

# Progressive Ranking of Range Aggregates <sup>\*</sup>

Hua-Gang Li, Hailing Yu, Divyakant Agrawal, and Amr El Abbadi

University of California, Santa Barbara, CA 93106, USA  
{huagang, hailing, agrawal, amr}@cs.ucsb.edu

**Abstract.** Ranking-aware queries have been gaining much attention recently in many applications such as search engines and data streams. They are, however, not only restricted to such applications but are also very useful in OLAP applications. In this paper, we introduce *aggregation ranking* queries in OLAP data cubes motivated by an online advertisement tracking data warehouse application. These queries aggregate information over a specified range and then return the ranked order of the aggregated values. They differ from range aggregate queries in that range aggregate queries are mainly concerned with an aggregate operator such as SUM and MIN/MAX over the selected ranges of all dimensions in the data cubes. Existing techniques for range aggregate queries are not able to process aggregation ranking queries efficiently. Hence, in this paper we propose new algorithms to handle this problem. The essence of the proposed algorithms is based on both ranking and cumulative information to progressively rank aggregation results. Furthermore we empirically evaluate our techniques and the experimental results show that the query cost is improved significantly.

## 1 Introduction

Traditionally, databases handle unordered sets of information and queries return unordered sets of values or tuples. However, recently, the ranking or ordering of members of the answer set has been gaining in importance. The most prevalent applications include search engines where the qualifying candidates to a given query are ordered based on some priority criterion [3]; ranking-aware query processing in relational databases [14, 11, 2, 10, 4, 7]; and network monitoring where top ranking sources of data packets need to be identified to detect denial-of-service attacks [1, 9]. Ranking of query answers is not only relevant to such applications, but is also crucial for OnLine Analytical Processing (OLAP) applications. More precisely ranking of aggregation results plays a critical role in decision making. Thus, in this paper, we propose and solve aggregation ranking over massive historical datasets.

As a motivating example, consider an online advertisement tracking company <sup>1</sup>, where each advertiser places its advertisements on different publishers' pages, e.g., CNN and BBC. In general an advertiser is interested in identifying the "top" publishers in terms of total sales or number of clicks during a specific time period. For instance, during a period of last 30 days while a particular advertisement campaign was conducted, or during the period of 15 days preceding the new year. Such an advertising company would need to maintain a data warehouse which stores data cube information regarding the sales (or clicks) of the various publishers and advertisers, and where an

---

<sup>\*</sup> This research is supported by the NSF grants under IIS-23022, CNF-0423336, and EIA-00-80134

<sup>1</sup> The proposed research is motivated by a real need for such type of algorithmic support in an application that arises in a large commercial entity, an online advertisement tracking company.

advertiser would like to ask queries of the form: “*find the top-10 publishers in terms of total sales from Dec 15, 2003 to Dec 31, 2003*”. Based on existing techniques, first the total sales from Dec 15, 2003 to Dec 31, 2003 for each publisher needs to be computed. Then the total sales for all publishers are sorted to identify the top-10 publishers. We refer to such queries as *aggregation ranking*, since they aggregate information over a specified range and then return the ranked order of the results. An alternative example is in the context of the stock market data. For example, given the trade volume of each stock, an analyst might be interested in the top trades during a certain period.

