1 Introduction

This paper [2] describes a simple and elegant algorithm called the Threshold Algorithm (TA) and it’s variants. These algorithms are used for Top-K processing. The applications of top-k processing include retrieving data from multimedia databases, information retrieval and scheduling large-scale on-demand data broadcast. The main objective of Top-k processing is to return the k highest ranked results quickly and efficiently. One common way to identify the top-k objects is by assigning a score to all objects based on some scoring function. An object’s score acts as a valuation for that object according to its characteristics (e.g., price and size of house objects in a real estate database, or color and texture of images in a multimedia database). Data objects are usually evaluated by multiple scoring predicates that contribute to the total object score. A scoring function is therefore usually defined as an aggregation over partial scores.

1.1 Motivation

The authors of this paper have mainly concentrated on aggregation algorithms for middle-ware. We can think of the middle-ware system as an interface between users and the multimedia databases. Here the middle-ware system access different heterogeneous varieties of data such as images, audio and video from different multimedia databases. Methods employed to access the data from a multimedia database is entirely different from that of traditional database systems. Traditional database systems store only small character strings and the data is structured, whereas multimedia data is unstructured and fuzzy. This means that we can retrieve tuples that exactly match the criteria in a query for a traditional database, whereas we cannot decide whether an object in a multimedia database exactly match an user’s requirement. For example if the user wants to retrieve all red objects, we cannot apply a boolean function that might decide whether an object is red or not, rather we would have to devise a function that returns how red an object is. In other words this function returns a numeric grade based on the redness of a given object factoring in the red component of its constituent pixels. Rarely we find the images which are entirely red. So we have to grade the objects by amount of redness. Similarly, we might have to grade the objects based on other attributes as well. This leads us to the fact that by choosing an appropriate aggregation or scoring function we can compute the overall grade of the object. The objective of the authors of this paper was to build optimal aggregation algorithms for top-k selection.

1.2 Previous Work

Fagin has proposed an algorithm called Fagin’s Algorithm(FA) and it works as follows. It scans all the lists in parallel, and stops only when all the lists contain k common objects. Then it will compute the overall grade of all the objects that are seen. It will do random access for the objects whose grades to some of the lists are not available. Then it will return the top-k objects.

Fagin’s algorithm is optimal when the aggregation function is monotone and strict. A strict aggregation function can take maximum grade, only when all of it’s attributes takes maximum grade. Min is strict aggregation function, where as max is not. Fagin’s Algorithm works in a similar fashion irrespective of the what the aggregation function was. If the aggregation function is max then a trivial algorithm do at most mk sorted accesses(where m is the number of lists), which is better than Fagin’s algorithm, hence Fagin’s algorithm is not best for all the aggregation functions. Thus the main motivation for the authors is to propose new algorithms which are optimal under natural assumptions.
2 Proposed Solutions

The authors present a generic data model for Top-k query processing. The database that is available to the middle-ware consists of \( m \) sorted lists, where each list contains \( N \) entries for \( N \) objects. The entry is of the form \((R, X_i)\) where \( R \) is the object and \( X_i \) is its score in the \( i \)th list. We can access these lists by two modes, Sorted Access and Random Access.

Sorted Access means sequential access, and Random access means given an object, we can directly get its score in \( i \)th list without accessing the previous entries in that list. So the middle-ware cost of this system would be \( sC_S + rC_R \), where \( C_S \) is Sorted access cost and \( C_R \) is the random access cost. \( s \) is the no of sorted accesses and \( r \) is no of random accesses. And the main objective here is minimize the middle-ware cost, while returning the top-k objects. The authors have proposed Threshold Algorithm (TA) and it’s variants to serve this purpose.

**Threshold Algorithm (TA)** also performs sorted access to all the lists in parallel. But it won’t wait till all the lists contain common \( k \) objects. It calculates the threshold \((T)\) at each level. The threshold is calculated using the same scoring function, and by considering the last grade in each of the list. If we found \( k \)-objects that are greater than threshold at any level, the algorithm will stop.

Another variation of the algorithm proposed by the authors involve premature termination of the top-k selection process. This variant produces an approximate of the top-k results. We can call this as \( \theta \)-approximation. According to this algorithm, if we found \( k \)-objects whose grades are greater than or equal to the threshold multiplied by \( \theta \), we can terminate the algorithm. If \( \theta = 1 \), this algorithm becomes the regular TA algorithm.

In certain scenarios, sorted access to the lists are not possible. If we restrict sorted access to the lists, then we can modify TA to \( TA_z \) algorithm. The difference between TA and \( TA_z \) is that, in \( TA_z \) algorithm, we will perform sorted access to some of the lists( the lists we are allowed to do sorted access) and at each level, we calculate the threshold, by taking the maximum grade for the missing attributes, for which sorted access is not possible.

