An Incremental Computation Scheme over Array Database

JIANG LI[†1] KAWASHIMA HIDEYUKI[†2]

Array database is a new kind of database, dealing with the big data storing and processing in science fields. The window aggregation is a typical aggregate query of it. However, in the current array databases, naive method is used to calculate the window aggregates, which will do much redundant computation and lead to low efficiency. In this thesis, I propose an algorithm improved with incremental computation for window aggregate queries. It buffers some intermediate aggregate results and reuses them when computing, eliminates the redundant computation. The time complexity analysis is also offered, which shows the advantages of the improved algorithm clearly. In order to test the performance, a simple array data process system is implemented, with both naive method and incremental computation algorithm implemented inside the system. The test result also shows that the improved algorithm is much better than the naive one, especially when the aggregated window is big.

1. Introduction

1.1 Background

Array database is a new kind of databases developed recently. Unlike traditional relational databases, array database take multidimensional arrays instead of table as the basic store and process data units. As science and industry are growing increasingly data-intensive, array database is designed to solve the big data storage and processing. The array structure naturally fits in the data schema very well in many science fields and has many advantages over table when storing or processing data analyze processes.

Array data model has two kinds of elements: dimensions and attributes. As showed in figure 1, 'I' and 'J' are dimensions of the array and the array has two attributes, one is integer and the other is float type. The attribute values exist in cells which are determined by the combination of dimension values, as something like coordinators.

Because the special data model, array database has some queries which relational database doesn't have. Window aggregates is a typical useful query in array database that towards array data analyzing and processing. It's also the major subject this thesis will discuss about.

1.2 Goal and plan

The goal of the algorithm design is to apply the idea of incremental computing to reduce the redundant calculation and speed up the window aggregate queries.

In order to actually be able to run and test designed method, need to implement an array processing system that supports the basic operators and queries in common array database. Then implement the incremental computing method for window queries as well as naive method so that the testing of efficiency differences between the two methods can be done. As window aggregates, mainly implement the 4 kinds of aggregate functions: sum, average, maximum and minimum. Other more complex functions are not discussed in this paper.

2. Basic and Plan

This section introduces some basic concepts which need to be explained first before discussing the incremental computation optimization over window aggregations.

2.1 Array data processing system

The topic is about optimizing a kind of query in array database, it will require a platform to implement the designed algorithm and process the test works. So I implemented a small array data processing system.

This system takes multidimensional arrays as basic process unit. Since the main point of this topic is to design algorithm to improve efficiency of window query, there is no need to study the lower level storage structure and mechanism in details. So in the system, the data used by the query will be simply maintained in the memory. In this array data processing system, fundamental operators of array as well as some array queries are implemented.

```
jl@jl-SVZ1311AJ:~/Arr/trunk$ ./test.exe
CMD> create A [i=1:3,j=1:3] <double a,double b>
CMD> list
A   [ i=1:3 j=1:3 ]   < double a,double b >
CMD> load A
Input the Array values as dense matrix format:
1 2  3 4  5 6
7 8  9 0  9 8
7 6  5 4  3 2
CMD> select A
 ( ( (1,2)  (3,4)  (5,6) )  ( (7,8)  (9,0)  (9,8) )  ( (7,6)  (5,4)  (3,2) ) )
CMD> select A <b>
 ( ( (2)  (4)  (6) )  ( (8)  (0)  (8) )  ( (6)  (4)  (2) ) )
CMD> aggr A avg(a)
5.44444
CMD> aggr A min(a) by regrid(2,2)
 ( ( (1)  (5) )  ( (5)  (3) ) )
CMD> aggr A sum(b) by window(2,2)
 ( ( (14)  (18)  (14) )  ( (18)  (14)  (10) )  ( (10)  (6)  (2) ) )
```

Figure 1 Interface of the Array Data Process System

Figure 1 above show the running interface of the system. The displayed last query command 'aggr A sum(b) by window(2,2)' is asking to run a query of window aggregate, which is exactly the discussing query in this thesis. The command means that

†1 Tsukuba University
†2 Tsukuba University

process window aggregate over array ‘A’ on attribute ‘b’ with the aggregate function ‘sum’ and the aggregating window size is 2*2. More information about window aggregates is introduced below.