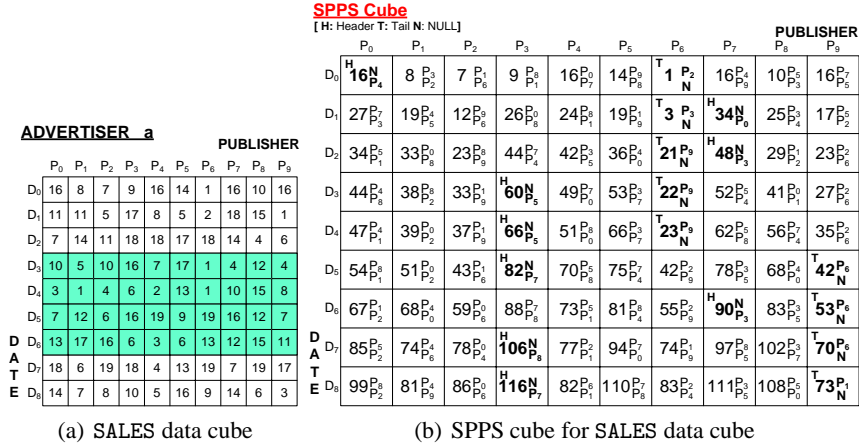
The problem of aggregation ranking is similar and yet differs from many related problems which have been addressed by the database and related research communities. We concentrate on the online analysis of massive amounts of data, which is similar to range aggregate queries prevalent in data warehouses. However, we are concerned with ranking of aggregated values over dimensions while prior research work on range aggregate queries has mainly concentrated on a single aggregate operator such as SUM and MIN/MAX over selected ranges of dimensions [6, 13]. To the best of our knowledge, this paper is the first attempt to address the ranking of aggregation in the context of OLAP applications. Our approach differs from the data stream research related to the TOP- $k$  operations [1, 9, 5] since the data is not continuously evolving. Moreover, queries in data streams are interested in more recent data. In contrast, our aggregation ranking queries can involve data in any arbitrary time range. In the context of relational databases, Bruno et al. [2] proposed to evaluate a top- $k$  selection by exploiting statistics stored in a RDBMS. Ilyas et al. [11, 10] proposed a new database operator, top- $k$  join, and efficiently implemented it using available sorting information of joined relations. This work addresses the optimization of top- $k$  selection and join operations in the context of relational databases. Our work, however, targets aggregation ranking queries in OLAP applications. In multimedia systems, Fagin [8] introduced ranking queries that combine information from multiple subsystems. Fagin’s algorithm can be directly applied if aggregates at multiple granularities (e.g. day, month, year) are considered. In particular aggregates on any specified range can be obtained by additions of multiple involved lists at different granularities. However when the number of involved lists grows large, Fagin’s algorithm tends to have a linear cost while our proposed algorithms in this paper always involve only two lists with sublinear cost. Furthermore, the framework in [8] is indeed useful for reasoning the correctness of our algorithms for aggregation ranking queries and therefore we adapt it to our context.

The rest of the paper is organized as follows. Section 2 gives the model and a motivating example. In Section 3, we present a new cube representation. Then we incrementally develop three different techniques for answering aggregation ranking queries in the following three sections, each of these improves a previous one. In Section 7 we empirically evaluate our proposed techniques and present the experimental results. Conclusions and future research work are given in Section 8.

## 2 Model and Motivating Example

In this paper we adopt the data cube [12] as a data model, A data cube can be conceptually viewed as a hyper-rectangle which contains  $d$  *dimensions* or *functional attributes* and one or more *measure attributes*. Dimensions describe the data space, and measure

attributes are the metrics of interest (e.g., sales volume). In particular, each cell in a data cube is described by a unique combination of dimension values and contains the corresponding value of the measure attribute. To introduce aggregation ranking queries, we assume that among the  $d$  functional attributes of a data cube, one of the functional attributes,  $A_r$ , is the *ranking functional attribute* and the rest  $d - 1$  functional attributes are the *range functional attributes*. An aggregation ranking query specifies ranges over the  $d - 1$  range functional attributes and requests a ranking of the values of the ranking functional attribute  $A_r$  based on the aggregated values of the measure attribute after applying some type of aggregation over the specified ranges.



**Fig. 1.** A SALES data cube and its SPPS cube

For instance, using the online advertisement tracking company example, we consider a 2-dimensional data cube SALES for a particular advertiser  $a$ , which has Publisher as the ranking functional attribute, Date as the range functional attribute and Sales as the measure attribute. Each cell in this data cube contains the daily sales of advertiser  $a$  through the advertisements placed on a publisher  $p$ 's website. Fig. 1(a) shows an example SALES data cube. A particular type of aggregation ranking query of interest to advertiser  $a$  in the SALES data cube is “find the top- $k$  publishers in terms of total sales from day  $D_s$  to  $D_e$ ”, and is specified as  $AR(k, D_s, D_e)$  for simplicity. The shaded area in Fig. 1(a) shows an instance of such a query from day  $D_3$  to  $D_6$ . Answering this kind of aggregation ranking queries with SUM operator efficiently is the focus of this paper. A basic way to answer such a query is to access each selected cell in the data cube to compute the total sales for each publisher within the time range from  $D_s$  to  $D_e$ . Then we sort the aggregated values to obtain the top- $k$  publishers. Since the number of involved cells is usually large, and data cubes are generally stored on disks, this will result in significant overhead. Also online sorting entails significant time overhead if there is a large number of publishers per advertiser. This in turn will impact the response time of interactive queries negatively.

