Ranking Aggregates

Hua-Gang Li Hailing Yu Divyakant Agrawal Amr El Abbadi

University of California, Santa Barbara, 93106, USA
Compute Science Department
Santa Barbara, 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. For instance, an advertiser might be interested in the top-*k* publishers over the last three months in terms of sales obtained through the online advertisements placed on the publishers. 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 sets 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 [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 the 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 [1], 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 the last 30 day period while a particular advertisement campaign was conducted, or during the period of 15 days preceding the new year. An alternative example is in the context of historical stock market data analysis. Given the trade volume of each stock, an analyst might be interested in the top trades during a certain period of the year, while studying stock market trends.

In general such applications would need to maintain a data warehouse which stores data cube information. For the advertising company, the data cube would store the sales (or clicks) of the various publishers and advertisers. Consider an advertiser who would like to ask queries of the form: "*find the top-10 publishers in terms of total sales from December 15, 2003 to December 31, 2003*". Based on existing techniques, first the total sales from December 15, 2003 to December 31, 2003 for each publisher needs to be aggregated. 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.

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* values of a certain dimension based on the aggre-

[1] 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.

gates over ranges in other dimensions while prior research work on range aggregate queries concentrated on calculating a single aggregate 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 el 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. When the number of involved lists grows large, Fagin's algorithm tends to have a linear cost while the proposed algorithms in this paper always involve only two lists with sublinear cost. However, 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 since it provides fast access to information in a data warehouse. Data cubes are usually built by extracting "raw" data from different tables in relational databases and provide ready information for aggregation and summarization. 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. Note that in this paper the data cube model only represents an abstract way of *viewing* the data. It does not imply or exclude a certain physical data organization.

To introduce aggregation ranking queries, we assume that one of the functional attributes, A_t, of the data cube corresponds to the time dimension on which query ranges are specified. An aggregation ranking query specifies a range over the time dimension and requests a ranking based on the values of a particular functional attribute A_r, or based on the aggregated values of the measure attribute after applying some type of aggregation over a subset of the functional attributes. For instance, using the online advertisement tracking company example, we consider a data cube SALES which has Advertiser, Publisher, Date as dimensions or functional attributes and Sales as the measure attribute. Date is the time dimension. Each cell in this data cube contains the daily sales of an advertiser *a* through the advertisements placed on a publisher *p*'s website.

For simplicity we concentrate on a particular advertiser, a, and hence project data cube SALES along the ADVERTISER dimension for a specific advertiser a. Figure 1 shows the slice of data cube SALES corresponding to advertiser a. In the PUBLISHER dimension, we have 10 publishers for advertiser *a* indexed by $P_i (0 \leq i \leq 9)$ and in the DATE dimension, we have historical data for 9 days indexed by $D_i (0 \leq i \leq 8)$. The measure attribute value in each cell corresponds to the sales accrued by advertiser a, due to advertisements by publisher P_i on day D_i.

ADVERTISER *a*

DATE \ PUBLISHER	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
D_0	16	8	7	9	16	14	1	16	10	16
D_1	11	11	5	17	8	5	2	18	15	1
D_2	7	14	11	18	18	17	18	14	4	6
D_3	10	5	10	16	7	17	1	4	12	4
D_4	3	1	4	6	2	13	1	10	15	8
D_5	7	12	6	16	19	9	19	16	12	7
D_6	13	17	16	6	3	6	13	12	15	11
D_7	18	6	19	18	4	13	19	7	19	17
D_8	14	7	8	10	5	16	9	14	6	3

Figure 1: Slice of data in the data cube for advertiser *a*

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 D_{start} to D_{end}*", and is specified as $AR(k, D_{start}, D_{end})$. The shaded area in Figure 1 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 an aggregation ranking query is to access each selected cell in the data cube and compute the total sales for each publisher within the time range from D_{start} to D_{end}. Then sort the aggregated values to obtain the top-*k* publishers. We refer to this method as the *naïve algorithm*. 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 cumulative information of values of A_r for each time unit. 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. Thus the ranking attribute is `PUBLISHER` and the time dimension is `DATE`. The proposed algorithms can be generalized to additional dimensions in a straightforward manner. An SPPS 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 smallest value no less than SPPS$[P_i, D_i]$.`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 SPPS$[P_i, D_i]$.`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 `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 cumulative publisher and the tail is the bottom ranked cumulative publisher.

Figure 2 shows the SPPS cube constructed from the original data cube shown in Figure 1. Each cell contains the `PPSUM`, `PPC`, `PNC` information in the form of $\texttt{PPSUM}^{\texttt{PPC}}_{\texttt{PNC}}$. Also for presentation simplicity, we use the publisher index for pointers `PPC` and `PNC`. For example SPPS$[P_3, D_5]$.`PPSUM` is the sum of sales for publisher P_3 since the starting day of the data cube to day D_5, which is $9 + 17 + 18 + 16 + 6 + 16 = 82$. As the total sales for publisher P_3 so far is the largest among all the publishers, there is no `PPC` pointer and hence `PPC` is set to NULL and represented by 'N'. The `PNC` pointer points to the cell (P_7, D_5) whose `PPSUM` is the largest value among all the cells in the row corresponding to D_5 which values less than

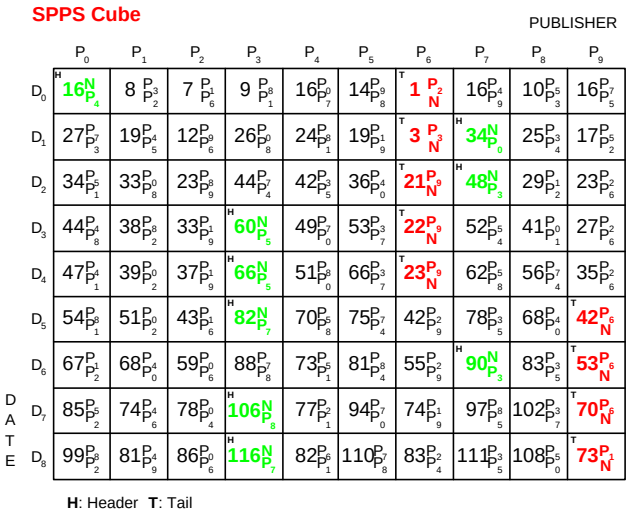

Figure 2: SPPS cube for the data cube shown in Figure 1

SPPS$[P_3, D_5]$.`PPSUM`. In the figure, the top ranked publishers are indicated by 'H', and the bottom ranked publishers are indicated by 'T'.

