AN EFFICIENT ALGORITHM TO COMPUTE MAX/MIN VALUES IN SLIDING WINDOW FOR DATA STREAMS

Ying Sha and Jianlong Tan
The Software Division, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

Keywords: Max/Min, Sliding Window, Data Streams.

Abstract: With the development of Internet, more and more data-stream based applications emerged, where calculation of aggregate functions plays an important role. Many studies were conducted on aggregation functions; however, an efficient algorithm to calculate Max/Min values remains an open problem. Here, we propose a novel, exact method to computer Max/Min values for the numerical input data. Employing an incrementally calculating strategy on sliding windows, this algorithm gains a high efficiency. We analyze the algorithm and prove the time-complexity and space-complexity in worst cases. Experimental results confirm its high performance on a testing dataset.

1 INTRODUCTION

With the development of Internet, more and more data-stream based applications emerged, such as financial applications (Cortes et al., 2000),telephone monitoring applications(Gilbert et al., 2001),etc. Generally speaking, a data stream is a sequence of data elements that enter into a data-stream processing system in order.(Dani and Getta, 2005; Guha et al., 2001; Henzinger et al., 1998); in other words, the arrival of input data is instant, continuous, and dynamic. For the limit of both memory space and time of response, only a small part of the input data sequence can be stored when a query is being processed. Here, we use sliding window to denote the latest n data elements.

Aggregate functions, the functions that compute out a single value from a group of values, are important to a data-stream system since query should be responded in real time. The common functions include COUNT, SUM, AVG, Max, and Min(Dobra et al., 2002). An aggregate function calculation method is exact if it always returns the exact value for a given query; otherwise, it is an approximate one. This paper focused on developing an efficient exact calculation method to calculate Max/Min value in a sliding window of data streams. In this paper, we only consider the situation that the window fits in memory.

The basic idea of the method in this paper is to simulate the data sequence in sliding window as a broken curve, and only the data elements between crests(troughs) of the broken curve is stored as historical data. Thus the new Max value can be calculated by comparing the value of the latest data element with the historical data, avoiding saving and scanning the whole previous data. Theoretical analysis shows that the time-complexity is $O(N\log M)$and the space-complexity is $O(M)$ in worst case, where N is the number of the input data elements and M is the size of sliding window; and experimental results confirms the high performance of this method. In (Cormode and S.Muthukrishnan, 2003)the authors prove that finding the maximum in a sliding window of size m, is impossible in space $O(m)$. But this paper also consider the situation that the window does not fit in memory. It is more difficult.

The remainder of the paper is organized as follows. Section 2 describes the algorithm to computing Max/Min value; Section 3 analyzes the time-complexity and space-complexity as well; Section 4 shows the experimental results; Section 5 concludes the whole paper.

Sha Y. and Tan J. (2007).
AN EFFICIENT ALGORITHM TO COMPUTE MAX/MIN VALUES IN SLIDING WINDOW FOR DATA STREAMS.
In *Proceedings of the Second International Conference on Software and Data Technologies - Volume ISDM/WsEHST/DC*, pages 123-128
DOI: 10.5220/0001338101230128

2 ALGORITHM TO COMPUTE MAX/MIN VALUE BASED ON SLIDING WINDOW

2.1 Basic Idea

Let $X = x_1, x_2, \ldots, x_n$ denote the arriving data sequence, W denote the width of the sliding window. We only consider the situation that the window fits in memory. In a diagram showing sliding window, x-axis represents time, and y-axis is the value of the data element. A newly arriving data will be inserted into the window from right, and step out of the window from left when expire. Calculation of the Max/Min value is executed when a new data element arrives or an old data element steps out of the sliding window, which are described in more details as follows:

Case 1.New data element arrives

We only need to store the current Max value of the sliding window. Let Max_{n-1} be the current Max value of the sliding window before a new data element x_n arrive, the new current Max value Max_n can be calculated as follows:

$$Max_n = \begin{cases} x_n & \text{if } x_n >= Max_{n-1} \\ Max_{n-1} & \text{if } x_n < Max_{n-1} \end{cases} \quad (1)$$

In the same way, we can get the new current Min value Min_n:

$$Min_n = \begin{cases} x_n & \text{if } x_n <= Min_{n-1} \\ Min_{n-1} & \text{if } x_n > Min_{n-1} \end{cases} \quad (2)$$

We can save Max_n and Min_n as historical data.