### 3 Sorted Partial Prefix Sum Cube

In order to process aggregation ranking queries efficiently, we propose to use cumulative information maintained for each value of the ranking attribute  $A_r$  along the time

dimension. This is based on the prefix sum approach [6] which can answer any range aggregate query in constant time. Furthermore we pre-sort the values of  $A_r$  for each time unit based on the cumulative information. Hence a new cube presentation, *Sorted Partial Prefix Sum Cube* (SPPS cube in short), is developed. SPPS cube has exactly the same size as the original data cube. For simplicity of presentation, we will use the online advertisement tracking company example to explain our data structures and algorithms. The proposed algorithms can be generalized to handle data cubes with any arbitrary number of dimensions in a straightforward manner. An SPPS cube for the SALES data cube contains cumulative information along the DATE dimension for each publisher and daily order information along the PUBLISHER dimension. Each cell in the SPPS cube, indexed by  $(P_i, D_i)$ , maintains the following three types of information:

- PPSUM (Partial Prefix Sum): total sales for publisher  $P_i$  within the time range from  $D_0$  to  $D_i$ , i.e., cumulative sum since the initial time of the SALES data cube.
- PPC (Pointer to Previous Cell): a pointer to a cell in the same row of the SPPS cube which contains the least value no less than  $\text{SPPS}[P_i, D_i].\text{PPSUM}$ ; if such a pointer does not exist, PPC is set to NULL.
- PNC (Pointer to Next Cell): a pointer to a cell in the same row of the SPPS cube which contains the largest value no greater than  $\text{SPPS}[P_i, D_i].\text{PPSUM}$ ; if such a pointer does not exist, PNC is set to NULL.

PPC and PNC for all cells in a given row or time unit,  $D_i$ , maintain a *doubly linked list* in decreasing order of PPSUM. We refer to this list as  $\text{PPSUM}(D_i)$ . In addition we maintain two pointers pointing to the *header* and the *tail* of each doubly linked list for the SPPS cube. The header is the top ranked publisher based on the cumulative sales from the initial date of the SALES data cube and the tail is the bottom ranked publisher.

Fig. 1(b) shows the SPPS cube for the SALES data cube in Fig. 1(a). Each cell contains PPSUM, PPC, PNC information in the form of  $\text{PPSUM}_{\text{PNC}}^{\text{PPC}}$ . Also for presentation simplicity, we use the publisher index for pointers PPC/PNC. Note that the preprocessing of SPPS cube is offline, which is typical in real data warehousing applications. Space and offline processing are usually sacrificed for online interactive query processing.

## 4 Complete Scan Algorithm

We first present a simple algorithm, *complete scan*, to process aggregation ranking queries by using the pre-computed cumulative information in SPPS cubes. Given a query  $AR(k, D_s, D_e)$ , we need to obtain the total sales  $\text{SUM}(P_i, D_s, D_e)$  for each publisher  $P_i$  from  $D_s$  to  $D_e$ . This can be computed from  $\text{PPSUM}(D_i)$  maintained for each publisher  $P_i$  in the SPPS cube, which is actually given by the following subtraction:

$$\text{SUM}(P_i, D_s, D_e) = \text{SPPS}[P_i, D_e].\text{PPSUM} - \text{SPPS}[P_i, D_s - 1].\text{PPSUM}.$$

Collecting the total sales of all publishers between  $D_s$  and  $D_e$  together, we get a list denoted by  $\text{SUM}(D_s, D_e) = \{\text{SUM}(P_0, D_s, D_e), \dots, \text{SUM}(P_{n-1}, D_s, D_e)\}$ . Then the top- $k$  publishers during the period  $(D_s, D_e)$  can be easily extracted from the list  $\text{SUM}(D_s, D_e)$  as follows. Take the first  $k$  publishers from the SUM list, sort and store them into a list called *list-k*. For each publisher in the sum list ranging from  $k + 1$  to  $n$ , insert it into *list-k*, then remove the smallest publisher from *list-k*. Therefore the publishers in the

final list- $k$  are the top- $k$  publishers. The query cost is  $O(n + n \log k)$ . If  $k$  is a constant or  $k$  is much smaller than  $n$  ( $k \ll n$ ), the query cost is linear.