4 Complete Scan Algorithm

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

$$\texttt{SUM}(P_i, D_{start}, D_{end}) = \text{SPPS}[P_i, D_{end}].\texttt{PPSUM} - \text{SPPS}[P_i, D_{start} - 1].\texttt{PPSUM}.$$

Collecting the total sales of all publishers between D_{start} and D_{end} together, we get a list denoted by

$$\texttt{SUM}(D_{start}, D_{end}) = \{\texttt{SUM}(P_1, D_{start}, D_{end}), \ldots, \texttt{SUM}(P_n, D_{start}, D_{end})\}$$

The top-k publishers during the period (D_{start}, D_{end}) are in the list $\texttt{SUM}(D_{start}, D_{end})$ and can be easily extracted by sorting the list `SUM` (D_{start}, D_{end}), and reporting the top-k publishers from the sorted list. Thus the query cost is $O(n + n \log n)$, where n is the total number of publishers. The first term n in the query cost is due to the computation of the `SUM` list, and the second term $n \log n$ is the cost of sorting the `SUM` list. Instead of sorting the entire n publishers, we could 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 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 ($k << n$), 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, i.e., it needs to process every publisher. 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. Note that for clarity of presentation, we discuss our algorithms using a multidimensional database (MOLAP) model. Our algorithms are also applicable for relational tables (ROLAP), which will be discussed in Appendix A.

5.1 Query Processing Algorithm

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_{start}, D_{end}), i.e., $\texttt{PPSUM}(D_{start} - 1)$ and $\texttt{PPSUM}(D_{end})$ respectively. Intuitively, for a given query $AR(k,\ D_{start}, D_{end})$, the publishers which are in the query result must have relatively larger values in list $\texttt{PPSUM}(D_{end})$ and relatively smaller values in list $\texttt{PPSUM}(D_{start} - 1)$. Therefore, instead of computing the entire list of $\texttt{SUM}\ (D_{start}, D_{end})$, we may only need to compute the total sales of publishers which have higher ranking in $\texttt{PPSUM}(D_{end})$, and lower ranking in $\texttt{PPSUM}(D_{start} - 1)$ 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.

In the bi-directional traversal algorithm, we extract publishers concurrently from list $\texttt{PPSUM}(D_{end})$ in decreasing order (starting from the header of $\texttt{PPSUM}(D_{end})$ down to the tail) into a list denoted by L_{end}, and from list $\texttt{PPSUM}(D_{start} - 1)$ in increasing order (starting from the tail of $\texttt{PPSUM}(D_{start} - 1)$ up to the header) into another list denoted by L_{start}, until the number of publishers in the intersection of their output sets $L_{start} \cap L_{end}$ is no smaller than k (k is specified as an input parameter to the aggregation ranking query). Hence scanning all the publishers is avoided. Then calculate the total sales of all publishers in $L = L_{start} \cup L_{end}$. Finally, compute the top-*k* publishers in $L = L_{start} \cup L_{end}$ based on their total sales during (D_{start}, D_{end}). These top-*k* publishers are actually the answer to the given query.

Algorithm 1 Bi-directional Traversal Algorithm

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

5.2 Analysis

The correctness of the bi-directional traversal algorithm can be argued by extending Fagin's framework for combining fuzzy information from multiple multimedia subsystems [8]. Essentially, Fagin proposed an algorithm to find the top-*k* matching objects by combining decreasing sorted lists of objects from multiple subsystems using the *min* conjunction rule in fuzzy logic. However, in our bi-directional traversal algorithm, the query result is based on subtracting values of list $\texttt{PPSUM}(D_{start} - 1)$ from $\texttt{PPSUM}(D_{end})$ instead of based on their boolean combination. Hence we extend Fagin's framework [8] to establish the correctness of our aggregation ranking query processing.

For any aggregation ranking query Q, if the query results are based on the combined information of multiple sorted lists of all objects, each of which is the output of a subquery of Q, these lists can be classified into two types: *positive* and *negative* lists.

Definition 1 *Assume a query Q can be answered by a set of sub-queries $Q_i (1 \leq i \leq m)$, and each subquery Q_i returns a sorted list of objects referred to as L_i. A list L_i is* **positive** *if an object with a larger value under sub-query Q_i contributes a larger value of this object under Q. Likewise a list L_j is* **negative** *if an object with a larger value under sub-query Q_j contributes a smaller value of this object under Q.*

The combination rule which is used to combine the lists of objects from m subqueries to answer an aggregation ranking query is referred to as an *m-ary scoring function*. *Monotonicity* and *strictness* are two important properties associated with an m-ary scoring function, which are defined as follows.

Definition 2 *Monotonic: An m-ary scoring function t is monotonic if $t(\mu_1(x), \ldots, \mu_m(x)) \leq t(\mu_1(x'), \ldots, \mu_m(x'))$ when $\mu_i(x) \leq \mu_i(x')$ in every positive list L_i, and $\mu_j(x) \geq \mu_j(x')$ in every negative list L_j.*

Definition 3 *Strict: An m-ary scoring function t is strict if $t(\mu_1(x), \ldots, \mu_m(x))$ has the maximum value iff $\mu_i(x) = max_i$ in every positive list i, and $\mu_j(x) = min_j$ in every negative list L_j, where max_i is the maximum value of all objects over the positive list L_i, and min_j is the minimum value of all objects over the negative list L_j.*

An m-ary scoring function is strict if and only if there is an object which has the maximum values in all positive lists, and the minimum values over all negative lists. In our aggregation ranking queries, we combine lists $\texttt{PPSUM}(D_{start}-1)$ (negative) and $\texttt{PPSUM}(D_{end})$ (positive) by subtracting $\texttt{PPSUM}(D_{start}-1)$ from $\texttt{PPSUM}(D_{end})$. We refer to this as the *binary subtraction rule* which is defined as follows.

Definition 4 *Given two lists A, B of objects and an object x in both A and B. Let $\mu_A(x)$ and $\mu_B(x)$ be object x's values in lists A and B respectively. C is the combined list, and the combining rule is the* binary subtraction rule*: $\mu_C(x) = \mu_A(x) - \mu_B(x)$.*

The binary subtraction rule is an m-ary scoring function, and it is monotonic and strict, which is established in the following proposition.

Proposition 1 *The binary subtraction rule is monotonic and strict according to Definitions 2 and 3.*

Proof Assume $\mu_A(x) \leq \mu_A(x')$ and $\mu_B(x) \geq \mu_B(x')$. Thus $-\mu_B(x) \leq -\mu_B(x')$ and hence $(\mu_{A-B}(x) = \mu_A(x) - \mu_B(x)) \leq (\mu_{A-B}(x') = \mu_A(x') - \mu_B(x'))$. Thereby monotonicity is proved. Strictness can be easily established. If $\mu_A(x)$ is the maximum value in the positive subsystem and $\mu_B(x)$is the minimum value in the negative subsystem, it is obvious $\mu_{A-B}(x)$ has the maximum value of all objects.□

More importantly the binary subtraction rule has a property which guarantees the correctness of the bi-directional traversal algorithm. This property is given in the following proposition.

Proposition 2 *Let L_{start} be a sublist of $\texttt{PPSUM}(D_{start}-1)$ starting from the tail of $\texttt{PPSUM}(D_{start}-1)$, and L_{end} be a sublist of $\texttt{PPSUM}(D_{end})$ starting from the header of $\texttt{PPSUM}(D_{end})$. If the number of publishers in $L_{start} \cap L_{end}$ is no smaller than k, then the top-k publishers must be in $L_{start} \cup L_{end}$.*

Proof Assume there exists a publisher P which is not in $L_{end} \cup L_{start}$ but is one of the top-*k* publishers. Since at least *k* publishers are in $L_{end} \cap L_{start}$, there must exist at least one publisher P_x in $L_{end} \cap L_{start}$ which is not a top-*k* publisher.