Case 2. Old data element steps out

When old data element expires, some data elements should be saved as historical data. First we need simulate the data sequence in the sliding window as a broken curve. Figure 1 illustrates the various cases of removing old data element from the window.

case a: The data sequence in the sliding window monotonically increases.

- **Max** In order to get the Max value, we only need to save one data element, the wave crest of the broken curve, as the historical data.
- **Min** In order to get the Min value, we need to save all elements in the sliding window(The data elements in blue).
- **Max/Min** In order to get both Max value and Min value, we need to save all elements in the sliding window.

case b: The data sequence in the sliding window monotonically decreases.

- **Max** In order to get the Max value, we need to save all elements in the sliding window(The data elements in red).
- **Min** In order to get the Min value, we only need to save one data element, the wave trough of the broken curve, as the historical data.
- **Max/Min** In order to get both Max value and Min value, we need to save all elements in the sliding window.

case c: The data sequence in the sliding window contains an increasing part followed by a decreasing part. It includes three cases.

- case c1:

  Max In order to get the Max value, we need to save elements between the crest and the newest data element in the sliding window(The data elements in red).

  Min In order to get the Min value, we only need to save one data element, the newest data element(The data element B).

  Max/Min In order to get both Max value and Min value, we need to save elements between the crest and the newest data element in the sliding window(The data elements in red).

- case c2:

  Max In order to get the Max value, we need to save elements between the crest and the newest data element in the sliding window(The data elements in red).

  Min In order to get the Min value, we only need to save one data element, the newest data element(The data element B).

  Max/Min In order to get both Max value and Min value, we need to save elements between the crest and the newest data element in the sliding window(The data elements in red).

- case c3:

  Max In order to get the Max value, we need to save elements between the crest and the newest data element in the sliding window(The data elements in red).

  Min In order to get the Min value, we need to save elements between the trough and the newest data element in the sliding window(The data elements in blue).

  Max/Min We need to save elements between C and B for Max value(The data elements in read), and elements between A and D for Min value.

case d: The data sequence has multiple peaks and troughs.

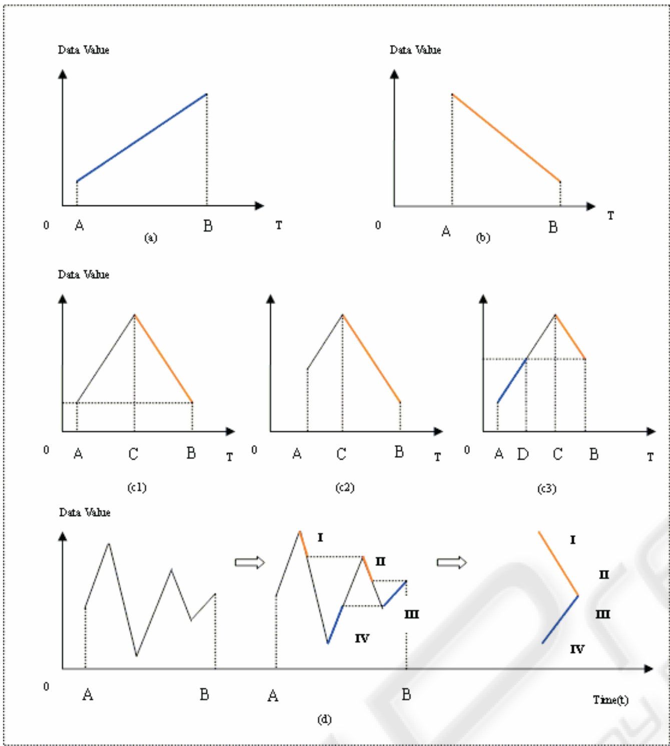

Figure 1: Analyze all kinds of broken curves simulated by data stream.

Max In order to get the Max value, we can transform it into case c by *compress* operation, that is, recursively remove the data elements below the right crest. Thus only the decreasing part(the data elements in red, I and II) should be saved as the historical data.

Min In order to get the Min value, we can also use the *compress* operation. Thus only the increasing part (the data elements in blue, III and IV) should be saved as the historical data.

Max/Min Thus the historical data is a data sequence in order including two parts: 1. I and II for Max value; 2. III and IV for Min value.

