only instance optimal, but whose optimality ratio is independent of c_R/c_S. Our original hope was that CA would be instance optimal (with optimality ratio independent of c_R/c_S) in those scenarios where TA is instance optimal. Not only does this hope fail, but interestingly enough, we prove that there does not exist any deterministic algorithm, or even probabilistic algorithm that does not make a mistake, that is instance optimal (with optimality ratio independent of c_R/c_S) in these scenarios! However, we find a new natural scenario where CA is instance optimal, with optimality ratio independent of c_R/c_S.

2. MODES OF ACCESS TO DATA

Issues of efficient query evaluation in a middleware system are very different from those in a traditional database system. This is because the middleware system receives answers to queries from various subsystems, which can be accessed only in limited ways. What do we assume about the interface between a middleware system and a subsystem? Let us consider QBIC[4] [10] ("Query By Image Content") as a subsystem. QBIC can search for images by various visual characteristics such as color and texture (and an experimental version can search also by 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 sub-

[4] QBIC is a trademark of IBM Corporation.

system 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. In both cases, 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. As before, if there are s sorted accesses and r random accesses, then 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) [4]. This algorithm is implemented in Garlic [3], an experimental IBM middleware system; see [13] 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.)[5] 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 ith field x_i of R.

3. Compute the grade[6] $t(R) = t(x_1, \dots, x_m)$ 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 $\{(R, t(R)) \mid R \in Y\}$.[7]

It is fairly easy to show [4] that this algorithm is correct for monotone aggregation functions t (that is, that the algorithm successfully finds the top k answers). 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 [4].

An aggregation function t is *strict* [4] if $t(x_1, \dots, x_m) = 1$ 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. In fact, it is reasonable to think of strictness as being a key characterizing feature of the conjunction.

Fagin shows that his algorithm is optimal (in a high-probability sense) if the aggregation function is strict (so that, intuitively, we are dealing with a notion of conjunction), and if the orderings in the sorted lists are probabilistically independent. In fact, under the assumption that the sorted lists are probabilistically independent, the middleware cost of FA is $\Theta(N^{(m-1)/m}k^{1/m})$, with arbitrarily high probability, no matter what the aggregation function is. This is true even for a constant aggregation function; in this case, of course, there is a trivial algorithm that gives us the top k answers (any k objects will do) with $O(1)$ middleware cost. So FA is not optimal in any sense for some monotone aggregation functions t. By contrast, as we shall see, the algorithm TA is instance optimal for every monotone aggregation function, under very weak assumptions.

Even in the cases where FA is optimal, this optimality holds only in a high-probability sense. This leaves open the possibility that there are some algorithms that have much better middleware cost than FA over certain databases. The algorithm TA, which we now discuss, is such an algorithm.

4. THE THRESHOLD ALGORITHM

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 $t(R) = t(x_1, \dots, x_m)$ 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 $t(\underline{x}_1, \dots, \underline{x}_m)$. 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 $\{(R, t(R)) \mid R \in Y\}$.

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

THEOREM 4.1. *If the aggregation function t is monotone, then TA correctly finds the top k answers.*

[5] It is not actually important that the lists be accessed "in lockstep". In practice, it may be convenient to allow the sorted lists to be accessed at different rates, in batches, etc. Throughout this paper, whenever we speak of "sorted access in parallel", all of our instance optimality results continue to hold even when sorted access is not in lockstep, as long as the rates of sorted access of the lists are within constant multiples of each other.

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

[7] Graded sets are often presented in sorted order, sorted by grade.

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.

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

THEOREM 4.2. *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.

There is a price to pay for the bounded buffers. Thus, for every time an object is found under sorted access, TA may do $m-1$ random accesses (where m is the number of lists), to find the grade of the object in the other lists. This is in spite of the fact that this object may have already been seen under sorted or random access in one of the other lists.

4.1 Instance Optimality of the Threshold Algorithm

In this section, we investigate the instance optimality of TA. We would have liked to be able to simply state that for every monotone aggregation function, TA is instance optimal over all algorithms that correctly find the top k answers, over the class of all databases. However, it turns out that the situation is more delicate than this. We first make a distinction between algorithms that "make wild guesses" (that is, perform random access on elements not previously encountered by sorted access) and those that do not. (Neither FA nor TA make wild guesses, and neither does any "natural" algorithm.) Our first theorem (Theorem 4.3) says that for every monotone aggregation function, TA is instance optimal over all algorithms that correctly find the top k answers *and that do not make wild guesses*, over the class of *all* databases. We then show that this distinction (wild guesses vs. no wild guesses) is essential: if algorithms that make wild guesses are allowed in the class $\mathbf{A}$ of algorithms that an instance optimal algorithm must compete against, then *no* algorithm is instance optimal (Example 4.4 and Theorem 4.5). The heart of this example (and the corresponding theorem) is the fact that there may be multiple objects with the same grade in some list. Indeed, once we restrict our attention to databases where no two objects have the same value in the same list, and make a slight, natural additional restriction on the aggregation function beyond monotonicity, then TA is instance optimal over *all* algorithms that correctly find the top k answers (Theorem 4.6).