Since L_{start} is a sublist of $\texttt{PPSUM}(D_{start}-1)$ which starts from the tail. P is not in L_{start} and $P_x \in L_{start}$.

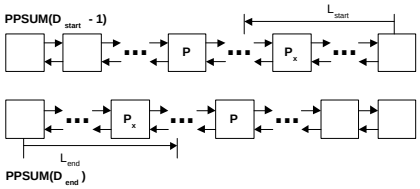

Figure 3: The relationships among L_{end}, L_{start}, P, and P_x

Their relative positions in $\texttt{PPSUM}(D_{start}-1)$ are shown in Figure 3. Thus, we have

$$\texttt{PPSUM}(D_{start}-1)[P] > \texttt{PPSUM}(D_{start}-1)[P_x]$$

Similarly, L_{end} is a sublist of $\texttt{PPSUM}(D_{end})$ which starts from the header, and P is not in L_{end} and $P_x \in L_{end}$. Then P and P_x have the following relation in $\texttt{PPSUM}(D_{end})$:

$$\texttt{PPSUM}(D_{end})[P_x] > \texttt{PPSUM}(D_{end})[P]$$

Therefore, the total sales of P in the time range from D_{start} to D_{end} is smaller than that of P_x,

$$\begin{aligned}&\texttt{PPSUM}(D_{end})[P] - \texttt{PPSUM}(D_{start}-1)[P] \\ &\quad < \texttt{PPSUM}(D_{end})[P_x] - \texttt{PPSUM}(D_{start}-1)[P_x]\end{aligned}$$

However, based on our assumption, the total sales of P in the time range is larger than that of P_x, since P is in the top-*k* publishers, and P_x is not. This leads to a contradiction. Therefore, the assumption is incorrect, i.e., publisher P can not be a top-*k* publisher.□

The bi-directional traversal algorithm is mainly designed to meet the needs of aggregation ranking queries. In particular, it only takes two lists into account. One is a positive list and the other is a negative list. In fact, it can be generalized to handle any number of ordered lists to meet the requirements of various applications which also address ranking-aware queries.

The query processing cost of the bi-directional traversal algorithm is given by the following proposition:

Proposition 3 *Given n publishers, if $\texttt{PPSUM}(D_{start}-1)$ and $\texttt{PPSUM}(D_{end})$ are independent, the bi-directional traversal algorithm can achieve query cost of*

$$P[cost(\texttt{TOP-}k) \leq \theta(nk)^{\frac{1}{2}}] \leq \theta^2$$

for every $\theta \geq 0$, where P stands for the probability.

Proposition 3 shows that the processing cost of an aggregation ranking query $AR(k, D_{start}, D_{end})$ is improved to $O(\sqrt{n})$ with arbitrarily high probability if the two lists $\texttt{PPSUM}(D_{start}-1)$ and $\texttt{PPSUM}(D_{end}-1)$ are independent and k is much smaller than n. The monotonicity and strictness properties of the binary subtraction rule guarantee the correctness of Proposition 3, which can be shown by using a proof similar to that given in [8]. Note that the combination rules of multiple lists for different applications could be different. However, as long as the combination rules have the monotonicity and strictness properties, the query processing cost can also be shown to be improved to $O(n^{\frac{m-1}{m}})$, where m is the total number of positive and negative lists.

6 Dominant-Set Oriented Algorithm

In this section we develop a practical aggregation ranking algorithm, *the dominant-set oriented algorithm*. This algorithm extends the bi-directional traversal algorithm by pruning the to-be-scanned publishers in practical settings where lists for different dates are not independent. The dominant-set oriented algorithm is much more efficient for real world applications.

6.1 Motivation

The bi-directional traversal algorithm can answer an aggregation ranking query $AR(k, D_{start}, D_{end})$ in $O(\sqrt{n})$ with arbitrarily high probability for n publishers if the two lists $\texttt{PPSUM}(D_{start}-1)$ and $\texttt{PPSUM}(D_{end})$ 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 degrees of popularity. However, in the 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 $\texttt{PPSUM}(D_{start}-1)$ and $\texttt{PPSUM}(D_{end})$ 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. Figure 4 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 $AR(k, D_{start}, D_{end})$ over the data cube shown in Figure 4, by using the bi-directional traversal algorithm, in order to get $L_{start} \cap L_{end} \geq k$, the number of publishers in $L_{start} \cup L_{end}$ can be up to n. For each publisher $p \in L_{start} \cup L_{end}$, we need to compute the total sales during the given date range. As a result, the query cost is 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. Hence, our goal now is to optimize the bi-directional traversal algorithm by pruning the search space in list $\texttt{PPSUM}(D_{start}-1)$ and by avoiding the need to always start from its tail.

In order to prune the search space of list $\texttt{PPSUM}(D_{start}-1)$, 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 be top-k publishers for some time ranges. Hence we need to identify such a set of candidates that may include the answer to an aggregation ranking query. In order to identify such a set of candidate publishers, we assume that an aggregation ranking query $AR(k, D_{start}, D_{end})$ requests 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 relatively high performance. We, therefore, assume that $k_{max} << n$ and k_{max} is an application-dependent and user-defined parameter.

PUBLISHER

DATE	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
D_0	65	51	61	9	16	14	1	16	10	16
D_1	64	56	60	17	8	5	2	18	15	1
D_2	50	69	66	18	18	17	18	14	4	6
D_3	62	67	61	16	7	17	1	4	12	4
D_4	63	53	59	6	2	13	1	10	15	8
D_5	67	61	62	16	19	9	19	16	12	7
D_6	60	54	50	6	3	6	13	12	15	11
D_7	65	54	60	18	4	13	19	7	19	17
D_8	68	52	54	10	5	16	9	14	6	3

Figure 4: An example of dominant set (shaded area)

6.2 Determining the Candidate Set

We now introduce the notation of the *candidate set* for a day D_i, denoted as $S_{can}(D_i)$. The candidate set $S_{can}(D_0)$ is initialized to contain the top-k_{max} publishers on the first day of the SALES data cube. The candidate set for day D_i, $S_{can}(D_i)$ 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 $S_{can}(D_{i-1}) \subseteq S_{can}(D_i)$.

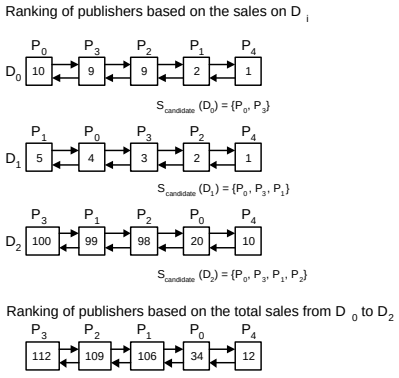

Figure 5: Identifying candidate sets

Consider the following example: assume there are 5 publishers P_0, P_1, P_2, P_3 and P_4 as shown in Figure 5. 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_{start}, 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 $AR(k, D_{start}, D_i)$, and the correctness is given in the following assertion.