2.2 Window aggregates

Window aggregation query is the major subject to be improved in this thesis. It is necessary to introduce how window aggregation works before discussing about the incremental computing method.

Window aggregates allow you to specify groups by a moving window and compute aggregates over these windows. Here, a window is more like a subarray of the original array, defined by a size in each dimension. The moving window starts at the first element of the array and moves in stride-major order from the lowest to highest value in each dimension.

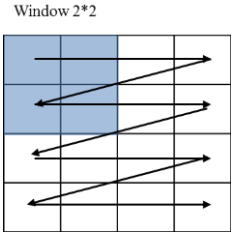

Figure 2 Window Aggregates

In different Array Databases, the syntax expression of window aggregates may be different and in this thesis I will describe it using the expressions in my Array Processing System. Check this simple example of window aggregates:

For a 2-dimensional array ‘A’ with size 4x5, which has a integer attribute named ‘a’, calculate the following window query over it:

Aggr A max(a) by window(2,3)

It means for array ‘A’, calculate max aggregate function on attribute ‘a’ over moving window with size 2x3. The query result is showed in figure 3.

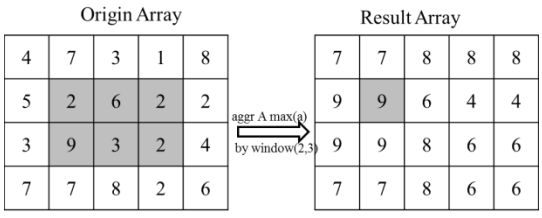

Figure 3 Window aggregate example

The result of a window query will be an array with the same size of the original array, the element value of a cell in the result array will be the aggregate value of the cell’s corresponding window in the original array. In my system, I define each cell’s corresponding window is the window taking the cell as the top-left unit, as the grey parts showed in figure 3. This definition of window may vary in different array databases, like SciDB take one cell’s corresponding window with the cell as its centroid. Still these differences over forms won’t affect the indeed property of window queries.

In the Array Processing System, I first implement the naive method of the window aggregates, which is also the common used algorithm for calculating window queries in current array databases. Is simply travel every cell for each window and calculate the aggregate value. But it is obvious that in this way, there is a lot of redundant computation. As the neighboring windows have many overlap areas, the naive method will scan and calculate many areas repeatedly. Although this method is easy to implement, lots of redundant operations will waste much process resource. So I bring the idea of incremental computing into the window aggregates, by taking uses of intermediate aggregate results, reuse them when computing to eliminate the redundant computation.

2.3 Base Window and Window Unit

In order to describe the incremental computation algorithm more clearly, some concept definitions will be shown first here.

- Base Window

Base windows represent these basic windows in incremental computing, treated as the start window for each computing round. In an n-dimensional array, each base window is determined by the first n-1 dimension value. In each computing round, based on a base window and move on the last dimension, I call the generated windows as the base window’s ‘derivative windows’.

- Window Unit

Window units are the basic computing units in the process of incremental computing of window aggregates. Each window consists of window units, as shown below:

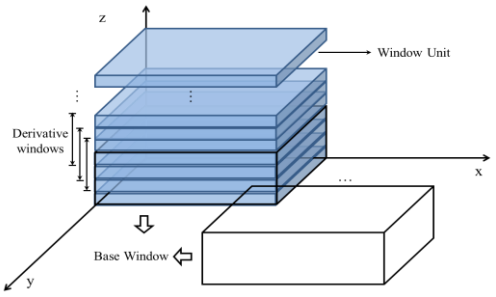

Figure 4 Base window and window units(in 3D array)

3. Incremental Compute

3.1 Algorithm overview

The main idea of the incremental computing improvement of

the window aggregates is to buffers some intermediate aggregate results and reuses them when computing, eliminates the redundant computation. For an N-dimensional array, the incremental compute steps will be like:

1. Generate a base window, scan the elements inside it and calculate the window's aggregate value in unit of window unit. Initialize the buffer for storing the intermediate aggregate values.
2. Start the compute round for the base window generated in step 1, take it as the start point, moving on the last dimension step by step. Each moving step will lead to a new corresponding derivative window. Compare to the previous computed window, this new window only add a new window unit and reduce an old window unit. The rest area of these two windows is exactly the same, can be computed incrementally.
3. Generate new base windows, repeat step 1~2, do the compute rounds until all windows of the origin array have been calculated.

