Exploiting the Multi-Append-Only-Trend Property of Historical Data in Data Warehouses*

Hua-Gang Li† Divyakant Agrawal† Amr El Abbadi† Mirek Riedewald‡

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

‡Cornell University
Ithaca, NY 14853, USA
mirek@cs.cornell.edu

Abstract

Data warehouses maintain historical information to enable the discovery of trends and developments over time. Hence data items usually contain time-related attributes like the time of a sales transaction or the order and shipping date of a product. Furthermore the values of these time-related attributes have a tendency to increase over time. We refer to this as the Multi-Append-Only-Trend (MAOT) property. In this paper we formalize the notion of MAOT and show how taking advantage of this property can improve query performance considerably. We focus on range aggregate queries which are essential for summarizing large data sets. Compared to MOLAP data cubes the amount of pre-computation and hence additional storage in the proposed technique is dramatically reduced.

1 Introduction

The notion of *time* has received considerable attention in data warehouses during the past few years due to the accumulation of large amounts of time evolving data. Data items often contain time-related attributes such as the time of a sales transaction or the order and shipping date of a product. This enables analysts to extract interesting information over a certain period of time, e.g., observing a sales trend or analyzing revenue and expense trends over time. The importance of time even resulted in a separate branch of research specifically concerned with *temporal databases* [11].

A main characteristic of data collections is that the data grows very often rapidly over time. To make these massive data collections digestible for human analysts, support for efficient aggregation and summarization is vital. An important tool for analyzing data is the orthogonal range aggregate query, for instance, "*what is the total value of all orders in California which were ordered in the first half of July 2002 and shipped in August or later*?". This query selects ranges on some of the attributes and computes aggregates over the selected data items. Besides, roll-up and drill-down queries [6] that aggregate on different levels of granularity are often collections of related range queries. In this paper, we concentrate on the most prevalent aggregation operation `SUM` in OLAP applications. Note that the aggregation operator `COUNT` is a special case of `SUM`, and `AVG` can be obtained through a tuple (*sum*,*count*).

In [21], Riedewald et al. developed a new framework supporting efficient aggregation over *append-only* data sets. An

*This research was supported by the NSF under IIS98-17432, EIA99-86057, EIA00-80134, and IIS02-09112.

append-only data set refers to a collection of data items with one of the describing attributes being a *transaction time dimension* (TT-dimension). Typically there is a correlation between the value of the transaction time attribute and when the data item is incorporated into the data collection. For instance, sales transactions and phone calls are recorded in a timely manner and hence the earlier a sales event or phone call takes place, the earlier it will be recorded in the data warehouses. Intuitively, a d-dimensional data set is append-only, if updates can only affect data items with the latest or a greater transaction time coordinate. Hence, updates are restricted to the $(d-1)$-dimensional "time slice" with the greatest transaction time coordinate.

In practice we observe that there are many data sets with multiple time-related attributes whose values increase over time. For example, the retail data set from an online shop has an order date, a shipping date, and may be even a delivery date to describe data items. Hence the data space grows along multiple dimensions which introduces a high degree of sparseness. For that reason data cubes like *prefix sum cube* and its variations [15, 5, 13, 20] which are based on array structures will not be space efficient for supporting aggregation. A "diagonal line" of data points (when looking only at the time-related dimensions with growing values) is also difficult to index with popular bounding-shape based trees like R-tree [14] and SS-tree [22]. On the other hand if there is truly a diagonal line, i.e., the data is *Multi-Append-Only* (MAO) in time-related dimensions, we can just reduce the multiple time-related dimensions to one-dimensional aggregation.

In real applications one will rarely find multiple dimensions such that newly inserted data items have increasing values in all these dimensions. For instance, once an order for a product is placed, the retailer processes it and ships the product to the customer within a certain amount of time. Typically given a later order date, there will be a later shipping date. However, varying order processing speed will cause exceptions from the rule. On the other hand such exceptions are not arbitrarily bad. In the order example most outliers will be off by at most 1 or 2 days. Hence such data sets with multiple time-related dimensions may exhibit a non-decreasing trend in some time-related dimensions while in others may maintain this trend *approximately*. That is to say, some data points in the data sets are slightly off, but still within a certain bound. We refer to this as the *Multi-Append-Only-Trend* (MAOT) property and data sets with such property are the focus of this paper. Intuitively in a MAOT data set the data points are within a narrow diagonal band (when considering only the time-related dimensions), hence we want to be able to deal with aggregation in MAOT data sets as efficiently as in MAO data sets. This idea is at the heart of our novel aggregation technique.

The rest of the paper is organized as follows. Section 2 gives an overview of related work. In Section 3, motivation of our research work is stated. In Section 4, we discuss and empirically evaluate our proposed technique for two-dimensional MAOT data sets. In Section 5 we generalize the proposed technique to d-dimensional data sets with two MAOT dimensions only. Conclusions and future research work are given in Section 6.

2 Related Work

Some of the most efficient OLAP aggregation techniques maintain a d-dimensional data set $\mathcal{D}$ in a d-dimensional array-like data cube structure. For example, Figure 1 shows a two-dimensional data set with its corresponding data cube. Ho et al. [15] introduced an elegant technique for computing range aggregate queries on general data sets, which is referred to as the *Prefix Sum* technique (PS). The basic idea is to compute the prefix sums of the original data cube (see Figure 1). In particular, each cell indexed by (x_1, x_2) in the prefix sum cube P maintains the sum of all cells (c_1, c_2) of the original data cube A that satisfy $0 \leq c_1 \leq x_1$ and $0 \leq c_2 \leq x_2$. The PS technique ensures constant query time, more precisely at most

2^d cell accesses per query. In Figure 1, the range sum query shown by the shaded area can be computed by accessing cells (3,2), (0,2), (3,0) and (0,0) in the prefix sum data cube P, i.e. $P[3,2]-P[0,2]-P[3,0]+P[0,0] = 40-11-12+3 = 20$. The storage cost for pre-aggregated information is $O(n^d)$ assuming each dimension size of n without loss of generality. Other array-based techniques [5, 13, 20] provide a variety of different tradeoffs between query and update costs. Similar to PS their storage requirement is in the order of $O(n^d)$. Unfortunately array-based techniques are not feasible for very sparse and high-dimensional data sets. For instance constructing the PS cube for a sparse data set will result in a high degree of redundancy (see also Section 3).

d1	d2	measure value
0	0	3
0	1	7
0	2	2
0	3	3
...	...	...
4	2	3
4	3	3

Data set D

d2					
3	3	2	1	5	3
2	2	4	2	3	3
1	7	3	2	6	8
0	3	5	1	2	2
	0	1	2	3	4
					d1

original data cube A

3	15	29	35	51	67
2	12	24	29	40	53
1	10	18	21	29	39
0	3	8	9	11	13
	0	1	2	3	4

prefix sum cube P

Figure 1: The original array and prefix sum array

A number of highly sophisticated aggregation techniques for sparse data have been proposed for computational geometry applications [8, 7, 23]. However, typically the storage overhead is super-linear, e.g., $O(N \log^{d-1} N)$ for a data set of size N, which is infeasible for large multidimensional data sets in data warehousing applications. Also, since the data structures are fairly involved, they are rarely used in practice.

Another approach for aggregating over sparse data is to take advantage of existing index structures. A broad survey on various index structures is given in [12]. More recent techniques can be found in [2, 4, 10, 17, 18]. Indexing can provide fast access to all selected data items. However, to retrieve and aggregate each selected item on-the-fly is still too slow. This is addressed by augmenting index structures with pre-computed aggregates as proposed in [19, 16]. None of these techniques can take advantage of the semantic knowledge provided by the time-related attributes.

The only previous aggregation framework that explicitly takes advantage of properties of time-related attributes was introduced by Riedewald et al. [21]. Their approach reduced the complexity of the aggregation problem by making query and update costs independent of the time dimension. However, taking advantage of multiple time-related dimensions was not supported.

3 Technique for Multi-Append-Only Data

Assume that all dimensions are time-related and that inserted data items have the MAO property, i.e., in each dimension the coordinate value of a newly inserted data point is at least as high as for all previously inserted points. In the following we describe a simple yet efficient aggregation approach for this setup.

As discussed in Section 1, the data space of an MAO data set grows along multiple time-related dimensions and hence high sparseness will be introduced due to the infinite domain of time-related dimensions. To apply the PS technique and its variations directly on an MAO data set will cause high storage overhead as illustrated in Example 1.

Example 1 Figure 2 shows the data cube of a two-dimensional MAO data set. The original data cube has a high degree of sparseness, therefore the corresponding PS cube contains a high amount of redundant information. Applying any of the other array-based aggregation techniques would have a similar effect.