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

Proof This assertion is equivalent to the following statement: any publisher P_x which is not contained in $S_{candidate}(D_{end})$ is not a qualifying publisher for $AR(k, D_{start}, D_{end})$, where $D_{start} \leq D_{end}$. Since P_x is not in $S_{candidate}(D_{end})$, the sales of P_x on any day D_i between D_{start} and D_{end} must be no larger than that of any publisher which is in the initial candidate set $S_{candidate}(D_0)$. For any publisher $P_y \in S_{candidate}(D_0)$, $\texttt{SALES}(P_x, D_i) \leq \texttt{SALES}(P_y, D_i)$ for all $D_0 \leq D_i \leq D_{end}$. Hence the total sales for P_x between D_{start} and D_{end} are no greater than the total sales for P_y. As $S_{candidate}(D_{end})$ contains all the publishers in $S_{candidate}(D_0)$ and the total number of publishers in $S_{candidate}(D_0)$ is k_{max}, publisher P_x cannot be ranked top-*k* for the given query $AR(k, D_{start}, D_{end})$. □

Algorithm 2 Compute $S_{candidate}(D_i)$ and $IDX_{max}(D_i)$

```
1:  Input:
2:  D_i;
3:  k_max;
4:  S_candidate(D_0);
5:  S_candidate(D_{i-1});
6:  P(D_i); // sorted publishers based on their sales of day D_i (in decreasing
    order);
7:  PPSUM(D_i);
8:  Procedure
9:  S_candidate(D_i) = φ
10: if D_i == D_0 then
11:    for every publisher P in P(D_i) whose index ≤ k_max do
12:       S_candidate(D_i) = S_candidate(D_i) ∪ {P};
13:    end for
14: else
15:    idx = −1;
16:    for every publisher P in S_candidate(D_0) do
17:       idx_P = the index of P in P(D_i);
18:       if idx_P > idx then
19:          idx = idx_P;
20:       end if
21:    end for
22:    S_candidate(D_i) = S_candidate(D_{i-1});
23:    for every publisher P in P(D_i) whose index ≤ idx do
24:       S_candidate(D_i) = S_candidate(D_i) ∪ {P};
25:    end for
26: end if
27: IDX_max(D_i) = −1
28: for every publisher P ∈ S_candidate(D_i) do
29:    idx_P = the index of P in list PPSUM(D_i);
30:    if idx_P > IDX_max(D_i) then
31:       IDX_max(D_i) = idx_P
32:    end if
33: end for
34: Output: S_candidate(D_i) and IDX_max(D_i);
35: End Procedure
```

From Assertion 1, we know that in order to answer a given aggregation ranking query $AR(k, D_{start}, D_{end})$, we need to consider all the publishers in $S_{can}(D_{end})$. A straightforward solution is to obtain for each publisher $p \in S_{can}(D_{end})$ its prefix sum of sales from list $\texttt{PPSUM}(D_{start} - 1)$ and list $\texttt{PPSUM}(D_{end})$. 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 $S_{can}(D_{end})$ during query processing, we need to track the maximum index of all publishers in $S_{can}\ (D_{end})$ in list $\texttt{PPSUM}(D_{start} - 1)$. 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 $\texttt{PPSUM}(D_{start} - 1)$ 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 $P \in S_{can}(D_{end})$ in list $\texttt{PPSUM}(D_{start} - 1)$ requires access to its corresponding cell in the SPPS cube. This can again degrade performance, especially when the size of $S_{can}(D_{end})$ is large. Since $S_{can}\ (D_{start} - 1)$ is a subset of $S_{can}(D_{end})$, 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 $\texttt{PPSUM}(D_i)$ of all publishers in $S_{can}(D_i)$. We refer to this index as $IDX_{max}(D_i)$.

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 $\texttt{IDX}_{\texttt{max}}(D_i)$ based on $S_{can}(D_{i-1})$ and $S_{can}(D_0)$. The algorithm is illustrated in Algorithm 2.

6.3 Dominant-set Oriented Query Processing

For an aggregation ranking query $AR(k, D_{start}, D_{end})$, we 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* given in Algorithm 3.

In the dominant-set oriented algorithm, we first calculate a set of candidate publishers S_{remain} that are in $S_{can}(D_{end})$ but not in $S_{can}(D_{start} - 1)$, i.e., $(S_{remain} = S_{can}(D_{end}) - S_{can}\ (D_{start} - 1))$. The publishers in S_{remain} may or may not be ranked higher than $IDX_{max}(D_{start} - 1)$ which is pre-computed. Let idx_{remain} be the maximum index of the publishers in S_{remain}. Hence we need to identify the pruning marker PM which is $\max(IDX_{max}(D_{start} - 1), idx_{remain})$.

Consider the example shown in Figure 5. Given an aggregation ranking query $AR(2, D_1, D_2)$, $S_{remain} = S_{can}(D_2) - S_{can}(D_0) = \{P_1, P_2\}$. The pruning marker PM for list $\texttt{PPSUM}(D_0)$ is the maximum value of idx_{remain} and $IDX_{max}(D_0)$. idx_{remain} 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 of list $\texttt{PPSUM}(D_0)$.

Algorithm 3 Dominant-set Oriented Algorithm

Input:
$AR(k, D_{start}, D_{end})$;
$S_{can}(D_{end})$;
$S_{can}(D_{start}-1)$;
$IDX_{max}(D_{start}-1)$;
Procedure
$PM = IDX_{max}(D_{start}-1)$; // PM is the pruning marker;
$S_{remain} = S_{can}(D_{end}) - S_{can}(D_{start}-1)$;
for each publisher P in S_{remain} **do**
  $IDX_P = P$'s index in list PPSUM$(D_{start}-1)$
  if $IDX_P > PM$ **then**
    $PM = IDX_P$;
  end if
end for
$L_{start} = \phi, L_{end} = \phi$;
$POINTER_{start}$ = The cell in PPSUM$(D_{start}-1)$ with index PM
$POINTER_{end}$ = Header of PPSUM(D_{end});
while $\mid L_{start} \cap L_{end} \mid < k$ **do**
  $L_{start} = L_{start} \cup POINTER_{start}$.publisher;
  $POINTER_{start} = POINTER_{start}$.PPC
  $L_{end} = L_{end} \cup POINTER_{end}$.publisher;
  $POINTER_{end} = POINTER_{start}$.PNC
end while
for each publisher P in $L = L_{start} \cup L_{end}$ **do**
  Compute the total sales in $[D_{start}, D_{end}]$ by SPPS$[P, D_{end}]$.PPSUM − SPPS$[P, D_{start}-1]$.PPSUM;
  Insert P into set R;
  if $\mid R \mid > k$ **then**
    Remove P_i from R if its SUM$(P_i, D_{start}, D_{end})$ is smaller than all other publishers in R;
  end if
end for
End Procedure
Output: R

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 $(D_{start}-1)$ is from the pruning marker PM. Since the dominant-set oriented algorithm prunes the search space in PPSUM$(D_{start}-1)$ by applying a pruning marker, it will always outperform the bi-directional traversal algorithm, especially when there is a dominant publisher set.