In Section 4.3 we consider instance optimality in the situation where we relax the problem of finding the top k objects into finding *approximately* the top k.

We now give our first positive result on instance optimality of TA. We say that an algorithm *makes wild guesses* if it does random access to find the grade of some object R in some list before the algorithm has seen R under sorted access. That is, an algorithm makes wild guesses if the first grade that it obtains for some object R is under random access. We would not normally implement algorithms that make wild guesses. In fact, there are some contexts where it would not even be possible to make wild guesses (such as a database context where the algorithm could not know the name of an object it has not already seen). However, making a lucky wild guess can help, as we show later (Example 4.4).

We now show instance optimality of TA among algorithms that do not make wild guesses. In this theorem, when we take $\mathbf{D}$ to be the class of all databases, we really mean that $\mathbf{D}$ is the class of all databases that involve sorted lists corresponding to the arguments of the aggregation function t. We are taking k (where we are trying to find the top k answers) and the aggregation function t to be fixed. Since we are taking t to be fixed, we are thereby taking the number m of arguments of t (that is, the number of sorted lists) to be fixed. In Section 4.2, we discuss the assumptions that k and m are constant.

THEOREM 4.3. *Assume that the aggregation function t is monotone. Let $\mathbf{D}$ be the class of all databases. Let $\mathbf{A}$ be the class of all algorithms that correctly find the top k answers for t for every database and that do not make wild guesses. Then TA is instance optimal over $\mathbf{A}$ and $\mathbf{D}$.*

Proof: Assume that $\mathcal{A} \in \mathbf{A}$, and that algorithm $\mathcal{A}$ is run over database $\mathcal{D}$. Assume that algorithm $\mathcal{A}$ halts at depth d (that is, if d_i is the number of objects seen under sorted access to list i, for $1 \leq i \leq m$, then $d = \max_i d_i$). Assume that $\mathcal{A}$ sees a distinct objects (some possibly multiple times). In particular, $a \geq d$. We shall show that TA halts on $\mathcal{D}$ by depth $a+k$. Hence, TA makes at most $m(a+k)$ accesses, which is ma plus an additive constant of mk. It follows easily that the optimality ratio of TA is at most cm, where $c = \max\{c_R/c_S, c_S/c_R\}$.

Note that for each choice of d', the algorithm TA sees at least d' objects by depth d' (this is because by depth d' it has made md' sorted accesses, and each object is accessed at most m times under sorted access). Let Y be the output set of $\mathcal{A}$ (consisting of the top k objects). If there are at most k objects that $\mathcal{A}$ does not see, then TA halts by depth $a+k$ (after having seen every object), and we are done. So assume that there are at least $k+1$ objects that $\mathcal{A}$ does not see. Since Y is of size k, there is some object V that $\mathcal{A}$ does not see and that is not in Y.

Let $\tau_{\mathcal{A}}$ be the threshold value when algorithm $\mathcal{A}$ halts. This means that if $\underline{x}_i$ is the grade of the last object seen under sorted access to list i for algorithm $\mathcal{A}$, for $1 \leq i \leq m$, then $\tau_{\mathcal{A}} = t(\underline{x}_1, \dots, \underline{x}_m)$. (For convenience, let us assume that algorithm $\mathcal{A}$ makes at least one sorted access to each list; this introduces at most m more sorted accesses.) Let us call an object R *big* if $t(R) \geq \tau_{\mathcal{A}}$, and otherwise call object R *small*.

We now show that every member R of Y is big. Define a database $\mathcal{D}'$ to be just like $\mathcal{D}$, except that object V has grade $\underline{x}_i$ in the ith list, for $1 \leq i \leq m$. Put V in list i below all other objects with grade $\underline{x}_i$ in list i (for $1 \leq i \leq m$). Algorithm $\mathcal{A}$ performs exactly the same, and in particular gives the same output, for databases $\mathcal{D}$ and $\mathcal{D}'$. Therefore, algorithm $\mathcal{A}$ has R, but not V, in its output for database $\mathcal{D}'$. Since the grade of V in $\mathcal{D}'$ is $\tau_{\mathcal{A}}$, it follows by correctness of $\mathcal{A}$ that R is big, as desired.