Note that the cost of the query is independent of the query range in the time dimension and is linearly dependent on the total number of publishers. Since the data cube is stored on disks, the cost of retrieving every publisher's information from disk can be relatively high. Furthermore an online advertisement tracking data warehouse serves a large number of advertisers at the same time. Thus the delay may not be acceptable for analysts who prefer interactive response time. In the next two sections, we extend the complete scan algorithm to improve the query cost by exploiting the ranking information maintained in the SPPS cube to minimize the number of publishers scanned.

## 5 Bi-directional Traversal Algorithm

In the complete scan algorithm, the first step computes the total sales for each publisher in a given time range, for which the best time complexity is linear. In order to reduce the total query cost, we need to avoid computing the entire SUM list. This is the premise of the bi-direction traversal algorithm discussed in this section.

The problem of evaluating aggregation ranking queries now reduces to the problem of combining two lists of ordered partial prefix sums corresponding to the given time range  $(D_s, D_e)$ , i.e.,  $\text{PPSUM}(D_s - 1)$  and  $\text{PPSUM}(D_e)$  respectively. Intuitively, for a given query  $AR(k, D_s, D_e)$ , the publishers which are in the query result must have relatively larger values in list  $\text{PPSUM}(D_e)$  and relatively smaller values in list  $\text{PPSUM}(D_s - 1)$ . Thus, instead of computing the entire list of  $\text{SUM}(D_s, D_e)$ , we may only need to compute the total sales of publishers which have higher ranking in  $\text{PPSUM}(D_e)$ , and lower ranking in  $\text{PPSUM}(D_s - 1)$  as long as the number of these publishers is large enough to answer the aggregation ranking query. Based on this intuition, we design the *bi-directional traversal algorithm* shown in Algorithm 1.

---

### Algorithm 1 Bi-directional Traversal Algorithm

---

```

1: Input:
2:  $AR(k, D_s, D_e)$ ;
3: Procedure
4:  $L_s = \phi, L_e = \phi$ ;
5:  $POINTER_s = \text{Tail of PPSUM}(D_s - 1)$ 
6:  $POINTER_e = \text{Header of PPSUM}(D_e)$ ;
7: while  $|L_s \cap L_e| < k$  do
8:    $L_s = L_s \cup POINTER_s.\text{publisher}$ ;
9:    $POINTER_s = POINTER_s.\text{PPC}$ 
10:   $L_e = L_e \cup POINTER_e.\text{publisher}$ ;
11:   $POINTER_e = POINTER_e.\text{PNC}$ 
12: end while
13: for each publisher  $P$  in  $L = L_s \cup L_e$  do
14:   Compute the total sales in  $[D_s, D_e]$  by  $\text{SPPS}[P, D_e].\text{PPSUM} - \text{SPPS}[P, D_s - 1].\text{PPSUM}$ ;
15:   Insert  $P$  into set  $R$ ;
16:   if  $|R| > k$  then
17:     Remove  $P_i$  from  $R$  if its  $\text{SUM}(P_i, D_s, D_e)$  is smaller than all other publishers in  $R$ ;
18:   end if
19: end for
20: End Procedure
21: Output:  $R$ ;

```

---

In the bi-directional traversal algorithm, we extract publishers concurrently from list  $\text{PPSUM}(D_e)$  in decreasing order (starting from the header of  $\text{PPSUM}(D_e)$ ) down to

the tail) into a list denoted by  $L_e$ , and from list  $\text{PPSUM}(D_s - 1)$  in increasing order (starting from the tail of  $\text{PPSUM}(D_s - 1)$  up to the header) into another list denoted by  $L_s$ , until the number of publishers in the intersection of their output sets  $L_s \cap L_e$  is no smaller than  $k$ . Hence scanning all the publishers is avoided. Then calculate the total sales of all publishers in  $L = L_s \cup L_e$ . Finally, compute the top- $k$  publishers in  $L = L_s \cup L_e$  based on their total sales during  $(D_s, D_e)$ . These top- $k$  publishers are actually the answer to the given query. The bi-directional traversal algorithm improves the processing cost of  $AR(k, D_s, D_e)$  to  $O(\sqrt{n})$  with arbitrarily high probability if the two lists  $\text{PPSUM}(D_s - 1)$  and  $\text{PPSUM}(D_e - 1)$  are independent and  $k$  is much smaller than  $n$ . Due to the space limit, please refer to [15] for further details of query cost analysis and the correctness proof of the bi-directional algorithm.