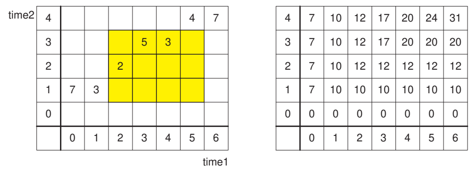


Figure 2: An MAO data set and its prefix sum cube

As the data set in Figure 2 has the MAO property, we can conceptually map all MAO dimensions to a single dimension and then apply a technique similar to [21] as follows. Let R denote a data structure such that for each pair of time coordinates (t_1, t_2), a *R-instance* $R(t_1, t_2)$ maintains the cumulative information of all data points whose `time1` attribute value is less than or equal to t_1 and `time2` attribute value is less than or equal to t_2. Figure 3 shows the R-instances constructed for the two-dimensional MAO data set in Figure 2. For instance, $R(2, 2)$ maintains the cumulative knowledge of data points $(2, 2)$, $(1, 1)$, $(0, 1)$.

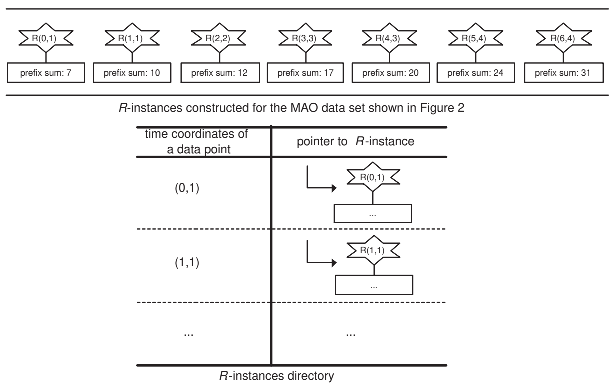


Figure 3: R-instances and R-instance directory

This cumulative knowledge enables us to reduce a two-dimensional range query to finding two R-instances as follows. For example, a range aggregate query is specified as $2 \leq$ `time1` ≤ 5 and $1 \leq$ `time2` ≤ 3 (shaded area in Figure 2). We first get the R-instance $R(4, 3)$ with the greatest time values, in both dimensions, which are less than or equal to the upper values of the selected time ranges. $R(4, 3)$ maintains the sum of all data points whose `time1` coordinate is less than or equal to 4 and `time2` coordinate is less than or equal to 3, which is 20. To answer the actual range query, we have to remove all the data points whose `time1` coordinate is less than 2 and `time2` coordinate is less than 1. Hence, we refer to the R-instance $R(1, 1)$ to get the sum of those data points, which is 10. Then we subtract the result from the initial value, obtaining the correct result of 10.

So far we have not described how to find the appropriate R-instances needed to answer range aggregate queries. In order to do that, we need a directory that maintains the correspondence between time coordinates and instances of R, called *R-instance directory*. For example, Figure 3 also shows the R-instance directory maintained for the given two-dimensional MAO data set. For any given range aggregate query, we have to perform two lookups of the R-instance directory, one for the lower and one for the upper end of the range query. The time coordinates of data points are ordered in non-decreasing order, therefore we can perform a `binary-search-like' lookup. The actual procedure is shown in Figure 4. We can observe that the cost of a lookup is at most logarithmic in the number of data points. Thus the total query cost is $O(\log N)$ where N is the total number of data points in the data set. Also it is obvious that the storage cost for additional information is $O(N)$. When compared with the existing techniques, this technique achieves significant reduction in storage cost while maintaining efficient query processing time. Note that our approach easily generalizes to higher dimensionality and to data sets with both time-related and "general" dimensions. In the latter case our technique reduces the complexity for queries and updates to that of a data set which is the projection of the original set to the lower-dimensional data space defined by the general dimensions only.

```
R-instance Directory Lookup

Function RDirLookUp;
Input: R-instance directory T, start, end, P^2
// start: starting index of T
// end: ending index of T
// P^2 is the upper/lower-bound query point

(1) if start == end  satisfied = true; // base case I
    for i in 1..2
     if T[start].Coordinates[i] > P^2[i]
         satisfied = false; break;
    if satisfied return T[start];
    else return NULL;
(2) if start - end == 1   // base case II
       satisfied_s = true;
       for i in 1..2
          if T[start].Coodinates[i] > P^2[i]
            satisfied_s = false; break;
       if satisfied_s return T[start];
       else
        satisfied_e = true;
        for i in 1..2
           if T[end].Coordinates[i] > P^2[i]
             satisfied_e = false; break;
        if satisfied_e return T[end];
        else return NULL;
(3) mid = start + (end - start)/2; equal = true; right = true;
(4) for i in 1..2
       if T[mid].Coordinates[i] != P^2[i] equal = false;
       if T[mid].Coordinates[i] > P^2[i] right = false;
(5) if equral return T[mid];
    else if right return RDirLookUp(T, mid + 1, end, P^2);
         else return RDirLookUp(T, start, mid, P^2);
```

Figure 4: R-instance directory lookup

However, the MAO property imposes a very strict condition on time-related attributes of historical data sets. In real applications, data sets with multiple time-related attributes very often only exhibit the weaker MAOT property. In this paper, we propose a space-efficient technique to handle range aggregate queries in MAOT data sets and our goal is to treat the data sets as if there were no outliers and to correct for possible mistakes by paying a small cost, which depends on the deviation of the outlier data point.

4 A Technique for two-dimensional MAOT Data Sets

In this section, we develop a new technique to answer range aggregate queries on two-dimensional MAOT data sets efficiently while maintaining space efficiency. Also this section introduces important notation that will be used throughout the paper.

4.1 Notation

Let $\mathcal{D}$ denote a **data set** with d dimensional attributes $\delta_1, \ldots, \delta_d$, and a single measure attribute m. Let (X^d, v), $X^d = (x_1, \ldots, x_d)$, refer to a **data point** in $\mathcal{D}$ and its measure value is v. A data point exists in the data set, if and only if its measure value is not NULL. A multi-dimensional range aggregate query specifies a range (L_i, U_i) in each dimension δ_i, the selection possibly being a single value or the entire domain. The query selects all data points X^d that satisfy $L_i \leq x_i \leq U_i$ for all dimensions δ_i and applies aggregate operator (e.g. `SUM`) over these cells. $L^d = (L_1, \ldots, L_d)$ and $U^d = (U_1, \ldots, U_d)$ are referred to as the **lower-bound query point** and the **upper-bound query point** of a multi-dimensional range aggregate query.

The d-dimensional data set $\mathcal{D}$ is a data set with **Multi-Append-Only-Trend** (MAOT) property if and only if it satisfies the following conditions:

1. one of its dimensions, say δ_1, is a *transaction time dimension* (TT-dimension).

2. d_t of its dimensions, say $\delta_2, \ldots, \delta_{d_t+1}$, are *valid time dimensions* (VT-dimensions).

3. the TT-coordinate of the latest update u always follows a non-decreasing trend, i.e., if an update to a data point X^d arrives at the data set before another update to a data point Y^d, then $x_1 \leq y_1$.

4. if the i^{th} VT-coordinate of the latest update u does not follow a non-decreasing trend, it should be at most ε time units earlier than the i^{th} VT-coordinate of the previous data point, which has the greatest TT-coordinate and whose i^{th} VT-coordinate follows a non-decreasing trend. We refer to this preceding point as the *Sentinel Data Point* (SDP) for i^{th} VT-dimension (ε-bound).

To simplify the following discussion we will further on assume that no two data points have the same coordinate in the TT-dimension δ_1. The generalization is straightforward.

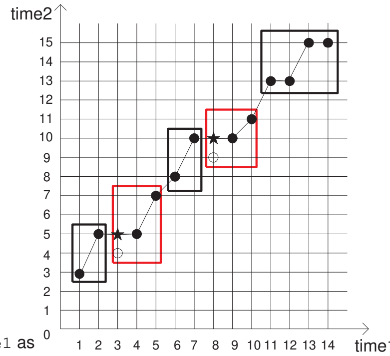
4.2 Illustration of the Technique

For simplicity, the technique is illustrated for a two-dimensional MAOT data set $\mathcal{D}$ with $\varepsilon = 1$ and the operator `SUM`, which is shown in Figure 5. Let the dimensions be `time1`, `time2`. New points arrive in the strict order of `time1`, i.e., `time1` dimension is a TT-dimension. The `time2` dimension is a VT-dimension, i.e., if the VT-coordinate of the latest update u does not follow a non-decreasing trend, it should be at most $\varepsilon = 1$ time unit earlier than the corresponding VT-coordinate of its SDP for `time2` dimension.

4.2.1 Adjustment of Outlier Data Points

Like the technique proposed for MAO data sets, we can capture the information of the MAOT data set $\mathcal{D}$ in a collection of instances of a data structure I (defined in Section 4.2.3) such that there is an I-*instance* for each pair of time coordinates (t_1, t_2) for which a data point exists. The information in these data structures is cumulative. The TT-dimension property ensures that an update u in $\mathcal{D}$ either affects the existing I-instance with the greatest TT-coordinate or appends a new I-instance with a greater TT-coordinate. Furthermore, if the VT-coordinate of the newly appended I-instance does not follow a non-decreasing trend (*an outlier data point*), we adjust it by increasing its value by at most ε time units to ensure its new value is at least as large as that of its SDP for `time2` dimension.

time1	time2	measure value
1	3	2
2	5	3
3	**4**	**2**
4	5	1
5	7	6
6	8	4
7	10	8
8	**9**	**5**
9	10	1
10	11	3
11	13	4
12	13	5
13	15	2
14	15	2
...	...	...