We observe that for the dominant-set oriented algorithm, if idx_{remain} is far greater than $IDX_{max}(D_{start}-1)$, we may consider too many non-qualifying publishers in list PPSUM$(D_{start}-1)$ within the range from $IDX_{max}(D_{start}-1)$ to idx_{remain}. Since these publishers are never top-k_{max} publishers, they can be pruned if we alternatively randomly access the publishers in S_{remain} one by one to reduce the query cost, and then only sequentially access PPSUM$(D_{start}-1)$ from the cell indexed by $IDX_{max}(D_{start}-1)$.

In calculating $S_{can}(D_i)$, we use the top-k_{max} publishers on day D_0 as the initial candidate set. Since the initial ranking may not be representative of the system behavior, some non-dominant publisher that happens to rank in the top-k_{max} on day D_0 will be contained in $S_{can}(D_0)$. This may result in large-sized candidate sets and hence longer query processing time. In order to avoid this, we propose an alternative practical approach for initializing the candidate set. Given k_{max}, we take the top-k_{max} publishers in list PPSUM(D_x) as the initial candidate set, where D_x is a date which is advanced enough to approximately determine the initial candidate set. In our experiments, we estimate the initial candidate set using the total sales of all publishers after a month. After deciding on the initial candidate set, we compute the candidate set for each day using the same procedure as illustrated in Section 6.2. The correctness of the dominant-set oriented algorithm still holds. In particular assume that the SALES data cube starts from an initial day $D_{initial}$ instead of D_0, all publishers on $D_{initial}$ have zero sales, and the candidate set for $D_{initial}$ contains the top-k_{max} publishers from list PPSUM(D_x). Assertion 1 holds for range $[D_{initial}, D_{max}]$ since every publisher on day $D_{initial}$ has zero sales, their values do not have any impact on the ranking of publishers for any day. Also Assertion 1 is correct for range $[D_0, D_{max}]$. In our experiments over the real datasets, we implement the dominant-set oriented algorithm using this method. The experimental results show its effectiveness.

7 Experiments

We conducted extensive experiments to evaluate the proposed techniques and to validate our assumptions regarding the characteristics of datasets. In this section, we first describe the datasets used. Then we analyze our experimental results.

7.1 Dataset Descriptions

In the experiments, we use both synthetic and real datasets to evaluate the performance of the three techniques proposed. We generate two different kinds of synthetic datasets: *near-random dataset* and *dominant-set dataset*. The real datasets are from anonymous.com [2], an online advertisement tracking company. The descriptions of datasets are as follows.

Near-random Datasets

Near-random datasets are generated to show how well the bi-directional traversal algorithm preforms on datasets without any correlation, i.e., any two rows in the SPPS cube are not correlated to each other. Since an SPPS cube contains the cumulative information of its original data cube, if we randomly (uniformly) generate the original SALES data cube, the cumulative sales of all publishers will be almost identical as the time range becomes large. In order to avoid this problem, we generate the data in reverse order: the cumulative information is generated first. In fact, as each publisher expects its daily sales to increase, the daily sales would vary in a larger range as time passes. Based on this observation, when we generate the cumulative data, the value on day D_i is generated randomly in a larger range than that on day D_{i-1} but should not be smaller than the value on day D_{i-1}. We refer to the datasets generated in this way as *near-random datasets*.

Dominant-set Datasets

As mentioned earlier, in the real world, some publishers often have more sales than others. Under this condition, we optimized the bi-directional traversal algorithm and proposed the dominant-set oriented algorithm. Hence

[2] Identity hidden to maintain anonymity.

we generate *dominant-set datasets* to test the dominant-set oriented algorithm. We use random walks to generate the data for each publisher, which is given as follows:

$$D[0] = s_{base} \text{ and } D[i] = D[i-1] + s_i.$$

s_{base} is the initial value and s_i is a random number in the range $[-r, +r]$. r is set to $c \times s_{base}$ where c is a constant which is less than or equal to 5%. We divide the publishers into three groups: (1) Dominant publishers (2) Non-dominant publishers (3) In-between publishers. Dominant publishers have a relatively large value of s_{base} while non-dominant publishers have a relatively small value of s_{base}. In order to model real world datasets with dominant publisher sets, we add some in-between publishers with a value of s_{base} between the values of s_{base} for the dominant and non-dominant publishers. Using the random walk method, we can guarantee that the generated datasets have dominant publisher sets and some variations, i.e., some non-dominant and in-between publishers can become dominant ones.

Real Clicks Datasets

The real clicks datasets are for a large number of advertisers where the number of publishers for each advertiser varies between 4,000 and 5,000. The maximum number of publishers is up to 10^5 for some advertisers, however for confidentiality, the datasets are restricted to about 4,000 to 5,000 publishers. Each clicks dataset contains the daily clicks of all publishers for about 180 days.

7.2 Experimental Results

The experiments were conducted on a Pentium IV 1.6GHz PC with 256MB of main memory and 30GB hard disk. Since data is stored on disk, in our experiments, we take the sequential access property of disks into account. Current disk technology favors sequential accesses instead of random accesses, because random accesses result in longer seek times. Therefore, in our experiments, we divide lists into chunks [14] each containing a large number of cells. The data is loaded into main memory by units of chunks instead of cells.

We generated the synthetic datasets by varying the `PUBLISHER` dimension size from 10,000 to 90,000 and the `DATE` dimension size is one year (365 days). We then precomputed the SPPS cubes and candidate set $S_{can}(D_i)$ for each day D_i together with $IDX_{max}(D_i)$. We compared our proposed techniques together with the naïve algorithm as discussed in Section 2 in two settings: one examines the effect of the `PUBLISHER` dimension size on the query cost and the other examines the effect of the value of k (used as the top-k publishers of interest in the aggregation ranking query) on the query cost. We executed two sets of aggregation ranking queries, each of them with 1,000 queries. The comparison of the different techniques is based on the average query time in milliseconds over 1,000 queries in a uniform query set and a biased query set.

The first set of aggregation ranking queries are uniformly generated along the `DATE` dimension, referred to as the *uniform query set*. The second one is generated to model real user query patterns, which is described as follows: Let $DATE_{min}$ be the initial date of the `SALES` data cube, $DATE_{max}$ be the most recent date, and $DATE_x$ and $DATE_y$ be randomly determined dates. The ranges over the `DATE` dimension were selected as follows:

1. $[DATE_{min}, DATE_x]$ with 10% probability, i.e., queries starting from the initial date;
2. $[DATE_x, DATE_y]$ with 20% probability, i.e., queries with randomly generated starting date and ending date;
3. $[DATE_x, DATE_x]$ with 10% probability, i.e., queries for a single date;
4. $[DATE_x, DATE_{max}]$ with 60% probability, i.e., queries ending at the most recent date;

The rationale for using such a query set is because queries for the immediate past are much more likely than other time ranges. We refer to this set of aggregation ranking queries as the *biased query set*.