There are now two cases, depending on whether or not algorithm $\mathcal{A}$ sees every member of its output set Y.[8]

Case 1: Algorithm $\mathcal{A}$ sees every member of Y. Then by depth d, TA will see every member of Y. Since, as we showed, each member of Y is big, it follows that TA halts by depth $d \leq a < a + k$, as desired.

Case 2: Algorithm $\mathcal{A}$ does not see some member R of Y. We now show that every object R' that is not seen by $\mathcal{A}$ must be big. Define a database $\mathcal{D}'$ that is just like $\mathcal{D}$ on every object seen by $\mathcal{A}$. Let the grade of V in list i be $\underline{x}_i$, and put V in list i below all other objects with grade $\underline{x}_i$ in list i (for $1 \leq i \leq m$). Therefore, the grade of V in database $\mathcal{D}'$ is $\tau_{\mathcal{A}}$. Since $\mathcal{A}$ cannot distinguish between $\mathcal{D}$ and $\mathcal{D}'$, it has the same output on $\mathcal{D}$ and $\mathcal{D}'$. Since $\mathcal{A}$ does not see R and does not see R', it has no information to distinguish between R and R'. Therefore, it must have been able to give R' in its output without making a mistake. But if R' is in the output and not V, then by correctness of $\mathcal{A}$, it follows that R' is big. So R' is big, as desired.

Since $\mathcal{A}$ sees a objects, and since TA sees at least $a + k$ objects by depth $a + k$, it follows that by depth $a + k$, TA sees at least k objects not seen by $\mathcal{A}$. We have shown that every object that is not seen by $\mathcal{A}$ is big. Therefore, by depth $a + k$, TA sees at least k big objects. So TA halts by depth $a + k$, as desired. □

We now show that making a lucky wild guess can help.

Example 4.4. Assume that there are $2n+1$ objects, which we will call simply $1, 2, \dots, 2n+1$, and there are two lists L_1 and L_2. Assume that in list L_1, the objects are in the order $1, 2, \dots, 2n+1$, where the top $n+1$ objects $1, 2, \dots, n+1$ all have grade 1, and the remaining n objects $n+2, n+3, \dots, 2n+1$ all have grade 0. Assume that in list L_2, the objects are in the reverse order $2n+1, 2n, \dots, 1$, where the bottom n objects $1, \dots, n$ all have grade 0, and the remaining $n+1$ objects $n+1, n+2, \dots, 2n+1$ all have grade 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 overall 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 sorted accesses.[9] However, let $\mathcal{A}$ be any algorithm (such as TA) that does not make wild guesses. Since the winning object $n+1$ is in the middle of both sorted lists, it follows that at least $n+1$ sorted accesses would be required before algorithm $\mathcal{A}$ would even see the winning object. □

Example 4.4 shows that TA is not instance optimal over the class $\mathbf{A}$ of all algorithms that find the top answer for min (with two arguments) and the class $\mathbf{D}$ of all databases. The next theorem says that no algorithm is instance optimal. The proof (and other missing proofs) appear in the full paper.

Theorem 4.5. *Let $\mathbf{D}$ be the class of all databases. Let $\mathbf{A}$ be the class of all algorithms that correctly find the top answer for min (with two arguments) for every database. There is no deterministic algorithm (or even probabilistic algorithm that never makes a mistake) that is instance optimal over $\mathbf{A}$ and $\mathbf{D}$.*

Although, as we noted earlier, algorithms that make wild guesses would not normally be implemented in practice, it is still interesting to consider them. This is because of our interpretation of instance optimality of an algorithm $\mathcal{A}$ as saying that its cost is essentially the same as the cost of the shortest proof for that database that these are really the top k answers. If we consider algorithms that allow wild guesses, then we are allowing a larger class of proofs. Thus, in Example 4.4, the fact that object $n+1$ has (overall) grade 1 is a proof that it is the top answer.

We say that an aggregation function t is *strictly monotone* if $t(x_1, \dots, x_m) < t(x'_1, \dots, x'_m)$ whenever $x_i < x'_i$ for every i. Although average and min are strictly monotone, there are aggregation functions suggested in the literature for representing conjunction and disjunction that are monotone but not strictly monotone (see [4] and [15] for examples). We say that a database $\mathcal{D}$ *satisfies the uniqueness property* if for each i, no two objects in $\mathcal{D}$ have the same grade in list L_i, that is, if the grades in list L_i are distinct. We now show that these conditions guarantee optimality of TA even among algorithms that make wild guesses.