A two-dimensional MAOT data set D with `time1` as TT-dimension, `time2` as VT-dimension, and `epsilon` = 1

Figure 5: A two-dimensional MAOT data set (`time1` as TT-dimension, `time2` as VT-dimension, $\varepsilon = 1$)

After adjustment, the data set $\mathcal{D}$ appears as though there were no outlier data points. Thus the technique proposed for MAO data sets can be applied to the MAOT data set. However, to correct for possible mistakes, we need to store additional information. For each pair of time coordinates (t_1, t_2) (t_2 may be an adjusted VT-coordinate), the corresponding I-instance $I(t_1, t_2)$ maintains not only cumulative knowledge of all the data points $X^2 = (x_1, x_2)$ where $x_1 \le t_1, x_2 \le t_2$, but also some other auxiliary information which is not required for MAO data sets.

Example 2 In Figure 5, we can observe that there are two outlier data points in the data set $\mathcal{D}$. The VT-coordinate of data point (3,4) does not follow a non-decreasing trend, thus we adjust it to ensure its VT-coordinate is at least as large as that of its SDP for `time2` dimension (2,5), i.e., 5 (as shown from circle to star in the figure). Similarly, we increase the VT-coordinate of data point (8,9) to that of its SDP for `time2` dimension (7,10), i.e., (8,9) $\rightarrow$ (8,10).

4.2.2 Classification of Data Points

Recall that the R-instance constructed for each pair of time coordinates in a two-dimensional MAO data set maintains the same kind of cumulative information. However, we cannot deal with two-dimensional MAOT data sets in the same way as we need additional information in I-instances with adjusted VT-coordinates to correct for the possible errors introduced by VT-coordinate adjustment. Thus it is necessary to classify data points into groups according to the characteristics of their original VT-coordinates and each group is either of the following two kinds of groups:

1. *Normal group*: a group in which the VT-coordinates of all data points always follow a non-decreasing trend;

2. *Outlier group*: a group starting at a data point s whose original VT-coordinate does not follow a non-decreasing trend and ending at a data point e whose VT-coordinate is later than that of s by at least ε time units; in order to keep the outlier group as small as possible, e is selected such that e has a VT-coordinate at most ε time units greater than that of s and the least TT-coordinate.

Intuitively an outlier group is the smallest group that begins with an outlier and "advances the clock" in the VT-dimension by at least ε.

Example 3 In the example of Figure 5 we can divide all the data points into groups as shown by rectangles. Among these groups, normal groups are {(1,3),(2,5)}, {(6,8),(7,10)} and {(11,13),(12,13),(13,15),(14,15)} and outlier groups are

$\{(3,5),(4,5),(5,7)\}$ and $\{(8,10),(9,10),(10,11)\}$. Note that the first outlier group starts at the data point (3,5) whose original VT-coordinate (4) does not follow a non-decreasing trend and ends at the data point (5,7) whose VT-coordinate (7) is later than that of (3,5) by 2 time units, which is greater than $\varepsilon = 1$ time unit(s).

From the definition and the example above, one may notice that there are some data points in the outlier group which are not outlier data points. We now briefly motivate why we extend outlier groups to an ending data point whose VT-coordinate is later than that of the starting data point by at least ε time units. The ε difference is chosen to ensure that for each range query there is at most one outlier group which is intersected by the query and which could contain data points whose VT-coordinates are less than that of the lower-bound query point. This leads to the efficient query algorithm presented later on which only has to deal with different cases for a single outlier group.

4.2.3 Cumulative and Auxiliary Information

In this subsection, we define the data structure I such that for each pair of time coordinates, $I(t_1, t_2)$ maintains cumulative and auxiliary information which exploits the MAOT property of data set $\mathcal{D}$. All I-instances maintain the following common information:

1. Similar to the technique for MAO data sets, in each I-instance $I(t_1, t_2)$ we maintain the cumulative values of data points $X^2 = (x_1, x_2)$ in data set $\mathcal{D}$ whose time coordinates satisfy $x_1 \leq t_1$ and $x_2 \leq t_2$, denoted as `PSum`. `PSum` can only answer range queries that select half-open time ranges, i.e., "all data points in $\mathcal{D}$ whose TT-coordinate is less than or equal to t_1 and whose VT-coordinate is less than or equal to t_2".
2. Original measure value of the data point (t_1, t_2) denoted as `OVal`.
3. The group type $(\texttt{normal}, \texttt{outlier})$ to which the data point (t_1, t_2) belongs, denoted as `GRP`.

In addition, for an I-instance constructed for a data point (t_1, t_2) of an outlier group, we need to maintain extra information as follows which is used to correct for the possible errors introduced by the VT-coordinate adjustments of outlier data points.

1. A one-dimensional array `EPSum` of size ε. The cell $\texttt{EPSum}[i] (0 \leq i \leq \varepsilon - 1)$ maintains the sum of measure values of data points $X^2 = (x_1, x_2)$ where $x_1 \leq t_1, x_2 \leq t_2$, but excludes those data points in the same outlier group with their original VT-coordinates earlier than $s - i$ where s is the VT-coordinate of the starting data point of its group;
2. A pointer to the I-instance for the starting data point of the outlier group, denoted as `SIns`;
3. A pointer to the I-instance for the ending data point of the outlier group, denoted as `EIns`; if there is no ending data point yet for the group, it is set to NULL;
4. A pointer to the I-instance for the data point arriving immediately after data point (t_1, t_2), denoted `IIns`, if data point (t_1, t_2) is the ending data point of its outlier group;
5. A boolean variable indicating whether the VT-coordinate of a data point (t_1, t_2) is adjusted or not, denoted as `ADJUSTED`;
6. Original time coordinates, if data point (t_1, t_2) is an adjusted one, denoted as `ORC`. Otherwise it is set to NULL;

Example 4 Referring to Figure 5 again, we can construct the I-instances of data set $\mathcal{D}$ as shown in Figure 6. Note that the I-instance $I(3,5)$ belongs to an outlier group, so its group information is `outlier'; PSum stores the sum of measure values of data points whose TT-coordinate is less than or equal to 3 and VT-coordinate is less than or equal to 5, i.e., $2+3+2=7$; OVal stores the original measure value of data point $(3,5)$. EPSum$[0]$ stores the sum of measure values of data points whose TT-coordinate is less than or equal to 3 and VT-coordinate is less than or equal to 5, but excludes those data points which are in the same group and whose original VT-coordinate is less than $5-0=5$, i.e., EPSum$[0]=2+3=5$.

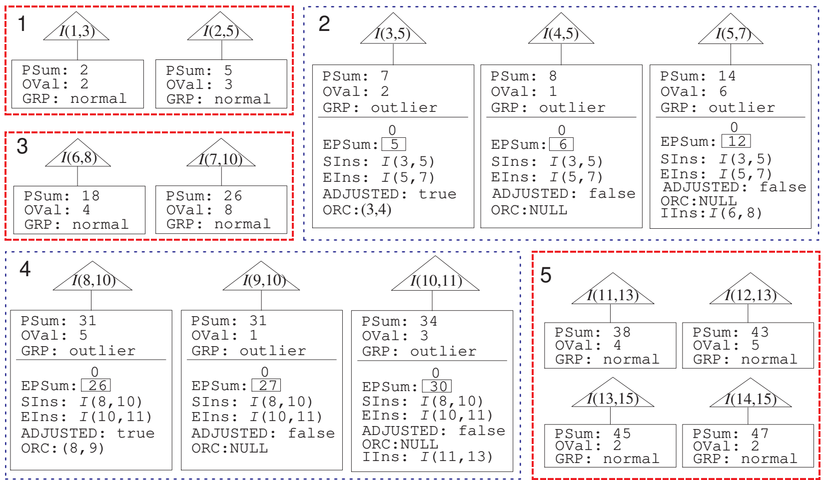


Figure 6: I-instances constructed for the two-dimensional MAOT data set $\mathcal{D}$ shown in Figure 5

4.2.4 Answering Range Aggregate Queries

Now we illustrate how to use the cumulative and auxiliary information to speed up range aggregate (SUM) queries. Consider a range query represented as $\texttt{Query}(L^2=(3,5),U^2=(8,9))$, which corresponds to the lighter gray area in Figure 7. We observe that the outlier data point (8,9) is actually inside the given query range, but due to the adjustment of its VT-coordinate, it appears to be outside. In order to guarantee that no point is missed, we have to expand the query range by increasing the VT-coordinate of the upper-bound query point by $\varepsilon=1$. Then we obtain $\texttt{Query}_{\texttt{expand}}(L^2=(3,5),U^2=(8,10))$, as shown by the lighter and darker grey areas together. A natural question is why an analogous correction is not necessary at the bottom (for data points that move up into the query region). The reason is that we can use the EPSum array for correction instead of examining the ε stripe at the query bottom.