7.2.1 Near-random Datasets

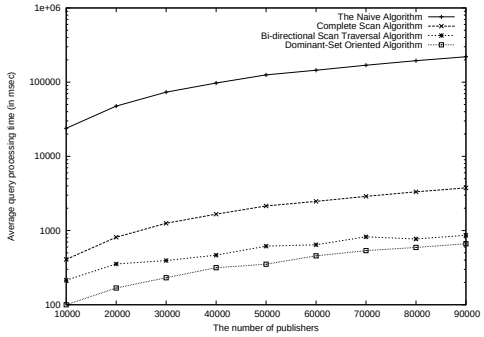

Figure 6: Effect of the `PUBLISHER` dimension size over near-random datasets using uniform query set

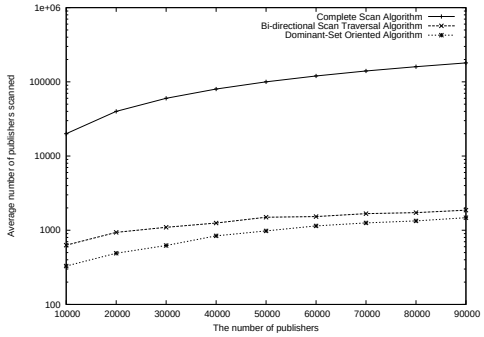

Figure 7: Average number of publishers scanned using uniform query set

The first setting examines how the `PUBLISHER` dimension size affects the performance of the proposed algorithms. The value of k is set to 10, i.e., obtaining the top-10 publishers and the value of k_{max} is set to 15. Figure 6

shows the experimental results for the aggregation ranking queries from the uniform query set over the near-random datasets. In order to show the effectiveness of our proposed algorithms, we compare them with the naïve solution where every cell involved in a query is accessed. As shown in Figure 6, our proposed methods result in significant improvements over the naïve algorithm. In our later experiments, we will focus on comparing our proposed algorithms without showing the results for the naïve algorithm. In Figure 6, as the number of publishers increases, the response time of the complete scan algorithm increases dramatically while that of the bi-directional traversal algorithm and the dominant-set oriented algorithm only increases marginally. The performance of the bi-directional traversal algorithm and the dominant-set oriented algorithm is much better than that of the complete scan algorithm. We can observe that for the near-random datasets, the dominant-set oriented algorithm always outperforms the bi-directional traversal algorithm since it improves the bi-directional traversal algorithm by pruning the search space. The results shown in Figure 6 coincide with our theoretical analysis in Section 5.2. Since the dataset is generated in a near random manner, the number of publishers in $L_{start} \cup L_{end}$ is much smaller than the total number of publishers and the bi-directional traversal algorithm performs very well. Furthermore the improvement of the dominant-set oriented algorithm is only marginal. Their relative performance can be explained by the number of publishers scanned. Figure 7 shows the average number of publishers scanned in these three algorithms. The bi-directional traversal algorithm and the dominant-set oriented algorithm scanned much fewer publishers than the complete scan algorithm. Since there is no clear dominant-set in the near-random datasets, the number of publishers scanned in the dominant-set oriented algorithm is comparable to that in the bi-directional traversal algorithm.

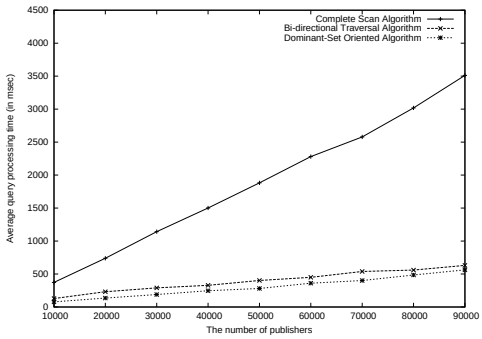

Figure 8: Effect of the PUBLISHER dimension size over near-random datasets using biased query set

The experimental results obtained by using the biased query set are shown in Figure 8. The relative behavior of the three algorithms is very similar to the ones presented in Figure 6 except that their absolute values are smaller. This is mainly because some aggregation ranking queries, such as those of the form $[DATE_{min}, DATE_x]$ in the biased query set, can be answered in constant time since only a small number of publishers from list $\texttt{PPSUM}(DATE_x)$ is retrieved from disk. Similar behavior was observed for the different datasets, therefore we will not show the results using the biased query set.

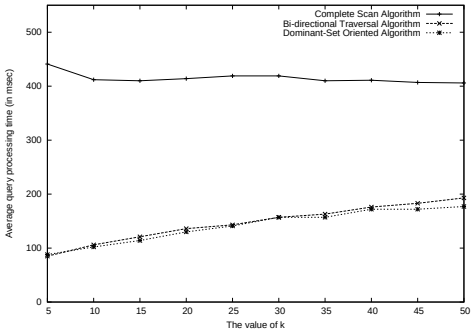

Figure 9: Effect of k over near-random datasets using uniform query set

We now examine how the value of k affects the performance of the proposed algorithms. We conducted this experiment on the near-random dataset with PUBLISHER dimension size 10,000 and $k_{max} = 50$. We varied the value of k from 5 to 50. We also executed aggregation ranking queries from both the uniform query set and the biased query set. Figure 9 shows the results for the uniform query set. The bi-directional traversal algorithm and the dominant-set oriented algorithm always outperform the complete scan 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 increased slightly when the value of k increases, since the number of publishers in $L_{start} \cap L_{end}$ becomes larger and therefore results in a larger number of publishers in $L_{start} \cup L_{end}$. In Figure 9, the cost difference between the bi-directional traversal algorithm and the dominant-set oriented algorithm with 10,000 publishers is smaller than that shown in Figure 6. This is because k_{max} is 50 for the second experimental setting.

Based on the experimental results over the near-random datasets, it is not surprising that the dominant-set oriented algorithm always outperforms the bi-directional traversal algorithm as the dominant-set oriented algorithm is an optimization of the bi-directional traversal algorithm.