## 6 Dominant-Set Oriented Algorithm

The bi-directional traversal algorithm can answer a query  $AR(k, D_s, D_e)$  in  $O(\sqrt{n})$  with arbitrarily high probability for  $n$  publishers if the two lists  $\text{PPSUM}(D_s - 1)$  and  $\text{PPSUM}(D_e)$  are independent. Unfortunately in most real applications, this is not the case. For example, considering the online advertisement tracking data warehouse application, the two lists are independent if the probability of daily sales is not dependent on a specific publisher, i.e., if all publishers have similar and independent degree of popularity. However, in real world, some publishers are usually more popular than others. Thus the daily sales obtained through the advertisements placed on those publishers are much more than that of other publishers. Under such circumstances, the cumulative sales in lists  $\text{PPSUM}(D_s - 1)$  and  $\text{PPSUM}(D_e)$  for a publisher may not be completely independent. Therefore the probability that the query cost is  $O(\sqrt{n})$  becomes low. In particular, the worst case could happen when the two lists have almost the same set of publishers that always have the most daily sales. Fig. 2(a) shows such an example, where publishers  $P_0, P_1,$  and  $P_2$  always have more daily sales than the rest of the publishers. Given any query  $AR(k, D_s, D_e)$  over the data cube shown in Fig. 2(a), by using the bi-directional traversal algorithm, in order to get  $L_s \cap L_e \geq k$ , the number of publishers in  $L_s \cup L_e$  can be up to  $n$ . As a result, the query cost is almost linear. This is mainly because the bi-directional traversal algorithm is unable to minimize the size of a superset of the top- $k$  publishers efficiently in the presence of correlation among publishers and skewed distributions.

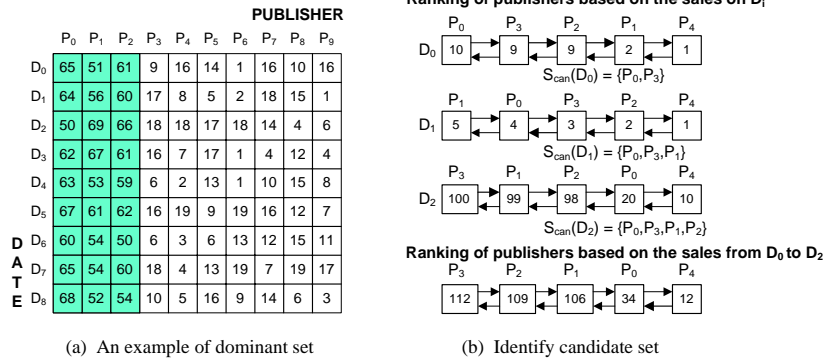


Fig. 2. An example of dominant set and how to identify candidate set

Hence, our goal now is to optimize the bi-directional traversal algorithm by pruning the search space in list PPSUM( $D_s - 1$ ). In order to do that, we need to identify the candidates for an aggregation ranking query. Without any doubt, dominant publishers usually dominate the top- $k$  slots and need to be considered in the candidate set. However some variations may occur, i.e., some non-dominant publishers may become dominant. Hence we need to identify such a set of candidates that may include the answer to an aggregation ranking query, for which we assume that all aggregation ranking queries  $AR(k, D_s, D_e)$  request a value of  $k$  no larger than  $k_{max}$  which is the maximum value of  $k$  specified in any aggregation ranking query. This is a realistic assumption, since advertisers are usually interested in a small number of publishers, especially those with a relatively high performance. We, therefore, assume that  $k_{max} \ll n$  and  $k_{max}$  is an application-dependent and user-defined parameter. We now introduce the notation of the *candidate set* for a day  $D_i$ , denoted as  $S_{can}(D_i)$ .  $S_{can}(D_0)$  is initialized to contain the top- $k_{max}$  publishers on the first day of the SALES data cube.  $S_{can}(D_i)$  contains all publishers in  $S_{can}(D_{i-1})$  and all publishers which are ranked on day  $D_i$  above any publisher in  $S_{can}(D_0)$  as well. We observe that  $S_{can}(D_{i-1}) \subseteq S_{can}(D_i)$ .