Theorem 4.6. *Assume that the aggregation function t is strictly monotone. Let $\mathbf{D}$ be the class of all databases that satisfy the uniqueness property. Let $\mathbf{A}$ be the class of all algorithms that correctly find the top k answers for t for every database in $\mathbf{D}$. Then TA is instance optimal over $\mathbf{A}$ and $\mathbf{D}$.*

Proof: Assume that $\mathcal{A} \in \mathbf{A}$, and that algorithm $\mathcal{A}$ is run over database $\mathcal{D} \in \mathbf{D}$. Assume that $\mathcal{A}$ sees a distinct objects (some possibly multiple times). We shall show that TA halts on $\mathcal{D}$ by depth $a + k$. As before, this shows that the optimality ratio of TA is at most cm, where $c = \max\{c_R/c_S, c_S/c_R\}$.

If there are at most k objects that $\mathcal{A}$ does not see, then TA halts by depth $a + k$ (after having seen every object),

[8] For the sake of generality, we are allowing the possibility that algorithm $\mathcal{A}$ can output an object that it has not seen. We discuss this issue more in Section 4.2.

[9] The algorithm could halt safely, since it "knows" that it has found an object with the maximal possible grade of 1 (this grade is maximal, since we are assuming that all grades lie between 0 and 1). Even if we did not assume that all grades lie between 0 and 1, one additional sorted access would provide the information that each overall grade in the database is at most 1.

and we are done. So assume that there are at least $k+1$ objects that $\mathcal{A}$ does not see. Since Y is of size k, there is some object V that $\mathcal{A}$ does not see and that is not in Y. We shall show that TA halts on $\mathcal{D}$ by depth $a+1$.

Let τ be the threshold value of TA at depth $a+1$. Thus, if $\underline{x}_i$ is the grade of the $(a+1)$th highest object in list i, then $\tau = t(\underline{x}_1, \ldots, \underline{x}_m)$. Let us call an object R *big* if $t(R) \geq \tau$, and otherwise call object R *small.* (Note that these definitions of "big" and "small" are different from those in the proof of Theorem 4.3.)

We now show that every member R of Y is big. Let x'_i be some grade in the top $a+1$ grades in list i that is not the grade in list i of any object seen by $\mathcal{A}$. There is such a grade, since all grades in list i are distinct, and $\mathcal{A}$ sees at most a objects. Let $\mathcal{D}'$ agree with $\mathcal{D}$ on all objects seen by A, and let object V have grade x'_i in the ith list of $\mathcal{D}'$, for $1 \leq i \leq m$. Hence, the grade of V in $\mathcal{D}'$ is $t(x'_1, \ldots, x'_m) \geq \tau$. Since V was unseen, and since V is assigned grades in each list in $\mathcal{D}'$ below the level that $\mathcal{A}$ reached by sorted access, it follows that algorithm $\mathcal{A}$ performs exactly the same, and in particular gives the same output, for databases $\mathcal{D}$ and $\mathcal{D}'$. Therefore, algorithm $\mathcal{A}$ has R, but not V, in its output for database $\mathcal{D}'$. By correctness of $\mathcal{A}$, it follows that R is big, as desired.

We claim that every member R of Y is one of the top $a+1$ members of some list i (and so is seen by TA by depth $a+1$). Assume by way of contradiction that R is not one of the top $a+1$ members of list i, for $1 \leq i \leq m$. By our assumptions that the aggregation function t is strictly monotone. and that $\mathcal{D}$ satisfies the uniqueness property, it follows easily that R is small. We already showed that every member of Y is big. This contradiction proves the claim. It follows that TA halts by depth $a+1$, as desired. □

The proofs of Theorems 4.3 and 4.6 have several nice properties:

- The proofs would still go through if we were in a scenario where, whenever a random access of object R in list i takes place, we learn not only the grade of R in list i, but also the relative rank.

- The proofs would still go through if we were to restrict the class of databases to those where each list i has a certain fixed domain.

- As we shall see, we can prove the instance optimality among approximation algorithms of an approximation version of TA, under the assumptions of Theorem 4.3, with only a small change to the proof (such a theorem does not hold under the assumptions of Theorem 4.6).

4.2 Treating k and m as Constants

In Theorems 4.3 and 4.6 about the instance optimality of TA, we are treating k (where we are trying to find the top k answers) and m (the number of sorted lists) as constants. We now discuss these assumptions.