3.2 Sum/Avg aggregates

The calculating method of sum and average aggregate functions are almost the same. To get the average value of the window, just need to divide the sum with the size of the calculating window. So here consider these two aggregate functions together.

For the sum/avg aggregate function, the main idea of accelerating is to compute the sum of each window's element values incrementally. Here, I use a sum list to store needed intermediate computed results. This sum list cost w_n space, storing sum values of every window unit belongs to the window that has been computed in the previous moving step. Since most window units of the current computing window are same as the previous computed window, we can reuse the sums stored in the sum list to calculate the current window's sum quickly. Here is a simple example to show how incremental computing works:

Array : A[i = 1:5, j = 1:5] <int a>
Query : aggr A sum(a) by window(3,3)

As the steps mentioned in Chapter 3.1:

1. Generate a base window, scan each window unit of it and compute these window units' sums, initialize the sum list with the sum values.

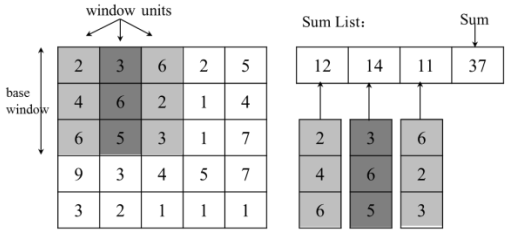

Figure 5 Initialize base window and sum list (process step 1)

2. Moving on the last dimension, in this case the second dimension 'j', moving the window to the right by step of one window unit, then will get a new derivative window. Scan the new coming window unit, calculate its sum and update the sum list, replace the sum value of the oldest window unit. After the updating, the total sum of the values in the sum list is exactly the aggregate sum of the current window.

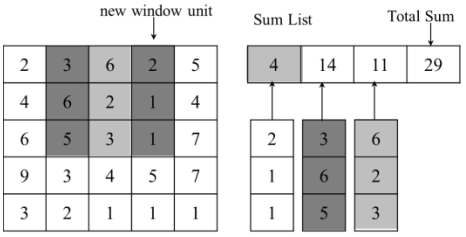

Figure 6 Process new window unit, update sum list(step 2)

3. Keep move forwards, calculate the aggregate value of all the windows derived by the generated base window in step 1. The figure 7 below shows the detailed value of sum list in each window moving step. The grey ones are the updated cells in that step and the total sum is exact the aggregate sum for the window in that step.

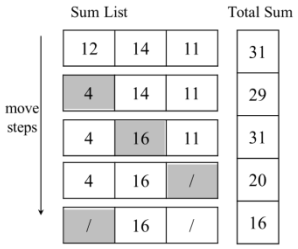

Figure 7 Sum list updating status

4. Keep moving the base window down to get new base windows, for each base window repeat step 1~3, which can be considered as a compute round. After finish all the base windows' compute rounds, the aggregate query also has been done completely.

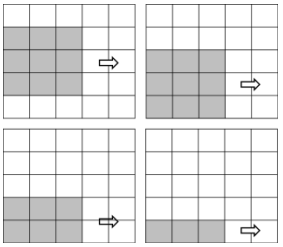

Figure 8 Move on base window, repeat compute round

For the arrays with more than 2 dimensions, the main idea of incremental computing of sum/avg functions is similar: use sum list to mention the window units' sums of the previous

calculated window. Then there is no need to compute the current window's most window units, whose aggregate sums can be easily get from the sum list. Only one window unit, the new coming one needs to be scanned.

Here is another example of 3-dimensinal array. Taken window size as wx*wy*wz. In order to show the figure more clear, take wz equals 4. As showed in figure 9, the outside cuboid represents the whole array, the small cuboid inside represents a base window, it is consisted of wz(4 in this case) window units.

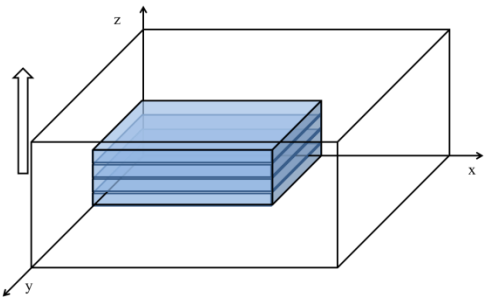

Figure 9 3D array incremental computing

The main process of incremental computing is exactly the same as 2-Dimensional array. Move the window towards the direction of z-axis, get new derivative windows, calculating their aggregate values incrementally by using the sum list. After all the windows that derived from the current base window are all completed, moving on the direction of x-axis, y-axis to generate new base windows and repeat the process described above to finish all the windows inside the original array. Then the window query has also got the aggregate result.