7.2.2 Dominant-set Datasets

On dominant-set datasets, we performed similar experiments. We first show the behavior of the three proposed algorithms when varying the number of publishers from 10,000 to 90,000, and $k_{max} = 15$. The value of k is set to 10 for both aggregation ranking queries from the uniform query set and the biased query set. Figure 10 shows the experimental results using the uniform query set. The dominant-set oriented algorithm outperforms both the complete scan algorithm and the bi-directional traversal algo-

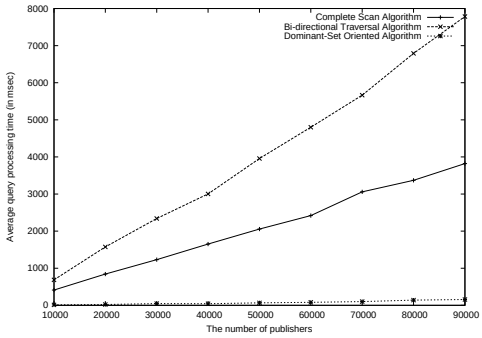

Figure 10: Effect of the PUBLISHER dimension size over dominant-set datasets using uniform query set

rithm by about two orders of magnitude. As the number of publishers increases, the average query cost of the dominant-set oriented algorithm slightly increases while the cost of the other two algorithms increases linearly. We also notice that the bi-directional traversal algorithm performs worse than the complete scan algorithm over the dominant datasets. However this does not contradict the theoretical analysis given in Section 5.2, 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 $L_{start} \cup L_{end}$ reaches n (n is the total number of publishers). Based on Algorithm 1, in order to compute the total sales for each publisher in $L_{start} \cup L_{end}$, we need to randomly access the prefix sums of the sales for publishers that are in L_{start} but not in L_{end} or vice versa. Since the number of publishers in $L_{start} \cup L_{end}$ is almost n, we need nearly n random accesses, which results in expensive disk I/O cost. However, in the complete scan algorithm, lists $\texttt{PPSUM}(D_{start} - 1)$ and $\texttt{PPSUM}(D_{end})$ 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 both algorithms process almost the same number of publishers.

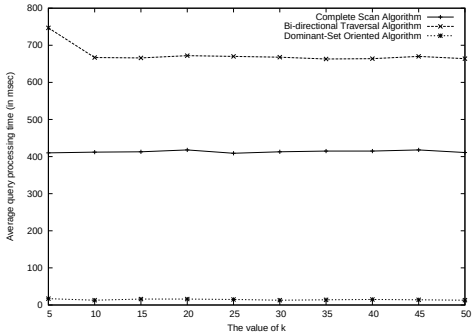

Figure 11: Effect of k over dominant-set datasets using uniform query set

In order to examine the effect of k's values on the average query cost, we generated a dominant-set dataset with 10,000 publishers and $k_{max} = 50$. Figure 11 presents the results using the uniform query set for different values of k. The query cost of the dominant-set oriented algorithm and the complete scan algorithm does not change for different values of k. The experimental results for the biased query set are similar and we do not include them in the paper.

7.2.3 Clicks Datasets

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 $k_{max} = 50$.

Figure 12 shows the experimental results for a uniform query set. The behaviors of the three proposed algorithms are similar to those over the synthetic dominant-set datasets. The dominant-set oriented algorithm performs much better than the other two algorithms. Again the bi-directional traversal algorithm performs worse than the complete scan algorithm due to the large number of expensive random accesses to disks.

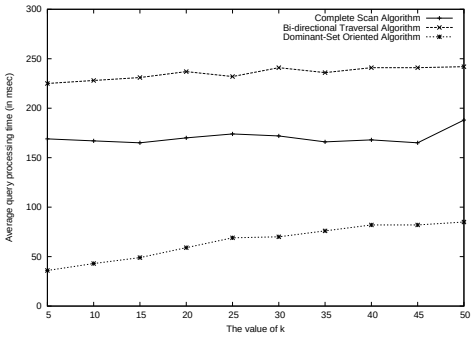

Figure 12: Effect of k over the real dataset using uniform query set

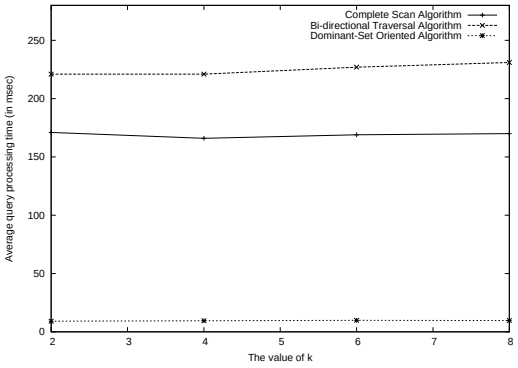

Figure 13: Effect of k over the real dataset using uniform query set and $k_{max} = 8$

We observe that the improvement of the dominant-set oriented algorithm over the other two algorithms for the

real clicks datasets is not as significant as that over the synthetic dominant-set datasets. This can be explained by the relation between the value of k_{max} and the number of the dominant publishers. In the experiments conducted over the real clicks dataset, k_{max} is set to 50 which is larger than the actual number of dominant publishers. In another words, some non-dominant publishers are considered in the initial candidate set, which will result in a larger size of the candidate sets and hence more expensive query cost. This analysis can be supported by the results shown in Figure 13. We set k_{max} equal to the number of dominant publishers, 8. The dominant-set oriented algorithm significantly outperforms the other two algorithms by a much larger margin. Overall the real dataset experiments validate both our theoretical analysis and the results of our experiments conducted on synthetically generated datasets. Furthermore they validate our hypothesis that the dominant-set oriented algorithm will give excellent performance with real datasets.

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. We empirically evaluated our proposed algorithms using both synthetic and real datasets from an online advertisement tracking company. The experimental results confirm our theoretical development that the dominant-set oriented algorithm evaluates aggregation ranking queries efficiently in practical real data settings. 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. Intl. Conf. on Managment 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. Intl. Conf. on Data Eng.(ICDE)*, pages 369–380, 2002.

[4] K. C. Chang and S. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[5] M. Charikar, K. Chen, and M. Farach-Colton. Approximate frequency counts over data streams. In *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. Int. Conf. on Management of Data (SIMGMOD)*, pages 73–88, 1997.

[7] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top N queries. In *In Proc of the 25 th VLDB Conf.*, pages p411–422, 1999.

[8] Ronald Fagin. Combining fuzzy information from multiple systems. In *Proc. 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 *VLDB'02*, 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. Int. Conf. on Very Large Data Base (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. Int. Conf. on Data Eng.(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. Int. Conf. on Very Large Databases (VLDB)*, pages 232–241, 2000.

[14] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algortihm for simultaneous multidimensional aggregates. In *Proc. Intl. Conf. on Managment of Data (SIGMOD)*, pages 159–170, 1997.

Appendix A Relational Framework

Our algorithms are presented in the MOLAP context. However, they can be easily adapted to the relational framework (ROLAP). In particular, they can be implemented by using procedure language in any RDBMS such as Oracle PL/SQL.

Assume the sales information of an advertiser is stored in a relation table SALES(publisher, date, sales). Given an aggregation ranking query $AR(k, D_{start}, D_{end})$, it can be executed by using a Top-*N* query which is supported in some commercial databases such as Oracle or DB2. The following SQL statement shows such an example using Oracle9i SQL dialect.

```
SELECT * FROM (
 SELECT publisher, SUM(sales) total_sales
 FROM SALES
 WHERE date >= start_date AND date <= end_date
 GROUP BY publisher
 ORDER BY total_sales
) WHERE ROWNUM < k;
```

The above SQL query scans all the tuples between time D_{start} and D_{end}. Then it groups the publishers by aggregating total sales. In the end, the total sales of all publishers need to be sorted in order to report the top-*k* publishers. Note that the most recent versions of Oracle such as Oracle9i and Oracle10g do support RANK analytical functions such as RANK and DENSE_RANK. However no information is available as to how these ranking features are implemented. As future work, we plan to compare our techniques with Oracle RANK functions implementation.