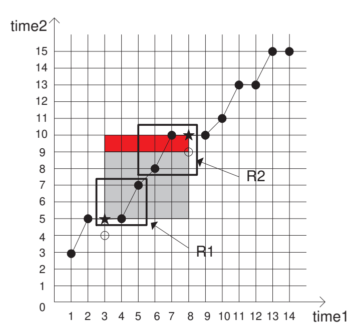


Figure 7: Ranges selected on time1 and time2 dimensions

After query range expansion, some data points which are not inside the given query range may now appear in the expanded query range. For example, data point (7,10) is not inside the given range query, but now appears in the expanded query range, which is shown as a darker gray region in Figure 7. Such points must be excluded from the final answer. Hence, the original range query is transformed into two sub-queries: $\texttt{subQuery}_{\texttt{expand}}(L^2 = (3,5), U^2 = (8,10))$ and $\texttt{subQuery}_{\texttt{surplus}}(L^2 = (3,9), U^2 = (8,10))$, i.e., the query result is equal to $\texttt{subQuery}_{\texttt{expand}} - \texttt{subQuery}_{\texttt{surplus}}$.

Therefore, given any range aggregate query, we expand the original query such that it is guaranteed that all selected points are included. Since data points can only move "up", we do not need to expand the bottom part of the query. After expanding, all we need to do is filter out the effect of false hits by using the pre-computed information.

Both sub-queries can be processed similarly. Here we only use $\texttt{subQuery}_{\texttt{expand}}$ as an example to show how to process the sub-query. We start by defining two I-instances LB_{ins}, UB_{ins} where LB_{ins} is the I-instance with the least time coordinates which is greater than or equal to the lower values of the selected time ranges and UB_{ins} is the I-instance with the greatest time coordinates which is less than or equal to the upper values of the selected time ranges. For our example, we get $LB_{ins} = I(3,5)$ and $UB_{ins} = I(8,10)$.

As LB_{ins} is an I-instance for a data point of an outlier group ($R1$ in Figure 7), we can locate the I-instance for the ending data point of the outlier group through $LB_{ins}.\texttt{EIns}$, which is $I(5,7)$. Recall that the VT-coordinate of the ending data point of the outlier group is greater than that of the starting data point of the outlier group by at least ε time units. Thus the VT-coordinate of all data points after $I(5,7)$ can not fall below the query range. Thus we can simply use the cumulative information stored in $I(8,10)$ and $I(5,7)$ to compute the sum of data points remaining in the query($R2$ in Figure 7), i.e., $I(8,10).\texttt{PSum} - I(5,7).\texttt{PSum} = 31 - 14 = 17$.

$R1$ only contains data points of an outlier group and not all of them are inside the query range due to the VT-coordinate adjustment. To exclude those data points whose original VT-coordinates are not within the selected range on `time2` dimension, we can use the `EPSum` maintained in the I-instances $I(3,5)$ and $I(5,7)$ to compute the sum of data points in $R1$. The lower bound of the selected range in `time2` dimension is 5, so we have to exclude those data points in the outlier group whose original VT-coordinate is earlier than 5. From the definition of `EPSum`, we know that $\texttt{EPSum}[0]$ will exclude those data points, i.e., sum of data points in $R1$ can be computed through $I(5,7).\texttt{EPSum}[0] - I(3,5).\texttt{EPSum}[0]$, which is $12 - 5 = 7$. Thereby we can get the result for $\texttt{Query}_{\texttt{expand}}$, which is $17 + 7 = 24$. Likewise, we can get the result for $\texttt{Query}_{\texttt{surplus}}$, which is 8. Thus the query result for $\texttt{Query}(L^2 = (3,5), U^2 = (8,9))$ is $24 - 8 = 16$.

4.3 Algorithm and Time Complexity Analysis

We now present a formal description of how sub-queries such as $\texttt{subQuery}_{\texttt{expand}}$ or $\texttt{subQuery}_{\texttt{surplus}}$ are processed. Recall that we subtract $\texttt{subQuery}_{\texttt{surplus}}$ from $\texttt{subQuery}_{\texttt{expand}}$ to get the final answer to any given range aggregate query, hence both sub-queries by themselves may include false hits. Also note that data points in the following refer to data points after VT-coordinate adjustment unless otherwise specified:

1. If both LB_{ins} and UB_{ins} are NULL, the sub-query range does not contain any data point and consequently we return 0 as the sub-query result.

2. If LB_{ins} is equal to UB_{ins} (an example sub-query shown in Figure 8(a)), the sub-query range contains only one I-instance. If it is not an adjusted data point (by checking `ADJUSTED`) or the original time coordinates of its data point

(by checking ORC) are inside the range, we return LB_{ins}.OVal as the sub-query result; otherwise we return 0 as the sub-query result.

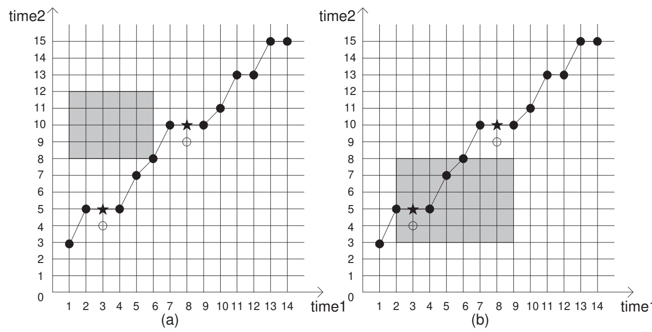


Figure 8: Processing a sub-query: case 2 and case 3

3. If LB_{ins} and UB_{ins} are two different I-instances and the VT-coordinate of LB_{ins} is greater than that of the lower-bound query point L^2 by at least ε time units (an example sub-query shown in Figure 8(b)), every data point between LB_{ins} and UB_{ins} must be inside the sub-query range. PSum maintained in UB_{ins} gives the sum of all data points whose time coordinates are less than or equal to those of UB_{ins}. To answer the actual sub-query, we have to remove all data points whose time coordinates are less than those of LB_{ins}. The sum of such data points can be obtained by subtracting LB_{ins}.OVal from LB_{ins}.PSum. Hence, the correct answer is $UB_{ins}.\texttt{PSum}-(LB_{ins}.\texttt{PSum}-LB_{ins}.\texttt{OVal})$.

4. If LB_{ins} and UB_{ins} are two different I-instances, and the difference between the VT-coordinate of LB_{ins} and that of L^2 is less than ε time units, then there might be data points between LB_{ins} and UB_{ins} which are not inside the sub-query range because of the VT-coordinate adjustments. In this case, the sub-query will be processed differently according to the characteristics of LB_{ins}, i.e., depending on LB_{ins} being the I-instance for a data point of a normal group, the starting data point of an outlier group, or the middle data point (including the ending data point) of an outlier group. The corresponding cases are called `normalQuery`, `outlierSQuery`, `outlierMQuery` respectively.

CASE i `normalQuery` (example sub-queries shown in Figure 9(a)): If LB_{ins} is the I-instance for a data point of a normal group, an I-instance O_{ins} will be obtained such that its data point is the starting data point of the outlier group which is immediately after LB_{ins}'s group. If O_{ins} is NULL or not inside the sub-query range, then the sub-query range only contains data points from a normal group and the processing is the same as described in case 3; otherwise, we divide the data points covered by the sub-query into two parts such that the first part contains data points arriving before the data point of O_{ins} and the second part contains the remaining data points. The second part can be processed by calling `outlierSQuery` because it starts with the data point of O_{ins} which is the starting data point of an outlier group. Thus we return $(O_{ins}.\texttt{PSum} - LB_{ins}.\texttt{PSum} + LB_{ins}.\texttt{OVal} - O_{ins}.\texttt{OVal}) + \texttt{outlierSQuery}(O_{ins}, UB_{ins})$ as the sub-query result.

CASE ii `outlierSQuery` (example sub-queries shown in Figure 9(b)): If LB_{ins} is the I-instance for the starting data point of an outlier group, an I-instance E_{ins} is obtained through LB_{ins}.EIns, whose data point is the ending data point of the outlier group. If E_{ins} is NULL or not inside the sub-query range or E_{ins} is equal to UB_{ins}, then all the data points between LB_{ins} and UB_{ins} are in the same group, i.e., the outlier group.

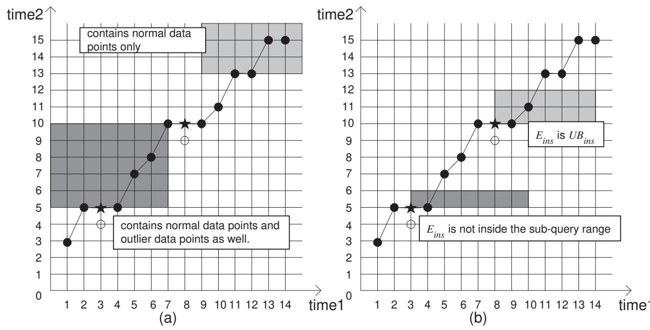


Figure 9: Process a sub-query: case 4.i and case 4.ii

If LB_{ins} aligns with the bottom line of the range, we return UB_{ins}.$\mathtt{EPSum}[0] - LB_{ins}$.$\mathtt{EPSum}[0]$ as the sub-query result; otherwise, we return UB_{ins}.$\mathtt{PSum} - LB_{ins}$.$\mathtt{PSum} + LB_{ins}$.$\mathtt{OVal}$ as the sub-query result. If E_{ins} is inside the range (as the query example in Section 4.2.4), then every data point between E_{ins} and UB_{ins} is inside the sub-query. Similarly, if LB_{ins} aligns with the bottom line of the range, we return $(E_{ins}.\mathtt{EPSum}[0] - LB_{ins}.\mathtt{EPSum}[0]) + (UB_{ins}.\mathtt{PSum} - E_{ins}.\mathtt{PSum})$ as the sub-query result; otherwise, we return $(E_{ins}.\mathtt{PSum} - LB_{ins}.\mathtt{PSum} + LB_{ins}.\mathtt{OVal}) + (UB_{ins}.\mathtt{PSum} - E_{ins}.\mathtt{PSum})$ as the sub-query result.