We begin first with the assumption that k is constant. As in the proofs of Theorems 4.3 and 4.6, let a be the number of accesses by an algorithm $\mathcal{A} \in \mathbf{A}$. If $a \geq k$, then there is no need to treat k as a constant. Thus, if we were to restrict the class $\mathbf{A}$ of algorithms to contain only algorithms that make at least k accesses to find the top k answers, then there would be no need to assume that k is constant. How can it arise that an algorithm $\mathcal{A}$ can find the top k answers without making at least k accesses, and in particular without accessing at least k objects? It must then happen that either there are at most k objects in the database, or else every object R that $\mathcal{A}$ has not seen has the same overall grade $t(R)$. The latter will occur, for example, if t is a constant function. Even under these circumstances, it is still not reasonable in some contexts (such as certain database contexts) to allow an algorithm $\mathcal{A}$ to output an object as a member of the top k objects without ever having seen it: how would the algorithm even know the name of the object? This is similar to an issue we raised earlier about wild guesses.

We see from the proofs of Theorems 4.3 and 4.6 that the optimality ratio depends only on m, and is in fact linear in m. The next theorem shows that the linear dependence of the optimality ratio of TA on m in these theorems is essential. In fact, the next theorem shows that a dependence that is at least linear holds not just for TA, but for every correct deterministic algorithm (or even probabilistic algorithm that never makes a mistake). This dependence holds even when the aggregation function is min, and when $k=1$ (so that we are interested only in the top answer). An analogous theorem about the dependence of the optimality ratio on m holds also under the scenario of Theorem 4.6.

THEOREM 4.7. *Let* $\mathbf{D}$ *be the class of all databases. Let* $\mathbf{A}$ *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 is no deterministic algorithm (or even probabilistic algorithm that never makes a mistake) with an optimality ratio over* $\mathbf{A}$ *and* $\mathbf{D}$ *that is less than* $m/2$.

4.3 Turning TA into an Approximation Algorithm

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. Let us say that an algorithm *finds a* θ*-approximation to the top* k *answers for* t *over database* $\mathcal{D}$ if it gives as output k objects (and their grades) such that for each y among these k objects and each z not among these k objects, $\theta t(y) \geq t(z)$. We can modify TA to work under these requirements by modifying the stopping rule in Part 2 to say "As soon as at least k objects have been seen whose grade, when multiplied by θ, is at least equal to τ, then halt." Let us call this approximation algorithm TA$_\theta$. A straightforward modification of the proof of Theorem 4.1 shows that TA$_\theta$ is correct. We now show that if no wild guesses are allowed, then TA$_\theta$ is instance optimal.

THEOREM 4.8. *Assume that* $\theta > 1$ *and that the aggregation function* t *is monotone. Let* $\mathbf{D}$ *be the class of all databases. Let* $\mathbf{A}$ *be the class of all algorithms that find a* θ*-approximation to the top* k *answers for* t *for every database and that do not make wild guesses. Then TA*$_\theta$ *is instance optimal over* $\mathbf{A}$ *and* $\mathbf{D}$.

Proof: The proof of Theorem 4.3 carries over verbatim provided we modify the definition of an object R being "big" to be that $\theta t(R) \geq \tau_{\mathcal{A}}$. □

Theorem 4.8 shows that the analog of Theorem 4.3 holds for TA$_\theta$. The next example, which is a modification of Example 4.4, shows that the analog of Theorem 4.6 does *not* hold for TA$_\theta$. One interpretation of these results is that

Theorem 4.3 is sufficiently robust that it can survive the perturbation of allowing approximations, whereas Theorem 4.6 is not.

EXAMPLE 4.9. Assume that $\theta > 1$, that there are $2n+1$ objects, which we will call simply $1, 2, \dots, 2n+1$, and that there are two lists L_1 and L_2. Assume that in list L_1, the grades are assigned so that all grades are different, the ordering of the objects by grade is $1, 2, \dots, 2n+1$, object $n+1$ has the grade $1/\theta$, and object $n+2$ has the grade $1/(2\theta^2)$. Assume that in list L_2, the grades are assigned so that all grades are different, the ordering of the objects by grade is $2n+1, 2n, \dots, 1$ (the reverse of the ordering in L_1), object $n+1$ has the grade $1/\theta$, and object $n+2$ has the grade $1/(2\theta^2)$. Assume that the aggregation function is min, and that $k = 1$ (so that we are interested in finding a θ-approximation to the top answer). The (overall) grade of each object other than object $n+1$ is at most $\alpha = 1/(2\theta^2)$. Since $\theta\alpha = 1/(2\theta)$, which is less than the grade $1/\theta$ of object $n+1$, it follows that the unique object that can be returned by an algorithm such as TA_θ that correctly finds a θ-approximation to the top answer is the object $n+1$.