Until now we have assumed that random accesses are possible. What happens if there are no random accesses possible. When the sorted list comes from a search engine, then definitely random accesses are impossible. So another variant of TA is used when random accesses are impossible. And this algorithm is known as **No Random Access (NRA) Algorithm**. This algorithm works by calculating the best and worst scores of an object. The best score is calculated by taking the last object score seen in that list, for all missing attributes. Worst score is calculated by taking 0 for all missing attributes. This algorithm maintains two lists one is present top-k list, and a candidate list of all objects that can go into the top-k list. The algorithm stops when there are \( k \)-objects in the top-k list and there are no objects in the candidate list. NRA can output the top-k objects but it can not output the overall grades of these objects. If we want to retrieve the top-k objects in the order, then we have to run the algorithm for the top object, top-2 objects and so on up-to top-k objects.

The assumption NRA makes is that random accesses are impossible. Another consideration that is studies in this paper is what happens when the random accesses are not impossible but expensive. This leads to the proposal of the combined algorithm (CA), which combines TA and NRA. The basic idea of CA is use to run NRA, but after every \( k \) ((\([|C_R/C_S|]\)) steps perform the random access step. When \( C_R = C_S \) (i.e cost of a single random access is equal to the cost of the sigle sorted access), then CA turns into TA. When the \( \frac{C_R}{C_S} \) is very large then CA turns into NRA.

3 Evaluation Highlights

Though the authors have not evaluated their algorithms by implementing them and by producing subsequent results, they have proved that their algorithms are better than existing algorithms using the concept of instance optimality. Instance optimality is defined as follows:

**Instance Optimality:** An algorithm is instance optimal over a set of algorithms A and databases D, if its cost is equal to \( \mathcal{O}(\text{cost}(A,D)) \) for all A and D.

**TA vs FA:** The advantage of the threshold algorithm is that it wont perform more sorted access than FA. TA require only bounded buffers, since it always maintains top-k objects in the buffer, where as the buffers required by the Fagin’s algorithm will go arbitrarily large as the database size grows. The TA might require
more random accesses than FA. However, TA will stop immediately after seeing top-k objects. TA is instance optimal when the aggregation function is monotone where as FA is optimal when the aggregation function is monotone and strict.

TAz is instance optimal for monotone aggregation function, and it is tightly instance optimal (no algorithm has lesser optimality ratio) when the aggregation function is monotone and strict. NRA is also instance optimal, when the aggregation function is monotone. CA is instance optimal when the aggregation function is monotone, and it’s optimality ratio is in independent of $C_R/C_S$.

4 Discussion

In this paper the authors have presented a simple and elegant algorithm called TA. The authors do not take into account the use of non-monotonic aggregation functions for grading the objects. As such their treatment of aggregation functions are limited to monotonic or strictly monotonic functions. In case of non-monotonic aggregation functions the algorithms presented fail. The authors do not consider the effects of a distributed system especially with respect to network latency and fault tolerance. The authors also assume that the number of objects in consideration for a Top-k query remains constant during its execution. A similar assumption is made with respect to the number of attributes of the objects being queried. Furthermore, the authors have not considered the computational costs involved in calculating the overall grades of objects, sorting the objects based on their grades in their treatment. The algorithms presented are studies with reference to the cost of access of an objects attribute. Thus in practical scenarios these algorithms might take a longer time than expected as. The other important assumption mentioned in this paper is that all the lists can be accessed in parallel. They have not discussed about the data structures to implement their algorithms. Moreover the authors assume that the attribute values themselves are independent of each other, i.e. the value of one attribute does not affect the value of another. What if this is not the case? Case in point, we can consider a situation where there is a relationship between the minimum value in a list and the maximum value in another list. A valid extension of the algorithms presented in this paper might take into account this relationship and effectively improve the overall performance of the algorithms.

They have also assumed that sorted access cost to all the lists is same. They have not discussed what happens if this is not the case. While proving their algorithms are instance optimal, they have considered the terms $k$ and $m$ as constants. The authors have also not discussed what might happen if the user changes the query by adding one more attributes. In accordance with the proposed algorithm, the implication of such a change might entail the need to the process the lists from scratch. A natural extension of this would be the ability to reuse the computed information from previous queries. Another important assumption they have made is the grades of objects in all lists are normalized between $[0,1]$. They have not discussed the ways to normalize these grades. It turn out that score normalization is expensive. Consider the case where the scores are not normalized. In such a scenarios a normalization process might involve performing sorted accesses to all the objects in the various lists and applying a normalization function to them followed by storing these values into another list. This leads to an alternative algorithm that does not normalize the scores. Instead of using normalized scores, such an algorithm performs $n$ sorted accesses to the lists which have lesser scores per one sorted access to the lists which have higher scores [3], thus reducing the threshold at each level.

The authors do not discuss any improvements in performance that can be brought about by efficiently representing the datasets. In this paper, the datasets being considered are mainly lists. Subsequent research might look into the aspect of using the different data structures for efficiently representing these datasets, for example, a region query tree [4] can be used where the objects are bucketed based on a distance function from objects of interest. Also of interest is to study the effect of a block-serial access rather than a single-serial access [1]. This technique is highly likely to improve the performance of the algorithm when the data resides on disks, and where access is done in blocks. Similarly, another line of investigation might include the effect of caches and their organization on the algorithms.

References