CASE iii `outlierMQuery`: This covers the remaining possibility of LB_{ins} being the I-instance for a data point in an outlier group which is not the starting point of the outlier group. If LB_{ins} is not the ending data point of the outlier group, the processing is similar to the `outlierSQuery`; otherwise, an I-instance I_{ins} is obtained through LB_{ins}.`IIns`, whose data point arrives immediately after LB_{ins}. We return $E_{ins}.\mathtt{OVal}$ + `normalQuery`(I_{ins}, UB_{ins}) as the sub-query result if I_{ins} is the I-instance for a data point of a normal group; otherwise return $LB_{ins}.\mathtt{OVal}$ + `outlierSQuery`(I_{ins}, UB_{ins})

The range aggregate query algorithm for two-dimensional MAOT data sets with $\varepsilon = 1$ is summarized in Figure 17 and Figure 18, which are in the Appendix. Note that the processing of lookups in the I-instance directory and outlier directory is similar to the algorithm shown in Figure 4.

So far we have not discussed how to find the appropriate I-instances LB_{ins}, UB_{ins} and O_{ins}. We need two directories: One directory, referred to as *I-instance directory*, maintains the correspondence between the time coordinates of data points and their I-instances and is used to search for LB_{ins} and UB_{ins}. The other directory, referred to as *outlier directory*, maintains the correspondence between the time coordinates of the starting data points of outlier groups and their I-instances. It is used to search for O_{ins}.

Example 5 For the two-dimensional MAOT data set shown in Figure 5, we maintain the I-instance directory and outlier directory as shown in Figure 10.

We now analyze the query and storage costs of the proposed approach. For any range query, two sub-queries are executed to compute the result, i.e., $\mathtt{subQuery}_{\mathtt{expand}}$ and $\mathtt{subQuery}_{\mathtt{surplus}}$. For either sub-query, we first need to find the appropriate I-instances for the lower and upper-bound query points of the query range. The cost of a lookup is logarithmic in the number of data points N of a data set $\mathcal{D}$ as it performs a `binary-search-like' search in the $I$-instance directory. Then according to the characteristics of $LB_{ins}$ and $UB_{ins}$, the sub-query is processed in four cases as discussed above. Cases 1, 2 and 3, take constant time. For case 4, a `normalQuery`, requires another lookup in the outlier directory and the

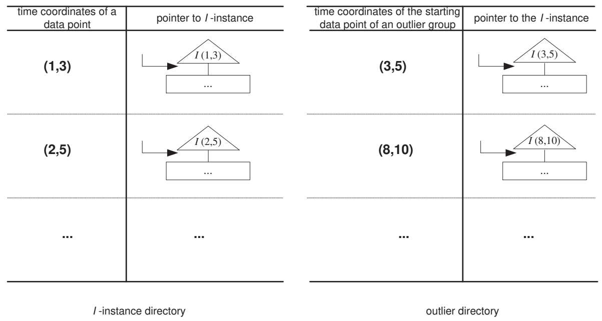


Figure 10: I-instance directory and outlier directory

lookup cost is logarithmic in the number of outlier groups, which is no more than N; if it is an `outlierSQuery`, it takes constant time; an `outlierMQuery`, on the other hand, may be reduced to another `normalQuery` and thus the worst cost is logarithmic in the number of outlier groups. Therefore the query cost for any range aggregate query is $O(\log N)$ in the worst case.

We maintain a historical I-instance for each data point, an I-instance directory and an outlier directory. The storage cost for the cumulative and auxiliary information maintained in a historical I-instance is constant and thus the storage cost for the historical I-instances is $O(N)$. The storage cost of the I-instance directory is also $O(N)$ since we maintain the correspondence between the time coordinates of data points and the pointers to their I-instances. Likewise, the storage cost of the outlier directory is $O(N)$ in the worst case. Consequently the storage cost of additional information is $O(N)$.

Since two-dimensional MAOT data sets with $\varepsilon = 1$ are special cases of two-dimensional MAOT data sets, when dealing with range aggregate queries on two-dimensional MAOT data sets with $\varepsilon > 1$, we need to apply some minor changes to the above-mentioned technique for two-dimension MAOT data sets with $\varepsilon = 1$. The changes and the algorithm are described as follows.

For convenience, we denote a two-dimensional MAOT data set with $\varepsilon > 1$ as $\mathcal{D}^2_{\varepsilon>1}$ and a two-dimensional MAOT data set with $\varepsilon = 1$ as $\mathcal{D}^2_{\varepsilon=1}$. Let I_e denote a data structure which extends I (as defined in Section 4.2.3) in the field of `EPSum` only such that for each pair of time coordinates in $\mathcal{D}^2_{\varepsilon>1}$, there is an I_e-instance of I_e which maintains cumulative and auxiliary information.

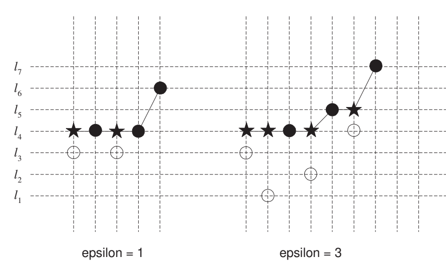


Figure 11: Modify `EPSum` filed for two-dimensional MAOT data sets with $\varepsilon > 1$

From Figure 11, we observe that the characteristics of an outlier group in $\mathcal{D}^2_{\varepsilon>1}$ is different from that of an outlier group in $\mathcal{D}^2_{\varepsilon=1}$. Once a data point whose VT-coordinate is greater than that of the starting data point of the outlier group, the outlier group will be ending for $\mathcal{D}^2_{\varepsilon=1}$ according to the definition of an outlier group. However, that is not the case for $\mathcal{D}^2_{\varepsilon=1}$. Even after some data point whose VT-coordinate is greater than that of the starting data point of the outlier group, there are still possibly some outlier data points and the outlier group will be ending at a data point whose VT-coordinate is later than that of the starting data point of the outlier group by at least ε time units. Thus it is enough for $\mathcal{D}^2_{\varepsilon=1}$ to keep `EPSum` sized of ε to correct for the possible errors introduced due to the VT-coordinate adjustments. However it is not enough for $\mathcal{D}^2_{\varepsilon>1}$. For example, for data point P in Figure 11, if we maintain `EPSum` sized of ε, i.e., the information of outlier data points on lines l_1, l_2 and l_3, then we are missing the information of outlier data points on line l_4.

Therefore, we redefine the `EPSum` of $I(t_1, t_2)$ for a data point (t_2, t_2) (excluding the ending data point) of an outlier group as a one-dimensional array structure with dimension size $\varepsilon + t_2 - s$ (s is the VT-coordinate of the starting data point of the outlier group), in which a cell $\texttt{EPSum}[i](0 \leq i \leq \varepsilon + t_2 - s - 1)$ maintains the sum of measure values of data points $X^2 = (x_1, x_2)$ where $x_1 \leq t_1, x_2 \leq t_2$ but excludes those data points in the same outlier group with original VT-coordinate earlier than $t_2 - i$.

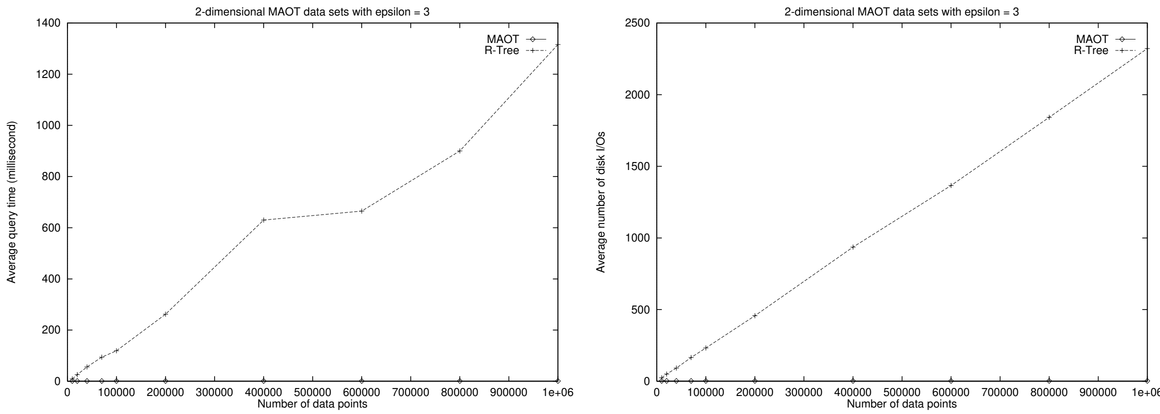
If $I(t_1, t2)$ is the I_e-instance for the ending data point of an outlier group, the `EPSum` maintains a one-dimensional array structure with dimension size $\varepsilon + t_{be} - s$ (s is the VT-coordinate of the starting data point of its group and t_{be} is the VT-coordinate of the data point immediately before the ending data point), in which a cell $\texttt{EPSum}[i](0 \leq i \leq \varepsilon + t_{be} - s - 1)$ maintains the sum of measure values of data points $X^2 = (x_1, x_2)$ where $x_1 \leq t_1, x_2 \leq t_2$ but excludes those data points in the same outlier group with original VT-coordinate earlier than $t_{be} - i$. As we know t_2 of a non-ending data point is at most $s + \varepsilon - 1$ and so is t_{be}, thus we know that the size of `EPSum` of any time instance within an outlier group is at most $2\varepsilon - 1$, which still promises the maintained information finite.