According to the discussion of case a,b,c and d, a data sequence in the sliding window can be simulated as a broken curve(Figure 2). We can "compress" the broken curve. The historical data for Max value are the data elements whose value is between Max value and the value of the newest data element, and these data elements are in the "downgrade","outboard" of the broken curve. The historical data for Min value are the data elements whose value is between Min value and the value of the newest data element, and these data elements are in the "upgrade","outboard" of the broken curve. For example, in Figure 2, the historical data for Max value are AB part and CD part(the data elements in red); the historical data for Min value are the data element D, EF part and GH part (the data elements in blue and the the newest data element). When multiple maximum(minimum) values exists, that is to say that there are some data elements which have the same value, we only need to save the newest data element.

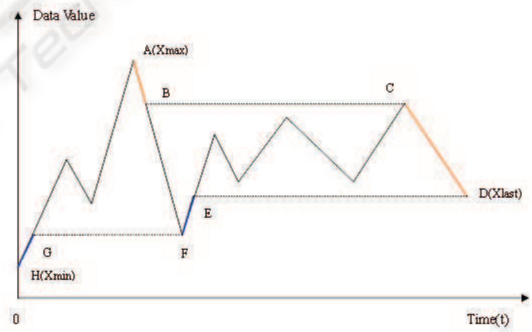

Figure 2: The broken curve simulated by data list:$X = x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_n$.

2.2 Algorithm Description

Let $X = x_1, x_2, \ldots, x_n$ denote the data sequence, W is the size of the sliding window, x_{max} denote the current Max value of the data sequence in the sliding window, x_{min} denote the current Min value of the data sequence in the sliding window, x_{last} denote the newest data element in the sliding window, $X[history]$ denote the historical data need to save. Using the idea of section 2.1 get the historical data. $X[history]$ includes $X[hist_{max}]$ and $X[hist_{min}]$. $X[hist_{max}]$ is used for getting the Max value, $X[hist_{min}]$ is used for getting the Min value. $X[hist_{max}]$ and $X[hist_{min}]$ are all in order. $X[hist_{max}]$ is from x_{max} to x_{last} , $X[hist_{min}]$ is from x_{last} to x_{min}.

case 1: When new data element x_{new} inserts into the sliding window

Update Max value (the new current Max value: x'_{max}):

$$x'_{max} = \begin{cases} x_{new} & \text{if } x_{new} >= x_{max} \\ x_{max} & \text{if } x_{new} < x_{max} \end{cases} \quad (3)$$

Update Min value (the new current Min value: x'_{min}):

$$x'_{min} = \begin{cases} x_{min} & \text{if } x_{new} > x_{min} \\ x_{new} & \text{if } x_{new} <= x_{min} \end{cases} \quad (4)$$

Update historical data $X'[hist_{max}]$:

$$X'[hist_{max}] = \begin{cases} X[hist_{max}] + x_{new} & \text{if} x_{new} < x_{last} \\ X[hist_{max}] - x_{last} + x_{new} & \text{if} x_{new} = x_{last} \\ H[hist_{max}](x_{max}, x_{new}) + x_{new} & \text{if } x_{new} > x_{last} \end{cases} \quad (5)$$

Update historical data $X'[hist_{min}]$:

$$X'[hist_{min}] = \begin{cases} X[hist_{min}](x_{new}, x_{min}) + x_{new} & \text{if } x_{new} < x_{last} \\ X[hist_{min}] - x_{last} + x_{new} & \text{if } x_{new} = x_{last} \\ H[hist_{min}] + x_{new} & \text{if } x_{new} > x_{last} \end{cases} \quad (6)$$

case 2: When old data element x_{old} expires,

Update Max value(the new current Max value: x'_{max}):

$$x'_{max} = \begin{cases} max\{x_i | x_i \neq x_{max}, x_i \in X[hist_{max}]\} & \text{if } x_{old} = x_{max} \\ x_{max} & \text{if } x_{old} \neq x_{max} \end{cases} \quad (7)$$

Update Min value(the new current Min value: x'_{min}):

$$x'_{min} = \begin{cases} min\{x_i | x_i \neq x_{min}, x_i \in X[hist_{min}]\} & \text{if } x_{old} = x_{min} \\ x_{min} & \text{if } x_{old} \neq x_{min} \end{cases} \quad (8)$$

Update historical data$X'[hist_{max}]$:

$$X'[hist_{max}] = \begin{cases} X[hist_{max}] - \{x_{max}\} & \text{if } x_{old} = x_{max} \\ X[hist_{max}] & \text{if } x_{old} \neq x_{max} \end{cases} \quad (9)$$