3.3 Min/Max aggregates

For the aggregate functions of maximum and minimum, the improvement of window queries by using incremental computing will get a little complex. Since only use a buffer sequence like sum list which is used in sum/avg functions, still can't get the min/max value among all these data in the buffer sequence quickly. We need to scan this buffer sequence of min/max values to get the current window's aggregate value. Of course this will cost some time and lower the efficiency of the method, especially when the window size is large.

In order to solve this problem, I apply the data structure of 'heap' instead of simple buffer sequence to maintain the max/min values of the previous derivative window. By using heap, it will save lots of time getting the min/max value among all the intermediate aggregate results. In the introduction below I will take min function as example for simple description.

The detailed process:

1. Generate a base window, compute the min value of every window unit inside it. Insert these min values into the heap. For the nodes in the heap, they contains two values, one is the node's corresponding window unit's min value 'v', the other one is the position 'p' of the window unit in the array.
2. Moving towards on the last dimension, process the derivative windows of the base window one by one and calculate their min aggregate incrementally. First scan the new window unit which has just joined the current window, get its min value. Then insert this min value together with the window unit's position into the heap.
3. Since the window is moving, the window unit has the minimum value may be already outside the computed window. Check the top node of heap, if its corresponding window unit is no longer in the current window (use position 'p' to judge), delete the top node. Keep doing this until the top node's corresponding window unit is inside current window. Then the top node's value is exactly the current window's min value.
4. Keep move the base window down to get new ones, for each base window repeat step 1~3, which can be considered as a compute round. After finish all the base windows' compute rounds, the aggregate query also has been done completely.

For window aggregate of min/max functions, the core idea remains the same. The usage of heap only speeds up the process of reusing the buffered intermediate aggregate results. The basic algorithm is same as sum/avg function. Still, because the heap operations need to spend some extra time, min/max function will be slower than sum/avg function at some degree.

4. Analysis

This chapter will analyze time complexity of naive method and incremental computing algorithm of window aggregates. Compare these two methods and check the improvement theoretically.

Before analysis, some parameters in window aggregate need to be defined so that the description can be clear and convenient:

2-dimensional array size	X·Y
Window size	a·b

General case:

N-dimensional array size	$D_1 \cdot D_2 \cdot \ldots \cdot D_n$
Window size	$w_1 \cdot w_2 \cdot \ldots \cdot w_n$

4.1 Naive method

First consider the simple 2-dimensional array, there are $X \cdot Y$ windows need to be calculated at all. For each window, every cell inside it needs to be scanned to compute the window's aggregate value. All the windows' size can be viewed as $a \cdot b$. No matter what aggregate function the window query uses, the compute process over each cell will be constant, so the time complexity will be

$$O(X \cdot Y \cdot a \cdot b)$$

Expand to general cases, time complexity of window aggregates' naive method is

$$O(\prod_{i=1}^{n} D_i \cdot \prod_{i=1}^{n} w_i)$$

4.2 Incremental computing

4.2.1 Sum/Avg

Also consider the 2-dimensional array first. At total X base windows exist. For each base window, need to move on the second dimension to get windows to process the incremental computing and there are Y move steps at all. In other words, each base window has Y derivative windows to be calculated.

When calculating each derivative window, need to scan the new window unit which has a cells, from the analysis above, the time complexity is:

$$O(X \cdot Y \cdot a)$$

N-dimensional array is similar, the number of base windows is $\prod_{i=1}^{n-1} D_i$, moving on the last dimension and derivate D_n windows. For each derivate window, the new window unit to be scanned has $\prod_{i=1}^{n-1} w_i$ cells. So the time complexity for general cases will be

$$O(\prod_{i=1}^{n} D_i \cdot \prod_{i=1}^{n-1} w_i)$$

Compare to the naive method, the incremental computing improvement reduce a multiply item 'w_n' in the time complexity. This means that the improved algorithm is w_n times faster than the naive method.