Now we give the range aggregate algorithm for two-dimensional MAOT data sets with $\varepsilon > 1$. As the algorithm is similar to that for two-dimension MAOT data sets with $\varepsilon = 1$, we here only present the changed part of the algorithm in Figure 19 in Appendix.

4.4 Experimental Results

We empirically evaluated the performance of our technique for two-dimensional MAOT data sets and also compared it with R-tree technique. We implemented both techniques in Java and all experiments were performed on a Pentium IV PC with 2.4GHZ CPU and 1GB of main memory which runs Linux RedHat 8.0. We generated synthetic two-dimensional MAOT data sets as data source by varying epsilon (ε) and data set size. We compared our proposed technique with the R-Tree technique in three settings and the metrics we used are average query time and average number of disk I/Os per query. For the purpose of a fair comparison, we cached part of the R-tree in main memory during searching for a query answer and the cache size is the same as that of the I-instance directory (It is maintained in main memory for our proposed technique when answering range queries). Moreover we used Least Recently Used (LRU) cache algorithm.

The first setting is to evaluate the effect of data set size on the performance. We generated MAOT data sets by fixing the epsilon (ε) and varying the data set size from 10^4 to 10^6. We executed about 5000 uniformly generated range queries on every data set to compute the average query time and average number of disk I/Os. The experimental results of this setting are shown in Figure 12. We can observe that changing the data set size does not change the performance of our technique too much while the performance of R-tree technique degrades actually grows linearly with respect to the data set size. As the dominant query time is the hard disk access time and our technique just requires a few number of hard disk accesses, thus even a large data set size will not affect overall query time too much, which comprises hard disk access time and I-instance directory searching time.

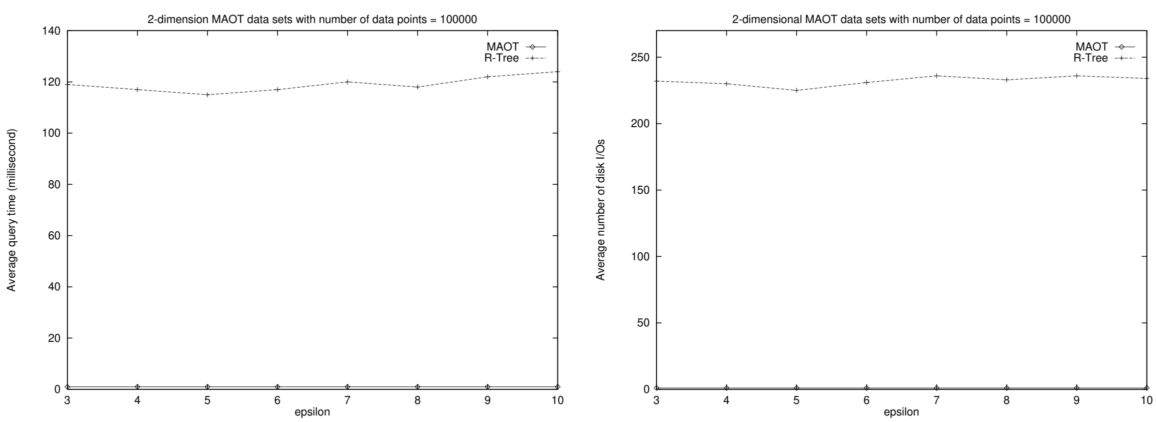


(a) Average query time comparison

(b) Average disk I/Os comparison

Figure 12: Effect of $data\ set\ size$ on performance

The second setting is to evaluate the effect of epsilon (ε) on the performance. We generated MAOT data sets by fixing the data set size (10^5) and varying epsilon (ε) from 3 to 10. Likewise, we executed about 5000 uniformly generated range queries on every data set to compute the average query time and average number of disk I/Os. As both techniques' searching algorithms are independent of the size of epsilon, thus changing epsilon size do not affect the performance of both techniques which coincides the experimental results shown in Figure 13. Besides we can observe that our proposed technique still outperforms the R-tree technique. However a larger epsilon definitely results in more storage space needed for outlier instances and more details are discussed in Section 4.5.

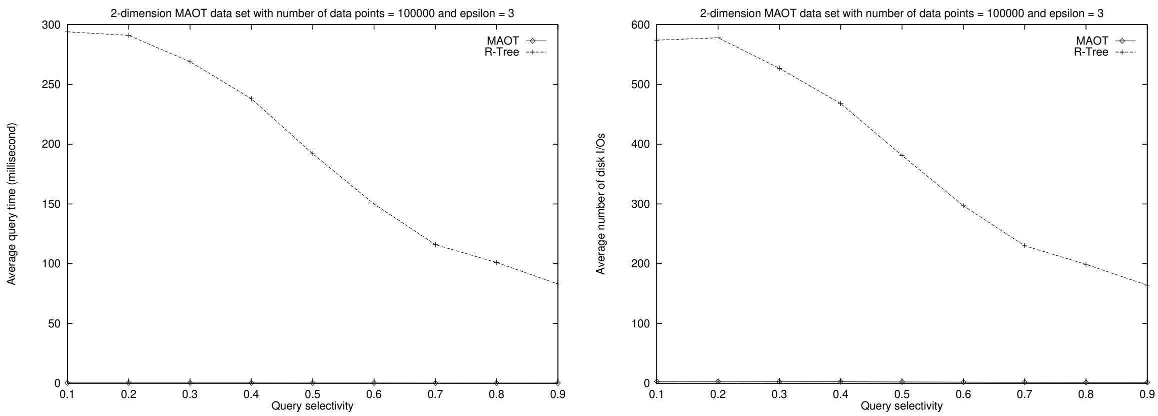


(a) Average query time comparison

(b) Average disk I/Os comparison

Figure 13: Effect of $epsilon$ on performance

The third setting is to evaluate the effect of query selectivity on the performance. In this setting, we ran queries on a two-dimensional MAOT data set with epsilon = 3 and data set size = 10^5. We then generated range queries by enforcing the selectivity constraint $k\%$ on time dimensions and then examined how they affected query time and disk I/Os. The selectivity constraint $k\%$ means that the length of any queried ranged is less than or equal to $k \times dom(i)/1000$ where $dom(i)$ represents the length of the domain. When generating the range queries, the start points of the ranges is selected uniformly from the domain of the dimensions and then the range is determined randomly according to the selectivity. Note that the range in every dimension is independently selected. The experimental results of about 40000 range queries are reported in Figure 14. We can observe that the performance of our technique does not change too much as the query selectivity decreases while the performance of R-tree technique degrades. Our technique requires at most a few number of hard disk accesses per query and thus query size do not really matter a lot when searching the query answer. However, that is the different case with R-tree technique, a small query size may need to go further down the tree to trace the data points which are actually are in the given query range. Consequently it may require more page swapping and thus lead more hard disk accesses, thereby more query time.



(a) Average query time comparison

(b) Average disk I/Os comparison

Figure 14: Effect of $query\ selectivity$ on performance

In all we can observe that our technique outperforms R-tree techniques in terms of answering range aggregate queries on MAOT data sets. Our technique has a good scalability with respect to the data set size and R-tree technique obviously has a poor scalability with respect to the data set size, i.e., query time and the number of disk I/Os grows linearly with respect to the data set size. Also our technique handle the skewness of range query size very well while the performance of R-tree degrades if the range query size is too small.

4.5 Practical Considerations

So far we assumed that ε is a given constant for a data set. However, since the update and storage costs depend linearly on its value, the choice is not arbitrary. This is intuitive because the larger the ε, the more the VT-dimension actually becomes a general dimension from which our technique can not benefit.

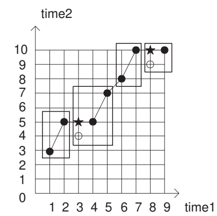
In practice it might occur that 99% of all data points satisfy the MAOT property for $\varepsilon = 1$, but there are a few extreme outliers which would require say $\varepsilon = 100$ to satisfy the MAOT condiditon for the VT-dimension. In that scenario it would be more efficient to maintain two separate data structures. The 99% of points that are at most $\varepsilon = 1$ off from their sentinel data point are maintained by our technique. The extreme outliers are stored in a general d-dimensional aggregation data structure, e.g., a general R-tree. Depending on the distribution of the outliers the ε threshold and hence the percentage of points maintained in each of the two data structures are selected such that the overall query cost is minimized. The exact examination of the optimization problem is part of our future work. Note that the more points are extreme outliers, the higher the percentage of points maintained in the general data structure. This is expected since the data set itself is also more general, i.e., our technique degrades gracefully with an increasing amount of extreme outliers.