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 sorted accesses. The algorithm could halt safely, since it "knows" that it has found an object R such that $\theta t(R) = 1$, and so $\theta t(R)$ is at least as big as every possible grade. However, under sorted access for list L_1, TA_θ would see the objects in the order $1, 2, \dots, 2n+1$, and under sorted access for list L_2, TA_θ would see the objects in the reverse order. Since the winning object $n+1$ is in the middle of both sorted lists, it follows that at least $n+1$ sorted accesses would be required before TA_θ would even see the winning object. □

Just as Example 4.4 was generalized into Theorem 4.5, we can generalize Example 4.9 into the following theorem.

THEOREM 4.10. *Assume that $\theta > 1$. Let* **D** *be the class of all databases that satisfy the uniqueness condition. Let* **A** *be the class of all algorithms that find a θ-approximation to the top answer for min 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**.

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.

There are, however, some scenarios where we must pay attention to the number of random accesses. The first scenario is where random accesses are impossible (which corresponds to $c_R = \infty$). As we discussed in Section 2, an example of this first scenario arises when the middleware system is a text retrieval system, and the sorted lists correspond to the results of search engines. Another scenario is where random accesses are not impossible, but simply expensive, relative to sorted access. An example of this second scenario arises when the costs correspond to disk access (sequential versus random). Then we would like the optimality ratio to be independent of c_R/c_S. That is, if instead of treating c_S and c_R as constants, we allow them to vary, we would still like the optimality ratio to be bounded.

In this section we describe algorithms that do not use random access frivolously. We give two algorithms. One uses no random accesses at all, and hence is called NRA ("No Random Access"). The second algorithm takes into account the cost of a random access. It is a combination of NRA and TA, and so we call it CA ("Combined Algorithm").

Both algorithms access the information in a natural way, and intuitively, halt when they know that no improvement can take place. In general, at each point in an execution of these algorithms where a number of sorted and random accesses have taken place, for each object R there is a subset $S(R) = \{i_1, i_2, \dots, i_\ell\} \subseteq \{1, \dots, m\}$ of the fields of R where the algorithm has determined the values $x_{i_1}, x_{i_2}, \dots, x_{i_\ell}$ of these fields. Given this information, we define functions of this information that are lower and upper bounds on the value $t(R)$ can obtain. The algorithm proceeds until there are no more candidates whose current upper bound is better than the current kth largest lower bound.

Lower Bound: Given an object R and subset $S(R) = \{i_1, i_2, \dots, i_\ell\} \subseteq \{1, \dots, m\}$ of known fields of R, with values $x_{i_1}, x_{i_2}, \dots, x_{i_\ell}$ for these known fields, we define $W_S(R)$ (or W(R) if the subset $S = S(R)$ is clear) as 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 for each missing field $i \in \{1, \dots, m\} \backslash S$ the value 0, and applying t to the result. For example, if $S = \{1, \dots, \ell\}$, then $W_S(R) = t(x_1, x_2, \dots, x_\ell, 0, \dots, 0)$. The following property is immediate from the definition:

PROPOSITION 5.1. *If S is the set of known fields of object R, then $t(R) \geq W_S(R)$.*

In other words, $W(R)$ represents a lower bound on $t(R)$. Is it the best possible? Yes, unless we have additional information, such as that the value 0 does not appear in the lists. In general, as an algorithm progresses and we learn more fields of an object R, its W value becomes larger (or at least not smaller). For some aggregation functions t the value $W(R)$ yields no knowledge until S includes all fields: for instance if t is min, then $W(R)$ is 0 until all values are discovered. For other functions it is more meaningful. For instance, when t is the median of three fields, then as soon as two of them are known $W(R)$ is at least the smaller of the two.

Upper Bound: 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 $S(R) = \{i_1, i_2, \dots, i_\ell\} \subseteq \{1, \dots, m\}$ of known fields of R, with values $x_{i_1}, x_{i_2}, \dots, x_{i_\ell}$ for these known fields, we define $B_S(R)$ (or B(R) if the subset S is clear) as 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 $i \in \{1, \dots, m\} \backslash S$ the value $\underline{x}_i$, and applying t to the result. For example, if $S = \{1, \dots, \ell\}$, 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 each 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 aggregation functions t 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

As we have discussed, there are situations where random accesses are forbidden. We now consider algorithms that make no random accesses. Since random accesses are forbidden, in this section we change our criterion for the desired output. In earlier sections, we demanded that the output be the "top k answers", which consists of the top k objects, along with their (overall) grades. In this section, we make the weaker requirement that the output consist of the top k objects, without their grades. The reason is that, since random access is impossible, it may be much cheaper (that is, require many fewer accesses) to find the top k answers without their grades. This is because, as we now show by example, 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.