From the expression above, we can also find a special feature: the time complexity of incremental computing for window sum/avg aggregates has no relationship with the window size on the last dimension - w_n. This means no matter how big the window is on the last dimension, it won't affect the efficiency of the incremental computing algorithm. I will confirm this feature in the test stage in next chapter.

4.2.2 Max/Min

Consider max/min functions of window aggregates, besides the analyzed part above, the part of heap operators also need to be considered. Here I analyze the time complexity in two aspects: one is the min/max calculation of window units, the other one is the operators in heap.

The calculation of window units is similar to the sum/avg function, which has been already analyzed before. The number of cells that need to be scanned is the same, only the computed value is max/min value instead of sum. So the time complexity of this part is still

$$O(\prod_{i=1}^{n} D_i \cdot \prod_{i=1}^{n-1} w_i)$$

Then consider the heap operation part. As a heap, to insert a new node will cost $O(\log(L))$, to delete the top node and maintain the rest nodes remaining the heap property will also cost $O(\log(L))$. Here L means the number of nodes inside the heap. The time complexity of heap operations above is result of average analysis, which is fundamental in data structure. So won't give detailed explanations here.

Back to the analysis of window query, there are $\prod_{i=1}^{n-1} D_i$ base windows, for each base window, at total D_n window units' min/max values need to be insert into the heap, among these values, at most $D_n - w_n$ ones will be deleted from the heap during the incremental computing process. So take heap's size 'L' as D_n, the number of insert and delete operation can also be approximate to D_n. So the time complexity of heap part will be

$$O(\prod_{i=1}^{n-1} D_i \cdot D_n \cdot (logD_n + logD_n) = O(\prod_{i=1}^{n} D_i \cdot logD_n)$$

In summary, for the max/min functions' window aggregation, its time complexity is

$$O(\prod_{i=1}^{n} D_i \cdot \prod_{i=1}^{n-1} w_i + \prod_{i=1}^{n} D_i \cdot logD_n)$$

$$= O(\prod_{i=1}^{n} D_i \cdot （\prod_{i=1}^{n-1} w_i + logD_n）)$$

Observe this expression, compare to the item $\prod_{i=1}^{n-1} w_i$, $\log D_n$ will be much slower in most cases, so the time complexity of max/min is only a little bigger than sum/avg. Of course, it is also about w_n times faster than naive method.

Also, we can see that min/max functions' incremental computation is like sum/avg functions, the time complexity is unrelated with aggregating window's size on last dimension - w_n. This is very interesting. It means that when process the window queries, no matter the aggregating window's size is 100*2, or 100*1000, the running time will be the same.

4.3 Further Optimization

From the previous analysis, it is clear that the improvement of incremental computing is mainly on the reducing of the multiply item w_n. w_n is the window size on the last dimension. When design the incremental computation algorithm, on purpose of simple implement, I choose the first n-1 dimension to determine the base windows and moving on the last dimension to get base windows' derivative windows. Apparently, this design can be optimized. Before process the window aggregate, we can figure

out in which dimension, the aggregating window's size is the biggest. Then treat this dimension instead of the last dimension as the window moving dimension and take the rest n-1 dimensions to generate the base windows. In this way, the reducing multiply item will be the biggest one among all the w_i. Then time complexity will be improved to

For Sum / Avg:

$$O(\frac{\prod_{i=1}^{n} D_i \cdot \prod_{i=1}^{n} w_i}{\max(w_i)})$$

Before optimization (form changed to compare):

$$O(\prod_{i=1}^{n} D_i \cdot \prod_{i=1}^{n-1} w_i) = O(\frac{\prod_{i=1}^{n} D_i \cdot \prod_{i=1}^{n} w_i}{w_n})$$

Similarly, for min/max functions, the optimized time complexity will be

$$O(\prod_{i=1}^{n} D_i \cdot (\frac{\prod_{i=1}^{n} w_i}{\max(w_i)} + logD_n))$$

Before optimization:

$$O(\prod_{i=1}^{n} D_i \cdot (\frac{\prod_{i=1}^{n} w_i}{w_n} + logD_n))$$

From the analysis above, we can see that this optimization will works very well when the window size is not so structured. It means the window get large size in some dimensions and get small size in other dimensions. But for the aggregate window that has almost same edge size in every dimension, this optimization will change nothing.