5 Technique for d-dimensional Data Sets with Two MAOT Dimensions

For simplicity our technique was described for two-dimensional MAOT data sets only, i.e., data sets with a TT- and a VT-dimension only. We did not take into account non-temporal "general" attributes, e.g., `location`. Now we extend the technique to general data sets with two MAOT dimensions. To simplify the presentation, we first discuss a three-dimensional MAOT data set (shown in Figure 15) with $\varepsilon = 1$ and consider the dimensions `time1`, `time2` and `location`.

time1	time2	location	measure value
1	3	1	2
1	3	3	4
2	5	2	5
2	5	3	6
3	4 **(5)**	1	4
3	4 **(5)**	2	8
4	5	2	2
4	5	4	2
5	7	2	1
5	7	3	1
5	7	4	1
6	8	1	4
6	8	2	6
7	10	2	2
7	10	4	1
8	9 **(10)**	1	1
9	10	2	2
9	10	3	2
9	10	4	1

A three-dimensional MAOT data set D with `time1` as TT-dimension, `time2` as VT-dimension, and `epsilon` = 1



each point (circle or star) actually represents a one-dimensional slice

Figure 15: A three-dimensional MAOT data set

The three-dimensional data set is modelled as a collection of one-dimensional "time slices", i.e., columns of location coordinates for each pair of time coordinates. We therefore need to extend the data structure I to incorporate the location information and the extended data structure is denoted as I_1. Here we only describe the fields that need to be extended, the other fields remain as before. Let R_1 denote a data structure that supports range sum queries on a one-dimensional data set whose dimension is `location`, e.g., a B-tree with `location` keys or a DDC array [20].

1. `PSum` is generalized as an instance of R_1 which maintains all data points $X^3 = (x_1, x_2, x_3)$ whose time coordinates satisfy $x_1 \le t_1, x_2 \le t_2$. To distinguish from the original `PSum`, this field is denoted as $\texttt{PSum}_1$.

2. `OVal` is generalized as an instance of R_1 which maintains all data points $X^3 = (x_1, x_2, x_3)$ whose time coordinates satisfy $x_1 = t_1, x_2 = t_2$. To distinguish from `OVal`, this field is denoted as $\texttt{OVal}_1$.

3. `EPSum` still is a one-dimensional array of size ε which is used for outlier groups. However, instead of containing single aggregate values its cells now point to instances of R_1. More precisely, cell $\texttt{EPSum}[i]$ points to the instance of R_1 which maintains all data points $X^3 = (x_1, x_2, x_3)$ whose time coordinates satisfy $x_1 \le t_1, x_2 \le t_2$ but excludes those data points in the same outlier group with original VT-coordinate earlier than $s - i$, where s is the VT-coordinate of the starting data point of its group. To distinguish from `EPSum`, this field is denoted as $\texttt{EPSum}_1$.

The data structures for the example data set are shown in Figure 16. Here R_1 is a B-tree (indicated by the triangle). For example, time instance $I(3, 5)$ belongs to an outlier group and thus `GRP` is `outlier'; For each location key $k$, $\texttt{PSum}_1$ stores the sum of measure values of data points whose TT-coordinate is less than or equal to 3, VT-coordinate is less than or equal to 5, and `location` is equal to $k$. $\texttt{OVal}_1$ stores the measure values of data points whose TT-coordinate is equal to 3, VT-coordinate is equal to 5, and `location` is equal to $k$. $\texttt{EPSum}_1[0]$ stores the sum of measure values of data points whose TT-coordinate is less than or equal to 3, and VT-coordinate is less than or equal to 5, and `location` is equal to k, but excluding those data points which are in the same outlier group and whose original VT-coordinate is less than $5 - 0 = 5$.

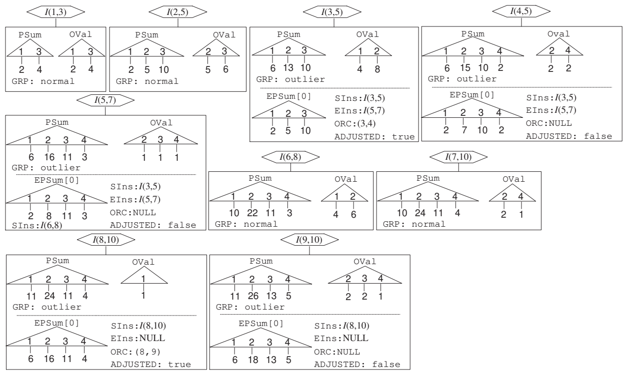


Figure 16: I_1-instances constructed for the three-dimensional MAOT data set shown in Figure 15

The query algorithms generalize naturally. Instead of retrieving a single value from `PSum`, `OVal`, or `EPSum`, a range query is performed on the corresponding $\mathtt{PSum_1}$, $\mathtt{OVal_1}$, or $\mathtt{EPSum_1}$. This range query returns the range sum for the selected range of `location` coordinates.

The generalization to data sets with $g > 1$ general dimensions is straightforward. Instead of using R_1, instances of a data structure R_g are used. This can be any data structure that supports aggregate range queries over the corresponding g-dimensional data space. A lookup of `PSum`, `OVal`, or `EPSum` now is replaced by a corresponding g-dimensional range query on $\mathtt{PSum_g}$, $\mathtt{OVal_g}$, or $\mathtt{EPSum_g}$, using the ranges selected by the original query in the general dimensions.

The algorithmic complexity of the approach is as follows. Assume the data set has N data points and d dimensions, where one is a TT- and one a VT-dimension (i.e., $g = d - 2$). The query complexity on the d-dimensional data set is reduced from $O(\mathrm{agg}(d))$ to $O(\mathrm{agg}(d-2) + \log N)$ with a small hidden constant (near 2). The proof is similar to the two-dimensional case an omitted due to space constraints. Here $\mathrm{agg}(x)$ denotes the best possible cost for answering an aggregation query on a *general* x-dimensional data set (e.g., using an R-tree or Range Tree). Hence for $d > 3$ the cost formula simplifies to $O(\mathrm{agg}(d-2))$. Intuitively we reduced the query complexity by two dimensions.

A simple implementation in the worst case constructs a new instance of the g-dimensional data structure R_g for each data point, hence consumes storage of $O(\varepsilon N \mathrm{mem}(d-2, N))$ where $\mathrm{mem}(x, y)$ denotes the storage requirements of an x-dimensional range aggregation data structure R_x for a data set with y points. This is not a bad result, but there is a high amount of redundancy because the instances of R_g for consecutive TT-dimension coordinates differ by only a few data points. To remove this redundancy, we use *multiversion* data structures [9]. More precisely, instead of having a separate instance of R_g for each data point's $\mathtt{PSum_g}$, we encode all R_g instances of the $\mathtt{PSum_g}$ of all data points in a *single* multiversion variant of R_g. We do the same for $\mathtt{EPSum_g}$. This somewhat restricts the choice of R_g (to those with existing efficient multiversion implementations), but we gain the following. Multiversion data structures typically support queries and updates at the same asymptotic cost as if the single instances were maintained separately. At the same time the storage overhead per insertion is constant. Stated differently, using multiversion constructions like the one by Brodal [3] or Becker et al. [1] we can reduce the storage requirement of our technique to $O(\varepsilon \cdot \mathrm{mem}(d-2, N))$. Similarly update costs are $O(\varepsilon \cdot \mathrm{upd}(d-2, N))$, i.e., asymptotically identical to the update cost $\mathrm{upd}(d-2, N)$ on a single instance of the g-dimensional (recall that $g = d - 2$) data structure R_g.

6 Conclusion and Future Work

Append-only data sets are increasing in popularity due to the increasing demand for both archival data as well as trend analysis of such data sets. OLAP applications that need aggregate support for different ranges for such data sets need to be efficient. In this paper, we formalize the notion of Multi-Append-Only-Trend (MAOT) property of historical data sets in data warehousing. Due to the sparsity of MAOT data sets, application of existing techniques leads to significant storage explosion. Thus, we propose a new technique for handling range aggregate (SUM) queries efficiently on MAOT data sets. This paper is thus a continuation of our development of techniques for managing large append-only data sets. In [21], we proposed a framework for efficient aggregation over data sets with a single append-only dimension. In this paper, we extended our development to data sets with multi-append-only-trend property. Data sets would benefit significantly if such dimensions are recognized and efficiently incorporated into analysis tools. The technique proposed in this paper essentially allows us to process a d-dimensional data set with two time-related dimensions as efficient as if it was only a $(d-2)$-dimensional data set. In particular a data set of size N that only has TT- and VT-dimensions can be searched and updated in $O(\log N)$ time, while requiring storage linear in N.

We are currently extending this approach to d-dimensional data set with arbitrary number of MAOT dimensions. Part of our future work is also to examine how to choose the ε parameter for data sets where the value is not known in advance.