Consider the following example: assume there are 5 publishers  $P_0, P_1, P_2, P_3$  and  $P_4$  as shown in Fig. 2(b). We have the sales tracking information for three days  $D_0, D_1$  and  $D_2$ . Let  $k_{max}$  be 2.  $P_0$  and  $P_3$  ranked top-2 on day  $D_0$ . Hence  $S_{can}(D_0) = \{P_0, P_3\}$ . On day  $D_1$ ,  $P_1$  is ranked above  $P_3$  and  $P_3 \in S_{can}(D_0)$ , therefore,  $S_{can}(D_1) = \{P_0, P_3, P_1\}$ . Similarly  $S_{can}(D_2) = \{P_0, P_3, P_1, P_2\}$ . It is possible that a publisher which is ranked above any publisher in  $S_{can}(D_0)$  on day  $D_i$  could have a large total sales within some time range  $(D_s, D_i)$ . For example,  $P_2$  ranked top-2 in terms of total sales within  $(D_0, D_2)$ . Moreover, since  $P_4$  does not have a higher rank than the publishers in  $S_{can}(D_0)$  for any day, it is impossible to be a top-2 publisher for any aggregation ranking query.  $S_{can}(D_i)$  is a superset of the top- $k$  publishers for a given aggregation ranking query  $AR(k, D_s, D_i)$ , and the correctness is given in the following assertion<sup>2</sup>.

**Assertion 1** *For a given aggregation ranking query  $AR(k, D_s, D_e)$ , all the qualifying publishers must be contained in the candidate set for day  $D_e$ ,  $S_{can}(D_e)$ .*

From Assertion 1, we know that in order to answer a given query  $AR(k, D_s, D_e)$ , we need to consider all the publishers in  $S_{can}(D_e)$ . A straightforward solution is to obtain for each publisher  $p \in S_{can}(D_e)$  its prefix sum of sales from list PPSUM( $D_s - 1$ ) and list PPSUM( $D_e$ ). However this requires random accesses to both lists which results in a lot of random I/Os. In order to reduce the random accesses as well as consider all publishers in  $S_{can}(D_e)$ , we need to track the maximum index of all publishers in  $S_{can}(D_e)$  in list PPSUM( $D_s - 1$ ). We refer to this maximum index as the *pruning marker*. Note that the indices of cells in a list are in increasing order from the header to the tail. The header has an index of 0 and the tail has an index of  $n - 1$ . All publishers after the pruning marker in list PPSUM( $D_s - 1$ ) will be pruned as they do not qualify to be top- $k$  publisher candidates, and hence the search space is reduced.

However it is not efficient to compute the pruning marker online since finding the index of each publisher  $P \in S_{can}(D_e)$  in list PPSUM( $D_s - 1$ ) requires access to its corresponding cell in the SPPS cube. This can again degrade performance, especially

<sup>2</sup> Please refer to [15] for proof.

when the size of  $S_{can}(D_e)$  is large. Since  $S_{can}(D_s - 1)$  is a subset of  $S_{can}(D_e)$ , we can pre-process the publishers in  $S_{can}(D_i - 1)$  for each date  $D_i$  and store the index corresponding to the smallest ranked publishers in  $S_{can}(D_i - 1)$ , and then process the remaining publishers for a given query. Hence for each day  $D_i$ , in addition to  $S_{can}(D_i)$ , we maintain the maximum index in list PPSUM( $D_i$ ) of all publishers in  $S_{can}(D_i)$ . We refer to this index as  $IDX_{max}(D_i)$ . Please refer to [15] for the pseudo-code of computing  $S_{can}(D_i)$  and  $IDX_{max}(D_i)$ . Note that a data cube such as SALES is updated in an append-only fashion. When the new sales data of date  $D_i$  are appended to the data cube, we simply compute  $S_{can}(D_i)$  and  $IDX_{max}(D_i)$  based on  $S_{can}(D_{i-1})$  and  $S_{can}(D_0)$ .