Update historical data$X'[hist_{min}]$:

$$X'[hist_{min}] = \begin{cases} X[hist_{min}] - \{x_{min}\} & \text{if } x_{old} = x_{min} \\ X[hist_{min}] & \text{if } x_{old} \neq x_{min} \end{cases} \quad (10)$$

2.3 The Implementation of Algorithm

From the above sections, we know that the historical data includes $X[hist_{max}]$ and $X[hist_{min}]$. $X[hist_{max}]$ is used for getting the Max value, $X[hist_{min}]$ is used for getting the Min value. $X[hist_{max}]$ and $X[hist_{min}]$ are all in order. We use a array $max[]$ to represent $X[hist_{max}]$ which is from the Max value to x_{last} in monotonic decrease order, a array $min[]$ to represent $X[hist_{min}]$ which is from the Min value to x_{last} in monotonic increase order. We use *LastValue* to represent x_{last}. So MaxValue=*max*[0], MinValue=*min*[0].

The details of the algorithm of computing MAX/MIN function as follows:

Case 1: insert new data element into the sliding window

```
void InsertData(const void *pData) {
  if(*pData >= MaxValue)
      //MaxValue is the current Max value
      MaxValue = *pData;
  else {
      if(*pData <= MinValue)
      // MinValue is the current Min value
      MinValue = *pData;    }

  //Update the historical data
  if(*pData < LastValue){
      max[].append(*pData); // Update max[]
      min[].insert(*pData); // Update min[]
      /* insert *pData into the array min[]
      which is in order. */

      min[].delete(*pData,LastValue);
      /* delete all data elements which
      are large than the *pData. */      }
  else {
      if(*pData > LastValue){
        max[].insert(*pData); // Updat max[]
        /* insert *pData into the array max[]
        which is in order. */

        max[].delete(*pData,LastValue);
        /* delete all data elements
          which are less than the *pData. */

        min.append(*pData); // Update min[]}
      else {
          max[LastValue]=*pData;
```

```
        min[LastValue]=*pData;
    /* replace the last element of max/min array
    with the new data element. */
} } }
```

Case 2: delete old data element from the sliding window

```
void DeleteData(const void *pData) {
  if(*pData == MaxValue){
     max[].delete(0);
     /* delete the first element of the array. */

     MaxValue=max[0];  // Update the Max value }
  else {
     if(*pData == MinValue) {
       min[].delete(0);
       //delete the first element of the array.

       MinValue=min[0];//Update the Min value.
  }}}
```

3 SPACE AND TIME COMPLEXITY

The time complexity of computing Max/Min function is mainly determined by looking up process. When a new data element which inserts into the sliding window is less than the current Max value or large than the current Min value, this algorithm need to find the proper position for this data element.

Suppose the length of data sequence is N. The size of the sliding window is M. When a data element x_k inserts into the sliding window, x_k is less than the current Max value or large than the current Min value. There are K data elements in Memory block, then the time complexity of finding the proper position of x_k is $O(\log K)$. So the time complexity of computing Max/Min function is $O(N \log M)$ in worst case. It is easy to know that the space complexity of computing Max/Min function is $O(M)$ in worst case, because we only consider the situation that the window fits in memory.

4 EXPERIMENTAL STUDY

In this section, we present the results of an extensive experimental study of our methods of Max/Min functions using the random data as input data elements. The type of input data element is double.

Let TotalCount denote the total number of data elements inserts into the sliding window. These data elements insert into the sliding window and step out of the sliding window. Let MaxStoreCount denote the max number of data elements need to save. Let AvgStorePercent denote the average percent of the number of data elements need to save according to the size of the sliding window.

In this experiment, let TotalCount be 500000, the size of sliding window is 100,200,500,1000,5000,10000,50000,1000-00,then record MaxStoreCount(the max number of data elements need to save) and AvgStorePercent(the average storage percent). In order to improve the correction of the experiment, we do 20 times experiments on the same size of sliding window, then compute the average value of MaxStoreCount and AvgStorePercent.

Table 1: sliding window size=100, input data=50000.