EXAMPLE 5.3. Consider the following scenario, where the aggregation function is the average, and where $k = 1$ (so that we are interested only in the top object). There are only two sorted lists L_1 and L_2, and the grade of every object in both L_1 and L_2 is 1/3, except that object R has grade 1 in L_1 and grade 0 in L_2. After two sorted accesses to L_1 and one sorted access to L_2, there is enough information to know that object R is the top object (its average grade is at least 1/2, and every other object has average grade at most 1/3). If we wished to find the grade of object R, we would need to do sorted access to all of L_2. □

Note that we are requiring only that the output 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. Let C_i be the cost of finding the top i objects. It is interesting to note that there is no necessary relationship between C_i and C_j for $i < j$. For example, in Example 5.3, we have $C_1 < C_2$. If we were to modify Example 5.3 so that there are two objects R and R' with grade 1 in L_1, where the grade of R in L_2 is 0, and the grade of R' in L_2 is 1/4 (and so that, as before, all remaining grades of all objects in both lists is 1/3), then $C_2 < C_1$.

The cost of finding the top k objects in sorted order is at most $k \max_i C_i$. Since we are treating k as a constant, it follows easily that we can convert our instance optimal algorithm (which we shall give shortly) for finding the top k objects into an instance optimal algorithm for finding the top k objects in sorted order. In practice, it is usually good enough to know the top k objects in sorted order, without knowing the grades. In fact, the major search engines on the web no longer give grades (possibly to prevent reverse engineering).

The algorithm NRA 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 $\underline{x}_1^{(d)}, \underline{x}_2^{(d)}, \ldots, \underline{x}_m^{(d)}$ encountered in the lists.
   - For every object R with discovered fields $S = S^{(d)}(R) \subseteq \{1, \ldots, m\}$ 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 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 $T_k^{(d)}$.

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

THEOREM 5.4. *If the aggregation function t is monotone, then NRA correctly finds the top k objects.*

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.

Since the algorithm halts at depth d, we know that R is nonviable at depth d, that is, $B^{(d)}(R) \leq M_k^{(d)}$. Now $t(R) \leq B^{(d)}(R)$ (Proposition 5.2). Also for each of the k objects R_i we have $M_k^{(d)} \leq W^{(d)}(R_i) \leq t(R_i)$ (from Proposition 5.1 and the definition of $M_k^{(d)}$). Combining the inequalities we have shown, we have

$$t(R) \leq B^{(d)}(R) \leq M_k^{(d)} \leq W^{(d)}(R_i) \leq t(R_i)$$

for each i, as desired. □

Note that the tie-breaking mechanism was not significant for correctness. We claim instance optimality of NRA over all algorithms that do not use random access:

THEOREM 5.5. *Assume that the aggregation function t is monotone. Let $\mathbf{D}$ be the class of all databases. Let $\mathbf{A}$ be the class of all algorithms that correctly find the top k objects for t for every database and that do not make random accesses. Then NRA is instance optimal over $\mathbf{A}$ and $\mathbf{D}$.*

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, the execution of 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. As in Section 5.1, in this section we require only that the output consist of the top k objects, without their grades. If we wish to obtain the grades, this requires only a constant number of additional random accesses, and so has no effect on instance optimality.

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 $\underline{x}_1^{(d)}, \underline{x}_2^{(d)}, \ldots, \underline{x}_m^{(d)}$ encountered in the lists.
   - For every object R with discovered fields $S = S^{(d)}(R) \subseteq \{1, \ldots, m\}$ 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 kth largest $W^{(d)}$ value in $T_k^{(d)}$.
2. Call an object R *viable* if $B^{(d)}(R) > M_k^{(d)}$. Every h steps (that is, every time the depth of sorted access increases by h), do the following: pick the viable object R whose $B^{(d)}$ value is the maximum and for which *not* all fields are known. Perform random accesses for all the (at most $m-1$) missing fields.
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 $T_k^{(d)}$.

Note that if h is very large (say larger than the number of objects in the database), then algorithm CA is the same as NRA, since no random access is performed. Similarly, if h is very small, say $h = 1$, then algorithm CA is essentially the same as TA, since for each step of doing sorted access in parallel we perform random accesses for all of the missing fields of some object. If instead of performing random accesses for all of the missing fields of some object, we performed random accesses for all of the missing fields of each object seen in sorted access, then the resulting algorithm would be identical to TA. However, for moderate values of h it is *not* the case that CA is equivalent to the intermittent algorithm that executes h steps of NRA and then one step of TA. In the full paper, we give an example where the intermittent algorithm performs much worse than CA. The difference between the algorithms is that CA picks "wisely" on which objects to perform the random access, namely, according to their $B^{(d)}$ values.