We now show how to use  $S_{can}(D_i)$  and  $IDX_{max}(D_i)$  to reduce the list traversals of the bi-directional traversal algorithm, resulting in the *dominant-set oriented algorithm*. We first calculate a set of candidate publishers  $S_r$  that are in  $S_{can}(D_e)$  but not in  $S_{can}(D_s - 1)$ . The publishers in  $S_r$  may or may not be ranked higher than  $IDX_{max}(D_s - 1)$  which is pre-computed. Let  $idx_r$  be the maximum index of the publishers in  $S_r$ . Hence we need to identify the pruning marker  $PM$  which is  $\max(IDX_{max}(D_s - 1), idx_r)$ . Consider the example shown in Fig. 2(b). Given  $AR(2, D_1, D_2)$ ,  $S_r = S_{can}(D_2) - S_{can}(D_0) = \{P_1, P_2\}$ . The  $PM$  for list PPSUM( $D_0$ ) is the maximum value of  $idx_r$  and  $IDX_{max}(D_0)$ .  $idx_r$  in this case is 3 while  $IDX_{max}(D_0)$  is 1. Hence  $PM = 3$ . Thus publisher  $P_4$  can be pruned from the search space.

The rest of the dominant-set oriented algorithm is the same as the bi-directional traversal algorithm except that the starting point of traversing PPSUM( $D_s - 1$ ) is from the pruning marker  $PM$ . Again, due to the space limit, please refer to [15] for the pseudo-code of the algorithm. Since the dominant-set oriented algorithm prunes the search space in PPSUM( $D_s - 1$ ) by applying a pruning marker, it will always outperform the bi-directional traversal algorithm, especially when there is a dominant publisher set.

## 7 Experiments

We conducted extensive experiments over both synthetic and real datasets to evaluate our proposed techniques. The experimental results validated our assumptions regarding the characteristics of datasets. Due to the lack of space, we only present partial experimental results over the real data sets. Please refer to [15] for more performance evaluation.

The real clicks datasets are from CJ.com, an online advertisement tracking company. They are for a larger number of advertisers where the number of publishers for each advertiser ranges between 4,000 and 5,000. The maximum number of publishers is up to 100,000 for some advertisers, however for confidentiality, we were only supplied with the datasets restricted to about 4,000 to 5,000 publishers. Each clicks dataset contains the daily clicks of all publishers for about 180 days.

The experiments were conducted on a Pentium IV 1.6GHz PC with 256MB RAM and 30GB hard disk. We executed two sets of aggregation ranking queries: a *uniform query set* and a *biased query set*, each of them with 1,000 queries. The uniform query set contains queries whose ranges are uniformly generated along the DATE dimension, The biased query set contains queries which are generated to model real user query patterns. The details of generating such a biased query set can be found in [15]. The comparison of the different techniques is based on the average query time in milliseconds.

We conducted experiments over a large number of real clicks datasets of different advertisers to examine how the value of  $k$  affects query cost. The experiments exhibited similar results. Thus, here we only present the experimental results for an advertiser with 4,000 publishers and  $k_{max} = 50$ . Fig. 3(a) and Fig. 3(b) show the experimental results over a uniform query set and a biased query set respectively. We observe that the dominant-set oriented algorithm outperforms both the complete scan algorithm and the bi-directional traversal algorithm. The value of  $k$  does not affect the complete scan approach, since the value of  $k$  does not have any impact on this algorithm assuming that the output time can be ignored. The average query cost of the two other techniques tends to increase slightly when the value of  $k$  increases, since the number of publishers in  $L_s \cap L_e$  becomes larger and therefore results in a larger number of publishers in  $L_s \cup L_e$ . We also notice that the bi-directional traversal algorithm performs worse than the complete scan algorithm. This is because the real clicks datasets demonstrate to have a set of dominant publishers. Also this does not contradict the theoretical analysis as given in [15], which states that the bi-directional traversal algorithm at most needs to process all publishers and has linear performance in the worst case. Due to the dominant publishers, when using the bi-directional traversal algorithm, the number of publishers in  $L_s \cup L_e$  reaches  $n$  ( $n$  is the total number of publishers). Based on the Algorithm 1, in order to compute the total sales for each publisher in  $L_s \cup L_e$ , we need to randomly access the prefix sums of the sales for publishers that are in  $L_s$  but not in  $L_e$  or vice versa. Since the number of publishers in  $L_s \cup L_e$  is almost  $n$ , we need nearly  $n$  random accesses, which results in expensive disk I/O cost. However, in the complete scan algorithm, lists  $PPSUM(D_s - 1)$  and  $PPSUM(D_e)$  are always loaded into main memory sequentially thus taking advantage of the fast sequential access property of disks. As a result, the bi-directional traversal algorithm has worse performance than the complete scan algorithm even though they process almost the same number of publishers.