Here is a simple example on 3-dimensional array, run window aggregate query with window size 500*50*5. The origin incremental computing algorithm will reduce time complexity by the multiply item $w_3 = 5$. Meanwhile, as the algorithm using this optimization, the reducing multiply item will be $\max(w_i) = w_1 = 500$. It leads to 100 times faster than the origin algorithm at the process speed theoretically. But if the window size is 50*50*50, this optimization won't affect the speed at all.

5. Evaluation

This chapter will introduce the test results of the incremental computing algorithm for window aggregates. All the tests are run in my array processing system.

5.1 Performance

Run same test data of window aggregates over incremental computing algorithm and naive method. Compare the running time.

First test series is 2-dimensinal array of size 1000×1000, attributes values are random integers range from 0 to 1000. Increase the window size while testing, the result is showed below:

Table 1 Test result on 2-dimensional array 1000*1000

Win. size	5*5	10*10	20*20	30*30	40*40	50*50
Avg(IC*)	50ms	70ms	120ms	180ms	250ms	300ms
Avg(N*)	590ms	1.89s	6.95s	14.93s	26.0s	39.89s
Max(IC)	260ms	320ms	390ms	470ms	550ms	630ms
Max(N)	590ms	1.88s	6.98s	14.96s	25.93s	40.04s

Note: IC stands for incremental computation algorithm, while N stands for Naive method, similarly hereinafter

From the result, we can see that the incremental computing method is much faster than the naive one. As the window size gets bigger, the effect of improvement gets greater. Since bigger the aggregating window is, more redundant calculation the naive method will process.

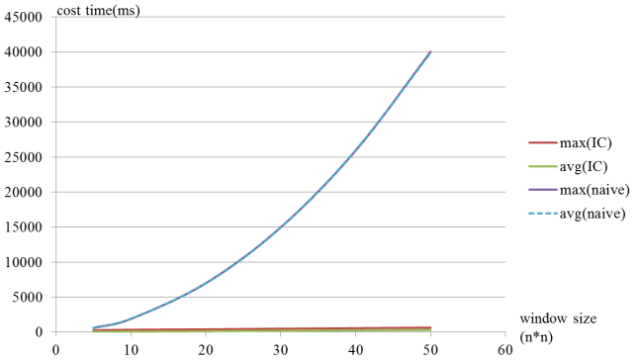

Figure 10 running time-window size Graph

In the graph above, because the disparity of running time between these two methods is really large when the window gets big, it's difficult to find a proper scale to display the graph well. Still, we can see that the incremental computing greatly improve the calculate efficiency. For 100*100 window query, the improved algorithm can output the result within one second. But for the same data, the naive method need more than 2 minutes.

Here is another set of test cases of 3-dimensional array.

Table 2 Test result on 3-dimensional array 80*80*80

Win. Size	5*5*5	10*10*10	15*15*15	20*20*20	25*25*25
Avg(IC)/ms	80	260	510	900	1330
Avg(N)/ms	1240	7680	23850	50540	88490
Max(IC)/ms	190	460	850	1400	2040
Max(N)/ms	1210	7640	23320	50610	88160

The test result also shows the superior performance of incremental computing algorithm. According to the time

complexity analysis in the last chapter, it should be w_n times faster than naive method. "w_n" represents the window size in the last dimension. Check the test result above and we can see that the real improvement of process speed almost fits in the analysis, even get better. It may be influenced by some constant running time items.

5.2 Feature test

The time complexity analysis of incremental computing method shows that its computation amount is not related with the window size on the last dimension - w_n. In other words, for the same array, no matter how w_n varies in window queries, the aggregate processing time will remain similar.In order to verify this feature as well as the analyzed time complexity's validity, I design a set of test cases specially.

First case is the same 1000*1000 2-dimension array used above. This time remain the first dimension's window size invariant as 40, and increase w_n, in this case w_2 to check how the running time behaves.

Table 3 Feature test for incremental computing (2-D)

Win. Size	40*10	40*20	40*30	40*40	40*50	40*60	40*70
Avg(IC) /ms	250	250	240	250	240	240	250
Max(IC) /ms	570	560	550	550	560	550	550

The result show clearly that no matter how w_2 varies, the processing time of window aggregate queries using incremental computing almost remain constant. Relatively, the naive method also behaves as the analyzed in last chapter, its consumed time has linear relation with the whole window size.