No	AvgStoPer	MaxStoCount
1	10.37	32
2	10.38	32
3	10.43	32
4	10.38	32
5	10.43	32
6	10.35	30
7	10.35	30
8	10.50	32
9	10.35	32
10	10.50	32
11	10.50	32
12	10.30	32
13	10.30	32
14	10.30	32
15	10.30	32
16	10.30	32
17	10.35	32
18	10.34	32
19	10.36	32
20	10.35	32

Table 1 shows the results of total 20 times experiments (TotalCount=500000, the size of slide window=100).Table 2 gives the average results of different size of sliding windows. We take 20 times experiments for each size of sliding windows, then get the average value of these experiments.

From Table 2, we can get that MaxStoreCount increases as the size of sliding window increases. However, the increase of MaxStoreCount is very slowly. We can get that AvgStorePercent decreases as the size of sliding window increases. Figure 3 and Figure 4 show the same result. Therefore, this method has better performance for larger size of sliding window.

In summary, the experimental results and performance study show that the data elements need to save

Table 2: Average Results of different sizes of sliding windows(20times).

No	WinSize	AvgStorePer(%)	MaxStoreCount
1	100	10.37	32
2	200	5.90	30
3	500	2.70	38
4	1000	1.52	42
5	5000	0.37	46
6	10000	0.20	48
7	50000	0.15	128
8	100000	0.08	178

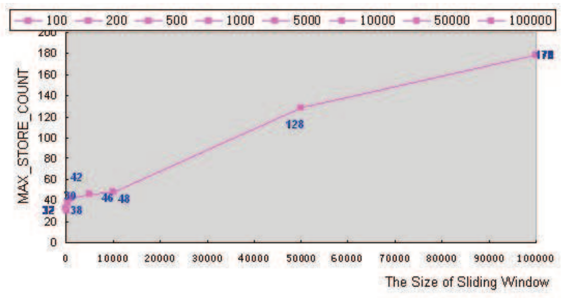

Figure 3: MaxStoreCount of different size of sliding window.

is rather small compare with the size of sliding window. The average store percent of the data elements decreases as the size of sliding window increases. So this method can largely reduce the number of historical data elements need to be saved.

5 CONCLUSION

This paper proposed a new exact aggregation method using as little as possible stored historical data for the Max/Min function for the numerical input data. This method is an inclemently calculatingly method based on sliding windows of data stream. The time complexity of this algorithm is $O(N \log M)$and the space-complexity is $O(M)$in worst cases(N is the length of the data sequence, M is the size of the sliding window). And experimental results confirm its high efficiency.

Ongoing work is comparing this algorithm with other methods and using several different data distributions. We plan to build a larger test dataset and thoroughly test this approach. An interesting problem is whether we can analysis the theoretical average cost for some given distribution.

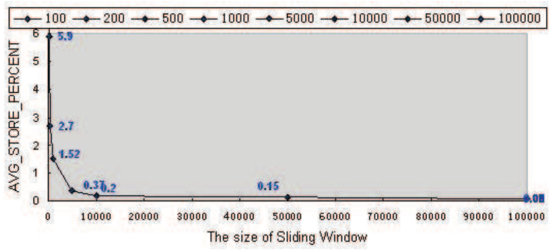

Figure 4: AvgStorePercent of different size of sliding window.

REFERENCES

Cormode, G. and S.Muthukrishnan (2003). What's hot and what's not: Tracking most frequent items dynamically. *ACM Principles of Database Systems(PDOS)*, pages 296–306.

Cortes, C., Fisher, K., Pregibon, D., Rogers, A., and Smith, F. (2000). Hancock: A language for extracting signatures from data streams. *Proc. Sixth Int'l Conf. Knowledge Discovery and Data Mining*, pages 9–17.

Dani, A. and Getta, J. (2005). Conceptual modeling of computations on data streams. *Conferences in Research and Practice in Information Technology Series*, 107:43–47.

Dobra, A., Gehrke, J., Garofalakis, M., and Rastogi, R. (2002). Processing complex aggregate queries over data streams. pages 61–72.

Gilbert, A., Kotidis, Y., Muthukrishnan, S., and Strauss, M. (2001). Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. *Proc. 27th Int'l Conf. Very Large Data Bases*, pages 79–88.

Guha, S., Koudas, N., and Shim, K. (2001). Data-streams and histograms. *Annual ACM Symposium on Theory of Computing*, pages 471–475.

Henzinger, M. R., Raghavan, P., and Rajagopalan, S. (1998). Computing on data streams. *Technical report TR 1998-011, Compaq Systems Research Center, Palo Alto, CA,.*