Now we discuss how to use our techniques to avoid the overhead of scanning all the involved tuples. In order to achieve this, we need to maintain a materialized view from the original SALES table, which stores the cumulative information. The materialized view can be created by the following SQL statement:

```
CREATE MATERIALIZED VIEW LOG ON sales
WITH PRIMARY KEY, ROWID(sales)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW mv_prefix_sum_sales
REFRESH FAST ON COMMIT
AS
SELECT s1.publisher, s1.date,
       SUM(s2.sales) prefix_sum_sales
FROM SALES s1, SALES s2
WHERE s1.publisher = s2.publisher
      AND s1.date >= s2.date
GROUP BY s1.publisher
ORDER BY s1.date, prefix_sum_sales;
```

Then Oracle9i PL/SQL is used to implement our proposed techniques over the created materialized view which incorporates the cumulative and ranking information as well. More specifically, for the given query $AR(k, D_{start}, D_{end})$, two cursors C_{start} and C_{end} are declared to simulate the traversing, which corresponds to pointers $POINTER_{start}$ and $POINTER_{end}$ as discussed in the MOLAP context. For example, the Oracle9i PL/SQL procedure for the bi-directional traversal algorithm is listed as follows.

```
/*
** ########################################################################
**
**       BI-DIRECTIONAL TRAVERSAL ALGORITHM IN Oracle 9.2 PL/SQL
**
** ########################################################################
**
** CODING STYLE USED
**
** 1. Uppercase is used for PL/SQL keywords and lower case for defined items.
**
** 2. Naming convention followed for user defined types :
**
**      Data Structure Name                    Prefixed with
**      -------------------                    -------------
**      Varray Type names                      vt_
**      Varray                                 v_
**      Local variables                        l_
**
** NOTE: This procedure does not handle prefix aggregation ranking queries,
**       that is start_date is the oldest date in the table.
*/
CREATE OR REPLACE PROCEDURE bi_directional_traversal
                            ( start_date IN NUMBER,
                              end_date IN NUMBER,
                              publisher_size IN NUMBER,
                              k IN NUMBER )
IS

 /* define vt_number of numbers */
 TYPE vt_number IS VARRAY (10000) OF NUMBER;
 TYPE vt_bool   IS VARRAY (10000) OF BOOLEAN;

 /* define cursor to scan list PPSUM(start_date - 1) */
 CURSOR start_cur IS
   SELECT publisher, prefix_sum_sales
   FROM mv_prefix_sum_sales
   WHERE date = start_date - 1
   ORDER BY prefix_sum_sales ASC;

  /* define cursor to scan list PPSUM(end_date) */
  CURSOR end_cur IS
    SELECT publisher, prefix_sum_sales
    FROM mv_prefix_sum_sales
    WHERE date = end_date
    ORDER BY prefix_sum_sales DESC;

  /* variables to store scanned publishers currently */
  l_publisher_start          NUMBER;
  l_prefix_sum_sales_start   NUMBER;
  l_publisher_end            NUMBER;
  l_prefix_sum_sales_end     NUMBER;

  /* vt_numbers to store scanned publishers */
  v_publisher_start          vt_bool   := vt_bool();
  v_prefix_sum_sales_start   vt_number := vt_number();
  v_publisher_end            vt_bool   := vt_bool();
  v_prefix_sum_sales_end     vt_number := vt_number();

  /* intersected publishers scanned from both lists */
  l_intersection             NUMBER;

  /* result sets */
  v_publisher_result         vt_number := vt_number();
  v_prefix_sum_sales_result  vt_number := vt_number();
  l_result_size              NUMBER;

  l_prefix_sum_sales_temp    NUMBER;

BEGIN

  /* sort access */
  OPEN start_cur;
  OPEN end_cur;

  l_intersection := 0;

  /* extend vt_numbers used */
  v_publisher_start.EXTEND(publisher_size);
  v_prefix_sum_sales_start.EXTEND(publisher_size);
  v_publisher_end.EXTEND(publisher_size);
  v_prefix_sum_sales_end.EXTEND(publisher_size);

  FETCH start_cur INTO l_publisher_start, l_prefix_sum_sales_start;
  FETCH end_cur INTO l_publisher_end, l_prefix_sum_sales_end;

  LOOP
    /* store the scanned publishers including prefix sum sales */
    v_publisher_start(l_publisher_start) := TRUE;
    v_prefix_sum_sales_start(l_publisher_start) :=
                          l_prefix_sum_sales_start;
```

```
      v_publisher_end(l_publisher_end) := TRUE;
      v_prefix_sum_sales_end(l_publisher_end) :=
                            l_prefix_sum_sales_end;

      /*
       *check the intersection of the
       *scanned publishers from both lists
       */
      IF l_publisher_start = l_publisher_end THEN
        l_intersection := l_intersection + 1;
      ELSIF v_publisher_start(l_publisher_end) THEN
        l_intersection := l_intersection + 1;
      ELSIF v_publisher_end(l_publisher_start) THEN
        l_intersection := l_intersection + 1;
      END IF;

      IF l_intersection >= k THEN
        EXIT;
      END IF;

      EXIT WHEN start_cur%NOTFOUND;

      FETCH start_cur INTO l_publisher_start, l_prefix_sum_sales_start;
      FETCH end_cur INTO l_publisher_end, l_prefix_sum_sales_end;

    END LOOP;

  v_publisher_result.EXTEND(k);
  v_prefix_sum_sales_result.EXTEND(k);
  l_result_size := 0;

  FOR i IN 1..publisher_size
    LOOP
      l_result_size := l_result_size + 1;

      IF v_publisher_start(i) AND v_publisher_end(i) THEN

        l_prefix_sum_sales_result :=
            v_prefix_sum_sales_end(i) - v_prefix_sum_sales_start(i);

      ELSIF v_publisher_start(i) THEN

        SELECT prefix_sum_sales INTO l_prefix_sum_sales_temp
        FROM prefix_sum_sales
        WHERE date = end_date AND publisher = i;

        l_prefix_sum_sales_result :=
             l_prefix_sum_sales_temp - v_prefix_sum_sales_start(i);

      ELSIF v_publisher_end(i) THEN

        SELECT prefix_sum_sales INTO l_prefix_sum_sales_temp
        FROM prefix_sum_sales_5
        WHERE date = start_date AND publisher = i;

        l_prefix_sum_sales_result :=
             v_prefix_sum_sales_end(i) - l_prefix_sum_sales_temp;

      END IF;
        v_publisher_result(l_result_size) := i;
        v_prefix_sum_sales_result(l_result_size) :=
                                      l_prefix_sum_sales_result;
    END LOOP;

   DELETE FROM temp_result;

   CLOSE start_cur;
   CLOSE end_cur;

END bi_directional_traversal;
```