Here is another 3-dimension test data for the same test purpose.

Table 4 Feature test for incremental computing (3-D)

Win. Size	20*20* 10	20*20* 20	20*20* 30	20*20* 40	20*20* 50	20*20* 60
Sum(IC) /ms	910	900	910	930	940	940
Min(IC) /ms	1410	1400	1380	1360	1360	1350

Apparently, the test result matches the analysis well.

5.3 Test summary

This chapter mainly introduces the test status of the topic. The test work has two aspects.

Firstly, test to compare the performance between naive method and incremental computing. The result shows that the improvement is quite good. Secondly, I design special cases to verify the time complexity analysis as well as this feature of incremental computing algorithm.

I tried to test against the window aggregates in current array databases, for example SciDB. But since these array databases of course will store data in disks and the major time will be cost on accessing the data required by the window query instead of the query processing itself. As my own array data processing system using data directly in memory, there is no meaning and no way to compare. However, I need to state that this algorithm is a high level method and does not concern with the low level data store structures and mechanisms. It takes array data as input and output query result. So basically it can be implemented into any array databases as long as the base data structure is "array".

6. Summary and Future Work

This thesis proposes an improvement algorithm for window aggregates in array database. The improvement is based on incremental computation.

First introduce the background of array databases and window aggregates. Then mainly talk about the incremental computing method. Describe the algorithm's design idea and working process. The idea is to buffer intermediate aggregate values and reuse them while calculating the following windows. In this way, big amount of redundant calculation will be reduced and will save much time processing the window queries.Then analyze the time complexity of the algorithm, get quantitative analysis about the improving effect. It turns out the incremental computing method will save a multiple item w_n comparing to the naive method. It's because for each window, naive one need to scan all the window units, while IC method only need to scan one window unit. As a window has w_n window units at total, it is nature the time efficiency of IC method will be w_n times better.

The test was run in the array data processing system implemented by my own. The result verifies the time complexity, prove that the incremental computing method can improve the efficiency of window queries greatly.

There are some further works can be expanded of this topic. First, the algorithm can be implemented into the current Array Database, like SciDB and try parallel processing among the cluster. Besides, the concept of array stream can be introduced into the system. Since stream data naturally has dimension of time, this time window aggregates can also use similar incremental computing method to optimize. Still, as stream data has requirement on real-time process, the method need to be modified in many aspects.

Reference

1) Adam Seering, Philippe Cudre-Mauroux, Samuel Madden Samuel, and Michael Stonebraker. Efficient Versioning for Scientific Array Databases, ICDE, 2012.
2) Alex van Ballegooij, Roberto Cornacchia, Arjen P. de Vries, Martin Kersten. Distribution Rules for Array Database Queries. Database and Expert Systems Applications Lecture Notes in Computer Science Volume 3588, 2005, p 55-64
3) Emad Soroush, Magdalena Balazinska, and Daniel Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. SIGMOD conference, 2011
4) Emad Soroush and Magdalena Balazinska. Time Travel in a Scientific Array Database. International Conference on Data Engineering (ICDE), 2013.
5) M.Stonebraker, J. Becla, D. DeWitt etc. Requirements for Science Data Bases and SciDB. CIDR Conference, Asilomar, CA, USA, 2009
6) M. Kersten, Y. Zhang, M. Ivanova; SciQL, A query language for science applications. EDBT/ICDT 2011 Workshop on Array Databases, Pages 1-12
7) P. Cudre-Mauroux, H. Kimura, K.-T. Lim etc. A Demonstration of SciDB: A Science-Oriented DBMS. VLDB'09 Volume 2, Number 1, 1534-1537, Lyon, France, 2009
8) Paul G. Brown. Overview of SciDB, Large Scale Array Storage, Processing and Analysis. SIGMOD conference, 2010
9) Tingjian Ge , Zdonik, S. Handling Uncertain Data in Array Database Systems. ICDE 2008. IEEE 24th International Conference, 2008
10) SciDB Development team. SciDB User Guide version 12.10, 2012.
11) Ying Zhang, Martin Kersten, Milena Ivanova; SciQL: bridging the gap between science and relational DBMS. IDEAS 2011, p 124-133