Correctness of CA is essentially the same as for NRA, since the same upper and lower bounds are maintained:

THEOREM 5.6. *If the aggregation function t is monotone, then CA correctly finds the top k objects.*

In the next section, we consider scenarios under which CA is instance optimal, with the optimality ratio independent of c_R/c_S.

5.3 Instance Optimality of CA: Positive and Negative Results

In Section 4, we gave two scenarios under which TA is instance optimal over $\mathbf{A}$ and $\mathbf{D}$. In the first scenario (from Theorem 4.3), (1) the aggregation function t is monotone; (2) $\mathbf{D}$ is the class of all databases; and (c) $\mathbf{A}$ is the class of all algorithms that correctly find the top k objects for t for every database and that do not make wild guesses. In the second scenario (from Theorem 4.6), (1) the aggregation function t is strictly monotone; (2) $\mathbf{D}$ is the class of all databases that satisfy the uniqueness property; and (3) $\mathbf{A}$ is the class of all algorithms that correctly find the top k objects for t for every database in $\mathbf{D}$. We might hope that under either of these two scenarios, CA is instance optimal, with optimality ratio independent of c_R/c_S. Unfortunately, this hope is false, in both scenarios. In fact, we shall give theorems that say 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 see 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

$$\begin{aligned} & t(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m) \\ < \quad & t(x_1, \ldots, x_{i-1}, x'_i, x_{i+1}, \ldots, x_m). \end{aligned}$$

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

We shall see (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 see 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

The next theorem says 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 5.7. *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 objects 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.*

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

THEOREM 5.8. *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 objects 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.*

5.5 Negative Results about CA

In this section, we see that even under the scenarios of Theorems 4.3 and 4.6, there is no algorithm that is instance optimal, with optimality ratio independent of c_R/c_S.

We begin with a theorem that says that the conditions of Theorem 4.3 (i.e., not allowing wild guesses) 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 object).

THEOREM 5.9. *Let* **D** *be the class of all databases. Let* **A** *be the class of all algorithms that correctly find the top object for* min *for every database and that do not make wild guesses. 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.*

We now give a theorem that says that the conditions of Theorem 4.6 (i.e., strict monotonicity and the uniqueness property) are not sufficient to guarantee the existence of an instance optimal algorithm with optimality ratio independent of c_R/c_S, even when $k = 1$ (so that we are interested only in the top object). In this counterexample, we take the aggregation function t to be given by $t(x_1, x_2, x_3) = \min(x_1 + x_2, x_3)$. Note that t is strictly monotone, although it is not strictly monotone in each argument. This shows that in Theorem 5.7, we needed to assume that t is strictly monotone in each argument, rather than simply assuming that t is strictly monotone.

THEOREM 5.10. *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 object 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.*

6. RELATED WORK

Nepal and Ramakrishna [9] define an algorithm that is equivalent to TA. Their notion of optimality is weaker than ours. F list should be accessed next under sorted access can be forced to be instance optimal simply by insuring that each list is accessed under sorted access at least every u steps, for some constant u.

In another paper, Güntzer, Balke, and Kiessling [6] consider the situation where random accesses are impossible. Once again, 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 our algorithm NRA, which considers both upper and lower bounds. They require that the top k objects be given with their grades (whereas as we discussed, we do not require the grades to be given in the case where random accesses are impossible). Their algorithm 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.

7. CONCLUSIONS

We studied the elegant and remarkably simple algorithm TA, as well as algorithms for the scenario where random access is forbidden or expensive relative to sorted access (NRA and CA). To study these algorithms, we introduced the instance optimality framework in the context of aggregation algorithms, and provided both positive and negative results. This framework is appropriate for analyzing and comparing the performance of algorithms, and provides a very strong notion of optimality. We also considered approximation algorithms, and provided positive and negative results about instance optimality there as well.

Two interesting lines of investigation are: (i) finding other scenarios where instance optimality can yield meaningful results, and (ii) finding other applications of our algorithms, such as in information retrieval.

8. ACKNOWLEDGMENTS

We are grateful to Michael Franklin for discussions that led to this research, and to Larry Stockmeyer for helpful comments that improved readability.

9. REFERENCES

[1] D. Aksoy and M. Franklin. RxW: A scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Transactions On Networking*, 7(6):846–880, December 1999.

[2] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, New York, 1998.

[3] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: the Garlic approach. In *RIDE-DOM '95 (5th Int'l Workshop on Research Issues in Data Engineering: Distributed Object Management)*, pages 124–131, 1995.

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

[5] 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, Cairo, Egypt, 2000.

[6] 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.

[7] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.

[8] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, U.K., 1995.

[9] 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.

[10] 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/`.

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

[12] D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28:202–208, 1985.

[13] E. L. Wimmers, L. M. Haas, M. T. 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.

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

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