References

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[2] C. Böhm and H.-P. Kriegel. Dynamically optimizing high-dimensional index structures. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 36–50, 2000.

[3] G. S. Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing*, 3(3):238–255, 1996.

[4] C.-Y. Chan and Y.E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 215–226, 1999.

[5] C.-Y. Chan and Y.E. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 675–686, 1999.

[6] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[7] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[8] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer Verlag, 2 edition, 2000.

[9] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences (JCSS)*, 38(1):86–124, 1989.

[10] M. Ester, J. Kohlhammer, and H.-P. Kriegel. The dc-tree: A fully dynamic index structure for data warehouses. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 379–388, 2000.

[11] C. S. Jensen et al. *Temporal Databases - Research and Practice*, volume 1399 of *LNCS*, chapter The Consensus Glossary of Temporal Database Concepts, pages 367–405. Springer Verlag, 1998.

[12] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[13] S. Geffner, D. Agrawal, and A. El Abbadi. The dynamic data cube. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 237–253, 2000.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[15] 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.

[16] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 401–412, 2001.

[17] V. Markl, F. Ramsak, and R. Bayer. Improving OLAP performance by multidimensional hierarchical clustering. In *Proc. Int. Conf. on Database Engineering and Applications Symp. (IDEAS)*, pages 165–177, 1999.

[18] P.E. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 38–49, 1997.

[19] M. Riedewald, D. Agrawal, and A. El Abbadi. pCube: Update-efficient online aggregation with progressive feedback and error bounds. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 95–108, 2000.

[20] M. Riedewald, D. Agrawal, and A. El Abbadi. Flexible data cubes for online aggregation. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 159–173, 2001.

[21] M. Riedewald, D. Agrawal, and A. El Abbadi. Efficient integration and aggregation of historical information. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, 2002.

[22] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 516–523, 1996.

[23] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, 1985.

Appendix

Algorithm Query
Input: *historical I-instances H,*
I-instance directory T,
outlier directory O,
$query(L^2, U^2)$

(1) $query(L^2, \cancel{U}^2)$ = expandQuery($query(L^2, U^2)$);
(2) $query(\cancel{L}^2, \cancel{U}^2)$ = surplusQuery($query(L^2, U^2)$);
(3) **return** subQuery($H, T, O, query(L^2, U^2)$)
− subQuery($H, T, O, query(\cancel{L}^2, \cancel{U}^2)$);

Function expandQuery
Input: $query(L^2, U^2)$

(1) $\cancel{U}^2.t_1 = U^2.t_1$; $\cancel{U}^2.t_2 = U^2.t_2$ + e;
(2) **return** $query(L^2, \cancel{U}^2)$);

Function surplusQuery
Input: $query(L^2, U^2)$

(1) $\cancel{L}^2.t_1 = \cancel{L}^2.t_1$; $L^2.t_2 = L^2.t_2$ + e;
(2) $\cancel{U}^2.t_1 = U^2.t_1$; $\cancel{U}^2.t_2 = U^2.t^2$ + e;
(3) **return** $query(\cancel{L}^2, \cancel{U}^2)$);

Note: e is the epsilon.

Figure 17: The range aggregate query algorithm and query reduction functions

```
Function subQuery
Input: historical I-instances H, I-instance directory T,
       outlier directory O, query(L^2,U^2)

(1) //search I-instance directory to find LB_ins and UB_ins
    LB_ins = IDirLookUp_LB(L^2);  UB_ins = IDirLookUp_UB(U^2);
(2) if LB_ins is NULL and UB_ins is NULL  return 0; //case 1
(3) else if LB_ins == UB_ins //case 2
           if LB_ins.ORC is NULL or inside the range
            return LB_ins.OVal;
(4)        else return 0;
(5)      if LB_ins.t_2 - L^2.t_2 >= e //case 3
           return UB_ins.PSum - LB_ins.PSum + LB_ins.Oval;
(6)      if LB_ins.GRP == 'normal' //case 4(i)
           return normalQuery(LB_ins, UB_ins);
(7)      else if LB_ins == LB_ins.SIns //case 4(ii)
                return outlierSQuery(H,T,LB_ins,UB_ins,query(L^2,U^2));
(8)           else //case 4(iii)
                return outlierMQuery(H,T,LB_ins,UB_ins,query(L^2,U^2));
```

```
Function outlierSQuery
Input: historical I-instances H,
       LB_ins, UB_ins, query(L^2,U^2)

(1) E_ins = LB_ins.EIns;
(2) if E_ins is NULL or not inside query range or E_ins == UB_ins
      if LB_ins aligns with the bottom line of the query
        return UB_ins.EPSum[0] - LB_ins.EPSum[0];
      else  return  UB_ins.PSum - (LB_ins.PSum - LB_ins.OVal);
(3) if E_ins is inside query range
      if LB_ins aligns with the bottom line of the query
        return UB_ins.PSum - E_ins.PSum
                + E_ins.EPSum[0] - LB_ins.EPSum[0];
        else return UB_ins.PSum - E_ins.PSum
                + E_ins.PSum - (LB_ins.PSum - LB_ins.OVal);
```

```
Function normalQuery
Input: historical I-instances H, outlier directory O,
       LB_ins, UB_ins, query(L^2,U^2)

(1) //find the time instance for the starting data point
    //of an outlier group which immediately follows its group
    OS_ins = ODirLookUp(time coordinates of LB_ins);
(2) //only contains data points of normal group
    if OS_ins is NULL or not inside  query range
      return UB_ins.PSum - LB_ins.PSum + LB_ins.OVal;
(3) else //contains data points of normal group and outlier groups
      return OS_ins.PSum - LB_ins.PSum + LB_ins.OVal - OS_ins.OVal
             + outlierSQuery(OS_ins,UB_ins,query(L^2,U^2));
```

```
Function outlierMQuery
Input: historical I-instances H, outlier directory O,
       LB_ins, UB_ins, query(L^2,U^2)

(1) E_ins = LB_ins.EIns;
(2) if LB_ins == E_ins
      I_ins = E_ins.IIns;
      if I_ins.GRP == 'normal'
        return E_ins.OVal + normalQuery(H,O,I_ins,UB_ins,query(L^2,U^2));
      else
        return E_ins.OVal + outlierSQuery(H,I_ins,UB_ins,query(L^2,U^2));
(3) if E_ins is NULL or not inside query range or E_ins == UB_ins
        return UB_ins.EPSum[0] - LB_ins.EPSum[0]
              + [LB_ins.OVal] <-- if LB_ins is not an adjusted one;
(4) if E_ins is inside query range
        return (UB_ins.PSum - E_ins.PSum) + (E_ins.EPSum[0] - LB_ins.EPSum[0]
              + [LB_ins.OVal] <-- if LB_ins is not an adjusted one);
```

Figure 18: Functions used in the range aggregate query algorithm

```
Function outlierSQuery
Input: historical I_e-instances H,
       LB_ins, UB_ins, query(L^2,U^2)

(1) E_ins = LB_ins.EIns;
    index = LB_ins.t_2 - L^2.t_2;
(2) if E_ins is NULL or not inside query range or E_ins == UB_ins
      if index < e
        return UB_ins.EPSum[index] - LB_ins.EPSum[index];
      else  return UB_ins.PSum - (LB_ins.PSum - LB_ins.OVal) ;
(3) if E_ins is inside query range
      if index < e
        return (UB_ins.PSum - E_ins.PSum)
                + (E_ins.EPSum[index] - LB_ins.EPSum[index]);
      else return (UB_ins.PSum - E_ins.PSum)
                + (Eins.PSum - (LB_ins.PSum - LB_ins.OVal));
```

```
Function outlierMQuery
Input: historical I_e-instances H, outlier directory O,
       LB_ins, UB_ins, query(L^2,U^2)

(1) E_ins = LB_ins.EIns;
    index = LB_ins.t_2 - L^2.t_2;
(2) if LB_ins == E_ins
      I_ins = E_ins.IIns;
      if I_ins.GRP == 'normal'
        return E_ins.OVal + normalQuery(H,O,I_ins,UB_ins,query(L^2,U^2));
      else
        return E_ins.OVal + outlierSQuery(H,I_ins,UB_ins,query(L^2,U^2));
(3) if E_ins is NULL or not inside query range or E_ins == UB_ins
      return UB_ins.EPSum[index] - LB_ins.EPSum[index]
            + [LB_ins.OVal] <-- if LB_ins is actually inside the range;
(4) if E_ins is inside query range
      I_ins = E_ins.IIns;
      if I_ins.GRP == 'normal'
        return (E_ins.EPSum[index] - LB_ins.EPSum[index] +
          + [LB_ins.OVal] <-- if LB_ins is actually inside the range;)
            + normalQuery(H,O,I_ins,UB_ins,query(L^2,U^2));
      else return (E_ins.EPSum[index] - LB_ins.EPSum[index] +
          + [LB_ins.OVal] <-- if LB_ins is actually inside the range;)
            + outlierSQuery(H,I_ins,UB_ins,query(L^2,U^2));
```

Figure 19: Algorithm for two-dimensional MAOT data sets with $\varepsilon > 1$