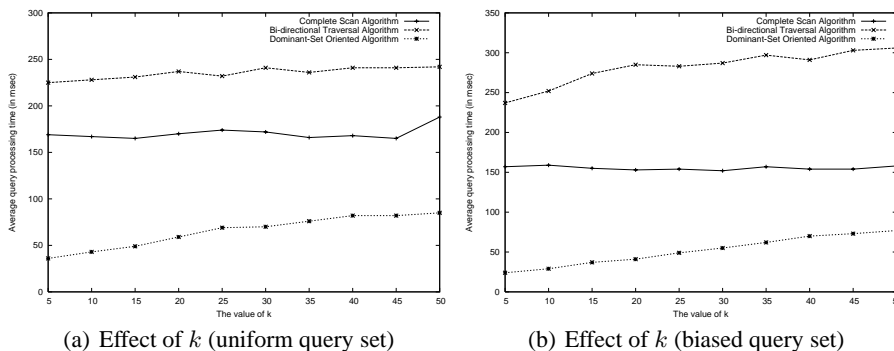


Fig. 3. Performance evaluation over real data set

## 8 Conclusion

In this paper, we formalized the notion of aggregation ranking for data warehouse applications. Aggregation ranking queries are critical in OLAP applications for decision makers in the sense that they provide ordered aggregation information. We have proposed a progression of three different algorithms to handle aggregation ranking queries.

Our final algorithm, the dominant-set oriented algorithm, is efficient and realistic, since it exploits the pre-computed cumulative information and the bi-directional traversal of lists while restricting the traversal to a small superset of the actual dominant set which is exhibited in real datasets. In general, with increasing reliance on online support for interactive analysis, there is a need to provide query processing support for complex aggregation queries in large data warehouses where sub-query results are correlated on a variety of metrics. Our future work will involve identifying such types of queries and developing database technologies for efficiently processing such queries. Furthermore, our proposed techniques can be generalized to handle aggregation ranking queries over high dimensional data cubes.

## References

1. Brian Babcock and Chris Olston. Distributed top-k monitoring. In *Proc. of Int. Conf. on Management of Data (SIGMOD)*, pages 563–574, 2003.
2. N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. on Database Systems*, 27(2):153–187, 2002.
3. N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web accessible databases. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 369–380, 2002.
4. K. C. Chang and S. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *Proc. of Int. Conf. on Management of Data (SIGMOD)*, pages 346–357, 2002.
5. M. Charikar, K. Chen, and M. Farach-Colton. Approximate frequency counts over data streams. In *Proc. of 29th Int. Colloq. on Automata, Languages and Programming*, pages 693 – 703, 2002.
6. C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *Proc. of Int. Conf. on Management of Data (SIGMOD)*, pages 73–88, 1997.
7. D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top N queries. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pages 411–422, 1999.
8. Ronald Fagin. Combining fuzzy information from multiple systems. In *Proc. of Symp. on Principles of Database Systems (PODS)*, pages 216–226, 1996.
9. L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. of the conference on Internet measurement conferenc*, pages 173–178, 2003.
10. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pages 950–961, 2002.
11. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pages 754–765, 2003.
12. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 152–159, 1996.
13. S. Y. Lee, T. W. Ling, and H.-G. Li. Hierarchical compact cube for range-max queries. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pages 232–241, 2000.
14. C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational topk queries. In *Proc. of Int. Conf. on Management of Data (SIGMOD)*, 2005.
15. H.-G. Li, H. Yu, D. Agrawal, and A. El Abbadi. Ranking aggregates. Technical Report 2004-07, University of California at Santa Barbara, <http://www.cs.ucsb.edu/research/trcs/docs/2004-07.pdf>, 2004.